Jinja with Threads

Andreas Lochbihler

March 17, 2025

Abstract. We extend the Jinja source code semantics by Klein and Nipkow with Java-style arrays and threads. Concurrency is captured in a generic framework semantics for adding concurrency through interleaving to a sequential semantics, which features dynamic thread creation, inter-thread communication via shared memory, lock synchronisation and joins. Also, threads can suspend themselves and be notified by others. We instantiate the framework with the adapted versions of both Jinja source and byte code and show type safety for the multithreaded case. Equally, the compiler from source to byte code is extended, for which we prove weak bisimilarity between the source code small step semantics and the defensive Jinja virtual machine. On top of this, we formalise the JMM and show the DRF guarantee and consistency.

For description of the different parts, see [1, 2, 4, 3].

Contents

Chapter 1

The generic multithreaded semantics

1.1 State of the multithreaded semantics

theory *FWState*
imports
../Basic/Auxiliary
begin

datatype *lock-action* =
Lock
| *Unlock*
| *UnlockFail*
| *ReleaseAcquire*

datatype ('*t*,'*x*,'*m*) *new-thread-action* =
NewThread '*t* '*x* '*m*
| *ThreadExists* '*t* *bool*

datatype '*t* *conditional-action* =
Join '*t*
| *Yield*

datatype ('*t*, '*w*) *wait-set-action* =
Suspend '*w*
| *Notify* '*w*
| *NotifyAll* '*w*
| *WakeUp* '*t*
| *Notified*
| *WokenUp*

datatype '*t* *interrupt-action*
= *IsInterrupted* '*t* *bool*
| *Interrupt* '*t*
| *ClearInterrupt* '*t*

type-synonym '*l* *lock-actions* = '*l* ⇒*f* *lock-action list*

translations
(*type*) *'l lock-actions* <= (*type*) *'l* ⇒*f lock-action list*

type-synonym
(*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* =
'l lock-actions × (*'t*,*'x*,*'m*) *new-thread-action list* ×
't conditional-action list × (*'t*, *'w*) *wait-set-action list* ×
't interrupt-action list × *'o list*

print-translation ‹
let
fun tr'
[*Const* (@{*type-syntax finfun*}, -) $ *l* $
(*Const* (@{*type-syntax list*}, -) $ *Const* (@{*type-syntax lock-action*}, -)),
Const (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax new-thread-action*}, -) $ *t1* $ *x* $ *m*))
$
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax conditional-action*}, -) $ *t2*)) $
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax wait-set-action*}, -) $ *t3* $ *w*)) $
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax interrupt-action*}, -) $ *t4*)) $
(*Const* (@{*type-syntax list*}, -) $ *o1*))))] =
if t1 = *t2 andalso t2* = *t3 andalso t3* = *t4 then Syntax.const* @{*type-syntax thread-action*} $ *l* $
t1 $ *x* $ *m* $ *w* $ *o1*
else raise Match;
in [(@{*type-syntax prod*}, *K tr'*)]
end
›
typ (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action*

definition *locks-a* :: (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* ⇒ *'l lock-actions* (‹⦇-⦈$_l$› [*0*] *1000*) **where**
locks-a ≡ *fst*

definition *thr-a* :: (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* ⇒ (*'t*,*'x*,*'m*) *new-thread-action list* (‹⦇-⦈$_t$› [*0*] *1000*)
where
thr-a ≡ *fst o snd*

definition *cond-a* :: (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* ⇒ *'t conditional-action list* (‹⦇-⦈$_c$› [*0*] *1000*)
where
cond-a = *fst o snd o snd*

definition *wset-a* :: (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* ⇒ (*'t*, *'w*) *wait-set-action list* (‹⦇-⦈$_w$› [*0*] *1000*)
where
wset-a = *fst o snd o snd o snd*

definition *interrupt-a* :: (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* ⇒ *'t interrupt-action list* (‹⦇-⦈$_i$› [*0*] *1000*)
where
interrupt-a = *fst o snd o snd o snd o snd*

definition *obs-a* :: (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* ⇒ *'o list* (‹⦇-⦈$_o$› [*0*] *1000*) **where**
obs-a ≡ *snd o snd o snd o snd o snd*

lemma *locks-a-conv* [*simp*]: *locks-a* (*ls*, *ntsjswss*) = *ls*
by(*simp add*:*locks-a-def*)

lemma *thr-a-conv* [*simp*]: *thr-a* (*ls*, *nts*, *jswss*) = *nts*
by(*simp add*: *thr-a-def*)

lemma *cond-a-conv* [*simp*]: *cond-a* (*ls*, *nts*, *js*, *wws*) = *js*
by(*simp add*: *cond-a-def*)

lemma *wset-a-conv* [*simp*]: *wset-a* (*ls*, *nts*, *js*, *wss*, *isobs*) = *wss*
by(*simp add*: *wset-a-def*)

lemma *interrupt-a-conv* [*simp*]: *interrupt-a* (*ls*, *nts*, *js*, *ws*, *is*, *obs*) = *is*
by(*simp add*: *interrupt-a-def*)

lemma *obs-a-conv* [*simp*]: *obs-a* (*ls*, *nts*, *js*, *wss*, *is*, *obs*) = *obs*
by(*simp add*: *obs-a-def*)

fun *ta-update-locks* :: ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* ⇒ *lock-action* ⇒ '*l* ⇒ ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action*
where
ta-update-locks (*ls*, *nts*, *js*, *wss*, *obs*) *lta l* = (*ls*(*l* $:= *ls* $ *l* @ [*lta*]), *nts*, *js*, *wss*, *obs*)

fun *ta-update-NewThread* :: ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* ⇒ ('*t*,'*x*,'*m*) *new-thread-action* ⇒ ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* **where**
ta-update-NewThread (*ls*, *nts*, *js*, *wss*, *is*, *obs*) *nt* = (*ls*, *nts* @ [*nt*], *js*, *wss*, *is*, *obs*)

fun *ta-update-Conditional* :: ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* ⇒ '*t conditional-action* ⇒ ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action*
where
ta-update-Conditional (*ls*, *nts*, *js*, *wss*, *is*, *obs*) *j* = (*ls*, *nts*, *js* @ [*j*], *wss*, *is*, *obs*)

fun *ta-update-wait-set* :: ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* ⇒ ('*t*, '*w*) *wait-set-action* ⇒ ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* **where**
ta-update-wait-set (*ls*, *nts*, *js*, *wss*, *is*, *obs*) *ws* = (*ls*, *nts*, *js*, *wss* @ [*ws*], *is*, *obs*)

fun *ta-update-interrupt* :: ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* ⇒ '*t interrupt-action* ⇒ ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action*
where
ta-update-interrupt (*ls*, *nts*, *js*, *wss*, *is*, *obs*) *i* = (*ls*, *nts*, *js*, *wss*, *is* @ [*i*], *obs*)

fun *ta-update-obs* :: ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* ⇒ '*o* ⇒ ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action*
where
ta-update-obs (*ls*, *nts*, *js*, *wss*, *is*, *obs*) *ob* = (*ls*, *nts*, *js*, *wss*, *is*, *obs* @ [*ob*])

abbreviation *empty-ta* :: ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* **where**
empty-ta ≡ (*K*$ [], [], [], [], [], [])

notation (*input*) *empty-ta* (‹ε›)

Pretty syntax for specifying thread actions: Write ⦃ *Lock→l*, *Unlock→l*, *Suspend w*, *Interrupt t*⦄ instead of ((*K*$ [])(*l* $:= [*Lock*, *Unlock*]), [], [*Suspend w*], [*Interrupt t*], []).

thread-action' is a type that contains of all basic thread actions. Automatically coerce basic thread actions into that type and then dispatch to the right update function by pattern matching. For coercion, adhoc overloading replaces the generic injection *inject-thread-action*

by the specific ones, i.e. constructors. To avoid ambiguities with observable actions, the observable actions must be of sort *obs-action*, which the basic thread action types are not.

class *obs-action*

datatype (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action'*
= *LockAction lock-action* × *'l*
| *NewThreadAction* (*'t*,*'x*,*'m*) *new-thread-action*
| *ConditionalAction 't conditional-action*
| *WaitSetAction* (*'t*, *'w*) *wait-set-action*
| *InterruptAction 't interrupt-action*
| *ObsAction 'o*

setup ‹
Sign.add-const-constraint (@{*const-name ObsAction*}, *SOME* @{*typ 'o* :: *obs-action* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action'*})
›

fun *thread-action'-to-thread-action* ::
(*'l*,*'t*,*'x*,*'m*,*'w*,*'o* :: *obs-action*) *thread-action'* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action*
where
thread-action'-to-thread-action (*LockAction* (*la*, *l*)) *ta* = *ta-update-locks ta la l*
| *thread-action'-to-thread-action* (*NewThreadAction nt*) *ta* = *ta-update-NewThread ta nt*
| *thread-action'-to-thread-action* (*ConditionalAction ca*) *ta* = *ta-update-Conditional ta ca*
| *thread-action'-to-thread-action* (*WaitSetAction wa*) *ta* = *ta-update-wait-set ta wa*
| *thread-action'-to-thread-action* (*InterruptAction ia*) *ta* = *ta-update-interrupt ta ia*
| *thread-action'-to-thread-action* (*ObsAction ob*) *ta* = *ta-update-obs ta ob*

consts *inject-thread-action* :: *'a* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action'*

nonterminal *ta-let* **and** *ta-lets*
syntax
-ta-snoc :: *ta-lets* ⇒ *ta-let* ⇒ *ta-lets* (‹-,/ -›)
-ta-block :: *ta-lets* ⇒ *'a* (‹⦃-⦄› [*0*] *1000*)
-ta-empty :: *ta-lets* (‹›)
-ta-single :: *ta-let* ⇒ *ta-lets* (‹-›)
-ta-inject :: *logic* ⇒ *ta-let* (‹(-)›)
-ta-lock :: *logic* ⇒ *logic* ⇒ *ta-let* (‹-⟶-›)

translations
-ta-block -ta-empty == *CONST empty-ta*
-ta-block (*-ta-single bta*) == *-ta-block* (*-ta-snoc -ta-empty bta*)
-ta-inject bta == *CONST inject-thread-action bta*
-ta-lock la l == *CONST inject-thread-action* (*CONST Pair la l*)
-ta-block (*-ta-snoc btas bta*) == *CONST thread-action'-to-thread-action bta* (*-ta-block btas*)

adhoc-overloading
inject-thread-action ⇌ *NewThreadAction ConditionalAction WaitSetAction InterruptAction ObsAction LockAction*

lemma *ta-upd-proj-simps* [*simp*]:
shows *ta-obs-proj-simps*:
⦃*ta-update-obs ta obs*⦄$_l$ = ⦃*ta*⦄$_l$ ⦃*ta-update-obs ta obs*⦄$_t$ = ⦃*ta*⦄$_t$ ⦃*ta-update-obs ta obs*⦄$_w$ = ⦃*ta*⦄$_w$

$\{\!|ta\text{-}update\text{-}obs\ ta\ obs|\!\}_c = \{\!|ta|\!\}_c$ $\{\!|ta\text{-}update\text{-}obs\ ta\ obs|\!\}_i = \{\!|ta|\!\}_i$ $\{\!|ta\text{-}update\text{-}obs\ ta\ obs|\!\}_o = \{\!|ta|\!\}_o$ @ $[obs]$

and *ta-lock-proj-simps*:

$\{\!|ta\text{-}update\text{-}locks\ ta\ x\ l|\!\}_l = (let\ ls = \{\!|ta|\!\}_l\ in\ ls(l\ \$:=\ ls\ \$\ l\ @\ [x]))$

$\{\!|ta\text{-}update\text{-}locks\ ta\ x\ l|\!\}_t = \{\!|ta|\!\}_t$ $\{\!|ta\text{-}update\text{-}locks\ ta\ x\ l|\!\}_w = \{\!|ta|\!\}_w$ $\{\!|ta\text{-}update\text{-}locks\ ta\ x\ l|\!\}_c = \{\!|ta|\!\}_c$

$\{\!|ta\text{-}update\text{-}locks\ ta\ x\ l|\!\}_i = \{\!|ta|\!\}_i$ $\{\!|ta\text{-}update\text{-}locks\ ta\ x\ l|\!\}_o = \{\!|ta|\!\}_o$

and *ta-thread-proj-simps*:

$\{\!|ta\text{-}update\text{-}NewThread\ ta\ t|\!\}_l = \{\!|ta|\!\}_l$ $\{\!|ta\text{-}update\text{-}NewThread\ ta\ t|\!\}_t = \{\!|ta|\!\}_t$ @ $[t]$ $\{\!|ta\text{-}update\text{-}NewThread\ ta\ t|\!\}_w = \{\!|ta|\!\}_w$

$\{\!|ta\text{-}update\text{-}NewThread\ ta\ t|\!\}_c = \{\!|ta|\!\}_c$ $\{\!|ta\text{-}update\text{-}NewThread\ ta\ t|\!\}_i = \{\!|ta|\!\}_i$ $\{\!|ta\text{-}update\text{-}NewThread\ ta\ t|\!\}_o = \{\!|ta|\!\}_o$

and *ta-wset-proj-simps*:

$\{\!|ta\text{-}update\text{-}wait\text{-}set\ ta\ w|\!\}_l = \{\!|ta|\!\}_l$ $\{\!|ta\text{-}update\text{-}wait\text{-}set\ ta\ w|\!\}_t = \{\!|ta|\!\}_t$ $\{\!|ta\text{-}update\text{-}wait\text{-}set\ ta\ w|\!\}_w = \{\!|ta|\!\}_w$ @ $[w]$

$\{\!|ta\text{-}update\text{-}wait\text{-}set\ ta\ w|\!\}_c = \{\!|ta|\!\}_c$ $\{\!|ta\text{-}update\text{-}wait\text{-}set\ ta\ w|\!\}_i = \{\!|ta|\!\}_i$ $\{\!|ta\text{-}update\text{-}wait\text{-}set\ ta\ w|\!\}_o = \{\!|ta|\!\}_o$

and *ta-cond-proj-simps*:

$\{\!|ta\text{-}update\text{-}Conditional\ ta\ c|\!\}_l = \{\!|ta|\!\}_l$ $\{\!|ta\text{-}update\text{-}Conditional\ ta\ c|\!\}_t = \{\!|ta|\!\}_t$ $\{\!|ta\text{-}update\text{-}Conditional\ ta\ c|\!\}_w = \{\!|ta|\!\}_w$

$\{\!|ta\text{-}update\text{-}Conditional\ ta\ c|\!\}_c = \{\!|ta|\!\}_c$ @ $[c]$ $\{\!|ta\text{-}update\text{-}Conditional\ ta\ c|\!\}_i = \{\!|ta|\!\}_i$ $\{\!|ta\text{-}update\text{-}Conditional\ ta\ c|\!\}_o = \{\!|ta|\!\}_o$

and *ta-interrupt-proj-simps*:

$\{\!|ta\text{-}update\text{-}interrupt\ ta\ i|\!\}_l = \{\!|ta|\!\}_l$ $\{\!|ta\text{-}update\text{-}interrupt\ ta\ i|\!\}_t = \{\!|ta|\!\}_t$ $\{\!|ta\text{-}update\text{-}interrupt\ ta\ i|\!\}_c = \{\!|ta|\!\}_c$

$\{\!|ta\text{-}update\text{-}interrupt\ ta\ i|\!\}_w = \{\!|ta|\!\}_w$ $\{\!|ta\text{-}update\text{-}interrupt\ ta\ i|\!\}_i = \{\!|ta|\!\}_i$ @ $[i]$ $\{\!|ta\text{-}update\text{-}interrupt\ ta\ i|\!\}_o = \{\!|ta|\!\}_o$

by(*cases ta*, *simp*)+

lemma *thread-action′-to-thread-action-proj-simps* [*simp*]:

shows *thread-action′-to-thread-action-proj-locks-simps*:

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (LockAction\ (la,\ l))\ ta|\!\}_l = \{\!|ta\text{-}update\text{-}locks\ ta\ la\ l|\!\}_l$

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (NewThreadAction\ nt)\ ta|\!\}_l = \{\!|ta\text{-}update\text{-}NewThread\ ta\ nt|\!\}_l$

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (ConditionalAction\ ca)\ ta|\!\}_l = \{\!|ta\text{-}update\text{-}Conditional\ ta\ ca|\!\}_l$

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (WaitSetAction\ wa)\ ta|\!\}_l = \{\!|ta\text{-}update\text{-}wait\text{-}set\ ta\ wa|\!\}_l$

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (InterruptAction\ ia)\ ta|\!\}_l = \{\!|ta\text{-}update\text{-}interrupt\ ta\ ia|\!\}_l$

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (ObsAction\ ob)\ ta|\!\}_l = \{\!|ta\text{-}update\text{-}obs\ ta\ ob|\!\}_l$

and *thread-action′-to-thread-action-proj-nt-simps*:

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (LockAction\ (la,\ l))\ ta|\!\}_t = \{\!|ta\text{-}update\text{-}locks\ ta\ la\ l|\!\}_t$

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (NewThreadAction\ nt)\ ta|\!\}_t = \{\!|ta\text{-}update\text{-}NewThread\ ta\ nt|\!\}_t$

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (ConditionalAction\ ca)\ ta|\!\}_t = \{\!|ta\text{-}update\text{-}Conditional\ ta\ ca|\!\}_t$

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (WaitSetAction\ wa)\ ta|\!\}_t = \{\!|ta\text{-}update\text{-}wait\text{-}set\ ta\ wa|\!\}_t$

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (InterruptAction\ ia)\ ta|\!\}_t = \{\!|ta\text{-}update\text{-}interrupt\ ta\ ia|\!\}_t$

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (ObsAction\ ob)\ ta|\!\}_t = \{\!|ta\text{-}update\text{-}obs\ ta\ ob|\!\}_t$

and *thread-action′-to-thread-action-proj-cond-simps*:

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (LockAction\ (la,\ l))\ ta|\!\}_c = \{\!|ta\text{-}update\text{-}locks\ ta\ la\ l|\!\}_c$

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (NewThreadAction\ nt)\ ta|\!\}_c = \{\!|ta\text{-}update\text{-}NewThread\ ta\ nt|\!\}_c$

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (ConditionalAction\ ca)\ ta|\!\}_c = \{\!|ta\text{-}update\text{-}Conditional\ ta\ ca|\!\}_c$

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (WaitSetAction\ wa)\ ta|\!\}_c = \{\!|ta\text{-}update\text{-}wait\text{-}set\ ta\ wa|\!\}_c$

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (InterruptAction\ ia)\ ta|\!\}_c = \{\!|ta\text{-}update\text{-}interrupt\ ta\ ia|\!\}_c$

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (ObsAction\ ob)\ ta|\!\}_c = \{\!|ta\text{-}update\text{-}obs\ ta\ ob|\!\}_c$

and *thread-action′-to-thread-action-proj-wset-simps*:

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (LockAction\ (la,\ l))\ ta|\!\}_w = \{\!|ta\text{-}update\text{-}locks\ ta\ la\ l|\!\}_w$

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (NewThreadAction\ nt)\ ta|\!\}_w = \{\!|ta\text{-}update\text{-}NewThread\ ta\ nt|\!\}_w$

$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (ConditionalAction\ ca)\ ta|\!\}_w = \{\!|ta\text{-}update\text{-}Conditional\ ta\ ca|\!\}_w$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (WaitSetAction\ wa)\ ta|\!\}_w = \{\!|ta\text{-}update\text{-}wait\text{-}set\ ta\ wa|\!\}_w$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (InterruptAction\ ia)\ ta|\!\}_w = \{\!|ta\text{-}update\text{-}interrupt\ ta\ ia|\!\}_w$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (ObsAction\ ob)\ ta|\!\}_w = \{\!|ta\text{-}update\text{-}obs\ ta\ ob|\!\}_w$
and *thread-action'-to-thread-action-proj-interrupt-simps*:
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (LockAction\ (la,\ l))\ ta|\!\}_i = \{\!|ta\text{-}update\text{-}locks\ ta\ la\ l|\!\}_i$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (NewThreadAction\ nt)\ ta|\!\}_i = \{\!|ta\text{-}update\text{-}NewThread\ ta\ nt|\!\}_i$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (ConditionalAction\ ca)\ ta|\!\}_i = \{\!|ta\text{-}update\text{-}Conditional\ ta\ ca|\!\}_i$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (WaitSetAction\ wa)\ ta|\!\}_i = \{\!|ta\text{-}update\text{-}wait\text{-}set\ ta\ wa|\!\}_i$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (InterruptAction\ ia)\ ta|\!\}_i = \{\!|ta\text{-}update\text{-}interrupt\ ta\ ia|\!\}_i$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (ObsAction\ ob)\ ta|\!\}_i = \{\!|ta\text{-}update\text{-}obs\ ta\ ob|\!\}_i$
and *thread-action'-to-thread-action-proj-obs-simps*:
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (LockAction\ (la,\ l))\ ta|\!\}_o = \{\!|ta\text{-}update\text{-}locks\ ta\ la\ l|\!\}_o$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (NewThreadAction\ nt)\ ta|\!\}_o = \{\!|ta\text{-}update\text{-}NewThread\ ta\ nt|\!\}_o$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (ConditionalAction\ ca)\ ta|\!\}_o = \{\!|ta\text{-}update\text{-}Conditional\ ta\ ca|\!\}_o$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (WaitSetAction\ wa)\ ta|\!\}_o = \{\!|ta\text{-}update\text{-}wait\text{-}set\ ta\ wa|\!\}_o$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (InterruptAction\ ia)\ ta|\!\}_o = \{\!|ta\text{-}update\text{-}interrupt\ ta\ ia|\!\}_o$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (ObsAction\ ob)\ ta|\!\}_o = \{\!|ta\text{-}update\text{-}obs\ ta\ ob|\!\}_o$
by(*simp-all*)

lemmas *ta-upd-simps* =
ta-update-locks.simps ta-update-NewThread.simps ta-update-Conditional.simps
ta-update-wait-set.simps ta-update-interrupt.simps ta-update-obs.simps
thread-action'-to-thread-action.simps

declare *ta-upd-simps* [*simp del*]

hide-const (**open**)
LockAction NewThreadAction ConditionalAction WaitSetAction InterruptAction ObsAction
thread-action'-to-thread-action
hide-type (**open**) *thread-action'*

datatype *wake-up-status* =
WSNotified
| *WSWokenUp*

datatype *'w wait-set-status* =
InWS 'w
| *PostWS wake-up-status*

type-synonym *'t lock* = (*'t* × *nat*) *option*
type-synonym (*'l*,*'t*) *locks* = *'l* ⇒*f* *'t lock*
type-synonym *'l released-locks* = *'l* ⇒*f nat*
type-synonym (*'l*,*'t*,*'x*) *thread-info* = *'t* ⇀ (*'x* × *'l released-locks*)
type-synonym (*'w*,*'t*) *wait-sets* = *'t* ⇀ *'w wait-set-status*
type-synonym *'t interrupts* = *'t set*
type-synonym (*'l*,*'t*,*'x*,*'m*,*'w*) *state* = (*'l*,*'t*) *locks* × ((*'l*,*'t*,*'x*) *thread-info* × *'m*) × (*'w*,*'t*) *wait-sets* × *'t interrupts*

translations
(*type*) (*'l*, *'t*) *locks* <= (*type*) *'l* ⇒*f* (*'t* × *nat*) *option*
(*type*) (*'l*, *'t*, *'x*) *thread-info* <= (*type*) *'t* ⇀ (*'x* × (*'l* ⇒*f nat*))

```
print-translation ‹
  let
    fun tr'
      [Const (@{type-syntax finfun}, -) $ l1 $
       (Const (@{type-syntax option}, -) $
         (Const (@{type-syntax prod}, -) $ t1 $ Const (@{type-syntax nat}, -))),
       Const (@{type-syntax prod}, -) $
         (Const (@{type-syntax prod}, -) $
           (Const (@{type-syntax fun}, -) $ t2 $
             (Const (@{type-syntax option}, -) $
               (Const (@{type-syntax prod}, -) $ x $
                 (Const (@{type-syntax finfun}, -) $ l2 $ Const (@{type-syntax nat}, -))))) $
           m) $
         (Const (@{type-syntax prod}, -) $
           (Const (@{type-syntax fun}, -) $ t3 $
             (Const (@{type-syntax option}, -) $ (Const (@{type-syntax wait-set-status}, -) $ w))) $
           (Const (@{type-syntax fun}, -) $ t4 $ (Const (@{type-syntax bool}, -))))] =
      if t1 = t2 andalso t1 = t3 andalso t1 = t4 andalso l1 = l2
      then Syntax.const @{type-syntax state} $ l1 $ t1 $ x $ m $ w
      else raise Match;
  in [(@{type-syntax prod}, K tr')]
  end
›
typ ('l,'t,'x,'m,'w) state


abbreviation no-wait-locks :: 'l ⇒f nat
where no-wait-locks ≡ (K$ 0)

lemma neq-no-wait-locks-conv:
  ⋀ln. ln ≠ no-wait-locks ⟷ (∃ l. ln $ l > 0)
by(auto simp add: expand-finfun-eq fun-eq-iff)

lemma neq-no-wait-locksE:
  fixes ln assumes ln ≠ no-wait-locks obtains l where ln $ l > 0
using assms
by(auto simp add: neq-no-wait-locks-conv)
```

Use type variables for components instead of ('l, 't, 'x, 'm, 'w) *state* in types for state projections to allow to reuse them for refined state implementations for code generation.

```
definition locks :: ('locks × ('thread-info × 'm) × 'wsets × 'interrupts) ⇒ 'locks where
  locks lstsmws ≡ fst lstsmws

definition thr :: ('locks × ('thread-info × 'm) × 'wsets × 'interrupts) ⇒ 'thread-info where
  thr lstsmws ≡ fst (fst (snd lstsmws))

definition shr :: ('locks × ('thread-info × 'm) × 'wsets × 'interrupts) ⇒ 'm where
  shr lstsmws ≡ snd (fst (snd lstsmws))

definition wset :: ('locks × ('thread-info × 'm) × 'wsets × 'interrupts) ⇒ 'wsets where
  wset lstsmws ≡ fst (snd (snd lstsmws))

definition interrupts :: ('locks × ('thread-info × 'm) × 'wsets × 'interrupts) ⇒ 'interrupts where
  interrupts lstsmws ≡ snd (snd (snd lstsmws))
```

lemma *locks-conv* [*simp*]: *locks* (*ls*, *tsmws*) = *ls*
by(*simp add*: *locks-def*)

lemma *thr-conv* [*simp*]: *thr* (*ls*, (*ts*, *m*), *ws*) = *ts*
by(*simp add*: *thr-def*)

lemma *shr-conv* [*simp*]: *shr* (*ls*, (*ts*, *m*), *ws*, *is*) = *m*
by(*simp add*: *shr-def*)

lemma *wset-conv* [*simp*]: *wset* (*ls*, (*ts*, *m*), *ws*, *is*) = *ws*
by(*simp add*: *wset-def*)

lemma *interrupts-conv* [*simp*]: *interrupts* (*ls*, (*ts*, *m*), *ws*, *is*) = *is*
by(*simp add*: *interrupts-def*)

primrec *convert-new-thread-action* :: ('*x* ⇒ '*x'*) ⇒ ('*t*,'*x*,'*m*) *new-thread-action* ⇒ ('*t*,'*x'*,'*m*) *new-thread-action*
where
  convert-new-thread-action f (*NewThread t x m*) = *NewThread t* (*f x*) *m*
| *convert-new-thread-action f* (*ThreadExists t b*) = *ThreadExists t b*

lemma *convert-new-thread-action-inv* [*simp*]:
  NewThread t x h = *convert-new-thread-action f nta* ⟷ (∃ *x'*. *nta* = *NewThread t x' h* ∧ *x* = *f x'*)
  ThreadExists t b = *convert-new-thread-action f nta* ⟷ *nta* = *ThreadExists t b*
  convert-new-thread-action f nta = *NewThread t x h* ⟷ (∃ *x'*. *nta* = *NewThread t x' h* ∧ *x* = *f x'*)
  convert-new-thread-action f nta = *ThreadExists t b* ⟷ *nta* = *ThreadExists t b*
by(*cases nta*, *auto*)+

lemma *convert-new-thread-action-eqI*:
  ⟦ ⋀*t x m*. *nta* = *NewThread t x m* ⟹ *nta'* = *NewThread t* (*f x*) *m*;
    ⋀*t b*. *nta* = *ThreadExists t b* ⟹ *nta'* = *ThreadExists t b* ⟧
  ⟹ *convert-new-thread-action f nta* = *nta'*
apply(*cases nta*)
apply *fastforce*+
done

lemma *convert-new-thread-action-compose* [*simp*]:
  convert-new-thread-action f (*convert-new-thread-action g ta*) = *convert-new-thread-action* (*f o g*) *ta*
apply(*cases ta*)
apply(*simp-all add*: *convert-new-thread-action-def*)
done

lemma *inj-convert-new-thread-action* [*simp*]:
  inj (*convert-new-thread-action f*) = *inj f*
apply(*rule iffI*)
 apply(*rule injI*)
 apply(*drule-tac x*=*NewThread undefined x undefined* **in** *injD*)
 apply *auto*[*2*]
apply(*rule injI*)
apply(*case-tac x*)
apply(*auto dest*: *injD*)
done

lemma *convert-new-thread-action-id*:

convert-new-thread-action id = (*id* :: ('*t*, '*x*, '*m*) *new-thread-action* ⇒ ('*t*, '*x*, '*m*) *new-thread-action*)
(**is** *?thesis1*)
convert-new-thread-action (λ*x*. *x*) = (*id* :: ('*t*, '*x*, '*m*) *new-thread-action* ⇒ ('*t*, '*x*, '*m*) *new-thread-action*)
(**is** *?thesis2*)
proof −
show *?thesis1* **by**(*rule ext*)(*case-tac x*, *simp-all*)
thus *?thesis2* **by**(*simp add*: *id-def*)
qed

definition *convert-extTA* :: ('*x* ⇒ '*x'*) ⇒ ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* ⇒ ('*l*,'*t*,'*x'*,'*m*,'*w*,'*o*) *thread-action*
where *convert-extTA f ta* = (⦃*ta*⦄$_l$, *map* (*convert-new-thread-action f*) ⦃*ta*⦄$_t$, *snd* (*snd ta*))

lemma *convert-extTA-simps* [*simp*]:
convert-extTA f ε = ε
⦃*convert-extTA f ta*⦄$_l$ = ⦃*ta*⦄$_l$
⦃*convert-extTA f ta*⦄$_t$ = *map* (*convert-new-thread-action f*) ⦃*ta*⦄$_t$
⦃*convert-extTA f ta*⦄$_c$ = ⦃*ta*⦄$_c$
⦃*convert-extTA f ta*⦄$_w$ = ⦃*ta*⦄$_w$
⦃*convert-extTA f ta*⦄$_i$ = ⦃*ta*⦄$_i$
convert-extTA f (*las*, *tas*, *was*, *cas*, *is*, *obs*) = (*las*, *map* (*convert-new-thread-action f*) *tas*, *was*, *cas*, *is*, *obs*)
apply(*simp-all add*: *convert-extTA-def*)
apply(*cases ta*, *simp*)+
done

lemma *convert-extTA-eq-conv*:
convert-extTA f ta = *ta'* ⟷
⦃*ta*⦄$_l$ = ⦃*ta'*⦄$_l$ ∧ ⦃*ta*⦄$_c$ = ⦃*ta'*⦄$_c$ ∧ ⦃*ta*⦄$_w$ = ⦃*ta'*⦄$_w$ ∧ ⦃*ta*⦄$_o$ = ⦃*ta'*⦄$_o$ ∧ ⦃*ta*⦄$_i$ = ⦃*ta'*⦄$_i$ ∧ *length* ⦃*ta*⦄$_t$ = *length* ⦃*ta'*⦄$_t$ ∧
(∀ *n* < *length* ⦃*ta*⦄$_t$. *convert-new-thread-action f* (⦃*ta*⦄$_t$! *n*) = ⦃*ta'*⦄$_t$! *n*)
apply(*cases ta*, *cases ta'*)
apply(*auto simp add*: *convert-extTA-def map-eq-all-nth-conv*)
done

lemma *convert-extTA-compose* [*simp*]:
convert-extTA f (*convert-extTA g ta*) = *convert-extTA* (*f o g*) *ta*
by(*simp add*: *convert-extTA-def*)

lemma *obs-a-convert-extTA* [*simp*]: *obs-a* (*convert-extTA f ta*) = *obs-a ta*
by(*cases ta*) *simp*

Actions for thread start/finish

datatype '*o action* =
NormalAction '*o*
| *InitialThreadAction*
| *ThreadFinishAction*

instance *action* :: (*type*) *obs-action*
proof qed

definition *convert-obs-initial* :: ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* ⇒ ('*l*,'*t*,'*x*,'*m*,'*w*,'*o action*) *thread-action*
where
convert-obs-initial ta = (⦃*ta*⦄$_l$, ⦃*ta*⦄$_t$, ⦃*ta*⦄$_c$, ⦃*ta*⦄$_w$, ⦃*ta*⦄$_i$, *map NormalAction* ⦃*ta*⦄$_o$)

lemma *inj-NormalAction* [*simp*]: *inj NormalAction*
by(*rule injI*) *auto*

lemma *convert-obs-initial-inject* [*simp*]:
convert-obs-initial ta = *convert-obs-initial ta′* ⟷ *ta* = *ta′*
by(*cases ta*)(*cases ta′*, *auto simp add*: *convert-obs-initial-def*)

lemma *convert-obs-initial-empty-TA* [*simp*]:
convert-obs-initial ε = ε
by(*simp add*: *convert-obs-initial-def*)

lemma *convert-obs-initial-eq-empty-TA* [*simp*]:
convert-obs-initial ta = ε ⟷ *ta* = ε
ε = *convert-obs-initial ta* ⟷ *ta* = ε
by(*case-tac* [!] *ta*)(*auto simp add*: *convert-obs-initial-def*)

lemma *convert-obs-initial-simps* [*simp*]:
{|*convert-obs-initial ta*|}$_o$ = *map NormalAction* {|*ta*|}$_o$
{|*convert-obs-initial ta*|}$_l$ = {|*ta*|}$_l$
{|*convert-obs-initial ta*|}$_t$ = {|*ta*|}$_t$
{|*convert-obs-initial ta*|}$_c$ = {|*ta*|}$_c$
{|*convert-obs-initial ta*|}$_w$ = {|*ta*|}$_w$
{|*convert-obs-initial ta*|}$_i$ = {|*ta*|}$_i$
by(*simp-all add*: *convert-obs-initial-def*)

type-synonym
('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *semantics* =
'*t* ⇒ '*x* × '*m* ⇒ ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* ⇒ '*x* × '*m* ⇒ *bool*

print-translation ‹
let
fun tr′
[*t4*,
Const (@{*type-syntax fun*}, -) $
(*Const* (@{*type-syntax prod*}, -) $ *x1* $ *m1*) $
(*Const* (@{*type-syntax fun*}, -) $
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax finfun*}, -) $ *l* $
(*Const* (@{*type-syntax list*}, -) $ *Const* (@{*type-syntax lock-action*}, -))) $
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax new-thread-action*}, -) $ *t1* $ *x2*
$ *m2*)) $
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax conditional-action*}, -) $ *t2*)) $
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax wait-set-action*}, -) $ *t3* $ *w*))
$
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax interrupt-action*}, -) $ *t5*))
$
(*Const* (@{*type-syntax list*}, -) $ *o1*)))))) $
(*Const* (@{*type-syntax fun*}, -) $ (*Const* (@{*type-syntax prod*}, -) $ *x3* $ *m3*) $
Const (@{*type-syntax bool*}, -)))] =

```
    if x1 = x2 andalso x1 = x3 andalso m1 = m2 andalso m1 = m3
      andalso t1 = t2 andalso t2 = t3 andalso t3 = t4 andalso t4 = t5
    then Syntax.const @{type-syntax semantics} $ l $ t1 $ x1 $ m1 $ w $ o1
    else raise Match;
  in [(@{type-syntax fun}, K tr')]
  end
›
typ ('l,'t,'x,'m,'w,'o) semantics

end
```

1.2 All about a managing a single lock

```
theory FWLock
imports
  FWState
begin

fun has-locks :: 't lock ⇒ 't ⇒ nat where
  has-locks None t = 0
| has-locks ⌊(t', n)⌋ t = (if t = t' then Suc n else 0)

lemma has-locks-iff:
  has-locks l t = n ⟷
  (l = None ∧ n = 0) ∨
  (∃ n'. l = ⌊(t, n')⌋ ∧ Suc n' = n) ∨ (∃ t' n'. l = ⌊(t', n')⌋ ∧ t' ≠ t ∧ n = 0)
by(cases l, auto)

lemma has-locksE:
  ⟦ has-locks l t = n;
    ⟦ l = None; n = 0 ⟧ ⟹ P;
    ⋀n'. ⟦ l = ⌊(t, n')⌋; Suc n' = n ⟧ ⟹ P;
    ⋀t' n'. ⟦ l = ⌊(t', n')⌋; t' ≠ t; n = 0 ⟧ ⟹ P ⟧
  ⟹ P
by(auto simp add: has-locks-iff split: if-split-asm prod.split-asm)


inductive may-lock :: 't lock ⇒ 't ⇒ bool where
  may-lock None t
| may-lock ⌊(t, n)⌋ t

lemma may-lock-iff [code]:
  may-lock l t = (case l of None ⇒ True | ⌊(t', n)⌋ ⇒ t = t')
by(auto intro: may-lock.intros elim: may-lock.cases)

lemma may-lockI:
  l = None ∨ (∃ n. l = ⌊(t, n)⌋) ⟹ may-lock l t
by(cases l, auto intro: may-lock.intros)

lemma may-lockE [consumes 1, case-names None Locked]:
  ⟦ may-lock l t; l = None ⟹ P; ⋀n. l = ⌊(t, n)⌋  ⟹ P ⟧ ⟹ P
by(auto elim: may-lock.cases)
```

lemma *may-lock-may-lock-t-eq*:
⟦ *may-lock l t*; *may-lock l t'* ⟧ ⟹ (*l* = *None*) ∨ (*t* = *t'*)
by(*auto elim*!: *may-lockE*)

abbreviation *has-lock* :: *'t lock* ⇒ *'t* ⇒ *bool*
where *has-lock l t* ≡ *0* < *has-locks l t*

lemma *has-locks-Suc-has-lock*:
has-locks l t = *Suc n* ⟹ *has-lock l t*
by(*auto*)

lemmas *has-lock-has-locks-Suc* = *gr0-implies-Suc*[**where** *n* = *has-locks l t*] **for** *l t*

lemma *has-lock-has-locks-conv*:
has-lock l t ⟷ (∃ *n*. *has-locks l t* = (*Suc n*))
by(*auto intro*: *has-locks-Suc-has-lock has-lock-has-locks-Suc*)

lemma *has-lock-may-lock*:
has-lock l t ⟹ *may-lock l t*
by(*cases l*, *auto intro*: *may-lockI*)

lemma *has-lock-may-lock-t-eq*:
⟦ *has-lock l t*; *may-lock l t'* ⟧ ⟹ *t* = *t'*
by(*auto elim*!: *may-lockE split*: *if-split-asm*)

lemma *has-locks-has-locks-t-eq*:
⟦*has-locks l t* = *Suc n*; *has-locks l t'* = *Suc n'*⟧ ⟹ *t* = *t'*
by(*auto elim*: *has-locksE*)

lemma *has-lock-has-lock-t-eq*:
⟦ *has-lock l t*; *has-lock l t'* ⟧ ⟹ *t* = *t'*
unfolding *has-lock-has-locks-conv*
by(*auto intro*: *has-locks-has-locks-t-eq*)

lemma *not-may-lock-conv*:
¬ *may-lock l t* ⟷ (∃ *t'*. *t'* ≠ *t* ∧ *has-lock l t'*)
by(*cases l*, *auto intro*: *may-lock.intros elim*: *may-lockE*)

fun *lock-lock* :: *'t lock* ⇒ *'t* ⇒ *'t lock* **where**
lock-lock None t = ⌊(*t*, *0*)⌋
| *lock-lock* ⌊(*t'*, *n*)⌋ *t* = ⌊(*t'*, *Suc n*)⌋

fun *unlock-lock* :: *'t lock* ⇒ *'t lock* **where**
unlock-lock None = *None*
| *unlock-lock* ⌊(*t*, *n*)⌋ = (*case n of 0* ⇒ *None* | *Suc n'* ⇒ ⌊(*t*, *n'*)⌋)

fun *release-all* :: *'t lock* ⇒ *'t* ⇒ *'t lock* **where**
release-all None t = *None*
| *release-all* ⌊(*t'*, *n*)⌋ *t* = (*if t* = *t' then None else* ⌊(*t'*, *n*)⌋)

fun *acquire-locks* :: *'t lock ⇒ 't ⇒ nat ⇒ 't lock* **where**
acquire-locks L t 0 = L
| *acquire-locks L t (Suc m) = acquire-locks (lock-lock L t) t m*

lemma *acquire-locks-conv*:
acquire-locks L t n = (case L of None ⇒ (case n of 0 ⇒ None | Suc m ⇒ ⌊(t, m)⌋) | ⌊(t', m)⌋ ⇒ ⌊(t', n + m)⌋)
by(*induct n arbitrary*: *L*)(*auto*)

lemma *lock-lock-ls-Some*:
∃ t' n. lock-lock l t = ⌊(t', n)⌋
by(*cases l*, *auto*)

lemma *unlock-lock-SomeD*:
unlock-lock l = ⌊(t', n)⌋ ⟹ l = ⌊(t', Suc n)⌋
by(*cases l*, *auto split*: *nat.split-asm*)

lemma *has-locks-Suc-lock-lock-has-locks-Suc-Suc*:
has-locks l t = Suc n ⟹ has-locks (lock-lock l t) t = Suc (Suc n)
by(*auto elim*!: *has-locksE*)

lemma *has-locks-lock-lock-conv* [*simp*]:
may-lock l t ⟹ has-locks (lock-lock l t) t = Suc (has-locks l t)
by(*auto elim*: *may-lockE*)

lemma *has-locks-release-all-conv* [*simp*]:
has-locks (release-all l t) t = 0
by(*cases l*, *auto split*: *if-split-asm*)

lemma *may-lock-lock-lock-conv* [*simp*]: *may-lock (lock-lock l t) t = may-lock l t*
by(*cases l*, *auto elim*!: *may-lock.cases intro*: *may-lock.intros*)

lemma *has-locks-acquire-locks-conv* [*simp*]:
may-lock l t ⟹ has-locks (acquire-locks l t n) t = has-locks l t + n
by(*induct n arbitrary*: *l*, *auto*)

lemma *may-lock-unlock-lock-conv* [*simp*]:
has-lock l t ⟹ may-lock (unlock-lock l) t = may-lock l t
by(*cases l*)(*auto split*: *if-split-asm nat.splits elim*!: *may-lock.cases intro*: *may-lock.intros*)

lemma *may-lock-release-all-conv* [*simp*]:
may-lock (release-all l t) t = may-lock l t
by(*cases l*, *auto split*: *if-split-asm intro*!: *may-lockI elim*: *may-lockE*)

lemma *may-lock-t-may-lock-unlock-lock-t*:
may-lock l t ⟹ may-lock (unlock-lock l) t
by(*auto intro*: *may-lock.intros elim*!: *may-lockE split*: *nat.split*)

lemma *may-lock-has-locks-lock-lock-0*:
⟦may-lock l t'; t ≠ t'⟧ ⟹ has-locks (lock-lock l t') t = 0
by(*auto elim*!: *may-lock.cases*)

lemma *has-locks-unlock-lock-conv* [*simp*]:

has-lock l t $\Longrightarrow$ *has-locks* (*unlock-lock l*) *t* = *has-locks l t* − *1*
by(*cases l*)(*auto split*: *nat.split*)

lemma *has-lock-lock-lock-unlock-lock-id* [*simp*]:
has-lock l t $\Longrightarrow$ *lock-lock* (*unlock-lock l*) *t* = *l*
by(*cases l*)(*auto split*: *if-split-asm nat.split*)

lemma *may-lock-unlock-lock-lock-lock-id* [*simp*]:
may-lock l t $\Longrightarrow$ *unlock-lock* (*lock-lock l t*) = *l*
by(*cases l*) *auto*

lemma *may-lock-has-locks-0*:
⟦ *may-lock l t*; *t* ≠ *t'* ⟧ $\Longrightarrow$ *has-locks l t'* = *0*
by(*auto elim*!: *may-lockE*)

fun *upd-lock* :: *'t lock* ⇒ *'t* ⇒ *lock-action* ⇒ *'t lock*
where
upd-lock l t Lock = *lock-lock l t*
| *upd-lock l t Unlock* = *unlock-lock l*
| *upd-lock l t UnlockFail* = *l*
| *upd-lock l t ReleaseAquire* = *release-all l t*

fun *upd-locks* :: *'t lock* ⇒ *'t* ⇒ *lock-action list* ⇒ *'t lock*
where
upd-locks l t [] = *l*
| *upd-locks l t* (*L* # *Ls*) = *upd-locks* (*upd-lock l t L*) *t Ls*

lemma *upd-locks-append* [*simp*]:
upd-locks l t (*Ls* @ *Ls'*) = *upd-locks* (*upd-locks l t Ls*) *t Ls'*
by(*induct Ls arbitrary*: *l*, *auto*)

lemma *upd-lock-Some-thread-idD*:
assumes *ul*: *upd-lock l t L* = ⌊(*t'*, *n*)⌋
and *tt'*: *t* ≠ *t'*
shows ∃ *n*. *l* = ⌊(*t'*, *n*)⌋
proof(*cases L*)
case *Lock*
with *ul tt'* **show** *?thesis*
by(*cases l*, *auto*)
next
case *Unlock*
with *ul tt'* **show** *?thesis*
by(*auto dest*: *unlock-lock-SomeD*)
next
case *UnlockFail*
with *ul* **show** *?thesis* **by**(*simp*)
next
case *ReleaseAcquire*
with *ul* **show** *?thesis*
by(*cases l*, *auto split*: *if-split-asm*)
qed

lemma *has-lock-upd-lock-implies-has-lock*:
⟦ *has-lock* (*upd-lock l t L*) *t'*; *t* ≠ *t'* ⟧ ⟹ *has-lock l t'*
by(*cases l L rule*: *option.exhaust*[*case-product lock-action.exhaust*])(*auto split*: *if-split-asm nat.split-asm*)

lemma *has-lock-upd-locks-implies-has-lock*:
⟦ *has-lock* (*upd-locks l t Ls*) *t'*; *t* ≠ *t'* ⟧ ⟹ *has-lock l t'*
by(*induct l t Ls rule*: *upd-locks.induct*)(*auto intro*: *has-lock-upd-lock-implies-has-lock*)

fun *lock-action-ok* :: *'t lock* ⇒ *'t* ⇒ *lock-action* ⇒ *bool* **where**
lock-action-ok l t Lock = *may-lock l t*
| *lock-action-ok l t Unlock* = *has-lock l t*
| *lock-action-ok l t UnlockFail* = (¬ *has-lock l t*)
| *lock-action-ok l t ReleaseAcquire* = *True*

fun *lock-actions-ok* :: *'t lock* ⇒ *'t* ⇒ *lock-action list* ⇒ *bool* **where**
lock-actions-ok l t [] = *True*
| *lock-actions-ok l t* (*L* # *Ls*) = (*lock-action-ok l t L* ∧ *lock-actions-ok* (*upd-lock l t L*) *t Ls*)

lemma *lock-actions-ok-append* [*simp*]:
lock-actions-ok l t (*Ls* @ *Ls'*) ⟷ *lock-actions-ok l t Ls* ∧ *lock-actions-ok* (*upd-locks l t Ls*) *t Ls'*
by(*induct Ls arbitrary*: *l*) *auto*

lemma *not-lock-action-okE* [*consumes 1*, *case-names Lock Unlock UnlockFail*]:
⟦ ¬ *lock-action-ok l t L*;
⟦ *L* = *Lock*; ¬ *may-lock l t* ⟧ ⟹ *Q*;
⟦ *L* = *Unlock*; ¬ *has-lock l t* ⟧ ⟹ *Q*;
⟦ *L* = *UnlockFail*; *has-lock l t* ⟧ ⟹ *Q*⟧
⟹ *Q*
by(*cases L*) *auto*

lemma *may-lock-upd-lock-conv* [*simp*]:
lock-action-ok l t L ⟹ *may-lock* (*upd-lock l t L*) *t* = *may-lock l t*
by(*cases L*) *auto*

lemma *may-lock-upd-locks-conv* [*simp*]:
lock-actions-ok l t Ls ⟹ *may-lock* (*upd-locks l t Ls*) *t* = *may-lock l t*
by(*induct l t Ls rule*: *upd-locks.induct*) *simp-all*

lemma *lock-actions-ok-Lock-may-lock*:
⟦ *lock-actions-ok l t Ls*; *Lock* ∈ *set Ls* ⟧ ⟹ *may-lock l t*
by(*induct l t Ls rule*: *lock-actions-ok.induct*) *auto*

lemma *has-locks-lock-lock-conv'* [*simp*]:
⟦ *may-lock l t'*; *t* ≠ *t'* ⟧ ⟹ *has-locks* (*lock-lock l t'*) *t* = *has-locks l t*
by(*cases l*)(*auto elim*: *may-lock.cases*)

lemma *has-locks-unlock-lock-conv'* [*simp*]:
⟦ *has-lock l t'*; *t* ≠ *t'* ⟧ ⟹ *has-locks* (*unlock-lock l*) *t* = *has-locks l t*
by(*cases l*)(*auto split*: *if-split-asm nat.split*)

lemma *has-locks-release-all-conv'* [*simp*]:

$t \neq t' \Longrightarrow$ *has-locks* (*release-all l t'*) *t* = *has-locks l t*
by(*cases l*) *auto*

lemma *has-locks-acquire-locks-conv'* [*simp*]:
⟦ *may-lock l t*; *t* ≠ *t'* ⟧ ⟹ *has-locks* (*acquire-locks l t n*) *t'* = *has-locks l t'*
by(*induct l t n rule*: *acquire-locks.induct*) *simp-all*

lemma *lock-action-ok-has-locks-upd-lock-eq-has-locks* [*simp*]:
⟦ *lock-action-ok l t' L*; *t* ≠ *t'* ⟧ ⟹ *has-locks* (*upd-lock l t' L*) *t* = *has-locks l t*
by(*cases L*) *auto*

lemma *lock-actions-ok-has-locks-upd-locks-eq-has-locks* [*simp*]:
⟦ *lock-actions-ok l t' Ls*; *t* ≠ *t'* ⟧ ⟹ *has-locks* (*upd-locks l t' Ls*) *t* = *has-locks l t*
by(*induct l t' Ls rule*: *upd-locks.induct*) *simp-all*

lemma *has-lock-acquire-locks-implies-has-lock*:
⟦ *has-lock* (*acquire-locks l t n*) *t'*; *t* ≠ *t'* ⟧ ⟹ *has-lock l t'*
unfolding *acquire-locks-conv*
by(*cases n*)(*auto split*: *if-split-asm*)

lemma *has-lock-has-lock-acquire-locks*:
has-lock l T ⟹ *has-lock* (*acquire-locks l t n*) *T*
unfolding *acquire-locks-conv*
by(*auto*)

fun *lock-actions-ok'* :: *'t lock* ⇒ *'t* ⇒ *lock-action list* ⇒ *bool* **where**
lock-actions-ok' l t [] = *True*
| *lock-actions-ok' l t* (*L*#*Ls*) = ((*L* = *Lock* ∧ ¬ *may-lock l t*) ∨
lock-action-ok l t L ∧ *lock-actions-ok'* (*upd-lock l t L*) *t Ls*)

lemma *lock-actions-ok'-iff*:
lock-actions-ok' l t las ⟷
lock-actions-ok l t las ∨ (∃ *xs ys*. *las* = *xs* @ *Lock* # *ys* ∧ *lock-actions-ok l t xs* ∧ ¬ *may-lock* (*upd-locks l t xs*) *t*)
proof(*induct l t las rule*: *lock-actions-ok.induct*)
case (*2 L t LA LAS*)
show *?case*
proof(*cases LA* = *Lock* ∧ ¬ *may-lock L t*)
case *True*
hence (∃ *ys*. *Lock* # *LAS* = [] @ *Lock* # *ys*) ∧ *lock-actions-ok L t* [] ∧ ¬ *may-lock* (*upd-locks L t* []) *t*
by(*simp*)
with *True* **show** *?thesis* **by**(*simp* (*no-asm*))(*blast*)
next
case *False*
with *2* **show** *?thesis*
by(*fastforce simp add*: *Cons-eq-append-conv elim*: *allE*[**where** *x*=*LA* # *xs* **for** *xs*])
qed
qed *simp*

lemma *lock-actions-ok'E*[*consumes 1*, *case-names ok Lock*]:
⟦ *lock-actions-ok' l t las*;
lock-actions-ok l t las ⟹ *P*;

$\bigwedge xs\ ys.\ [\![\ las = xs\ @\ Lock\ \#\ ys;\ lock\text{-}actions\text{-}ok\ l\ t\ xs;\ \neg\ may\text{-}lock\ (upd\text{-}locks\ l\ t\ xs)\ t\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
by(*auto simp add*: *lock-actions-ok'-iff*)

end

1.3 Semantics of the thread actions for locking

theory *FWLocking*
imports
FWLock
begin

definition *redT-updLs* :: ('*l*,'*t*) *locks* ⇒ '*t* ⇒ '*l lock-actions* ⇒ ('*l*,'*t*) *locks* **where**
redT-updLs ls t las ≡ (λ(*l*, *la*). *upd-locks l t la*) ∘$ (($*ls*, *las*$))

lemma *redT-updLs-iff* [*simp*]: *redT-updLs ls t las* $ *l* = *upd-locks* (*ls* $ *l*) *t* (*las* $ *l*)
by(*simp add*: *redT-updLs-def*)

lemma *upd-locks-empty-conv* [*simp*]: (λ(*l*, *las*). *upd-locks l t las*) ∘$ ($*ls*, *K*$ []$) = *ls*
by(*auto intro*: *finfun-ext*)

lemma *redT-updLs-Some-thread-idD*:
[*has-lock* (*redT-updLs ls t las* $ *l*) *t*'; *t* ≠ *t*'] ⟹ *has-lock* (*ls* $ *l*) *t*'
by(*auto simp add*: *redT-updLs-def intro*: *has-lock-upd-locks-implies-has-lock*)

definition *acquire-all* :: ('*l*, '*t*) *locks* ⇒ '*t* ⇒ ('*l* ⇒*f* *nat*) ⇒ ('*l*, '*t*) *locks*
where ⋀*ln*. *acquire-all ls t ln* ≡ (λ(*l*, *la*). *acquire-locks l t la*) ∘$ (($*ls*, *ln*$))

lemma *acquire-all-iff* [*simp*]:
⋀*ln*. *acquire-all ls t ln* $ *l* = *acquire-locks* (*ls* $ *l*) *t* (*ln* $ *l*)
by(*simp add*: *acquire-all-def*)

definition *lock-ok-las* :: ('*l*,'*t*) *locks* ⇒ '*t* ⇒ '*l lock-actions* ⇒ *bool* **where**
lock-ok-las ls t las ≡ ∀ *l*. *lock-actions-ok* (*ls* $ *l*) *t* (*las* $ *l*)

lemma *lock-ok-lasI* [*intro*]:
(⋀*l*. *lock-actions-ok* (*ls* $ *l*) *t* (*las* $ *l*)) ⟹ *lock-ok-las ls t las*
by(*simp add*: *lock-ok-las-def*)

lemma *lock-ok-lasE*:
[*lock-ok-las ls t las*; (⋀*l*. *lock-actions-ok* (*ls* $ *l*) *t* (*las* $ *l*)) ⟹ *Q*] ⟹ *Q*
by(*simp add*: *lock-ok-las-def*)

lemma *lock-ok-lasD*:
lock-ok-las ls t las ⟹ *lock-actions-ok* (*ls* $ *l*) *t* (*las* $ *l*)
by(*simp add*: *lock-ok-las-def*)

lemma *lock-ok-las-code* [*code*]:
lock-ok-las ls t las = *finfun-All* ((λ(*l*, *la*). *lock-actions-ok l t la*) ∘$ ($*ls*, *las*$))
by(*simp add*: *lock-ok-las-def finfun-All-All o-def*)

lemma *lock-ok-las-may-lock*:
⟦ *lock-ok-las ls t las*; *Lock* ∈ *set* (*las* $ *l*) ⟧ ⟹ *may-lock* (*ls* $ *l*) *t*
by(*erule lock-ok-lasE*)(*rule lock-actions-ok-Lock-may-lock*)

lemma *redT-updLs-may-lock* [*simp*]:
lock-ok-las ls t las ⟹ *may-lock* (*redT-updLs ls t las* $ *l*) *t* = *may-lock* (*ls* $ *l*) *t*
by(*auto dest*!: *lock-ok-lasD*[**where** *l*=*l*])

lemma *redT-updLs-has-locks* [*simp*]:
⟦ *lock-ok-las ls t*′ *las*; *t* ≠ *t*′ ⟧ ⟹ *has-locks* (*redT-updLs ls t*′ *las* $ *l*) *t* = *has-locks* (*ls* $ *l*) *t*
by(*auto dest*!: *lock-ok-lasD*[**where** *l*=*l*])

definition *may-acquire-all* :: (′*l*, ′*t*) *locks* ⇒ ′*t* ⇒ (′*l* ⇒*f* *nat*) ⇒ *bool*
where ⋀*ln*. *may-acquire-all ls t ln* ≡ ∀ *l*. *ln* $ *l* > 0 ⟶ *may-lock* (*ls* $ *l*) *t*

lemma *may-acquire-allI* [*intro*]:
⋀*ln*. (⋀*l*. *ln* $ *l* > 0 ⟹ *may-lock* (*ls* $ *l*) *t*) ⟹ *may-acquire-all ls t ln*
by(*simp add*: *may-acquire-all-def*)

lemma *may-acquire-allE*:
⋀*ln*. ⟦ *may-acquire-all ls t ln*; ∀ *l*. *ln* $ *l* > 0 ⟶ *may-lock* (*ls* $ *l*) *t* ⟹ *P* ⟧ ⟹ *P*
by(*auto simp add*: *may-acquire-all-def*)

lemma *may-acquire-allD* [*dest*]:
⋀*ln*. ⟦ *may-acquire-all ls t ln*; *ln* $ *l* > 0 ⟧ ⟹ *may-lock* (*ls* $ *l*) *t*
by(*auto simp add*: *may-acquire-all-def*)

lemma *may-acquire-all-has-locks-acquire-locks* [*simp*]:
fixes *ln*
shows ⟦ *may-acquire-all ls t ln*; *t* ≠ *t*′ ⟧ ⟹ *has-locks* (*acquire-locks* (*ls* $ *l*) *t* (*ln* $ *l*)) *t*′ = *has-locks* (*ls* $ *l*) *t*′
by(*cases ln* $ *l* > 0)(*auto dest*: *may-acquire-allD*)

lemma *may-acquire-all-code* [*code*]:
⋀*ln*. *may-acquire-all ls t ln* ⟷ *finfun-All* ((λ(*lock*, *n*). *n* > 0 ⟶ *may-lock lock t*) ∘$ ($*ls*, *ln*$))
by(*auto simp add*: *may-acquire-all-def finfun-All-All o-def*)

definition *collect-locks* :: ′*l lock-actions* ⇒ ′*l set* **where**
collect-locks las = {*l*. *Lock* ∈ *set* (*las* $ *l*)}

lemma *collect-locksI*:
Lock ∈ *set* (*las* $ *l*) ⟹ *l* ∈ *collect-locks las*
by(*simp add*: *collect-locks-def*)

lemma *collect-locksE*:
⟦ *l* ∈ *collect-locks las*; *Lock* ∈ *set* (*las* $ *l*) ⟹ *P* ⟧ ⟹ *P*
by(*simp add*: *collect-locks-def*)

lemma *collect-locksD*:
l ∈ *collect-locks las* ⟹ *Lock* ∈ *set* (*las* $ *l*)
by(*simp add*: *collect-locks-def*)

fun *must-acquire-lock* :: *lock-action list* ⇒ *bool* **where**
must-acquire-lock [] = *False*
| *must-acquire-lock* (*Lock* # *las*) = *True*
| *must-acquire-lock* (*Unlock* # *las*) = *False*
| *must-acquire-lock* (- # *las*) = *must-acquire-lock las*

lemma *must-acquire-lock-append*:
must-acquire-lock (*xs* @ *ys*) ⟷ (*if Lock* ∈ *set xs* ∨ *Unlock* ∈ *set xs then must-acquire-lock xs else must-acquire-lock ys*)
proof(*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons L Ls*)
thus *?case* **by** (*cases L*, *simp-all*)
qed

lemma *must-acquire-lock-contains-lock*:
must-acquire-lock las ⟹ *Lock* ∈ *set las*
proof(*induct las*)
case (*Cons l las*) **thus** *?case* **by**(*cases l*) *auto*
qed *simp*

lemma *must-acquire-lock-conv*:
must-acquire-lock las = (*case* (*filter* (λ*L*. *L* = *Lock* ∨ *L* = *Unlock*) *las*) *of* [] ⇒ *False* | *L* # *Ls* ⇒ *L* = *Lock*)
proof(*induct las*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons LA LAS*) **thus** *?case*
by(*cases LA*, *auto split*: *list.split-asm*)
qed

definition *collect-locks'* :: *'l lock-actions* ⇒ *'l set* **where**
collect-locks' las ≡ {*l*. *must-acquire-lock* (*las* $ *l*)}

lemma *collect-locks'I*:
must-acquire-lock (*las* $ *l*) ⟹ *l* ∈ *collect-locks' las*
by(*simp add*: *collect-locks'-def*)

lemma *collect-locks'E*:
⟦ *l* ∈ *collect-locks' las*; *must-acquire-lock* (*las* $ *l*) ⟹ *P* ⟧ ⟹ *P*
by(*simp add*: *collect-locks'-def*)

lemma *collect-locks'-subset-collect-locks*:
collect-locks' las ⊆ *collect-locks las*
by(*auto simp add*: *collect-locks'-def collect-locks-def intro*: *must-acquire-lock-contains-lock*)

definition *lock-ok-las'* :: ('*l*,'*t*) *locks* ⇒ *'t* ⇒ *'l lock-actions* ⇒ *bool* **where**
lock-ok-las' ls t las ≡ ∀ *l*. *lock-actions-ok'* (*ls* $ *l*) *t* (*las* $ *l*)

lemma *lock-ok-las'I*: (⋀*l*. *lock-actions-ok'* (*ls* $ *l*) *t* (*las* $ *l*)) ⟹ *lock-ok-las' ls t las*
by(*simp add*: *lock-ok-las'-def*)

lemma *lock-ok-las'D*: *lock-ok-las' ls t las* $\Longrightarrow$ *lock-actions-ok' (ls $ l) t (las $ l)*
by(*simp add*: *lock-ok-las'-def*)

lemma *not-lock-ok-las'-conv*:
$\neg$ *lock-ok-las' ls t las* $\longleftrightarrow$ ($\exists$ *l*. $\neg$ *lock-actions-ok' (ls $ l) t (las $ l)*)
by(*simp add*: *lock-ok-las'-def*)

lemma *lock-ok-las'-code*:
lock-ok-las' ls t las = finfun-All ((*λ*(*l*, *la*). *lock-actions-ok' l t la*) ∘$ ($*ls*, *las*$))
by(*simp add*: *lock-ok-las'-def finfun-All-All o-def*)

lemma *lock-ok-las'-collect-locks'-may-lock*:
assumes *lot'*: *lock-ok-las' ls t las*
and *mayl*: ∀ *l* ∈ *collect-locks' las*. *may-lock (ls $ l) t*
and *l*: *l* ∈ *collect-locks las*
shows *may-lock (ls $ l) t*
proof(*cases l* ∈ *collect-locks' las*)
case *True* **thus** *?thesis* **using** *mayl* **by** *auto*
next
case *False*
hence *nmal*: $\neg$ *must-acquire-lock (las $ l)*
by(*auto intro*: *collect-locks'I*)
from *l* **have** *locklasl*: *Lock* ∈ *set (las $ l)*
by(*rule collect-locksD*)
then obtain *ys zs*
where *las*: *las $ l = ys @ Lock # zs*
and *notin*: *Lock* ∉ *set ys*
by(*auto dest*: *split-list-first*)
from *lot'* **have** *lock-actions-ok' (ls $ l) t (las $ l)*
by(*auto simp add*: *lock-ok-las'-def*)
thus *?thesis*
proof(*induct rule*: *lock-actions-ok'E*)
case *ok*
with *locklasl* **show** *?thesis*
by −(*rule lock-actions-ok-Lock-may-lock*)
next
case (*Lock YS ZS*)
note *LAS* = ‹*las $ l = YS @ Lock # ZS*›
note *lao* = ‹*lock-actions-ok (ls $ l) t YS*›
note *nml* = ‹$\neg$ *may-lock (upd-locks (ls $ l) t YS) t*›
from *LAS las nmal notin* **have** *Unlock* ∈ *set YS*
by −(*erule contrapos-np*, *auto simp add*: *must-acquire-lock-append append-eq-append-conv2 append-eq-Cons-conv*)
then obtain *ys' zs'*
where *YS*: *YS = ys' @ Unlock # zs'*
and *unlock*: *Unlock* ∉ *set ys'*
by(*auto dest*: *split-list-first*)
from *YS las LAS lao* **have** *lao'*: *lock-actions-ok (ls $ l) t (ys' @ [Unlock])* **by**(*auto*)
hence *has-lock (upd-locks (ls $ l) t ys') t* **by** *simp*
hence *may-lock (upd-locks (ls $ l) t ys') t*
by(*rule has-lock-may-lock*)
moreover from *lao'* **have** *lock-actions-ok (ls $ l) t ys'* **by** *simp*
ultimately show *?thesis* **by** *simp*

qed
qed

lemma *lock-actions-ok'-must-acquire-lock-lock-actions-ok*:
⟦ *lock-actions-ok' l t Ls*; *must-acquire-lock Ls* ⟶ *may-lock l t*⟧ ⟹ *lock-actions-ok l t Ls*
proof(*induct l t Ls rule*: *lock-actions-ok.induct*)
case *1* **thus** *?case* **by** *simp*
next
case (*2 l t L LS*) **thus** *?case*
proof(*cases L* = *Lock* ∨ *L* = *Unlock*)
case *True*
with *2* **show** *?thesis* **by**(*auto simp add*: *lock-actions-ok'-iff Cons-eq-append-conv intro*: *has-lock-may-lock*)
qed(*cases L*, *auto*)
qed

lemma *lock-ok-las'-collect-locks-lock-ok-las*:
assumes *lol'*: *lock-ok-las' ls t las*
and *clml*: ⋀*l*. *l* ∈ *collect-locks las* ⟹ *may-lock* (*ls* $ *l*) *t*
shows *lock-ok-las ls t las*
proof(*rule lock-ok-lasI*)
fix *l*
from *lol'* **have** *lock-actions-ok'* (*ls* $ *l*) *t* (*las* $ *l*) **by**(*rule lock-ok-las'D*)
thus *lock-actions-ok* (*ls* $ *l*) *t* (*las* $ *l*)
proof(*rule lock-actions-ok'-must-acquire-lock-lock-actions-ok*[*OF* - *impI*])
assume *mal*: *must-acquire-lock* (*las* $ *l*)
thus *may-lock* (*ls* $ *l*) *t*
by(*auto intro*!: *clml collect-locksI elim*: *must-acquire-lock-contains-lock*)
qed
qed

lemma *lock-ok-las'-into-lock-on-las*:
⟦*lock-ok-las' ls t las*; ⋀*l*. *l* ∈ *collect-locks' las* ⟹ *may-lock* (*ls* $ *l*) *t*⟧ ⟹ *lock-ok-las ls t las*
by (*metis lock-ok-las'-collect-locks'-may-lock lock-ok-las'-collect-locks-lock-ok-las*)

end

1.4 Semantics of the thread actions for thread creation

theory *FWThread*
imports
FWState
begin

Abstractions for thread ids

context
notes [[*inductive-internals*]]
begin

inductive *free-thread-id* :: ('*l*,'*t*,'*x*) *thread-info* ⇒ '*t* ⇒ *bool*
for *ts* :: ('*l*,'*t*,'*x*) *thread-info* **and** *t* :: '*t*
where *ts t* = *None* ⟹ *free-thread-id ts t*

declare *free-thread-id.cases* [*elim*]

end

lemma *free-thread-id-iff*: *free-thread-id ts t* = (*ts t* = *None*)
by(*auto elim*: *free-thread-id.cases intro*: *free-thread-id.intros*)

Update functions for the multithreaded state

fun *redT-updT* :: ('*l*,'*t*,'*x*) *thread-info* ⇒ ('*t*,'*x*,'*m*) *new-thread-action* ⇒ ('*l*,'*t*,'*x*) *thread-info*
where
redT-updT ts (*NewThread t' x m*) = *ts*(*t'* ↦ (*x*, *no-wait-locks*))
| *redT-updT ts* - = *ts*

fun *redT-updTs* :: ('*l*,'*t*,'*x*) *thread-info* ⇒ ('*t*,'*x*,'*m*) *new-thread-action list* ⇒ ('*l*,'*t*,'*x*) *thread-info*
where
redT-updTs ts [] = *ts*
| *redT-updTs ts* (*ta*#*tas*) = *redT-updTs* (*redT-updT ts ta*) *tas*

lemma *redT-updTs-append* [*simp*]:
redT-updTs ts (*tas* @ *tas'*) = *redT-updTs* (*redT-updTs ts tas*) *tas'*
by(*induct ts tas rule*: *redT-updTs.induct*) *auto*

lemma *redT-updT-None*:
redT-updT ts ta t = *None* ⟹ *ts t* = *None*
by(*cases ta*)(*auto split*: *if-splits*)

lemma *redT-updTs-None*: *redT-updTs ts tas t* = *None* ⟹ *ts t* = *None*
by(*induct ts tas rule*: *redT-updTs.induct*)(*auto intro*: *redT-updT-None*)

lemma *redT-updT-Some1*:
ts t = ⌊*xw*⌋ ⟹ ∃ *xw*. *redT-updT ts ta t* = ⌊*xw*⌋
by(*cases ta*) *auto*

lemma *redT-updTs-Some1*:
ts t = ⌊*xw*⌋ ⟹ ∃ *xw*. *redT-updTs ts tas t* = ⌊*xw*⌋
unfolding *not-None-eq*[*symmetric*]
by(*induct ts tas arbitrary*: *xw rule*: *redT-updTs.induct*)(*simp-all del*: *split-paired-Ex*, *blast dest*: *redT-updT-Some1*)

lemma *redT-updT-finite-dom-inv*:
finite (*dom* (*redT-updT ts ta*)) = *finite* (*dom ts*)
by(*cases ta*) *auto*

lemma *redT-updTs-finite-dom-inv*:
finite (*dom* (*redT-updTs ts tas*)) = *finite* (*dom ts*)
by(*induct ts tas rule*: *redT-updTs.induct*)(*simp-all add*: *redT-updT-finite-dom-inv*)

Preconditions for thread creation actions

These primed versions are for checking preconditions only. They allow the thread actions to have a type for thread-local information that is different than the thread info state itself.

fun *redT-updT'* :: ('*l*,'*t*,'*x*) *thread-info* ⇒ ('*t*,'*x'*,'*m*) *new-thread-action* ⇒ ('*l*,'*t*,'*x*) *thread-info*
where
redT-updT' ts (*NewThread t' x m*) = *ts*(*t'* ↦ (*undefined*, *no-wait-locks*))
| *redT-updT' ts* - = *ts*

fun *redT-updTs'* :: ('*l*,'*t*,'*x*) *thread-info* ⇒ ('*t*,'*x'*,'*m*) *new-thread-action list* ⇒ ('*l*,'*t*,'*x*) *thread-info*

where
  redT-updTs' *ts* [] = *ts*
| *redT-updTs'* *ts* (*ta*#*tas*) = *redT-updTs'* (*redT-updT'* *ts* *ta*) *tas*

lemma *redT-updT'-None*:
  redT-updT' *ts* *ta* *t* = *None* ⟹ *ts* *t* = *None*
by(*cases ta*)(*auto split*: *if-splits*)

primrec *thread-ok* :: ('*l*,'*t*,'*x*) *thread-info* ⇒ ('*t*,'*x*','*m*) *new-thread-action* ⇒ *bool*
where
  thread-ok *ts* (*NewThread* *t* *x* *m*) = *free-thread-id* *ts* *t*
| *thread-ok* *ts* (*ThreadExists* *t* *b*) = (*b* ≠ *free-thread-id* *ts* *t*)

fun *thread-oks* :: ('*l*,'*t*,'*x*) *thread-info* ⇒ ('*t*,'*x*','*m*) *new-thread-action list* ⇒ *bool*
where
  thread-oks *ts* [] = *True*
| *thread-oks* *ts* (*ta*#*tas*) = (*thread-ok* *ts* *ta* ∧ *thread-oks* (*redT-updT'* *ts* *ta*) *tas*)

lemma *thread-ok-ts-change*:
  (⋀*t*. *ts* *t* = *None* ⟷ *ts'* *t* = *None*) ⟹ *thread-ok* *ts* *ta* ⟷ *thread-ok* *ts'* *ta*
by(*cases ta*)(*auto simp add*: *free-thread-id-iff*)

lemma *thread-oks-ts-change*:
  (⋀*t*. *ts* *t* = *None* ⟷ *ts'* *t* = *None*) ⟹ *thread-oks* *ts* *tas* ⟷ *thread-oks* *ts'* *tas*
proof(*induct tas arbitrary*: *ts ts'*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons ta tas ts ts'*)
  note *IH* = ‹⋀*ts ts'*. (⋀*t*. (*ts t* = *None*) = (*ts'* *t* = *None*)) ⟹ *thread-oks ts tas* = *thread-oks ts' tas*›
  note *eq* = ‹⋀*t*. (*ts t* = *None*) = (*ts'* *t* = *None*)›
  from *eq* **have** *thread-ok ts ta* ⟷ *thread-ok ts' ta* **by**(*rule thread-ok-ts-change*)
  moreover from *eq* **have** ⋀*t*. (*redT-updT'* *ts* *ta* *t* = *None*) = (*redT-updT'* *ts'* *ta* *t* = *None*)
    by(*cases ta*)(*auto*)
  hence *thread-oks* (*redT-updT'* *ts* *ta*) *tas* = *thread-oks* (*redT-updT'* *ts'* *ta*) *tas* **by**(*rule IH*)
  ultimately show *?case* **by** *simp*
qed

lemma *redT-updT'-eq-None-conv*:
  (⋀*t*. *ts* *t* = *None* ⟷ *ts'* *t* = *None*) ⟹ *redT-updT'* *ts* *ta* *t* = *None* ⟷ *redT-updT* *ts'* *ta* *t* = *None*
by(*cases ta*) *simp-all*

lemma *redT-updTs'-eq-None-conv*:
  (⋀*t*. *ts* *t* = *None* ⟷ *ts'* *t* = *None*) ⟹ *redT-updTs'* *ts* *tas* *t* = *None* ⟷ *redT-updTs* *ts'* *tas* *t* = *None*
apply(*induct tas arbitrary*: *ts ts'*)
apply *simp-all*
apply(*blast intro*: *redT-updT'-eq-None-conv del*: *iffI*)
done

lemma *thread-oks-redT-updT-conv* [*simp*]:
  thread-oks (*redT-updT'* *ts* *ta*) *tas* = *thread-oks* (*redT-updT* *ts* *ta*) *tas*
by(*rule thread-oks-ts-change*)(*rule redT-updT'-eq-None-conv refl*)+

lemma *thread-oks-append* [*simp*]:

thread-oks ts (tas @ tas′) = (thread-oks ts tas ∧ thread-oks (redT-updTs′ ts tas) tas′)
by(*induct tas arbitrary*: *ts*, *auto*)

lemma *thread-oks-redT-updTs-conv* [*simp*]:
thread-oks (redT-updTs′ ts ta) tas = thread-oks (redT-updTs ts ta) tas
by(*rule thread-oks-ts-change*)(*rule redT-updTs′-eq-None-conv refl*)+

lemma *redT-updT-Some*:
⟦ *ts t = ⌊xw⌋*; *thread-ok ts ta* ⟧ ⟹ *redT-updT ts ta t = ⌊xw⌋*
by(*cases ta*) *auto*

lemma *redT-updTs-Some*:
⟦ *ts t = ⌊xw⌋*; *thread-oks ts tas* ⟧ ⟹ *redT-updTs ts tas t = ⌊xw⌋*
by(*induct ts tas rule*: *redT-updTs.induct*)(*auto intro*: *redT-updT-Some*)

lemma *redT-updT′-Some*:
⟦ *ts t = ⌊xw⌋*; *thread-ok ts ta* ⟧ ⟹ *redT-updT′ ts ta t = ⌊xw⌋*
by(*cases ta*) *auto*

lemma *redT-updTs′-Some*:
⟦ *ts t = ⌊xw⌋*; *thread-oks ts tas* ⟧ ⟹ *redT-updTs′ ts tas t = ⌊xw⌋*
by(*induct ts tas rule*: *redT-updTs′.induct*)(*auto intro*: *redT-updT′-Some*)

lemma *thread-ok-new-thread*:
thread-ok ts (NewThread t m′ x) ⟹ *ts t = None*
by(*auto*)

lemma *thread-oks-new-thread*:
⟦ *thread-oks ts tas*; *NewThread t x m* ∈ *set tas* ⟧ ⟹ *ts t = None*
by(*induct ts tas rule*: *thread-oks.induct*)(*auto intro*: *redT-updT′-None*)

lemma *redT-updT-new-thread-ts*:
thread-ok ts (NewThread t x m) ⟹ *redT-updT ts (NewThread t x m) t = ⌊(x, no-wait-locks)⌋*
by(*simp*)

lemma *redT-updTs-new-thread-ts*:
⟦ *thread-oks ts tas*; *NewThread t x m* ∈ *set tas* ⟧ ⟹ *redT-updTs ts tas t = ⌊(x, no-wait-locks)⌋*
by(*induct ts tas rule*: *redT-updTs.induct*)(*auto intro*: *redT-updTs-Some*)

lemma *redT-updT-new-thread*:
⟦ *redT-updT ts ta t = ⌊(x, w)⌋*; *thread-ok ts ta*; *ts t = None* ⟧ ⟹ ∃ *m*. *ta = NewThread t x m* ∧ *w = no-wait-locks*
by(*cases ta*)(*auto split*: *if-split-asm*)

lemma *redT-updTs-new-thread*:
⟦ *redT-updTs ts tas t = ⌊(x, w)⌋*; *thread-oks ts tas*; *ts t = None* ⟧
⟹ ∃ *m* .*NewThread t x m* ∈ *set tas* ∧ *w = no-wait-locks*
proof(*induct tas arbitrary*: *ts*)
case *Nil* **thus** *?case* **by** *simp*
next

case (*Cons TA TAS TS*)
note *IH* = ‹$\bigwedge$*ts*. ⟦*redT-updTs ts TAS t* = ⌊(*x*, *w*)⌋; *thread-oks ts TAS*; *ts t* = *None*⟧ ⟹ ∃ *m*. *NewThread t x m* ∈ *set TAS* ∧ *w* = *no-wait-locks*›
note *es't* = ‹*redT-updTs TS* (*TA* # *TAS*) *t* = ⌊(*x*, *w*)⌋›
note *cct* = ‹*thread-oks TS* (*TA* # *TAS*)›
hence *cctta*: *thread-ok TS TA* **and** *ccts*: *thread-oks* (*redT-updT TS TA*) *TAS* **by** *auto*
note *est* = ‹*TS t* = *None*›
{ fix *X W*
assume *rest*: *redT-updT TS TA t* = ⌊(*X*, *W*)⌋
then obtain *m* **where** *TA* = *NewThread t X m* ∧ *W* = *no-wait-locks* **using** *cctta est*
by (*auto dest*!: *redT-updT-new-thread*)
then obtain *TA* = *NewThread t X m W* = *no-wait-locks* **..**
moreover from *rest ccts*
have *redT-updTs TS* (*TA* # *TAS*) *t* = ⌊(*X*, *W*)⌋
by(*auto intro*:*redT-updTs-Some*)
with *es't* **have** *X* = *x W* = *w* **by** *auto*
ultimately have *?case* **by** *auto* **}**
moreover
{ assume *rest*: *redT-updT TS TA t* = *None*
hence ⋀*m*. *TA* ≠ *NewThread t x m* **using** *est cct*
by(*clarsimp*)
with *rest ccts es't* **have** *?case* **by**(*auto dest*: *IH*) **}**
ultimately show *?case* **by**(*cases redT-updT TS TA t*, *auto*)
qed

lemma *redT-updT-upd*:
⟦ *ts t* = ⌊*xw*⌋; *thread-ok ts ta* ⟧ ⟹ (*redT-updT ts ta*)(*t* ↦ *xw'*) = *redT-updT* (*ts*(*t* ↦ *xw'*)) *ta*
by(*cases ta*)(*fastforce intro*: *fun-upd-twist*)+

lemma *redT-updTs-upd*:
⟦ *ts t* = ⌊*xw*⌋; *thread-oks ts tas* ⟧ ⟹ (*redT-updTs ts tas*)(*t* ↦ *xw'*) = *redT-updTs* (*ts*(*t* ↦ *xw'*)) *tas*
by(*induct ts tas rule*: *redT-updTs.induct*)(*auto simp del*: *fun-upd-apply simp add*: *redT-updT-upd dest*: *redT-updT-Some*)

lemma *thread-ok-upd*:
ts t = ⌊*xln*⌋ ⟹ *thread-ok* (*ts*(*t* ↦ *xln'*)) *ta* = *thread-ok ts ta*
by(*rule thread-ok-ts-change*) *simp*

lemma *thread-oks-upd*:
ts t = ⌊*xln*⌋ ⟹ *thread-oks* (*ts*(*t* ↦ *xln'*)) *tas* = *thread-oks ts tas*
by(*rule thread-oks-ts-change*) *simp*

lemma *thread-ok-convert-new-thread-action* [*simp*]:
thread-ok ts (*convert-new-thread-action f ta*) = *thread-ok ts ta*
by(*cases ta*) *auto*

lemma *redT-updT'-convert-new-thread-action-eq-None*:
redT-updT' ts (*convert-new-thread-action f ta*) *t* = *None* ⟷ *redT-updT' ts ta t* = *None*
by(*cases ta*) *auto*

lemma *thread-oks-convert-new-thread-action* [*simp*]:
thread-oks ts (*map* (*convert-new-thread-action f*) *tas*) = *thread-oks ts tas*
by(*induct ts tas rule*: *thread-oks.induct*)(*simp-all add*: *thread-oks-ts-change*[*OF redT-updT'-convert-new-thread-action-eq-None*

lemma *map-redT-updT*:
map-option (*map-prod f id*) (*redT-updT ts ta t*) =
redT-updT (λ*t*. *map-option* (*map-prod f id*) (*ts t*)) (*convert-new-thread-action f ta*) *t*
by(*cases ta*) *auto*

lemma *map-redT-updTs*:
map-option (*map-prod f id*) (*redT-updTs ts tas t*) =
redT-updTs (λ*t*. *map-option* (*map-prod f id*) (*ts t*)) (*map* (*convert-new-thread-action f*) *tas*) *t*
by(*induct tas arbitrary*: *ts*)(*auto simp add*: *map-redT-updT*)

end

1.5 Semantics of the thread actions for wait, notify and interrupt

theory *FWWait*
imports
FWState
begin

Update functions for the wait sets in the multithreaded state

inductive *redT-updW* :: *'t* ⇒ (*'w*, *'t*) *wait-sets* ⇒ (*'t*,*'w*) *wait-set-action* ⇒ (*'w*,*'t*) *wait-sets* ⇒ *bool*
for *t* :: *'t* **and** *ws* :: (*'w*, *'t*) *wait-sets*
where
ws t' = ⌊*InWS w*⌋ ⟹ *redT-updW t ws* (*Notify w*) (*ws*(*t'* ↦ *PostWS WSNotified*))
| (⋀*t'*. *ws t'* ≠ ⌊*InWS w*⌋) ⟹ *redT-updW t ws* (*Notify w*) *ws*
| *redT-updW t ws* (*NotifyAll w*) (λ*t*. *if ws t* = ⌊*InWS w*⌋ *then* ⌊*PostWS WSNotified*⌋ *else ws t*)
| *redT-updW t ws* (*Suspend w*) (*ws*(*t* ↦ *InWS w*))
| *ws t'* = ⌊*InWS w*⌋ ⟹ *redT-updW t ws* (*WakeUp t'*) (*ws*(*t'* ↦ *PostWS WSInterrupted*))
| (⋀*w*. *ws t'* ≠ ⌊*InWS w*⌋) ⟹ *redT-updW t ws* (*WakeUp t'*) *ws*
| *redT-updW t ws Notified* (*ws*(*t* := *None*))
| *redT-updW t ws WokenUp* (*ws*(*t* := *None*))

definition *redT-updWs* :: *'t* ⇒ (*'w*,*'t*) *wait-sets* ⇒ (*'t*,*'w*) *wait-set-action list* ⇒ (*'w*,*'t*) *wait-sets* ⇒ *bool*
where *redT-updWs t* = *rtrancl3p* (*redT-updW t*)

inductive-simps *redT-updW-simps* [*simp*]:
redT-updW t ws (*Notify w*) *ws'*
redT-updW t ws (*NotifyAll w*) *ws'*
redT-updW t ws (*Suspend w*) *ws'*
redT-updW t ws (*WakeUp t'*) *ws'*
redT-updW t ws WokenUp ws'
redT-updW t ws Notified ws'

lemma *redT-updW-total*: ∃ *ws'*. *redT-updW t ws wa ws'*
by(*cases wa*)(*auto simp add*: *redT-updW.simps*)

lemma *redT-updWs-total*: ∃ *ws'*. *redT-updWs t ws was ws'*
proof(*induct was rule*: *rev-induct*)
case *Nil* **thus** *?case* **by**(*auto simp add*: *redT-updWs-def*)
next
case (*snoc wa was*)

then obtain *ws′* **where** *redT-updWs t ws was ws′* **..**
also from *redT-updW-total*[*of t ws′ wa*]
obtain *ws″* **where** *redT-updW t ws′ wa ws″* **..**
ultimately show *?case* **unfolding** *redT-updWs-def* **by**(*auto intro*: *rtrancl3p-step*)
qed

lemma *redT-updWs-trans*: ⟦ *redT-updWs t ws was ws′*; *redT-updWs t ws′ was′ ws″* ⟧ ⟹ *redT-updWs t ws* (*was* @ *was′*) *ws″*
unfolding *redT-updWs-def* **by**(*rule rtrancl3p-trans*)

lemma *redT-updW-None-implies-None*:
⟦ *redT-updW t′ ws wa ws′*; *ws t* = *None*; *t* ≠ *t′* ⟧ ⟹ *ws′ t* = *None*
by(*auto simp add*: *redT-updW.simps*)

lemma *redT-updWs-None-implies-None*:
assumes *redT-updWs t′ ws was ws′*
and *t* ≠ *t′* **and** *ws t* = *None*
shows *ws′ t* = *None*
using ‹*redT-updWs t′ ws was ws′*› ‹*ws t* = *None*› **unfolding** *redT-updWs-def*
by *induct*(*auto intro*: *redT-updW-None-implies-None*[*OF - -* ‹*t* ≠ *t′*›])

lemma *redT-updW-PostWS-imp-PostWS*:
⟦ *redT-updW t ws wa ws′*; *ws t″* = ⌊*PostWS w*⌋; *t″* ≠ *t* ⟧ ⟹ *ws′ t″* = ⌊*PostWS w*⌋
by(*auto simp add*: *redT-updW.simps*)

lemma *redT-updWs-PostWS-imp-PostWS*:
⟦ *redT-updWs t ws was ws′*; *t″* ≠ *t*; *ws t″* = ⌊*PostWS w*⌋ ⟧ ⟹ *ws′ t″* = ⌊*PostWS w*⌋
unfolding *redT-updWs-def*
by(*induct rule*: *rtrancl3p.induct*)(*auto dest*: *redT-updW-PostWS-imp-PostWS*)

lemma *redT-updW-Some-otherD*:
⟦ *redT-updW t′ ws wa ws′*; *ws′ t* = ⌊*w*⌋; *t* ≠ *t′* ⟧
⟹ (*case w of InWS w′* ⇒ *ws t* = ⌊*InWS w′*⌋ | - ⇒ *ws t* = ⌊*w*⌋ ∨ (∃ *w′*. *ws t* = ⌊*InWS w′*⌋))
by(*auto simp add*: *redT-updW.simps split*: *if-split-asm wait-set-status.split*)

lemma *redT-updWs-Some-otherD*:
⟦ *redT-updWs t′ ws was ws′*; *ws′ t* = ⌊*w*⌋; *t* ≠ *t′* ⟧
⟹ (*case w of InWS w′* ⇒ *ws t* = ⌊*InWS w′*⌋ | - ⇒ *ws t* = ⌊*w*⌋ ∨ (∃ *w′*. *ws t* = ⌊*InWS w′*⌋))
unfolding *redT-updWs-def*
apply(*induct arbitrary*: *w rule*: *rtrancl3p.induct*)
apply(*fastforce split*: *wait-set-status.splits dest*: *redT-updW-Some-otherD*)+
done

lemma *redT-updW-None-SomeD*:
⟦ *redT-updW t ws wa ws′*; *ws′ t′* = ⌊*w*⌋; *ws t′* = *None* ⟧ ⟹ *t* = *t′* ∧ (∃ *w′*. *w* = *InWS w′* ∧ *wa* = *Suspend w′*)
by(*auto simp add*: *redT-updW.simps split*: *if-split-asm*)

lemma *redT-updWs-None-SomeD*:
⟦ *redT-updWs t ws was ws′*; *ws′ t′* = ⌊*w*⌋; *ws t′* = *None* ⟧ ⟹ *t* = *t′* ∧ (∃ *w′*. *Suspend w′* ∈ *set was*)
unfolding *redT-updWs-def*
proof(*induct arbitrary*: *w rule*: *rtrancl3p.induct*)
case (*rtrancl3p-refl ws*) **thus** *?case* **by** *simp*
next

```
  case (rtrancl3p-step ws was ws' wa ws'')
  show ?case
  proof(cases ws' t')
    case None
    from redT-updW-None-SomeD[OF ‹redT-updW t ws' wa ws''›, OF ‹ws'' t' = ⌊w⌋› this]
    show ?thesis by auto
  next
    case (Some w')
    with ‹ws t' = None› rtrancl3p-step.hyps(2) show ?thesis by auto
  qed
qed

lemma redT-updW-neq-Some-SomeD:
  ⟦ redT-updW t' ws wa ws'; ws' t = ⌊InWS w⌋; ws t ≠ ⌊InWS w⌋ ⟧ ⟹ t = t' ∧ wa = Suspend w
by(auto simp add: redT-updW.simps split: if-split-asm)

lemma redT-updWs-neq-Some-SomeD:
  ⟦ redT-updWs t ws was ws'; ws' t' = ⌊InWS w⌋; ws t' ≠ ⌊InWS w⌋ ⟧ ⟹ t = t' ∧ Suspend w ∈ set
was
unfolding redT-updWs-def
proof(induct rule: rtrancl3p.induct)
  case rtrancl3p-refl thus ?case by simp
next
  case (rtrancl3p-step ws was ws' wa ws'')
  show ?case
  proof(cases ws' t' = ⌊InWS w⌋)
    case True
    with ‹ws t' ≠ ⌊InWS w⌋› ‹⟦ws' t' = ⌊InWS w⌋; ws t' ≠ ⌊InWS w⌋⟧ ⟹ t = t' ∧ Suspend w ∈ set
was›
    show ?thesis by simp
  next
    case False
    with ‹redT-updW t ws' wa ws''› ‹ws'' t' = ⌊InWS w⌋›
    have t' = t ∧ wa = Suspend w by(rule redT-updW-neq-Some-SomeD)
    thus ?thesis by auto
  qed
qed

lemma redT-updW-not-Suspend-Some:
  ⟦ redT-updW t ws wa ws'; ws' t = ⌊w'⌋; ws t = ⌊w⌋; ⋀w. wa ≠ Suspend w ⟧
  ⟹ w' = w ∨ (∃ w'' w'''. w = InWS w'' ∧ w' = PostWS w''')
by(auto simp add: redT-updW.simps split: if-split-asm)

lemma redT-updWs-not-Suspend-Some:
  ⟦ redT-updWs t ws was ws'; ws' t = ⌊w'⌋; ws t = ⌊w⌋; ⋀w. Suspend w ∉ set was ⟧
  ⟹ w' = w ∨ (∃ w'' w'''. w = InWS w'' ∧ w' = PostWS w''')
unfolding redT-updWs-def
proof(induct arbitrary: w rule: rtrancl3p-converse-induct)
  case refl thus ?case by simp
next
  case (step ws wa ws' was ws'')
  note ‹ws'' t = ⌊w'⌋›
  moreover
  have ws' t ≠ None
```

proof
assume *ws′ t = None*
with ‹*rtrancl3p (redT-updW t) ws′ was ws″*› ‹*ws″ t = ⌊w′⌋*›
obtain *w′* **where** *Suspend w′ ∈ set was* **unfolding** *redT-updWs-def*[*symmetric*]
by(*auto dest*: *redT-updWs-None-SomeD*)
with ‹*Suspend w′ ∉ set (wa # was)*› **show** *False* **by** *simp*
qed
then obtain *w″* **where** *ws′ t = ⌊w″⌋* **by** *auto*
moreover {
fix *w*
from ‹*Suspend w ∉ set (wa # was)*› **have** *Suspend w ∉ set was* **by** *simp* **}**
ultimately have *w′ = w″ ∨ (∃ w‴ w⁗. w″ = InWS w‴ ∧ w′ = PostWS w⁗)* **by**(*rule step.hyps*)
moreover { fix *w*
from ‹*Suspend w ∉ set (wa # was)*› **have** *wa ≠ Suspend w* **by** *auto* **}**
note *redT-updW-not-Suspend-Some*[*OF* ‹*redT-updW t ws wa ws′*›, *OF* ‹*ws′ t = ⌊w″⌋*› ‹*ws t = ⌊w⌋*› *this*]
ultimately show *?case* **by** *auto*
qed

lemma *redT-updWs-WokenUp-SuspendD*:
⟦ *redT-updWs t ws was ws′*; *Notified ∈ set was ∨ WokenUp ∈ set was*; *ws′ t = ⌊w⌋* ⟧ ⟹ ∃ *w*. *Suspend w ∈ set was*
unfolding *redT-updWs-def*
by(*induct rule*: *rtrancl3p-converse-induct*)(*auto dest*: *redT-updWs-None-SomeD*[*unfolded redT-updWs-def*])

lemma *redT-updW-Woken-Up-same-no-Notified-Interrupted*:
⟦ *redT-updW t ws wa ws′*; *ws′ t = ⌊PostWS w⌋*; *ws t = ⌊PostWS w⌋*; ⋀*w*. *wa ≠ Suspend w* ⟧
⟹ *wa ≠ Notified ∧ wa ≠ WokenUp*
by(*fastforce*)

lemma *redT-updWs-Woken-Up-same-no-Notified-Interrupted*:
⟦ *redT-updWs t ws was ws′*; *ws′ t = ⌊PostWS w⌋*; *ws t = ⌊PostWS w⌋*; ⋀*w*. *Suspend w ∉ set was* ⟧
⟹ *Notified ∉ set was ∧ WokenUp ∉ set was*
unfolding *redT-updWs-def*
proof(*induct rule*: *rtrancl3p-converse-induct*)
case *refl* **thus** *?case* **by** *simp*
next
case (*step ws wa ws′ was ws″*)
note *Suspend* = ‹⋀*w*. *Suspend w ∉ set (wa # was)*›
note ‹*ws″ t = ⌊PostWS w⌋*›
moreover have *ws′ t = ⌊PostWS w⌋*
proof(*cases ws′ t*)
case *None*
with ‹*rtrancl3p (redT-updW t) ws′ was ws″*› ‹*ws″ t = ⌊PostWS w⌋*›
obtain *w* **where** *Suspend w ∈ set was* **unfolding** *redT-updWs-def*[*symmetric*]
by(*auto dest*: *redT-updWs-None-SomeD*)
with *Suspend*[*of w*] **have** *False* **by** *simp*
thus *?thesis* **..**
next
case (*Some w′*)
thus *?thesis* **using** ‹*ws t = ⌊PostWS w⌋*› *Suspend* ‹*redT-updW t ws wa ws′*›
by(*auto simp add*: *redT-updW.simps split*: *if-split-asm*)
qed
moreover

```
  { fix w from Suspend[of w] have Suspend w ∉ set was by simp }
  ultimately have Notified ∉ set was ∧ WokenUp ∉ set was by(rule step.hyps)
  moreover
  { fix w from Suspend[of w] have wa ≠ Suspend w by auto }
  with ‹redT-updW t ws wa ws'› ‹ws' t = ⌊PostWS w⌋› ‹ws t = ⌊PostWS w⌋›
  have wa ≠ Notified ∧ wa ≠ WokenUp by(rule redT-updW-Woken-Up-same-no-Notified-Interrupted)
  ultimately show ?case by auto
qed
```

Preconditions for wait set actions

```
definition wset-actions-ok :: ('w,'t) wait-sets ⇒ 't ⇒ ('t,'w) wait-set-action list ⇒ bool
where
  wset-actions-ok ws t was ⟷
  (if Notified ∈ set was then ws t = ⌊PostWS WSNotified⌋
   else if WokenUp ∈ set was then ws t = ⌊PostWS WSWokenUp⌋
   else ws t = None)

lemma wset-actions-ok-Nil [simp]:
  wset-actions-ok ws t [] ⟷ ws t = None
by(simp add: wset-actions-ok-def)

definition waiting :: 'w wait-set-status option ⇒ bool
where waiting w ⟷ (∃ w'. w = ⌊InWS w'⌋)

lemma not-waiting-iff:
  ¬ waiting w ⟷ w = None ∨ (∃ w'. w = ⌊PostWS w'⌋)
apply(cases w)
apply(case-tac [2] a)
apply(auto simp add: waiting-def)
done

lemma waiting-code [code]:
  waiting None = False
  ⋀w. waiting ⌊PostWS w⌋ = False
  ⋀w. waiting ⌊InWS w⌋ = True
by(simp-all add: waiting-def)

end
```

1.6 Semantics of the thread actions for purely conditional purpose such as Join

```
theory FWCondAction
imports
  FWState
begin

locale final-thread =
  fixes final :: 'x ⇒ bool
begin

primrec cond-action-ok :: ('l,'t,'x,'m,'w) state ⇒ 't ⇒ 't conditional-action ⇒ bool where
  ⋀ln. cond-action-ok s t (Join T) =
```

(*case thr s T of None ⇒ True | ⌊(x, ln)⌋ ⇒ t ≠ T ∧ final x ∧ ln = no-wait-locks ∧ wset s T = None*)
| *cond-action-ok s t Yield = True*

primrec *cond-action-oks* :: (*'l*,*'t*,*'x*,*'m*,*'w*) *state* ⇒ *'t* ⇒ *'t conditional-action list* ⇒ *bool* **where**
cond-action-oks s t [] = *True*
| *cond-action-oks s t* (*ct*#*cts*) = (*cond-action-ok s t ct* ∧ *cond-action-oks s t cts*)

lemma *cond-action-oks-append* [*simp*]:
cond-action-oks s t (*cts* @ *cts'*) ⟷ *cond-action-oks s t cts* ∧ *cond-action-oks s t cts'*
by(*induct cts*, *auto*)

lemma *cond-action-oks-conv-set*:
cond-action-oks s t cts ⟷ (∀ *ct* ∈ *set cts*. *cond-action-ok s t ct*)
by(*induct cts*) *simp-all*

lemma *cond-action-ok-Join*:
⋀*ln*. ⟦ *cond-action-ok s t* (*Join T*); *thr s T* = ⌊(*x*, *ln*)⌋ ⟧ ⟹ *final x* ∧ *ln* = *no-wait-locks* ∧ *wset s T* = *None*
by(*auto*)

lemma *cond-action-oks-Join*:
⋀*ln*. ⟦ *cond-action-oks s t cas*; *Join T* ∈ *set cas*; *thr s T* = ⌊(*x*, *ln*)⌋ ⟧
⟹ *final x* ∧ *ln* = *no-wait-locks* ∧ *wset s T* = *None* ∧ *t* ≠ *T*
by(*induct cas*)(*auto*)

lemma *cond-action-oks-upd*:
assumes *tst*: *thr s t* = ⌊*xln*⌋
shows *cond-action-oks* (*locks s*, ((*thr s*)(*t* ↦ *xln'*), *shr s*), *wset s*, *interrupts s*) *t cas* = *cond-action-oks s t cas*
proof(*induct cas*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons ca cas*)
from *tst* **have** *eq*: *cond-action-ok* (*locks s*, ((*thr s*)(*t* ↦ *xln'*), *shr s*), *wset s*, *interrupts s*) *t ca* = *cond-action-ok s t ca*
by(*cases ca*) *auto*
with *Cons* **show** *?case* **by**(*auto simp del*: *fun-upd-apply*)
qed

lemma *cond-action-ok-shr-change*:
cond-action-ok (*ls*, (*ts*, *m*), *ws*, *is*) *t ct* ⟹ *cond-action-ok* (*ls*, (*ts*, *m'*), *ws*, *is*) *t ct*
by(*cases ct*) *auto*

lemma *cond-action-oks-shr-change*:
cond-action-oks (*ls*, (*ts*, *m*), *ws*, *is*) *t cts* ⟹ *cond-action-oks* (*ls*, (*ts*, *m'*), *ws*, *is*) *t cts*
by(*auto simp add*: *cond-action-oks-conv-set intro*: *cond-action-ok-shr-change*)

primrec *cond-action-ok'* :: (*'l*,*'t*,*'x*,*'m*,*'w*) *state* ⇒ *'t* ⇒ *'t conditional-action* ⇒ *bool*
where
cond-action-ok' - - (*Join t*) = *True*
| *cond-action-ok'* - - *Yield* = *True*

primrec *cond-action-oks'* :: (*'l*,*'t*,*'x*,*'m*,*'w*) *state* ⇒ *'t* ⇒ *'t conditional-action list* ⇒ *bool* **where**

cond-action-oks′ s t [] = *True*
| *cond-action-oks′ s t* (*ct*#*cts*) = (*cond-action-ok′ s t ct* ∧ *cond-action-oks′ s t cts*)

lemma *cond-action-oks′-append* [*simp*]:
cond-action-oks′ s t (*cts* @ *cts′*) ⟷ *cond-action-oks′ s t cts* ∧ *cond-action-oks′ s t cts′*
by(*induct cts*, *auto*)

lemma *cond-action-oks′-subset-Join*:
set cts ⊆ *insert Yield* (*range Join*) ⟹ *cond-action-oks′ s t cts*
apply(*induct cts*)
apply(*auto*)
done

end

definition *collect-cond-actions* :: *′t conditional-action list* ⇒ *′t set* **where**
collect-cond-actions cts = {*t*. *Join t* ∈ *set cts*}

declare *collect-cond-actions-def* [*simp*]

lemma *cond-action-ok-final-change*:
⟦ *final-thread.cond-action-ok final1 s1 t ca*;
⋀*t*. *thr s1 t* = *None* ⟷ *thr s2 t* = *None*;
⋀*t x1*. ⟦ *thr s1 t* = ⌊(*x1*, *no-wait-locks*)⌋; *final1 x1*; *wset s1 t* = *None* ⟧
⟹ ∃ *x2*. *thr s2 t* = ⌊(*x2*, *no-wait-locks*)⌋ ∧ *final2 x2* ∧ *ln2* = *no-wait-locks* ∧ *wset s2 t* = *None* ⟧
⟹ *final-thread.cond-action-ok final2 s2 t ca*
apply(*cases ca*)
apply(*fastforce simp add*: *final-thread.cond-action-ok.simps*)+
done

lemma *cond-action-oks-final-change*:
assumes *major*: *final-thread.cond-action-oks final1 s1 t cas*
and *minor*: ⋀*t*. *thr s1 t* = *None* ⟷ *thr s2 t* = *None*
⋀*t x1*. ⟦ *thr s1 t* = ⌊(*x1*, *no-wait-locks*)⌋; *final1 x1*; *wset s1 t* = *None* ⟧
⟹ ∃ *x2*. *thr s2 t* = ⌊(*x2*, *no-wait-locks*)⌋ ∧ *final2 x2* ∧ *ln2* = *no-wait-locks* ∧ *wset s2 t* = *None*
shows *final-thread.cond-action-oks final2 s2 t cas*
using *major*
by(*induct cas*)(*auto simp add*: *final-thread.cond-action-oks.simps intro*: *cond-action-ok-final-change*[*OF* - *minor*])

end

1.7 Wellformedness conditions for the multithreaded state

theory *FWWellform*
imports
FWLocking
FWThread
FWWait
FWCondAction
begin

Well-formedness property: Locks are held by real threads

definition

lock-thread-ok :: ('*l*, '*t*) *locks* ⇒ ('*l*, '*t*,'*x*) *thread-info* ⇒ *bool*
where [*code del*]:
lock-thread-ok ls ts ≡ ∀ *l t*. *has-lock* (*ls* $ *l*) *t* ⟶ (∃ *xw*. *ts t* = ⌊*xw*⌋)

lemma *lock-thread-ok-code* [*code*]:
lock-thread-ok ls ts = *finfun-All* ((λ*l*. *case l of None* ⇒ *True* | ⌊(*t*, *n*)⌋ ⇒ (*ts t* ≠ *None*)) ∘$ *ls*)
by(*simp add*: *lock-thread-ok-def finfun-All-All has-lock-has-locks-conv has-locks-iff o-def*)

lemma *lock-thread-okI*:
(⋀*l t*. *has-lock* (*ls* $ *l*) *t* ⟹ ∃ *xw*. *ts t* = ⌊*xw*⌋) ⟹ *lock-thread-ok ls ts*
by(*auto simp add*: *lock-thread-ok-def*)

lemma *lock-thread-okD*:
⟦ *lock-thread-ok ls ts*; *has-lock* (*ls* $ *l*) *t* ⟧ ⟹ ∃ *xw*. *ts t* = ⌊*xw*⌋
by(*fastforce simp add*: *lock-thread-ok-def*)

lemma *lock-thread-okD'*:
⟦ *lock-thread-ok ls ts*; *has-locks* (*ls* $ *l*) *t* = *Suc n* ⟧ ⟹ ∃ *xw*. *ts t* = ⌊*xw*⌋
by(*auto elim*: *lock-thread-okD*[**where** *l*=*l*] *simp del*: *split-paired-Ex*)

lemma *lock-thread-okE*:
⟦ *lock-thread-ok ls ts*; ∀ *l t*. *has-lock* (*ls* $ *l*) *t* ⟶ (∃ *xw*. *ts t* = ⌊*xw*⌋) ⟹ *P* ⟧ ⟹ *P*
by(*auto simp add*: *lock-thread-ok-def simp del*: *split-paired-Ex*)

lemma *lock-thread-ok-upd*:
lock-thread-ok ls ts ⟹ *lock-thread-ok ls* (*ts*(*t* ↦ *xw*))
by(*auto intro*!: *lock-thread-okI dest*: *lock-thread-okD*)

lemma *lock-thread-ok-has-lockE*:
assumes *lock-thread-ok ls ts*
and *has-lock* (*ls* $ *l*) *t*
obtains *x ln'* **where** *ts t* = ⌊(*x*, *ln'*)⌋
using *assms*
by(*auto dest*!: *lock-thread-okD*)

lemma *redT-updLs-preserves-lock-thread-ok*:
assumes *lto*: *lock-thread-ok ls ts*
and *tst*: *ts t* = ⌊*xw*⌋
shows *lock-thread-ok* (*redT-updLs ls t las*) *ts*
proof(*rule lock-thread-okI*)
fix *L T*
assume *ru*: *has-lock* (*redT-updLs ls t las* $ *L*) *T*
show ∃ *xw*. *ts T* = ⌊*xw*⌋
proof(*cases t* = *T*)
case *True*
thus *?thesis* **using** *tst lto*
by(*auto elim*: *lock-thread-okE*)
next
case *False*
with *ru* **have** *has-lock* (*ls* $ *L*) *T*
by(*rule redT-updLs-Some-thread-idD*)
thus *?thesis* **using** *lto*
by(*auto elim*!: *lock-thread-okE simp del*: *split-paired-Ex*)
qed

qed

lemma *redT-updTs-preserves-lock-thread-ok*:
  assumes *lto*: *lock-thread-ok ls ts*
  shows *lock-thread-ok ls* (*redT-updTs ts nts*)
proof(*rule lock-thread-okI*)
  fix *l t*
  assume *has-lock* (*ls* $ *l*) *t*
  with *lto* **have** ∃ *xw*. *ts t* = ⌊*xw*⌋
    by(*auto elim*!: *lock-thread-okE simp del*: *split-paired-Ex*)
  thus ∃ *xw*. *redT-updTs ts nts t* = ⌊*xw*⌋
    by(*auto intro*: *redT-updTs-Some1 simp del*: *split-paired-Ex*)
qed

lemma *lock-thread-ok-has-lock*:
  assumes *lock-thread-ok ls ts*
  and *has-lock* (*ls* $ *l*) *t*
  obtains *xw* **where** *ts t* = ⌊*xw*⌋
using *assms*
by(*auto dest*!: *lock-thread-okD*)

lemma *lock-thread-ok-None-has-locks-0*:
  ⟦ *lock-thread-ok ls ts*; *ts t* = *None* ⟧ ⟹ *has-locks* (*ls* $ *l*) *t* = *0*
by(*rule ccontr*)(*auto dest*: *lock-thread-okD*)

lemma *redT-upds-preserves-lock-thread-ok*:
  ⟦*lock-thread-ok ls ts*; *ts t* = ⌊*xw*⌋; *thread-oks ts tas*⟧
  ⟹ *lock-thread-ok* (*redT-updLs ls t las*) ((*redT-updTs ts tas*)(*t* ↦ *xw*′))
apply(*rule lock-thread-okI*)
apply(*clarsimp simp del*: *split-paired-Ex*)
apply(*drule has-lock-upd-locks-implies-has-lock*, *simp*)
apply(*drule lock-thread-okD*, *assumption*)
apply(*erule exE*)
by(*rule redT-updTs-Some1*)

lemma *acquire-all-preserves-lock-thread-ok*:
  fixes *ln*
  shows ⟦ *lock-thread-ok ls ts*; *ts t* = ⌊(*x*, *ln*)⌋ ⟧ ⟹ *lock-thread-ok* (*acquire-all ls t ln*) (*ts*(*t* ↦ *xw*))
by(*rule lock-thread-okI*)(*auto dest*!: *has-lock-acquire-locks-implies-has-lock dest*: *lock-thread-okD*)

Well-formedness condition: Wait sets contain only real threads

definition *wset-thread-ok* :: (′*w*, ′*t*) *wait-sets* ⇒ (′*l*, ′*t*, ′*x*) *thread-info* ⇒ *bool*
where *wset-thread-ok ws ts* ≡ ∀ *t*. *ts t* = *None* ⟶ *ws t* = *None*

lemma *wset-thread-okI*:
  (⋀*t*. *ts t* = *None* ⟹ *ws t* = *None*) ⟹ *wset-thread-ok ws ts*
by(*simp add*: *wset-thread-ok-def*)

lemma *wset-thread-okD*:
  ⟦ *wset-thread-ok ws ts*; *ts t* = *None* ⟧ ⟹ *ws t* = *None*
by(*simp add*: *wset-thread-ok-def*)

lemma *wset-thread-ok-conv-dom*:
  wset-thread-ok ws ts ⟷ *dom ws* ⊆ *dom ts*

by(*auto simp add*: *wset-thread-ok-def*)

lemma *wset-thread-ok-upd*:
wset-thread-ok ls ts $\Longrightarrow$ *wset-thread-ok ls* (*ts*($t \mapsto xw$))
by(*auto intro*!: *wset-thread-okI dest*: *wset-thread-okD split*: *if-split-asm*)

lemma *wset-thread-ok-upd-None*:
wset-thread-ok ws ts $\Longrightarrow$ *wset-thread-ok* (*ws*(*t* := *None*)) (*ts*(*t* := *None*))
by(*auto intro*!: *wset-thread-okI dest*: *wset-thread-okD split*: *if-split-asm*)

lemma *wset-thread-ok-upd-Some*:
wset-thread-ok ws ts $\Longrightarrow$ *wset-thread-ok* (*ws*(*t* := *wo*)) (*ts*($t \mapsto xln$))
by(*auto intro*!: *wset-thread-okI dest*: *wset-thread-okD split*: *if-split-asm*)

lemma *wset-thread-ok-upd-ws*:
$[\![$ *wset-thread-ok ws ts*; *ts t* = $\lfloor xln \rfloor$ $]\!]$ $\Longrightarrow$ *wset-thread-ok* (*ws*(*t* := *w*)) *ts*
by(*auto intro*!: *wset-thread-okI dest*: *wset-thread-okD*)

lemma *wset-thread-ok-NotifyAllI*:
wset-thread-ok ws ts $\Longrightarrow$ *wset-thread-ok* (λt. *if ws t* = $\lfloor w\ t \rfloor$ *then* $\lfloor w'\ t \rfloor$ *else ws t*) *ts*
by(*simp add*: *wset-thread-ok-def*)

lemma *redT-updTs-preserves-wset-thread-ok*:
assumes *wto*: *wset-thread-ok ws ts*
shows *wset-thread-ok ws* (*redT-updTs ts nts*)
proof(*rule wset-thread-okI*)
fix *t*
assume *redT-updTs ts nts t* = *None*
hence *ts t* = *None* **by**(*rule redT-updTs-None*)
with *wto* **show** *ws t* = *None* **by**(*rule wset-thread-okD*)
qed

lemma *redT-updW-preserve-wset-thread-ok*:
$[\![$ *wset-thread-ok ws ts*; *redT-updW t ws wa ws'*; *ts t* = $\lfloor xln \rfloor$ $]\!]$ $\Longrightarrow$ *wset-thread-ok ws' ts*
by(*fastforce simp add*: *redT-updW.simps intro*: *wset-thread-okI wset-thread-ok-NotifyAllI wset-thread-ok-upd-ws dest*: *wset-thread-okD*)

lemma *redT-updWs-preserve-wset-thread-ok*:
$[\![$ *wset-thread-ok ws ts*; *redT-updWs t ws was ws'*; *ts t* = $\lfloor xln \rfloor$ $]\!]$ $\Longrightarrow$ *wset-thread-ok ws' ts*
unfolding *redT-updWs-def* **apply**(*rotate-tac 1*)
by(*induct rule*: *rtrancl3p-converse-induct*)(*auto intro*: *redT-updW-preserve-wset-thread-ok*)

Well-formedness condition: Wait sets contain only non-final threads

context *final-thread* **begin**

definition *wset-final-ok* :: ('*w*, '*t*) *wait-sets* $\Rightarrow$ ('*l*, '*t*, '*x*) *thread-info* $\Rightarrow$ *bool*
where *wset-final-ok ws ts* $\longleftrightarrow$ ($\forall t \in$ *dom ws*. $\exists x\ ln$. *ts t* = $\lfloor (x, ln) \rfloor \wedge \neg$ *final x*)

lemma *wset-final-okI*:
($\bigwedge t\ w$. *ws t* = $\lfloor w \rfloor \Longrightarrow \exists x\ ln$. *ts t* = $\lfloor (x, ln) \rfloor \wedge \neg$ *final x*) $\Longrightarrow$ *wset-final-ok ws ts*
unfolding *wset-final-ok-def* **by**(*blast*)

lemma *wset-final-okD*:
$[\![$ *wset-final-ok ws ts*; *ws t* = $\lfloor w \rfloor$ $]\!]$ $\Longrightarrow$ $\exists x\ ln$. *ts t* = $\lfloor (x, ln) \rfloor \wedge \neg$ *final x*

unfolding *wset-final-ok-def* **by**(*blast*)

lemma *wset-final-okE*:
  assumes *wset-final-ok ws ts ws t* = ⌊*w*⌋
  and ⋀*x ln*. *ts t* = ⌊(*x*, *ln*)⌋ ⟹ ¬ *final x* ⟹ *thesis*
  shows *thesis*
using *assms* **by**(*blast dest*: *wset-final-okD*)

lemma *wset-final-ok-imp-wset-thread-ok*:
  wset-final-ok ws ts ⟹ *wset-thread-ok ws ts*
apply(*rule wset-thread-okI*)
apply(*rule ccontr*)
apply(*auto elim*: *wset-final-okE*)
done

end

end

1.8 Semantics of the thread action ReleaseAcquire for the thread state

theory *FWLockingThread*
imports
  FWLocking
begin

fun *upd-threadR* :: *nat* ⇒ *'t lock* ⇒ *'t* ⇒ *lock-action* ⇒ *nat*
where
  upd-threadR n l t ReleaseAcquire = *n* + *has-locks l t*
| *upd-threadR n l t* - = *n*

primrec *upd-threadRs* :: *nat* ⇒ *'t lock* ⇒ *'t* ⇒ *lock-action list* ⇒ *nat*
where
  upd-threadRs n l t [] = *n*
| *upd-threadRs n l t* (*la* # *las*) = *upd-threadRs* (*upd-threadR n l t la*) (*upd-lock l t la*) *t las*

lemma *upd-threadRs-append* [*simp*]:
  upd-threadRs n l t (*las* @ *las'*) = *upd-threadRs* (*upd-threadRs n l t las*) (*upd-locks l t las*) *t las'*
by(*induct las arbitrary*: *n l*, *auto*)

definition *redT-updLns* :: (*'l*,*'t*) *locks* ⇒ *'t* ⇒ (*'l* ⇒*f nat*) ⇒ *'l lock-actions* ⇒ (*'l* ⇒*f nat*)
where ⋀*ln*. *redT-updLns ls t ln las* = (λ(*l*, *n*, *la*). *upd-threadRs n l t la*) ∘$ ($*ls*, ($*ln*, *las*$)$)

lemma *redT-updLns-iff* [*simp*]:
  ⋀*ln*. *redT-updLns ls t ln las* $ *l* = *upd-threadRs* (*ln* $ *l*) (*ls* $ *l*) *t* (*las* $ *l*)
by(*simp add*: *redT-updLns-def*)

lemma *upd-threadRs-comp-empty* [*simp*]: (λ(*l*, *n*, *las*). *upd-threadRs n l t las*) ∘$ ($*ls*, ($*lns*, *K*$ []$)$)
= *lns*
by(*auto intro*!: *finfun-ext*)

lemma *redT-updLs-empty* [*simp*]: *redT-updLs ls t* (*K*$ []) = *ls*

by(*simp add*: *redT-updLs-def*)

end

1.9 Semantics of the thread actions for interruption

theory *FWInterrupt*
imports
FWState
begin

primrec *redT-updI* :: *'t interrupts* ⇒ *'t interrupt-action* ⇒ *'t interrupts*
where
redT-updI is (*Interrupt t*) = *insert t is*
| *redT-updI is* (*ClearInterrupt t*) = *is* − {*t*}
| *redT-updI is* (*IsInterrupted t b*) = *is*

fun *redT-updIs* :: *'t interrupts* ⇒ *'t interrupt-action list* ⇒ *'t interrupts*
where
redT-updIs is [] = *is*
| *redT-updIs is* (*ia* # *ias*) = *redT-updIs* (*redT-updI is ia*) *ias*

primrec *interrupt-action-ok* :: *'t interrupts* ⇒ *'t interrupt-action* ⇒ *bool*
where
interrupt-action-ok is (*Interrupt t*) = *True*
| *interrupt-action-ok is* (*ClearInterrupt t*) = *True*
| *interrupt-action-ok is* (*IsInterrupted t b*) = (*b* = (*t* ∈ *is*))

fun *interrupt-actions-ok* :: *'t interrupts* ⇒ *'t interrupt-action list* ⇒ *bool*
where
interrupt-actions-ok is [] = *True*
| *interrupt-actions-ok is* (*ia* # *ias*) ⟷ *interrupt-action-ok is ia* ∧ *interrupt-actions-ok* (*redT-updI is ia*) *ias*

primrec *interrupt-action-ok'* :: *'t interrupts* ⇒ *'t interrupt-action* ⇒ *bool*
where
interrupt-action-ok' is (*Interrupt t*) = *True*
| *interrupt-action-ok' is* (*ClearInterrupt t*) = *True*
| *interrupt-action-ok' is* (*IsInterrupted t b*) = (*b* ∨ *t* ∉ *is*)

fun *interrupt-actions-ok'* :: *'t interrupts* ⇒ *'t interrupt-action list* ⇒ *bool*
where
interrupt-actions-ok' is [] = *True*
| *interrupt-actions-ok' is* (*ia* # *ias*) ⟷ *interrupt-action-ok' is ia* ∧ *interrupt-actions-ok'* (*redT-updI is ia*) *ias*

fun *collect-interrupt* :: *'t interrupt-action* ⇒ *'t set* ⇒ *'t set*
where
collect-interrupt (*IsInterrupted t True*) *Ts* = *insert t Ts*
| *collect-interrupt* (*Interrupt t*) *Ts* = *Ts* − {*t*}
| *collect-interrupt* - *Ts* = *Ts*

definition *collect-interrupts* :: *'t interrupt-action list* ⇒ *'t set*

where *collect-interrupts ias* = *foldr collect-interrupt ias* {}

lemma *collect-interrupts-interrupted*:
⟦ *interrupt-actions-ok is ias*; $t' \in$ *collect-interrupts ias* ⟧ ⟹ $t' \in$ *is*
unfolding *collect-interrupts-def*
proof(*induct ias arbitrary*: *is*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons ia ias*) **thus** *?case*
by(*cases* (*ia*, *foldr collect-interrupt ias* {}) *rule*: *collect-interrupt.cases*) *auto*
qed

lemma *interrupt-actions-ok-append* [*simp*]:
interrupt-actions-ok is (*ias* @ *ias'*) ⟷ *interrupt-actions-ok is ias* ∧ *interrupt-actions-ok* (*redT-updIs is ias*) *ias'*
by(*induct ias arbitrary*: *is*) *auto*

lemma *collect-interrupt-subset*: *Ts* ⊆ *Ts'* ⟹ *collect-interrupt ia Ts* ⊆ *collect-interrupt ia Ts'*
by(*cases* (*ia*, *Ts*) *rule*: *collect-interrupt.cases*) *auto*

lemma *foldr-collect-interrupt-subset*:
Ts ⊆ *Ts'* ⟹ *foldr collect-interrupt ias Ts* ⊆ *foldr collect-interrupt ias Ts'*
by(*induct ias*)(*simp-all add*: *collect-interrupt-subset*)

lemma *interrupt-actions-ok-all-nthI*:
assumes ⋀*n*. *n* < *length ias* ⟹ *interrupt-action-ok* (*redT-updIs is* (*take n ias*)) (*ias* ! *n*)
shows *interrupt-actions-ok is ias*
using *assms*
proof(*induct ias arbitrary*: *is*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons ia ias*)
from *Cons.prems*[*of 0*] **have** *interrupt-action-ok is ia* **by** *simp*
moreover
{ fix *n*
assume *n* < *length ias*
hence *interrupt-action-ok* (*redT-updIs* (*redT-updI is ia*) (*take n ias*)) (*ias* ! *n*)
using *Cons.prems*[*of Suc n*] **by** *simp* **}**
hence *interrupt-actions-ok* (*redT-updI is ia*) *ias* **by**(*rule Cons.hyps*)
ultimately show *?case* **by** *simp*
qed

lemma *interrupt-actions-ok-nthD*:
assumes *interrupt-actions-ok is ias*
and *n* < *length ias*
shows *interrupt-action-ok* (*redT-updIs is* (*take n ias*)) (*ias* ! *n*)
using *assms*
by(*induct n arbitrary*: *is ias*)(*case-tac* [!] *ias*, *auto*)

lemma *interrupt-actions-ok'-all-nthI*:
assumes ⋀*n*. *n* < *length ias* ⟹ *interrupt-action-ok'* (*redT-updIs is* (*take n ias*)) (*ias* ! *n*)
shows *interrupt-actions-ok' is ias*
using *assms*
proof(*induct ias arbitrary*: *is*)

```
  case Nil thus ?case by simp
next
  case (Cons ia ias)
  from Cons.prems[of 0] have interrupt-action-ok' is ia by simp
  moreover
  { fix n
    assume n < length ias
    hence interrupt-action-ok' (redT-updIs (redT-updI is ia) (take n ias)) (ias ! n)
      using Cons.prems[of Suc n] by simp }
  hence interrupt-actions-ok' (redT-updI is ia) ias by(rule Cons.hyps)
  ultimately show ?case by simp
qed

lemma interrupt-actions-ok'-nthD:
  assumes interrupt-actions-ok' is ias
  and n < length ias
  shows interrupt-action-ok' (redT-updIs is (take n ias)) (ias ! n)
using assms
by(induct n arbitrary: is ias)(case-tac [!] ias, auto)

lemma interrupt-action-ok-imp-interrupt-action-ok' [simp]:
  interrupt-action-ok is ia ⟹ interrupt-action-ok' is ia
by(cases ia) simp-all

lemma interrupt-actions-ok-imp-interrupt-actions-ok' [simp]:
  interrupt-actions-ok is ias ⟹ interrupt-actions-ok' is ias
by(induct ias arbitrary: is)(simp-all)

lemma collect-interruptsE:
  assumes t' ∈ collect-interrupts ias'
  obtains n' where n' < length ias' ias' ! n' = IsInterrupted t' True
  and Interrupt t' ∉ set (take n' ias')
proof(atomize-elim)
  from assms show ∃ n'<length ias'. ias' ! n' = IsInterrupted t' True ∧ Interrupt t' ∉ set (take n'
ias')
    unfolding collect-interrupts-def
  proof(induct ias' arbitrary: t')
    case Nil thus ?case by simp
  next
    case (Cons ia ias) thus ?case
      by(cases (ia, foldr collect-interrupt ias {}) rule: collect-interrupt.cases) fastforce+
  qed
qed

lemma collect-interrupts-prefix:
  collect-interrupts ias ⊆ collect-interrupts (ias @ ias')
by (metis Un-empty collect-interrupts-def foldr-append foldr-collect-interrupt-subset inf-sup-ord(1) inf-sup-ord(2)
subset-Un-eq)

lemma redT-updI-insert-Interrupt:
  ⟦ t ∈ redT-updI is ia; t ∉ is ⟧ ⟹ ia = Interrupt t
by(cases ia) simp-all

lemma redT-updIs-insert-Interrupt:
```

```
  ⟦ t ∈ redT-updIs is ias; t ∉ is ⟧ ⟹ Interrupt t ∈ set ias
proof(induct ias arbitrary: is)
  case Nil thus ?case by simp
next
  case (Cons ia ias) thus ?case
    by(cases t ∈ redT-updI is ia)(auto dest: redT-updI-insert-Interrupt)
qed

lemma interrupt-actions-ok-takeI:
  interrupt-actions-ok is ias ⟹ interrupt-actions-ok is (take n ias)
by(subst (asm) append-take-drop-id[symmetric, where n=n])(simp del: append-take-drop-id)

lemma interrupt-actions-ok'-collect-interrupts-imp-interrupt-actions-ok:
  assumes int: interrupt-actions-ok' is ias
  and ci: collect-interrupts ias ⊆ is
  and int': interrupt-actions-ok is' ias
  shows interrupt-actions-ok is ias
proof(rule interrupt-actions-ok-all-nthI)
  fix n
  assume n: n < length ias
  show interrupt-action-ok (redT-updIs is (take n ias)) (ias ! n)
  proof(cases ∃ t. ias ! n = IsInterrupted t True)
    case False
    with interrupt-actions-ok'-nthD[OF int n] show ?thesis by(cases ias ! n) simp-all
  next
    case True
    then obtain t where ia: ias ! n = IsInterrupted t True ..
      from int' n have interrupt-action-ok (redT-updIs is' (take n ias)) (ias ! n) by(rule inter-
rupt-actions-ok-nthD)
    with ia have t ∈ redT-updIs is' (take n ias) by simp
    moreover have ias = take (Suc n) ias @ drop (Suc n) ias by simp
    with ci have collect-interrupts (take (Suc n) ias) ⊆ is
      by (metis collect-interrupts-prefix subset-trans)
    ultimately have t ∈ redT-updIs is (take n ias) using n ia int int'
    proof(induct n arbitrary: is is' ias)
      case 0 thus ?case by(clarsimp simp add: neq-Nil-conv collect-interrupts-def)
    next
      case (Suc n)
      from ‹Suc n < length ias› obtain ia ias'
        where ias [simp]: ias = ia # ias' by(cases ias) auto
      from ‹interrupt-actions-ok is' ias›
      have ia-ok: interrupt-action-ok is' ia by simp

      from ‹t ∈ redT-updIs is' (take (Suc n) ias)›
      have t ∈ redT-updIs (redT-updI is' ia) (take n ias') by simp
      moreover from ‹collect-interrupts (take (Suc (Suc n)) ias) ⊆ is› ia-ok
      have collect-interrupts (take (Suc n) ias') ⊆ redT-updI is ia
      proof(cases (ia, is) rule: collect-interrupt.cases)
        case (3-2 t' Ts)
        hence [simp]: ia = ClearInterrupt t' Ts = is by simp-all
        have t' ∉ collect-interrupts (take (Suc n) ias')
        proof
          assume t' ∈ collect-interrupts (take (Suc n) ias')
          then obtain n' where n' < length (take (Suc n) ias') take (Suc n) ias' ! n' = IsInterrupted
```

t′ True
Interrupt t′ ∉ set (take n′ (take (Suc n) ias′)) **by**(*rule collect-interruptsE*)
hence *n′ ≤ n ias′ ! n′ = IsInterrupted t′ True Interrupt t′ ∉ set (take n′ ias′)*
using ‹*Suc n < length ias*› **by**(*simp-all add*: *min-def split*: *if-split-asm*)
hence *Suc n′ < length ias* **using** ‹*Suc n < length ias*› **by**(*simp add*: *min-def*)
with ‹*interrupt-actions-ok is′ ias*›
have *interrupt-action-ok (redT-updIs is′ (take (Suc n′) ias)) (ias ! Suc n′)*
by(*rule interrupt-actions-ok-nthD*)
with ‹*Suc n < length ias*› ‹*ias′ ! n′ = IsInterrupted t′ True*›
have *t′ ∈ redT-updIs (is′ − {t′}) (take n′ ias′)* **by** *simp*
hence *Interrupt t′ ∈ set (take n′ ias′)*
by(*rule redT-updIs-insert-Interrupt*) *simp*
with ‹*Interrupt t′ ∉ set (take n′ ias′)*› **show** *False* **by** *contradiction*
qed
thus *?thesis* **using** ‹*collect-interrupts (take (Suc (Suc n)) ias) ⊆ is*›
by(*auto simp add*: *collect-interrupts-def*)
qed(*auto simp add*: *collect-interrupts-def*)
moreover from ‹*Suc n < length ias*› **have** *n < length ias′* **by** *simp*
moreover from ‹*ias ! Suc n = IsInterrupted t True*› **have** *ias′ ! n = IsInterrupted t True* **by** *simp*
moreover from ‹*interrupt-actions-ok′ is ias*› **have** *interrupt-actions-ok′ (redT-updI is ia) ias′*
unfolding *ias* **by** *simp*
moreover from ‹*interrupt-actions-ok is′ ias*› **have** *interrupt-actions-ok (redT-updI is′ ia) ias′* **by** *simp*
ultimately have *t ∈ redT-updIs (redT-updI is ia) (take n ias′)* **by**(*rule Suc*)
thus *?case* **by** *simp*
qed
thus *?thesis* **unfolding** *ia* **by** *simp*
qed
qed

end

1.10 The multithreaded semantics

theory *FWSemantics*
imports
FWWellform
FWLockingThread
FWCondAction
FWInterrupt
begin

inductive *redT-upd* :: *('l,'t,'x,'m,'w) state ⇒ 't ⇒ ('l,'t,'x,'m,'w,'o) thread-action ⇒ 'x ⇒ 'm ⇒ ('l,'t,'x,'m,'w) state ⇒ bool*
for *s t ta x′ m′*
where
redT-updWs t (wset s) {|ta|}$_w$ ws′
⟹ redT-upd s t ta x′ m′ (redT-updLs (locks s) t {|ta|}$_l$, ((redT-updTs (thr s) {|ta|}$_t$)(t ↦ (x′, redT-updLns (locks s) t (snd (the (thr s t))) {|ta|}$_l$)), m′), ws′, redT-updIs (interrupts s) {|ta|}$_i$)

inductive-simps *redT-upd-simps* [*simp*]:
redT-upd s t ta x′ m′ s′

definition *redT-acq* :: ('l,'t,'x,'m,'w) *state* ⇒ 't ⇒ ('l ⇒*f* *nat*) ⇒ ('l,'t,'x,'m,'w) *state*
where
⋀*ln*. *redT-acq s t ln* = (*acquire-all* (*locks s*) *t ln*, ((*thr s*)(*t* ↦ (*fst* (*the* (*thr s t*)), *no-wait-locks*)), *shr s*), *wset s*, *interrupts s*)

context *final-thread* **begin**

inductive *actions-ok* :: ('l,'t,'x,'m,'w) *state* ⇒ 't ⇒ ('l,'t,'x',m,'w,'o) *thread-action* ⇒ *bool*
for *s* :: ('l,'t,'x,'m,'w) *state* **and** *t* :: 't **and** *ta* :: ('l,'t,'x',m,'w,'o) *thread-action*
where
⟦ *lock-ok-las* (*locks s*) *t* ⦃*ta*⦄*l*; *thread-oks* (*thr s*) ⦃*ta*⦄*t*; *cond-action-oks s t* ⦃*ta*⦄*c*;
wset-actions-ok (*wset s*) *t* ⦃*ta*⦄*w*; *interrupt-actions-ok* (*interrupts s*) ⦃*ta*⦄*i* ⟧
⟹ *actions-ok s t ta*

declare *actions-ok.intros* [*intro*!]
declare *actions-ok.cases* [*elim*!]

lemma *actions-ok-iff* [*simp*]:
actions-ok s t ta ⟷
lock-ok-las (*locks s*) *t* ⦃*ta*⦄*l* ∧ *thread-oks* (*thr s*) ⦃*ta*⦄*t* ∧ *cond-action-oks s t* ⦃*ta*⦄*c* ∧
wset-actions-ok (*wset s*) *t* ⦃*ta*⦄*w* ∧ *interrupt-actions-ok* (*interrupts s*) ⦃*ta*⦄*i*
by(*auto*)

lemma *actions-ok-thread-oksD*:
actions-ok s t ta ⟹ *thread-oks* (*thr s*) ⦃*ta*⦄*t*
by(*erule actions-ok.cases*)

inductive *actions-ok'* :: ('l,'t,'x,'m,'w) *state* ⇒ 't ⇒ ('l,'t,'x',m,'w,'o) *thread-action* ⇒ *bool* **where**
⟦ *lock-ok-las'* (*locks s*) *t* ⦃*ta*⦄*l*; *thread-oks* (*thr s*) ⦃*ta*⦄*t*; *cond-action-oks' s t* ⦃*ta*⦄*c*;
wset-actions-ok (*wset s*) *t* ⦃*ta*⦄*w*; *interrupt-actions-ok'* (*interrupts s*) ⦃*ta*⦄*i* ⟧
⟹ *actions-ok' s t ta*

declare *actions-ok'.intros* [*intro*!]
declare *actions-ok'.cases* [*elim*!]

lemma *actions-ok'-iff*:
actions-ok' s t ta ⟷
lock-ok-las' (*locks s*) *t* ⦃*ta*⦄*l* ∧ *thread-oks* (*thr s*) ⦃*ta*⦄*t* ∧ *cond-action-oks' s t* ⦃*ta*⦄*c* ∧
wset-actions-ok (*wset s*) *t* ⦃*ta*⦄*w* ∧ *interrupt-actions-ok'* (*interrupts s*) ⦃*ta*⦄*i*
by *auto*

lemma *actions-ok'-ta-upd-obs*:
actions-ok' s t (*ta-update-obs ta obs*) ⟷ *actions-ok' s t ta*
by(*auto simp add*: *actions-ok'-iff lock-ok-las'-def ta-upd-simps wset-actions-ok-def*)

lemma *actions-ok'-empty*: *actions-ok' s t ε* ⟷ *wset s t* = *None*
by(*simp add*: *actions-ok'-iff lock-ok-las'-def*)

lemma *actions-ok'-convert-extTA*:
actions-ok' s t (*convert-extTA f ta*) = *actions-ok' s t ta*
by(*simp add*: *actions-ok'-iff*)

inductive *actions-subset* :: ('l,'t,'x,'m,'w,'o) *thread-action* ⇒ ('l,'t,'x',m,'w,'o) *thread-action* ⇒ *bool*

where
⟦ *collect-locks′* ⦃*ta′*⦄$_l$ ⊆ *collect-locks* ⦃*ta*⦄$_l$;
collect-cond-actions ⦃*ta′*⦄$_c$ ⊆ *collect-cond-actions* ⦃*ta*⦄$_c$;
collect-interrupts ⦃*ta′*⦄$_i$ ⊆ *collect-interrupts* ⦃*ta*⦄$_i$ ⟧
⟹ *actions-subset ta′ ta*

declare *actions-subset.intros* [*intro*!]
declare *actions-subset.cases* [*elim*!]

lemma *actions-subset-iff*:
actions-subset ta′ ta ⟷
collect-locks′ ⦃*ta′*⦄$_l$ ⊆ *collect-locks* ⦃*ta*⦄$_l$ ∧
collect-cond-actions ⦃*ta′*⦄$_c$ ⊆ *collect-cond-actions* ⦃*ta*⦄$_c$ ∧
collect-interrupts ⦃*ta′*⦄$_i$ ⊆ *collect-interrupts* ⦃*ta*⦄$_i$
by *auto*

lemma *actions-subset-refl* [*intro*]:
actions-subset ta ta
by(*auto intro*: *actions-subset.intros collect-locks′-subset-collect-locks del*: *subsetI*)

definition *final-thread* :: ('*l*,'*t*,'*x*,'*m*,'*w*) *state* ⇒ '*t* ⇒ *bool* **where**
⋀*ln*. *final-thread s t* ≡ (*case thr s t of None* ⇒ *False* | ⌊(*x*, *ln*)⌋ ⇒ *final x* ∧ *ln* = *no-wait-locks* ∧ *wset s t* = *None*)

definition *final-threads* :: ('*l*,'*t*,'*x*,'*m*,'*w*) *state* ⇒ '*t set*
where *final-threads s* ≡ {*t*. *final-thread s t*}

lemma [*iff*]: *t* ∈ *final-threads s* = *final-thread s t*
by (*simp add*: *final-threads-def*)

lemma [*pred-set-conv*]: *final-thread s* = (λ*t*. *t* ∈ *final-threads s*)
by *simp*

definition *mfinal* :: ('*l*,'*t*,'*x*,'*m*,'*w*) *state* ⇒ *bool*
where *mfinal s* ⟷ (∀ *t x ln*. *thr s t* = ⌊(*x*, *ln*)⌋ ⟶ *final x* ∧ *ln* = *no-wait-locks* ∧ *wset s t* = *None*)

lemma *final-threadI*:
⟦ *thr s t* = ⌊(*x*, *no-wait-locks*)⌋; *final x*; *wset s t* = *None* ⟧ ⟹ *final-thread s t*
by(*simp add*: *final-thread-def*)

lemma *final-threadE*:
assumes *final-thread s t*
obtains *x* **where** *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ *final x wset s t* = *None*
using *assms* **by**(*auto simp add*: *final-thread-def*)

lemma *mfinalI*:
(⋀*t x ln*. *thr s t* = ⌊(*x*, *ln*)⌋ ⟹ *final x* ∧ *ln* = *no-wait-locks* ∧ *wset s t* = *None*) ⟹ *mfinal s*
unfolding *mfinal-def* **by** *blast*

lemma *mfinalD*:
fixes *ln*
assumes *mfinal s thr s t* = ⌊(*x*, *ln*)⌋
shows *final x ln* = *no-wait-locks wset s t* = *None*
using *assms* **unfolding** *mfinal-def* **by** *blast*+

lemma *mfinalE*:
fixes *ln*
assumes *mfinal s thr s t* = ⌊(*x*, *ln*)⌋
obtains *final x ln* = *no-wait-locks wset s t* = *None*
using *mfinalD*[*OF assms*] **by**(*rule that*)

lemma *mfinal-def2*: *mfinal s* ⟷ *dom* (*thr s*) ⊆ *final-threads s*
by(*fastforce elim*: *mfinalE final-threadE intro*: *mfinalI final-threadI*)

end

locale *multithreaded-base* = *final-thread* +
constrains *final* :: *'x* ⇒ *bool*
fixes *r* :: (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *semantics* (‹- ⊢ - −-→ -› [*50*,*0*,*0*,*50*] *80*)
and *convert-RA* :: *'l released-locks* ⇒ *'o list*
begin

abbreviation
r-syntax :: *'t* ⇒ *'x* ⇒ *'m* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* ⇒ *'x* ⇒ *'m* ⇒ *bool*
(‹- ⊢ ⟨-, -⟩ −-→ ⟨-, -⟩› [*50*,*0*,*0*,*0*,*0*,*0*] *80*)
where
t ⊢ ⟨*x*, *m*⟩ −*ta*→ ⟨*x'*, *m'*⟩ ≡ *t* ⊢ (*x*, *m*) −*ta*→ (*x'*, *m'*)

inductive
redT :: (*'l*,*'t*,*'x*,*'m*,*'w*) *state* ⇒ *'t* × (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*) *state* ⇒ *bool*
and
redT-syntax1 :: (*'l*,*'t*,*'x*,*'m*,*'w*) *state* ⇒ *'t* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*) *state*
⇒ *bool* (‹- −-▷-→ -› [*50*,*0*,*0*,*50*] *80*)
where
s −*t*▷*ta*→ *s'* ≡ *redT s* (*t*, *ta*) *s'*

| *redT-normal*:
⟦ *t* ⊢ ⟨*x*, *shr s*⟩ −*ta*→ ⟨*x'*, *m'*⟩;
thr s t = ⌊(*x*, *no-wait-locks*)⌋;
actions-ok s t ta;
redT-upd s t ta x' m' s' ⟧
⟹ *s* −*t*▷*ta*→ *s'*

| *redT-acquire*:
⋀*ln*. ⟦ *thr s t* = ⌊(*x*, *ln*)⌋; ¬ *waiting* (*wset s t*);
may-acquire-all (*locks s*) *t ln*; *ln* $ *n* > *0*;
s' = (*acquire-all* (*locks s*) *t ln*, ((*thr s*)(*t* ↦ (*x*, *no-wait-locks*)), *shr s*), *wset s*, *interrupts s*) ⟧
⟹ *s* −*t*▷((*K*$ []), [], [], [], [], *convert-RA ln*)→ *s'*

abbreviation
redT-syntax2 :: (*'l*,*'t*) *locks* ⇒ (*'l*,*'t*,*'x*) *thread-info* × *'m* ⇒ (*'w*,*'t*) *wait-sets* ⇒ *'t interrupts*
⇒ *'t* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action*
⇒ (*'l*,*'t*) *locks* ⇒ (*'l*,*'t*,*'x*) *thread-info* × *'m* ⇒ (*'w*,*'t*) *wait-sets* ⇒ *'t interrupts* ⇒ *bool*
(‹⟨-, -, -, -⟩ −-▷-→ ⟨-, -, -, -⟩› [*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*] *80*)
where
⟨*ls*, *tsm*, *ws*, *is*⟩ −*t*▷*ta*→ ⟨*ls'*, *tsm'*, *ws'*, *is'*⟩ ≡ (*ls*, *tsm*, *ws*, *is*) −*t*▷*ta*→ (*ls'*, *tsm'*, *ws'*, *is'*)

lemma *redT-elims* [*consumes 1*, *case-names normal acquire*]:
assumes *red*: $s\ -t\triangleright ta\rightarrow\ s'$
and *normal*: $\bigwedge x\ x'\ m'\ ws'.$
$[\![\ t \vdash \langle x,\ shr\ s\rangle\ -ta\rightarrow\ \langle x',\ m'\rangle;$
thr s t = $\lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor;$
lock-ok-las (*locks s*) *t* $\{\!|ta|\!\}_l$;
thread-oks (*thr s*) $\{\!|ta|\!\}_t$;
cond-action-oks s t $\{\!|ta|\!\}_c$;
wset-actions-ok (*wset s*) *t* $\{\!|ta|\!\}_w$;
interrupt-actions-ok (*interrupts s*) $\{\!|ta|\!\}_i$;
redT-updWs t (*wset s*) $\{\!|ta|\!\}_w$ *ws'*;
s' = (*redT-updLs* (*locks s*) *t* $\{\!|ta|\!\}_l$, ((*redT-updTs* (*thr s*) $\{\!|ta|\!\}_t$)($t \mapsto (x',$ *redT-updLns* (*locks s*) *t no-wait-locks* $\{\!|ta|\!\}_l$)), *m'*), *ws'*, *redT-updIs* (*interrupts s*) $\{\!|ta|\!\}_i$) $]\!]$
$\Longrightarrow$ *thesis*
and *acquire*: $\bigwedge x\ ln\ n.$
$[\![$ *thr s t* = $\lfloor(x,\ ln)\rfloor;$
ta = (*K$* [], [], [], [], [], *convert-RA ln*);
$\neg$ *waiting* (*wset s t*);
may-acquire-all (*locks s*) *t ln*; *0* < *ln* $ *n*;
s' = (*acquire-all* (*locks s*) *t ln*, ((*thr s*)($t \mapsto (x,$ *no-wait-locks*)), *shr s*), *wset s*, *interrupts s*) $]\!]$
$\Longrightarrow$ *thesis*
shows *thesis*
using *red*
proof *cases*
case *redT-normal*
thus *?thesis* **using** *normal* **by**(*cases s'*)(*auto*)
next
case *redT-acquire*
thus *?thesis* **by**−(*rule acquire*, *fastforce*+)
qed

definition
RedT :: ('*l*,'*t*,'*x*,'*m*,'*w*) *state* ⇒ ('*t* × ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action*) *list* ⇒ ('*l*,'*t*,'*x*,'*m*,'*w*) *state* ⇒ *bool*
(‹- −▷-→* -› [*50*,*0*,*50*] *80*)
where
RedT ≡ *rtrancl3p redT*

lemma *RedTI*:
rtrancl3p redT s ttas s' $\Longrightarrow$ *RedT s ttas s'*
by(*simp add*: *RedT-def*)

lemma *RedTE*:
$[\![$ *RedT s ttas s'*; *rtrancl3p redT s ttas s'* $\Longrightarrow$ *P* $]\!] \Longrightarrow$ *P*
by(*auto simp add*: *RedT-def*)

lemma *RedTD*:
RedT s ttas s' $\Longrightarrow$ *rtrancl3p redT s ttas s'*
by(*simp add*: *RedT-def*)

lemma *RedT-induct* [*consumes 1*, *case-names refl step*]:
$[\![\ s\ -\triangleright ttas\rightarrow *\ s';$
$\bigwedge s.\ P\ s\ []\ s;$
$\bigwedge s\ ttas\ s'\ t\ ta\ s''.\ [\![\ s\ -\triangleright ttas\rightarrow *\ s';\ P\ s\ ttas\ s';\ s'\ -t\triangleright ta\rightarrow\ s''\ ]\!] \Longrightarrow P\ s\ (ttas\ @\ [(t,\ ta)])\ s'']\!]$

$\Longrightarrow P\ s\ ttas\ s'$
unfolding *RedT-def*
by(*erule rtrancl3p.induct*) *auto*

lemma *RedT-induct'* [*consumes 1*, *case-names refl step*]:
$[\![\ s\ -\triangleright ttas\rightarrow *\ s';$
$P\ s\ []\ s;$
$\bigwedge ttas\ s'\ t\ ta\ s''.\ [\![\ s\ -\triangleright ttas\rightarrow *\ s';\ P\ s\ ttas\ s';\ s'\ -t\triangleright ta\rightarrow\ s''\ ]\!] \Longrightarrow P\ s\ (ttas\ @\ [(t,\ ta)])\ s'']\!]$
$\Longrightarrow P\ s\ ttas\ s'$
unfolding *RedT-def*
apply(*erule rtrancl3p-induct'*, *blast*)
apply(*case-tac b*, *blast*)
done

lemma *RedT-lift-preserveD*:
assumes *Red*: $s\ -\triangleright ttas\rightarrow *\ s'$
and *P*: $P\ s$
and *preserve*: $\bigwedge s\ t\ tas\ s'.\ [\![\ s\ -t\triangleright tas\rightarrow\ s';\ P\ s\ ]\!] \Longrightarrow P\ s'$
shows $P\ s'$
using *Red P*
by(*induct rule*: *RedT-induct*)(*auto intro*: *preserve*)

lemma *RedT-refl* [*intro*, *simp*]:
$s\ -\triangleright[]\rightarrow *\ s$
by(*rule RedTI*)(*rule rtrancl3p-refl*)

lemma *redT-has-locks-inv*:
$[\![\ \langle ls,\ (ts,\ m),\ ws,\ is\rangle\ -t\triangleright ta\rightarrow\ \langle ls',\ (ts',\ m'),\ ws',\ is'\rangle;\ t \neq t'\ ]\!] \Longrightarrow$
$has\text{-}locks\ (ls\ \$\ l)\ t' = has\text{-}locks\ (ls'\ \$\ l)\ t'$
by(*auto elim*!: *redT.cases intro*: *redT-updLs-has-locks*[*THEN sym*, *simplified*] *may-acquire-all-has-locks-acquire-lock*

lemma *redT-has-lock-inv*:
$[\![\ \langle ls,\ (ts,\ m),\ ws,\ is\rangle\ -t\triangleright ta\rightarrow\ \langle ls',\ (ts',\ m'),\ ws',\ is'\rangle;\ t \neq t'\ ]\!]$
$\Longrightarrow has\text{-}lock\ (ls'\ \$\ l)\ t' = has\text{-}lock\ (ls\ \$\ l)\ t'$
by(*auto simp add*: *redT-has-locks-inv*)

lemma *redT-ts-Some-inv*:
$[\![\ \langle ls,\ (ts,\ m),\ ws,\ is\rangle\ -t\triangleright ta\rightarrow\ \langle ls',\ (ts',\ m'),\ ws',\ is'\rangle;\ t \neq t';\ ts\ t' = \lfloor x\rfloor\ ]\!] \Longrightarrow ts'\ t' = \lfloor x\rfloor$
by(*fastforce elim*!: *redT.cases simp*: *redT-updTs-upd*[*THEN sym*] *intro*: *redT-updTs-Some*)

lemma *redT-thread-not-disappear*:
$[\![\ s\ -t\triangleright ta\rightarrow\ s';\ thr\ s'\ t' = None]\!] \Longrightarrow thr\ s\ t' = None$
apply(*cases* $t \neq t'$)
apply(*auto elim*!: *redT-elims simp add*: *redT-updTs-upd*[*THEN sym*] *intro*: *redT-updTs-None*)
done

lemma *RedT-thread-not-disappear*:
$[\![\ s\ -\triangleright ttas\rightarrow *\ s';\ thr\ s'\ t' = None]\!] \Longrightarrow thr\ s\ t' = None$
apply(*erule contrapos-pp*[**where** Q=$thr\ s'\ t' = None$])
apply(*drule* (*1*) *RedT-lift-preserveD*)
apply(*erule-tac* Q=$thr\ sa\ t' = None$ **in** *contrapos-nn*)
apply(*erule redT-thread-not-disappear*)
apply(*auto*)
done

lemma *redT-preserves-wset-thread-ok*:
⟦ s $-t▷ta→$ s'; *wset-thread-ok* (*wset* s) (*thr* s) ⟧ ⟹ *wset-thread-ok* (*wset* s') (*thr* s')
by(*fastforce elim*!: *redT.cases intro*: *wset-thread-ok-upd redT-updTs-preserves-wset-thread-ok redT-updWs-preserve-wset-thread-*

lemma *RedT-preserves-wset-thread-ok*:
⟦ s $-▷ttas→*$ s'; *wset-thread-ok* (*wset* s) (*thr* s) ⟧ ⟹ *wset-thread-ok* (*wset* s') (*thr* s')
by(*erule* (*1*) *RedT-lift-preserveD*)(*erule redT-preserves-wset-thread-ok*)

lemma *redT-new-thread-ts-Some*:
⟦ s $-t▷ta→$ s'; *NewThread* t' x m'' ∈ *set* ⦃*ta*⦄$_t$; *wset-thread-ok* (*wset* s) (*thr* s) ⟧
⟹ *thr* s' t' = ⌊(x, *no-wait-locks*)⌋
by(*erule redT-elims*)(*auto dest*: *thread-oks-new-thread elim*: *redT-updTs-new-thread-ts*)

lemma *RedT-new-thread-ts-not-None*:
⟦ s $-▷ttas→*$ s'; *NewThread* t x m'' ∈ *set* (*concat* (*map* (*thr-a* ∘ *snd*) *ttas*)); *wset-thread-ok* (*wset* s)
(*thr* s) ⟧
⟹ *thr* s' t ≠ *None*
proof(*induct rule*: *RedT-induct*)
case *refl* **thus** *?case* **by** *simp*
next
case (*step* S *TTAS* S' T *TA* S'')
note *Red* = ‹S $-▷TTAS→*$ S'›
note *IH* = ‹⟦ *NewThread* t x m'' ∈ *set* (*concat* (*map* (*thr-a* ∘ *snd*) *TTAS*)); *wset-thread-ok* (*wset* S)
(*thr* S) ⟧ ⟹ *thr* S' t ≠ *None*›
note *red* = ‹S' $-T▷TA→$ S''›
note *ins* = ‹*NewThread* t x m'' ∈ *set* (*concat* (*map* (*thr-a* ∘ *snd*) (*TTAS* @ [(T, *TA*)])))›
note *wto* = ‹*wset-thread-ok* (*wset* S) (*thr* S)›
from *Red wto* **have** *wto'*: *wset-thread-ok* (*wset* S') (*thr* S') **by**(*auto dest*: *RedT-preserves-wset-thread-ok*)

show *?case*
proof(*cases NewThread* t x m'' ∈ *set* ⦃*TA*⦄$_t$)
case *True* **thus** *?thesis* **using** *red wto'*
by(*auto dest*!: *redT-new-thread-ts-Some*)
next
case *False*
hence *NewThread* t x m'' ∈ *set* (*concat* (*map* (*thr-a* ∘ *snd*) *TTAS*)) **using** *ins* **by**(*auto*)
hence *thr* S' t ≠ *None* **using** *wto* **by**(*rule IH*)
with *red* **show** *?thesis*
by −(*erule contrapos-nn*, *auto dest*: *redT-thread-not-disappear*)
qed
qed

lemma *redT-preserves-lock-thread-ok*:
⟦ s $-t▷ta→$ s'; *lock-thread-ok* (*locks* s) (*thr* s) ⟧ ⟹ *lock-thread-ok* (*locks* s') (*thr* s')
by(*auto elim*!: *redT-elims intro*: *redT-upds-preserves-lock-thread-ok acquire-all-preserves-lock-thread-ok*)

lemma *RedT-preserves-lock-thread-ok*:
⟦ s $-▷ttas→*$ s'; *lock-thread-ok* (*locks* s) (*thr* s) ⟧ ⟹ *lock-thread-ok* (*locks* s') (*thr* s')
by(*erule* (*1*) *RedT-lift-preserveD*)(*erule redT-preserves-lock-thread-ok*)

lemma *redT-ex-new-thread*:
assumes s $-t▷ta→$ s' *wset-thread-ok* (*wset* s) (*thr* s) *thr* s' t = ⌊(x, w)⌋ *thr* s t = *None*
shows ∃ m. *NewThread* t x m ∈ *set* ⦃*ta*⦄$_t$ ∧ w = *no-wait-locks*

using *assms*
by *cases* (*fastforce split*: *if-split-asm dest*: *wset-thread-okD redT-updTs-new-thread*)+

lemma *redT-ex-new-thread'*:
  assumes $s\ -t'\triangleright ta\rightarrow\ s'\ thr\ s'\ t = \lfloor(x,\ w)\rfloor\ thr\ s\ t = None$
  shows $\exists\, m\ x.\ NewThread\ t\ x\ m \in set\ \{\!|ta|\!\}_t$
using *assms*
by(*cases*)(*fastforce split*: *if-split-asm dest*!: *redT-updTs-new-thread*)+

definition *deterministic* :: $('l,'t,'x,'m,'w)\ state\ set \Rightarrow bool$
where
  $deterministic\ I \longleftrightarrow$
  $(\forall\, s\ t\ x\ ta'\ x'\ m'\ ta''\ x''\ m''.$
    $s \in I$
    $\longrightarrow thr\ s\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor$
    $\longrightarrow t \vdash \langle x,\ shr\ s\rangle\ -ta'\rightarrow \langle x',\ m'\rangle$
    $\longrightarrow t \vdash \langle x,\ shr\ s\rangle\ -ta''\rightarrow \langle x'',\ m''\rangle$
    $\longrightarrow actions\text{-}ok\ s\ t\ ta' \longrightarrow actions\text{-}ok\ s\ t\ ta''$
    $\longrightarrow ta' = ta'' \wedge x' = x'' \wedge m' = m'')\ \wedge\ invariant3p\ redT\ I$

lemma *deterministicI*:
  $[\![\bigwedge s\ t\ x\ ta'\ x'\ m'\ ta''\ x''\ m''.$
      $[\![\ s \in I;\ thr\ s\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor;$
        $t \vdash \langle x,\ shr\ s\rangle\ -ta'\rightarrow \langle x',\ m'\rangle;\ t \vdash \langle x,\ shr\ s\rangle\ -ta''\rightarrow \langle x'',\ m''\rangle;$
        $actions\text{-}ok\ s\ t\ ta';\ actions\text{-}ok\ s\ t\ ta''\ ]\!]$
      $\Longrightarrow ta' = ta'' \wedge x' = x'' \wedge m' = m'';$
    $invariant3p\ redT\ I\ ]\!]$
  $\Longrightarrow deterministic\ I$
unfolding *deterministic-def* **by** *blast*

lemma *deterministicD*:
  $[\![\ deterministic\ I;$
    $t \vdash \langle x,\ shr\ s\rangle\ -ta'\rightarrow \langle x',\ m'\rangle;\ t \vdash \langle x,\ shr\ s\rangle\ -ta''\rightarrow \langle x'',\ m''\rangle;$
    $thr\ s\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor;\ actions\text{-}ok\ s\ t\ ta';\ actions\text{-}ok\ s\ t\ ta'';\ s \in I\ ]\!]$
  $\Longrightarrow ta' = ta'' \wedge x' = x'' \wedge m' = m''$
unfolding *deterministic-def* **by** *blast*

lemma *deterministic-invariant3p*:
  $deterministic\ I \Longrightarrow invariant3p\ redT\ I$
unfolding *deterministic-def* **by** *blast*

lemma *deterministic-THE*:
  $[\![\ deterministic\ I;\ thr\ s\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor;\ t \vdash \langle x,\ shr\ s\rangle\ -ta\rightarrow \langle x',\ m'\rangle;\ actions\text{-}ok\ s\ t\ ta;\ s \in I\ ]\!]$
  $\Longrightarrow (THE\ (ta,\ x',\ m').\ t \vdash \langle x,\ shr\ s\rangle\ -ta\rightarrow \langle x',\ m'\rangle \wedge actions\text{-}ok\ s\ t\ ta) = (ta,\ x',\ m')$
by(*rule the-equality*)(*blast dest*: *deterministicD*)+

end

locale *multithreaded* = *multithreaded-base* +
  constrains *final* :: $'x \Rightarrow bool$
  and *r* :: $('l,'t,'x,'m,'w,'o)\ semantics$
  and *convert-RA* :: $'l\ released\text{-}locks \Rightarrow 'o\ list$
  assumes *new-thread-memory*: $[\![\ t \vdash s\ -ta\rightarrow\ s';\ NewThread\ t'\ x\ m \in set\ \{\!|ta|\!\}_t\ ]\!] \Longrightarrow m = snd\ s'$

and *final-no-red*: ⟦ $t \vdash (x, m) -ta\rightarrow (x', m')$; *final* x ⟧ ⟹ *False*
begin

lemma *redT-new-thread-common*:
⟦ $s -t▷ta\rightarrow s'$; *NewThread* $t' x m'' \in$ *set* $\{\!|ta|\!\}_t$; $\{\!|ta|\!\}_w = []$ ⟧ ⟹ $m'' = shr\ s'$
by(*auto elim*!: *redT-elims rtrancl3p-cases dest*: *new-thread-memory*)

lemma *redT-new-thread*:
assumes $s -t'▷ta\rightarrow s'$ *thr* $s'\ t = \lfloor(x, w)\rfloor$ *thr* $s\ t =$ *None* $\{\!|ta|\!\}_w = []$
shows *NewThread* $t\ x\ (shr\ s') \in$ *set* $\{\!|ta|\!\}_t \wedge w =$ *no-wait-locks*
using *assms*
apply(*cases rule*: *redT-elims*)
apply(*auto split*: *if-split-asm del*: *conjI elim*!: *rtrancl3p-cases*)
apply(*drule* (*2*) *redT-updTs-new-thread*)
apply(*auto dest*: *new-thread-memory*)
done

lemma *final-no-redT*:
⟦ $s -t▷ta\rightarrow s'$; *thr* $s\ t = \lfloor(x,$ *no-wait-locks*$)\rfloor$ ⟧ ⟹ ¬ *final* x
by(*auto elim*!: *redT-elims dest*: *final-no-red*)

lemma *mfinal-no-redT*:
assumes *redT*: $s -t▷ta\rightarrow s'$ **and** *mfinal*: *mfinal* s
shows *False*
using *redT mfinalD*[*OF mfinal*, *of t*]
by *cases* (*metis final-no-red*, *metis neq-no-wait-locks-conv*)

end

end

1.11 Auxiliary definitions for the progress theorem for the multithreaded semantics

theory *FWProgressAux*
imports
FWSemantics
begin

abbreviation *collect-waits* :: ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* ⇒ ('*l* + '*t* + '*t*) *set*
where *collect-waits ta* ≡ *collect-locks* $\{\!|ta|\!\}_l$ <+> *collect-cond-actions* $\{\!|ta|\!\}_c$ <+> *collect-interrupts* $\{\!|ta|\!\}_i$

lemma *collect-waits-unfold*:
collect-waits ta = {*l*. *Lock* ∈ *set* ($\{\!|ta|\!\}_l$ $ *l*)} <+> {*t*. *Join t* ∈ *set* $\{\!|ta|\!\}_c$} <+> *collect-interrupts* $\{\!|ta|\!\}_i$
by(*simp add*: *collect-locks-def*)

context *multithreaded-base* **begin**

definition *must-sync* :: '*t* ⇒ '*x* ⇒ '*m* ⇒ *bool* (‹- ⊢ ⟨-,/ -⟩/ ≀› [*50*, *0*,*0*] *81*) **where**
$t \vdash \langle x, m\rangle$ ≀ ⟷ (∃ *ta x' m' s*. $t \vdash \langle x, m\rangle -ta\rightarrow \langle x', m'\rangle \wedge shr\ s = m \wedge$ *actions-ok s t ta*)

lemma *must-sync-def2*:
$t \vdash \langle x,\ m\rangle \wr \longleftrightarrow (\exists\, ta\ x'\ m'.\ t \vdash \langle x,\ m\rangle\ -ta\rightarrow\ \langle x',\ m'\rangle \wedge \mathit{actions\text{-}ok}\ s\ t\ ta)$
by(*fastforce simp add*: *must-sync-def intro*: *cond-action-oks-shr-change*)

lemma *must-syncI*:
$\exists\, ta\ x'\ m'.\ t \vdash \langle x,\ m\rangle\ -ta\rightarrow\ \langle x',\ m'\rangle \wedge \mathit{actions\text{-}ok}\ s\ t\ ta \Longrightarrow t \vdash \langle x,\ m\rangle \wr$
by(*fastforce simp add*: *must-sync-def2*)

lemma *must-syncE*:
$[\![\ t \vdash \langle x,\ m\rangle \wr;\ \bigwedge ta\ x'\ m'.\ [\![\ t \vdash \langle x,\ m\rangle\ -ta\rightarrow\ \langle x',\ m'\rangle;\ \mathit{actions\text{-}ok}\ s\ t\ ta;\ m = \mathit{shr}\ s\ ]\!] \Longrightarrow \mathit{thesis}\ ]\!]$
$\Longrightarrow \mathit{thesis}$
by(*fastforce simp only*: *must-sync-def*)

definition *can-sync* :: $'t \Rightarrow 'x \Rightarrow 'm \Rightarrow ('l + 't + 't)\ \mathit{set} \Rightarrow \mathit{bool}$ ($\langle$- $\vdash$ $\langle$-,/ -$\rangle$/ -/ $\wr\rangle$ [50,0,0,0] 81)
where
$t \vdash \langle x,\ m\rangle\ LT \wr \equiv \exists\, ta\ x'\ m'.\ t \vdash \langle x,\ m\rangle\ -ta\rightarrow\ \langle x',\ m'\rangle \wedge (LT = \mathit{collect\text{-}waits}\ ta)$

lemma *can-syncI*:
$[\![\ t \vdash \langle x,\ m\rangle\ -ta\rightarrow\ \langle x',\ m'\rangle;$
$\quad LT = \mathit{collect\text{-}waits}\ ta\ ]\!]$
$\Longrightarrow t \vdash \langle x,\ m\rangle\ LT \wr$
by(*cases ta*)(*fastforce simp add*: *can-sync-def*)

lemma *can-syncE*:
assumes $t \vdash \langle x,\ m\rangle\ LT \wr$
obtains $ta\ x'\ m'$
where $t \vdash \langle x,\ m\rangle\ -ta\rightarrow\ \langle x',\ m'\rangle$
and $LT = \mathit{collect\text{-}waits}\ ta$
using *assms*
by(*clarsimp simp add*: *can-sync-def*)

inductive-set *active-threads* :: $('l,'t,'x,'m,'w)\ \mathit{state} \Rightarrow 't\ \mathit{set}$
for s :: $('l,'t,'x,'m,'w)\ \mathit{state}$
where
normal:
$\bigwedge ln.\ [\![\ \mathit{thr}\ s\ t = \mathit{Some}\ (x,\ ln);$
$\quad ln = \mathit{no\text{-}wait\text{-}locks};$
$\quad t \vdash (x,\ \mathit{shr}\ s)\ -ta\rightarrow\ x'm';$
$\quad \mathit{actions\text{-}ok}\ s\ t\ ta\ ]\!]$
$\Longrightarrow t \in \mathit{active\text{-}threads}\ s$
| *acquire*:
$\bigwedge ln.\ [\![\ \mathit{thr}\ s\ t = \mathit{Some}\ (x,\ ln);$
$\quad ln \neq \mathit{no\text{-}wait\text{-}locks};$
$\quad \neg\ \mathit{waiting}\ (\mathit{wset}\ s\ t);$
$\quad \mathit{may\text{-}acquire\text{-}all}\ (\mathit{locks}\ s)\ t\ ln\ ]\!]$
$\Longrightarrow t \in \mathit{active\text{-}threads}\ s$

lemma *active-threads-iff*:
$\mathit{active\text{-}threads}\ s =$
$\{t.\ \exists\, x\ ln.\ \mathit{thr}\ s\ t = \mathit{Some}\ (x,\ ln) \wedge$
$\qquad (\mathit{if}\ ln = \mathit{no\text{-}wait\text{-}locks}$
$\qquad\ \mathit{then}\ \exists\, ta\ x'\ m'.\ t \vdash (x,\ \mathit{shr}\ s)\ -ta\rightarrow\ (x',\ m') \wedge \mathit{actions\text{-}ok}\ s\ t\ ta$
$\qquad\ \mathit{else}\ \neg\ \mathit{waiting}\ (\mathit{wset}\ s\ t) \wedge \mathit{may\text{-}acquire\text{-}all}\ (\mathit{locks}\ s)\ t\ ln)\}$
apply(*auto elim*!: *active-threads.cases intro*: *active-threads.intros*)

apply *blast*
done

lemma *active-thread-ex-red*:
assumes $t \in$ *active-threads s*
shows $\exists$ *ta s'*. *s* $-t\triangleright ta\rightarrow$ *s'*
using *assms*
proof *cases*
case (*normal x ta x'm' ln*)
with *redT-updWs-total*[*of t wset s* $\{\!|ta|\!\}_w$]
show *?thesis*
by(*cases x'm'*)(*fastforce intro*!: *redT-normal simp del*: *split-paired-Ex*)
next
case *acquire* **thus** *?thesis*
by(*fastforce intro*: *redT-acquire simp del*: *split-paired-Ex simp add*: *neq-no-wait-locks-conv*)
qed

end

Well-formedness conditions for final

context *final-thread* **begin**

inductive *not-final-thread* :: ('*l*,'*t*,'*x*,'*m*,'*w*) *state* $\Rightarrow$ '*t* $\Rightarrow$ *bool*
for *s* :: ('*l*,'*t*,'*x*,'*m*,'*w*) *state* **and** *t* :: '*t* **where**
not-final-thread-final: $\bigwedge$*ln*. $[\![$ *thr s t* = $\lfloor$(*x*, *ln*)$\rfloor$; $\neg$ *final x* $]\!] \Longrightarrow$ *not-final-thread s t*
| *not-final-thread-wait-locks*: $\bigwedge$*ln*. $[\![$ *thr s t* = $\lfloor$(*x*, *ln*)$\rfloor$; *ln* $\neq$ *no-wait-locks* $]\!] \Longrightarrow$ *not-final-thread s t*
| *not-final-thread-wait-set*: $\bigwedge$*ln*. $[\![$ *thr s t* = $\lfloor$(*x*, *ln*)$\rfloor$; *wset s t* = $\lfloor w \rfloor$ $]\!] \Longrightarrow$ *not-final-thread s t*

declare *not-final-thread.cases* [*elim*]

lemmas *not-final-thread-cases* = *not-final-thread.cases* [*consumes 1*, *case-names final wait-locks wait-set*]

lemma *not-final-thread-cases2* [*consumes 2*, *case-names final wait-locks wait-set*]:
$\bigwedge$*ln*. $[\![$ *not-final-thread s t*; *thr s t* = $\lfloor$(*x*, *ln*)$\rfloor$;
$\neg$ *final x* $\Longrightarrow$ *thesis*; *ln* $\neq$ *no-wait-locks* $\Longrightarrow$ *thesis*; $\bigwedge$*w*. *wset s t* = $\lfloor w \rfloor \Longrightarrow$ *thesis* $]\!]$
$\Longrightarrow$ *thesis*
by(*auto*)

lemma *not-final-thread-iff*:
not-final-thread s t $\longleftrightarrow$ ($\exists$ *x ln*. *thr s t* = $\lfloor$(*x*, *ln*)$\rfloor$ $\wedge$ ($\neg$ *final x* $\vee$ *ln* $\neq$ *no-wait-locks* $\vee$ ($\exists$ *w*. *wset s t* = $\lfloor w \rfloor$)))
by(*auto intro*: *not-final-thread.intros*)

lemma *not-final-thread-conv*:
not-final-thread s t $\longleftrightarrow$ *thr s t* $\neq$ *None* $\wedge$ $\neg$ *final-thread s t*
by(*auto simp add*: *final-thread-def intro*: *not-final-thread.intros*)

lemma *not-final-thread-existsE*:
assumes *not-final-thread s t*
and $\bigwedge$*x ln*. *thr s t* = $\lfloor$(*x*, *ln*)$\rfloor \Longrightarrow$ *thesis*
shows *thesis*
using *assms* **by** *blast*

lemma *not-final-thread-final-thread-conv*:
thr s t ≠ *None* ⟹ ¬ *final-thread s t* ⟷ *not-final-thread s t*
by(*simp add*: *not-final-thread-iff final-thread-def*)

lemma *may-join-cond-action-oks*:
assumes ⋀*t'*. *Join t'* ∈ *set cas* ⟹ ¬ *not-final-thread s t'* ∧ *t* ≠ *t'*
shows *cond-action-oks s t cas*
using *assms*
proof (*induct cas*)
case *Nil* **thus** *?case* **by** *clarsimp*
next
case (*Cons ca cas*)
note *IH* = ‹⟦ ⋀*t'*. *Join t'* ∈ *set cas* ⟹ ¬ *not-final-thread s t'* ∧ *t* ≠ *t'* ⟧
⟹ *cond-action-oks s t cas*›
note *ass* = ‹⋀*t'*. *Join t'* ∈ *set* (*ca* # *cas*) ⟹ ¬ *not-final-thread s t'* ∧ *t* ≠ *t'*›
hence ⋀*t'*. *Join t'* ∈ *set cas* ⟹ ¬ *not-final-thread s t'* ∧ *t* ≠ *t'* **by** *simp*
hence *cond-action-oks s t cas* **by**(*rule IH*)
moreover have *cond-action-ok s t ca*
proof(*cases ca*)
case (*Join t'*)
with *ass* **have** ¬ *not-final-thread s t' t* ≠ *t'* **by** *auto*
thus *?thesis* **using** *Join* **by**(*auto simp add*: *not-final-thread-iff*)
next
case *Yield* **thus** *?thesis* **by** *simp*
qed
ultimately show *?case* **by** *simp*
qed

end

context *multithreaded* **begin**

lemma *red-not-final-thread*:
s −*t*▷*ta*→ *s'* ⟹ *not-final-thread s t*
by(*fastforce elim*: *redT.cases intro*: *not-final-thread.intros dest*: *final-no-red*)

lemma *redT-preserves-final-thread*:
⟦ *s* −*t'*▷*ta*→ *s'*; *final-thread s t* ⟧ ⟹ *final-thread s' t*
apply(*erule redT.cases*)
apply(*clarsimp simp add*: *final-thread-def*)
apply(*auto simp add*: *final-thread-def dest*: *redT-updTs-None redT-updTs-Some final-no-red intro*: *redT-updWs-Non*
done

end

context *multithreaded-base* **begin**

definition *wset-Suspend-ok* :: ('*l*,'*t*,'*x*,'*m*,'*w*) *state set* ⇒ ('*l*,'*t*,'*x*,'*m*,'*w*) *state set*
where
wset-Suspend-ok I =
{*s*. *s* ∈ *I* ∧
(∀ *t* ∈ *dom* (*wset s*). ∃ *s0*∈*I*. ∃ *s1*∈*I*. ∃ *ttas x x0 ta w' ln' ln''*. *s0* −*t*▷*ta*→ *s1* ∧ *s1* −▷*ttas*→* *s* ∧
thr s0 t = ⌊(*x0*, *no-wait-locks*)⌋ ∧ *t* ⊢ ⟨*x0*, *shr s0*⟩ −*ta*→ ⟨*x*, *shr s1*⟩ ∧ *Suspend w'* ∈ *set*
{|*ta*|}*w* ∧

actions-ok s0 t ta ∧ *thr s1 t* = ⌊(*x*, *ln*′)⌋ ∧ *thr s t* = ⌊(*x*, *ln*″)⌋)}

lemma *wset-Suspend-okI*:
⟦ *s* ∈ *I*;
⋀*t w*. *wset s t* = ⌊*w*⌋ ⟹ ∃ *s0*∈*I*. ∃ *s1*∈*I*. ∃ *ttas x x0 ta w*′ *ln*′ *ln*″. *s0* −*t*▷*ta*→ *s1* ∧ *s1* −▷*ttas*→∗ *s* ∧
thr s0 t = ⌊(*x0*, *no-wait-locks*)⌋ ∧ *t* ⊢ ⟨*x0*, *shr s0*⟩ −*ta*→ ⟨*x*, *shr s1*⟩ ∧ *Suspend w*′ ∈ *set* {|*ta*|}*w* ∧
actions-ok s0 t ta ∧ *thr s1 t* = ⌊(*x*, *ln*′)⌋ ∧ *thr s t* = ⌊(*x*, *ln*″)⌋ ⟧
⟹ *s* ∈ *wset-Suspend-ok I*
unfolding *wset-Suspend-ok-def* **by** *blast*

lemma *wset-Suspend-okD1*:
s ∈ *wset-Suspend-ok I* ⟹ *s* ∈ *I*
unfolding *wset-Suspend-ok-def* **by** *blast*

lemma *wset-Suspend-okD2*:
⟦ *s* ∈ *wset-Suspend-ok I*; *wset s t* = ⌊*w*⌋ ⟧
⟹ ∃ *s0*∈*I*. ∃ *s1*∈*I*. ∃ *ttas x x0 ta w*′ *ln*′ *ln*″. *s0* −*t*▷*ta*→ *s1* ∧ *s1* −▷*ttas*→∗ *s* ∧
thr s0 t = ⌊(*x0*, *no-wait-locks*)⌋ ∧ *t* ⊢ ⟨*x0*, *shr s0*⟩ −*ta*→ ⟨*x*, *shr s1*⟩ ∧ *Suspend w*′ ∈ *set* {|*ta*|}*w* ∧
actions-ok s0 t ta ∧ *thr s1 t* = ⌊(*x*, *ln*′)⌋ ∧ *thr s t* = ⌊(*x*, *ln*″)⌋
unfolding *wset-Suspend-ok-def* **by** *blast*

lemma *wset-Suspend-ok-imp-wset-thread-ok*:
s ∈ *wset-Suspend-ok I* ⟹ *wset-thread-ok* (*wset s*) (*thr s*)
apply(*rule wset-thread-okI*)
apply(*rule ccontr*)
apply(*auto dest*: *wset-Suspend-okD2*)
done

lemma *invariant3p-wset-Suspend-ok*:
assumes *I*: *invariant3p redT I*
shows *invariant3p redT* (*wset-Suspend-ok I*)
proof(*rule invariant3pI*)
fix *s tl s*′
assume *wso*: *s* ∈ *wset-Suspend-ok I*
and *redT s tl s*′
moreover obtain *t*′ *ta* **where** *tl*: *tl* = (*t*′, *ta*) **by**(*cases tl*)
ultimately have *red*: *s* −*t*′▷*ta*→ *s*′ **by** *simp*
moreover from *wso* **have** *s* ∈ *I* **by**(*rule wset-Suspend-okD1*)
ultimately have *s*′ ∈ *I* **by**(*rule invariant3pD*[*OF I*])
thus *s*′ ∈ *wset-Suspend-ok I*
proof(*rule wset-Suspend-okI*)
fix *t w*
assume *ws*′*t*: *wset s*′ *t* = ⌊*w*⌋
show ∃ *s0*∈*I*. ∃ *s1*∈*I*. ∃ *ttas x x0 ta w*′ *ln*′ *ln*″. *s0* −*t*▷*ta*→ *s1* ∧ *s1* −▷*ttas*→∗ *s*′ ∧
thr s0 t = ⌊(*x0*, *no-wait-locks*)⌋ ∧ *t* ⊢ ⟨*x0*, *shr s0*⟩ −*ta*→ ⟨*x*, *shr s1*⟩ ∧
Suspend w′ ∈ *set* {|*ta*|}*w* ∧ *actions-ok s0 t ta* ∧
thr s1 t = ⌊(*x*, *ln*′)⌋ ∧ *thr s*′ *t* = ⌊(*x*, *ln*″)⌋
proof(*cases t* = *t*′)
case *False*
with *red ws*′*t* **obtain** *w*′ **where** *wst*: *wset s t* = ⌊*w*′⌋
by *cases*(*auto 4 4 dest*: *redT-updWs-Some-otherD split*: *wait-set-status.split-asm*)

```
    from wset-Suspend-okD2[OF wso this] obtain s0 s1 ttas x x0 ta' w' ln' ln''
      where reuse: s0 ∈ I s1 ∈ I s0 −t▷ta'→ s1 thr s0 t = ⌊(x0, no-wait-locks)⌋
       t ⊢ ⟨x0, shr s0⟩ −ta'→ ⟨x, shr s1⟩ Suspend w' ∈ set {|ta'|}_w actions-ok s0 t ta' thr s1 t = ⌊(x,
ln')⌋
      and step: s1 −▷ttas→* s and tst: thr s t = ⌊(x, ln'')⌋ by blast
    from step red have s1 −▷ttas@[(t', ta)]→* s' unfolding RedT-def by(rule rtrancl3p-step)
    moreover from red tst False have thr s' t = ⌊(x, ln'')⌋
      by(cases)(auto intro: redT-updTs-Some)
    ultimately show ?thesis using reuse by blast
  next
    case True
    from red show ?thesis
    proof(cases)
      case (redT-normal x x' m)
      note red' = ‹t' ⊢ ⟨x, shr s⟩ −ta→ ⟨x', m⟩›
        and tst' = ‹thr s t' = ⌊(x, no-wait-locks)⌋›
        and aok = ‹actions-ok s t' ta›
        and s' = ‹redT-upd s t' ta x' m s'›
      from s' have ws': redT-updWs t' (wset s) {|ta|}_w (wset s')
        and m: m = shr s'
        and ts't: thr s' t' = ⌊(x', redT-updLns (locks s) t' (snd (the (thr s t'))) {|ta|}_l)⌋ by auto
      from aok have nwait: ¬ waiting (wset s t')
        by(auto simp add: wset-actions-ok-def waiting-def split: if-split-asm)
      have ∃ w'. Suspend w' ∈ set {|ta|}_w
      proof(cases wset s t)
        case None
        from redT-updWs-None-SomeD[OF ws', OF ws't None]
        show ?thesis ..
      next
        case (Some w')
        with True aok have Notified ∈ set {|ta|}_w ∨ WokenUp ∈ set {|ta|}_w
          by(auto simp add: wset-actions-ok-def split: if-split-asm)
        with ws' show ?thesis using ws't unfolding True
          by(rule redT-updWs-WokenUp-SuspendD)
      qed
      with tst' ts't aok ‹s ∈ I› ‹s' ∈ I› red red' show ?thesis
        unfolding True m by blast
    next
      case (redT-acquire x n ln)
      with ws't True have wset s t = ⌊w⌋ by auto
      from wset-Suspend-okD2[OF wso this] ‹thr s t' = ⌊(x, ln)⌋› True
      obtain s0 s1 ttas x0 ta' w' ln' ln''
        where reuse: s0 ∈ I s1 ∈ I s0 −t▷ta'→ s1 thr s0 t = ⌊(x0, no-wait-locks)⌋
          t ⊢ ⟨x0, shr s0⟩ −ta'→ ⟨x, shr s1⟩ Suspend w' ∈ set {|ta'|}_w actions-ok s0 t ta' thr s1 t =
⌊(x, ln')⌋
        and step: s1 −▷ttas→* s by fastforce
      from step red have s1 −▷ttas@[(t', ta)]→* s' unfolding RedT-def by(rule rtrancl3p-step)
      moreover from redT-acquire True have thr s' t = ⌊(x, no-wait-locks)⌋ by simp
      ultimately show ?thesis using reuse by blast
    qed
  qed
 qed
qed
```

end

end

1.12 Deadlock formalisation

theory *FWDeadlock*
imports
FWProgressAux
begin

context *final-thread* **begin**

definition *all-final-except* :: ('*l*,'*t*,'*x*,'*m*,'*w*) *state* ⇒ '*t set* ⇒ *bool* **where**
all-final-except s Ts ≡ ∀ *t*. *not-final-thread s t* ⟶ *t* ∈ *Ts*

lemma *all-final-except-mono* [*mono*]:
(⋀*x*. *x* ∈ *A* ⟶ *x* ∈ *B*) ⟹ *all-final-except ts A* ⟶ *all-final-except ts B*
by(*auto simp add*: *all-final-except-def*)

lemma *all-final-except-mono'*:
⟦ *all-final-except ts A*; ⋀*x*. *x* ∈ *A* ⟹ *x* ∈ *B* ⟧ ⟹ *all-final-except ts B*
by(*blast intro*: *all-final-except-mono*[*rule-format*])

lemma *all-final-exceptI*:
(⋀*t*. *not-final-thread s t* ⟹ *t* ∈ *Ts*) ⟹ *all-final-except s Ts*
by(*auto simp add*: *all-final-except-def*)

lemma *all-final-exceptD*:
⟦ *all-final-except s Ts*; *not-final-thread s t* ⟧ ⟹ *t* ∈ *Ts*
by(*auto simp add*: *all-final-except-def*)

inductive *must-wait* :: ('*l*,'*t*,'*x*,'*m*,'*w*) *state* ⇒ '*t* ⇒ ('*l* + '*t* + '*t*) ⇒ '*t set* ⇒ *bool*
for *s* :: ('*l*,'*t*,'*x*,'*m*,'*w*) *state* **and** *t* :: '*t* **where**
— Lock l
⟦ *has-lock* (*locks s* $ *l*) *t'*; *t'* ≠ *t*; *t'* ∈ *Ts* ⟧ ⟹ *must-wait s t* (*Inl l*) *Ts*
| — Join t'
⟦ *not-final-thread s t'*; *t'* ∈ *Ts* ⟧ ⟹ *must-wait s t* (*Inr* (*Inl t'*)) *Ts*
| — IsInterrupted t' True
⟦ *all-final-except s Ts*; *t'* ∉ *interrupts s* ⟧ ⟹ *must-wait s t* (*Inr* (*Inr t'*)) *Ts*

declare *must-wait.cases* [*elim*]
declare *must-wait.intros* [*intro*]

lemma *must-wait-elims* [*consumes 1*, *case-names lock join interrupt*, *cases pred*]:
assumes *must-wait s t lt Ts*
obtains *l t'* **where** *lt* = *Inl l has-lock* (*locks s* $ *l*) *t' t'* ≠ *t t'* ∈ *Ts*
| *t'* **where** *lt* = *Inr* (*Inl t'*) *not-final-thread s t' t'* ∈ *Ts*
| *t'* **where** *lt* = *Inr* (*Inr t'*) *all-final-except s Ts t'* ∉ *interrupts s*
using *assms*
by(*auto*)

inductive-cases *must-wait-elims2* [*elim*!]:
must-wait s t (Inl l) Ts
must-wait s t (Inr (Inl t′)) Ts
must-wait s t (Inr (Inr t′)) Ts

lemma *must-wait-iff*:
must-wait s t lt Ts ⟷
(*case lt of Inl l* ⇒ ∃ *t′*∈*Ts*. *t* ≠ *t′* ∧ *has-lock* (*locks s* $ *l*) *t′*
| *Inr* (*Inl t′*) ⇒ *not-final-thread s t′* ∧ *t′* ∈ *Ts*
| *Inr* (*Inr t′*) ⇒ *all-final-except s Ts* ∧ *t′* ∉ *interrupts s*)
by(*auto simp add*: *must-wait.simps split*: *sum.splits*)

end

Deadlock as a system-wide property

context *multithreaded-base* **begin**

definition
deadlock :: (*′l*,*′t*,*′x*,*′m*,*′w*) *state* ⇒ *bool*
where
deadlock s
≡ (∀ *t x*. *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ ∧ ¬ *final x* ∧ *wset s t* = *None*
⟶ *t* ⊢ ⟨*x*, *shr s*⟩ ≀ ∧ (∀ *LT*. *t* ⊢ ⟨*x*, *shr s*⟩ *LT* ≀ ⟶ (∃ *lt* ∈ *LT*. *must-wait s t lt* (*dom* (*thr s*)))))
∧ (∀ *t x ln*. *thr s t* = ⌊(*x*, *ln*)⌋ ∧ (∃ *l*. *ln* $ *l* > *0*) ∧ ¬ *waiting* (*wset s t*)
⟶ (∃ *l t′*. *ln* $ *l* > *0* ∧ *t* ≠ *t′* ∧ *thr s t′* ≠ *None* ∧ *has-lock* (*locks s* $ *l*) *t′*))
∧ (∀ *t x w*. *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ ⟶ *wset s t* ≠ ⌊*PostWS w*⌋)

lemma *deadlockI*:
⟦ ⋀*t x*. ⟦ *thr s t* = ⌊(*x*, *no-wait-locks*)⌋; ¬ *final x*; *wset s t* = *None* ⟧
⟹ *t* ⊢ ⟨*x*, *shr s*⟩ ≀ ∧ (∀ *LT*. *t* ⊢ ⟨*x*, *shr s*⟩ *LT* ≀ ⟶ (∃ *lt* ∈ *LT*. *must-wait s t lt* (*dom* (*thr s*))));
⋀*t x ln l*. ⟦ *thr s t* = ⌊(*x*, *ln*)⌋; *ln* $ *l* > *0*; ¬ *waiting* (*wset s t*) ⟧
⟹ ∃ *l t′*. *ln* $ *l* > *0* ∧ *t* ≠ *t′* ∧ *thr s t′* ≠ *None* ∧ *has-lock* (*locks s* $ *l*) *t′*;
⋀*t x w*. *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ ⟹ *wset s t* ≠ ⌊*PostWS w*⌋ ⟧
⟹ *deadlock s*
by(*auto simp add*: *deadlock-def*)

lemma *deadlockE*:
assumes *deadlock s*
obtains ∀ *t x*. *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ ∧ ¬ *final x* ∧ *wset s t* = *None*
⟶ *t* ⊢ ⟨*x*, *shr s*⟩ ≀ ∧ (∀ *LT*. *t* ⊢ ⟨*x*, *shr s*⟩ *LT* ≀ ⟶ (∃ *lt* ∈ *LT*. *must-wait s t lt* (*dom* (*thr s*))))
and ∀ *t x ln*. *thr s t* = ⌊(*x*, *ln*)⌋ ∧ (∃ *l*. *ln* $ *l* > *0*) ∧ ¬ *waiting* (*wset s t*)
⟶ (∃ *l t′*. *ln* $ *l* > *0* ∧ *t* ≠ *t′* ∧ *thr s t′* ≠ *None* ∧ *has-lock* (*locks s* $ *l*) *t′*)
and ∀ *t x w*. *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ ⟶ *wset s t* ≠ ⌊*PostWS w*⌋
using *assms* **unfolding** *deadlock-def* **by**(*blast*)

lemma *deadlockD1*:
assumes *deadlock s*
and *thr s t* = ⌊(*x*, *no-wait-locks*)⌋
and ¬ *final x*
and *wset s t* = *None*
obtains *t* ⊢ ⟨*x*, *shr s*⟩ ≀
and ∀ *LT*. *t* ⊢ ⟨*x*, *shr s*⟩ *LT* ≀ ⟶ (∃ *lt* ∈ *LT*. *must-wait s t lt* (*dom* (*thr s*)))
using *assms* **unfolding** *deadlock-def* **by**(*blast*)

lemma *deadlockD2*:
fixes *ln*
assumes *deadlock s*
and *thr s t* = ⌊(*x*, *ln*)⌋
and *ln* $ *l* > *0*
and ¬ *waiting* (*wset s t*)
obtains *l' t'* **where** *ln* $ *l'* > *0 t* ≠ *t' thr s t'* ≠ *None has-lock* (*locks s* $ *l'*) *t'*
using *assms* **unfolding** *deadlock-def* **by** *blast*

lemma *deadlockD3*:
assumes *deadlock s*
and *thr s t* = ⌊(*x*, *no-wait-locks*)⌋
shows ∀ *w*. *wset s t* ≠ ⌊*PostWS w*⌋
using *assms* **unfolding** *deadlock-def* **by** *blast*

lemma *deadlock-def2*:
deadlock s ⟷
(∀ *t x*. *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ ∧ ¬ *final x* ∧ *wset s t* = *None*
⟶ *t* ⊢ ⟨*x*, *shr s*⟩ ≀ ∧ (∀ *LT*. *t* ⊢ ⟨*x*, *shr s*⟩ *LT* ≀ ⟶ (∃ *lt* ∈ *LT*. *must-wait s t lt* (*dom* (*thr s*)))))
∧ (∀ *t x ln*. *thr s t* = ⌊(*x*, *ln*)⌋ ∧ *ln* ≠ *no-wait-locks* ∧ ¬ *waiting* (*wset s t*)
⟶ (∃ *l*. *ln* $ *l* > *0* ∧ *must-wait s t* (*Inl l*) (*dom* (*thr s*))))
∧ (∀ *t x w*. *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ ⟶ *wset s t* ≠ ⌊*PostWS WSNotified*⌋ ∧ *wset s t* ≠ ⌊*PostWS WSWokenUp*⌋)
unfolding *neq-no-wait-locks-conv*
apply(*rule iffI*)
apply(*intro strip conjI*)
apply(*blast dest*: *deadlockD1*)
apply(*blast dest*: *deadlockD1*)
apply(*blast elim*: *deadlockD2*)
apply(*blast dest*: *deadlockD3*)
apply(*blast dest*: *deadlockD3*)
apply(*elim conjE exE*)
apply(*rule deadlockI*)
apply *blast*
apply(*rotate-tac 1*)
apply(*erule allE*, *rotate-tac −1*)
apply(*erule allE*, *rotate-tac −1*)
apply(*erule allE*, *rotate-tac −1*)
apply(*erule impE*, *blast*)
apply(*elim exE conjE*)
apply(*erule must-wait.cases*)
apply(*clarify*)
apply(*rotate-tac 3*)
apply(*rule exI conjI*|*erule not-sym*|*assumption*)+
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply(*case-tac w*)
apply *blast*
apply *blast*
done

lemma *all-waiting-implies-deadlock*:

assumes *lock-thread-ok* (*locks s*) (*thr s*)
and *normal*: $\bigwedge t\ x.\ [\![\ thr\ s\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor;\ \neg\ final\ x;\ wset\ s\ t = None\ ]\!]$
$\Longrightarrow t \vdash \langle x,\ shr\ s\rangle\ \wr \wedge (\forall LT.\ t \vdash \langle x,\ shr\ s\rangle\ LT\ \wr \longrightarrow (\exists\, lt \in LT.\ must\text{-}wait\ s\ t\ lt\ (dom\ (thr\ s))))$
and *acquire*: $\bigwedge t\ x\ ln\ l.\ [\![\ thr\ s\ t = \lfloor(x,\ ln)\rfloor;\ \neg\ waiting\ (wset\ s\ t);\ ln\ \$\ l > 0\ ]\!]$
$\Longrightarrow \exists\, l'.\ ln\ \$\ l' > 0 \wedge \neg\ may\text{-}lock\ (locks\ s\ \$\ l')\ t$
and *wakeup*: $\bigwedge t\ x\ w.\ thr\ s\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor \Longrightarrow wset\ s\ t \neq \lfloor PostWS\ w\rfloor$
shows *deadlock s*
proof(*rule deadlockI*)
fix *T X*
assume *thr s T* = $\lfloor(X,\ no\text{-}wait\text{-}locks)\rfloor$ ¬ *final X wset s T* = *None*
thus *T* ⊢ ⟨*X*, *shr s*⟩ ≀ ∧ (∀ *LT*. *T* ⊢ ⟨*X*, *shr s*⟩ *LT* ≀ ⟶ (∃ *lt*∈*LT*. *must-wait s T lt* (*dom* (*thr s*))))
by(*rule normal*)
next
fix *T X LN l′*
assume *thr s T* = $\lfloor(X,\ LN)\rfloor$
and *0* < *LN* $ *l′*
and *wset*: ¬ *waiting* (*wset s T*)
from *acquire*[*OF* ‹*thr s T* = $\lfloor(X,\ LN)\rfloor$› *wset*, *OF* ‹*0* < *LN* $ *l′*›]
obtain *l′* **where** *0* < *LN* $ *l′* ¬ *may-lock* (*locks s* $ *l′*) *T* **by** *blast*
then obtain *t′* **where** *T* ≠ *t′ has-lock* (*locks s* $ *l′*) *t′*
unfolding *not-may-lock-conv* **by** *fastforce*
moreover with ‹*lock-thread-ok* (*locks s*) (*thr s*)›
have *thr s t′* ≠ *None* **by**(*auto dest*: *lock-thread-okD*)
ultimately show ∃ *l t′*. *0* < *LN* $ *l* ∧ *T* ≠ *t′* ∧ *thr s t′* ≠ *None* ∧ *has-lock* (*locks s* $ *l*) *t′*
using ‹*0* < *LN* $ *l′*› **by**(*auto*)
qed(*rule wakeup*)

lemma *mfinal-deadlock*:
mfinal s ⟹ *deadlock s*
unfolding *mfinal-def2*
by(*rule deadlockI*)(*auto simp add*: *final-thread-def*)

Now deadlock for single threads

lemma *must-wait-mono*:
(⋀*x*. *x* ∈ *A* ⟶ *x* ∈ *B*) ⟹ *must-wait s t lt A* ⟶ *must-wait s t lt B*
by(*auto simp add*: *must-wait-iff split*: *sum.split elim*: *all-final-except-mono′*)

lemma *must-wait-mono′*:
[\![*must-wait s t lt A*; *A* ⊆ *B*]\!] ⟹ *must-wait s t lt B*
using *must-wait-mono*[*of A B s t lt*]
by *blast*

end

lemma *UN-mono*: [\![*x* ∈ *A* ⟶ *x* ∈ *A′*; *x* ∈ *B* ⟶ *x* ∈ *B′*]\!] ⟹ *x* ∈ *A* ∪ *B* ⟶ *x* ∈ *A′* ∪ *B′*
by *blast*

lemma *Collect-mono-conv* [*mono*]: *x* ∈ {*x*. *P x*} ⟷ *P x*
by *blast*

context *multithreaded-base* **begin**

coinductive-set *deadlocked* :: ('*l*,'*t*,'*x*,'*m*,'*w*) *state* ⇒ '*t set*

for $s :: ('l,'t,'x,'m,'w)$ *state* **where**
deadlockedLock:
$[\![$ *thr s t* $= \lfloor(x,$ *no-wait-locks*$)\rfloor$; $t \vdash \langle x,$ *shr s*$\rangle$ $\wr$; *wset s t* $=$ *None*;
$\bigwedge LT.\ t \vdash \langle x,$ *shr s*$\rangle$ *LT* $\wr \Longrightarrow \exists\, lt \in LT.$ *must-wait s t lt* (*deadlocked s* $\cup$ *final-threads s*) $]\!]$
$\Longrightarrow t \in$ *deadlocked s*

| *deadlockedWait*:
$\bigwedge ln.\ [\![$ *thr s t* $= \lfloor(x,$ *ln*$)\rfloor$; *all-final-except s* (*deadlocked s*); *waiting* (*wset s t*) $]\!] \Longrightarrow t \in$ *deadlocked s*

| *deadlockedAcquire*:
$\bigwedge ln.\ [\![$ *thr s t* $= \lfloor(x,$ *ln*$)\rfloor$; $\neg$ *waiting* (*wset s t*); *ln* \$ *l* > 0; *has-lock* (*locks s* \$ *l*) *t'*; $t' \neq t$;
$t' \in$ *deadlocked s* $\vee$ *final-thread s t'* $]\!]$
$\Longrightarrow t \in$ *deadlocked s*
monos *must-wait-mono UN-mono*

lemma *deadlockedAcquire-must-wait*:
$\bigwedge ln.\ [\![$ *thr s t* $= \lfloor(x,$ *ln*$)\rfloor$; $\neg$ *waiting* (*wset s t*); *ln* \$ *l* > 0; *must-wait s t* (*Inl l*) (*deadlocked s* $\cup$ *final-threads s*) $]\!]$
$\Longrightarrow t \in$ *deadlocked s*
apply(*erule must-wait-elims*)
apply(*erule* (*2*) *deadlockedAcquire*)
apply *auto*
done

lemma *deadlocked-elims* [*consumes 1*, *case-names lock wait acquire*]:
assumes $t \in$ *deadlocked s*
and *lock*: $\bigwedge x.\ [\![$ *thr s t* $= \lfloor(x,$ *no-wait-locks*$)\rfloor$; $t \vdash \langle x,$ *shr s*$\rangle$ $\wr$; *wset s t* $=$ *None*;
$\bigwedge LT.\ t \vdash \langle x,$ *shr s*$\rangle$ *LT* $\wr \Longrightarrow \exists\, lt \in LT.$ *must-wait s t lt* (*deadlocked s* $\cup$ *final-threads s*) $]\!]$
$\Longrightarrow$ *thesis*
and *wait*: $\bigwedge x\ ln.\ [\![$ *thr s t* $= \lfloor(x,$ *ln*$)\rfloor$; *all-final-except s* (*deadlocked s*); *waiting* (*wset s t*) $]\!]$
$\Longrightarrow$ *thesis*
and *acquire*: $\bigwedge x\ ln\ l\ t'.$
$[\![$ *thr s t* $= \lfloor(x,$ *ln*$)\rfloor$; $\neg$ *waiting* (*wset s t*); $0 <$ *ln* \$ *l*; *has-lock* (*locks s* \$ *l*) *t'*; $t \neq t'$;
$t' \in$ *deadlocked s* $\vee$ *final-thread s t'* $]\!] \Longrightarrow$ *thesis*
shows *thesis*
using *assms* **by** *cases blast+*

lemma *deadlocked-coinduct*
[*consumes 1*, *case-names deadlocked*, *case-conclusion deadlocked Lock Wait Acquire*, *coinduct set*: *deadlocked*]:
assumes *major*: $t \in X$
and *step*:
$\bigwedge t.\ t \in X \Longrightarrow$
$(\exists\, x.$ *thr s t* $= \lfloor(x,$ *no-wait-locks*$)\rfloor \wedge t \vdash \langle x,$ *shr s*$\rangle$ $\wr \wedge$ *wset s t* $=$ *None* $\wedge$
$(\forall\, LT.\ t \vdash \langle x,$ *shr s*$\rangle$ *LT* $\wr \longrightarrow (\exists\, lt \in LT.$ *must-wait s t lt* ($X \cup$ *deadlocked s* $\cup$ *final-threads s*))))
$\vee$
$(\exists\, x\ ln.$ *thr s t* $= \lfloor(x,$ *ln*$)\rfloor \wedge$ *all-final-except s* ($X \cup$ *deadlocked s*) $\wedge$ *waiting* (*wset s t*)) $\vee$
$(\exists\, x\ l\ t'\ ln.$ *thr s t* $= \lfloor(x,$ *ln*$)\rfloor \wedge \neg$ *waiting* (*wset s t*) $\wedge\ 0 <$ *ln* \$ *l* $\wedge$ *has-lock* (*locks s* \$ *l*) *t'* $\wedge$
$t' \neq t \wedge ((t' \in X \vee t' \in$ *deadlocked s*$) \vee$ *final-thread s t'*))
shows $t \in$ *deadlocked s*
using *major*
proof(*coinduct*)
case (*deadlocked t*)
have $X \cup$ *deadlocked s* $\cup$ *final-threads s* $= \{x.\ x \in X \vee x \in$ *deadlocked s* $\vee\ x \in$ *final-threads s*$\}$

by *auto*
moreover have $X \cup$ *deadlocked* $s = \{x.\ x \in X \vee x \in$ *deadlocked* $s\}$ **by** *blast*
ultimately show *?case* **using** *step*[*OF deadlocked*] **by**(*elim disjE*) *simp-all*
qed

definition *deadlocked'* :: ('*l*,'*t*,'*x*,'*m*,'*w*) *state* ⇒ *bool* **where**
deadlocked' $s \equiv (\forall t.$ *not-final-thread* $s\ t \longrightarrow t \in$ *deadlocked* $s)$

lemma *deadlocked'I*:
$(\bigwedge t.$ *not-final-thread* $s\ t \Longrightarrow t \in$ *deadlocked* $s) \Longrightarrow$ *deadlocked'* s
by(*auto simp add*: *deadlocked'-def*)

lemma *deadlocked'D2*:
$[\![$ *deadlocked'* s; *not-final-thread* $s\ t$; $t \in$ *deadlocked* $s \Longrightarrow$ *thesis* $]\!] \Longrightarrow$ *thesis*
by(*auto simp add*: *deadlocked'-def*)

lemma *not-deadlocked'I*:
$[\![$ *not-final-thread* $s\ t$; $t \notin$ *deadlocked* $s\ ]\!] \Longrightarrow \neg$ *deadlocked'* s
by(*auto dest*: *deadlocked'D2*)

lemma *deadlocked'-intro*:
$[\![\forall t.$ *not-final-thread* $s\ t \longrightarrow t \in$ *deadlocked* $s\ ]\!] \Longrightarrow$ *deadlocked'* s
by(*rule deadlocked'I*)(*blast*)+

lemma *deadlocked-thread-exists*:
assumes $t \in$ *deadlocked* s
and $\bigwedge x\ ln.$ *thr* $s\ t = \lfloor(x,\ ln)\rfloor \Longrightarrow$ *thesis*
shows *thesis*
using *assms*
by *cases blast*+

end

context *multithreaded* **begin**

lemma *red-no-deadlock*:
assumes *P*: $s\ -t▷ta→\ s'$
and *dead*: $t \in$ *deadlocked* s
shows *False*
proof −
from *P* **show** *False*
proof(*cases*)
case (*redT-normal* $x\ x'\ m'$)
note *red* = ‹$t \vdash \langle x,$ *shr* $s\rangle\ -ta→\ \langle x',\ m'\rangle$›
note *tst* = ‹*thr* $s\ t = \lfloor(x,$ *no-wait-locks*$)\rfloor$›
note *aok* = ‹*actions-ok* $s\ t$ *ta*›
show *False*
proof(*cases* ∃ *w*. *wset* $s\ t = \lfloor$*InWS* $w\rfloor$)
case *True* **with** *aok* **show** *?thesis* **by**(*auto simp add*: *wset-actions-ok-def split*: *if-split-asm*)
next
case *False*
with *dead tst*
have *mle*: $t \vdash \langle x,$ *shr* $s\rangle\ \wr$
and *cledead*: ∀ *LT*. $t \vdash \langle x,$ *shr* $s\rangle$ *LT* $\wr \longrightarrow (\exists lt \in LT.$ *must-wait* $s\ t\ lt$ (*deadlocked* $s \cup$ *final-threads*

```
s))
      by(cases, auto simp add: waiting-def)+
    let ?LT = collect-waits ta
    from red have t ⊢ ⟨x, shr s⟩ ?LT ≀ by(auto intro: can-syncI)
    then obtain lt where lt: lt ∈ ?LT and mw: must-wait s t lt (deadlocked s ∪ final-threads s)
      by(blast dest: cledead[rule-format])
    from mw show False
    proof(cases rule: must-wait-elims)
      case (lock l t')
      from ‹lt = Inl l› lt have l ∈ collect-locks {|ta|}_l by(auto)
      with aok have may-lock (locks s $ l) t
        by(auto elim!: collect-locksE lock-ok-las-may-lock)
      with ‹has-lock (locks s $ l) t'› have t' = t
        by(auto dest: has-lock-may-lock-t-eq)
      with ‹t' ≠ t› show False by contradiction
    next
      case (join t')
      from ‹lt = Inr (Inl t')› lt have Join t' ∈ set {|ta|}_c by auto
      from ‹not-final-thread s t'› obtain x'' ln''
        where thr s t' = ⌊(x'', ln'')⌋ by(rule not-final-thread-existsE)
      moreover with ‹Join t' ∈ set {|ta|}_c› aok
      have final x'' ln'' = no-wait-locks wset s t' = None
        by(auto dest: cond-action-oks-Join)
      ultimately show False using ‹not-final-thread s t'› by(auto)
    next
      case (interrupt t')
      from aok lt ‹lt = Inr (Inr t')›
      have t' ∈ interrupts s
        by(auto intro: collect-interrupts-interrupted)
      with ‹t' ∉ interrupts s› show False by contradiction
    qed
  qed
next
  case (redT-acquire x n ln)
  show False
  proof(cases ∃ w. wset s t = ⌊InWS w⌋)
    case True with ‹¬ waiting (wset s t)› show ?thesis
      by(auto simp add: not-waiting-iff)
  next
    case False
    with dead ‹thr s t = ⌊(x, ln)⌋› ‹0 < ln $ n›
    obtain l t' where 0 < ln $ l t ≠ t'
      and has-lock (locks s $ l) t'
      by(cases)(fastforce simp add: waiting-def)+
    hence ¬ may-acquire-all (locks s) t ln
      by(auto elim: may-acquire-allE dest: has-lock-may-lock-t-eq)
    with ‹may-acquire-all (locks s) t ln› show ?thesis by contradiction
  qed
 qed
qed

lemma deadlocked'-no-red:
  ⟦ s -t▷ta→ s'; deadlocked' s ⟧ ⟹ False
apply(rule red-no-deadlock)
```

apply(*assumption*)
apply(*erule deadlocked'D2*)
by(*rule red-not-final-thread*)

lemma *not-final-thread-deadlocked-final-thread* [*iff*]:
thr s t = ⌊*xln*⌋ ⟹ *not-final-thread s t* ∨ *t* ∈ *deadlocked s* ∨ *final-thread s t*
by(*auto simp add*: *not-final-thread-final-thread-conv*[*symmetric*])

lemma *all-waiting-deadlocked*:
assumes *not-final-thread s t*
and *lock-thread-ok* (*locks s*) (*thr s*)
and *normal*: ⋀*t x*. ⟦ *thr s t* = ⌊(*x*, *no-wait-locks*)⌋; ¬ *final x*; *wset s t* = *None* ⟧
⟹ *t* ⊢ ⟨*x*, *shr s*⟩ ≀ ∧ (∀ *LT*. *t* ⊢ ⟨*x*, *shr s*⟩ *LT* ≀ ⟶ (∃ *lt*∈*LT*. *must-wait s t lt* (*final-threads s*)))
and *acquire*: ⋀*t x ln l*. ⟦ *thr s t* = ⌊(*x*, *ln*)⌋; ¬ *waiting* (*wset s t*); *ln* $ *l* > *0* ⟧
⟹ ∃ *l'*. *ln* $ *l'* > *0* ∧ ¬ *may-lock* (*locks s* $ *l'*) *t*
and *wakeup*: ⋀*t x w*. *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ ⟹ *wset s t* ≠ ⌊*PostWS w*⌋
shows *t* ∈ *deadlocked s*
proof −
from ‹*not-final-thread s t*›
have *t* ∈ {*t*. *not-final-thread s t*} **by** *simp*
thus *?thesis*
proof(*coinduct*)
case (*deadlocked z*)
hence *not-final-thread s z* **by** *simp*
then obtain *x' ln'* **where** *thr s z* = ⌊(*x'*, *ln'*)⌋ **by**(*fastforce elim*!: *not-final-thread-existsE*)
{
assume *wset s z* = *None* ¬ *final x'*
and [*simp*]: *ln'* = *no-wait-locks*
with ‹*thr s z* = ⌊(*x'*, *ln'*)⌋›
have *z* ⊢ ⟨*x'*, *shr s*⟩ ≀ ∧ (∀ *LT*. *z* ⊢ ⟨*x'*, *shr s*⟩ *LT* ≀ ⟶ (∃ *lt* ∈ *LT*. *must-wait s z lt* (*final-threads s*)))
by(*auto dest*: *normal*)
then obtain *z* ⊢ ⟨*x'*, *shr s*⟩ ≀
and *clnml*: ⋀*LT*. *z* ⊢ ⟨*x'*, *shr s*⟩ *LT* ≀ ⟹ ∃ *lt* ∈ *LT*. *must-wait s z lt* (*final-threads s*) **by**(*blast*)
{ fix *LT*
assume *z* ⊢ ⟨*x'*, *shr s*⟩ *LT* ≀
then obtain *lt* **where** *mw*: *must-wait s z lt* (*final-threads s*) **and** *lt*: *lt* ∈ *LT*
by(*blast dest*: *clnml*)
from *mw* **have** *must-wait s z lt* ({*t*. *not-final-thread s t*} ∪ *deadlocked s* ∪ *final-threads s*)
by(*blast intro*: *must-wait-mono'*)
with *lt* **have** ∃ *lt* ∈ *LT*. *must-wait s z lt* ({*t*. *not-final-thread s t*} ∪ *deadlocked s* ∪ *final-threads s*)
by *blast* **}**
with ‹*z* ⊢ ⟨*x'*, *shr s*⟩ ≀› ‹*thr s z* = ⌊(*x'*, *ln'*)⌋› ‹*wset s z* = *None*› **have** *?case* **by**(*simp*) **}**
note *c1* = *this*
{
assume *wsz*: ¬ *waiting* (*wset s z*)
and *ln'* ≠ *no-wait-locks*
from ‹*ln'* ≠ *no-wait-locks*› **obtain** *l* **where** *0* < *ln'* $ *l*
by(*auto simp add*: *neq-no-wait-locks-conv*)
with *wsz* ‹*thr s z* = ⌊(*x'*, *ln'*)⌋›
obtain *l'* **where** *0* < *ln'* $ *l'* ¬ *may-lock* (*locks s* $ *l'*) *z*
by(*blast dest*: *acquire*)

```
  then obtain t'' where t'' ≠ z has-lock (locks s $ l') t''
    unfolding not-may-lock-conv by blast
  with ‹lock-thread-ok (locks s) (thr s)›
  obtain x'' ln'' where thr s t'' = ⌊(x'', ln'')⌋
    by(auto elim!: lock-thread-ok-has-lockE)
  hence (not-final-thread s t'' ∨ t'' ∈ deadlocked s) ∨ final-thread s t''
    by(clarsimp simp add: not-final-thread-iff final-thread-def)
  with wsz ‹0 < ln' $ l'› ‹thr s z = ⌊(x', ln')⌋› ‹t'' ≠ z› ‹has-lock (locks s $ l') t''›
  have ?Acquire by simp blast
  hence ?case by simp }
note c2 = this
{ fix w
  assume waiting (wset s z)
  with ‹thr s z = ⌊(x', ln')⌋›
  have ?Wait by(clarsimp simp add: all-final-except-def)
  hence ?case by simp }
note c3 = this
from ‹not-final-thread s z› ‹thr s z = ⌊(x', ln')⌋› show ?case
proof(cases rule: not-final-thread-cases2)
  case final show ?thesis
  proof(cases wset s z)
    case None show ?thesis
    proof(cases ln' = no-wait-locks)
      case True with None final show ?thesis by(rule c1)
    next
      case False
      from None have ¬ waiting (wset s z) by(simp add: not-waiting-iff)
      thus ?thesis using False by(rule c2)
    qed
  next
    case (Some w)
    show ?thesis
    proof(cases w)
      case (InWS w')
      with Some have waiting (wset s z) by(simp add: waiting-def)
      thus ?thesis by(rule c3)
    next
      case (PostWS w')
      with Some have ¬ waiting (wset s z) by(simp add: not-waiting-iff)
      moreover from PostWS ‹thr s z = ⌊(x', ln')⌋› Some
      have ln' ≠ no-wait-locks by(auto dest: wakeup)
      ultimately show ?thesis by(rule c2)
    qed
  qed
next
  case wait-locks show ?thesis
  proof(cases wset s z)
    case None
    hence ¬ waiting (wset s z) by(simp add: not-waiting-iff)
    thus ?thesis using wait-locks by(rule c2)
  next
    case (Some w)
    show ?thesis
    proof(cases w)
```

```
        case (InWS w')
        with Some have waiting (wset s z) by(simp add: waiting-def)
        thus ?thesis by(rule c3)
      next
        case (PostWS w')
        with Some have ¬ waiting (wset s z) by(simp add: not-waiting-iff)
        moreover from PostWS ‹thr s z = ⌊(x', ln')⌋› Some
        have ln' ≠ no-wait-locks by(auto dest: wakeup)
        ultimately show ?thesis by(rule c2)
      qed
    qed
  next
    case (wait-set w)
    show ?thesis
    proof(cases w)
      case (InWS w')
      with wait-set have waiting (wset s z) by(simp add: waiting-def)
      thus ?thesis by(rule c3)
    next
      case (PostWS w')
      with wait-set have ¬ waiting (wset s z) by(simp add: not-waiting-iff)
      moreover from PostWS ‹thr s z = ⌊(x', ln')⌋› wait-set
      have ln' ≠ no-wait-locks by(auto dest: wakeup[simplified])
      ultimately show ?thesis by(rule c2)
    qed
  qed
 qed
qed
```

Equivalence proof for both notions of deadlock

```
lemma deadlock-implies-deadlocked':
  assumes dead: deadlock s
  shows deadlocked' s
proof −
  show ?thesis
  proof(rule deadlocked'I)
    fix t
    assume not-final-thread s t
    hence t ∈ {t. not-final-thread s t} ..
    thus t ∈ deadlocked s
    proof(coinduct)
      case (deadlocked t'')
      hence not-final-thread s t'' ..
      then obtain x'' ln'' where tst'': thr s t'' = ⌊(x'', ln'')⌋
        by(rule not-final-thread-existsE)
      { assume waiting (wset s t'')
        moreover
        with tst'' have nfine: not-final-thread s t''
          unfolding waiting-def
          by(blast intro: not-final-thread.intros)
        ultimately have ?case using tst''
          by(blast intro: all-final-exceptI not-final-thread-final) }
      note c1 = this
      {
```

```
      assume wst'': ¬ waiting (wset s t'')
        and ln'' ≠ no-wait-locks
      then obtain l where l: ln'' $ l > 0
        by(auto simp add: neq-no-wait-locks-conv)
      with dead wst'' tst'' obtain l' T
        where ln'' $ l' > 0 t'' ≠ T
        and hl: has-lock (locks s $ l') T
        and tsT: thr s T ≠ None
        by − (erule deadlockD2)
      moreover from ‹thr s T ≠ None›
      obtain xln where tsT: thr s T = ⌊xln⌋ by auto
      then obtain X LN where thr s T = ⌊(X, LN)⌋
        by(cases xln, auto)
      moreover hence not-final-thread s T ∨ final-thread s T
        by(auto simp add: final-thread-def not-final-thread-iff)
      ultimately have ?case using wst'' tst'' by blast }
    note c2 = this
    { assume wset s t'' = None
      and [simp]: ln'' = no-wait-locks
      moreover
      with ‹not-final-thread s t''› tst''
      have ¬ final x'' by(auto)
      ultimately obtain t'' ⊢ ⟨x'', shr s⟩ ≀
        and clnml: ⋀LT. t'' ⊢ ⟨x'', shr s⟩ LT ≀ ⟹ ∃ t'. thr s t' ≠ None ∧ (∃ lt∈LT. must-wait s t''
lt (dom (thr s)))
        using ‹thr s t'' = ⌊(x'', ln'')⌋› ‹deadlock s›
        by(blast elim: deadlockD1)
      { fix LT
        assume t'' ⊢ ⟨x'', shr s⟩ LT ≀
        then obtain lt where lt: lt ∈ LT
          and mw: must-wait s t'' lt (dom (thr s))
          by(blast dest: clnml)
        note mw
        also have dom (thr s) = {t. not-final-thread s t} ∪ deadlocked s ∪ final-threads s
          by(auto simp add: not-final-thread-conv dest: deadlocked-thread-exists elim: final-threadE)
       finally have ∃ lt∈LT. must-wait s t'' lt ({t. not-final-thread s t} ∪ deadlocked s ∪ final-threads
s)
          using lt by blast }
      with ‹t'' ⊢ ⟨x'', shr s⟩ ≀› tst'' ‹wset s t'' = None› have ?case by(simp) }
    note c3 = this
    from ‹not-final-thread s t''› tst'' show ?case
    proof(cases rule: not-final-thread-cases2)
      case final show ?thesis
      proof(cases wset s t'')
        case None show ?thesis
        proof(cases ln'' = no-wait-locks)
          case True with None show ?thesis by(rule c3)
        next
          case False
          from None have ¬ waiting (wset s t'') by(simp add: not-waiting-iff)
          thus ?thesis using False by(rule c2)
        qed
      next
        case (Some w)
```

```
        show ?thesis
        proof(cases w)
          case (InWS w')
          with Some have waiting (wset s t'') by(simp add: waiting-def)
          thus ?thesis by(rule c1)
        next
          case (PostWS w')
          hence ¬ waiting (wset s t'') using Some by(simp add: not-waiting-iff)
          moreover from PostWS Some tst''
          have ln'' ≠ no-wait-locks by(auto dest: deadlockD3[OF dead])
          ultimately show ?thesis by(rule c2)
        qed
      qed
    next
      case wait-locks show ?thesis
      proof(cases waiting (wset s t''))
        case False
        thus ?thesis using wait-locks by(rule c2)
      next
        case True thus ?thesis by(rule c1)
      qed
    next
      case (wait-set w)
      show ?thesis
      proof(cases w)
        case InWS
        with wait-set have waiting (wset s t'') by(simp add: waiting-def)
        thus ?thesis by(rule c1)
      next
        case (PostWS w')
        hence ¬ waiting (wset s t'') using wait-set
          by(simp add: not-waiting-iff)
        moreover from PostWS wait-set tst''
        have ln'' ≠ no-wait-locks by(auto dest: deadlockD3[OF dead])
        ultimately show ?thesis by(rule c2)
      qed
    qed
  qed
qed

lemma deadlocked'-implies-deadlock:
  assumes dead: deadlocked' s
  shows deadlock s
proof −
  have deadlocked: ⋀t. not-final-thread s t ⟹ t ∈ deadlocked s
    using dead by(rule deadlocked'D2)
  show ?thesis
  proof(rule deadlockI)
    fix t' x'
    assume thr s t' = ⌊(x', no-wait-locks)⌋
      and ¬ final x'
      and wset s t' = None
    hence not-final-thread s t' by(auto intro: not-final-thread-final)
```

```
    hence t' ∈ deadlocked s by(rule deadlocked)
    thus t' ⊢ ⟨x', shr s⟩ ≀ ∧ (∀ LT. t' ⊢ ⟨x', shr s⟩ LT ≀ ⟶ (∃ lt ∈ LT. must-wait s t' lt (dom (thr
s))))
    proof(cases rule: deadlocked-elims)
      case (lock x'')
        note lock = ‹⋀LT. t' ⊢ ⟨x'', shr s⟩ LT ≀ ⟹ ∃ lt ∈ LT. must-wait s t' lt (deadlocked s ∪
final-threads s)›
      from ‹thr s t' = ⌊(x'', no-wait-locks)⌋› ‹thr s t' = ⌊(x', no-wait-locks)⌋›
      have [simp]: x' = x'' by auto
      { fix LT
        assume t' ⊢ ⟨x'', shr s⟩ LT ≀
        from lock[OF this] obtain lt where lt: lt ∈ LT
          and mw: must-wait s t' lt (deadlocked s ∪ final-threads s) by blast
        have deadlocked s ∪ final-threads s ⊆ dom (thr s)
          by(auto elim: final-threadE dest: deadlocked-thread-exists)
        with mw have must-wait s t' lt (dom (thr s)) by(rule must-wait-mono')
        with lt have ∃ lt∈LT. must-wait s t' lt (dom (thr s)) by blast }
      with ‹t' ⊢ ⟨x'', shr s⟩ ≀› show ?thesis by(auto)
    next
      case (wait x'' ln'')
      from ‹wset s t' = None› ‹waiting (wset s t')›
      have False by(simp add: waiting-def)
      thus ?thesis ..
    next
      case (acquire x'' ln'' l'' T)
      from ‹thr s t' = ⌊(x'', ln'')⌋› ‹thr s t' = ⌊(x', no-wait-locks)⌋› ‹0 < ln'' $ l''›
      have False by(auto)
      thus ?thesis ..
    qed
  next
    fix t' x' ln' l
    assume thr s t' = ⌊(x', ln')⌋
      and 0 < ln' $ l
      and wst': ¬ waiting (wset s t')
    hence not-final-thread s t' by(auto intro: not-final-thread-wait-locks)
    hence t' ∈ deadlocked s by(rule deadlocked)
    thus ∃ l T. 0 < ln' $ l ∧ t' ≠ T ∧ thr s T ≠ None ∧ has-lock (locks s $ l) T
    proof(cases rule: deadlocked-elims)
      case (lock x'')
      from ‹thr s t' = ⌊(x', ln')⌋› ‹thr s t' = ⌊(x'', no-wait-locks)⌋› ‹0 < ln' $ l›
      have False by auto
      thus ?thesis ..
    next
      case (wait x' ln')
      from wst' ‹waiting (wset s t')›
      have False by contradiction
      thus ?thesis ..
    next
      case (acquire x'' ln'' l'' t'')
      from ‹thr s t' = ⌊(x'', ln'')⌋› ‹thr s t' = ⌊(x', ln')⌋›
      have [simp]: x' = x'' ln' = ln'' by auto
      moreover from ‹t'' ∈ deadlocked s ∨ final-thread s t''›
      have thr s t'' ≠ None
        by(auto elim: deadlocked-thread-exists simp add: final-thread-def)
```

```
      with ‹0 < ln'' $ l''› ‹has-lock (locks s $ l'') t''› ‹t' ≠ t''› ‹thr s t' = ⌊(x'', ln'')⌋›
      show ?thesis by auto
    qed
  next
    fix t x w
    assume tst: thr s t = ⌊(x, no-wait-locks)⌋
    show wset s t ≠ ⌊PostWS w⌋
    proof
      assume wset s t = ⌊PostWS w⌋
      moreover with tst have not-final-thread s t
        by(auto simp add: not-final-thread-iff)
      hence t ∈ deadlocked s by(rule deadlocked)
      ultimately show False using tst
        by(auto elim: deadlocked.cases simp add: waiting-def)
    qed
  qed
qed

lemma deadlock-eq-deadlocked':
  deadlock = deadlocked'
by(rule ext)(auto intro: deadlock-implies-deadlocked' deadlocked'-implies-deadlock)

lemma deadlock-no-red:
  ⟦ s -t▷ta→ s'; deadlock s ⟧ ⟹ False
unfolding deadlock-eq-deadlocked'
by(rule deadlocked'-no-red)

lemma deadlock-no-active-threads:
  assumes dead: deadlock s
  shows active-threads s = {}
proof(rule equals0I)
  fix t
  assume active: t ∈ active-threads s
  then obtain ta s' where s -t▷ta→ s' by(auto dest: active-thread-ex-red)
  thus False using dead by(rule deadlock-no-red)
qed

end

locale preserve-deadlocked = multithreaded final r convert-RA
  for final :: 'x ⇒ bool
  and r :: ('l,'t,'x,'m,'w,'o) semantics (‹- ⊢ - -→ -› [50,0,0,50] 80)
  and convert-RA :: 'l released-locks ⇒ 'o list
  +
  fixes wf-state :: ('l,'t,'x,'m,'w) state set
  assumes invariant3p-wf-state: invariant3p redT wf-state
  assumes can-lock-preserved:
    ⟦ s ∈ wf-state; s -t'▷ta'→ s';
       thr s t = ⌊(x, no-wait-locks)⌋; t ⊢ ⟨x, shr s⟩ ≀ ⟧
    ⟹ t ⊢ ⟨x, shr s'⟩ ≀
  and can-lock-devreserp:
    ⟦ s ∈ wf-state; s -t'▷ta'→ s';
       thr s t = ⌊(x, no-wait-locks)⌋; t ⊢ ⟨x, shr s'⟩ L ≀ ⟧
    ⟹ ∃ L'⊆L. t ⊢ ⟨x, shr s⟩ L' ≀
```

begin

lemma *redT-deadlocked-subset*:
  assumes *wfs*: $s \in$ *wf-state*
  and *Red*: $s -t\triangleright ta\rightarrow s'$
  shows *deadlocked* $s \subseteq$ *deadlocked* s'
proof
  fix t'
  assume t'*dead*: $t' \in$ *deadlocked* s
  from *Red* **have** *tndead*: $t \notin$ *deadlocked* s
    by(*auto dest*: *red-no-deadlock*)
  with t'*dead* **have** $t't$: $t' \neq t$ **by** *auto*
  { fix t'
    assume *final-thread* s t'
    then obtain x' ln' **where** *tst'*: *thr* s $t' = \lfloor(x', ln')\rfloor$ **by**(*auto elim*!: *final-threadE*)
    with ‹*final-thread* s t'› **have** *final* x'
      and *wset* s $t' =$ *None* **and** [*simp*]: $ln' =$ *no-wait-locks*
      by(*auto elim*: *final-threadE*)
    with *Red tst'* **have** $t \neq t'$ **by** *cases*(*auto dest*: *final-no-red*)
    with *Red tst'* **have** *thr* s' $t' = \lfloor(x', ln')\rfloor$
      by *cases*(*auto intro*: *redT-updTs-Some*)
    moreover from *Red* ‹$t \neq t'$› ‹*wset* s $t' =$ *None*›
    have *wset* s' $t' =$ *None* **by** *cases*(*auto simp*: *redT-updWs-None-implies-None*)
    ultimately have *final-thread* s' t' **using** *tst'* ‹*final* x'›
      by(*auto simp add*: *final-thread-def*) **}**
  hence *subset*: *deadlocked* $s \cup$ *final-threads* $s \subseteq$ *deadlocked* $s \cup$ *deadlocked* $s' \cup$ *final-threads* s' **by**(*auto*)

  from *Red* **show** $t' \in$ *deadlocked* s'
  proof(*cases*)
    case (*redT-normal* x x' m')
    note *red* = ‹$t \vdash \langle x, shr\ s\rangle -ta\rightarrow \langle x', m'\rangle$›
      and *tst* = ‹*thr* s $t = \lfloor(x,$ *no-wait-locks*$)\rfloor$›
      and *aok* = ‹*actions-ok* s t *ta*›
      and $s' =$ ‹*redT-upd* s t *ta* x' m' s'›
    from *red* **have** ¬ *final* x **by**(*auto dest*: *final-no-red*)
    with *tndead tst* **have** *nafe*: ¬ *all-final-except* s (*deadlocked* s)
      by(*fastforce simp add*: *all-final-except-def not-final-thread-iff*)
    from t'*dead* **show** *?thesis*
    proof(*coinduct*)
      case (*deadlocked* t'')
      note t''*dead* = *this*
      with *Red* **have** $t''t$: $t'' \neq t$
        by(*auto dest*: *red-no-deadlock*)
      from t''*dead* **show** *?case*
      proof(*cases rule*: *deadlocked-elims*)
        case (*lock* X)
        hence *est''*: *thr* s $t'' = \lfloor(X,$ *no-wait-locks*$)\rfloor$
          and *msE*: $t'' \vdash \langle X, shr\ s\rangle\ \wr$
          and *csexdead*: $\bigwedge LT.\ t'' \vdash \langle X, shr\ s\rangle\ LT\ \wr \Longrightarrow \exists\, lt \in LT.$ *must-wait* s t'' lt (*deadlocked* $s \cup$ *final-threads* s)
          by *auto*
        from $t''t$ *Red est''*
        have *es't''*: *thr* s' $t'' = \lfloor(X,$ *no-wait-locks*$)\rfloor$
          by(*cases* s)(*cases* s', *auto elim*!: *redT-ts-Some-inv*)

```
note es't'' moreover
from wfs Red est'' msE have msE': t'' ⊢ ⟨X, shr s'⟩ ≀ by(rule can-lock-preserved)
moreover
{ fix LT
  assume clL'': t'' ⊢ ⟨X, shr s'⟩ LT ≀
  with est'' have ∃ LT'⊆LT. t'' ⊢ ⟨X, shr s⟩ LT' ≀
    by(rule can-lock-devreserp[OF wfs Red])
  then obtain LT' where clL': t'' ⊢ ⟨X, shr s⟩ LT' ≀
    and LL': LT' ⊆ LT by blast
  with csexdead obtain lt
    where lt: lt ∈ LT and mw: must-wait s t'' lt (deadlocked s ∪ final-threads s)
    by blast
  from mw have must-wait s' t'' lt (deadlocked s ∪ deadlocked s' ∪ final-threads s')
  proof(cases rule: must-wait-elims)
    case (lock l t')
    from ‹t' ∈ deadlocked s ∪ final-threads s› Red have tt': t ≠ t'
      by(auto dest: red-no-deadlock final-no-redT elim: final-threadE)
    from aok have lock-actions-ok (locks s $ l) t (¶ta¦l $ l)
      by(auto simp add: lock-ok-las-def)
    with tt' ‹has-lock (locks s $ l) t'› s'
    have hl't': has-lock (locks s' $ l) t' by(auto)
    moreover note ‹t' ≠ t''›
    moreover from ‹t' ∈ deadlocked s ∪ final-threads s›
    have t' ∈ (deadlocked s ∪ deadlocked s' ∪ final-threads s')
      using subset by blast
    ultimately show ?thesis unfolding ‹lt = Inl l› ..
  next
    case (join t')
    note t'dead = ‹t' ∈ deadlocked s ∪ final-threads s›
    with Red have tt': t ≠ t'
      by(auto dest: red-no-deadlock final-no-redT elim: final-threadE)
    note nftt' = ‹not-final-thread s t'›
    from t'dead Red aok s' tt' have ts't': thr s' t' = thr s t'
      by(auto elim!: deadlocked-thread-exists final-threadE intro: redT-updTs-Some)
    from nftt' have thr s t' ≠ None by auto
    with nftt' t'dead have t' ∈ deadlocked s
      by(simp add: not-final-thread-final-thread-conv[symmetric])
    hence not-final-thread s' t'
    proof(cases rule: deadlocked-elims)
      case (lock x'')
      from ‹t' ⊢ ⟨x'', shr s⟩ ≀› have ¬ final x''
        by(auto elim: must-syncE dest: final-no-red)
      with ‹thr s t' = ⌊(x'', no-wait-locks)⌋› ts't' show ?thesis
        by(auto intro: not-final-thread.intros)
    next
      case (wait x'' ln'')
      from ‹¬ final x› tst ‹all-final-except s (deadlocked s)›
      have t ∈ deadlocked s by(fastforce dest: all-final-exceptD simp add: not-final-thread-iff)
      with Red have False by(auto dest: red-no-deadlock)
      thus ?thesis ..
    next
      case (acquire x'' ln'' l'' T'')
      from ‹thr s t' = ⌊(x'', ln'')⌋› ‹0 < ln'' $ l''› ts't'
      show ?thesis by(auto intro: not-final-thread.intros(2))
```

```
        qed
        moreover from t'dead subset have t' ∈ deadlocked s ∪ deadlocked s' ∪ final-threads s' ..
        ultimately show ?thesis unfolding ‹lt = Inr (Inl t')› ..
      next
        case (interrupt t')
        from tst red aok have not-final-thread s t
       by(auto simp add: wset-actions-ok-def not-final-thread-iff split: if-split-asm dest: final-no-red)
        with ‹all-final-except s (deadlocked s ∪ final-threads s)›
        have t ∈ deadlocked s ∪ final-threads s by(rule all-final-exceptD)
        moreover have t ∉ deadlocked s using Red by(blast dest: red-no-deadlock)
         moreover have ¬ final-thread s t using red tst by(auto simp add: final-thread-def dest:
final-no-red)
        ultimately have False by blast
        thus ?thesis ..
      qed
      with lt have ∃ lt∈LT. must-wait s' t'' lt (deadlocked s ∪ deadlocked s' ∪ final-threads s') by
blast }
    moreover have wset s' t'' = None using s' t''t ‹wset s t'' = None›
      by(auto intro: redT-updWs-None-implies-None)
    ultimately show ?thesis by(auto)
  next
    case (wait x ln)
    from ‹all-final-except s (deadlocked s)› nafe have False by simp
    thus ?thesis by simp
  next
    case (acquire X ln l T)
    from t''t Red ‹thr s t'' = ⌊(X, ln)⌋› s'
    have es't'': thr s' t'' = ⌊(X, ln)⌋
      by(cases s)(auto dest: redT-ts-Some-inv)
    moreover
    from ‹T ∈ deadlocked s ∨ final-thread s T›
    have T ≠ t
    proof(rule disjE)
      assume T ∈ deadlocked s
      with Red show ?thesis by(auto dest: red-no-deadlock)
    next
      assume final-thread s T
      with Red show ?thesis
        by(auto dest!: final-no-redT simp add: final-thread-def)
    qed
    with s' tst Red ‹has-lock (locks s $ l) T› have has-lock (locks s' $ l) T
      by −(cases s, auto dest: redT-has-lock-inv[THEN iffD2])
    moreover
    from s' ‹T ≠ t› have wset: wset s T = None ⟹ wset s' T = None
      by(auto intro: redT-updWs-None-implies-None)
    { fix x
      assume thr s T = ⌊(x, no-wait-locks)⌋
      with ‹T ≠ t› Red s' aok tst have thr s' T = ⌊(x, no-wait-locks)⌋
        by(auto intro: redT-updTs-Some) }
    moreover
    hence final-thread s T ⟹ final-thread s' T
      by(auto simp add: final-thread-def intro: wset)
    moreover from ‹¬ waiting (wset s t'')› s' t''t
    have ¬ waiting (wset s' t'')
```

```
      by(auto simp add: redT-updWs-None-implies-None redT-updWs-PostWS-imp-PostWS not-waiting-iff)
      ultimately have ?Acquire
        using ‹0 < ln $ l› ‹t'' ≠ T› ‹T ∈ deadlocked s ∨ final-thread s T› by(auto)
      thus ?thesis by simp
    qed
  qed
next
  case (redT-acquire x n ln)
  hence [simp]: ta = (K$ [], [], [], [], [], convert-RA ln)
    and s': s' = (acquire-all (locks s) t ln, ((thr s)(t ↦ (x, no-wait-locks))), shr s), wset s, interrupts
s)
    and tst: thr s t = ⌊(x, ln)⌋
    and wst: ¬ waiting (wset s t) by auto
  from t'dead show ?thesis
  proof(coinduct)
    case (deadlocked t'')
    note t''dead = this
    with Red have t''t: t'' ≠ t
      by(auto dest: red-no-deadlock)
    from t''dead show ?case
    proof(cases rule: deadlocked-elims)
      case (lock X)
      note clnml = ‹⋀LT. t'' ⊢ ⟨X, shr s⟩ LT ≀ ⟹ ∃ lt ∈ LT. must-wait s t'' lt (deadlocked s ∪
final-threads s)›
      note tst'' = ‹thr s t'' = ⌊(X, no-wait-locks)⌋›
      with s' t''t have ts't'': thr s' t'' = ⌊(X, no-wait-locks)⌋ by simp
      moreover
      { fix LT
        assume t'' ⊢ ⟨X, shr s'⟩ LT ≀
        hence t'' ⊢ ⟨X, shr s⟩ LT ≀ using s' by simp
       then obtain lt where lt: lt ∈ LT and hlnft: must-wait s t'' lt (deadlocked s ∪ final-threads s)
          by(blast dest: clnml)
        from hlnft have must-wait s' t'' lt (deadlocked s ∪ deadlocked s' ∪ final-threads s')
        proof(cases rule: must-wait-elims)
          case (lock l' T)
          from ‹has-lock (locks s $ l') T› s'
          have has-lock (locks s' $ l') T
            by(auto intro: has-lock-has-lock-acquire-locks)
          moreover note ‹T ≠ t''›
          moreover from ‹T ∈ deadlocked s ∪ final-threads s›
          have T ∈ deadlocked s ∪ deadlocked s' ∪ final-threads s' using subset by blast
          ultimately show ?thesis unfolding ‹lt = Inl l'› ..
        next
          case (join T)
          from ‹not-final-thread s T› have thr s T ≠ None
            by(auto simp add: not-final-thread-iff)
          moreover
          from ‹T ∈ deadlocked s ∪ final-threads s›
          have T ≠ t
          proof
            assume T ∈ deadlocked s
            with Red show ?thesis by(auto dest: red-no-deadlock)
          next
            assume T ∈ final-threads s
```

```
          with ‹0 < ln $ n› tst show ?thesis
            by(auto simp add: final-thread-def)
        qed
        ultimately have not-final-thread s' T using ‹not-final-thread s T› s'
          by(auto simp add: not-final-thread-iff)
        moreover from ‹T ∈ deadlocked s ∪ final-threads s›
        have T ∈ deadlocked s ∪ deadlocked s' ∪ final-threads s' using subset by blast
        ultimately show ?thesis unfolding ‹lt = Inr (Inl T)› ..
      next
        case (interrupt T)
        from tst wst ‹0 < ln $ n› have not-final-thread s t
          by(auto simp add: waiting-def not-final-thread-iff)
        with ‹all-final-except s (deadlocked s ∪ final-threads s)›
        have t ∈ deadlocked s ∪ final-threads s by(rule all-final-exceptD)
        moreover have t ∉ deadlocked s using Red by(blast dest: red-no-deadlock)
       moreover have ¬ final-thread s t using tst ‹0 < ln $ n› by(auto simp add: final-thread-def)
        ultimately have False by blast
        thus ?thesis ..
      qed
      with lt have ∃ lt∈LT. must-wait s' t'' lt (deadlocked s ∪ deadlocked s' ∪ final-threads s') by
blast }
     moreover from ‹wset s t'' = None› s' have wset s' t'' = None by simp
      ultimately show ?thesis using ‹thr s t'' = ⌊(X, no-wait-locks)⌋› ‹t'' ⊢ ⟨X, shr s⟩ ↓› s' by
fastforce
    next
      case (wait X LN)
      have all-final-except s' (deadlocked s)
      proof(rule all-final-exceptI)
        fix T
        assume not-final-thread s' T
        hence not-final-thread s T using wst tst s'
          by(auto simp add: not-final-thread-iff split: if-split-asm)
        with ‹all-final-except s (deadlocked s)› ‹thr s t = ⌊(x, ln)⌋›
        show T ∈ deadlocked s by−(erule all-final-exceptD)
      qed
      hence all-final-except s' (deadlocked s ∪ deadlocked s')
        by(rule all-final-except-mono') blast
      with t''t ‹thr s t'' = ⌊(X, LN)⌋› ‹waiting (wset s t'')› s'
      have ?Wait by simp
      thus ?thesis by simp
    next
      case (acquire X LN l T)
      from ‹thr s t'' = ⌊(X, LN)⌋› t''t s'
      have thr s' t'' = ⌊(X, LN)⌋ by(simp)
      moreover from ‹T ∈ deadlocked s ∨ final-thread s T› s' tst
      have T ∈ deadlocked s ∨ final-thread s' T
        by(clarsimp simp add: final-thread-def)
      moreover from ‹has-lock (locks s $ l) T› s'
      have has-lock (locks s' $ l) T
        by(auto intro: has-lock-has-lock-acquire-locks)
      moreover have ¬ waiting (wset s' t'') using ‹¬ waiting (wset s t'')› s' by simp
      ultimately show ?thesis using ‹0 < LN $ l› ‹t'' ≠ T› by blast
    qed
  qed
```

qed
qed

corollary *RedT-deadlocked-subset*:
assumes *wfs*: $s \in$ *wf-state*
and *Red*: $s\ -\triangleright ttas\rightarrow *\ s'$
shows *deadlocked* $s \subseteq$ *deadlocked* s'
using *Red*
apply(*induct rule*: *RedT-induct'*)
apply(*unfold RedT-def*)
apply(*blast dest*: *invariant3p-rtrancl3p*[*OF invariant3p-wf-state - wfs*] *redT-deadlocked-subset*)+
done

end

end

1.13 Progress theorem for the multithreaded semantics

theory *FWProgress*
imports
FWDeadlock
begin

locale *progress* = *multithreaded final r convert-RA*
for *final* :: $'x \Rightarrow$ *bool*
and *r* :: $('l,'t,'x,'m,'w,'o)$ *semantics* $(\langle\text{-} \vdash \text{-} -\text{-}\rightarrow \text{-}\rangle\ [50,0,0,50]\ 80)$
and *convert-RA* :: $'l$ *released-locks* $\Rightarrow$ $'o$ *list*
+
fixes *wf-state* :: $('l,'t,'x,'m,'w)$ *state set*
assumes *wf-stateD*: $s \in$ *wf-state* $\Longrightarrow$ *lock-thread-ok* (*locks s*) (*thr s*) $\wedge$ *wset-final-ok* (*wset s*) (*thr s*)
and *wf-red*:
$[\![\ s \in$ *wf-state*; *thr s t* $= \lfloor(x,$ *no-wait-locks*$)\rfloor$;
$t \vdash (x,$ *shr s*$) -ta\rightarrow (x',\ m')$; $\neg$ *waiting* (*wset s t*) $]\!]$
$\Longrightarrow \exists ta'\ x'\ m'.\ t \vdash (x,$ *shr s*$) -ta'\rightarrow (x',\ m') \wedge$ (*actions-ok s t ta'* $\vee$ *actions-ok' s t ta'* $\wedge$ *actions-subset ta' ta*)

and *red-wait-set-not-final*:
$[\![\ s \in$ *wf-state*; *thr s t* $= \lfloor(x,$ *no-wait-locks*$)\rfloor$;
$t \vdash (x,$ *shr s*$) -ta\rightarrow (x',\ m')$; $\neg$ *waiting* (*wset s t*); *Suspend w* $\in$ *set* $\{\!|ta|\!\}_w$ $]\!]$
$\Longrightarrow \neg$ *final* x'

and *wf-progress*:
$[\![\ s \in$ *wf-state*; *thr s t* $= \lfloor(x,$ *no-wait-locks*$)\rfloor$; $\neg$ *final x* $]\!]$
$\Longrightarrow \exists ta\ x'\ m'.\ t \vdash \langle x,$ *shr s*$\rangle -ta\rightarrow \langle x',\ m'\rangle$

and *ta-Wakeup-no-join-no-lock-no-interrupt*:
$[\![\ s \in$ *wf-state*; *thr s t* $= \lfloor(x,$ *no-wait-locks*$)\rfloor$; $t \vdash xm\ -ta\rightarrow xm'$; *Notified* $\in$ *set* $\{\!|ta|\!\}_w \vee$ *WokenUp* $\in$ *set* $\{\!|ta|\!\}_w$ $]\!]$
$\Longrightarrow$ *collect-waits ta* $= \{\}$

and *ta-satisfiable*:
$[\![\ s \in$ *wf-state*; *thr s t* $= \lfloor(x,$ *no-wait-locks*$)\rfloor$; $t \vdash \langle x,$ *shr s*$\rangle -ta\rightarrow \langle x',\ m'\rangle$ $]\!]$

$\Longrightarrow \exists s'.\ actions\text{-}ok\ s'\ t\ ta$
begin

lemma *wf-redE*:
assumes $s \in wf\text{-}state\ thr\ s\ t = \lfloor (x,\ no\text{-}wait\text{-}locks) \rfloor$
and $t \vdash \langle x,\ shr\ s \rangle\ -ta\rightarrow\ \langle x'',\ m'' \rangle\ \neg\ waiting\ (wset\ s\ t)$
obtains $ta'\ x'\ m'$
where $t \vdash \langle x,\ shr\ s \rangle\ -ta'\rightarrow\ \langle x',\ m' \rangle\ actions\text{-}ok'\ s\ t\ ta'\ actions\text{-}subset\ ta'\ ta$
| $ta'\ x'\ m'$ **where** $t \vdash \langle x,\ shr\ s \rangle\ -ta'\rightarrow\ \langle x',\ m' \rangle\ actions\text{-}ok\ s\ t\ ta'$
using *wf-red*[*OF assms*] **by** *blast*

lemma *wf-progressE*:
assumes $s \in wf\text{-}state$
and $thr\ s\ t = \lfloor (x,\ no\text{-}wait\text{-}locks) \rfloor\ \neg\ final\ x$
obtains $ta\ x'\ m'$ **where** $t \vdash \langle x,\ shr\ s \rangle\ -ta\rightarrow\ \langle x',\ m' \rangle$
using *assms*
by(*blast dest*: *wf-progress*)

lemma *wf-progress-satisfiable*:
$[\![\ s \in wf\text{-}state;\ thr\ s\ t = \lfloor (x,\ no\text{-}wait\text{-}locks) \rfloor;\ \neg\ final\ x\ ]\!]$
$\Longrightarrow \exists ta\ x'\ m'\ s'.\ t \vdash \langle x,\ shr\ s \rangle\ -ta\rightarrow\ \langle x',\ m' \rangle\ \wedge\ actions\text{-}ok\ s'\ t\ ta$
apply(*frule* (*2*) *wf-progress*)
apply(*blast dest*: *ta-satisfiable*)
done

theorem *redT-progress*:
assumes *wfs*: $s \in wf\text{-}state$
and *ndead*: $\neg\ deadlock\ s$
shows $\exists t'\ ta'\ s'.\ s\ -t' \triangleright ta'\rightarrow\ s'$
proof −
from *wfs* **have** *lok*: *lock-thread-ok* (*locks s*) (*thr s*)
and *wfin*: *wset-final-ok* (*wset s*) (*thr s*)
by(*auto dest*: *wf-stateD*)
from *ndead*
have $\exists t\ x\ ln\ l.\ thr\ s\ t = \lfloor (x,\ ln) \rfloor\ \wedge$
$(wset\ s\ t = None\ \wedge\ ln = no\text{-}wait\text{-}locks\ \wedge\ \neg\ final\ x\ \wedge\ (\exists LT.\ t \vdash \langle x,\ shr\ s \rangle\ LT\ \wr\ \wedge\ (\forall lt \in LT.$
$\neg\ must\text{-}wait\ s\ t\ lt\ (dom\ (thr\ s)))) \vee$
$\neg\ waiting\ (wset\ s\ t) \wedge ln\ \$\ l > 0\ \wedge\ (\forall l.\ ln\ \$\ l > 0 \longrightarrow may\text{-}lock\ (locks\ s\ \$\ l)\ t) \vee$
$(\exists w.\ ln = no\text{-}wait\text{-}locks \wedge wset\ s\ t = \lfloor PostWS\ w \rfloor))$
by(*rule contrapos-np*)(*blast intro*!: *all-waiting-implies-deadlock*[*OF lok*] *intro*: *must-syncI*[*OF wf-progress-satisfiable*[*OF wfs*]])
then obtain $t\ x\ ln\ l$
where *tst*: $thr\ s\ t = \lfloor (x,\ ln) \rfloor$
and *a*: $wset\ s\ t = None\ \wedge\ ln = no\text{-}wait\text{-}locks\ \wedge\ \neg\ final\ x\ \wedge$
$(\exists LT.\ t \vdash \langle x,\ shr\ s \rangle\ LT\ \wr\ \wedge\ (\forall lt \in LT.\ \neg\ must\text{-}wait\ s\ t\ lt\ (dom\ (thr\ s)))) \vee$
$\neg\ waiting\ (wset\ s\ t) \wedge ln\ \$\ l > 0\ \wedge\ (\forall l.\ ln\ \$\ l > 0 \longrightarrow may\text{-}lock\ (locks\ s\ \$\ l)\ t) \vee$
$(\exists w.\ ln = no\text{-}wait\text{-}locks \wedge wset\ s\ t = \lfloor PostWS\ w \rfloor)$
by *blast*
from *a* **have** *cases*[*case-names normal acquire wakeup*]:
$\bigwedge thesis.$
$[\![\ \bigwedge LT.\ [\![\ wset\ s\ t = None;\ ln = no\text{-}wait\text{-}locks;\ \neg\ final\ x;\ t \vdash \langle x,\ shr\ s \rangle\ LT\ \wr;$
$\bigwedge lt.\ lt \in LT \Longrightarrow \neg\ must\text{-}wait\ s\ t\ lt\ (dom\ (thr\ s))\ ]\!] \Longrightarrow thesis;$
$[\![\ \neg\ waiting\ (wset\ s\ t);\ ln\ \$\ l > 0;\ \bigwedge l.\ ln\ \$\ l > 0 \Longrightarrow may\text{-}lock\ (locks\ s\ \$\ l)\ t\ ]\!] \Longrightarrow thesis;$
$\bigwedge w.\ [\![\ ln = no\text{-}wait\text{-}locks;\ wset\ s\ t = \lfloor PostWS\ w \rfloor\ ]\!] \Longrightarrow thesis\ ]\!] \Longrightarrow thesis$

```
  by auto
show ?thesis
proof(cases rule: cases)
  case (normal LT)
  note [simp] = ‹ln = no-wait-locks›
    and nfine' = ‹¬ final x›
    and cl' = ‹t ⊢ ⟨x, shr s⟩ LT ↓›
    and mw = ‹⋀lt. lt∈LT ⟹ ¬ must-wait s t lt (dom (thr s))›
  from tst nfine' obtain x'' m'' ta'
    where red: t ⊢ ⟨x, shr s⟩ −ta'→ ⟨x'', m''⟩
    by(auto intro: wf-progressE[OF wfs])
  from cl'
  have ∃ta''' x''' m'''. t ⊢ ⟨x, shr s⟩ −ta'''→ ⟨x''', m'''⟩ ∧
        LT = collect-waits ta'''
    by (fastforce elim!: can-syncE)
  then obtain ta''' x''' m'''
    where red'': t ⊢ ⟨x, shr s⟩ −ta'''→ ⟨x''', m'''⟩
    and L: LT = collect-waits ta'''
    by blast
  from ‹wset s t = None› have ¬ waiting (wset s t) by(simp add: not-waiting-iff)
  with tst obtain ta'' x'' m''
    where red': t ⊢ ⟨x, shr s⟩ −ta''→ ⟨x'', m''⟩
    and aok': actions-ok s t ta'' ∨ actions-ok' s t ta'' ∧ actions-subset ta'' ta'''
    by −(rule wf-redE[OF wfs - red''], auto)
  from aok' have actions-ok s t ta''
  proof
    assume actions-ok' s t ta'' ∧ actions-subset ta'' ta'''
    hence aok': actions-ok' s t ta'' and aos: actions-subset ta'' ta''' by simp-all

    { fix l
      assume Inl l ∈ LT
      { fix t'
        assume t ≠ t'
        have ¬ has-lock (locks s $ l) t'
        proof
          assume has-lock (locks s $ l) t'
          moreover with lok have thr s t' ≠ None by(auto dest: lock-thread-okD)
          ultimately have must-wait s t (Inl l) (dom (thr s)) using ‹t ≠ t'› by(auto)
          moreover from ‹Inl l ∈ LT› have ¬ must-wait s t (Inl l) (dom (thr s)) by(rule mw)
          ultimately show False by contradiction
        qed }
      hence may-lock (locks s $ l) t
        by−(rule classical, auto simp add: not-may-lock-conv) }
    note mayl = this
    { fix t'
      assume t'LT: Inr (Inl t') ∈ LT
      hence ¬ not-final-thread s t' ∧ t' ≠ t
      proof(cases t' = t)
        case False with t'LT mw L show ?thesis by(fastforce)
      next
        case True with tst mw[OF t'LT] nfine' L have False
          by(auto intro!: must-wait.intros simp add: not-final-thread-iff)
        thus ?thesis ..
      qed }
```

```
    note mayj = this
    { fix t'
      assume t': Inr (Inr t') ∈ LT
      from t' have ¬ must-wait s t (Inr (Inr t')) (dom (thr s)) by(rule mw)
      hence t' ∈ interrupts s
        by(rule contrapos-np)(fastforce intro: all-final-exceptI simp add: not-final-thread-iff) }
    note interrupt = this
    from aos L mayl
    have ⋀l. l ∈ collect-locks' {|ta''|}_l ⟹ may-lock (locks s $ l) t by auto
    with aok' have lock-ok-las (locks s) t {|ta''|}_l by(auto intro: lock-ok-las'-into-lock-on-las)
    moreover
    from mayj aos L
    have cond-action-oks s t {|ta''|}_c
      by(fastforce intro: may-join-cond-action-oks)
    moreover
    from ta-satisfiable[OF wfs tst[simplified] red']
    obtain is' where interrupt-actions-ok is' {|ta''|}_i by auto
    with interrupt aos aok' L have interrupt-actions-ok (interrupts s) {|ta''|}_i
      by(auto 5 2 intro: interrupt-actions-ok'-collect-interrupts-imp-interrupt-actions-ok)
    ultimately show actions-ok s t ta'' using aok' by auto
  qed
  moreover obtain ws'' where redT-updWs t (wset s) {|ta''|}_w ws''
    using redT-updWs-total[of t wset s {|ta''|}_w] ..
  then obtain s' where redT-upd s t ta'' x'' m'' s' by fastforce
  ultimately have s -t▷ta''→ s'
    using red' tst ‹wset s t = None› by(auto intro: redT-normal)
  thus ?thesis by blast
next
  case acquire
  hence may-acquire-all (locks s) t ln by(auto)
  with tst ‹¬ waiting (wset s t)› ‹0 < ln $ l›
  show ?thesis by(fastforce intro: redT-acquire)
next
  case (wakeup w)
  from ‹wset s t = ⌊PostWS w⌋›
  have ¬ waiting (wset s t) by(simp add: not-waiting-iff)
  from tst wakeup have tst: thr s t = ⌊(x, no-wait-locks)⌋ by simp
  from wakeup tst wfin have ¬ final x by(auto dest: wset-final-okD)
  from wf-progress[OF wfs tst this]
  obtain ta x' m' where red: t ⊢ ⟨x, shr s⟩ -ta→ ⟨x', m'⟩ by auto
  from wf-red[OF wfs tst red ‹¬ waiting (wset s t)›]
  obtain ta' x'' m''
    where red': t ⊢ ⟨x, shr s⟩ -ta'→ ⟨x'', m''⟩
    and aok': actions-ok s t ta' ∨ actions-ok' s t ta' ∧ actions-subset ta' ta by blast
  from aok' have actions-ok s t ta'
  proof
    assume actions-ok' s t ta' ∧ actions-subset ta' ta
    hence aok': actions-ok' s t ta'
      and subset: actions-subset ta' ta by simp-all
    from wakeup aok' have Notified ∈ set {|ta'|}_w ∨ WokenUp ∈ set {|ta'|}_w
      by(auto simp add: wset-actions-ok-def split: if-split-asm)
    from ta-Wakeup-no-join-no-lock-no-interrupt[OF wfs tst red' this]
    have no-join: collect-cond-actions {|ta'|}_c = {}
      and no-lock: collect-locks {|ta'|}_l = {}
```

and *no-interrupt*: *collect-interrupts* $\{\!|ta'|\!\}_i = \{\}$ **by** *auto*
from *no-lock* **have** *no-lock'*: *collect-locks'* $\{\!|ta'|\!\}_l = \{\}$
using *collect-locks'-subset-collect-locks*[*of* $\{\!|ta'|\!\}_l$] **by** *auto*
from *aok'* **have** *lock-ok-las'* (*locks s*) *t* $\{\!|ta'|\!\}_l$ **by** *auto*
hence *lock-ok-las* (*locks s*) *t* $\{\!|ta'|\!\}_l$
by(*rule lock-ok-las'-into-lock-on-las*)(*simp add*: *no-lock'*)
moreover from *subset aok' no-join* **have** *cond-action-oks s t* $\{\!|ta'|\!\}_c$
by(*auto intro*: *may-join-cond-action-oks*)
moreover from *ta-satisfiable*[*OF wfs tst*[*simplified*] *red'*]
obtain *is'* **where** *interrupt-actions-ok is'* $\{\!|ta'|\!\}_i$ **by** *auto*
with *aok' no-interrupt* **have** *interrupt-actions-ok* (*interrupts s*) $\{\!|ta'|\!\}_i$
by(*auto intro*: *interrupt-actions-ok'-collect-interrupts-imp-interrupt-actions-ok*)
ultimately show *actions-ok s t ta'* **using** *aok'* **by** *auto*
qed
moreover obtain *ws''* **where** *redT-updWs t* (*wset s*) $\{\!|ta'|\!\}_w$ *ws''*
using *redT-updWs-total*[*of t wset s* $\{\!|ta'|\!\}_w$] **..**
then obtain *s'* **where** *redT-upd s t ta' x'' m'' s'* **by** *fastforce*
ultimately have $s\ -t\triangleright ta'\rightarrow\ s'$ **using** *tst red' wakeup*
by(*auto intro*: *redT-normal*)
thus *?thesis* **by** *blast*
qed
qed

end

end

1.14 Lifting of thread-local properties to the multithreaded case

theory *FWLifting*
imports
FWWellform
begin

Lifting for properties that only involve thread-local state information and the shared memory.

definition
ts-ok :: $('t \Rightarrow 'x \Rightarrow 'm \Rightarrow bool) \Rightarrow ('l, 't, 'x)$ *thread-info* $\Rightarrow 'm \Rightarrow bool$
where
$\bigwedge ln.$ *ts-ok P ts m* $\equiv \forall t.$ *case* (*ts t*) *of None* $\Rightarrow$ *True* $\mid \lfloor(x, ln)\rfloor \Rightarrow P\ t\ x\ m$

lemma *ts-okI*:
$[\![\bigwedge t\ x\ ln.\ ts\ t = \lfloor(x, ln)\rfloor \Longrightarrow P\ t\ x\ m\ ]\!] \Longrightarrow$ *ts-ok P ts m*
by(*auto simp add*: *ts-ok-def*)

lemma *ts-okE*:
$[\![$ *ts-ok P ts m*; $[\![\bigwedge t\ x\ ln.\ ts\ t = \lfloor(x, ln)\rfloor \Longrightarrow P\ t\ x\ m\ ]\!] \Longrightarrow Q\ ]\!] \Longrightarrow Q$
by(*auto simp add*: *ts-ok-def*)

lemma *ts-okD*:
$\bigwedge ln.\ [\![$ *ts-ok P ts m*; *ts t* $= \lfloor(x, ln)\rfloor\ ]\!] \Longrightarrow P\ t\ x\ m$
by(*auto simp add*: *ts-ok-def*)

lemma *ts-ok-True* [*simp*]:
ts-ok (λ*t m x*. *True*) *ts m*
by(*auto intro*: *ts-okI*)

lemma *ts-ok-conj*:
ts-ok (λ*t x m*. *P t x m* ∧ *Q t x m*) = (λ*ts m*. *ts-ok P ts m* ∧ *ts-ok Q ts m*)
by(*auto intro*: *ts-okI intro*!: *ext dest*: *ts-okD*)

lemma *ts-ok-mono*:
⟦ *ts-ok P ts m*; ⋀*t x*. *P t x m* ⟹ *Q t x m* ⟧ ⟹ *ts-ok Q ts m*
by(*auto intro*!: *ts-okI dest*: *ts-okD*)

Lifting for properites, that also require additional data that does not change during execution

definition
ts-inv :: (′*i* ⇒ ′*t* ⇒ ′*x* ⇒ ′*m* ⇒ *bool*) ⇒ (′*t* ⇀ ′*i*) ⇒ (′*l*,′*t*,′*x*) *thread-info* ⇒ ′*m* ⇒ *bool*
where
⋀*ln*. *ts-inv P I ts m* ≡ ∀ *t*. *case* (*ts t*) *of None* ⇒ *True* | ⌊(*x*, *ln*)⌋ ⇒ ∃ *i*. *I t* = ⌊*i*⌋ ∧ *P i t x m*

lemma *ts-invI*:
⟦ ⋀*t x ln*. *ts t* = ⌊(*x*, *ln*)⌋ ⟹ ∃ *i*. *I t* = ⌊*i*⌋ ∧ *P i t x m* ⟧ ⟹ *ts-inv P I ts m*
by(*simp add*: *ts-inv-def*)

lemma *ts-invE*:
⟦ *ts-inv P I ts m*; ∀ *t x ln*. *ts t* = ⌊(*x*, *ln*)⌋ ⟶ (∃ *i*. *I t* = ⌊*i*⌋ ∧ *P i t x m*) ⟹ *R* ⟧ ⟹ *R*
by(*auto simp add*: *ts-inv-def*)

lemma *ts-invD*:
⋀*ln*. ⟦ *ts-inv P I ts m*; *ts t* = ⌊(*x*, *ln*)⌋ ⟧ ⟹ ∃ *i*. *I t* = ⌊*i*⌋ ∧ *P i t x m*
by(*auto simp add*: *ts-inv-def*)

Wellformedness properties for lifting

definition
ts-inv-ok :: (′*l*,′*t*,′*x*) *thread-info* ⇒ (′*t* ⇀ ′*i*) ⇒ *bool*
where
ts-inv-ok ts I ≡ ∀ *t*. *ts t* = *None* ⟷ *I t* = *None*

lemma *ts-inv-okI*:
(⋀*t*. *ts t* = *None* ⟷ *I t* = *None*) ⟹ *ts-inv-ok ts I*
by(*clarsimp simp add*: *ts-inv-ok-def*)

lemma *ts-inv-okI2*:
(⋀*t*. (∃ *v*. *ts t* = ⌊*v*⌋) ⟷ (∃ *v*. *I t* = ⌊*v*⌋)) ⟹ *ts-inv-ok ts I*
by(*force simp add*: *ts-inv-ok-def*)

lemma *ts-inv-okE*:
⟦ *ts-inv-ok ts I*; ∀ *t*. *ts t* = *None* ⟷ *I t* = *None* ⟹ *P* ⟧ ⟹ *P*
by(*force simp add*: *ts-inv-ok-def*)

lemma *ts-inv-okE2*:
⟦ *ts-inv-ok ts I*; ∀ *t*. (∃ *v*. *ts t* = ⌊*v*⌋) ⟷ (∃ *v*. *I t* = ⌊*v*⌋) ⟹ *P* ⟧ ⟹ *P*
by(*force simp add*: *ts-inv-ok-def*)

lemma *ts-inv-okD*:
ts-inv-ok ts I $\Longrightarrow$ (*ts t* = *None*) $\longleftrightarrow$ (*I t* = *None*)
by(*erule ts-inv-okE*, *blast*)

lemma *ts-inv-okD2*:
ts-inv-ok ts I $\Longrightarrow$ ($\exists$ *v*. *ts t* = $\lfloor$*v*$\rfloor$) $\longleftrightarrow$ ($\exists$ *v*. *I t* = $\lfloor$*v*$\rfloor$)
by(*erule ts-inv-okE2*, *blast*)

lemma *ts-inv-ok-conv-dom-eq*:
ts-inv-ok ts I $\longleftrightarrow$ (*dom ts* = *dom I*)
proof −
have *ts-inv-ok ts I* $\longleftrightarrow$ ($\forall$ *t*. *ts t* = *None* $\longleftrightarrow$ *I t* = *None*)
unfolding *ts-inv-ok-def* **by** *blast*
also have ... $\longleftrightarrow$ ($\forall$ *t*. *t* $\in$ − *dom ts* $\longleftrightarrow$ *t* $\in$ − *dom I*) **by**(*force*)
also have ... $\longleftrightarrow$ *dom ts* = *dom I* **by** *auto*
finally show *?thesis* **.**
qed

lemma *ts-inv-ok-upd-ts*:
⟦ *ts t* = $\lfloor$*x*$\rfloor$; *ts-inv-ok ts I* ⟧ $\Longrightarrow$ *ts-inv-ok* (*ts*(*t* $\mapsto$ *x*′)) *I*
by(*auto dest*!: *ts-inv-okD intro*!: *ts-inv-okI split*: *if-splits*)

lemma *ts-inv-upd-map-option*:
assumes *ts-inv P I ts m*
and $\bigwedge$*x ln*. *ts t* = $\lfloor$(*x*, *ln*)$\rfloor$ $\Longrightarrow$ *P* (*the* (*I t*)) *t* (*fst* (*f* (*x*, *ln*))) *m*
shows *ts-inv P I* (*ts*(*t* := (*map-option f* (*ts t*)))) *m*
using *assms*
by(*fastforce intro*!: *ts-invI split*: *if-split-asm dest*: *ts-invD*)

fun *upd-inv* :: (′*t* ⇀ ′*i*) ⇒ (′*i* ⇒ ′*t* ⇒ ′*x* ⇒ ′*m* ⇒ *bool*) ⇒ (′*t*,′*x*,′*m*) *new-thread-action* ⇒ (′*t* ⇀ ′*i*)
where
upd-inv I P (*NewThread t x m*) = *I*(*t* $\mapsto$ *SOME i*. *P i t x m*)
| *upd-inv I P* - = *I*

fun *upd-invs* :: (′*t* ⇀ ′*i*) ⇒ (′*i* ⇒ ′*t* ⇒ ′*x* ⇒ ′*m* ⇒ *bool*) ⇒ (′*t*,′*x*,′*m*) *new-thread-action list* ⇒ (′*t* ⇀ ′*i*)
where
upd-invs I P [] = *I*
| *upd-invs I P* (*ta*#*tas*) = *upd-invs* (*upd-inv I P ta*) *P tas*

lemma *upd-invs-append* [*simp*]:
upd-invs I P (*xs* @ *ys*) = *upd-invs* (*upd-invs I P xs*) *P ys*
by(*induct xs arbitrary*: *I*)(*auto*)

lemma *ts-inv-ok-upd-inv*′:
ts-inv-ok ts I $\Longrightarrow$ *ts-inv-ok* (*redT-updT*′ *ts ta*) (*upd-inv I P ta*)
by(*cases ta*)(*auto intro*!: *ts-inv-okI elim*: *ts-inv-okD del*: *iffI*)

lemma *ts-inv-ok-upd-invs*′:
ts-inv-ok ts I $\Longrightarrow$ *ts-inv-ok* (*redT-updTs*′ *ts tas*) (*upd-invs I P tas*)
proof(*induct tas arbitrary*: *ts I*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons TA TAS TS I*)

```
  note IH = ‹⋀ts I. ts-inv-ok ts I ⟹ ts-inv-ok (redT-updTs' ts TAS) (upd-invs I P TAS)›
  note esok = ‹ts-inv-ok TS I›
  from esok have ts-inv-ok (redT-updT' TS TA) (upd-inv I P TA)
    by −(rule ts-inv-ok-upd-inv')
  hence ts-inv-ok (redT-updTs' (redT-updT' TS TA) TAS) (upd-invs (upd-inv I P TA) P TAS)
    by (rule IH)
  thus ?case by simp
qed

lemma ts-inv-ok-upd-inv:
 ts-inv-ok ts I ⟹ ts-inv-ok (redT-updT ts ta) (upd-inv I P ta)
apply(cases ta)
apply(auto intro!: ts-inv-okI elim: ts-inv-okD del: iffI)
done

lemma ts-inv-ok-upd-invs:
  ts-inv-ok ts I ⟹ ts-inv-ok (redT-updTs ts tas) (upd-invs I P tas)
proof(induct tas arbitrary: ts I)
  case Nil thus ?case by simp
next
  case (Cons TA TAS TS I)
  note IH = ‹⋀ts I. ts-inv-ok ts I ⟹ ts-inv-ok (redT-updTs ts TAS) (upd-invs I P TAS)›
  note esok = ‹ts-inv-ok TS I›
  from esok have ts-inv-ok (redT-updT TS TA) (upd-inv I P TA)
    by −(rule ts-inv-ok-upd-inv)
  hence ts-inv-ok (redT-updTs (redT-updT TS TA) TAS) (upd-invs (upd-inv I P TA) P TAS)
    by (rule IH)
  thus ?case by simp
qed

lemma ts-inv-ok-inv-ext-upd-inv:
  ⟦ ts-inv-ok ts I; thread-ok ts ta ⟧ ⟹ I ⊆m upd-inv I P ta
by(cases ta)(auto intro!: map-le-same-upd dest: ts-inv-okD)

lemma ts-inv-ok-inv-ext-upd-invs:
  ⟦ ts-inv-ok ts I; thread-oks ts tas⟧
  ⟹ I ⊆m upd-invs I P tas
proof(induct tas arbitrary: ts I)
  case Nil thus ?case by simp
next
  case (Cons TA TAS TS I)
  note IH = ‹⋀ts I. ⟦ ts-inv-ok ts I; thread-oks ts TAS⟧ ⟹ I ⊆m upd-invs I P TAS›
  note esinv = ‹ts-inv-ok TS I›
  note cct = ‹thread-oks TS (TA # TAS)›
  from esinv cct have I ⊆m upd-inv I P TA
    by(auto intro: ts-inv-ok-inv-ext-upd-inv)
  also from esinv cct have ts-inv-ok (redT-updT' TS TA) (upd-inv I P TA)
    by(auto intro: ts-inv-ok-upd-inv')
  with cct have upd-inv I P TA ⊆m upd-invs (upd-inv I P TA) P TAS
    by(auto intro: IH)
  finally show ?case by simp
qed

lemma upd-invs-Some:
```

⟦ *thread-oks ts tas*; *I t* = ⌊*i*⌋; *ts t* = ⌊*x*⌋ ⟧ ⟹ *upd-invs I Q tas t* = ⌊*i*⌋
proof(*induct tas arbitrary*: *ts I*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons TA TAS TS I*)
note *IH* = ‹⋀*ts I*. ⟦*thread-oks ts TAS*; *I t* = ⌊*i*⌋; *ts t* = ⌊*x*⌋⟧ ⟹ *upd-invs I Q TAS t* = ⌊*i*⌋›
note *cct* = ‹*thread-oks TS* (*TA* # *TAS*)›
note *it* = ‹*I t* = ⌊*i*⌋›
note *est* = ‹*TS t* = ⌊*x*⌋›
from *cct* **have** *cctta*: *thread-ok TS TA*
and *ccttas*: *thread-oks* (*redT-updT′ TS TA*) *TAS* **by** *auto*
from *cctta it est* **have** *upd-inv I Q TA t* = ⌊*i*⌋
by(*cases TA*, *auto*)
moreover
have *redT-updT′ TS TA t* = ⌊*x*⌋ **using** *cctta est*
by − (*rule redT-updT′-Some*)
ultimately have *upd-invs* (*upd-inv I Q TA*) *Q TAS t* = ⌊*i*⌋ **using** *ccttas*
by −(*erule IH*)
thus *?case* **by** *simp*
qed

lemma *upd-inv-Some-eq*:
⟦ *thread-ok ts ta*; *ts t* = ⌊*x*⌋ ⟧ ⟹ *upd-inv I Q ta t* = *I t*
by(*cases ta*, *auto*)

lemma *upd-invs-Some-eq*: ⟦ *thread-oks ts tas*; *ts t* = ⌊*x*⌋ ⟧ ⟹ *upd-invs I Q tas t* = *I t*
proof(*induct tas arbitrary*: *ts I*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons TA TAS TS I*)
note *IH* = ‹⋀*ts I*. ⟦*thread-oks ts TAS*; *ts t* = ⌊*x*⌋⟧ ⟹ *upd-invs I Q TAS t* = *I t*›
note *cct* = ‹*thread-oks TS* (*TA* # *TAS*)›
note *est* = ‹*TS t* = ⌊*x*⌋›
from *cct est* **have** *upd-invs* (*upd-inv I Q TA*) *Q TAS t* = *upd-inv I Q TA t*
apply(*clarsimp*)
apply(*erule IH*)
by(*rule redT-updT′-Some*)
also from *cct est* **have** . . . = *I t*
by(*auto elim*: *upd-inv-Some-eq*)
finally show *?case* **by** *simp*
qed

lemma *SOME-new-thread-upd-invs*:
assumes *Qsome*: *Q* (*SOME i*. *Q i t x m*) *t x m*
and *nt*: *NewThread t x m* ∈ *set tas*
and *cct*: *thread-oks ts tas*
shows ∃ *i*. *upd-invs I Q tas t* = ⌊*i*⌋ ∧ *Q i t x m*
proof(*rule exI*[**where** *x*=*SOME i*. *Q i t x m*])
from *nt cct* **have** *upd-invs I Q tas t* = ⌊*SOME i*. *Q i t x m*⌋
proof(*induct tas arbitrary*: *ts I*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons TA TAS TS I*)
note *IH* = ‹⋀*ts I*. ⟦ *NewThread t x m* ∈ *set TAS*; *thread-oks ts TAS* ⟧ ⟹ *upd-invs I Q TAS t* =

⌊*SOME i. Q i t x m*⌋›
note *nt* = ‹*NewThread t x m* ∈ *set* (*TA* # *TAS*)›
note *cct* = ‹*thread-oks TS* (*TA* # *TAS*)›
{ assume *nt'*: *NewThread t x m* ∈ *set TAS*
from *cct* **have** *?case*
apply(*clarsimp*)
by(*rule IH*[*OF nt'*]) **}**
moreover
{ assume *ta*: *TA* = *NewThread t x m*
with *cct* **have** *rup*: *redT-updT' TS TA t* = ⌊(*undefined*, *no-wait-locks*)⌋
by(*simp*)
from *cct* **have** *cctta*: *thread-oks* (*redT-updT' TS TA*) *TAS* **by** *simp*
from *ta* **have** *upd-inv I Q TA t* = ⌊*SOME i. Q i t x m*⌋
by(*simp*)
hence *?case*
by(*clarsimp simp add*: *upd-invs-Some-eq*[*OF cctta*, *OF rup*]) **}**
ultimately show *?case* **using** *nt* **by** *auto*
qed
with *Qsome* **show** *upd-invs I Q tas t* = ⌊*SOME i. Q i t x m*⌋ ∧ *Q* (*SOME i. Q i t x m*) *t x m*
by(*simp*)
qed

lemma *ts-ok-into-ts-inv-const*:
assumes *ts-ok P ts m*
obtains *I* **where** *ts-inv* (λ-. *P*) *I ts m*
proof −
from *assms* **have** *ts-inv* (λ-. *P*) (λ*t*. *if t* ∈ *dom ts then Some undefined else None*) *ts m*
by(*auto intro*!: *ts-invI dest*: *ts-okD*)
thus *thesis* **by**(*rule that*)
qed

lemma *ts-inv-const-into-ts-ok*:
ts-inv (λ-. *P*) *I ts m* ⟹ *ts-ok P ts m*
by(*auto intro*!: *ts-okI dest*: *ts-invD*)

lemma *ts-inv-into-ts-ok-Ex*:
ts-inv Q I ts m ⟹ *ts-ok* (λ*t x m*. ∃ *i*. *Q i t x m*) *ts m*
by(*rule ts-okI*)(*blast dest*: *ts-invD*)

lemma *ts-ok-Ex-into-ts-inv*:
ts-ok (λ*t x m*. ∃ *i*. *Q i t x m*) *ts m* ⟹ ∃ *I*. *ts-inv Q I ts m*
by(*rule exI*[**where** *x*=λ*t*. ⌊*SOME i. Q i t* (*fst* (*the* (*ts t*))) *m*⌋])(*auto 4 4 dest*: *ts-okD intro*: *someI intro*: *ts-invI*)

lemma *Ex-ts-inv-conv-ts-ok*:
(∃ *I*. *ts-inv Q I ts m*) ⟷ (*ts-ok* (λ*t x m*. ∃ *i*. *Q i t x m*) *ts m*)
by(*auto dest*: *ts-inv-into-ts-ok-Ex ts-ok-Ex-into-ts-inv*)

end

1.15 Labelled transition systems

theory *LTS*

imports
../Basic/Auxiliary
Coinductive.TLList
begin

unbundle *no floor-ceiling-syntax*

lemma *rel-option-mono*:
⟦ *rel-option R x y*; ⋀*x y*. *R x y* ⟹ *R′ x y* ⟧ ⟹ *rel-option R′ x y*
by(*cases x*)(*case-tac* [!] *y*, *auto*)

lemma *nth-concat-conv*:
n < *length* (*concat xss*)
⟹ ∃ *m n′*. *concat xss* ! *n* = (*xss* ! *m*) ! *n′* ∧ *n′* < *length* (*xss* ! *m*) ∧
m < *length xss* ∧ *n* = (∑ *i*<*m*. *length* (*xss* ! *i*)) + *n′*
using *lnth-lconcat-conv*[*of n llist-of* (*map llist-of xss*)]
sum-comp-morphism[**where** *h* = *enat* **and** *g* = λ*i*. *length* (*xss* ! *i*)]
by(*clarsimp simp add*: *lconcat-llist-of zero-enat-def*[*symmetric*]) *blast*

definition *flip* :: (′*a* ⇒ ′*b* ⇒ ′*c*) ⇒ ′*b* ⇒ ′*a* ⇒ ′*c*
where *flip f* = (λ*b a*. *f a b*)

Create a dynamic list *flip-simps* of theorems for flip

ML ‹
structure FlipSimpRules = *Named-Thms*
(
val name = @{*binding flip-simps*}
val description = *Simplification rules for flip in bisimulations*
)
›
setup ‹*FlipSimpRules.setup*›

lemma *flip-conv* [*flip-simps*]: *flip f b a* = *f a b*
by(*simp add*: *flip-def*)

lemma *flip-flip* [*flip-simps*, *simp*]: *flip* (*flip f*) = *f*
by(*simp add*: *flip-def*)

lemma *list-all2-flip* [*flip-simps*]: *list-all2* (*flip P*) *xs ys* = *list-all2 P ys xs*
unfolding *flip-def list-all2-conv-all-nth* **by** *auto*

lemma *llist-all2-flip* [*flip-simps*]: *llist-all2* (*flip P*) *xs ys* = *llist-all2 P ys xs*
unfolding *flip-def llist-all2-conv-all-lnth* **by** *auto*

lemma *rtranclp-flipD*:
assumes (*flip r*)^** *x y*
shows *r*^** *y x*
using *assms*
by(*induct rule*: *rtranclp-induct*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl simp add*: *flip-conv*)

lemma *rtranclp-flip* [*flip-simps*]:
(*flip r*)^** = *flip r*^**
by(*auto intro*!: *ext simp add*: *flip-conv intro*: *rtranclp-flipD*)

lemma *rel-prod-flip* [*flip-simps*]:
rel-prod (*flip R*) (*flip S*) = *flip* (*rel-prod R S*)
by(*auto intro*!: *ext simp add*: *flip-def*)

lemma *rel-option-flip* [*flip-simps*]:
rel-option (*flip R*) = *flip* (*rel-option R*)
by(*simp add*: *fun-eq-iff rel-option-iff flip-def*)

lemma *tllist-all2-flip* [*flip-simps*]:
tllist-all2 (*flip P*) (*flip Q*) *xs ys* ⟷ *tllist-all2 P Q ys xs*
proof
assume *tllist-all2* (*flip P*) (*flip Q*) *xs ys*
thus *tllist-all2 P Q ys xs*
by(*coinduct rule*: *tllist-all2-coinduct*)(*auto dest*: *tllist-all2-is-TNilD tllist-all2-tfinite2-terminalD tllist-all2-thdD intro*: *tllist-all2-ttlI simp add*: *flip-def*)
next
assume *tllist-all2 P Q ys xs*
thus *tllist-all2* (*flip P*) (*flip Q*) *xs ys*
by(*coinduct rule*: *tllist-all2-coinduct*)(*auto dest*: *tllist-all2-is-TNilD tllist-all2-tfinite2-terminalD tllist-all2-thdD intro*: *tllist-all2-ttlI simp add*: *flip-def*)
qed

1.15.1 Labelled transition systems

type-synonym ('*a*, '*b*) *trsys* = '*a* ⇒ '*b* ⇒ '*a* ⇒ *bool*

locale *trsys* =
fixes *trsys* :: ('*s*, '*tl*) *trsys* (‹-/ −-→/ -› [*50*, *0*, *50*] *60*)
begin

abbreviation *Trsys* :: ('*s*, '*tl list*) *trsys* (‹-/ −-→*/ -› [*50*,*0*,*50*] *60*)
where ⋀*tl*. *s* −*tl*→* *s'* ≡ *rtrancl3p trsys s tl s'*

coinductive *inf-step* :: '*s* ⇒ '*tl llist* ⇒ *bool* (‹- −-→* ∞› [*50*, *0*] *80*)
where *inf-stepI*: ⟦ *trsys a b a'*; *a'* −*bs*→* ∞ ⟧ ⟹ *a* −*LCons b bs*→* ∞

coinductive *inf-step-table* :: '*s* ⇒ ('*s* × '*tl* × '*s*) *llist* ⇒ *bool* (‹- −-→*t ∞› [*50*, *0*] *80*)
where
inf-step-tableI:
⋀*tl*. ⟦ *trsys s tl s'*; *s'* −*stls*→*t* ∞ ⟧
⟹ *s* −*LCons* (*s*, *tl*, *s'*) *stls*→*t* ∞

definition *inf-step2inf-step-table* :: '*s* ⇒ '*tl llist* ⇒ ('*s* × '*tl* × '*s*) *llist*
where
inf-step2inf-step-table s tls =
unfold-llist
(λ(*s*, *tls*). *lnull tls*)
(λ(*s*, *tls*). (*s*, *lhd tls*, *SOME s'*. *trsys s* (*lhd tls*) *s'* ∧ *s'* −*ltl tls*→* ∞))
(λ(*s*, *tls*). (*SOME s'*. *trsys s* (*lhd tls*) *s'* ∧ *s'* −*ltl tls*→* ∞, *ltl tls*))
(*s*, *tls*)

coinductive *Rtrancl3p* :: '*s* ⇒ ('*tl*, '*s*) *tllist* ⇒ *bool*
where

Rtrancl3p-stop: $(\bigwedge tl\ s'.\ \neg\ s\ -tl\rightarrow\ s') \Longrightarrow$ *Rtrancl3p s* (*TNil s*)
| *Rtrancl3p-into-Rtrancl3p*: $\bigwedge tl.\ [\![\ s\ -tl\rightarrow\ s';\ Rtrancl3p\ s'\ tlss\ ]\!] \Longrightarrow Rtrancl3p\ s\ (TCons\ tl\ tlss)$

inductive-simps *Rtrancl3p-simps*:
Rtrancl3p s (*TNil s'*)
Rtrancl3p s (*TCons tl' tlss*)

inductive-cases *Rtrancl3p-cases*:
Rtrancl3p s (*TNil s'*)
Rtrancl3p s (*TCons tl' tlss*)

coinductive *Runs* :: *'s* ⇒ *'tl llist* ⇒ *bool*
where
Stuck: $(\bigwedge tl\ s'.\ \neg\ s\ -tl\rightarrow\ s') \Longrightarrow Runs\ s\ LNil$
| *Step*: $\bigwedge tl.\ [\![\ s\ -tl\rightarrow\ s';\ Runs\ s'\ tls\ ]\!] \Longrightarrow Runs\ s\ (LCons\ tl\ tls)$

coinductive *Runs-table* :: *'s* ⇒ (*'s* × *'tl* × *'s*) *llist* ⇒ *bool*
where
Stuck: $(\bigwedge tl\ s'.\ \neg\ s\ -tl\rightarrow\ s') \Longrightarrow Runs\text{-}table\ s\ LNil$
| *Step*: $\bigwedge tl.\ [\![\ s\ -tl\rightarrow\ s';\ Runs\text{-}table\ s'\ stlss\ ]\!] \Longrightarrow Runs\text{-}table\ s\ (LCons\ (s,\ tl,\ s')\ stlss)$

inductive-simps *Runs-table-simps*:
Runs-table s LNil
Runs-table s (*LCons stls stlss*)

lemma *inf-step-not-finite-llist*:
assumes *r*: $s\ -bs\rightarrow*\ \infty$
shows ¬ *lfinite bs*
proof
assume *lfinite bs* **thus** *False* **using** *r*
by(*induct arbitrary*: *s rule*: *lfinite.induct*)(*auto elim*: *inf-step.cases*)
qed

lemma *inf-step2inf-step-table-LNil* [*simp*]: *inf-step2inf-step-table s LNil* = *LNil*
by(*simp add*: *inf-step2inf-step-table-def*)

lemma *inf-step2inf-step-table-LCons* [*simp*]:
fixes *tl* **shows**
inf-step2inf-step-table s (*LCons tl tls*) =
LCons (*s*, *tl*, *SOME s'*. *trsys s tl s'* ∧ $s'\ -tls\rightarrow*\ \infty$)
(*inf-step2inf-step-table* (*SOME s'*. *trsys s tl s'* ∧ $s'\ -tls\rightarrow*\ \infty$) *tls*)
by(*simp add*: *inf-step2inf-step-table-def*)

lemma *lnull-inf-step2inf-step-table* [*simp*]:
lnull (*inf-step2inf-step-table s tls*) ⟷ *lnull tls*
by(*simp add*: *inf-step2inf-step-table-def*)

lemma *inf-step2inf-step-table-eq-LNil*:
inf-step2inf-step-table s tls = *LNil* ⟷ *tls* = *LNil*
using *lnull-inf-step2inf-step-table* **unfolding** *lnull-def* **.**

lemma *lhd-inf-step2inf-step-table* [*simp*]:
¬ *lnull tls*
⟹ *lhd* (*inf-step2inf-step-table s tls*) =

```
    (s, lhd tls, SOME s'. trsys s (lhd tls) s' ∧ s' −ltl tls→* ∞)
by(simp add: inf-step2inf-step-table-def)

lemma ltl-inf-step2inf-step-table [simp]:
  ltl (inf-step2inf-step-table s tls) =
   inf-step2inf-step-table (SOME s'. trsys s (lhd tls) s' ∧ s' −ltl tls→* ∞) (ltl tls)
by(cases tls) simp-all

lemma lmap-inf-step2inf-step-table: lmap (fst ∘ snd) (inf-step2inf-step-table s tls) = tls
by(coinduction arbitrary: s tls) auto

lemma inf-step-imp-inf-step-table:
  assumes s −tls→* ∞
  shows ∃ stls. s −stls→*t ∞ ∧ tls = lmap (fst ∘ snd) stls
proof −
  from assms have s −inf-step2inf-step-table s tls→*t ∞
  proof(coinduction arbitrary: s tls)
    case (inf-step-table s tls)
    thus ?case
    proof cases
      case (inf-stepI tl s' tls')
      let ?s' = SOME s'. trsys s tl s' ∧ s' −tls'→* ∞
      have trsys s tl ?s' ∧ ?s' −tls'→* ∞ by(rule someI)(blast intro: inf-stepI)
      thus ?thesis using ‹tls = LCons tl tls'› by auto
    qed
  qed
  moreover have tls = lmap (fst ∘ snd) (inf-step2inf-step-table s tls)
    by(simp only: lmap-inf-step2inf-step-table)
  ultimately show ?thesis by blast
qed

lemma inf-step-table-imp-inf-step:
  s−stls→*t ∞ ⟹s −lmap (fst ∘ snd) stls→* ∞
proof(coinduction arbitrary: s stls rule: inf-step.coinduct)
  case (inf-step s tls)
  thus ?case by cases auto
qed

lemma Runs-table-into-Runs:
  Runs-table s stlss ⟹ Runs s (lmap (λ(s, tl, s'). tl) stlss)
proof(coinduction arbitrary: s stlss)
  case (Runs s tls)
  thus ?case by (cases)auto
qed

lemma Runs-into-Runs-table:
  assumes Runs s tls
  obtains stlss
  where tls = lmap (λ(s, tl, s'). tl) stlss
  and Runs-table s stlss
proof −
  define stlss where stlss s tls = unfold-llist
    (λ(s, tls). lnull tls)
    (λ(s, tls). (s, lhd tls, SOME s'. s −lhd tls→ s' ∧ Runs s' (ltl tls)))
```

```
    (λ(s, tls). (SOME s'. s −lhd tls→ s' ∧ Runs s' (ltl tls), ltl tls))
    (s, tls)
    for s tls
  have [simp]:
    ⋀s. stlss s LNil = LNil
    ⋀s tl tls. stlss s (LCons tl tls) = LCons (s, tl, SOME s'. s −tl→ s' ∧ Runs s' tls) (stlss (SOME s'.
s −tl→ s' ∧ Runs s' tls) tls)
    ⋀s tls. lnull (stlss s tls) ⟷ lnull tls
    ⋀s tls. ¬ lnull tls ⟹ lhd (stlss s tls) = (s, lhd tls, SOME s'. s −lhd tls→ s' ∧ Runs s' (ltl tls))
    ⋀s tls. ¬ lnull tls ⟹ ltl (stlss s tls) = stlss (SOME s'. s −lhd tls→ s' ∧ Runs s' (ltl tls)) (ltl tls)
    by(simp-all add: stlss-def)

  from assms have tls = lmap (λ(s, tl, s'). tl) (stlss s tls)
  proof(coinduction arbitrary: s tls)
    case Eq-llist
    thus ?case by cases(auto 4 3 intro: someI2)
  qed
  moreover
  from assms have Runs-table s (stlss s tls)
  proof(coinduction arbitrary: s tls)
    case (Runs-table s stlss')
    thus ?case
    proof(cases)
      case (Step s' tls' tl)
      let ?P = λs'. s −tl→ s' ∧ Runs s' tls'
      from ‹s −tl→ s'› ‹Runs s' tls'› have ?P s' ..
      hence ?P (Eps ?P) by(rule someI)
      with Step have ?Step by auto
      thus ?thesis ..
    qed simp
  qed
  ultimately show ?thesis by(rule that)
qed

lemma Runs-lappendE:
  assumes Runs σ (lappend tls tls')
  and lfinite tls
  obtains σ' where σ −list-of tls→* σ'
  and Runs σ' tls'
proof(atomize-elim)
  from ‹lfinite tls› ‹Runs σ (lappend tls tls')›
  show ∃σ'. σ −list-of tls→* σ' ∧ Runs σ' tls'
  proof(induct arbitrary: σ)
    case lfinite-LNil thus ?case by(auto)
  next
    case (lfinite-LConsI tls tl)
    from ‹Runs σ (lappend (LCons tl tls) tls')›
    show ?case unfolding lappend-code
    proof(cases)
      case (Step σ')
     from ‹Runs σ' (lappend tls tls') ⟹ ∃σ''. σ' −list-of tls→* σ'' ∧ Runs σ'' tls'› ‹Runs σ' (lappend
tls tls')›
      obtain σ'' where σ' −list-of tls→* σ'' Runs σ'' tls' by blast
      from ‹σ −tl→ σ'› ‹σ' −list-of tls→* σ''›
```

```
      have σ −tl # list-of tls→* σ'' by(rule rtrancl3p-step-converse)
      with ‹lfinite tls› have σ −list-of (LCons tl tls)→* σ'' by(simp)
      with ‹Runs σ'' tls'› show ?thesis by blast
    qed
  qed
qed

lemma Trsys-into-Runs:
  assumes s −tls→* s'
  and Runs s' tls'
  shows Runs s (lappend (llist-of tls) tls')
using assms
by(induct rule: rtrancl3p-converse-induct)(auto intro: Runs.Step)

lemma rtrancl3p-into-Rtrancl3p:
  ⟦ rtrancl3p trsys a bs a'; ⋀b a''. ¬ a' −b→ a'' ⟧ ⟹ Rtrancl3p a (tllist-of-llist a' (llist-of bs))
  by(induct rule: rtrancl3p-converse-induct)(auto intro: Rtrancl3p.intros)

lemma Rtrancl3p-into-Runs:
  Rtrancl3p s tlss ⟹ Runs s (llist-of-tllist tlss)
by(coinduction arbitrary: s tlss rule: Runs.coinduct)(auto elim: Rtrancl3p.cases)

lemma Runs-into-Rtrancl3p:
  assumes Runs s tls
  obtains tlss where tls = llist-of-tllist tlss Rtrancl3p s tlss
proof
  let ?Q = λs tls s'. s −lhd tls→ s' ∧ Runs s' (ltl tls)
  define tlss where tlss = corec-tllist
    (λ(s, tls). lnull tls) (λ(s, tls). s)
    (λ(s, tls). lhd tls)
    (λ-. False) undefined (λ(s, tls). (SOME s'. ?Q s tls s', ltl tls))
  have [simp]:
    tlss (s, LNil) = TNil s
    tlss (s, LCons tl tls) = TCons tl (tlss (SOME s'. ?Q s (LCons tl tls) s', tls))
    for s tl tls by(auto simp add: tlss-def intro: tllist.expand)

  show tls = llist-of-tllist (tlss (s, tls)) using assms
    by(coinduction arbitrary: s tls)(erule Runs.cases; fastforce intro: someI2)

  show Rtrancl3p s (tlss (s, tls)) using assms
    by(coinduction arbitrary: s tls)(erule Runs.cases; simp; iprover intro: someI2[where Q=trsys - -]
someI2[where Q=λs'. Runs s' -])
qed

lemma fixes tl
  assumes Rtrancl3p s tlss tfinite tlss
  shows Rtrancl3p-into-Trsys: Trsys s (list-of (llist-of-tllist tlss)) (terminal tlss)
    and terminal-Rtrancl3p-final: ¬ terminal tlss −tl→ s'
using assms(2,1) by(induction arbitrary: s rule: tfinite-induct)(auto simp add: Rtrancl3p-simps intro:
rtrancl3p-step-converse)

end
```

1.15.2 Labelled transition systems with internal actions

locale $\tau trsys = trsys +$
  constrains $trsys :: ('s, 'tl)\ trsys$
  fixes $\tau move :: ('s, 'tl)\ trsys$
begin

inductive *silent-move* $:: 's \Rightarrow 's \Rightarrow bool$ $(\langle\text{-}\ -\tau\rightarrow\ \text{-}\rangle\ [50, 50]\ 60)$
where [*intro*]: $!!tl.\ [\![\ trsys\ s\ tl\ s';\ \tau move\ s\ tl\ s'\ ]\!] \Longrightarrow s\ -\tau\rightarrow\ s'$

declare *silent-move.cases* [*elim*]

lemma *silent-move-iff*: $silent\text{-}move = (\lambda s\ s'.\ (\exists\, tl.\ trsys\ s\ tl\ s' \wedge \tau move\ s\ tl\ s'))$
by(*auto simp add*: *fun-eq-iff*)

abbreviation *silent-moves* $:: 's \Rightarrow 's \Rightarrow bool$ $(\langle\text{-}\ -\tau\rightarrow*\ \text{-}\rangle\ [50, 50]\ 60)$
where $silent\text{-}moves == silent\text{-}move\hat{}**$

abbreviation *silent-movet* $:: 's \Rightarrow 's \Rightarrow bool$ $(\langle\text{-}\ -\tau\rightarrow+\ \text{-}\rangle\ [50, 50]\ 60)$
where $silent\text{-}movet == silent\text{-}move\hat{}++$

coinductive $\tau diverge :: 's \Rightarrow bool$ $(\langle\text{-}\ -\tau\rightarrow\ \infty\rangle\ [50]\ 60)$
where
  $\tau divergeI$: $[\![\ s\ -\tau\rightarrow\ s';\ s'\ -\tau\rightarrow\ \infty\ ]\!] \Longrightarrow s\ -\tau\rightarrow\ \infty$

coinductive $\tau inf\text{-}step :: 's \Rightarrow 'tl\ llist \Rightarrow bool$ $(\langle\text{-}\ -\tau\text{-}\text{-}\rightarrow*\ \infty\rangle\ [50, 0]\ 60)$
where
  $\tau inf\text{-}step\text{-}Cons$: $\bigwedge tl.\ [\![\ s\ -\tau\rightarrow*\ s';\ s'\ -tl\rightarrow\ s'';\ \neg\ \tau move\ s'\ tl\ s'';\ s''\ -\tau-tls\rightarrow*\ \infty\ ]\!] \Longrightarrow s\ -\tau-LCons\ tl\ tls\rightarrow*\ \infty$
| $\tau inf\text{-}step\text{-}Nil$: $s\ -\tau\rightarrow\ \infty \Longrightarrow s\ -\tau-LNil\rightarrow*\ \infty$

coinductive $\tau inf\text{-}step\text{-}table :: 's \Rightarrow ('s \times 's \times 'tl \times 's)\ llist \Rightarrow bool$ $(\langle\text{-}\ -\tau\text{-}\text{-}\rightarrow*t\ \infty\rangle\ [50, 0]\ 80)$
where
  $\tau inf\text{-}step\text{-}table\text{-}Cons$:
  $\bigwedge tl.\ [\![\ s\ -\tau\rightarrow*\ s';\ s'\ -tl\rightarrow\ s'';\ \neg\ \tau move\ s'\ tl\ s'';\ s''\ -\tau-tls\rightarrow*t\ \infty\ ]\!] \Longrightarrow s\ -\tau-LCons\ (s, s', tl, s'')\ tls\rightarrow*t\ \infty$

| $\tau inf\text{-}step\text{-}table\text{-}Nil$:
  $s\ -\tau\rightarrow\ \infty \Longrightarrow s\ -\tau-LNil\rightarrow*t\ \infty$

definition $\tau inf\text{-}step2\tau inf\text{-}step\text{-}table :: 's \Rightarrow 'tl\ llist \Rightarrow ('s \times 's \times 'tl \times 's)\ llist$
where
  $\tau inf\text{-}step2\tau inf\text{-}step\text{-}table\ s\ tls =$
   unfold-llist
     $(\lambda(s, tls).\ lnull\ tls)$
     $(\lambda(s, tls).\ let\ (s', s'') = SOME\ (s', s'').\ s\ -\tau\rightarrow*\ s' \wedge s'\ -lhd\ tls\rightarrow\ s'' \wedge \neg\ \tau move\ s'\ (lhd\ tls)\ s'' \wedge s''\ -\tau-ltl\ tls\rightarrow*\ \infty$
        in $(s, s', lhd\ tls, s''))$
     $(\lambda(s, tls).\ let\ (s', s'') = SOME\ (s', s'').\ s\ -\tau\rightarrow*\ s' \wedge s'\ -lhd\ tls\rightarrow\ s'' \wedge \neg\ \tau move\ s'\ (lhd\ tls)\ s'' \wedge s''\ -\tau-ltl\ tls\rightarrow*\ \infty$
        in $(s'', ltl\ tls))$
     (s, tls)

definition *silent-move-from* $:: 's \Rightarrow 's \Rightarrow 's \Rightarrow bool$

where *silent-move-from s0 s1 s2* $\longleftrightarrow$ *silent-moves s0 s1* $\wedge$ *silent-move s1 s2*

inductive τ*rtrancl3p* :: *'s* $\Rightarrow$ *'tl list* $\Rightarrow$ *'s* $\Rightarrow$ *bool* (‹- $-\tau-\!\!\to*$ -› [*50*, *0*, *50*] *60*)
where
  τ*rtrancl3p-refl*: τ*rtrancl3p s* [] *s*
| τ*rtrancl3p-step*: $\bigwedge$*tl*. ⟦ *s* $-$*tl*$\to$ *s'*; ¬ τ*move s tl s'*; τ*rtrancl3p s' tls s''* ⟧ $\Longrightarrow$ τ*rtrancl3p s* (*tl* # *tls*) *s''*
| τ*rtrancl3p-*τ*step*: $\bigwedge$*tl*. ⟦ *s* $-$*tl*$\to$ *s'*; τ*move s tl s'*; τ*rtrancl3p s' tls s''* ⟧ $\Longrightarrow$ τ*rtrancl3p s tls s''*

coinductive τ*Runs* :: *'s* $\Rightarrow$ (*'tl*, *'s option*) *tllist* $\Rightarrow$ *bool* (‹- $\Downarrow$ -› [*50*, *50*] *51*)
where
  Terminate: ⟦ *s* $-\tau\to*$ *s'*; $\bigwedge$*tl s''*. ¬ *s'* $-$*tl*$\to$ *s''* ⟧ $\Longrightarrow$ *s* $\Downarrow$ *TNil* ⌊*s'*⌋
| *Diverge*: *s* $-\tau\to$ ∞ $\Longrightarrow$ *s* $\Downarrow$ *TNil None*
| *Proceed*: $\bigwedge$*tl*. ⟦ *s* $-\tau\to*$ *s'*; *s'* $-$*tl*$\to$ *s''*; ¬ τ*move s' tl s''*; *s''* $\Downarrow$ *tls* ⟧ $\Longrightarrow$ *s* $\Downarrow$ *TCons tl tls*

inductive-simps τ*Runs-simps*:
  s $\Downarrow$ *TNil* (*Some s'*)
  s $\Downarrow$ *TNil None*
  s $\Downarrow$ *TCons tl' tls*

coinductive τ*Runs-table* :: *'s* $\Rightarrow$ (*'tl* $\times$ *'s*, *'s option*) *tllist* $\Rightarrow$ *bool*
where
  Terminate: ⟦ *s* $-\tau\to*$ *s'*; $\bigwedge$*tl s''*. ¬ *s'* $-$*tl*$\to$ *s''* ⟧ $\Longrightarrow$ τ*Runs-table s* (*TNil* ⌊*s'*⌋)
| *Diverge*: *s* $-\tau\to$ ∞ $\Longrightarrow$ τ*Runs-table s* (*TNil None*)
| *Proceed*:
  $\bigwedge$*tl*. ⟦ *s* $-\tau\to*$ *s'*; *s'* $-$*tl*$\to$ *s''*; ¬ τ*move s' tl s''*; τ*Runs-table s'' tls* ⟧
  $\Longrightarrow$ τ*Runs-table s* (*TCons* (*tl*, *s''*) *tls*)

definition *silent-move2* :: *'s* $\Rightarrow$ *'tl* $\Rightarrow$ *'s* $\Rightarrow$ *bool*
where $\bigwedge$*tl*. *silent-move2 s tl s'* $\longleftrightarrow$ *s* $-$*tl*$\to$ *s'* $\wedge$ τ*move s tl s'*

abbreviation *silent-moves2* :: *'s* $\Rightarrow$ *'tl list* $\Rightarrow$ *'s* $\Rightarrow$ *bool*
where *silent-moves2* $\equiv$ *rtrancl3p silent-move2*

coinductive τ*Runs-table2* :: *'s* $\Rightarrow$ (*'tl list* $\times$ *'s* $\times$ *'tl* $\times$ *'s*, (*'tl list* $\times$ *'s*) + *'tl llist*) *tllist* $\Rightarrow$ *bool*
where
  Terminate: ⟦ *silent-moves2 s tls s'*; $\bigwedge$*tl s''*. ¬ *s'* $-$*tl*$\to$ *s''* ⟧ $\Longrightarrow$ τ*Runs-table2 s* (*TNil* (*Inl* (*tls*, *s'*)))
| *Diverge*: *trsys.inf-step silent-move2 s tls* $\Longrightarrow$ τ*Runs-table2 s* (*TNil* (*Inr tls*))
| *Proceed*:
  $\bigwedge$*tl*. ⟦ *silent-moves2 s tls s'*; *s'* $-$*tl*$\to$ *s''*; ¬ τ*move s' tl s''*; τ*Runs-table2 s'' tlsstlss* ⟧
  $\Longrightarrow$ τ*Runs-table2 s* (*TCons* (*tls*, *s'*, *tl*, *s''*) *tlsstlss*)

inductive-simps τ*Runs-table2-simps*:
  τ*Runs-table2 s* (*TNil tlss*)
  τ*Runs-table2 s* (*TCons tlsstls tlsstlss*)

lemma *inf-step-table-all-*τ*-into-*τ*diverge*:
  ⟦ *s* $-$*stls*$\to*t$ ∞; $\forall$ (*s*, *tl*, *s'*) $\in$ *lset stls*. τ*move s tl s'* ⟧ $\Longrightarrow$ *s* $-\tau\to$ ∞
proof(*coinduction arbitrary*: *s stls*)
  case (τ*diverge s*)
  thus *?case* **by** *cases* (*auto simp add*: *silent-move-iff*, *blast*)
qed

lemma *inf-step-table-lappend-llist-ofD*:
  s $-$*lappend* (*llist-of stls*) (*LCons* (*x*, *tl'*, *x'*) *xs*)$\to*t$ ∞

$\Longrightarrow$ (s $-map$ (fst $\circ$ snd) $stls\rightarrow*$ x) $\wedge$ (x $-LCons$ (x, tl', x') $xs\rightarrow*t$ ∞)
proof(*induct stls arbitrary*: *s*)
case *Nil* **thus** *?case* **by**(*auto elim*: *inf-step-table.cases intro*: *inf-step-table.intros rtrancl3p-refl*)
next
case (*Cons st stls*)
note *IH* = ‹$\bigwedge$*s*. *s* $-lappend$ (*llist-of stls*) (*LCons* (*x*, *tl'*, *x'*) *xs*)$\rightarrow*t$ ∞ $\Longrightarrow$
s $-map$ (*fst* $\circ$ *snd*) *stls*$\rightarrow*$ *x* $\wedge$ *x* $-LCons$ (*x*, *tl'*, *x'*) *xs*$\rightarrow*t$ ∞›
from ‹*s* $-lappend$ (*llist-of* (*st* # *stls*)) (*LCons* (*x*, *tl'*, *x'*) *xs*)$\rightarrow*t$ ∞›
show *?case*
proof *cases*
case (*inf-step-tableI s' stls' tl*)
hence [*simp*]: *st* = (*s*, *tl*, *s'*) *stls'* = *lappend* (*llist-of stls*) (*LCons* (*x*, *tl'*, *x'*) *xs*)
and *s* $-tl\rightarrow$ *s' s'* $-lappend$ (*llist-of stls*) (*LCons* (*x*, *tl'*, *x'*) *xs*)$\rightarrow*t$ ∞ **by** *simp-all*
from *IH*[*OF* ‹*s'* $-lappend$ (*llist-of stls*) (*LCons* (*x*, *tl'*, *x'*) *xs*)$\rightarrow*t$ ∞›]
have *s'* $-map$ (*fst* $\circ$ *snd*) *stls*$\rightarrow*$ *x x* $-LCons$ (*x*, *tl'*, *x'*) *xs*$\rightarrow*t$ ∞ **by** *auto*
with ‹*s* $-tl\rightarrow$ *s'*› **show** *?thesis* **by**(*auto simp add*: *o-def intro*: *rtrancl3p-step-converse*)
qed
qed

lemma *inf-step-table-lappend-llist-of-τ-into-τmoves*:
assumes *lfinite stls*
shows ⟦ *s* $-lappend$ *stls* (*LCons* (*x*, *tl' x'*) *xs*)$\rightarrow*t$ ∞; $\forall$(*s*, *tl*, *s'*)$\in$*lset stls*. τ*move s tl s'* ⟧ $\Longrightarrow$ *s* $-\tau\rightarrow*$ *x*
using *assms*
proof(*induct arbitrary*: *s rule*: *lfinite.induct*)
case *lfinite-LNil* **thus** *?case* **by**(*auto elim*: *inf-step-table.cases*)
next
case (*lfinite-LConsI stls st*)
note *IH* = ‹$\bigwedge$*s*. ⟦*s* $-lappend$ *stls* (*LCons* (*x*, *tl' x'*) *xs*)$\rightarrow*t$ ∞; $\forall$(*s*, *tl*, *s'*)$\in$*lset stls*. τ*move s tl s'* ⟧ $\Longrightarrow$ *s* $-\tau\rightarrow*$ *x*›
obtain *s1 tl1 s1'* **where** [*simp*]: *st* = (*s1*, *tl1*, *s1'*) **by**(*cases st*)
from ‹*s* $-lappend$ (*LCons st stls*) (*LCons* (*x*, *tl' x'*) *xs*)$\rightarrow*t$ ∞›
show *?case*
proof *cases*
case (*inf-step-tableI X' STLS TL*)
hence [*simp*]: *s1* = *s TL* = *tl1 X'* = *s1' STLS* = *lappend stls* (*LCons* (*x*, *tl' x'*) *xs*)
and *s* $-tl1\rightarrow$ *s1'* **and** *s1'* $-lappend$ *stls* (*LCons* (*x*, *tl' x'*) *xs*)$\rightarrow*t$ ∞ **by** *simp-all*
from ‹$\forall$(*s*, *tl*, *s'*)$\in$*lset* (*LCons st stls*). τ*move s tl s'*› **have** τ*move s tl1 s1'* **by** *simp*
moreover
from *IH*[*OF* ‹*s1'* $-lappend$ *stls* (*LCons* (*x*, *tl' x'*) *xs*)$\rightarrow*t$ ∞›] ‹$\forall$(*s*, *tl*, *s'*)$\in$*lset* (*LCons st stls*). τ*move s tl s'*›
have *s1'* $-\tau\rightarrow*$ *x* **by** *simp*
ultimately show *?thesis* **using** ‹*s* $-tl1\rightarrow$ *s1'*› **by**(*auto intro*: *converse-rtranclp-into-rtranclp*)
qed
qed

lemma *inf-step-table-into-τinf-step*:
s $-stls\rightarrow*t$ ∞ $\Longrightarrow$ *s* $-\tau-lmap$ (*fst* $\circ$ *snd*) (*lfilter* (λ(*s*, *tl*, *s'*). $\neg$ τ*move s tl s'*) *stls*)$\rightarrow*$ ∞
proof(*coinduction arbitrary*: *s stls*)
case (*τinf-step s stls*)
let *?P* = λ(*s*, *tl*, *s'*). $\neg$ τ*move s tl s'*
show *?case*
proof(*cases lfilter ?P stls*)

```
    case LNil
    with τinf-step have ?τinf-step-Nil
      by(auto intro: inf-step-table-all-τ-into-τdiverge simp add: lfilter-eq-LNil)
    thus ?thesis ..
  next
    case (LCons stls' xs)
    obtain x tl x' where stls' = (x, tl, x') by(cases stls')
    with LCons have stls: lfilter ?P stls = LCons (x, tl, x') xs by simp
    from lfilter-eq-LConsD[OF this] obtain stls1 stls2
      where stls1: stls = lappend stls1 (LCons (x, tl, x') stls2)
      and lfinite stls1
      and τs: ∀(s, tl, s')∈lset stls1. τmove s tl s'
      and nτ: ¬ τmove x tl x' and xs: xs = lfilter ?P stls2 by blast
    from ‹lfinite stls1› τinf-step τs have s −τ→* x unfolding stls1
      by(rule inf-step-table-lappend-llist-of-τ-into-τmoves)
    moreover from ‹lfinite stls1› have llist-of (list-of stls1) = stls1 by(simp add: llist-of-list-of)
    with τinf-step stls1 have s −lappend (llist-of (list-of stls1)) (LCons (x, tl, x') stls2)→*t ∞ by
simp
    from inf-step-table-lappend-llist-ofD[OF this]
    have x −LCons (x, tl, x') stls2→*t ∞ ..
    hence x −tl→ x' x' −stls2→*t ∞ by(auto elim: inf-step-table.cases)
    ultimately have ?τinf-step-Cons using xs nτ by(auto simp add: stls o-def)
    thus ?thesis ..
  qed
qed

lemma inf-step-into-τinf-step:
  assumes s −tls→* ∞
  shows ∃A. s −τ−lnths tls A→* ∞
proof −
  from inf-step-imp-inf-step-table[OF assms]
  obtain stls where s −stls→*t ∞ and tls: tls = lmap (fst ∘ snd) stls by blast
  from ‹s −stls→*t ∞› have s −τ−lmap (fst ∘ snd) (lfilter (λ(s, tl, s'). ¬ τmove s tl s') stls)→* ∞
    by(rule inf-step-table-into-τinf-step)
  hence s −τ−lnths tls {n. enat n < llength stls ∧ (λ(s, tl, s'). ¬ τmove s tl s') (lnth stls n)}→* ∞
    unfolding lfilter-conv-lnths tls by simp
  thus ?thesis by blast
qed

lemma silent-moves-into-τrtrancl3p:
  s −τ→* s' ⟹ s −τ−[]→* s'
by(induct rule: converse-rtranclp-induct)(blast intro: τrtrancl3p.intros)+

lemma τrtrancl3p-into-silent-moves:
  s −τ−[]→* s' ⟹ s −τ→* s'
apply(induct s tls≡[] :: 'tl list s' rule: τrtrancl3p.induct)
apply(auto intro: converse-rtranclp-into-rtranclp)
done

lemma τrtrancl3p-Nil-eq-τmoves:
  s −τ−[]→* s' ⟷ s −τ→* s'
by(blast intro: silent-moves-into-τrtrancl3p τrtrancl3p-into-silent-moves)

lemma τrtrancl3p-trans [trans]:
```

```
  ⟦ s −τ−tls→* s'; s' −τ−tls'→* s'' ⟧ ⟹ s −τ−tls @ tls'→* s''
apply(induct rule: τrtrancl3p.induct)
apply(auto intro: τrtrancl3p.intros)
done

lemma τrtrancl3p-SingletonE:
  fixes tl
  assumes red: s −τ−[tl]→* s'''
  obtains s' s'' where s −τ→* s' s' −tl→ s'' ¬ τmove s' tl s'' s'' −τ→* s'''
proof(atomize-elim)
  from red show ∃ s' s''. s −τ→* s' ∧ s' −tl→ s'' ∧ ¬ τmove s' tl s'' ∧ s'' −τ→* s'''
  proof(induct s tls≡[tl] s''')
    case (τrtrancl3p-step s s' s'')
    from ‹s −tl→ s'› ‹¬ τmove s tl s'› ‹s' −τ−[]→* s''› show ?case
      by(auto simp add: τrtrancl3p-Nil-eq-τmoves)
   next
    case (τrtrancl3p-τstep s s' s'' tl')
    then obtain t' t'' where s' −τ→* t' t' −tl→ t'' ¬ τmove t' tl t'' t'' −τ→* s'' by auto
    moreover
    from ‹s −tl'→ s'› ‹τmove s tl' s'› have s −τ→* s' by blast
    ultimately show ?case by(auto intro: rtranclp-trans)
  qed
qed

lemma τrtrancl3p-snocI:
  ⋀tl. ⟦ τrtrancl3p s tls s''; s'' −τ→* s'''; s''' −tl→ s'; ¬ τmove s''' tl s' ⟧
  ⟹ τrtrancl3p s (tls @ [tl]) s'
apply(erule τrtrancl3p-trans)
apply(fold τrtrancl3p-Nil-eq-τmoves)
apply(drule τrtrancl3p-trans)
 apply(erule (1) τrtrancl3p-step)
 apply(rule τrtrancl3p-refl)
apply simp
done

lemma τdiverge-rtranclp-silent-move:
  ⟦ silent-move^** s s'; s' −τ→ ∞ ⟧ ⟹ s −τ→ ∞
by(induct rule: converse-rtranclp-induct)(auto intro: τdivergeI)

lemma τdiverge-trancl-coinduct [consumes 1, case-names τdiverge]:
  assumes X: X s
  and step: ⋀s. X s ⟹ ∃ s'. silent-move^++ s s' ∧ (X s' ∨ s' −τ→ ∞)
  shows s −τ→ ∞
proof −
  from X have ∃ s'. silent-move^** s s' ∧ X s' by blast
  thus ?thesis
  proof(coinduct)
    case (τdiverge s)
    then obtain s' where silent-move** s s' X s' by blast
    from step[OF ‹X s'›] obtain s'''
      where silent-move^++ s' s''' X s''' ∨ s''' −τ→ ∞ by blast
    from ‹silent-move** s s'› show ?case
    proof(cases rule: converse-rtranclpE[consumes 1, case-names refl step])
      case refl
```

```
    moreover from tranclpD[OF ‹silent-move^++ s' s'''›] obtain s''
      where silent-move s' s'' silent-move^** s'' s''' by blast
    ultimately show ?thesis using ‹silent-move^** s'' s'''› ‹X s''' ∨ s''' −τ→ ∞›
      by(auto intro: τdiverge-rtranclp-silent-move)
   next
    case (step S)
    moreover from ‹silent-move** S s'› ‹silent-move^++ s' s'''›
    have silent-move^** S s''' by(rule rtranclp-trans[OF - tranclp-into-rtranclp])
   ultimately show ?thesis using ‹X s''' ∨ s''' −τ→ ∞› by(auto intro: τdiverge-rtranclp-silent-move)
   qed
  qed
qed

lemma τdiverge-trancl-measure-coinduct [consumes 2, case-names τdiverge]:
  assumes major: X s t wfP μ
  and step: ⋀s t. X s t ⟹ ∃s' t'. (μ t' t ∧ s' = s ∨ silent-move^++ s s') ∧ (X s' t' ∨ s' −τ→ ∞)
  shows s −τ→ ∞
proof −
  { fix s t
    assume X s t
    with ‹wfP μ› have ∃s' t'. silent-move^++ s s' ∧ (X s' t' ∨ s' −τ→ ∞)
    proof(induct arbitrary: s rule: wfp-induct[consumes 1])
      case (1 t)
      hence IH: ⋀s' t'. ⟦ μ t' t; X s' t' ⟧ ⟹
                ∃s'' t''. silent-move^++ s' s'' ∧ (X s'' t'' ∨ s'' −τ→ ∞) by blast
      from step[OF ‹X s t›] obtain s' t'
        where μ t' t ∧ s' = s ∨ silent-move^++ s s' X s' t' ∨ s' −τ→ ∞ by blast
      from ‹μ t' t ∧ s' = s ∨ silent-move^++ s s'› show ?case
      proof
        assume μ t' t ∧ s' = s
        hence  μ t' t and [simp]: s' = s by simp-all
        from ‹X s' t' ∨ s' −τ→ ∞› show ?thesis
        proof
          assume X s' t'
          from IH[OF ‹μ t' t› this] show ?thesis by simp
        next
          assume s' −τ→ ∞ thus ?thesis
            by cases(auto simp add: silent-move-iff)
        qed
      next
        assume silent-move^++ s s'
        thus ?thesis using ‹X s' t' ∨ s' −τ→ ∞› by blast
      qed
    qed }
  note X = this
  from ‹X s t› have ∃t. X s t ..
  thus ?thesis
  proof(coinduct rule: τdiverge-trancl-coinduct)
    case (τdiverge s)
    then obtain t where X s t ..
    from X[OF this] show ?case by blast
  qed
qed
```

lemma *τinf-step2τinf-step-table-LNil* [*simp*]: *τinf-step2τinf-step-table s LNil = LNil*
by(*simp add*: *τinf-step2τinf-step-table-def*)

lemma *τinf-step2τinf-step-table-LCons* [*simp*]:
fixes *s tl ss tls*
defines *ss* ≡ *SOME* (*s'*, *s''*). *s* −*τ*→∗ *s'* ∧ *s'* −*tl*→ *s''* ∧ ¬ *τmove s' tl s''* ∧ *s''* −*τ*−*tls*→∗ ∞
shows
τinf-step2τinf-step-table s (*LCons tl tls*) =
LCons (*s*, *fst ss*, *tl*, *snd ss*) (*τinf-step2τinf-step-table* (*snd ss*) *tls*)
by(*simp add*: *ss-def τinf-step2τinf-step-table-def split-beta*)

lemma *lnull-τinf-step2τinf-step-table* [*simp*]:
lnull (*τinf-step2τinf-step-table s tls*) ⟷ *lnull tls*
by(*simp add*: *τinf-step2τinf-step-table-def*)

lemma *lhd-τinf-step2τinf-step-table* [*simp*]:
¬ *lnull tls* ⟹ *lhd* (*τinf-step2τinf-step-table s tls*) =
(*let* (*s'*, *s''*) = *SOME* (*s'*, *s''*). *s* −*τ*→∗ *s'* ∧ *s'* −*lhd tls*→ *s''* ∧ ¬ *τmove s'* (*lhd tls*) *s''* ∧ *s''* −*τ*−*ltl tls*→∗ ∞
in (*s*, *s'*, *lhd tls*, *s''*))
unfolding *τinf-step2τinf-step-table-def Let-def* **by** *simp*

lemma *ltl-τinf-step2τinf-step-table* [*simp*]:
¬ *lnull tls* ⟹ *ltl* (*τinf-step2τinf-step-table s tls*) =
(*let* (*s'*, *s''*) = *SOME* (*s'*, *s''*). *s* −*τ*→∗ *s'* ∧ *s'* −*lhd tls*→ *s''* ∧ ¬ *τmove s'* (*lhd tls*) *s''* ∧ *s''* −*τ*−*ltl tls*→∗ ∞
in τinf-step2τinf-step-table s'' (*ltl tls*))
unfolding *τinf-step2τinf-step-table-def Let-def*
by(*simp add*: *split-beta*)

lemma *lmap-τinf-step2τinf-step-table*: *lmap* (*fst* ∘ *snd* ∘ *snd*) (*τinf-step2τinf-step-table s tls*) = *tls*
by(*coinduction arbitrary*: *s tls*)(*auto simp add*: *split-beta*)

lemma *τinf-step-into-τinf-step-table*:
s −*τ*−*tls*→∗ ∞ ⟹ *s* −*τ*−*τinf-step2τinf-step-table s tls*→∗*t* ∞
proof(*coinduction arbitrary*: *s tls*)
case (*τinf-step-table s tls*)
thus *?case*
proof(*cases*)
case (*τinf-step-Cons s' s'' tls' tl*)
let *?ss* = *SOME* (*s'*, *s''*). *s* −*τ*→∗ *s'* ∧ *s'* −*tl*→ *s''* ∧ ¬ *τmove s' tl s''* ∧ *s''* −*τ*−*tls'*→∗ ∞
from *τinf-step-Cons* **have** *tls*: *tls* = *LCons tl tls'* **and** *s* −*τ*→∗ *s' s'* −*tl*→ *s''*
¬ *τmove s' tl s'' s''* −*τ*−*tls'*→∗ ∞ **by** *simp-all*
hence (λ(*s'*, *s''*). *s* −*τ*→∗ *s'* ∧ *s'* −*tl*→ *s''* ∧ ¬ *τmove s' tl s''* ∧ *s''* −*τ*−*tls'*→∗ ∞) (*s'*, *s''*) **by** *simp*
hence (λ(*s'*, *s''*). *s* −*τ*→∗ *s'* ∧ *s'* −*tl*→ *s''* ∧ ¬ *τmove s' tl s''* ∧ *s''* −*τ*−*tls'*→∗ ∞) *?ss* **by**(*rule someI*)
with *tls* **have** *?τinf-step-table-Cons* **by** *auto*
thus *?thesis* **..**
next
case *τinf-step-Nil*
then have *?τinf-step-table-Nil* **by** *simp*
thus *?thesis* **..**
qed

qed

lemma *τinf-step-imp-τinf-step-table*:
  assumes $s\ -\tau-tls\to*\ \infty$
  shows $\exists\, sstls.\ s\ -\tau-sstls\to*t\ \infty \wedge tls = lmap\ (fst \circ snd \circ snd)\ sstls$
using *τinf-step-into-τinf-step-table*[*OF assms*]
by(*auto simp only*: *lmap-τinf-step2τinf-step-table*)

lemma *τinf-step-table-into-τinf-step*:
  $s\ -\tau-sstls\to*t\ \infty \Longrightarrow s\ -\tau-lmap\ (fst \circ snd \circ snd)\ sstls\to*\ \infty$
proof(*coinduction arbitrary*: *s sstls*)
  case (*τinf-step s tls*)
  thus *?case* **by** *cases*(*auto simp add*: *o-def*)
qed

lemma *silent-move-fromI* [*intro*]:
  ⟦ *silent-moves s0 s1*; *silent-move s1 s2* ⟧ ⟹ *silent-move-from s0 s1 s2*
by(*simp add*: *silent-move-from-def*)

lemma *silent-move-fromE* [*elim*]:
  assumes *silent-move-from s0 s1 s2*
  obtains *silent-moves s0 s1 silent-move s1 s2*
using *assms* **by**(*auto simp add*: *silent-move-from-def*)

lemma *rtranclp-silent-move-from-imp-silent-moves*:
  assumes *s′x*: *silent-move*** *s′ x*
  shows (*silent-move-from s′*)^** *x z* ⟹ *silent-moves s′ z*
by(*induct rule*: *rtranclp-induct*)(*auto intro*: *s′x*)

lemma *τdiverge-not-wfP-silent-move-from*:
  assumes $s\ -\tau\to\ \infty$
  shows ¬ *wfP* (*flip* (*silent-move-from s*))
proof
  assume *wfP* (*flip* (*silent-move-from s*))
  moreover define *Q* **where** $Q = \{s'.\ silent\text{-}moves\ s\ s' \wedge s'\ -\tau\to\ \infty\}$
  hence $s \in Q$ **using** ‹$s\ -\tau\to\ \infty$› **by**(*auto*)
  ultimately have $\exists\, z{\in}Q.\ \forall\, y.\ silent\text{-}move\text{-}from\ s\ z\ y \longrightarrow y \notin Q$
    unfolding *wfp-eq-minimal flip-simps* **by** *blast*
  then obtain *z* **where** $z \in Q$
    and *min*: $\bigwedge y.\ silent\text{-}move\text{-}from\ s\ z\ y \Longrightarrow y \notin Q$ **by** *blast*
  from ‹$z \in Q$› **have** *silent-moves s z z* $-\tau\to\ \infty$ **unfolding** *Q-def* **by** *auto*
  from ‹$z\ -\tau\to\ \infty$› **obtain** *y* **where** *silent-move z y y* $-\tau\to\ \infty$ **by** *cases auto*
  from ‹*silent-moves s z*› ‹*silent-move z y*› **have** *silent-move-from s z y* **..**
  hence $y \notin Q$ **by**(*rule min*)
  moreover from ‹*silent-moves s z*› ‹*silent-move z y*› ‹$y\ -\tau\to\ \infty$›
  have $y \in Q$ **unfolding** *Q-def* **by** *auto*
  ultimately show *False* **by** *contradiction*
qed

lemma *wfP-silent-move-from-unroll*:
  assumes *wfPs′*: $\bigwedge s'.\ s\ -\tau\to\ s' \Longrightarrow wfP\ (flip\ (silent\text{-}move\text{-}from\ s'))$
  shows *wfP* (*flip* (*silent-move-from s*))
  unfolding *wfp-eq-minimal flip-conv*
proof(*intro allI impI*)

```
  fix Q and x :: 's
  assume x ∈ Q
  show ∃ z∈Q. ∀ y. silent-move-from s z y ⟶ y ∉ Q
  proof(cases ∃ s'. s −τ→ s' ∧ (∃ x'. silent-moves s' x' ∧ x' ∈ Q))
    case False
    hence ∀ y. silent-move-from s x y ⟶ ¬ y ∈ Q
      by(cases x=s)(auto, blast elim: converse-rtranclpE intro: rtranclp.rtrancl-into-rtrancl)
    with ‹x ∈ Q› show ?thesis by blast
  next
    case True
    then obtain s' x' where s −τ→ s' and silent-moves s' x' and x' ∈ Q
      by auto
    from ‹s −τ→ s'› have wfP (flip (silent-move-from s')) by(rule wfPs')
    from this ‹x' ∈ Q› obtain z where z ∈ Q and min: ⋀y. silent-move-from s' z y ⟹ ¬ y ∈ Q
      and (silent-move-from s')^** x' z
      by (rule wfP-minimalE) (unfold flip-simps, blast)
    { fix y
      assume silent-move-from s z y
      with ‹(silent-move-from s')^** x' z› ‹silent-move^** s' x'›
      have silent-move-from s' z y
        by(blast intro: rtranclp-silent-move-from-imp-silent-moves)
      hence ¬ y ∈ Q by(rule min) }
    with ‹z ∈ Q› show ?thesis by(auto simp add: intro!: bexI)
  qed
qed

lemma not-wfP-silent-move-from-τdiverge:
  assumes ¬ wfP (flip (silent-move-from s))
  shows s −τ→ ∞
using assms
proof(coinduct)
  case (τdiverge s)
  { assume wfPs': ⋀s'. s −τ→ s' ⟹ wfP (flip (silent-move-from s'))
    hence wfP (flip (silent-move-from s)) by(rule wfP-silent-move-from-unroll) }
  with τdiverge have ∃ s'. s −τ→ s' ∧ ¬ wfP (flip (silent-move-from s')) by auto
  thus ?case by blast
qed

lemma τdiverge-neq-wfP-silent-move-from:
  s −τ→ ∞ ≠ wfP (flip (silent-move-from s))
by(auto intro: not-wfP-silent-move-from-τdiverge dest: τdiverge-not-wfP-silent-move-from)

lemma not-τdiverge-to-no-τmove:
  assumes ¬ s −τ→ ∞
  shows ∃ s'. s −τ→* s' ∧ (∀ s''. ¬ s' −τ→ s'')
proof −
  define S where S = s
  from ‹¬ τdiverge s› have wfP (flip (silent-move-from S)) unfolding S-def
    using τdiverge-neq-wfP-silent-move-from[of s] by simp
  moreover have silent-moves S s unfolding S-def ..
  ultimately show ?thesis
  proof(induct rule: wfp-induct')
    case (wfP s)
    note IH = ‹⋀y. ⟦flip (silent-move-from S) y s; S −τ→* y ⟧
```

```
        ⟹ ∃ s'. y −τ→* s' ∧ (∀ s''. ¬ s' −τ→ s'')›
    show ?case
    proof(cases ∃ s'. silent-move s s')
      case False thus ?thesis by auto
    next
      case True
      then obtain s' where s −τ→ s' ..
      with ‹S −τ→* s› have flip (silent-move-from S) s' s
        unfolding flip-conv by(rule silent-move-fromI)
      moreover from ‹S −τ→* s› ‹s −τ→ s'› have S −τ→* s' ..
      ultimately have ∃ s''. s' −τ→* s'' ∧ (∀ s'''. ¬ s'' −τ→ s''') by(rule IH)
      then obtain s'' where s' −τ→* s'' ∀ s'''. ¬ s'' −τ→ s''' by blast
      from ‹s −τ→ s'› ‹s' −τ→* s''› have s −τ→* s'' by(rule converse-rtranclp-into-rtranclp)
      with ‹∀ s'''. ¬ s'' −τ→ s'''› show ?thesis by blast
    qed
  qed
qed

lemma τdiverge-conv-τRuns:
  s −τ→ ∞ ⟷ s ⇓ TNil None
by(auto intro: τRuns.Diverge elim: τRuns.cases)

lemma τinf-step-into-τRuns:
  s −τ−tls→* ∞ ⟹ s ⇓ tllist-of-llist None tls
proof(coinduction arbitrary: s tls)
  case (τRuns s tls')
  thus ?case by cases(auto simp add: τdiverge-conv-τRuns)
qed

lemma τ-into-τRuns:
  ⟦ s −τ→ s'; s' ⇓ tls ⟧ ⟹ s ⇓ tls
by(blast elim: τRuns.cases intro: τRuns.intros τdiverge.intros converse-rtranclp-into-rtranclp)

lemma τrtrancl3p-into-τRuns:
  assumes s −τ−tls→* s'
  and s' ⇓ tls'
  shows s ⇓ lappendt (llist-of tls) tls'
using assms
by induct(auto intro: τRuns.Proceed τ-into-τRuns)

lemma τRuns-table-into-τRuns:
  τRuns-table s stlsss ⟹ s ⇓ tmap fst id stlsss
proof(coinduction arbitrary: s stlsss)
  case (τRuns s tls)
  thus ?case by cases(auto simp add: o-def id-def)
qed

definition τRuns2τRuns-table :: 's ⇒ ('tl, 's option) tllist ⇒ ('tl × 's, 's option) tllist
where
  τRuns2τRuns-table s tls = unfold-tllist
     (λ(s, tls). is-TNil tls)
     (λ(s, tls). terminal tls)
     (λ(s, tls). (thd tls, SOME s''. ∃ s'. s −τ→* s' ∧ s' −thd tls→ s'' ∧ ¬ τmove s' (thd tls) s'' ∧ s'' ⇓
ttl tls))
```

(λ(*s*, *tls*). (*SOME s''*. ∃ *s'*. *s* −τ→∗ *s'* ∧ *s'* −*thd tls*→ *s''* ∧ ¬ τ*move s'* (*thd tls*) *s''* ∧ *s''* ⇓ *ttl tls*, *ttl tls*))
(*s*, *tls*)

lemma *is-TNil-τRuns2τRuns-table* [*simp*]:
is-TNil (τ*Runs2τRuns-table s tls*) ⟷ *is-TNil tls*
thm *unfold-tllist.disc*
by(*simp add*: τ*Runs2τRuns-table-def*)

lemma *thd-τRuns2τRuns-table* [*simp*]:
¬ *is-TNil tls* ⟹
thd (τ*Runs2τRuns-table s tls*) =
(*thd tls*, *SOME s''*. ∃ *s'*. *s* −τ→∗ *s'* ∧ *s'* −*thd tls*→ *s''* ∧ ¬ τ*move s'* (*thd tls*) *s''* ∧ *s''* ⇓ *ttl tls*)
by(*simp add*: τ*Runs2τRuns-table-def*)

lemma *ttl-τRuns2τRuns-table* [*simp*]:
¬ *is-TNil tls* ⟹
ttl (τ*Runs2τRuns-table s tls*) =
τ*Runs2τRuns-table* (*SOME s''*. ∃ *s'*. *s* −τ→∗ *s'* ∧ *s'* −*thd tls*→ *s''* ∧ ¬ τ*move s'* (*thd tls*) *s''* ∧ *s''* ⇓ *ttl tls*) (*ttl tls*)
by(*simp add*: τ*Runs2τRuns-table-def*)

lemma *terminal-τRuns2τRuns-table* [*simp*]:
is-TNil tls ⟹ *terminal* (τ*Runs2τRuns-table s tls*) = *terminal tls*
by(*simp add*: τ*Runs2τRuns-table-def*)

lemma τ*Runs2τRuns-table-simps* [*simp*, *nitpick-simp*]:
τ*Runs2τRuns-table s* (*TNil so*) = *TNil so*
⋀*tl*.
τ*Runs2τRuns-table s* (*TCons tl tls*) =
(*let s''* = *SOME s''*. ∃ *s'*. *s* −τ→∗ *s'* ∧ *s'* −*tl*→ *s''* ∧ ¬ τ*move s' tl s''* ∧ *s''* ⇓ *tls*
in TCons (*tl*, *s''*) (τ*Runs2τRuns-table s''* *tls*))
apply(*simp add*: τ*Runs2τRuns-table-def*)
apply(*rule tllist.expand*)
apply(*simp-all*)
done

lemma τ*Runs2τRuns-table-inverse*:
tmap fst id (τ*Runs2τRuns-table s tls*) = *tls*
by(*coinduction arbitrary*: *s tls*) *auto*

lemma τ*Runs-into-τRuns-table*:
assumes *s* ⇓ *tls*
shows ∃ *stlsss*. *tls* = *tmap fst id stlsss* ∧ τ*Runs-table s stlsss*
proof(*intro exI conjI*)
from *assms* **show** τ*Runs-table s* (τ*Runs2τRuns-table s tls*)
proof(*coinduction arbitrary*: *s tls*)
case (τ*Runs-table s tls*)
thus *?case*
proof *cases*
case (*Terminate s'*)
hence *?Terminate* **by** *simp*
thus *?thesis* **..**
next

```
      case Diverge
      hence ?Diverge by simp
      thus ?thesis by simp
    next
      case (Proceed s' s'' tls' tl)
      let ?P = λs''. ∃ s'. s −τ→* s' ∧ s' −tl→ s'' ∧ ¬ τmove s' tl s'' ∧ s'' ⇓ tls'
      from Proceed have ?P s'' by auto
      hence ?P (Eps ?P) by(rule someI)
      hence ?Proceed using ‹tls = TCons tl tls'›
        by(auto simp add: split-beta)
      thus ?thesis by simp
    qed
  qed
qed(simp add: τRuns2τRuns-table-inverse)

lemma τRuns-lappendtE:
  assumes σ ⇓ lappendt tls tls'
  and lfinite tls
  obtains σ' where σ −τ−list-of tls→* σ'
  and σ' ⇓ tls'
proof(atomize-elim)
  from ‹lfinite tls› ‹σ ⇓ lappendt tls tls'›
  show ∃σ'. σ −τ−list-of tls→* σ' ∧ σ' ⇓ tls'
  proof(induct arbitrary: σ)
    case lfinite-LNil thus ?case by(auto intro: τrtrancl3p-refl)
  next
    case (lfinite-LConsI tls tl)
    from ‹σ ⇓ lappendt (LCons tl tls) tls'›
    show ?case unfolding lappendt-LCons
    proof(cases)
      case (Proceed σ' σ'')
      from ‹σ'' ⇓ lappendt tls tls' ⟹ ∃σ'''. σ'' −τ−list-of tls→* σ''' ∧ σ''' ⇓ tls'› ‹σ'' ⇓ lappendt tls
tls'›
      obtain σ''' where σ'' −τ−list-of tls→* σ''' σ''' ⇓ tls' by blast
      from ‹σ' −tl→ σ''› ‹¬ τmove σ' tl σ''› ‹σ'' −τ−list-of tls→* σ'''›
      have σ' −τ−tl # list-of tls→* σ''' by(rule τrtrancl3p-step)
      with ‹σ −τ→* σ'› have σ −τ−[] @ (tl # list-of tls)→* σ'''
        unfolding τrtrancl3p-Nil-eq-τmoves[symmetric] by(rule τrtrancl3p-trans)
      with ‹lfinite tls› have σ −τ−list-of (LCons tl tls)→* σ''' by(simp add: list-of-LCons)
      with ‹σ''' ⇓ tls'› show ?thesis by blast
    qed
  qed
qed

lemma τRuns-total:
  ∃ tls. σ ⇓ tls
proof
  let ?τhalt = λσ σ'. σ −τ→* σ' ∧ (∀ tl σ''. ¬ σ' −tl→ σ'')
  let ?τdiverge = λσ. σ −τ→ ∞
  let ?proceed = λσ (tl, σ''). ∃σ'. σ −τ→* σ' ∧ σ' −tl→ σ'' ∧ ¬ τmove σ' tl σ''

  define tls where tls = unfold-tllist
    (λσ. (∃σ'. ?τhalt σ σ') ∨ ?τdiverge σ)
    (λσ. if ∃σ'. ?τhalt σ σ' then Some (SOME σ'. ?τhalt σ σ') else None)
```

```
    (λσ. fst (SOME tlσ'. ?proceed σ tlσ'))
    (λσ. snd (SOME tlσ'. ?proceed σ tlσ')) σ
  then show σ ⇓ tls
  proof(coinduct σ tls rule: τRuns.coinduct)
    case (τRuns σ tls)
    show ?case
    proof(cases ∃σ'. ?τhalt σ σ')
      case True
      hence ?τhalt σ (SOME σ'. ?τhalt σ σ') by(rule someI-ex)
      hence ?Terminate using True unfolding τRuns by simp
      thus ?thesis ..
    next
      case False
      note τhalt = this
      show ?thesis
      proof(cases ?τdiverge σ)
        case True
        hence ?Diverge using False unfolding τRuns by simp
        thus ?thesis by simp
      next
        case False
        from not-τdiverge-to-no-τmove[OF this]
        obtain σ' where σ-σ': σ −τ→* σ'
          and no-τ: ⋀σ''. ¬ σ' −τ→ σ'' by blast
        from σ-σ' τhalt obtain tl σ'' where σ' −tl→ σ'' by auto
        moreover with no-τ[of σ''] have ¬ τmove σ' tl σ'' by auto
        ultimately have ?proceed σ (tl, σ'') using σ-σ' by auto
        hence ?proceed σ (SOME tlσ. ?proceed σ tlσ) by(rule someI)
        hence ?Proceed using False τhalt unfolding τRuns
          by(subst unfold-tllist.code) fastforce
        thus ?thesis by simp
      qed
    qed
  qed
qed

lemma silent-move2-into-silent-move:
  fixes tl
  assumes silent-move2 s tl s'
  shows s −τ→ s'
using assms by(auto simp add: silent-move2-def)

lemma silent-move-into-silent-move2:
  assumes s −τ→ s'
  shows ∃ tl. silent-move2 s tl s'
using assms by(auto simp add: silent-move2-def)

lemma silent-moves2-into-silent-moves:
  assumes silent-moves2 s tls s'
  shows s −τ→* s'
using assms
by(induct)(blast intro: silent-move2-into-silent-move rtranclp.rtrancl-into-rtrancl)+

lemma silent-moves-into-silent-moves2:
```

assumes $s\ -\tau\rightarrow*\ s'$
shows $\exists\, tls.\ silent\text{-}moves2\ s\ tls\ s'$
using *assms*
by(*induct*)(*blast dest*: *silent-move-into-silent-move2 intro*: *rtrancl3p-step*)+

lemma *inf-step-silent-move2-into-*τ*diverge*:
trsys.inf-step silent-move2 s tls $\Longrightarrow$ $s\ -\tau\rightarrow\ \infty$
proof(*coinduction arbitrary*: *s tls*)
case (τ*diverge s*)
thus *?case*
by(*cases rule*: *trsys.inf-step.cases*[*consumes 1*])(*auto intro*: *silent-move2-into-silent-move*)
qed

lemma τ*diverge-into-inf-step-silent-move2*:
assumes $s\ -\tau\rightarrow\ \infty$
obtains *tls* **where** *trsys.inf-step silent-move2 s tls*
proof −
define *tls* **where** *tls* = *unfold-llist*
(λ-. *False*)
(λ*s*. *fst* (*SOME* (*tl*, *s'*). *silent-move2 s tl s'* $\wedge$ $s'\ -\tau\rightarrow\ \infty$))
(λ*s*. *snd* (*SOME* (*tl*, *s'*). *silent-move2 s tl s'* $\wedge$ $s'\ -\tau\rightarrow\ \infty$))
s (**is** - = *?tls s*)

with *assms* **have** $s\ -\tau\rightarrow\ \infty\ \wedge$ *tls* = *?tls s* **by** *simp*
hence *trsys.inf-step silent-move2 s tls*
proof(*coinduct rule*: *trsys.inf-step.coinduct*[*consumes 1*, *case-names inf-step*, *case-conclusion inf-step step*])
case (*inf-step s tls*)
let *?P* = λ(*tl*, *s'*). *silent-move2 s tl s'* $\wedge$ $s'\ -\tau\rightarrow\ \infty$
from *inf-step* **obtain** $s\ -\tau\rightarrow\ \infty$ **and** *tls*: *tls* = *?tls s* **..**
from ‹$s\ -\tau\rightarrow\ \infty$› **obtain** s' **where** $s\ -\tau\rightarrow\ s'$ $s'\ -\tau\rightarrow\ \infty$ **by** *cases*
from ‹$s\ -\tau\rightarrow\ s'$› **obtain** *tl* **where** *silent-move2 s tl s'*
by(*blast dest*: *silent-move-into-silent-move2*)
with ‹$s'\ -\tau\rightarrow\ \infty$› **have** *?P* (*tl*, *s'*) **by** *simp*
hence *?P* (*Eps ?P*) **by**(*rule someI*)
thus *?case* **using** *tls*
by(*subst* (*asm*) *unfold-llist.code*)(*auto*)
qed
thus *thesis* **by**(*rule that*)
qed

lemma τ*Runs-into-*τ*rtrancl3p*:
assumes *runs*: *s* $\Downarrow$ *tlss*
and *fin*: *tfinite tlss*
and *terminal*: *terminal tlss* = *Some s'*
shows τ*rtrancl3p s* (*list-of* (*llist-of-tllist tlss*)) *s'*
using *fin runs terminal*
proof(*induct arbitrary*: *s rule*: *tfinite-induct*)
case *TNil* **thus** *?case* **by** *cases*(*auto intro*: *silent-moves-into-*τ*rtrancl3p*)
next
case (*TCons tl tlss*)
from ‹*s* $\Downarrow$ *TCons tl tlss*› **obtain** *s''* *s'''*
where *step*: $s\ -\tau\rightarrow*\ s''$
and *step2*: $s''\ -tl\rightarrow\ s'''$ $\neg\ \tau$*move s'' tl s'''*

```
    and s''' ⇓ tlss by cases
  from ‹terminal (TCons tl tlss) = ⌊s'⌋› have terminal tlss = ⌊s'⌋ by simp
  with ‹s''' ⇓ tlss› have s''' −τ−list-of (llist-of-tllist tlss)→* s' by(rule TCons)
  with step2 have s'' −τ−tl # list-of (llist-of-tllist tlss)→* s' by(rule τrtrancl3p-step)
  with step have s −τ−[] @ tl # list-of (llist-of-tllist tlss)→* s'
    by(rule τrtrancl3p-trans[OF silent-moves-into-τrtrancl3p])
  thus ?case using ‹tfinite tlss› by simp
qed

lemma τRuns-terminal-stuck:
  assumes Runs: s ⇓ tlss
  and fin: tfinite tlss
  and terminal: terminal tlss = Some s'
  and proceed: s' −tls→ s''
  shows False
using fin Runs terminal
proof(induct arbitrary: s rule: tfinite-induct)
  case TNil thus ?case using proceed by cases auto
next
  case TCons thus ?case by(fastforce elim: τRuns.cases)
qed

lemma Runs-table-silent-diverge:
  ⟦ Runs-table s stlss; ∀(s, tl, s') ∈ lset stlss. τmove s tl s'; ¬ lfinite stlss ⟧
  ⟹ s −τ→ ∞
proof(coinduction arbitrary: s stlss)
  case (τdiverge s)
  thus ?case by cases(auto 5 2)
qed

lemma Runs-table-silent-rtrancl:
  assumes lfinite stlss
  and Runs-table s stlss
  and ∀(s, tl, s') ∈ lset stlss. τmove s tl s'
  shows s −τ→* llast (LCons s (lmap (λ(s, tl, s'). s') stlss)) (is ?thesis1)
  and llast (LCons s (lmap (λ(s, tl, s'). s') stlss)) −tl'→ s'' ⟹ False (is PROP ?thesis2)
proof −
  from assms have ?thesis1 ∧ (llast (LCons s (lmap (λ(s, tl, s'). s') stlss)) −tl'→ s'' ⟶ False)
  proof(induct arbitrary: s)
    case lfinite-LNil thus ?case by(auto elim: Runs-table.cases)
  next
    case (lfinite-LConsI stlss stls)
    from ‹Runs-table s (LCons stls stlss)›
    obtain tl s' where [simp]: stls = (s, tl, s')
      and s −tl→ s' and Run': Runs-table s' stlss by cases
    from ‹∀(s, tl, s')∈lset (LCons stls stlss). τmove s tl s'›
    have τmove s tl s' and silent': ∀(s, tl, s')∈lset stlss. τmove s tl s' by simp-all
    from ‹s −tl→ s'› ‹τmove s tl s'› have s −τ→ s' by auto
    moreover from Run' silent'
    have s' −τ→* llast (LCons s' (lmap (λ(s, tl, s'). s') stlss)) ∧
          (llast (LCons s' (lmap (λ(s, tl, s'). s') stlss)) −tl'→ s'' ⟶ False)
      by(rule lfinite-LConsI)
    ultimately show ?case by(auto)
  qed
```

```
  thus ?thesis1 PROP ?thesis2 by blast+
qed

lemma Runs-table-silent-lappendD:
  fixes s stlss
  defines s' ≡ llast (LCons s (lmap (λ(s, tl, s'). s') stlss))
  assumes Runs: Runs-table s (lappend stlss stlss')
  and fin: lfinite stlss
  and silent: ∀(s, tl, s') ∈ lset stlss. τmove s tl s'
  shows s −τ→* s' (is ?thesis1)
  and Runs-table s' stlss' (is ?thesis2)
  and stlss' ≠ LNil ⟹ s' = fst (lhd stlss') (is PROP ?thesis3)
proof −
  from fin Runs silent
  have ?thesis1 ∧ ?thesis2 ∧ (stlss' ≠ LNil ⟶ s' = fst (lhd stlss'))
    unfolding s'-def
  proof(induct arbitrary: s)
    case lfinite-LNil thus ?case
      by(auto simp add: neq-LNil-conv Runs-table-simps)
  next
    case lfinite-LConsI thus ?case
    by(clarsimp simp add: neq-LNil-conv Runs-table-simps)(blast intro: converse-rtranclp-into-rtranclp)
  qed
  thus ?thesis1 ?thesis2 PROP ?thesis3 by simp-all
qed

lemma Runs-table-into-τRuns:
  fixes s stlss
  defines tls ≡ tmap (λ(s, tl, s'). tl) id (tfilter None (λ(s, tl, s'). ¬ τmove s tl s') (tllist-of-llist (Some
(llast (LCons s (lmap (λ(s, tl, s'). s') stlss)))) stlss))
  (is - ≡ ?conv s stlss)
  assumes Runs-table s stlss
  shows τRuns s tls
using assms
proof(coinduction arbitrary: s tls stlss)
  case (τRuns s tls stlss)
  note tls = ‹tls = ?conv s stlss›
    and Run = ‹Runs-table s stlss›
  show ?case
  proof(cases tls)
    case [simp]: (TNil so)
    from tls
    have silent: ∀(s, tl, s') ∈ lset stlss. τmove s tl s'
      by(auto simp add: TNil-eq-tmap-conv tfilter-empty-conv)
    show ?thesis
    proof(cases lfinite stlss)
      case False
      with Run silent have s −τ→ ∞ by(rule Runs-table-silent-diverge)
      hence ?Diverge using False tls by(simp add: TNil-eq-tmap-conv tfilter-empty-conv)
      thus ?thesis by simp
    next
      case True
      with Runs-table-silent-rtrancl[OF this Run silent]
      have ?Terminate using tls
```

```
      by(auto simp add: TNil-eq-tmap-conv tfilter-empty-conv terminal-tllist-of-llist split-def)
    thus ?thesis by simp
  qed
 next
  case [simp]: (TCons tl tls')
  from tls obtain s' s'' stlss'
   where tl': tfilter None (λ(s, tl, s'). ¬ τmove s tl s') (tllist-of-llist ⌊llast (LCons s (lmap (λ(s, tl,
s'). s') stlss))⌋ stlss) = TCons (s', tl, s'') stlss'
   and tls': tls' = tmap (λ(s, tl, s'). tl) id stlss'
   by(simp add: TCons-eq-tmap-conv split-def id-def split-paired-Ex) blast
  from tfilter-eq-TConsD[OF tl']
  obtain stlsτ rest
   where stlss-eq: tllist-of-llist ⌊llast (LCons s (lmap (λ(s, tl, s'). s') stlss))⌋ stlss = lappendt stlsτ
(TCons (s', tl, s'') rest)
   and fin: lfinite stlsτ
   and silent: ∀(s, tl, s')∈lset stlsτ. τmove s tl s'
   and ¬ τmove s' tl s''
   and stlss': stlss' = tfilter None (λ(s, tl, s'). ¬ τmove s tl s') rest
   by(auto simp add: split-def)
  from stlss-eq fin obtain rest'
   where stlss: stlss = lappend stlsτ rest'
   and rest': tllist-of-llist ⌊llast (LCons s (lmap (λ(s, tl, s'). s') stlss))⌋ rest' = TCons (s', tl, s'')
rest
   unfolding tllist-of-llist-eq-lappendt-conv by auto
  hence rest' ≠ LNil by clarsimp
  from Run[unfolded stlss] fin silent
  have s −τ→* llast (LCons s (lmap (λ(s, tl, s'). s') stlsτ))
   and Runs-table (llast (LCons s (lmap (λ(s, tl, s'). s') stlsτ))) rest'
   and llast (LCons s (lmap (λ(s, tl, s'). s') stlsτ)) = fst (lhd rest')
   by(rule Runs-table-silent-lappendD)+(simp add: ‹rest' ≠ LNil›)
  moreover with rest' ‹rest' ≠ LNil› stlss fin obtain rest''
   where rest': rest' = LCons (s', tl, s'') rest''
   and rest: rest = tllist-of-llist ⌊llast (LCons s'' (lmap (λ(s, tl, s'). s') rest''))⌋ rest''
   by(clarsimp simp add: neq-LNil-conv llast-LCons lmap-lappend-distrib)
  ultimately have s −τ→* s' s' −tl→ s'' Runs-table s'' rest''
   by(simp-all add: Runs-table-simps)
  hence ?Proceed using ‹¬ τmove s' tl s''› tls' stlss' rest
   by(auto simp add: id-def)
  thus ?thesis by simp
 qed
qed

lemma τRuns-table2-into-τRuns:
 τRuns-table2 s tlsstlss
 ⟹ s ⇓ tmap (λ(tls, s', tl, s''). tl) (λx. case x of Inl (tls, s') ⇒ Some s' | Inr - ⇒ None) tlsstlss
proof(coinduction arbitrary: s tlsstlss)
 case (τRuns s tlsstlss)
 thus ?case by cases(auto intro: silent-moves2-into-silent-moves inf-step-silent-move2-into-τdiverge)
qed

lemma τRuns-into-τRuns-table2:
 assumes s ⇓ tls
 obtains tlsstlss
 where τRuns-table2 s tlsstlss
```

and *tls* = *tmap* (λ(*tls*, *s'*, *tl*, *s''*). *tl*) (λ*x*. *case x of Inl* (*tls*, *s'*) ⇒ *Some s'* | *Inr* - ⇒ *None*) *tlsstlss*
proof −
let *?terminal* = λ*s tls*. *case terminal tls of*
None ⇒ *Inr* (*SOME tls'*. *trsys.inf-step silent-move2 s tls'*)
| *Some s'* ⇒ *let tls'* = *SOME tls'*. *silent-moves2 s tls' s' in Inl* (*tls'*, *s'*)
let *?P* = λ*s tls* (*tls''*, *s'*, *s''*). *silent-moves2 s tls'' s'* ∧ *s'* −*thd tls*→ *s''* ∧ ¬ *τmove s'* (*thd tls*) *s''* ∧ *s''* ⇓ *ttl tls*
define *tlsstlss* **where** *tlsstlss s tls* = *unfold-tllist*
(λ(*s*, *tls*). *is-TNil tls*)
(λ(*s*, *tls*). *?terminal s tls*)
(λ(*s*, *tls*). *let* (*tls''*, *s'*, *s''*) = *Eps* (*?P s tls*) *in* (*tls''*, *s'*, *thd tls*, *s''*))
(λ(*s*, *tls*). *let* (*tls''*, *s'*, *s''*) = *Eps* (*?P s tls*) *in* (*s''*, *ttl tls*))
(*s*, *tls*)
for *s tls*

have [*simp*]:
⋀*s tls*. *is-TNil* (*tlsstlss s tls*) ⟷ *is-TNil tls*
⋀*s tls*. *is-TNil tls* ⟹ *terminal* (*tlsstlss s tls*) = *?terminal s tls*
⋀*s tls*. ¬ *is-TNil tls* ⟹ *thd* (*tlsstlss s tls*) = (*let* (*tls''*, *s'*, *s''*) = *Eps* (*?P s tls*) *in* (*tls''*, *s'*, *thd tls*, *s''*))
⋀*s tls*. ¬ *is-TNil tls* ⟹ *ttl* (*tlsstlss s tls*) = (*let* (*tls''*, *s'*, *s''*) = *Eps* (*?P s tls*) *in tlsstlss s''* (*ttl tls*))
by(*simp-all add*: *tlsstlss-def split-beta*)

have [*simp*]:
⋀*s*. *tlsstlss s* (*TNil None*) = *TNil* (*Inr* (*SOME tls'*. *trsys.inf-step silent-move2 s tls'*))
⋀*s s'*. *tlsstlss s* (*TNil* (*Some s'*)) = *TNil* (*Inl* (*SOME tls'*. *silent-moves2 s tls' s'*, *s'*))
unfolding *tlsstlss-def* **by** *simp-all*

let *?conv* = *tmap* (λ(*tls*, *s'*, *tl*, *s''*). *tl*) (λ*x*. *case x of Inl* (*tls*, *s'*) ⇒ *Some s'* | *Inr* - ⇒ *None*)
from *assms* **have** *τRuns-table2 s* (*tlsstlss s tls*)
proof(*coinduction arbitrary*: *s tls*)
case (*τRuns-table2 s tls*)
thus *?case*
proof(*cases*)
case (*Terminate s'*)
let *?P* = λ*tls'*. *silent-moves2 s tls' s'*
from ‹*s* −*τ*→∗ *s'*› **obtain** *tls'* **where** *?P tls'* **by**(*blast dest*: *silent-moves-into-silent-moves2*)
hence *?P* (*Eps ?P*) **by**(*rule someI*)
with *Terminate* **have** *?Terminate* **by** *auto*
thus *?thesis* **by** *simp*
next
case *Diverge*
let *?P* = λ*tls'*. *trsys.inf-step silent-move2 s tls'*
from ‹*s* −*τ*→ ∞› **obtain** *tls'* **where** *?P tls'* **by**(*rule τdiverge-into-inf-step-silent-move2*)
hence *?P* (*Eps ?P*) **by**(*rule someI*)
hence *?Diverge* **using** ‹*tls* = *TNil None*› **by** *simp*
thus *?thesis* **by** *simp*
next
case (*Proceed s' s'' tls' tl*)
from ‹*s* −*τ*→∗ *s'*› **obtain** *tls''* **where** *silent-moves2 s tls'' s'*
by(*blast dest*: *silent-moves-into-silent-moves2*)
with *Proceed* **have** *?P s tls* (*tls''*, *s'*, *s''*) **by** *simp*
hence *?P s tls* (*Eps* (*?P s tls*)) **by**(*rule someI*)
hence *?Proceed* **using** *Proceed* **unfolding** *tlsstlss-def*

```
      by(subst unfold-tllist.code)(auto simp add: split-def)
    thus ?thesis by simp
  qed
 qed
 moreover
 from assms have tls = ?conv (tlsstlss s tls)
 proof(coinduction arbitrary: s tls)
  case (Eq-tllist s tls)
  thus ?case
  proof(cases)
   case (Proceed s' s'' tls' tl)
   from ‹s −τ→* s'› obtain tls'' where silent-moves2 s tls'' s'
    by(blast dest: silent-moves-into-silent-moves2)
   with Proceed have ?P s tls (tls'', s', s'') by simp
   hence ?P s tls (Eps (?P s tls)) by(rule someI)
   thus ?thesis using ‹tls = TCons tl tls'› by auto
  qed auto
 qed
 ultimately show thesis by(rule that)
qed

lemma τRuns-table2-into-Runs:
 assumes τRuns-table2 s tlsstlss
 shows Runs s (lconcat (lappend (lmap (λ(tls, s, tl, s'). llist-of (tls @ [tl])) (llist-of-tllist tlsstlss))
(LCons (case terminal tlsstlss of Inl (tls, s') ⇒ llist-of tls | Inr tls ⇒ tls) LNil)))
 (is Runs - (?conv tlsstlss))
using assms
proof(coinduction arbitrary: s tlsstlss)
 case (Runs s tlsstlss)
 thus ?case
 proof(cases)
  case (Terminate tls' s')
  from ‹silent-moves2 s tls' s'› show ?thesis
  proof(cases rule: rtrancl3p-converseE)
   case refl
   hence ?Stuck using Terminate by simp
   thus ?thesis ..
  next
   case (step tls'' tl s'')
   from ‹silent-moves2 s'' tls'' s'› ‹⋀tl s''. ¬ s' −tl→ s''›
   have τRuns-table2 s'' (TNil (Inl (tls'', s'))) ..
   with ‹tls' = tl # tls''› ‹silent-move2 s tl s''› ‹tlsstlss = TNil (Inl (tls', s'))›
   have ?Step by(auto simp add: silent-move2-def intro!: exI)
   thus ?thesis ..
  qed
 next
  case (Diverge tls')
  from ‹trsys.inf-step silent-move2 s tls'›
  obtain tl tls'' s' where silent-move2 s tl s'
   and tls' = LCons tl tls'' trsys.inf-step silent-move2 s' tls''
   by(cases rule: trsys.inf-step.cases[consumes 1]) auto
  from ‹trsys.inf-step silent-move2 s' tls''›
  have τRuns-table2 s' (TNil (Inr tls'')) ..
  hence ?Step using ‹tlsstlss = TNil (Inr tls')› ‹tls' = LCons tl tls''› ‹silent-move2 s tl s'›
```

```
      by(auto simp add: silent-move2-def intro!: exI)
    thus ?thesis ..
  next
    case (Proceed tls' s' s'' tlsstlss' tl)
    from ‹silent-moves2 s tls' s'› have ?Step
    proof(cases rule: rtrancl3p-converseE)
      case refl with Proceed show ?thesis by auto
    next
      case (step tls'' tl' s''')
      from ‹silent-moves2 s''' tls'' s'› ‹s' −tl→ s''› ‹¬ τmove s' tl s''› ‹τRuns-table2 s'' tlsstlss'›
      have τRuns-table2 s''' (TCons (tls'', s', tl, s'') tlsstlss') ..
      with ‹tls' = tl' # tls''› ‹silent-move2 s tl' s'''› ‹tlsstlss = TCons (tls', s', tl, s'') tlsstlss'›
      show ?thesis by(auto simp add: silent-move2-def intro!: exI)
    qed
    thus ?thesis ..
  qed
qed

lemma τRuns-table2-silentsD:
  fixes tl
  assumes Runs: τRuns-table2 s tlsstlss
  and tset: (tls, s', tl', s'') ∈ tset tlsstlss
  and set: tl ∈ set tls
  shows ∃ s''' s''''. silent-move2 s''' tl s''''
using tset Runs
proof(induct arbitrary: s rule: tset-induct)
  case (find tlsstlss')
  from ‹τRuns-table2 s (TCons (tls, s', tl', s'') tlsstlss')›
  have silent-moves2 s tls s' by cases
  thus ?case using set by induct auto
next
  case step thus ?case by(auto simp add: τRuns-table2-simps)
qed

lemma τRuns-table2-terminal-silentsD:
  assumes Runs: τRuns-table2 s tlsstlss
  and fin: lfinite (llist-of-tllist tlsstlss)
  and terminal: terminal tlsstlss = Inl (tls, s'')
  shows ∃ s'. silent-moves2 s' tls s''
using fin Runs terminal
proof(induct llist-of-tllist tlsstlss arbitrary: tlsstlss s)
  case lfinite-LNil thus ?case
    by(cases tlsstlss)(auto simp add: τRuns-table2-simps)
next
  case (lfinite-LConsI xs tlsstls)
  thus ?case by(cases tlsstlss)(auto simp add: τRuns-table2-simps)
qed

lemma τRuns-table2-terminal-inf-stepD:
  assumes Runs: τRuns-table2 s tlsstlss
  and fin: lfinite (llist-of-tllist tlsstlss)
  and terminal: terminal tlsstlss = Inr tls
  shows ∃ s'. trsys.inf-step silent-move2 s' tls
using fin Runs terminal
```

proof(*induct llist-of-tllist tlsstlss arbitrary*: *s tlsstlss*)
case *lfinite-LNil* **thus** *?case*
by(*cases tlsstlss*)(*auto simp add*: *τRuns-table2-simps*)
next
case (*lfinite-LConsI xs tlsstls*)
thus *?case* **by**(*cases tlsstlss*)(*auto simp add*: *τRuns-table2-simps*)
qed

lemma *τRuns-table2-lappendtD*:
assumes *Runs*: *τRuns-table2 s* (*lappendt tlsstlss tlsstlss′*)
and *fin*: *lfinite tlsstlss*
shows ∃ *s′*. *τRuns-table2 s′ tlsstlss′*
using *fin Runs*
by(*induct arbitrary*: *s*)(*auto simp add*: *τRuns-table2-simps*)

end

lemma *τmoves-False*: *τtrsys.silent-move r* (*λs ta s′*. *False*) = (*λs s′*. *False*)
by(*auto simp add*: *τtrsys.silent-move-iff*)

lemma *τrtrancl3p-False-eq-rtrancl3p*: *τtrsys.τrtrancl3p r* (*λs tl s′*. *False*) = *rtrancl3p r*
proof(*intro ext iffI*)
fix *s tls s′*
assume *τtrsys.τrtrancl3p r* (*λs tl s′*. *False*) *s tls s′*
thus *rtrancl3p r s tls s′* **by**(*rule τtrsys.τrtrancl3p.induct*)(*blast intro*: *rtrancl3p-step-converse*)+
next
fix *s tls s′*
assume *rtrancl3p r s tls s′*
thus *τtrsys.τrtrancl3p r* (*λs tl s′*. *False*) *s tls s′*
by(*induct rule*: *rtrancl3p-converse-induct*)(*auto intro*: *τtrsys.τrtrancl3p.intros*)
qed

lemma *τdiverge-empty-τmove*:
τtrsys.τdiverge r (*λs ta s′*. *False*) = (*λs*. *False*)
by(*auto intro*!: *ext elim*: *τtrsys.τdiverge.cases τtrsys.silent-move.cases*)

end

1.16 The multithreaded semantics as a labelled transition system

theory *FWLTS*
imports
FWProgressAux
FWLifting
LTS
begin

sublocale *multithreaded-base* < *trsys r t* **for** *t* **.**
sublocale *multithreaded-base* < *mthr*: *trsys redT* **.**

— Move to FWSemantics?
definition *redT-upd-ε* :: (*'l*,*'t*,*'x*,*'m*,*'w*) *state* ⇒ *'t* ⇒ *'x* ⇒ *'m* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*) *state*

where [*simp*]: *redT-upd-ε s t x′ m′* = (*locks s*, ((*thr s*)(*t* ↦ (*x′*, *snd* (*the* (*thr s t*)))), *m′*), *wset s*, *interrupts s*)

lemma *redT-upd-ε-redT-upd*:
redT-upd s t ε x′ m′ (*redT-upd-ε s t x′ m′*)
by(*auto simp add*: *redT-updLns-def redT-updWs-def*)

context *multithreaded* **begin**

sublocale *trsys r t* **for** *t* **.**

sublocale *mthr*: *trsys redT* **.**

end

1.16.1 The multithreaded semantics with internal actions

type-synonym
('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *τmoves* =
'*x* × '*m* ⇒ ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* ⇒ '*x* × '*m* ⇒ *bool*

pretty printing for *τmoves*

print-translation ‹
let
fun tr′ [(*Const* (@{*type-syntax prod*}, -) $ *x1* $ *m1*),
(*Const* (@{*type-syntax fun*}, -) $
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax finfun*}, -) $ *l* $
(*Const* (@{*type-syntax list*}, -) $ *Const* (@{*type-syntax lock-action*}, -))) $
(*Const* (@{*type-syntax prod*},-) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax new-thread-action*}, -) $ *t1* $ *x2* $ *m2*)) $
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax conditional-action*}, -) $ *t2*)) $
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax wait-set-action*}, -) $ *t3* $ *w*)) $
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax interrupt-action*}, -) $ *t4*)) $
(*Const* (@{*type-syntax list*}, -) $ *o1*))))))) $
(*Const* (@{*type-syntax fun*}, -) $
(*Const* (@{*type-syntax prod*}, -) $ *x3* $ *m3*) $
(*Const* (@{*type-syntax bool*}, -))))] =
if x1 = *x2 andalso x1* = *x3 andalso m1* = *m2 andalso m1* = *m3 andalso t1* = *t2 andalso t2* = *t3 andalso t3* = *t4*
then Syntax.const (@{*type-syntax τmoves*}) $ *l* $ *t1* $ *x1* $ *m1* $ *w* $ *o1*
else raise Match;
in [(@{*type-syntax fun*}, *K tr′*)]
end
›
typ ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *τmoves*

locale *τmultithreaded = multithreaded-base +*
  constrains *final :: 'x ⇒ bool*
  and *r :: ('l,'t,'x,'m,'w,'o) semantics*
  and *convert-RA :: 'l released-locks ⇒ 'o list*
  fixes *τmove :: ('l,'t,'x,'m,'w,'o) τmoves*

sublocale *τmultithreaded < τtrsys r t τmove* **for** *t* **.**

context *τmultithreaded* **begin**

inductive *mτmove :: (('l,'t,'x,'m,'w) state, 't × ('l,'t,'x,'m,'w,'o) thread-action) trsys*
where
  ⟦ thr s t = ⌊(x, no-wait-locks)⌋; thr s' t = ⌊(x', ln')⌋; τmove (x, shr s) ta (x', shr s') ⟧
  ⟹ mτmove s (t, ta) s'

end

sublocale *τmultithreaded < mthr: τtrsys redT mτmove* **.**

context *τmultithreaded* **begin**

abbreviation *τmredT :: ('l,'t,'x,'m,'w) state ⇒ ('l,'t,'x,'m,'w) state ⇒ bool*
where *τmredT == mthr.silent-move*

end

lemma (**in** *multithreaded-base*) *τrtrancl3p-redT-thread-not-disappear:*
  assumes *τtrsys.τrtrancl3p redT τmove s ttas s' thr s t ≠ None*
  shows *thr s' t ≠ None*
proof −
  interpret *T: τtrsys redT τmove* **.**
  show *?thesis*
  proof
    assume *thr s' t = None*
    with *‹τtrsys.τrtrancl3p redT τmove s ttas s'›* **have** *thr s t = None*
    by(*induct rule: T.τrtrancl3p.induct*)(*auto simp add: split-paired-all dest: redT-thread-not-disappear*)
    with *‹thr s t ≠ None›* **show** *False* **by** *contradiction*
  qed
qed

lemma *mτmove-False: τmultithreaded.mτmove (λs ta s'. False) = (λs ta s'. False)*
by(*auto intro*!: *ext elim: τmultithreaded.mτmove.cases*)

declare *split-paired-Ex* [*simp del*]

locale *τmultithreaded-wf =*
  τmultithreaded - - - τmove +
  multithreaded final r convert-RA
  for *τmove :: ('l,'t,'x,'m,'w,'o) τmoves +*
  assumes *τmove-heap: ⟦ t ⊢ (x, m) −ta→ (x', m'); τmove (x, m) ta (x', m') ⟧ ⟹ m = m'*
  assumes *silent-tl: τmove s ta s' ⟹ ta = ε*
begin

lemma *mτmove-silentD: mτmove s (t, ta) s' ⟹ ta = (K$ [], [], [], [], [], [])*

by(*auto elim*!: *mτmove.cases dest*: *silent-tl*)

lemma *mτmove-heap*:
assumes *redT*: *redT s* (*t*, *ta*) *s′*
and *mτmove*: *mτmove s* (*t*, *ta*) *s′*
shows *shr s′* = *shr s*
using *mτmove redT*
by *cases*(*auto dest*: *τmove-heap elim*!: *redT.cases*)

lemma *τmredT-thread-preserved*:
τmredT s s′ ⟹ *thr s t* = *None* ⟷ *thr s′ t* = *None*
by(*auto simp add*: *mthr.silent-move-iff elim*!: *redT.cases dest*!: *mτmove-silentD split*: *if-split-asm*)

lemma *τmRedT-thread-preserved*:
τmredT^** *s s′* ⟹ *thr s t* = *None* ⟷ *thr s′ t* = *None*
by(*induct rule*: *rtranclp.induct*)(*auto dest*: *τmredT-thread-preserved*[**where** *t*=*t*])

lemma *τmtRedT-thread-preserved*:
τmredT^++ *s s′* ⟹ *thr s t* = *None* ⟷ *thr s′ t* = *None*
by(*induct rule*: *tranclp.induct*)(*auto dest*: *τmredT-thread-preserved*[**where** *t*=*t*])

lemma *τmredT-add-thread-inv*:
assumes *τred*: *τmredT s s′* **and** *tst*: *thr s t* = *None*
shows *τmredT* (*locks s*, ((*thr s*)(*t* ↦ *xln*), *shr s*), *wset s*, *interrupts s*) (*locks s′*, ((*thr s′*)(*t* ↦ *xln*), *shr s′*), *wset s′*, *interrupts s′*)
proof −
obtain *ls ts m ws is* **where** *s*: *s* = (*ls*, (*ts*, *m*), *ws*, *is*) **by**(*cases s*) *fastforce*
obtain *ls′ ts′ m′ ws′ is′* **where** *s′*: *s′* = (*ls′*, (*ts′*, *m′*), *ws′*, *is′*) **by**(*cases s′*) *fastforce*
from *τred s s′* **obtain** *t′* **where** *red*: (*ls*, (*ts*, *m*), *ws*, *is*) −*t′*▷*ε*→ (*ls′*, (*ts′*, *m′*), *ws′*, *is′*)
and *τ*: *mτmove* (*ls*, (*ts*, *m*), *ws*, *is*) (*t′*, *ε*) (*ls′*, (*ts′*, *m′*), *ws′*, *is′*)
by(*auto simp add*: *mthr.silent-move-iff dest*: *mτmove-silentD*)
from *red* **have** (*ls*, (*ts*(*t* ↦ *xln*), *m*), *ws*, *is*) −*t′*▷*ε*→ (*ls′*, (*ts′*(*t* ↦ *xln*), *m′*), *ws′*, *is′*)
proof(*cases rule*: *redT-elims*)
case (*normal x x′ m′*) **with** *tst s* **show** *?thesis*
by−(*rule redT-normal*, *auto simp add*: *fun-upd-twist elim*!: *rtrancl3p-cases*)
next
case (*acquire x ln n*)
with *tst s* **show** *?thesis*
unfolding ‹*ε* = (*K*$ [], [], [], [], [], *convert-RA ln*)›
by −(*rule redT-acquire*, *auto intro*: *fun-upd-twist*)
qed
moreover from *red tst s* **have** *tt′*: *t* ≠ *t′* **by**(*cases*) *auto*
have (λ*t″*. (*ts*(*t* ↦ *xln*)) *t″* ≠ *None* ∧ (*ts*(*t* ↦ *xln*)) *t″* ≠ (*ts′*(*t* ↦ *xln*)) *t″*) =
(λ*t″*. *ts t″* ≠ *None* ∧ *ts t″* ≠ *ts′ t″*) **using** *tst s* **by**(*auto simp add*: *fun-eq-iff*)
with *τ tst tt′* **have** *mτmove* (*ls*, (*ts*(*t* ↦ *xln*), *m*), *ws*, *is*) (*t′*, *ε*) (*ls′*, (*ts′*(*t* ↦ *xln*), *m′*), *ws′*, *is′*)
by *cases*(*rule mτmove.intros*, *auto*)
ultimately show *?thesis* **unfolding** *s s′* **by** *auto*
qed

lemma *τmRedT-add-thread-inv*:
⟦ *τmredT*^** *s s′*; *thr s t* = *None* ⟧
⟹ *τmredT*^** (*locks s*, ((*thr s*)(*t* ↦ *xln*), *shr s*), *wset s*, *interrupts s*) (*locks s′*, ((*thr s′*)(*t* ↦ *xln*), *shr s′*), *wset s′*, *interrupts s′*)
apply(*induct rule*: *rtranclp-induct*)

apply(*blast dest*: τ*mRedT-thread-preserved*[**where** *t*=*t*] τ*mredT-add-thread-inv*[**where** *xln*=*xln*] *intro*: *rtranclp.rtrancl-into-rtrancl*)+
done

lemma τ*mtRed-add-thread-inv*:
⟦ τ*mredT*^++ *s s'*; *thr s t* = *None* ⟧
⟹ τ*mredT*^++ (*locks s*, ((*thr s*)(*t* ↦ *xln*), *shr s*), *wset s*, *interrupts s*) (*locks s'*, ((*thr s'*)(*t* ↦ *xln*), *shr s'*), *wset s'*, *interrupts s'*)
apply(*induct rule*: *tranclp-induct*)
apply(*blast dest*: τ*mtRedT-thread-preserved*[**where** *t*=*t*] τ*mredT-add-thread-inv*[**where** *xln*=*xln*] *intro*: *tranclp.trancl-into-trancl*)+
done

lemma *silent-move-into-RedT-*τ*-inv*:
assumes *move*: *silent-move t* (*x*, *shr s*) (*x'*, *m'*)
and *state*: *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ *wset s t* = *None*
shows τ*mredT s* (*redT-upd-*ε *s t x' m'*)
proof −
from *move* **obtain** *red*: *t* ⊢ (*x*, *shr s*) −ε→ (*x'*, *m'*) **and** τ: τ*move* (*x*, *shr s*) ε (*x'*, *m'*)
by(*auto simp add*: *silent-move-iff dest*: *silent-tl*)
from *red state* **have** *s* −*t*▷ε→ *redT-upd-*ε *s t x' m'*
by −(*rule redT-normal*, *auto simp add*: *redT-updLns-def o-def finfun-Diag-const2 redT-updWs-def*)
moreover from τ *red state* **have** *m*τ*move s* (*t*, ε) (*redT-upd-*ε *s t x' m'*)
by −(*rule m*τ*move.intros*, *auto dest*: τ*move-heap simp add*: *redT-updLns-def*)
ultimately show *?thesis* **by** *auto*
qed

lemma *silent-moves-into-RedT-*τ*-inv*:
assumes *major*: *silent-moves t* (*x*, *shr s*) (*x'*, *m'*)
and *state*: *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ *wset s t* = *None*
shows τ*mredT*^** *s* (*redT-upd-*ε *s t x' m'*)
using *major*
proof(*induct rule*: *rtranclp-induct2*)
case *refl* **with** *state* **show** *?case* **by**(*cases s*)(*auto simp add*: *fun-upd-idem*)
next
case (*step x' m' x'' m''*)
from ‹*silent-move t* (*x'*, *m'*) (*x''*, *m''*)› *state*
have τ*mredT* (*redT-upd-*ε *s t x' m'*) (*redT-upd-*ε (*redT-upd-*ε *s t x' m'*) *t x'' m''*)
by −(*rule silent-move-into-RedT-*τ*-inv*, *auto*)
hence τ*mredT* (*redT-upd-*ε *s t x' m'*) (*redT-upd-*ε *s t x'' m''*) **by**(*simp*)
with ‹τ*mredT*^** *s* (*redT-upd-*ε *s t x' m'*)› **show** *?case* **..**
qed

lemma *red-rtrancl-*τ*-heapD-inv*:
⟦ *silent-moves t s s'*; *wfs t s* ⟧ ⟹ *snd s'* = *snd s*
proof(*induct rule*: *rtranclp-induct*)
case *base* **show** *?case* **..**
next
case (*step s' s''*)
thus *?case* **by**(*cases s*, *cases s'*, *cases s''*)(*auto dest*: τ*move-heap*)
qed

lemma *red-trancl-*τ*-heapD-inv*:
⟦ *silent-movet t s s'*; *wfs t s* ⟧ ⟹ *snd s'* = *snd s*

proof(*induct rule*: *tranclp-induct*)
case (*base s'*) **thus** *?case* **by**(*cases s'*)(*cases s*, *auto simp add*: *silent-move-iff dest*: *τmove-heap*)
next
case (*step s' s''*)
thus *?case* **by**(*cases s*, *cases s'*, *cases s''*)(*auto simp add*: *silent-move-iff dest*: *τmove-heap*)
qed

lemma *red-trancl-τ-into-RedT-τ-inv*:
assumes *major*: *silent-movet t* (*x*, *shr s*) (*x'*, *m'*)
and *state*: *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ *wset s t* = *None*
shows *τmredT*^++ *s* (*redT-upd-ε s t x' m'*)
using *major*
proof(*induct rule*: *tranclp-induct2*)
case (*base x' m'*)
thus *?case* **using** *state*
by −(*rule tranclp.r-into-trancl*, *rule silent-move-into-RedT-τ-inv*, *auto*)
next
case (*step x' m' x'' m''*)
hence *τmredT*^++ *s* (*redT-upd-ε s t x' m'*) **by** *blast*
moreover from ‹*silent-move t* (*x'*, *m'*) (*x''*, *m''*)› *state*
have *τmredT* (*redT-upd-ε s t x' m'*) (*redT-upd-ε* (*redT-upd-ε s t x' m'*) *t x'' m''*)
by −(*rule silent-move-into-RedT-τ-inv*, *auto simp add*: *redT-updLns-def*)
hence *τmredT* (*redT-upd-ε s t x' m'*) (*redT-upd-ε s t x'' m''*)
by(*simp add*: *redT-updLns-def*)
ultimately show *?case* **..**
qed

lemma *τdiverge-into-τmredT*:
assumes *τdiverge t* (*x*, *shr s*)
and *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ *wset s t* = *None*
shows *mthr.τdiverge s*
using *assms*
proof(*coinduction arbitrary*: *s x*)
case (*τdiverge s x*)
note *tst* = ‹*thr s t* = ⌊(*x*, *no-wait-locks*)⌋›
from ‹*τdiverge t* (*x*, *shr s*)› **obtain** *x' m'* **where** *silent-move t* (*x*, *shr s*) (*x'*, *m'*)
and *τdiverge t* (*x'*, *m'*) **by** *cases auto*
from ‹*silent-move t* (*x*, *shr s*) (*x'*, *m'*)› *tst* ‹*wset s t* = *None*›
have *τmredT s* (*redT-upd-ε s t x' m'*) **by**(*rule silent-move-into-RedT-τ-inv*)
moreover have *thr* (*redT-upd-ε s t x' m'*) *t* = ⌊(*x'*, *no-wait-locks*)⌋
using *tst* **by**(*auto simp add*: *redT-updLns-def*)
moreover have *wset* (*redT-upd-ε s t x' m'*) *t* = *None* **using** ‹*wset s t* = *None*› **by** *simp*
moreover from ‹*τdiverge t* (*x'*, *m'*)› **have** *τdiverge t* (*x'*, *shr* (*redT-upd-ε s t x' m'*)) **by** *simp*
ultimately show *?case* **using** ‹*τdiverge t* (*x'*, *m'*)› **by** *blast*
qed

lemma *τdiverge-τmredTD*:
assumes *div*: *mthr.τdiverge s*
and *fin*: *finite* (*dom* (*thr s*))
shows ∃ *t x*. *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ ∧ *wset s t* = *None* ∧ *τdiverge t* (*x*, *shr s*)
using *fin div*
proof(*induct A*≡*dom* (*thr s*) *arbitrary*: *s rule*: *finite-induct*)
case *empty*
from ‹*mthr.τdiverge s*› **obtain** *s'* **where** *τmredT s s'* **by** *cases auto*

```
  with ‹{} = dom (thr s)›[symmetric] have False by(auto elim!: mthr.silent-move.cases redT.cases)
  thus ?case ..
next
  case (insert t A)
  note IH = ‹⋀s. ⟦ A = dom (thr s); mthr.τdiverge s ⟧
          ⟹ ∃t x. thr s t = ⌊(x, no-wait-locks)⌋ ∧ wset s t = None ∧ τdiverge t (x, shr s)›
  from ‹insert t A = dom (thr s)›
  obtain x ln where tst: thr s t = ⌊(x, ln)⌋ by(fastforce simp add: dom-def)
  define s' where s' = (locks s, ((thr s)(t := None)), shr s), wset s, interrupts s)
  show ?case
  proof(cases ln = no-wait-locks ∧ τdiverge t (x, shr s) ∧ wset s t = None)
    case True
    with tst show ?thesis by blast
  next
    case False
    define xm where xm = (x, shr s)
    define xm' where xm' = (x, shr s)
    have A = dom (thr s') using ‹t ∉ A› ‹insert t A = dom (thr s)›
      unfolding s'-def by auto
    moreover {
      from xm'-def tst ‹mthr.τdiverge s› False
      have ∃s x. thr s t = ⌊(x, ln)⌋ ∧ (ln ≠ no-wait-locks ∨ wset s t ≠ None ∨ ¬ τdiverge t xm') ∧
                 s' = (locks s, ((thr s)(t := None)), shr s), wset s, interrupts s) ∧ xm = (x, shr s) ∧
                 mthr.τdiverge s ∧ silent-moves t xm' xm
        unfolding s'-def xm-def by blast
      moreover
      from False have wfP (if τdiverge t xm' then (λs s'. False) else flip (silent-move-from t xm'))
        using τdiverge-neq-wfP-silent-move-from[of t (x, shr s)] unfolding xm'-def by(auto)
      ultimately have mthr.τdiverge s'
      proof(coinduct s' xm rule: mthr.τdiverge-trancl-measure-coinduct)
        case (τdiverge s' xm)
        then obtain s x where tst: thr s t = ⌊(x, ln)⌋
          and blocked: ln ≠ no-wait-locks ∨ wset s t ≠ None ∨ ¬ τdiverge t xm'
          and s'-def: s' = (locks s, ((thr s)(t := None)), shr s), wset s, interrupts s)
          and xm-def: xm = (x, shr s)
          and xmxm': silent-moves t xm' (x, shr s)
          and mthr.τdiverge s by blast
        from ‹mthr.τdiverge s› obtain s'' where τmredT s s'' mthr.τdiverge s'' by cases auto
        from ‹τmredT s s''› obtain t' ta where s -t'▷ta→ s'' and mτmove s (t', ta) s'' by auto
        then obtain x' x'' m'' where red: t' ⊢ ⟨x', shr s⟩ -ta→ ⟨x'', m''⟩
          and tst': thr s t' = ⌊(x', no-wait-locks)⌋
          and aoe: actions-ok s t' ta
          and s'': redT-upd s t' ta x'' m'' s''
          by cases(fastforce elim: mτmove.cases)+
        from ‹mτmove s (t', ta) s''› have [simp]: ta = ε
          by(auto elim!: mτmove.cases dest!: silent-tl)
        hence wst': wset s t' = None using aoe by auto
        from ‹mτmove s (t', ta) s''› tst' s''
        have τmove (x', shr s) ε (x'', m'') by(auto elim: mτmove.cases)
        show ?case
        proof(cases t' = t)
          case False
          with tst' wst' have thr s' t' = ⌊(x', no-wait-locks)⌋
            wset s' t' = None shr s' = shr s unfolding s'-def by auto
```

with *red* **have** $s' -t'\triangleright\varepsilon\rightarrow$ *redT-upd-ε s' t' x'' m''*
by −(*rule redT-normal*, *auto simp add*: *redT-updLns-def o-def finfun-Diag-const2 redT-updWs-def*)
moreover from ‹τ*move* (*x'*, *shr s*) ε (*x''*, *m''*)› ‹*thr s' t'* = ⌊(*x'*, *no-wait-locks*)⌋› ‹*shr s'* = *shr s*›
have *mτmove s'* (*t'*, *ta*) (*redT-upd-ε s' t' x'' m''*)
by −(*rule mτmove.intros*, *auto*)
ultimately have *τmredT s'* (*redT-upd-ε s' t' x'' m''*)
unfolding ‹*ta* = ε› **by**(*rule mthr.silent-move.intros*)
hence *τmredT*^++ *s'* (*redT-upd-ε s' t' x'' m''*) **..**
moreover have *thr s'' t* = ⌊(*x*, *ln*)⌋
using *tst* ‹*t'* ≠ *t*› *s''* **by** *auto*
moreover from ‹τ*move* (*x'*, *shr s*) ε (*x''*, *m''*)› *red*
have [*simp*]: *m''* = *shr s* **by**(*auto dest*: *τmove-heap*)
hence *shr s* = *shr s''* **using** *s''* **by**(*auto*)
have *ln* ≠ *no-wait-locks* ∨ *wset s'' t* ≠ *None* ∨ ¬ *τdiverge t xm'*
using *blocked s''* **by**(*auto simp add*: *redT-updWs-def elim*!: *rtrancl3p-cases*)
moreover have *redT-upd-ε s' t' x'' m''* = (*locks s''*, ((*thr s''*)(*t* := *None*)), *shr s''*), *wset s''*, *interrupts s''*)
unfolding *s'-def* **using** *tst s''* ‹*t'* ≠ *t*›
by(*auto intro*: *ext elim*!: *rtrancl3p-cases simp add*: *redT-updLns-def redT-updWs-def*)
ultimately show *?thesis* **using** ‹*mthr.τdiverge s''*› *xmxm'*
unfolding ‹*shr s* = *shr s''*› **by** *blast*
next
case *True*
with *tst tst' wst' blocked* **have** ¬ *τdiverge t xm'*
and [*simp*]: *x'* = *x* **by** *auto*
moreover from *red* ‹τ*move* (*x'*, *shr s*) ε (*x''*, *m''*)› *True*
have *silent-move t* (*x*, *shr s*) (*x''*, *m''*) **by** *auto*
with *xmxm'* **have** *silent-move-from t xm'* (*x*, *shr s*) (*x''*, *m''*)
by(*rule silent-move-fromI*)
ultimately have (*if τdiverge t xm' then λs s'. False else flip* (*silent-move-from t xm'*)) (*x''*, *m''*) *xm*
by(*auto simp add*: *flip-conv xm-def*)
moreover have *thr s'' t* = ⌊(*x''*, *ln*)⌋ **using** *tst True s''*
by(*auto simp add*: *redT-updLns-def*)
moreover from ‹τ*move* (*x'*, *shr s*) ε (*x''*, *m''*)› *red*
have [*simp*]: *m''* = *shr s* **by**(*auto dest*: *τmove-heap*)
hence *shr s* = *shr s''* **using** *s''* **by** *auto*
have *s'* = (*locks s''*, ((*thr s''*)(*t* := *None*)), *shr s''*), *wset s''*, *interrupts s''*)
using *True s''* **unfolding** *s'-def*
by(*auto intro*: *ext elim*!: *rtrancl3p-cases simp add*: *redT-updLns-def redT-updWs-def*)
moreover have (*x''*, *m''*) = (*x''*, *shr s''*) **using** *s''* **by** *auto*
moreover from *xmxm'* ‹*silent-move t* (*x*, *shr s*) (*x''*, *m''*)›
have *silent-moves t xm'* (*x''*, *shr s''*)
unfolding ‹*m''* = *shr s*› ‹*shr s* = *shr s''*› **by** *auto*
ultimately show *?thesis* **using** ‹¬ *τdiverge t xm'*› ‹*mthr.τdiverge s''*› **by** *blast*
qed
qed **}**
ultimately have ∃ *t x*. *thr s' t* = ⌊(*x*, *no-wait-locks*)⌋ ∧ *wset s' t* = *None* ∧ *τdiverge t* (*x*, *shr s'*) **by**(*rule IH*)
then obtain *t' x'* **where** *thr s' t'* = ⌊(*x'*, *no-wait-locks*)⌋
and *wset s' t'* = *None* **and** *τdiverge t'* (*x'*, *shr s'*) **by** *blast*
moreover with *False* **have** *t'* ≠ *t* **by**(*auto simp add*: *s'-def*)
ultimately have *thr s t'* = ⌊(*x'*, *no-wait-locks*)⌋ *wset s t'* = *None τdiverge t'* (*x'*, *shr s*)

unfolding *s'-def* **by** *auto*
thus *?thesis* **by** *blast*
qed
qed

lemma *τmredT-preserves-final-thread*:
⟦ *τmredT s s'*; *final-thread s t* ⟧ ⟹ *final-thread s' t*
by(*auto elim*: *mthr.silent-move.cases intro*: *redT-preserves-final-thread*)

lemma *τmRedT-preserves-final-thread*:
⟦ *τmredT^** s s'*; *final-thread s t* ⟧ ⟹ *final-thread s' t*
by(*induct rule*: *rtranclp.induct*)(*blast intro*: *τmredT-preserves-final-thread*)+

lemma *silent-moves2-silentD*:
assumes *rtrancl3p mthr.silent-move2 s ttas s'*
and (*t*, *ta*) ∈ *set ttas*
shows *ta* = ε
using *assms*
by(*induct*)(*auto simp add*: *mthr.silent-move2-def dest*: *mτmove-silentD*)

lemma *inf-step-silentD*:
assumes *step*: *trsys.inf-step mthr.silent-move2 s ttas*
and *lset*: (*t*, *ta*) ∈ *lset ttas*
shows *ta* = ε
using *lset step*
by(*induct arbitrary*: *s rule*: *lset-induct*)(*fastforce elim*: *trsys.inf-step.cases simp add*: *mthr.silent-move2-def dest*: *mτmove-silentD*)+

end

1.16.2 The multithreaded semantics with a well-founded relation on states

locale *multithreaded-base-measure* = *multithreaded-base* +
constrains *final* :: *'x* ⇒ *bool*
and *r* :: (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *semantics*
and *convert-RA* :: *'l released-locks* ⇒ *'o list*
fixes *μ* :: (*'x* × *'m*) ⇒ (*'x* × *'m*) ⇒ *bool*
begin

inductive *mμt* :: *'m* ⇒ (*'l*,*'t*,*'x*) *thread-info* ⇒ (*'l*,*'t*,*'x*) *thread-info* ⇒ *bool*
for *m* **and** *ts* **and** *ts'*
where
mμtI:
⋀*ln*. ⟦ *finite* (*dom ts*); *ts t* = ⌊(*x*, *ln*)⌋; *ts' t* = ⌊(*x'*, *ln'*)⌋; *μ* (*x*, *m*) (*x'*, *m*); ⋀*t'*. *t'* ≠ *t* ⟹ *ts t'* = *ts' t'* ⟧
⟹ *mμt m ts ts'*

definition *mμ* :: (*'l*,*'t*,*'x*,*'m*,*'w*) *state* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*) *state* ⇒ *bool*
where *mμ s s'* ⟷ *shr s* = *shr s'* ∧ *mμt* (*shr s*) (*thr s*) (*thr s'*)

lemma *mμt-thr-dom-eq*: *mμt m ts ts'* ⟹ *dom ts* = *dom ts'*
apply(*erule mμt.cases*)
apply(*rule equalityI*)
apply(*rule subsetI*)

```
 apply(case-tac xa = t)
  apply(auto)[2]
apply(rule subsetI)
apply(case-tac xa = t)
apply auto
done

lemma mμ-finite-thrD:
  assumes mμt m ts ts'
  shows finite (dom ts) finite (dom ts')
using assms
by(simp-all add: mμt-thr-dom-eq[symmetric])(auto elim: mμt.cases)

end

locale multithreaded-base-measure-wf = multithreaded-base-measure +
  constrains final :: 'x ⇒ bool
  and r :: ('l,'t,'x,'m,'w,'o) semantics
  and convert-RA :: 'l released-locks ⇒ 'o list
  and μ :: ('x × 'm) ⇒ ('x × 'm) ⇒ bool
  assumes wf-μ: wfP μ
begin

lemma wf-mμt: wfP (mμt m)
unfolding wfp-eq-minimal
proof(intro strip)
  fix Q :: ('l,'t,'x) thread-info set and ts
  assume ts ∈ Q
  show ∃ z∈Q. ∀ y. mμt m y z ⟶ y ∉ Q
  proof(cases finite (dom ts))
    case False
    hence ∀ y. mμt m y ts ⟶ y ∉ Q by(auto dest: mμ-finite-thrD)
    thus ?thesis using ‹ts ∈ Q› by blast
  next
    case True
    thus ?thesis using ‹ts ∈ Q›
    proof(induct A≡dom ts arbitrary: ts Q rule: finite-induct)
      case empty
      hence dom ts = {} by simp
      with ‹ts ∈ Q› show ?case by(auto elim: mμt.cases)
    next
      case (insert t A)
      note IH = ‹⋀ts Q. ⟦A = dom ts; ts ∈ Q⟧ ⟹ ∃ z∈Q. ∀ y. mμt m y z ⟶ y ∉ Q›
      define Q' where Q' = {ts. ts t = None ∧ (∃ xln. ts(t ↦ xln) ∈ Q)}
      let ?ts' = ts(t := None)
      from ‹insert t A = dom ts› ‹t ∉ A› have A = dom ?ts' by auto
      moreover from ‹insert t A = dom ts› obtain xln where ts t = ⌊xln⌋ by(cases ts t) auto
      hence ts(t ↦ xln) = ts by(auto simp add: fun-eq-iff)
      with ‹ts ∈ Q› have ts(t ↦ xln) ∈ Q by(auto)
      hence ?ts' ∈ Q' unfolding Q'-def by(auto simp del: split-paired-Ex)
      ultimately have ∃ z∈Q'. ∀ y. mμt m y z ⟶ y ∉ Q' by(rule IH)
      then obtain ts' where ts' ∈ Q'
        and min: ⋀ts''. mμt m ts'' ts' ⟹ ts'' ∉ Q' by blast
      from ‹ts' ∈ Q'› obtain x' ln' where ts' t = None ts'(t ↦ (x', ln')) ∈ Q
```

```
      unfolding Q'-def by auto
    define Q'' where Q'' = {(x, m)|x. ∃ ln. ts'(t ↦ (x, ln)) ∈ Q}
    from ‹ts'(t ↦ (x', ln')) ∈ Q› have (x', m) ∈ Q'' unfolding Q''-def by blast
    hence ∃ xm''∈Q''. ∀ xm'''. μ xm''' xm'' ⟶ xm''' ∉ Q'' by(rule wf-μ[unfolded wfp-eq-minimal,
rule-format])
    then obtain xm'' where xm'' ∈ Q'' and min': ⋀xm'''. μ xm''' xm'' ⟹ xm''' ∉ Q'' by blast
     from ‹xm'' ∈ Q''› obtain x'' ln'' where xm'' = (x'', m) ts'(t ↦ (x'', ln'')) ∈ Q unfolding
Q''-def by blast
    moreover {
     fix ts''
     assume mμt m ts'' (ts'(t ↦ (x'', ln'')))
     then obtain T X'' LN'' X' LN'
       where finite (dom ts'') ts'' T = ⌊(X'', LN'')⌋
       and (ts'(t ↦ (x'', ln''))) T = ⌊(X', LN')⌋ μ (X'', m) (X', m)
       and eq: ⋀t'. t' ≠ T ⟹ ts'' t' = (ts'(t ↦ (x'', ln''))) t' by(cases) blast
     have ts'' ∉ Q
     proof(cases T = t)
       case True
       from True ‹(ts'(t ↦ (x'', ln''))) T = ⌊(X', LN')⌋› have X' = x'' by simp
        with ‹μ (X'', m) (X', m)› have (X'', m) ∉ Q'' by(auto dest: min'[unfolded ‹xm'' = (x'',
m)›])
       hence ∀ ln. ts'(t ↦ (X'', ln)) ∉ Q by(simp add: Q''-def)
       moreover from ‹ts' t = None› eq True
       have ts''(t := None) = ts' by(auto simp add: fun-eq-iff)
       with ‹ts'' T = ⌊(X'', LN'')⌋› True
       have ts'': ts'' = ts'(t ↦ (X'', LN'')) by(auto intro!: ext)
       ultimately show ?thesis by blast
     next
       case False
       from ‹finite (dom ts'')› have finite (dom (ts''(t := None))) by simp
       moreover from ‹ts'' T = ⌊(X'', LN'')⌋› False
       have (ts''(t := None)) T = ⌊(X'', LN'')⌋ by simp
       moreover from ‹(ts'(t ↦ (x'', ln''))) T = ⌊(X', LN')⌋› False
       have ts' T = ⌊(X', LN')⌋ by simp
       ultimately have mμt m (ts''(t := None)) ts' using ‹μ (X'', m) (X', m)›
       proof(rule mμtI)
         fix t'
         assume t' ≠ T
         with eq[OF False[symmetric]] eq[OF this] ‹ts' t = None›
         show (ts''(t := None)) t' = ts' t' by auto
       qed
       hence ts''(t := None) ∉ Q' by(rule min)
       thus ?thesis
       proof(rule contrapos-nn)
         assume ts'' ∈ Q
         from eq[OF False[symmetric]] have ts'' t = ⌊(x'', ln'')⌋ by simp
         hence ts'': ts''(t ↦ (x'', ln'')) = ts'' by(auto simp add: fun-eq-iff)
         from ‹ts'' ∈ Q› have ts''(t ↦ (x'', ln'')) ∈ Q unfolding ts'' .
         thus ts''(t := None) ∈ Q' unfolding Q'-def by auto
       qed
     qed
    }
    ultimately show ?case by blast
   qed
```

qed
qed

lemma *wf-mμ*: *wfP mμ*
proof −
have *wf* (*inv-image* (*same-fst* (λ*m*. *True*) (λ*m*. {(*ts*, *ts'*). *mμt m ts ts'*})) (λ*s*. (*shr s*, *thr s*)))
by(*rule wf-inv-image*)(*rule wf-same-fst*, *rule wf-mμt*[*unfolded wfp-def*])
also have *inv-image* (*same-fst* (λ*m*. *True*) (λ*m*. {(*ts*, *ts'*). *mμt m ts ts'*})) (λ*s*. (*shr s*, *thr s*)) = {(*s*, *s'*). *mμ s s'*}
by(*auto simp add*: *mμ-def same-fst-def*)
finally show *?thesis* **by**(*simp add*: *wfp-def*)
qed

end

end

1.17 Various notions of bisimulation

theory *Bisimulation*
imports
LTS
begin

type-synonym ('*a*, '*b*) *bisim* = '*a* ⇒ '*b* ⇒ *bool*

1.17.1 Strong bisimulation

locale *bisimulation-base* = *r1*: *trsys trsys1* + *r2*: *trsys trsys2*
for *trsys1* :: ('*s1*, '*tl1*) *trsys* (‹-/ −*1*−-→/ -› [*50*,*0*,*50*] *60*)
and *trsys2* :: ('*s2*, '*tl2*) *trsys* (‹-/ −*2*−-→/ -› [*50*,*0*,*50*] *60*) +
fixes *bisim* :: ('*s1*, '*s2*) *bisim* (‹-/ ≈ -› [*50*, *50*] *60*)
and *tlsim* :: ('*tl1*, '*tl2*) *bisim* (‹-/ ∼ -› [*50*, *50*] *60*)
begin

notation
r1.Trsys (‹-/ −*1*−-→∗/ -› [*50*,*0*,*50*] *60*) **and**
r2.Trsys (‹-/ −*2*−-→∗/ -› [*50*,*0*,*50*] *60*)

notation
r1.inf-step (‹- −*1*−-→∗ ∞› [*50*, *0*] *80*) **and**
r2.inf-step (‹- −*2*−-→∗ ∞› [*50*, *0*] *80*)

notation
r1.inf-step-table (‹- −*1*−-→∗*t* ∞› [*50*, *0*] *80*) **and**
r2.inf-step-table (‹- −*2*−-→∗*t* ∞› [*50*, *0*] *80*)

abbreviation *Tlsim* :: ('*tl1 list*, '*tl2 list*) *bisim* (‹-/ [∼] -› [*50*, *50*] *60*)
where *Tlsim tl1 tl2* ≡ *list-all2 tlsim tl1 tl2*

abbreviation *Tlsiml* :: ('*tl1 llist*, '*tl2 llist*) *bisim* (‹-/ [[∼]] -› [*50*, *50*] *60*)
where *Tlsiml tl1 tl2* ≡ *llist-all2 tlsim tl1 tl2*

end

locale *bisimulation* = *bisimulation-base* +
  constrains *trsys1* :: (*'s1*, *'tl1*) *trsys*
  and *trsys2* :: (*'s2*, *'tl2*) *trsys*
  and *bisim* :: (*'s1*, *'s2*) *bisim*
  and *tlsim* :: (*'tl1*, *'tl2*) *bisim*
  assumes *simulation1*: ⟦ *s1* ≈ *s2*; *s1* −*1*−*tl1*→ *s1'* ⟧ ⟹ ∃ *s2' tl2*. *s2* −*2*−*tl2*→ *s2'* ∧ *s1'* ≈ *s2'* ∧ *tl1* ∼ *tl2*
  and *simulation2*: ⟦ *s1* ≈ *s2*; *s2* −*2*−*tl2*→ *s2'* ⟧ ⟹ ∃ *s1' tl1*. *s1* −*1*−*tl1*→ *s1'* ∧ *s1'* ≈ *s2'* ∧ *tl1* ∼ *tl2*
begin

lemma *bisimulation-flip*:
  bisimulation trsys2 trsys1 (*flip bisim*) (*flip tlsim*)
by(*unfold-locales*)(*unfold flip-simps*,(*blast intro*: *simulation1 simulation2*)+)

end

lemma *bisimulation-flip-simps* [*flip-simps*]:
  bisimulation trsys2 trsys1 (*flip bisim*) (*flip tlsim*) = *bisimulation trsys1 trsys2 bisim tlsim*
by(*auto dest*: *bisimulation.bisimulation-flip simp only*: *flip-flip*)

context *bisimulation* **begin**

lemma *simulation1-rtrancl*:
  ⟦*s1* −*1*−*tls1*→∗ *s1'*; *s1* ≈ *s2*⟧
  ⟹ ∃ *s2' tls2*. *s2* −*2*−*tls2*→∗ *s2'* ∧ *s1'* ≈ *s2'* ∧ *tls1* [∼] *tls2*
proof(*induct rule*: *rtrancl3p.induct*)
  case *rtrancl3p-refl* **thus** *?case* **by**(*auto intro*: *rtrancl3p.rtrancl3p-refl*)
next
  case (*rtrancl3p-step s1 tls1 s1' tl1 s1''*)
  from ‹*s1* ≈ *s2* ⟹ ∃ *s2' tls2*. *s2* −*2*−*tls2*→∗ *s2'* ∧ *s1'* ≈ *s2'* ∧ *tls1* [∼] *tls2*› ‹*s1* ≈ *s2*›
  obtain *s2' tls2* **where** *s2* −*2*−*tls2*→∗ *s2' s1'* ≈ *s2' tls1* [∼] *tls2* **by** *blast*
  moreover from ‹*s1'* −*1*−*tl1*→ *s1''*› ‹*s1'* ≈ *s2'*›
  obtain *s2'' tl2* **where** *s2'* −*2*−*tl2*→ *s2'' s1''* ≈ *s2'' tl1* ∼ *tl2* **by**(*auto dest*: *simulation1*)
  ultimately have *s2* −*2*−*tls2* @ [*tl2*]→∗ *s2''* *tls1* @ [*tl1*] [∼] *tls2* @ [*tl2*]
    by(*auto intro*: *rtrancl3p.rtrancl3p-step list-all2-appendI*)
  with ‹*s1''* ≈ *s2''*› **show** *?case* **by**(*blast*)
qed

lemma *simulation2-rtrancl*:
  ⟦*s2* −*2*−*tls2*→∗ *s2'*; *s1* ≈ *s2*⟧
  ⟹ ∃ *s1' tls1*. *s1* −*1*−*tls1*→∗ *s1'* ∧ *s1'* ≈ *s2'* ∧ *tls1* [∼] *tls2*
using *bisimulation.simulation1-rtrancl*[*OF bisimulation-flip*]
unfolding *flip-simps* **.**

lemma *simulation1-inf-step*:
  assumes *red1*: *s1* −*1*−*tls1*→∗ ∞ **and** *bisim*: *s1* ≈ *s2*
  shows ∃ *tls2*. *s2* −*2*−*tls2*→∗ ∞ ∧ *tls1* [[∼]] *tls2*
proof −
  from *r1.inf-step-imp-inf-step-table*[*OF red1*]
  obtain *stls1* **where** *red1'*: *s1* −*1*−*stls1*→∗*t* ∞
    and *tls1*: *tls1* = *lmap* (*fst* ∘ *snd*) *stls1* **by** *blast*
  define *tl1-to-tl2* **where** *tl1-to-tl2 s2 stls1* = *unfold-llist*

```
  (λ(s2, stls1). lnull stls1)
  (λ(s2, stls1). let (s1, tl1, s1') = lhd stls1;
                     (tl2, s2') = SOME (tl2, s2'). trsys2 s2 tl2 s2' ∧ s1' ≈ s2' ∧ tl1 ∼ tl2
                 in (s2, tl2, s2'))
  (λ(s2, stls1). let (s1, tl1, s1') = lhd stls1;
                     (tl2, s2') = SOME (tl2, s2'). trsys2 s2 tl2 s2' ∧ s1' ≈ s2' ∧ tl1 ∼ tl2
                 in (s2', ltl stls1))
  (s2, stls1)
 for s2 :: 's2 and stls1 :: ('s1 × 'tl1 × 's1) llist

have tl1-to-tl2-simps [simp]:
 ⋀s2 stls1. lnull (tl1-to-tl2 s2 stls1) ⟷ lnull stls1
 ⋀s2 stls1. ¬ lnull stls1 ⟹ lhd (tl1-to-tl2 s2 stls1) =
 (let (s1, tl1, s1') = lhd stls1;
      (tl2, s2') = SOME (tl2, s2'). trsys2 s2 tl2 s2' ∧ s1' ≈ s2' ∧ tl1 ∼ tl2
  in (s2, tl2, s2'))
 ⋀s2 stls1. ¬ lnull stls1 ⟹ ltl (tl1-to-tl2 s2 stls1) =
 (let (s1, tl1, s1') = lhd stls1;
      (tl2, s2') = SOME (tl2, s2'). trsys2 s2 tl2 s2' ∧ s1' ≈ s2' ∧ tl1 ∼ tl2
  in tl1-to-tl2 s2' (ltl stls1))
 ⋀s2. tl1-to-tl2 s2 LNil = LNil
 ⋀s2 s1 tl1 s1' stls1'. tl1-to-tl2 s2 (LCons (s1, tl1, s1') stls1') =
    LCons (s2, SOME (tl2, s2'). trsys2 s2 tl2 s2' ∧ s1' ≈ s2' ∧ tl1 ∼ tl2)
          (tl1-to-tl2 (snd (SOME (tl2, s2'). trsys2 s2 tl2 s2' ∧ s1' ≈ s2' ∧ tl1 ∼ tl2)) stls1')
 by(simp-all add: tl1-to-tl2-def split-beta)

have [simp]: llength (tl1-to-tl2 s2 stls1) = llength stls1
 by(coinduction arbitrary: s2 stls1 rule: enat-coinduct)(auto simp add: epred-llength split-beta)

from red1' bisim have s2 −2−tl1-to-tl2 s2 stls1→∗t ∞
proof(coinduction arbitrary: s2 s1 stls1)
 case (inf-step-table s2 s1 stls1)
 note red1' = ‹s1 −1−stls1→∗t ∞› and bisim = ‹s1 ≈ s2›
 from red1' show ?case
 proof(cases)
  case (inf-step-tableI s1' stls1' tl1)
  hence stls1: stls1 = LCons (s1, tl1, s1') stls1'
   and r: s1 −1−tl1→ s1' and reds1: s1' −1−stls1'→∗t ∞ by simp-all
  let ?tl2s2' = SOME (tl2, s2'). s2 −2−tl2→ s2' ∧ s1' ≈ s2' ∧ tl1 ∼ tl2
  let ?tl2 = fst ?tl2s2' let ?s2' = snd ?tl2s2'
  from simulation1[OF bisim r] obtain s2' tl2
   where s2 −2−tl2→ s2' s1' ≈ s2' tl1 ∼ tl2 by blast
  hence (λ(tl2, s2'). s2 −2−tl2→ s2' ∧ s1' ≈ s2' ∧ tl1 ∼ tl2) (tl2, s2') by simp
  hence (λ(tl2, s2'). s2 −2−tl2→ s2' ∧ s1' ≈ s2' ∧ tl1 ∼ tl2) ?tl2s2' by(rule someI)
  hence s2 −2−?tl2→ ?s2' s1' ≈ ?s2' tl1 ∼ ?tl2 by(simp-all add: split-beta)
  then show ?thesis using reds1 stls1 by(fastforce intro: prod-eqI)
 qed
qed
hence s2 −2−lmap (fst ∘ snd) (tl1-to-tl2 s2 stls1)→∗ ∞
 by(rule r2.inf-step-table-imp-inf-step)
moreover have tls1 [[∼]] lmap (fst ∘ snd) (tl1-to-tl2 s2 stls1)
proof(rule llist-all2-all-lnthI)
 show llength tls1 = llength (lmap (fst ∘ snd) (tl1-to-tl2 s2 stls1))
  using tls1 by simp
```

```
  next
    fix n
    assume enat n < llength tls1
    thus lnth tls1 n ∼ lnth (lmap (fst ∘ snd) (tl1-to-tl2 s2 stls1)) n
      using red1' bisim unfolding tls1
    proof(induct n arbitrary: s1 s2 stls1 rule: nat-less-induct)
      case (1 n)
      hence IH: ⋀m s1 s2 stls1. ⟦ m < n; enat m < llength (lmap (fst ∘ snd) stls1);
                                   s1 −1−stls1→∗t ∞; s1 ≈ s2 ⟧
                ⟹ lnth (lmap (fst ∘ snd) stls1) m ∼ lnth (lmap (fst ∘ snd) (tl1-to-tl2 s2 stls1)) m
        by blast
      from ‹s1 −1−stls1→∗t ∞› show ?case
      proof cases
        case (inf-step-tableI s1' stls1' tl1)
        hence  stls1: stls1 = LCons (s1, tl1, s1') stls1'
          and r: s1 −1−tl1→ s1' and reds: s1' −1−stls1'→∗t ∞ by simp-all
        let ?tl2s2' = SOME (tl2, s2'). s2 −2−tl2→ s2' ∧ s1' ≈ s2' ∧ tl1 ∼ tl2
        let ?tl2 = fst ?tl2s2' let ?s2' = snd ?tl2s2'
        from simulation1[OF ‹s1 ≈ s2› r] obtain s2' tl2
          where s2 −2−tl2→ s2' ∧ s1' ≈ s2' ∧ tl1 ∼ tl2 by blast
        hence (λ(tl2, s2'). s2 −2−tl2→ s2' ∧ s1' ≈ s2' ∧ tl1 ∼ tl2) (tl2, s2') by simp
        hence (λ(tl2, s2'). s2 −2−tl2→ s2' ∧ s1' ≈ s2' ∧ tl1 ∼ tl2) ?tl2s2' by(rule someI)
        hence bisim': s1' ≈ ?s2' and tlsim: tl1 ∼ ?tl2 by(simp-all add: split-beta)
        show ?thesis
        proof(cases n)
          case 0
          with stls1 tlsim show ?thesis by simp
        next
          case (Suc m)
          hence m < n by simp
          moreover have enat m < llength (lmap (fst ∘ snd) stls1')
            using stls1 ‹enat n < llength (lmap (fst ∘ snd) stls1)› Suc by(simp add: Suc-ile-eq)
          ultimately have lnth (lmap (fst ∘ snd) stls1') m ∼ lnth (lmap (fst ∘ snd) (tl1-to-tl2 ?s2'
stls1')) m
            using reds bisim' by(rule IH)
          with Suc stls1 show ?thesis by(simp del: o-apply)
        qed
      qed
    qed
  qed
  ultimately show ?thesis by blast
qed

lemma simulation2-inf-step:
  ⟦ s2 −2−tls2→∗ ∞; s1 ≈ s2 ⟧ ⟹ ∃ tls1. s1 −1−tls1→∗ ∞ ∧ tls1 [[∼]] tls2
using bisimulation.simulation1-inf-step[OF bisimulation-flip]
unfolding flip-simps .

end

locale bisimulation-final-base =
  bisimulation-base +
  constrains trsys1 :: ('s1, 'tl1) trsys
  and trsys2 :: ('s2, 'tl2) trsys
```

and *bisim* :: (*'s1*, *'s2*) *bisim*
and *tlsim* :: (*'tl1*, *'tl2*) *bisim*
fixes *final1* :: *'s1* ⇒ *bool*
and *final2* :: *'s2* ⇒ *bool*

locale *bisimulation-final* = *bisimulation-final-base* + *bisimulation* +
constrains *trsys1* :: (*'s1*, *'tl1*) *trsys*
and *trsys2* :: (*'s2*, *'tl2*) *trsys*
and *bisim* :: (*'s1*, *'s2*) *bisim*
and *tlsim* :: (*'tl1*, *'tl2*) *bisim*
and *final1* :: *'s1* ⇒ *bool*
and *final2* :: *'s2* ⇒ *bool*
assumes *bisim-final*: *s1* ≈ *s2* ⟹ *final1 s1* ⟷ *final2 s2*

begin

lemma *bisimulation-final-flip*:
bisimulation-final trsys2 trsys1 (*flip bisim*) (*flip tlsim*) *final2 final1*
apply(*intro-locales*)
apply(*rule bisimulation-flip*)
apply(*unfold-locales*)
by(*unfold flip-simps*, *rule bisim-final*[*symmetric*])

end

lemma *bisimulation-final-flip-simps* [*flip-simps*]:
bisimulation-final trsys2 trsys1 (*flip bisim*) (*flip tlsim*) *final2 final1* =
bisimulation-final trsys1 trsys2 bisim tlsim final1 final2
by(*auto dest*: *bisimulation-final.bisimulation-final-flip simp only*: *flip-flip*)

context *bisimulation-final* **begin**

lemma *final-simulation1*:
⟦ *s1* ≈ *s2*; *s1* −*1*−*tls1*→∗ *s1'*; *final1 s1'* ⟧
⟹ ∃ *s2' tls2*. *s2* −*2*−*tls2*→∗ *s2'* ∧ *s1'* ≈ *s2'* ∧ *final2 s2'* ∧ *tls1* [∼] *tls2*
by(*auto dest*: *bisim-final dest*!: *simulation1-rtrancl*)

lemma *final-simulation2*:
⟦ *s1* ≈ *s2*; *s2* −*2*−*tls2*→∗ *s2'*; *final2 s2'* ⟧
⟹ ∃ *s1' tls1*. *s1* −*1*−*tls1*→∗ *s1'* ∧ *s1'* ≈ *s2'* ∧ *final1 s1'* ∧ *tls1* [∼] *tls2*
by(*auto dest*: *bisim-final dest*!: *simulation2-rtrancl*)

end

1.17.2 Delay bisimulation

locale *delay-bisimulation-base* =
bisimulation-base +
trsys1?: *τtrsys trsys1 τmove1* +
trsys2?: *τtrsys trsys2 τmove2*
for *τmove1 τmove2* +
constrains *trsys1* :: (*'s1*, *'tl1*) *trsys*
and *trsys2* :: (*'s2*, *'tl2*) *trsys*
and *bisim* :: (*'s1*, *'s2*) *bisim*

and *tlsim* :: (*'tl1*, *'tl2*) *bisim*
and *τmove1* :: (*'s1*, *'tl1*) *trsys*
and *τmove2* :: (*'s2*, *'tl2*) *trsys*
begin

notation
trsys1.silent-move (‹-/ $-\tau 1\rightarrow$ -› [*50*, *50*] *60*) **and**
trsys2.silent-move (‹-/ $-\tau 2\rightarrow$ -› [*50*, *50*] *60*)

notation
trsys1.silent-moves (‹-/ $-\tau 1\rightarrow*$ -› [*50*, *50*] *60*) **and**
trsys2.silent-moves (‹-/ $-\tau 2\rightarrow*$ -› [*50*, *50*] *60*)

notation
trsys1.silent-movet (‹-/ $-\tau 1\rightarrow+$ -› [*50*, *50*] *60*) **and**
trsys2.silent-movet (‹-/ $-\tau 2\rightarrow+$ -› [*50*, *50*] *60*)

notation
trsys1.τrtrancl3p (‹- $-\tau 1-\!\!\rightarrow*$ -› [*50*, *0*, *50*] *60*) **and**
trsys2.τrtrancl3p (‹- $-\tau 2-\!\!\rightarrow*$ -› [*50*, *0*, *50*] *60*)

notation
trsys1.τinf-step (‹- $-\tau 1-\!\!\rightarrow* \infty$› [*50*, *0*] *80*) **and**
trsys2.τinf-step (‹- $-\tau 2-\!\!\rightarrow* \infty$› [*50*, *0*] *80*)

notation
trsys1.τdiverge (‹- $-\tau 1\rightarrow \infty$› [*50*] *80*) **and**
trsys2.τdiverge (‹- $-\tau 2\rightarrow \infty$› [*50*] *80*)

notation
trsys1.τinf-step-table (‹- $-\tau 1-\!\!\rightarrow* t \infty$› [*50*, *0*] *80*) **and**
trsys2.τinf-step-table (‹- $-\tau 2-\!\!\rightarrow* t \infty$› [*50*, *0*] *80*)

notation
trsys1.τRuns (‹- $\Downarrow 1$ -› [*50*, *50*] *51*) **and**
trsys2.τRuns (‹- $\Downarrow 2$ -› [*50*, *50*] *51*)

lemma *simulation-silent1I'*:
assumes $\exists\, s2'.$ (*if* $\mu 1\ s1'\ s1$ *then trsys2.silent-moves else trsys2.silent-movet*) *s2 s2'* $\wedge$ $s1' \approx s2'$
shows $s1' \approx s2 \wedge \mu 1 \hat{}{+}{+}\ s1'\ s1 \vee (\exists\, s2'.\ s2\ -\tau 2\rightarrow+\ s2' \wedge s1' \approx s2')$
proof −
from *assms* **obtain** *s2'* **where** *red*: (*if* $\mu 1\ s1'\ s1$ *then trsys2.silent-moves else trsys2.silent-movet*) *s2 s2'*
and *bisim*: $s1' \approx s2'$ **by** *blast*
show *?thesis*
proof(*cases* $\mu 1\ s1'\ s1$)
case *True*
with *red* **have** $s2\ -\tau 2\rightarrow*\ s2'$ **by** *simp*
thus *?thesis* **using** *bisim True* **by** *cases*(*blast intro*: *rtranclp-into-tranclp1*)+
next
case *False*
with *red bisim* **show** *?thesis* **by** *auto*
qed
qed

lemma *simulation-silent2I'*:
assumes $\exists s1'.$ (*if* $\mu 2\ s2'\ s2$ *then trsys1.silent-moves else trsys1.silent-movet*) $s1\ s1' \wedge s1' \approx s2'$
shows $s1 \approx s2' \wedge \mu 2\hat{}{+}{+}\ s2'\ s2 \vee (\exists s1'.\ s1\ -\tau 1\rightarrow{+}\ s1' \wedge s1' \approx s2')$
using *assms*
by(*rule delay-bisimulation-base.simulation-silent1I'*)

end

locale *delay-bisimulation-obs* = *delay-bisimulation-base* - - - - $\tau move1\ \tau move2$
for $\tau move1 :: 's1 \Rightarrow 'tl1 \Rightarrow 's1 \Rightarrow bool$
and $\tau move2 :: 's2 \Rightarrow 'tl2 \Rightarrow 's2 \Rightarrow bool$ +
assumes *simulation1*:
$[\![\ s1 \approx s2;\ s1\ -1-tl1\rightarrow\ s1';\ \neg\ \tau move1\ s1\ tl1\ s1'\ ]\!]$
$\Longrightarrow \exists s2'\ s2''\ tl2.\ s2\ -\tau 2\rightarrow{*}\ s2' \wedge s2'\ -2-tl2\rightarrow\ s2'' \wedge \neg\ \tau move2\ s2'\ tl2\ s2'' \wedge s1' \approx s2'' \wedge tl1 \sim tl2$
and *simulation2*:
$[\![\ s1 \approx s2;\ s2\ -2-tl2\rightarrow\ s2';\ \neg\ \tau move2\ s2\ tl2\ s2'\ ]\!]$
$\Longrightarrow \exists s1'\ s1''\ tl1.\ s1\ -\tau 1\rightarrow{*}\ s1' \wedge s1'\ -1-tl1\rightarrow\ s1'' \wedge \neg\ \tau move1\ s1'\ tl1\ s1'' \wedge s1'' \approx s2' \wedge tl1 \sim tl2$
begin

lemma *delay-bisimulation-obs-flip*: *delay-bisimulation-obs trsys2 trsys1* (*flip bisim*) (*flip tlsim*) $\tau move2\ \tau move1$
apply(*unfold-locales*)
apply(*unfold flip-simps*)
by(*blast intro*: *simulation1 simulation2*)+

end

lemma *delay-bisimulation-obs-flip-simps* [*flip-simps*]:
delay-bisimulation-obs trsys2 trsys1 (*flip bisim*) (*flip tlsim*) $\tau move2\ \tau move1$ =
delay-bisimulation-obs trsys1 trsys2 bisim tlsim $\tau move1\ \tau move2$
by(*auto dest*: *delay-bisimulation-obs.delay-bisimulation-obs-flip simp only*: *flip-flip*)

locale *delay-bisimulation-diverge* = *delay-bisimulation-obs* - - - - $\tau move1\ \tau move2$
for $\tau move1 :: 's1 \Rightarrow 'tl1 \Rightarrow 's1 \Rightarrow bool$
and $\tau move2 :: 's2 \Rightarrow 'tl2 \Rightarrow 's2 \Rightarrow bool$ +
assumes *simulation-silent1*:
$[\![\ s1 \approx s2;\ s1\ -\tau 1\rightarrow\ s1'\ ]\!] \Longrightarrow \exists s2'.\ s2\ -\tau 2\rightarrow{*}\ s2' \wedge s1' \approx s2'$
and *simulation-silent2*:
$[\![\ s1 \approx s2;\ s2\ -\tau 2\rightarrow\ s2'\ ]\!] \Longrightarrow \exists s1'.\ s1\ -\tau 1\rightarrow{*}\ s1' \wedge s1' \approx s2'$
and τ*diverge-bisim-inv*: $s1 \approx s2 \Longrightarrow s1\ -\tau 1\rightarrow \infty \longleftrightarrow s2\ -\tau 2\rightarrow \infty$
begin

lemma *delay-bisimulation-diverge-flip*: *delay-bisimulation-diverge trsys2 trsys1* (*flip bisim*) (*flip tlsim*) $\tau move2\ \tau move1$
apply(*rule delay-bisimulation-diverge.intro*)
apply(*rule delay-bisimulation-obs-flip*)
apply(*unfold-locales*)
apply(*unfold flip-simps*)
by(*blast intro*: *simulation-silent1 simulation-silent2* τ*diverge-bisim-inv*[*symmetric*] *del*: *iffI*)+

end

lemma *delay-bisimulation-diverge-flip-simps* [*flip-simps*]:
delay-bisimulation-diverge trsys2 trsys1 (*flip bisim*) (*flip tlsim*) *τmove2 τmove1* =
delay-bisimulation-diverge trsys1 trsys2 bisim tlsim τmove1 τmove2
by(*auto dest*: *delay-bisimulation-diverge.delay-bisimulation-diverge-flip simp only*: *flip-flip*)

context *delay-bisimulation-diverge* **begin**

lemma *simulation-silents1*:
assumes *bisim*: *s1* ≈ *s2* **and** *moves*: *s1* −*τ1*→∗ *s1′*
shows ∃ *s2′*. *s2* −*τ2*→∗ *s2′* ∧ *s1′* ≈ *s2′*
using *moves bisim*
proof *induct*
case *base* **thus** *?case* **by**(*blast*)
next
case (*step s1′ s1″*)
from ‹*s1* ≈ *s2* ⟹ ∃ *s2′*. *s2* −*τ2*→∗ *s2′* ∧ *s1′* ≈ *s2′*› ‹*s1* ≈ *s2*›
obtain *s2′* **where** *s2* −*τ2*→∗ *s2′ s1′* ≈ *s2′* **by** *blast*
from *simulation-silent1*[*OF* ‹*s1′* ≈ *s2′*› ‹*s1′* −*τ1*→ *s1″*›]
obtain *s2″* **where** *s2′* −*τ2*→∗ *s2″ s1″* ≈ *s2″* **by** *blast*
from ‹*s2* −*τ2*→∗ *s2′*› ‹*s2′* −*τ2*→∗ *s2″*› **have** *s2* −*τ2*→∗ *s2″* **by**(*rule rtranclp-trans*)
with ‹*s1″* ≈ *s2″*› **show** *?case* **by** *blast*
qed

lemma *simulation-silents2*:
⟦ *s1* ≈ *s2*; *s2* −*τ2*→∗ *s2′* ⟧ ⟹ ∃ *s1′*. *s1* −*τ1*→∗ *s1′* ∧ *s1′* ≈ *s2′*
using *delay-bisimulation-diverge.simulation-silents1*[*OF delay-bisimulation-diverge-flip*]
unfolding *flip-simps* **.**

lemma *simulation1-τrtrancl3p*:
⟦ *s1* −*τ1*−*tls1*→∗ *s1′*; *s1* ≈ *s2* ⟧
⟹ ∃ *tls2 s2′*. *s2* −*τ2*−*tls2*→∗ *s2′* ∧ *s1′* ≈ *s2′* ∧ *tls1* [∼] *tls2*
proof(*induct arbitrary*: *s2 rule*: *trsys1.τrtrancl3p.induct*)
case (*τrtrancl3p-refl s*)
thus *?case* **by**(*auto intro*: *τtrsys.τrtrancl3p.intros*)
next
case (*τrtrancl3p-step s1 s1′ tls1 s1″ tl1*)
from *simulation1*[*OF* ‹*s1* ≈ *s2*› ‹*s1* −*1*−*tl1*→ *s1′*› ‹¬ *τmove1 s1 tl1 s1′*›]
obtain *s2′ s2″ tl2* **where** *τred*: *s2* −*τ2*→∗ *s2′*
and *red*: *s2′* −*2*−*tl2*→ *s2″* **and** *nτ*: ¬ *τmove2 s2′ tl2 s2″*
and *bisim′*: *s1′* ≈ *s2″* **and** *tlsim*: *tl1* ∼ *tl2* **by** *blast*
from *bisim′* ‹*s1′* ≈ *s2″* ⟹ ∃ *tls2 s2′*. *s2″* −*τ2*−*tls2*→∗ *s2′* ∧ *s1″* ≈ *s2′* ∧ *tls1* [∼] *tls2*›
obtain *tls2 s2‴* **where** *IH*: *s2″* −*τ2*−*tls2*→∗ *s2‴ s1″* ≈ *s2‴ tls1* [∼] *tls2* **by** *blast*
from *τred* **have** *s2* −*τ2*−[]→∗ *s2′* **by**(*rule trsys2.silent-moves-into-τrtrancl3p*)
also from *red nτ IH*(*1*) **have** *s2′* −*τ2*−*tl2* # *tls2*→∗ *s2‴* **by**(*rule τrtrancl3p.τrtrancl3p-step*)
finally show *?case* **using** *IH tlsim* **by** *fastforce*
next
case (*τrtrancl3p-τstep s1 s1′ tls1 s1″ tl1*)
from ‹*s1* −*1*−*tl1*→ *s1′*› ‹*τmove1 s1 tl1 s1′*› **have** *s1* −*τ1*→ *s1′* **..**
from *simulation-silent1*[*OF* ‹*s1* ≈ *s2*› *this*]
obtain *s2′* **where** *τred*: *s2* −*τ2*→∗ *s2′* **and** *bisim′*: *s1′* ≈ *s2′* **by** *blast*
from *τred* **have** *s2* −*τ2*−[]→∗ *s2′* **by**(*rule trsys2.silent-moves-into-τrtrancl3p*)
also from *bisim′* ‹*s1′* ≈ *s2′* ⟹ ∃ *tls2 s2″*. *s2′* −*τ2*−*tls2*→∗ *s2″* ∧ *s1″* ≈ *s2″* ∧ *tls1* [∼] *tls2*›

obtain *tls2 s2''* **where** *IH*: $s2' -\tau 2-tls2\twoheadrightarrow* s2''\ s1'' \approx s2''\ tls1\ [\sim]\ tls2$ **by** *blast*
note $\langle s2' -\tau 2-tls2\twoheadrightarrow* s2''\rangle$
finally show *?case* **using** *IH* **by** *auto*
qed

lemma *simulation2-τrtrancl3p*:
$[\![\ s2\ -\tau 2-tls2\twoheadrightarrow* s2';\ s1 \approx s2\ ]\!]$
$\Longrightarrow \exists\, tls1\ s1'.\ s1\ -\tau 1-tls1\twoheadrightarrow* s1' \wedge s1' \approx s2' \wedge tls1\ [\sim]\ tls2$
using *delay-bisimulation-diverge.simulation1-τrtrancl3p*[*OF delay-bisimulation-diverge-flip*]
unfolding *flip-simps* **.**

lemma *simulation1-τinf-step*:
assumes *τinf1*: $s1\ -\tau 1-tls1\twoheadrightarrow* \infty$ **and** *bisim*: $s1 \approx s2$
shows $\exists\, tls2.\ s2\ -\tau 2-tls2\twoheadrightarrow* \infty \wedge tls1\ [[\sim]]\ tls2$
proof −
from *trsys1.τinf-step-imp-τinf-step-table*[*OF τinf1*]
obtain *sstls1* **where** *τinf1'*: $s1\ -\tau 1-sstls1\twoheadrightarrow* t\ \infty$
and *tls1*: $tls1 = lmap\ (fst \circ snd \circ snd)\ sstls1$ **by** *blast*
define *tl1-to-tl2* **where** $tl1\text{-}to\text{-}tl2\ s2\ sstls1 = unfold\text{-}llist$
$(\lambda(s2,\ sstls1).\ lnull\ sstls1)$
$(\lambda(s2,\ sstls1).$
let $(s1,\ s1',\ tl1,\ s1'') = lhd\ sstls1$;
$(s2',\ tl2,\ s2'') = SOME\ (s2',\ tl2,\ s2'').\ s2\ -\tau 2\twoheadrightarrow* s2' \wedge trsys2\ s2'\ tl2\ s2'' \wedge$
$\neg\ \tau move2\ s2'\ tl2\ s2'' \wedge\ s1'' \approx s2'' \wedge tl1 \sim tl2$
in $(s2,\ s2',\ tl2,\ s2''))$
$(\lambda(s2,\ sstls1).$
let $(s1,\ s1',\ tl1,\ s1'') = lhd\ sstls1$;
$(s2',\ tl2,\ s2'') = SOME\ (s2',\ tl2,\ s2'').\ s2\ -\tau 2\twoheadrightarrow* s2' \wedge trsys2\ s2'\ tl2\ s2'' \wedge$
$\neg\ \tau move2\ s2'\ tl2\ s2'' \wedge\ s1'' \approx s2'' \wedge tl1 \sim tl2$
in $(s2'',\ ltl\ sstls1))$
$(s2,\ sstls1)$
for *s2* :: *'s2* **and** *sstls1* :: $('s1 \times 's1 \times 'tl1 \times 's1)\ llist$
have [*simp*]:
$\bigwedge s2\ sstls1.\ lnull\ (tl1\text{-}to\text{-}tl2\ s2\ sstls1) \longleftrightarrow lnull\ sstls1$
$\bigwedge s2\ sstls1.\ \neg\ lnull\ sstls1 \Longrightarrow lhd\ (tl1\text{-}to\text{-}tl2\ s2\ sstls1) =$
$(let\ (s1,\ s1',\ tl1,\ s1'') = lhd\ sstls1;$
$(s2',\ tl2,\ s2'') = SOME\ (s2',\ tl2,\ s2'').\ s2\ -\tau 2\twoheadrightarrow* s2' \wedge trsys2\ s2'\ tl2\ s2'' \wedge$
$\neg\ \tau move2\ s2'\ tl2\ s2'' \wedge\ s1'' \approx s2'' \wedge tl1 \sim tl2$
$in\ (s2,\ s2',\ tl2,\ s2''))$
$\bigwedge s2\ sstls1.\ \neg\ lnull\ sstls1 \Longrightarrow ltl\ (tl1\text{-}to\text{-}tl2\ s2\ sstls1) =$
$(let\ (s1,\ s1',\ tl1,\ s1'') = lhd\ sstls1;$
$(s2',\ tl2,\ s2'') = SOME\ (s2',\ tl2,\ s2'').\ s2\ -\tau 2\twoheadrightarrow* s2' \wedge trsys2\ s2'\ tl2\ s2'' \wedge$
$\neg\ \tau move2\ s2'\ tl2\ s2'' \wedge\ s1'' \approx s2'' \wedge tl1 \sim tl2$
$in\ tl1\text{-}to\text{-}tl2\ s2''\ (ltl\ sstls1))$
$\bigwedge s2.\ tl1\text{-}to\text{-}tl2\ s2\ LNil = LNil$
$\bigwedge s2\ s1\ s1'\ tl1\ s1''\ stls1'.\ tl1\text{-}to\text{-}tl2\ s2\ (LCons\ (s1,\ s1',\ tl1,\ s1')\ stls1') =$
$LCons\ (s2,\ SOME\ (s2',\ tl2,\ s2'').\ s2\ -\tau 2\twoheadrightarrow* s2' \wedge trsys2\ s2'\ tl2\ s2'' \wedge$
$\neg\ \tau move2\ s2'\ tl2\ s2'' \wedge s1'' \approx s2'' \wedge tl1 \sim tl2)$
$(tl1\text{-}to\text{-}tl2\ (snd\ (snd\ (SOME\ (s2',\ tl2,\ s2'').\ s2\ -\tau 2\twoheadrightarrow* s2' \wedge trsys2\ s2'\ tl2\ s2'' \wedge$
$\neg\ \tau move2\ s2'\ tl2\ s2'' \wedge s1'' \approx s2'' \wedge tl1 \sim tl2)))$
$stls1')$
by(*simp-all add*: *tl1-to-tl2-def split-beta*)

have [*simp*]: $llength\ (tl1\text{-}to\text{-}tl2\ s2\ sstls1) = llength\ sstls1$

by(*coinduction arbitrary*: *s2 sstls1 rule*: *enat-coinduct*)(*auto simp add*: *epred-llength split-beta*)

define *sstls2* **where** *sstls2* = *tl1-to-tl2 s2 sstls1*
with *τinf1′ bisim* **have** ∃ *s1 sstls1*. *s1* −*τ1*−*sstls1*→∗*t* ∞ ∧ *sstls2* = *tl1-to-tl2 s2 sstls1* ∧ *s1* ≈ *s2*
by *blast*

from *τinf1′ bisim* **have** *s2* −*τ2*−*tl1-to-tl2 s2 sstls1*→∗*t* ∞
proof(*coinduction arbitrary*: *s2 s1 sstls1*)
case (*τinf-step-table s2 s1 sstls1*)
note *τinf′* = ‹*s1* −*τ1*−*sstls1*→∗*t* ∞› **and** *bisim* = ‹*s1* ≈ *s2*›
from *τinf′* **show** *?case*
proof(*cases*)
case (*τinf-step-table-Cons s1′ s1″ sstls1′ tl1*)
hence *sstls1*: *sstls1* = *LCons* (*s1*, *s1′*, *tl1*, *s1″*) *sstls1′*
and *τs*: *s1* −*τ1*→∗ *s1′* **and** *r*: *s1′* −1−*tl1*→ *s1″* **and** *nτ*: ¬ *τmove1 s1′ tl1 s1″*
and *reds1*: *s1″* −*τ1*−*sstls1′*→∗*t* ∞ **by** *simp-all*
let *?P* = λ(*s2′*, *tl2*, *s2″*). *s2* −*τ2*→∗ *s2′* ∧ *trsys2 s2′ tl2 s2″* ∧ ¬ *τmove2 s2′ tl2 s2″* ∧ *s1″* ≈ *s2″* ∧ *tl1* ∼ *tl2*
let *?s2tl2s2′* = *Eps ?P*
let *?s2″* = *snd* (*snd ?s2tl2s2′*)
from *simulation-silents1*[*OF* ‹*s1* ≈ *s2*› *τs*]
obtain *s2′* **where** *s2* −*τ2*→∗ *s2′ s1′* ≈ *s2′* **by** *blast*
from *simulation1*[*OF* ‹*s1′* ≈ *s2′*› *r nτ*] **obtain** *s2″ s2‴ tl2*
where *s2′* −*τ2*→∗ *s2″*
and *rest*: *s2″* −2−*tl2*→ *s2‴* ¬ *τmove2 s2″ tl2 s2‴ s1″* ≈ *s2‴ tl1* ∼ *tl2* **by** *blast*
from ‹*s2* −*τ2*→∗ *s2′*› ‹*s2′* −*τ2*→∗ *s2″*› **have** *s2* −*τ2*→∗ *s2″* **by**(*rule rtranclp-trans*)
with *rest* **have** *?P* (*s2″*, *tl2*, *s2‴*) **by** *simp*
hence *?P ?s2tl2s2′* **by**(*rule someI*)
then show *?thesis* **using** *reds1 sstls1* **by** *fastforce*
next
case *τinf-step-table-Nil*
hence [*simp*]: *sstls1* = *LNil* **and** *s1* −*τ1*→ ∞ **by** *simp-all*
from ‹*s1* −*τ1*→ ∞› ‹*s1* ≈ *s2*› **have** *s2* −*τ2*→ ∞ **by**(*simp add*: *τdiverge-bisim-inv*)
thus *?thesis* **using** *sstls2-def* **by** *simp*
qed
qed
hence *s2* −*τ2*−*lmap* (*fst* ∘ *snd* ∘ *snd*) (*tl1-to-tl2 s2 sstls1*)→∗ ∞
by(*rule trsys2.τinf-step-table-into-τinf-step*)
moreover have *tls1* [[∼]] *lmap* (*fst* ∘ *snd* ∘ *snd*) (*tl1-to-tl2 s2 sstls1*)
proof(*rule llist-all2-all-lnthI*)
show *llength tls1* = *llength* (*lmap* (*fst* ∘ *snd* ∘ *snd*) (*tl1-to-tl2 s2 sstls1*))
using *tls1* **by** *simp*
next
fix *n*
assume *enat n* < *llength tls1*
thus *lnth tls1 n* ∼ *lnth* (*lmap* (*fst* ∘ *snd* ∘ *snd*) (*tl1-to-tl2 s2 sstls1*)) *n*
using *τinf1′ bisim* **unfolding** *tls1*
proof(*induct n arbitrary*: *s1 s2 sstls1 rule*: *less-induct*)
case (*less n*)
note *IH* = ‹⋀*m s1 s2 sstls1*. ⟦ *m* < *n*; *enat m* < *llength* (*lmap* (*fst* ∘ *snd* ∘ *snd*) *sstls1*);
s1 −*τ1*−*sstls1*→∗*t* ∞; *s1* ≈ *s2* ⟧
⟹ *lnth* (*lmap* (*fst* ∘ *snd* ∘ *snd*) *sstls1*) *m* ∼ *lnth* (*lmap* (*fst* ∘ *snd* ∘ *snd*) (*tl1-to-tl2 s2 sstls1*)) *m*›
from ‹*s1* −*τ1*−*sstls1*→∗*t* ∞› **show** *?case*

```
      proof cases
        case (τinf-step-table-Cons s1' s1'' sstls1' tl1)
        hence sstls1: sstls1 = LCons (s1, s1', tl1, s1'') sstls1'
          and τs: s1 −τ1→* s1' and r: s1' −1−tl1→ s1''
          and nτ: ¬ τmove1 s1' tl1 s1'' and reds: s1'' −τ1−sstls1'→*t ∞ by simp-all
        let ?P = λ(s2', tl2, s2''). s2 −τ2→* s2' ∧ trsys2 s2' tl2 s2'' ∧ ¬ τmove2 s2' tl2 s2'' ∧ s1''
≈ s2'' ∧ tl1 ∼ tl2
        let ?s2tl2s2' = Eps ?P let ?tl2 = fst (snd ?s2tl2s2') let ?s2'' = snd (snd ?s2tl2s2')
        from simulation-silents1[OF ‹s1 ≈ s2› τs] obtain s2'
          where s2 −τ2→* s2' s1' ≈ s2' by blast
        from simulation1[OF ‹s1' ≈ s2'› r nτ] obtain s2'' s2''' tl2
          where s2' −τ2→* s2''
          and rest: s2'' −2−tl2→ s2''' ¬ τmove2 s2'' tl2 s2''' s1'' ≈ s2''' tl1 ∼ tl2 by blast
        from ‹s2 −τ2→* s2'› ‹s2' −τ2→* s2''› have s2 −τ2→* s2'' by(rule rtranclp-trans)
        with rest have ?P (s2'', tl2, s2''') by auto
        hence ?P ?s2tl2s2' by(rule someI)
        hence s1'' ≈ ?s2'' tl1 ∼ ?tl2 by(simp-all add: split-beta)
        show ?thesis
        proof(cases n)
          case 0
          with sstls1 ‹tl1 ∼ ?tl2› show ?thesis by simp
        next
          case (Suc m)
          hence m < n by simp
          moreover have enat m < llength (lmap (fst ∘ snd ∘ snd) sstls1')
            using sstls1 ‹enat n < llength (lmap (fst ∘ snd ∘ snd) sstls1)› Suc by(simp add: Suc-ile-eq)
            ultimately have lnth (lmap (fst ∘ snd ∘ snd) sstls1') m ∼ lnth (lmap (fst ∘ snd ∘ snd)
(tl1-to-tl2 ?s2'' sstls1')) m
            using reds ‹s1'' ≈ ?s2''› by(rule IH)
          with Suc sstls1 show ?thesis by(simp del: o-apply)
        qed
      next
        case τinf-step-table-Nil
        with ‹enat n < llength (lmap (fst ∘ snd ∘ snd) sstls1)› have False by simp
        thus ?thesis ..
      qed
    qed
  qed
  ultimately show ?thesis by blast
qed

lemma simulation2-τinf-step:
  ⟦ s2 −τ2−tls2→* ∞; s1 ≈ s2 ⟧ ⟹ ∃ tls1. s1 −τ1−tls1→* ∞ ∧ tls1 [[∼]] tls2
using delay-bisimulation-diverge.simulation1-τinf-step[OF delay-bisimulation-diverge-flip]
unfolding flip-simps .

lemma no-τmove1-τs-to-no-τmove2:
  assumes s1 ≈ s2
  and no-τmoves1: ⋀s1'. ¬ s1 −τ1→ s1'
  shows ∃ s2'. s2 −τ2→* s2' ∧ (∀ s2''. ¬ s2' −τ2→ s2'') ∧ s1 ≈ s2'
proof −
  have ¬ s1 −τ1→ ∞
  proof
    assume s1 −τ1→ ∞
```

then obtain *s1'* **where** *s1* $-\tau 1\rightarrow$ *s1'* **by** *cases*
with *no-τmoves1*[*of s1'*] **show** *False* **by** *contradiction*
qed
with ‹*s1* $\approx$ *s2*› **have** $\neg$ *s2* $-\tau 2\rightarrow \infty$ **by**(*simp add*: *τdiverge-bisim-inv*)
from *trsys2.not-τdiverge-to-no-τmove*[*OF this*]
obtain *s2'* **where** *s2* $-\tau 2\rightarrow *$ *s2'* **and** $\bigwedge$*s2''*. $\neg$ *s2'* $-\tau 2\rightarrow$ *s2''* **by** *blast*
moreover from *simulation-silents2*[*OF* ‹*s1* $\approx$ *s2*› ‹*s2* $-\tau 2\rightarrow *$ *s2'*›]
obtain *s1'* **where** *s1* $-\tau 1\rightarrow *$ *s1'* **and** *s1'* $\approx$ *s2'* **by** *blast*
from ‹*s1* $-\tau 1\rightarrow *$ *s1'*› *no-τmoves1* **have** *s1'* = *s1*
by(*auto elim*: *converse-rtranclpE*)
ultimately show *?thesis* **using** ‹*s1'* $\approx$ *s2'*› **by** *blast*
qed

lemma *no-τmove2-τs-to-no-τmove1*:
⟦ *s1* $\approx$ *s2*; $\bigwedge$*s2'*. $\neg$ *s2* $-\tau 2\rightarrow$ *s2'* ⟧ $\Longrightarrow$ $\exists$ *s1'*. *s1* $-\tau 1\rightarrow *$ *s1'* $\wedge$ ($\forall$ *s1''*. $\neg$ *s1'* $-\tau 1\rightarrow$ *s1''*) $\wedge$ *s1'* $\approx$ *s2*
using *delay-bisimulation-diverge.no-τmove1-τs-to-no-τmove2*[*OF delay-bisimulation-diverge-flip*]
unfolding *flip-simps* **.**

lemma *no-move1-to-no-move2*:
assumes *s1* $\approx$ *s2*
and *no-moves1*: $\bigwedge$*tl1 s1'*. $\neg$ *s1* $-1-$*tl1*$\rightarrow$ *s1'*
shows $\exists$ *s2'*. *s2* $-\tau 2\rightarrow *$ *s2'* $\wedge$ ($\forall$ *tl2 s2''*. $\neg$ *s2'* $-2-$*tl2*$\rightarrow$ *s2''*) $\wedge$ *s1* $\approx$ *s2'*
proof −
from *no-moves1* **have** $\bigwedge$*s1'*. $\neg$ *s1* $-\tau 1\rightarrow$ *s1'* **by**(*auto*)
from *no-τmove1-τs-to-no-τmove2*[*OF* ‹*s1* $\approx$ *s2*› *this*]
obtain *s2'* **where** *s2* $-\tau 2\rightarrow *$ *s2'* **and** *s1* $\approx$ *s2'*
and *no-τmoves2*: $\bigwedge$*s2''*. $\neg$ *s2'* $-\tau 2\rightarrow$ *s2''* **by** *blast*
moreover
have $\bigwedge$*tl2 s2''*. $\neg$ *s2'* $-2-$*tl2*$\rightarrow$ *s2''*
proof
fix *tl2 s2''*
assume *s2'* $-2-$*tl2*$\rightarrow$ *s2''*
with *no-τmoves2*[*of s2''*] **have** $\neg$ *τmove2 s2' tl2 s2''* **by**(*auto*)
from *simulation2*[*OF* ‹*s1* $\approx$ *s2'*› ‹*s2'* $-2-$*tl2*$\rightarrow$ *s2''*› *this*]
obtain *s1' s1'' tl1* **where** *s1* $-\tau 1\rightarrow *$ *s1'* **and** *s1'* $-1-$*tl1*$\rightarrow$ *s1''* **by** *blast*
with *no-moves1* **show** *False* **by**(*fastforce elim*: *converse-rtranclpE*)
qed
ultimately show *?thesis* **by** *blast*
qed

lemma *no-move2-to-no-move1*:
⟦ *s1* $\approx$ *s2*; $\bigwedge$*tl2 s2'*. $\neg$ *s2* $-2-$*tl2*$\rightarrow$ *s2'* ⟧
$\Longrightarrow$ $\exists$ *s1'*. *s1* $-\tau 1\rightarrow *$ *s1'* $\wedge$ ($\forall$ *tl1 s1''*. $\neg$ *s1'* $-1-$*tl1*$\rightarrow$ *s1''*) $\wedge$ *s1'* $\approx$ *s2*
using *delay-bisimulation-diverge.no-move1-to-no-move2*[*OF delay-bisimulation-diverge-flip*]
unfolding *flip-simps* **.**

lemma *simulation-τRuns-table1*:
assumes *bisim*: *s1* $\approx$ *s2*
and *run1*: *trsys1.τRuns-table s1 stlsss1*
shows $\exists$ *stlsss2*. *trsys2.τRuns-table s2 stlsss2* $\wedge$ *tllist-all2* (λ(*tl1*, *s1''*) (*tl2*, *s2''*). *tl1* $\sim$ *tl2* $\wedge$ *s1''* $\approx$ *s2''*) (*rel-option bisim*) *stlsss1 stlsss2*
proof(*intro exI conjI*)
let *?P* = λ(*s2* :: *'s2*) (*stlsss1* :: (*'tl1* × *'s1*, *'s1 option*) *tllist*) (*tl2*, *s2''*).

$\exists s2'.\ s2\ -\tau 2\rightarrow *\ s2' \wedge s2' -2-tl2\rightarrow s2'' \wedge \neg\ \tau move2\ s2'\ tl2\ s2'' \wedge snd\ (thd\ stlsss1) \approx s2'' \wedge fst\ (thd\ stlsss1) \sim tl2$

define *tls1-to-tls2* **where** *tls1-to-tls2 s2 stlsss1 = unfold-tllist*
(λ(*s2*, *stlsss1*). *is-TNil stlsss1*)
(λ(*s2*, *stlsss1*). *map-option* (λ*s1'*. *SOME s2'*. *s2* $-\tau 2\rightarrow *$ *s2'* ∧ (∀ *tl s2''*. ¬ *s2'* $-2-tl\rightarrow$ *s2''*) ∧ *s1'* ≈ *s2'*) (*terminal stlsss1*))
(λ(*s2*, *stlsss1*). *let* (*tl2*, *s2''*) = *Eps* (*?P s2 stlsss1*) *in* (*tl2*, *s2''*))
(λ(*s2*, *stlsss1*). *let* (*tl2*, *s2''*) = *Eps* (*?P s2 stlsss1*) *in* (*s2''*, *ttl stlsss1*))
(*s2*, *stlsss1*)
for *s2 stlsss1*
have [*simp*]:
⋀*s2 stlsss1*. *is-TNil* (*tls1-to-tls2 s2 stlsss1*) ⟷ *is-TNil stlsss1*
⋀*s2 stlsss1*. *is-TNil stlsss1* ⟹ *terminal* (*tls1-to-tls2 s2 stlsss1*) = *map-option* (λ*s1'*. *SOME s2'*. *s2* $-\tau 2\rightarrow *$ *s2'* ∧ (∀ *tl s2''*. ¬ *s2'* $-2-tl\rightarrow$ *s2''*) ∧ *s1'* ≈ *s2'*) (*terminal stlsss1*)
⋀*s2 stlsss1*. ¬ *is-TNil stlsss1* ⟹ *thd* (*tls1-to-tls2 s2 stlsss1*) = (*let* (*tl2*, *s2''*) = *Eps* (*?P s2 stlsss1*) *in* (*tl2*, *s2''*))
⋀*s2 stlsss1*. ¬ *is-TNil stlsss1* ⟹ *ttl* (*tls1-to-tls2 s2 stlsss1*) = (*let* (*tl2*, *s2''*) = *Eps* (*?P s2 stlsss1*) *in tls1-to-tls2 s2''* (*ttl stlsss1*))
⋀*s2 os1*. *tls1-to-tls2 s2* (*TNil os1*) =
TNil (*map-option* (λ*s1'*. *SOME s2'*. *s2* $-\tau 2\rightarrow *$ *s2'* ∧ (∀ *tl s2''*. ¬ *s2'* $-2-tl\rightarrow$ *s2''*) ∧ *s1'* ≈ *s2'*) *os1*)
by(*simp-all add*: *tls1-to-tls2-def split-beta*)
have [*simp*]:
⋀*s2 s1 s1' tl1 s1'' stlsss1*.
tls1-to-tls2 s2 (*TCons* (*tl1*, *s1''*) *stlsss1*) =
(*let* (*tl2*, *s2''*) = *SOME* (*tl2*, *s2''*). ∃ *s2'*. *s2* $-\tau 2\rightarrow *$ *s2'* ∧ *s2'* $-2-tl2\rightarrow$ *s2''* ∧
¬ *τmove2 s2' tl2 s2''* ∧ *s1''* ≈ *s2''* ∧ *tl1* ∼ *tl2*
in TCons (*tl2*, *s2''*) (*tls1-to-tls2 s2''* *stlsss1*))
by(*rule tllist.expand*)(*simp-all add*: *split-beta*)

from *bisim run1*
show *trsys2.τRuns-table s2* (*tls1-to-tls2 s2 stlsss1*)
proof(*coinduction arbitrary*: *s2 s1 stlsss1*)
case (*τRuns-table s2 s1 stlsss1*)
note *bisim* = ‹*s1* ≈ *s2*›
and *run1* = ‹*trsys1.τRuns-table s1 stlsss1*›
from *run1* **show** *?case*
proof *cases*
case (*Terminate s1'*)
let *?P* = λ*s2'*. *s2* $-\tau 2\rightarrow *$ *s2'* ∧ (∀ *tl2 s2''*. ¬ *s2'* $-2-tl2\rightarrow$ *s2''*) ∧ *s1'* ≈ *s2'*
from *simulation-silents1*[*OF bisim* ‹*s1* $-\tau 1\rightarrow *$ *s1'*›]
obtain *s2'* **where** *s2* $-\tau 2\rightarrow *$ *s2'* **and** *s1'* ≈ *s2'* **by** *blast*
moreover from *no-move1-to-no-move2*[*OF* ‹*s1'* ≈ *s2'*› ‹⋀*tl1 s1''*. ¬ *s1'* $-1-tl1\rightarrow$ *s1''*›]
obtain *s2''* **where** *s2'* $-\tau 2\rightarrow *$ *s2''* **and** *s1'* ≈ *s2''*
and ⋀*tl2 s2'''*. ¬ *s2''* $-2-tl2\rightarrow$ *s2'''* **by** *blast*
ultimately have *?P s2''* **by**(*blast intro*: *rtranclp-trans*)
hence *?P* (*Eps ?P*) **by**(*rule someI*)
hence *?Terminate* **using** ‹*stlsss1* = *TNil* ⌊*s1'*⌋› **by** *simp*
thus *?thesis* **..**
next
case *Diverge*
with *τdiverge-bisim-inv*[*OF bisim*]
have *?Diverge* **by** *simp*
thus *?thesis* **by** *simp*

```
    next
      case (Proceed s1' s1'' stlsss1' tl1)
      let ?P = λ(tl2, s2''). ∃ s2'. s2 −τ2→* s2' ∧ s2' −2−tl2→ s2'' ∧ ¬ τmove2 s2' tl2 s2'' ∧ s1''
≈ s2'' ∧ tl1 ∼ tl2
      from simulation-silents1[OF bisim ‹s1 −τ1→* s1'›]
      obtain s2' where s2 −τ2→* s2' and s1' ≈ s2' by blast
      moreover from simulation1[OF ‹s1' ≈ s2'› ‹s1' −1−tl1→ s1''› ‹¬ τmove1 s1' tl1 s1''›]
      obtain s2'' s2''' tl2 where s2' −τ2→* s2''
        and s2'' −2−tl2→ s2''' and ¬ τmove2 s2'' tl2 s2'''
        and s1'' ≈ s2''' and tl1 ∼ tl2 by blast
      ultimately have ?P (tl2, s2''') by(blast intro: rtranclp-trans)
      hence ?P (Eps ?P) by(rule someI)
      hence ?Proceed
        using ‹stlsss1 = TCons (tl1, s1'') stlsss1'› ‹trsys1.τRuns-table s1'' stlsss1'›
        by auto blast
      thus ?thesis by simp
    qed
  qed

  let ?Tlsim = λ(tl1, s1'') (tl2, s2''). tl1 ∼ tl2 ∧ s1'' ≈ s2''
  let ?Bisim = rel-option bisim
  from run1 bisim
  show tllist-all2 ?Tlsim ?Bisim stlsss1 (tls1-to-tls2 s2 stlsss1)
  proof(coinduction arbitrary: s1 s2 stlsss1)
    case (tllist-all2 s1 s2 stlsss1)
    note Runs = ‹trsys1.τRuns-table s1 stlsss1› and bisim = ‹s1 ≈ s2›
    from Runs show ?case
    proof cases
      case (Terminate s1')
      let ?P = λs2'. s2 −τ2→* s2' ∧ (∀ tl2 s2''. ¬ s2' −2−tl2→ s2'') ∧ s1' ≈ s2'
      from simulation-silents1[OF bisim ‹s1 −τ1→* s1'›]
      obtain s2' where s2 −τ2→* s2' and s1' ≈ s2' by blast
      moreover
      from no-move1-to-no-move2[OF ‹s1' ≈ s2'› ‹⋀tl1 s1''. ¬ s1' −1−tl1→ s1''›]
      obtain s2'' where s2' −τ2→* s2'' and s1' ≈ s2''
        and ⋀tl2 s2'''. ¬ s2'' −2−tl2→ s2''' by blast
      ultimately have ?P s2'' by(blast intro: rtranclp-trans)
      hence ?P (Eps ?P) by(rule someI)
      thus ?thesis using ‹stlsss1 = TNil ⌊s1'⌋› bisim by(simp)
    next
      case (Proceed s1' s1'' stlsss1' tl1)
      from simulation-silents1[OF bisim ‹s1 −τ1→* s1'›]
      obtain s2' where s2 −τ2→* s2' and s1' ≈ s2' by blast
      moreover from simulation1[OF ‹s1' ≈ s2'› ‹s1' −1−tl1→ s1''› ‹¬ τmove1 s1' tl1 s1''›]
      obtain s2'' s2''' tl2 where s2' −τ2→* s2''
        and s2'' −2−tl2→ s2''' and ¬ τmove2 s2'' tl2 s2'''
        and s1'' ≈ s2''' and tl1 ∼ tl2 by blast
      ultimately have ?P s2 stlsss1 (tl2, s2''')
        using ‹stlsss1 = TCons (tl1, s1'') stlsss1'› by(auto intro: rtranclp-trans)
      hence ?P s2 stlsss1 (Eps (?P s2 stlsss1)) by(rule someI)
      thus ?thesis using ‹stlsss1 = TCons (tl1, s1'') stlsss1'› ‹trsys1.τRuns-table s1'' stlsss1'› bisim
        by auto blast
    qed simp
  qed
```

qed

lemma *simulation-τRuns-table2*:
assumes *s1* ≈ *s2*
and *trsys2.τRuns-table s2 stlsss2*
shows ∃ *stlsss1*. *trsys1.τRuns-table s1 stlsss1* ∧ *tllist-all2* (λ(*tl1*, *s1''*) (*tl2*, *s2''*). *tl1* ∼ *tl2* ∧ *s1''* ≈ *s2''*) (*rel-option bisim*) *stlsss1 stlsss2*
using *delay-bisimulation-diverge.simulation-τRuns-table1*[*OF delay-bisimulation-diverge-flip*, *unfolded flip-simps*, *OF assms*]
by(*subst tllist-all2-flip*[*symmetric*])(*simp only*: *flip-def split-def*)

lemma *simulation-τRuns1*:
assumes *bisim*: *s1* ≈ *s2*
and *run1*: *s1* ⇓*1 tls1*
shows ∃ *tls2*. *s2* ⇓*2 tls2* ∧ *tllist-all2 tlsim* (*rel-option bisim*) *tls1 tls2*
proof −
from *trsys1.τRuns-into-τRuns-table*[*OF run1*]
obtain *stlsss1* **where** *tls1*: *tls1* = *tmap fst id stlsss1*
and *τRuns1*: *trsys1.τRuns-table s1 stlsss1* **by** *blast*
from *simulation-τRuns-table1*[*OF bisim τRuns1*]
obtain *stlsss2* **where** *τRuns2*: *trsys2.τRuns-table s2 stlsss2*
and *tlsim*: *tllist-all2* (λ(*tl1*, *s1''*) (*tl2*, *s2''*). *tl1* ∼ *tl2* ∧ *s1''* ≈ *s2''*)
(*rel-option bisim*) *stlsss1 stlsss2* **by** *blast*
from *τRuns2* **have** *s2* ⇓*2 tmap fst id stlsss2*
by(*rule τRuns-table-into-τRuns*)
moreover have *tllist-all2 tlsim* (*rel-option bisim*) *tls1* (*tmap fst id stlsss2*)
using *tlsim* **unfolding** *tls1*
by(*fastforce simp add*: *tllist-all2-tmap1 tllist-all2-tmap2 elim*: *tllist-all2-mono rel-option-mono*)
ultimately show *?thesis* **by** *blast*
qed

lemma *simulation-τRuns2*:
⟦ *s1* ≈ *s2*; *s2* ⇓*2 tls2* ⟧
⟹ ∃ *tls1*. *s1* ⇓*1 tls1* ∧ *tllist-all2 tlsim* (*rel-option bisim*) *tls1 tls2*
using *delay-bisimulation-diverge.simulation-τRuns1*[*OF delay-bisimulation-diverge-flip*]
unfolding *flip-simps* **.**

end

locale *delay-bisimulation-final-base* =
delay-bisimulation-base - - - - *τmove1 τmove2* +
bisimulation-final-base - - - - *final1 final2*
for *τmove1* :: (*'s1*, *'tl1*) *trsys*
and *τmove2* :: (*'s2*, *'tl2*) *trsys*
and *final1* :: *'s1* ⇒ *bool*
and *final2* :: *'s2* ⇒ *bool* +
assumes *final1-simulation*: ⟦ *s1* ≈ *s2*; *final1 s1* ⟧ ⟹ ∃ *s2'*. *s2* −*τ2*→* *s2'* ∧ *s1* ≈ *s2'* ∧ *final2 s2'*
and *final2-simulation*: ⟦ *s1* ≈ *s2*; *final2 s2* ⟧ ⟹ ∃ *s1'*. *s1* −*τ1*→* *s1'* ∧ *s1'* ≈ *s2* ∧ *final1 s1'*
begin

lemma *delay-bisimulation-final-base-flip*:
delay-bisimulation-final-base trsys2 trsys1 (*flip bisim*) *τmove2 τmove1 final2 final1*
apply(*unfold-locales*)
apply(*unfold flip-simps*)

by(*blast intro*: *final1-simulation final2-simulation*)+

end

lemma *delay-bisimulation-final-base-flip-simps* [*flip-simps*]:
delay-bisimulation-final-base trsys2 trsys1 (*flip bisim*) *τmove2 τmove1 final2 final1* =
delay-bisimulation-final-base trsys1 trsys2 bisim τmove1 τmove2 final1 final2
by(*auto dest*: *delay-bisimulation-final-base.delay-bisimulation-final-base-flip simp only*: *flip-flip*)

context *delay-bisimulation-final-base* **begin**

lemma *τRuns-terminate-final1*:
assumes *s1* ⇓*1 tls1*
and *s2* ⇓*2 tls2*
and *tllist-all2 tlsim* (*rel-option bisim*) *tls1 tls2*
and *tfinite tls1*
and *terminal tls1* = *Some s1′*
and *final1 s1′*
shows ∃ *s2′*. *tfinite tls2* ∧ *terminal tls2* = *Some s2′* ∧ *final2 s2′*
using *assms*(*4*) *assms*(*1−3*,*5−*)
apply(*induct arbitrary*: *tls2 s1 s2 rule*: *tfinite-induct*)
apply(*auto 4 4 simp add*: *tllist-all2-TCons1 tllist-all2-TNil1 option-rel-Some1 trsys1.τRuns-simps*
trsys2.τRuns-simps dest: *final1-simulation elim*: *converse-rtranclpE*)
done

lemma *τRuns-terminate-final2*:
⟦ *s1* ⇓*1 tls1*; *s2* ⇓*2 tls2*; *tllist-all2 tlsim* (*rel-option bisim*) *tls1 tls2*;
tfinite tls2; *terminal tls2* = *Some s2′*; *final2 s2′* ⟧
⟹ ∃ *s1′*. *tfinite tls1* ∧ *terminal tls1* = *Some s1′* ∧ *final1 s1′*
using *delay-bisimulation-final-base.τRuns-terminate-final1* [**where** *tlsim* = *flip tlsim*, *OF delay-bisimulation-final-b*
unfolding *flip-simps* **by** −

end

locale *delay-bisimulation-diverge-final* =
delay-bisimulation-diverge +
delay-bisimulation-final-base +
constrains *trsys1* :: (*'s1*, *'tl1*) *trsys*
and *trsys2* :: (*'s2*, *'tl2*) *trsys*
and *bisim* :: (*'s1*, *'s2*) *bisim*
and *tlsim* :: (*'tl1*, *'tl2*) *bisim*
and *τmove1* :: (*'s1*, *'tl1*) *trsys*
and *τmove2* :: (*'s2*, *'tl2*) *trsys*
and *final1* :: *'s1* ⇒ *bool*
and *final2* :: *'s2* ⇒ *bool*
begin

lemma *delay-bisimulation-diverge-final-flip*:
delay-bisimulation-diverge-final trsys2 trsys1 (*flip bisim*) (*flip tlsim*) *τmove2 τmove1 final2 final1*
apply(*rule delay-bisimulation-diverge-final.intro*)
apply(*rule delay-bisimulation-diverge-flip*)
apply(*unfold-locales*, *unfold flip-simps*)
apply(*blast intro*: *final1-simulation final2-simulation*)+
done

end

lemma *delay-bisimulation-diverge-final-flip-simps* [*flip-simps*]:
delay-bisimulation-diverge-final trsys2 trsys1 (*flip bisim*) (*flip tlsim*) *τmove2 τmove1 final2 final1* =
delay-bisimulation-diverge-final trsys1 trsys2 bisim tlsim τmove1 τmove2 final1 final2
by(*auto dest*: *delay-bisimulation-diverge-final.delay-bisimulation-diverge-final-flip simp only*: *flip-flip*)

context *delay-bisimulation-diverge-final* **begin**

lemma *delay-bisimulation-diverge*:
delay-bisimulation-diverge trsys1 trsys2 bisim tlsim τmove1 τmove2
by(*unfold-locales*)

lemma *delay-bisimulation-final-base*:
delay-bisimulation-final-base trsys1 trsys2 bisim τmove1 τmove2 final1 final2
by(*unfold-locales*)

lemma *final-simulation1*:
⟦ *s1* ≈ *s2*; *s1* −*τ1*−*tls1*→∗ *s1'*; *final1 s1'* ⟧
⟹ ∃ *s2' tls2*. *s2* −*τ2*−*tls2*→∗ *s2'* ∧ *s1'* ≈ *s2'* ∧ *final2 s2'* ∧ *tls1* [∼] *tls2*
by(*blast dest*: *simulation1-τrtrancl3p final1-simulation intro*: *τrtrancl3p-trans*[*OF - silent-moves-into-τrtrancl3p*, *simplified*])

lemma *final-simulation2*:
⟦ *s1* ≈ *s2*; *s2* −*τ2*−*tls2*→∗ *s2'*; *final2 s2'* ⟧
⟹ ∃ *s1' tls1*. *s1* −*τ1*−*tls1*→∗ *s1'* ∧ *s1'* ≈ *s2'* ∧ *final1 s1'* ∧ *tls1* [∼] *tls2*
by(*rule delay-bisimulation-diverge-final.final-simulation1*[*OF delay-bisimulation-diverge-final-flip*, *unfolded flip-simps*])

end

locale *delay-bisimulation-measure-base* =
delay-bisimulation-base +
constrains *trsys1* :: *'s1* ⇒ *'tl1* ⇒ *'s1* ⇒ *bool*
and *trsys2* :: *'s2* ⇒ *'tl2* ⇒ *'s2* ⇒ *bool*
and *bisim* :: *'s1* ⇒ *'s2* ⇒ *bool*
and *tlsim* :: *'tl1* ⇒ *'tl2* ⇒ *bool*
and *τmove1* :: *'s1* ⇒ *'tl1* ⇒ *'s1* ⇒ *bool*
and *τmove2* :: *'s2* ⇒ *'tl2* ⇒ *'s2* ⇒ *bool*
fixes *μ1* :: *'s1* ⇒ *'s1* ⇒ *bool*
and *μ2* :: *'s2* ⇒ *'s2* ⇒ *bool*

locale *delay-bisimulation-measure* =
delay-bisimulation-measure-base - - - - τmove1 τmove2 μ1 μ2 +
delay-bisimulation-obs trsys1 trsys2 bisim tlsim τmove1 τmove2
for *τmove1* :: *'s1* ⇒ *'tl1* ⇒ *'s1* ⇒ *bool*
and *τmove2* :: *'s2* ⇒ *'tl2* ⇒ *'s2* ⇒ *bool*
and *μ1* :: *'s1* ⇒ *'s1* ⇒ *bool*
and *μ2* :: *'s2* ⇒ *'s2* ⇒ *bool* +
assumes *simulation-silent1*:
⟦ *s1* ≈ *s2*; *s1* −*τ1*→ *s1'* ⟧ ⟹ *s1'* ≈ *s2* ∧ *μ1*^++ *s1' s1* ∨ (∃ *s2'*. *s2* −*τ2*→+ *s2'* ∧ *s1'* ≈ *s2'*)
and *simulation-silent2*:
⟦ *s1* ≈ *s2*; *s2* −*τ2*→ *s2'* ⟧ ⟹ *s1* ≈ *s2'* ∧ *μ2*^++ *s2' s2* ∨ (∃ *s1'*. *s1* −*τ1*→+ *s1'* ∧ *s1'* ≈ *s2'*)

and *wf-μ1*: *wfP μ1*
and *wf-μ2*: *wfP μ2*
begin

lemma *delay-bisimulation-measure-flip*:
delay-bisimulation-measure trsys2 trsys1 (*flip bisim*) (*flip tlsim*) *τmove2 τmove1 μ2 μ1*
apply(*rule delay-bisimulation-measure.intro*)
apply(*rule delay-bisimulation-obs-flip*)
apply(*unfold-locales*)
apply(*unfold flip-simps*)
apply(*rule simulation-silent1 simulation-silent2 wf-μ1 wf-μ2*|*assumption*)+
done

end

lemma *delay-bisimulation-measure-flip-simps* [*flip-simps*]:
delay-bisimulation-measure trsys2 trsys1 (*flip bisim*) (*flip tlsim*) *τmove2 τmove1 μ2 μ1* =
delay-bisimulation-measure trsys1 trsys2 bisim tlsim τmove1 τmove2 μ1 μ2
by(*auto dest*: *delay-bisimulation-measure.delay-bisimulation-measure-flip simp only*: *flip-simps*)

context *delay-bisimulation-measure* **begin**

lemma *simulation-silentst1*:
assumes *bisim*: *s1* ≈ *s2* **and** *moves*: *s1* $-\tau 1\rightarrow+$ *s1'*
shows *s1'* ≈ *s2* ∧ *μ1*^++ *s1' s1* ∨ (∃ *s2'*. *s2* $-\tau 2\rightarrow+$ *s2'* ∧ *s1'* ≈ *s2'*)
using *moves bisim*
proof *induct*
case (*base s1'*) **thus** *?case* **by**(*auto dest*: *simulation-silent1*)
next
case (*step s1' s1''*)
hence *s1'* ≈ *s2* ∧ *μ1*^++ *s1' s1* ∨ (∃ *s2'*. *s2* $-\tau 2\rightarrow+$ *s2'* ∧ *s1'* ≈ *s2'*) **by** *blast*
thus *?case*
proof
assume *s1'* ≈ *s2* ∧ *μ1*^++ *s1' s1*
hence *s1'* ≈ *s2 μ1*^++ *s1' s1* **by** *simp-all*
with *simulation-silent1*[*OF* ‹*s1'* ≈ *s2*› ‹*s1'* $-\tau 1\rightarrow$ *s1''*›]
show *?thesis* **by**(*auto*)
next
assume ∃ *s2'*. *trsys2.silent-move*^++ *s2 s2'* ∧ *s1'* ≈ *s2'*
then obtain *s2'* **where** *s2* $-\tau 2\rightarrow+$ *s2' s1'* ≈ *s2'* **by** *blast*
with *simulation-silent1*[*OF* ‹*s1'* ≈ *s2'*› ‹*s1'* $-\tau 1\rightarrow$ *s1''*›]
show *?thesis* **by**(*auto intro*: *tranclp-trans*)
qed
qed

lemma *simulation-silentst2*:
⟦ *s1* ≈ *s2*; *s2* $-\tau 2\rightarrow+$ *s2'* ⟧ ⟹ *s1* ≈ *s2'* ∧ *μ2*^++ *s2' s2* ∨ (∃ *s1'*. *s1* $-\tau 1\rightarrow+$ *s1'* ∧ *s1'* ≈ *s2'*)
using *delay-bisimulation-measure.simulation-silentst1*[*OF delay-bisimulation-measure-flip*]
unfolding *flip-simps* **.**

lemma *τdiverge-simulation1*:
assumes *diverge1*: *s1* $-\tau 1\rightarrow$ ∞
and *bisim*: *s1* ≈ *s2*
shows *s2* $-\tau 2\rightarrow$ ∞

proof −
from *assms* **have** *s1* $-\tau 1\rightarrow \infty \wedge$ *s1* $\approx$ *s2* **by** *blast*
thus *?thesis* **using** *wfp-tranclp*[*OF wf-*$\mu 1$]
proof(*coinduct rule*: *trsys2.*τ*diverge-trancl-measure-coinduct*)
case (τ*diverge s2 s1*)
hence *s1* $-\tau 1\rightarrow \infty$ *s1* $\approx$ *s2* **by** *simp-all*
then obtain *s1'* **where** *trsys1.silent-move s1 s1' s1'* $-\tau 1\rightarrow \infty$
by(*fastforce elim*: *trsys1.*τ*diverge.cases*)
from *simulation-silent1*[*OF* ‹*s1* $\approx$ *s2*› ‹*trsys1.silent-move s1 s1'*›] ‹*s1'* $-\tau 1\rightarrow \infty$›
show *?case* **by** *auto*
qed
qed

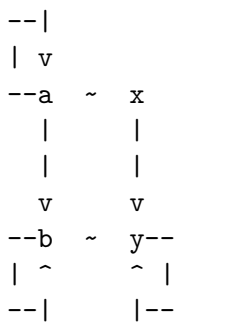
lemma τ*diverge-simulation2*:
⟦ *s2* $-\tau 2\rightarrow \infty$; *s1* $\approx$ *s2* ⟧ $\Longrightarrow$ *s1* $-\tau 1\rightarrow \infty$
using *delay-bisimulation-measure.*τ*diverge-simulation1*[*OF delay-bisimulation-measure-flip*]
unfolding *flip-simps* **.**

lemma τ*diverge-bisim-inv*:
s1 $\approx$ *s2* $\Longrightarrow$ *s1* $-\tau 1\rightarrow \infty \longleftrightarrow$ *s2* $-\tau 2\rightarrow \infty$
by(*blast intro*: τ*diverge-simulation1* τ*diverge-simulation2*)

end

sublocale *delay-bisimulation-measure* < *delay-bisimulation-diverge*
proof
fix *s1 s2 s1'*
assume *s1* $\approx$ *s2 s1* $-\tau 1\rightarrow$ *s1'*
from *simulation-silent1*[*OF this*]
show $\exists$ *s2'*. *s2* $-\tau 2\rightarrow *$ *s2'* $\wedge$ *s1'* $\approx$ *s2'* **by**(*auto intro*: *tranclp-into-rtranclp*)
next
fix *s1 s2 s2'*
assume *s1* $\approx$ *s2 s2* $-\tau 2\rightarrow$ *s2'*
from *simulation-silent2*[*OF this*]
show $\exists$ *s1'*. *s1* $-\tau 1\rightarrow *$ *s1'* $\wedge$ *s1'* $\approx$ *s2'* **by**(*auto intro*: *tranclp-into-rtranclp*)
next
fix *s1 s2*
assume *s1* $\approx$ *s2*
thus *s1* $-\tau 1\rightarrow \infty \longleftrightarrow$ *s2* $-\tau 2\rightarrow \infty$ **by**(*rule* τ*diverge-bisim-inv*)
qed

Counter example for *delay-bisimulation-diverge trsys1 trsys2 bisim tlsim* τ*move1* τ*move2* $\Longrightarrow \exists\, \mu 1\ \mu 2$. *delay-bisimulation-measure trsys1 trsys2 bisim tlsim* τ*move1* τ*move2* $\mu 1\ \mu 2$
(only τmoves):



locale *delay-bisimulation-measure-final* =
  delay-bisimulation-measure +
  delay-bisimulation-final-base +
  constrains *trsys1* :: (*'s1*, *'tl1*) *trsys*
  and *trsys2* :: (*'s2*, *'tl2*) *trsys*
  and *bisim* :: (*'s1*, *'s2*) *bisim*
  and *tlsim* :: (*'tl1*, *'tl2*) *bisim*
  and τ*move1* :: (*'s1*, *'tl1*) *trsys*
  and τ*move2* :: (*'s2*, *'tl2*) *trsys*
  and μ*1* :: *'s1* $\Rightarrow$ *'s1* $\Rightarrow$ *bool*
  and μ*2* :: *'s2* $\Rightarrow$ *'s2* $\Rightarrow$ *bool*
  and *final1* :: *'s1* $\Rightarrow$ *bool*
  and *final2* :: *'s2* $\Rightarrow$ *bool*

sublocale *delay-bisimulation-measure-final* $<$ *delay-bisimulation-diverge-final*
by *unfold-locales*

locale τ*inv* = *delay-bisimulation-base* +
  constrains *trsys1* :: (*'s1*, *'tl1*) *trsys*
  and *trsys2* :: (*'s2*, *'tl2*) *trsys*
  and *bisim* :: (*'s1*, *'s2*) *bisim*
  and *tlsim* :: (*'tl1*, *'tl2*) *bisim*
  and τ*move1* :: (*'s1*, *'tl1*) *trsys*
  and τ*move2* :: (*'s2*, *'tl2*) *trsys*
  and τ*moves1* :: *'s1* $\Rightarrow$ *'s1* $\Rightarrow$ *bool*
  and τ*moves2* :: *'s2* $\Rightarrow$ *'s2* $\Rightarrow$ *bool*
  assumes τ*inv*: $[\![$ *s1* $\approx$ *s2*; *s1* $-1-tl1\rightarrow$ *s1'*; *s2* $-2-tl2\rightarrow$ *s2'*; *s1'* $\approx$ *s2'*; *tl1* $\sim$ *tl2* $]\!]$
    $\Longrightarrow$ τ*move1 s1 tl1 s1'* $\longleftrightarrow$ τ*move2 s2 tl2 s2'*
begin

lemma τ*inv-flip*:
  τ*inv trsys2 trsys1* (*flip bisim*) (*flip tlsim*) τ*move2* τ*move1*
by(*unfold-locales*)(*unfold flip-simps*,*rule* τ*inv*[*symmetric*])

end

lemma τ*inv-flip-simps* [*flip-simps*]:
  τ*inv trsys2 trsys1* (*flip bisim*) (*flip tlsim*) τ*move2* τ*move1* = τ*inv trsys1 trsys2 bisim tlsim* τ*move1* τ*move2*
by(*auto dest*: τ*inv.*τ*inv-flip simp only*: *flip-simps*)

locale *bisimulation-into-delay* =
  bisimulation + τ*inv* +
  constrains *trsys1* :: (*'s1*, *'tl1*) *trsys*
  and *trsys2* :: (*'s2*, *'tl2*) *trsys*
  and *bisim* :: (*'s1*, *'s2*) *bisim*
  and *tlsim* :: (*'tl1*, *'tl2*) *bisim*
  and τ*move1* :: (*'s1*, *'tl1*) *trsys*
  and τ*move2* :: (*'s2*, *'tl2*) *trsys*
begin

lemma *bisimulation-into-delay-flip*:
  bisimulation-into-delay trsys2 trsys1 (*flip bisim*) (*flip tlsim*) τ*move2* τ*move1*
by(*intro-locales*)(*intro bisimulation-flip* τ*inv-flip*)+

end

lemma *bisimulation-into-delay-flip-simps* [*flip-simps*]:
bisimulation-into-delay trsys2 trsys1 (*flip bisim*) (*flip tlsim*) *τmove2 τmove1* =
bisimulation-into-delay trsys1 trsys2 bisim tlsim τmove1 τmove2
by(*auto dest*: *bisimulation-into-delay.bisimulation-into-delay-flip simp only*: *flip-simps*)

context *bisimulation-into-delay* **begin**

lemma *simulation-silent1-aux*:
assumes *bisim*: $s1 \approx s2$ **and** $s1\ -\tau 1\rightarrow\ s1'$
shows $s1' \approx s2 \wedge \mu 1^{++}\ s1'\ s1 \vee (\exists\, s2'.\ s2\ -\tau 2\rightarrow +\ s2' \wedge s1' \approx s2')$
proof −
from *assms* **obtain** *tl1* **where** *tr1*: $s1\ -1-tl1\rightarrow\ s1'$
and *τ1*: *τmove1 s1 tl1 s1'* **by**(*auto*)
from *simulation1*[*OF bisim tr1*]
obtain *s2' tl2* **where** *tr2*: $s2\ -2-tl2\rightarrow\ s2'$
and *bisim'*: $s1' \approx s2'$ **and** *tlsim*: $tl1 \sim tl2$ **by** *blast*
from *τinv*[*OF bisim tr1 tr2 bisim' tlsim*] *τ1* **have** *τ2*: *τmove2 s2 tl2 s2'* **by** *simp*
from *tr2 τ2* **have** $s2\ -\tau 2\rightarrow +\ s2'$ **by**(*auto*)
with *bisim'* **show** *?thesis* **by** *blast*
qed

lemma *simulation-silent2-aux*:
⟦ $s1 \approx s2$; $s2\ -\tau 2\rightarrow\ s2'$ ⟧ ⟹ $s1 \approx s2' \wedge \mu 2^{++}\ s2'\ s2 \vee (\exists\, s1'.\ s1\ -\tau 1\rightarrow +\ s1' \wedge s1' \approx s2')$
using *bisimulation-into-delay.simulation-silent1-aux*[*OF bisimulation-into-delay-flip*]
unfolding *flip-simps* **.**

lemma *simulation1-aux*:
assumes *bisim*: $s1 \approx s2$ **and** *tr1*: $s1\ -1-tl1\rightarrow\ s1'$ **and** *τ1*: ¬ *τmove1 s1 tl1 s1'*
shows ∃ *s2' s2'' tl2*. $s2\ -\tau 2\rightarrow *\ s2' \wedge s2'\ -2-tl2\rightarrow\ s2'' \wedge$ ¬ *τmove2 s2' tl2 s2''* $\wedge\ s1' \approx s2'' \wedge$ $tl1 \sim tl2$
proof −
from *simulation1*[*OF bisim tr1*]
obtain *s2' tl2* **where** *tr2*: $s2\ -2-tl2\rightarrow\ s2'$
and *bisim'*: $s1' \approx s2'$ **and** *tlsim*: $tl1 \sim tl2$ **by** *blast*
from *τinv*[*OF bisim tr1 tr2 bisim' tlsim*] *τ1* **have** *τ2*: ¬ *τmove2 s2 tl2 s2'* **by** *simp*
with *bisim' tr2 tlsim* **show** *?thesis* **by** *blast*
qed

lemma *simulation2-aux*:
⟦ $s1 \approx s2$; $s2\ -2-tl2\rightarrow\ s2'$; ¬ *τmove2 s2 tl2 s2'* ⟧
⟹ ∃ *s1' s1'' tl1*. $s1\ -\tau 1\rightarrow *\ s1' \wedge s1'\ -1-tl1\rightarrow\ s1'' \wedge$ ¬ *τmove1 s1' tl1 s1''* $\wedge\ s1'' \approx s2' \wedge tl1$ $\sim tl2$
using *bisimulation-into-delay.simulation1-aux*[*OF bisimulation-into-delay-flip*]
unfolding *flip-simps* **.**

lemma *delay-bisimulation-measure*:
assumes *wf-μ1*: *wfP μ1*
and *wf-μ2*: *wfP μ2*
shows *delay-bisimulation-measure trsys1 trsys2 bisim tlsim τmove1 τmove2 μ1 μ2*
apply(*unfold-locales*)
apply(*rule simulation-silent1-aux simulation-silent2-aux simulation1-aux simulation2-aux wf-μ1 wf-μ2*|*assumption*)+

done

lemma *delay-bisimulation*:
delay-bisimulation-diverge trsys1 trsys2 bisim tlsim τmove1 τmove2
proof −
interpret *delay-bisimulation-measure trsys1 trsys2 bisim tlsim τmove1 τmove2 λs s′. False λs s′. False*
by(*blast intro*: *delay-bisimulation-measure wfp-empty*)
show *?thesis* **..**
qed

end

sublocale *bisimulation-into-delay* < *delay-bisimulation-diverge*
by(*rule delay-bisimulation*)

lemma *delay-bisimulation-conv-bisimulation*:
delay-bisimulation-diverge trsys1 trsys2 bisim tlsim (*λs tl s′. False*) (*λs tl s′. False*) =
bisimulation trsys1 trsys2 bisim tlsim
(**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then interpret *delay-bisimulation-diverge trsys1 trsys2 bisim tlsim λs tl s′. False λs tl s′. False* **.**
show *?rhs* **by**(*unfold-locales*)(*fastforce simp add*: *τmoves-False dest*: *simulation1 simulation2*)+
next
assume *?rhs*
then interpret *bisimulation trsys1 trsys2 bisim tlsim* **.**
interpret *bisimulation-into-delay trsys1 trsys2 bisim tlsim λs tl s′. False λs tl s′. False*
by(*unfold-locales*)(*rule refl*)
show *?lhs* **by** *unfold-locales*
qed

context *bisimulation-final* **begin**

lemma *delay-bisimulation-final-base*:
delay-bisimulation-final-base trsys1 trsys2 bisim τmove1 τmove2 final1 final2
by(*unfold-locales*)(*auto simp add*: *bisim-final*)

end

sublocale *bisimulation-final* < *delay-bisimulation-final-base*
by(*rule delay-bisimulation-final-base*)

1.17.3 Transitivity for bisimulations

definition *bisim-compose* :: (*'s1*, *'s2*) *bisim* ⇒ (*'s2*, *'s3*) *bisim* ⇒ (*'s1*, *'s3*) *bisim* (**infixr** ‹$\circ_B$› *60*)
where (*bisim1* $\circ_B$ *bisim2*) *s1 s3* ≡ ∃ *s2*. *bisim1 s1 s2* ∧ *bisim2 s2 s3*

lemma *bisim-composeI* [*intro*]:
⟦ *bisim12 s1 s2*; *bisim23 s2 s3* ⟧ ⟹ (*bisim12* $\circ_B$ *bisim23*) *s1 s3*
by(*auto simp add*: *bisim-compose-def*)

lemma *bisim-composeE* [*elim*!]:
assumes *bisim*: (*bisim12* $\circ_B$ *bisim23*) *s1 s3*

obtains *s2* **where** *bisim12 s1 s2 bisim23 s2 s3*
by(*atomize-elim*)(*rule bisim*[*unfolded bisim-compose-def*])

lemma *bisim-compose-assoc* [*simp*]:
(*bisim12* $\circ_B$ *bisim23*) $\circ_B$ *bisim34* = *bisim12* $\circ_B$ *bisim23* $\circ_B$ *bisim34*
by(*auto simp add*: *fun-eq-iff*)

lemma *bisim-compose-conv-relcomp*:
case-prod (*bisim-compose bisim12 bisim23*) = ($\lambda x.\ x \in$ *relcomp* (*Collect* (*case-prod bisim12*)) (*Collect* (*case-prod bisim23*)))
by(*auto simp add*: *relcomp-unfold*)

lemma *list-all2-bisim-composeI*:
⟦ *list-all2 A xs ys*; *list-all2 B ys zs* ⟧
$\Longrightarrow$ *list-all2* (*A* $\circ_B$ *B*) *xs zs*
by(*rule list-all2-trans*) *auto*+

lemma *delay-bisimulation-diverge-compose*:
assumes *wbisim12*: *delay-bisimulation-diverge trsys1 trsys2 bisim12 tlsim12* τ*move1* τ*move2*
and *wbisim23*: *delay-bisimulation-diverge trsys2 trsys3 bisim23 tlsim23* τ*move2* τ*move3*
shows *delay-bisimulation-diverge trsys1 trsys3* (*bisim12* $\circ_B$ *bisim23*) (*tlsim12* $\circ_B$ *tlsim23*) τ*move1* τ*move3*
proof −
interpret *trsys1*: τ*trsys trsys1* τ*move1* **.**
interpret *trsys2*: τ*trsys trsys2* τ*move2* **.**
interpret *trsys3*: τ*trsys trsys3* τ*move3* **.**
interpret *wb12*: *delay-bisimulation-diverge trsys1 trsys2 bisim12 tlsim12* τ*move1* τ*move2* **by**(*auto intro*: *wbisim12*)
interpret *wb23*: *delay-bisimulation-diverge trsys2 trsys3 bisim23 tlsim23* τ*move2* τ*move3* **by**(*auto intro*: *wbisim23*)
show *?thesis*
proof
fix *s1 s3 s1'*
assume *bisim*: (*bisim12* $\circ_B$ *bisim23*) *s1 s3* **and** *tr1*: *trsys1.silent-move s1 s1'*
from *bisim* **obtain** *s2* **where** *bisim1*: *bisim12 s1 s2* **and** *bisim2*: *bisim23 s2 s3* **by** *blast*
from *wb12.simulation-silent1*[*OF bisim1 tr1*] **obtain** *s2'*
where *tr2*: *trsys2.silent-moves s2 s2'* **and** *bisim1'*: *bisim12 s1' s2'* **by** *blast*
from *wb23.simulation-silents1*[*OF bisim2 tr2*] **obtain** *s3'*
where *trsys3.silent-moves s3 s3' bisim23 s2' s3'* **by** *blast*
with *bisim1'* **show** $\exists$ *s3'*. *trsys3.silent-moves s3 s3'* $\wedge$ (*bisim12* $\circ_B$ *bisim23*) *s1' s3'*
by(*blast intro*: *bisim-composeI*)
next
fix *s1 s3 s3'*
assume *bisim*: (*bisim12* $\circ_B$ *bisim23*) *s1 s3* **and** *tr3*: *trsys3.silent-move s3 s3'*
from *bisim* **obtain** *s2* **where** *bisim1*: *bisim12 s1 s2* **and** *bisim2*: *bisim23 s2 s3* **by** *blast*
from *wb23.simulation-silent2*[*OF bisim2 tr3*] **obtain** *s2'*
where *tr2*: *trsys2.silent-moves s2 s2'* **and** *bisim2'*: *bisim23 s2' s3'* **by** *blast*
from *wb12.simulation-silents2*[*OF bisim1 tr2*] **obtain** *s1'*
where *trsys1.silent-moves s1 s1' bisim12 s1' s2'* **by** *blast*
with *bisim2'* **show** $\exists$ *s1'*. *trsys1.silent-moves s1 s1'* $\wedge$ (*bisim12* $\circ_B$ *bisim23*) *s1' s3'*
by(*blast intro*: *bisim-composeI*)
next
fix *s1 s3 tl1 s1'*
assume *bisim*: (*bisim12* $\circ_B$ *bisim23*) *s1 s3*

and *tr1*: *trsys1 s1 tl1 s1'* **and** *τ1*: ¬ *τmove1 s1 tl1 s1'*
from *bisim* **obtain** *s2* **where** *bisim1*: *bisim12 s1 s2* **and** *bisim2*: *bisim23 s2 s3* **by** *blast*
from *wb12.simulation1*[*OF bisim1 tr1 τ1*] **obtain** *s2' s2'' tl2*
where *tr21*: *trsys2.silent-moves s2 s2'* **and** *tr22*: *trsys2 s2' tl2 s2''* **and** *τ2*: ¬ *τmove2 s2' tl2 s2''*
and *bisim1'*: *bisim12 s1' s2''* **and** *tlsim1*: *tlsim12 tl1 tl2* **by** *blast*
from *wb23.simulation-silents1*[*OF bisim2 tr21*] **obtain** *s3'*
where *tr31*: *trsys3.silent-moves s3 s3'* **and** *bisim2'*: *bisim23 s2' s3'* **by** *blast*
from *wb23.simulation1*[*OF bisim2' tr22 τ2*] **obtain** *s3'' s3''' tl3*
where *trsys3.silent-moves s3' s3'' trsys3 s3'' tl3 s3'''*
¬ *τmove3 s3'' tl3 s3'''* *bisim23 s2'' s3'''* *tlsim23 tl2 tl3* **by** *blast*
with *tr31 bisim1' tlsim1*
show ∃ *s3' s3'' tl3*. *trsys3.silent-moves s3 s3'* ∧ *trsys3 s3' tl3 s3''* ∧ ¬ *τmove3 s3' tl3 s3''* ∧
(*bisim12* ∘$_B$ *bisim23*) *s1' s3''* ∧ (*tlsim12* ∘$_B$ *tlsim23*) *tl1 tl3*
by(*blast intro*: *rtranclp-trans bisim-composeI*)
next
fix *s1 s3 tl3 s3'*
assume *bisim*: (*bisim12* ∘$_B$ *bisim23*) *s1 s3*
and *tr3*: *trsys3 s3 tl3 s3'* **and** *τ3*: ¬ *τmove3 s3 tl3 s3'*
from *bisim* **obtain** *s2* **where** *bisim1*: *bisim12 s1 s2* **and** *bisim2*: *bisim23 s2 s3* **by** *blast*
from *wb23.simulation2*[*OF bisim2 tr3 τ3*] **obtain** *s2' s2'' tl2*
where *tr21*: *trsys2.silent-moves s2 s2'* **and** *tr22*: *trsys2 s2' tl2 s2''* **and** *τ2*: ¬ *τmove2 s2' tl2 s2''*
and *bisim2'*: *bisim23 s2'' s3'* **and** *tlsim2*: *tlsim23 tl2 tl3* **by** *blast*
from *wb12.simulation-silents2*[*OF bisim1 tr21*] **obtain** *s1'*
where *tr11*: *trsys1.silent-moves s1 s1'* **and** *bisim1'*: *bisim12 s1' s2'* **by** *blast*
from *wb12.simulation2*[*OF bisim1' tr22 τ2*] **obtain** *s1'' s1''' tl1*
where *trsys1.silent-moves s1' s1'' trsys1 s1'' tl1 s1'''*
¬ *τmove1 s1'' tl1 s1'''* *bisim12 s1''' s2''* *tlsim12 tl1 tl2* **by** *blast*
with *tr11 bisim2' tlsim2*
show ∃ *s1' s1'' tl1*. *trsys1.silent-moves s1 s1'* ∧ *trsys1 s1' tl1 s1''* ∧ ¬ *τmove1 s1' tl1 s1''* ∧
(*bisim12* ∘$_B$ *bisim23*) *s1'' s3'* ∧ (*tlsim12* ∘$_B$ *tlsim23*) *tl1 tl3*
by(*blast intro*: *rtranclp-trans bisim-composeI*)
next
fix *s1 s2*
assume (*bisim12* ∘$_B$ *bisim23*) *s1 s2*
thus *τtrsys.τdiverge trsys1 τmove1 s1* = *τtrsys.τdiverge trsys3 τmove3 s2*
by(*auto simp add*: *wb12.τdiverge-bisim-inv wb23.τdiverge-bisim-inv*)
qed
qed

lemma *bisimulation-bisim-compose*:
⟦ *bisimulation trsys1 trsys2 bisim12 tlsim12*; *bisimulation trsys2 trsys3 bisim23 tlsim23* ⟧
⟹ *bisimulation trsys1 trsys3* (*bisim-compose bisim12 bisim23*) (*bisim-compose tlsim12 tlsim23*)
unfolding *delay-bisimulation-conv-bisimulation*[*symmetric*]
by(*rule delay-bisimulation-diverge-compose*)

lemma *delay-bisimulation-diverge-final-compose*:
fixes *τmove1 τmove2*
assumes *wbisim12*: *delay-bisimulation-diverge-final trsys1 trsys2 bisim12 tlsim12 τmove1 τmove2 final1 final2*
and *wbisim23*: *delay-bisimulation-diverge-final trsys2 trsys3 bisim23 tlsim23 τmove2 τmove3 final2 final3*
shows *delay-bisimulation-diverge-final trsys1 trsys3* (*bisim12* ∘$_B$ *bisim23*) (*tlsim12* ∘$_B$ *tlsim23*)

τmove1 τmove3 final1 final3
proof −
interpret *trsys1*: *τtrsys trsys1 τmove1* **.**
interpret *trsys2*: *τtrsys trsys2 τmove2* **.**
interpret *trsys3*: *τtrsys trsys3 τmove3* **.**
interpret *wb12*: *delay-bisimulation-diverge-final trsys1 trsys2 bisim12 tlsim12 τmove1 τmove2 final1 final2*
by(*auto intro*: *wbisim12*)
interpret *wb23*: *delay-bisimulation-diverge-final trsys2 trsys3 bisim23 tlsim23 τmove2 τmove3 final2 final3*
by(*auto intro*: *wbisim23*)
interpret *delay-bisimulation-diverge trsys1 trsys3 bisim12 ∘B bisim23 tlsim12 ∘B tlsim23 τmove1 τmove3*
by(*rule delay-bisimulation-diverge-compose*)(*unfold-locales*)
show *?thesis*
proof
fix *s1 s3*
assume (*bisim12 ∘B bisim23*) *s1 s3 final1 s1*
from ‹(*bisim12 ∘B bisim23*) *s1 s3*› **obtain** *s2* **where** *bisim12 s1 s2* **and** *bisim23 s2 s3* **..**
from *wb12.final1-simulation*[*OF* ‹*bisim12 s1 s2*› ‹*final1 s1*›]
obtain *s2′* **where** *trsys2.silent-moves s2 s2′ bisim12 s1 s2′ final2 s2′* **by** *blast*
from *wb23.simulation-silents1*[*OF* ‹*bisim23 s2 s3*› ‹*trsys2.silent-moves s2 s2′*›]
obtain *s3′* **where** *trsys3.silent-moves s3 s3′ bisim23 s2′ s3′* **by** *blast*
from *wb23.final1-simulation*[*OF* ‹*bisim23 s2′ s3′*› ‹*final2 s2′*›]
obtain *s3″* **where** *trsys3.silent-moves s3′ s3″ bisim23 s2′ s3″ final3 s3″* **by** *blast*
from ‹*trsys3.silent-moves s3 s3′*› ‹*trsys3.silent-moves s3′ s3″*›
have *trsys3.silent-moves s3 s3″* **by**(*rule rtranclp-trans*)
moreover from ‹*bisim12 s1 s2′*› ‹*bisim23 s2′ s3″*›
have (*bisim12 ∘B bisim23*) *s1 s3″* **..**
ultimately show ∃ *s3′*. *trsys3.silent-moves s3 s3′* ∧ (*bisim12 ∘B bisim23*) *s1 s3′* ∧ *final3 s3′*
using ‹*final3 s3″*› **by** *iprover*
next
fix *s1 s3*
assume (*bisim12 ∘B bisim23*) *s1 s3 final3 s3*
from ‹(*bisim12 ∘B bisim23*) *s1 s3*› **obtain** *s2* **where** *bisim12 s1 s2* **and** *bisim23 s2 s3* **..**
from *wb23.final2-simulation*[*OF* ‹*bisim23 s2 s3*› ‹*final3 s3*›]
obtain *s2′* **where** *trsys2.silent-moves s2 s2′ bisim23 s2′ s3 final2 s2′* **by** *blast*
from *wb12.simulation-silents2*[*OF* ‹*bisim12 s1 s2*› ‹*trsys2.silent-moves s2 s2′*›]
obtain *s1′* **where** *trsys1.silent-moves s1 s1′ bisim12 s1′ s2′* **by** *blast*
from *wb12.final2-simulation*[*OF* ‹*bisim12 s1′ s2′*› ‹*final2 s2′*›]
obtain *s1″* **where** *trsys1.silent-moves s1′ s1″ bisim12 s1″ s2′ final1 s1″* **by** *blast*
from ‹*trsys1.silent-moves s1 s1′*› ‹*trsys1.silent-moves s1′ s1″*›
have *trsys1.silent-moves s1 s1″* **by**(*rule rtranclp-trans*)
moreover from ‹*bisim12 s1″ s2′*› ‹*bisim23 s2′ s3*›
have (*bisim12 ∘B bisim23*) *s1″ s3* **..**
ultimately show ∃ *s1′*. *trsys1.silent-moves s1 s1′* ∧ (*bisim12 ∘B bisim23*) *s1′ s3* ∧ *final1 s1′*
using ‹*final1 s1″*› **by** *iprover*
qed
qed

end

1.18 Bisimulation relations for the multithreaded semantics

theory *FWBisimulation*
imports
FWLTS
Bisimulation
begin

1.18.1 Definitions for lifting bisimulation relations

primrec *nta-bisim* :: $('t \Rightarrow ('x1 \times 'm1, 'x2 \times 'm2)\ bisim) \Rightarrow (('t,'x1,'m1)\ new\text{-}thread\text{-}action, ('t,'x2,'m2)\ new\text{-}thread\text{-}action)\ bisim$
where
[*code del*]: *nta-bisim bisim* (*NewThread t x m*) *ta* = ($\exists x'\ m'$. *ta* = *NewThread t x' m'* $\wedge$ *bisim t* (*x*, *m*) (*x'*, *m'*))
| *nta-bisim bisim* (*ThreadExists t b*) *ta* = (*ta* = *ThreadExists t b*)

lemma *nta-bisim-1-code* [*code*]:
nta-bisim bisim (*NewThread t x m*) *ta* = (*case ta of NewThread t' x' m'* $\Rightarrow$ *t* = *t'* $\wedge$ *bisim t* (*x*, *m*) (*x'*, *m'*) | - $\Rightarrow$ *False*)
by(*auto split*: *new-thread-action.split*)

lemma *nta-bisim-simps-sym* [*simp*]:
nta-bisim bisim ta (*NewThread t x m*) = ($\exists x'\ m'$. *ta* = *NewThread t x' m'* $\wedge$ *bisim t* (*x'*, *m'*) (*x*, *m*))
nta-bisim bisim ta (*ThreadExists t b*) = (*ta* = *ThreadExists t b*)
by(*cases ta*, *auto*)+

definition *ta-bisim* :: $('t \Rightarrow ('x1 \times 'm1, 'x2 \times 'm2)\ bisim) \Rightarrow (('l,'t,'x1,'m1,'w,'o)\ thread\text{-}action, ('l,'t,'x2,'m2,'w,'o)\ thread\text{-}action)\ bisim$
where
ta-bisim bisim ta1 ta2 $\equiv$
$\{\!|\ ta1\ |\!\}_l = \{\!|\ ta2\ |\!\}_l \wedge \{\!|\ ta1\ |\!\}_w = \{\!|\ ta2\ |\!\}_w \wedge \{\!|\ ta1\ |\!\}_c = \{\!|\ ta2\ |\!\}_c \wedge \{\!|\ ta1\ |\!\}_o = \{\!|\ ta2\ |\!\}_o \wedge \{\!|\ ta1\ |\!\}_i = \{\!|\ ta2\ |\!\}_i \wedge$
list-all2 (*nta-bisim bisim*) $\{\!|\ ta1\ |\!\}_t$ $\{\!|\ ta2\ |\!\}_t$

lemma *ta-bisim-empty* [*iff*]: *ta-bisim bisim* ε ε
by(*auto simp add*: *ta-bisim-def*)

lemma *ta-bisim-*ε [*simp*]:
ta-bisim b ε *ta'* $\longleftrightarrow$ *ta'* = ε *ta-bisim b ta* ε $\longleftrightarrow$ *ta* = ε
apply(*cases ta'*, *fastforce simp add*: *ta-bisim-def*)
apply(*cases ta*, *fastforce simp add*: *ta-bisim-def*)
done

lemma *nta-bisim-mono*:
assumes *major*: *nta-bisim bisim ta ta'*
and *mono*: $\bigwedge t\ s1\ s2.\ bisim\ t\ s1\ s2 \Longrightarrow bisim'\ t\ s1\ s2$
shows *nta-bisim bisim' ta ta'*
using *major* **by**(*cases ta*)(*auto intro*: *mono*)

lemma *ta-bisim-mono*:
assumes *major*: *ta-bisim bisim ta1 ta2*
and *mono*: $\bigwedge t\ s1\ s2.\ bisim\ t\ s1\ s2 \Longrightarrow bisim'\ t\ s1\ s2$
shows *ta-bisim bisim' ta1 ta2*

using *major*
by(*auto simp add*: *ta-bisim-def elim*!: *List.list-all2-mono nta-bisim-mono intro*: *mono*)

lemma *nta-bisim-flip* [*flip-simps*]:
nta-bisim (λ*t*. *flip* (*bisim t*)) = *flip* (*nta-bisim bisim*)
by(*rule ext*)(*case-tac x*, *auto simp add*: *flip-simps*)

lemma *ta-bisim-flip* [*flip-simps*]:
ta-bisim (λ*t*. *flip* (*bisim t*)) = *flip* (*ta-bisim bisim*)
by(*auto simp add*: *fun-eq-iff flip-simps ta-bisim-def*)

locale *FWbisimulation-base* =
r1: *multithreaded-base final1 r1 convert-RA* +
r2: *multithreaded-base final2 r2 convert-RA*
for *final1* :: *'x1* ⇒ *bool*
and *r1* :: (*'l*,*'t*,*'x1*,*'m1*,*'w*,*'o*) *semantics* (‹- ⊢ - *−1−*→ -› [*50*, *0*, *0*, *50*] *80*)
and *final2* :: *'x2* ⇒ *bool*
and *r2* :: (*'l*,*'t*,*'x2*,*'m2*,*'w*,*'o*) *semantics* (‹- ⊢ - *−2−*→ -› [*50*, *0*, *0*, *50*] *80*)
and *convert-RA* :: *'l released-locks* ⇒ *'o list*
+
fixes *bisim* :: *'t* ⇒ (*'x1* × *'m1*, *'x2* × *'m2*) *bisim* (‹- ⊢ -/ ≈ -› [*50*, *50*, *50*] *60*)
and *bisim-wait* :: (*'x1*, *'x2*) *bisim* (‹-/ ≈*w* -› [*50*, *50*] *60*)
begin

notation *r1.redT-syntax1* (‹- *−1−*▷→ -› [*50*,*0*,*0*,*50*] *80*)
notation *r2.redT-syntax1* (‹- *−2−*▷→ -› [*50*,*0*,*0*,*50*] *80*)

notation *r1.RedT* (‹- *−1−*▷→∗ -› [*50*,*0*,*50*] *80*)
notation *r2.RedT* (‹- *−2−*▷→∗ -› [*50*,*0*,*50*] *80*)

notation *r1.must-sync* (‹- ⊢ ⟨-,/ -⟩/ ≀*1*› [*50*,*0*,*0*] *81*)
notation *r2.must-sync* (‹- ⊢ ⟨-,/ -⟩/ ≀*2*› [*50*,*0*,*0*] *81*)

notation *r1.can-sync* (‹- ⊢ ⟨-,/ -⟩/ -/ ≀*1*› [*50*,*0*,*0*,*0*] *81*)
notation *r2.can-sync* (‹- ⊢ ⟨-,/ -⟩/ -/ ≀*2*› [*50*,*0*,*0*,*0*] *81*)

abbreviation *ta-bisim-bisim-syntax* (‹-/ ∼*m* -› [*50*, *50*] *60*)
where *ta1* ∼*m ta2* ≡ *ta-bisim bisim ta1 ta2*

definition *tbisim* :: *bool* ⇒ *'t* ⇒ (*'x1* × *'l released-locks*) *option* ⇒ *'m1* ⇒ (*'x2* × *'l released-locks*) *option* ⇒ *'m2* ⇒ *bool* **where**
⋀*ln*. *tbisim nw t ts1 m1 ts2 m2* ⟷
(*case ts1 of None* ⇒ *ts2* = *None*
| ⌊(*x1*, *ln*)⌋ ⇒ (∃ *x2*. *ts2* = ⌊(*x2*, *ln*)⌋ ∧ *t* ⊢ (*x1*, *m1*) ≈ (*x2*, *m2*) ∧ (*nw* ∨ *x1* ≈*w x2*)))

lemma *tbisim-NoneI*: *tbisim w t None m None m'*
by(*simp add*: *tbisim-def*)

lemma *tbisim-SomeI*:
⋀*ln*. ⟦ *t* ⊢ (*x*, *m*) ≈ (*x'*, *m'*); *nw* ∨ *x* ≈*w x'* ⟧ ⟹ *tbisim nw t* (*Some* (*x*, *ln*)) *m* (*Some* (*x'*, *ln*)) *m'*
by(*simp add*: *tbisim-def*)

lemma *tbisim-cases*[*consumes 1*, *case-names None Some*]:
assumes *major*: *tbisim nw t ts1 m1 ts2 m2*

and $[\![$ *ts1* = *None*; *ts2* = *None* $]\!] \Longrightarrow$ *thesis*
and $\bigwedge x\ ln\ x'.\ [\![$ *ts1* = $\lfloor(x,\ ln)\rfloor$; *ts2* = $\lfloor(x',\ ln)\rfloor$; $t \vdash (x,\ m1) \approx (x',\ m2)$; $nw \vee x \approx_w x'$ $]\!] \Longrightarrow$ *thesis*
shows *thesis*
using *assms*
by(*auto simp add*: *tbisim-def*)

definition *mbisim* :: (('*l*,'*t*,'*x1*,'*m1*,'*w*) *state*, ('*l*,'*t*,'*x2*,'*m2*,'*w*) *state*) *bisim* (‹- ≈*m* -› [*50*, *50*] *60*)
where
s1 ≈*m s2* ≡
finite (*dom* (*thr s1*)) ∧ *locks s1* = *locks s2* ∧ *wset s1* = *wset s2* ∧ *wset-thread-ok* (*wset s1*) (*thr s1*)
∧
interrupts s1 = *interrupts s2* ∧
(∀ *t*. *tbisim* (*wset s2 t* = *None*) *t* (*thr s1 t*) (*shr s1*) (*thr s2 t*) (*shr s2*))

lemma *mbisim-thrNone-eq*: *s1* ≈*m s2* ⟹ *thr s1 t* = *None* ⟷ *thr s2 t* = *None*
unfolding *mbisim-def tbisim-def*
apply(*clarify*)
apply(*erule allE*[**where** *x*=*t*])
apply(*clarsimp*)
done

lemma *mbisim-thrD1*:
$\bigwedge ln.\ [\![$ *s1* ≈*m s2*; *thr s1 t* = $\lfloor(x,\ ln)\rfloor$ $]\!]$
⟹ ∃ *x'*. *thr s2 t* = $\lfloor(x',\ ln)\rfloor$ ∧ $t \vdash (x,\ shr\ s1) \approx (x',\ shr\ s2)$ ∧ (*wset s1 t* = *None* ∨ $x \approx_w x'$)
by(*fastforce simp add*: *mbisim-def tbisim-def*)

lemma *mbisim-thrD2*:
$\bigwedge ln.\ [\![$ *s1* ≈*m s2*; *thr s2 t* = $\lfloor(x,\ ln)\rfloor$ $]\!]$
⟹ ∃ *x'*. *thr s1 t* = $\lfloor(x',\ ln)\rfloor$ ∧ $t \vdash (x',\ shr\ s1) \approx (x,\ shr\ s2)$ ∧ (*wset s2 t* = *None* ∨ $x' \approx_w x$)
by(*frule mbisim-thrNone-eq*[**where** *t*=*t*])(*cases thr s1 t*,(*fastforce simp add*: *mbisim-def tbisim-def*)+)

lemma *mbisim-dom-eq*: *s1* ≈*m s2* ⟹ *dom* (*thr s1*) = *dom* (*thr s2*)
apply(*clarsimp simp add*: *dom-def fun-eq-iff simp del*: *not-None-eq*)
apply(*rule Collect-cong*)
apply(*drule mbisim-thrNone-eq*)
apply(*simp del*: *not-None-eq*)
done

lemma *mbisim-wset-thread-ok1*:
s1 ≈*m s2* ⟹ *wset-thread-ok* (*wset s1*) (*thr s1*)
by(*clarsimp simp add*: *mbisim-def*)

lemma *mbisim-wset-thread-ok2*:
assumes *s1* ≈*m s2*
shows *wset-thread-ok* (*wset s2*) (*thr s2*)
using *assms*
apply(*clarsimp simp add*: *mbisim-def*)
apply(*auto intro*!: *wset-thread-okI simp add*: *mbisim-thrNone-eq*[*OF assms*, *THEN sym*] *dest*: *wset-thread-okD*)
done

lemma *mbisimI*:
[$\![$ *finite* (*dom* (*thr s1*)); *locks s1* = *locks s2*; *wset s1* = *wset s2*; *interrupts s1* = *interrupts s2*;
wset-thread-ok (*wset s1*) (*thr s1*);
$\bigwedge t$. *thr s1 t* = *None* ⟹ *thr s2 t* = *None*;

$\bigwedge t\ x1\ ln.\ thr\ s1\ t = \lfloor (x1,\ ln) \rfloor \Longrightarrow \exists\, x2.\ thr\ s2\ t = \lfloor (x2,\ ln) \rfloor \wedge t \vdash (x1,\ shr\ s1) \approx (x2,\ shr\ s2)$
$\wedge\ (wset\ s2\ t = None \vee x1 \approx w\ x2)\ ]\!]$
$\Longrightarrow s1 \approx m\ s2$
by(*fastforce simp add*: *mbisim-def tbisim-def*)

lemma *mbisimI2*:
$[\![\ finite\ (dom\ (thr\ s2));\ locks\ s1 = locks\ s2;\ wset\ s1 = wset\ s2;\ interrupts\ s1 = interrupts\ s2;$
wset-thread-ok (*wset s2*) (*thr s2*);
$\bigwedge t.\ thr\ s2\ t = None \Longrightarrow thr\ s1\ t = None;$
$\bigwedge t\ x2\ ln.\ thr\ s2\ t = \lfloor (x2,\ ln) \rfloor \Longrightarrow \exists\, x1.\ thr\ s1\ t = \lfloor (x1,\ ln) \rfloor \wedge t \vdash (x1,\ shr\ s1) \approx (x2,\ shr\ s2)$
$\wedge\ (wset\ s2\ t = None \vee x1 \approx w\ x2)\ ]\!]$
$\Longrightarrow s1 \approx m\ s2$
apply(*auto simp add*: *mbisim-def tbisim-def*)
prefer *2*
apply(*rule wset-thread-okI*)
apply(*case-tac thr s2 t*)
apply(*auto dest*!: *wset-thread-okD*)[*1*]
apply *fastforce*
apply(*erule back-subst*[**where** *P*=*finite*])
apply(*clarsimp simp add*: *dom-def fun-eq-iff simp del*: *not-None-eq*)
defer
apply(*rename-tac t*)
apply(*case-tac* [!] *thr s2 t*)
by *fastforce*+

lemma *mbisim-finite1*:
$s1 \approx m\ s2 \Longrightarrow finite\ (dom\ (thr\ s1))$
by(*simp add*: *mbisim-def*)

lemma *mbisim-finite2*:
$s1 \approx m\ s2 \Longrightarrow finite\ (dom\ (thr\ s2))$
by(*frule mbisim-finite1*)(*simp add*: *mbisim-dom-eq*)

definition *mta-bisim* :: ('*t* × ('*l*,'*t*,'*x1*,'*m1*,'*w*,'*o*) *thread-action*,
'*t* × ('*l*,'*t*,'*x2*,'*m2*,'*w*,'*o*) *thread-action*) *bisim*
(‹-/ ∼*T* -› [*50*, *50*] *60*)
where *tta1* ∼*T tta2* ≡ *fst tta1* = *fst tta2* ∧ *snd tta1* ∼*m snd tta2*

lemma *mta-bisim-conv* [*simp*]: $(t,\ ta1) \sim T\ (t',\ ta2) \longleftrightarrow t = t' \wedge ta1 \sim m\ ta2$
by(*simp add*: *mta-bisim-def*)

definition *bisim-inv* :: *bool* **where**
bisim-inv ≡ $(\forall\, s1\ ta1\ s1'\ s2\ t.\ t \vdash s1 \approx s2 \longrightarrow t \vdash s1\ -1-ta1\rightarrow\ s1' \longrightarrow (\exists\, s2'.\ t \vdash s1' \approx s2'))\ \wedge$
$(\forall\, s2\ ta2\ s2'\ s1\ t.\ t \vdash s1 \approx s2 \longrightarrow t \vdash s2\ -2-ta2\rightarrow\ s2' \longrightarrow (\exists\, s1'.\ t \vdash s1' \approx s2'))$

lemma *bisim-invI*:
$[\![\ \bigwedge s1\ ta1\ s1'\ s2\ t.\ [\![\ t \vdash s1 \approx s2;\ t \vdash s1\ -1-ta1\rightarrow\ s1'\ ]\!] \Longrightarrow \exists\, s2'.\ t \vdash s1' \approx s2';$
$\bigwedge s2\ ta2\ s2'\ s1\ t.\ [\![\ t \vdash s1 \approx s2;\ t \vdash s2\ -2-ta2\rightarrow\ s2'\ ]\!] \Longrightarrow \exists\, s1'.\ t \vdash s1' \approx s2'\ ]\!]$
$\Longrightarrow bisim\text{-}inv$
by(*auto simp add*: *bisim-inv-def*)

lemma *bisim-invD1*:
$[\![\ bisim\text{-}inv;\ t \vdash s1 \approx s2;\ t \vdash s1\ -1-ta1\rightarrow\ s1'\ ]\!] \Longrightarrow \exists\, s2'.\ t \vdash s1' \approx s2'$
unfolding *bisim-inv-def* **by** *blast*

lemma *bisim-invD2*:
⟦ *bisim-inv*; $t \vdash s1 \approx s2$; $t \vdash s2\ -2-ta2\rightarrow\ s2'$ ⟧ ⟹ ∃ *s1'*. $t \vdash s1' \approx s2'$
unfolding *bisim-inv-def* **by** *blast*

lemma *thread-oks-bisim-inv*:
⟦ ∀ *t*. *ts1 t* = *None* ⟷ *ts2 t* = *None*; *list-all2* (*nta-bisim bisim*) *tas1 tas2* ⟧
⟹ *thread-oks ts1 tas1* ⟷ *thread-oks ts2 tas2*
proof(*induct tas2 arbitrary*: *tas1 ts1 ts2*)
case *Nil* **thus** *?case* **by**(*simp*)
next
case (*Cons ta2 TAS2 tas1 TS1 TS2*)
note *IH* = ‹⋀*ts1 tas1 ts2*. ⟦ ∀ *t*. *ts1 t* = *None* ⟷ *ts2 t* = *None*; *list-all2* (*nta-bisim bisim*) *tas1 TAS2* ⟧
⟹ *thread-oks ts1 tas1* ⟷ *thread-oks ts2 TAS2*›
note *eqNone* = ‹∀ *t*. *TS1 t* = *None* ⟷ *TS2 t* = *None*›[*rule-format*]
hence *fti*: *free-thread-id TS1* = *free-thread-id TS2* **by**(*auto simp add*: *free-thread-id-def*)
from ‹*list-all2* (*nta-bisim bisim*) *tas1* (*ta2* # *TAS2*)›
obtain *ta1 TAS1* **where** *tas1* = *ta1* # *TAS1 nta-bisim bisim ta1 ta2 list-all2* (*nta-bisim bisim*) *TAS1 TAS2*
by(*auto simp add*: *list-all2-Cons2*)
moreover
{ fix *t*
from ‹*nta-bisim bisim ta1 ta2*› **have** *redT-updT' TS1 ta1 t* = *None* ⟷ *redT-updT' TS2 ta2 t* = *None*
by(*cases ta1*, *auto split*: *if-split-asm simp add*: *eqNone*) **}**
ultimately have *thread-oks* (*redT-updT' TS1 ta1*) *TAS1* ⟷ *thread-oks* (*redT-updT' TS2 ta2*) *TAS2*
by −(*rule IH*, *auto*)
moreover from ‹*nta-bisim bisim ta1 ta2*› *fti* **have** *thread-ok TS1 ta1* = *thread-ok TS2 ta2* **by**(*cases ta1*, *auto*)
ultimately show *?case* **using** ‹*tas1* = *ta1* # *TAS1*› **by** *auto*
qed

lemma *redT-updT-nta-bisim-inv*:
⟦ *nta-bisim bisim ta1 ta2*; *ts1 T* = *None* ⟷ *ts2 T* = *None* ⟧ ⟹ *redT-updT ts1 ta1 T* = *None* ⟷ *redT-updT ts2 ta2 T* = *None*
by(*cases ta1*, *auto*)

lemma *redT-updTs-nta-bisim-inv*:
⟦ *list-all2* (*nta-bisim bisim*) *tas1 tas2*; *ts1 T* = *None* ⟷ *ts2 T* = *None* ⟧
⟹ *redT-updTs ts1 tas1 T* = *None* ⟷ *redT-updTs ts2 tas2 T* = *None*
proof(*induct tas1 arbitrary*: *tas2 ts1 ts2*)
case *Nil* **thus** *?case* **by**(*simp*)
next
case (*Cons TA1 TAS1 tas2 TS1 TS2*)
note *IH* = ‹⋀*tas2 ts1 ts2*. ⟦*list-all2* (*nta-bisim bisim*) *TAS1 tas2*; (*ts1 T* = *None*) = (*ts2 T* = *None*)⟧
⟹ (*redT-updTs ts1 TAS1 T* = *None*) = (*redT-updTs ts2 tas2 T* = *None*)›
from ‹*list-all2* (*nta-bisim bisim*) (*TA1* # *TAS1*) *tas2*›
obtain *TA2 TAS2* **where** *tas2* = *TA2* # *TAS2 nta-bisim bisim TA1 TA2 list-all2* (*nta-bisim bisim*) *TAS1 TAS2*
by(*auto simp add*: *list-all2-Cons1*)
from ‹*nta-bisim bisim TA1 TA2*› ‹(*TS1 T* = *None*) = (*TS2 T* = *None*)›

have *redT-updT TS1 TA1 T = None* ⟷ *redT-updT TS2 TA2 T = None*
by(*rule redT-updT-nta-bisim-inv*)
with *IH*[*OF* ‹*list-all2* (*nta-bisim bisim*) *TAS1 TAS2*›, *of redT-updT TS1 TA1 redT-updT TS2 TA2*]
‹*tas2 = TA2 # TAS2*›
show *?case* **by** *simp*
qed

end

lemma *tbisim-flip* [*flip-simps*]:
FWbisimulation-base.tbisim (λ*t*. *flip* (*bisim t*)) (*flip bisim-wait*) *w t ts2 m2 ts1 m1 =*
FWbisimulation-base.tbisim bisim bisim-wait w t ts1 m1 ts2 m2
unfolding *FWbisimulation-base.tbisim-def flip-simps* **by** *auto*

lemma *mbisim-flip* [*flip-simps*]:
FWbisimulation-base.mbisim (λ*t*. *flip* (*bisim t*)) (*flip bisim-wait*) *s2 s1 =*
FWbisimulation-base.mbisim bisim bisim-wait s1 s2
apply(*rule iffI*)
apply(*frule FWbisimulation-base.mbisim-dom-eq*)
apply(*frule FWbisimulation-base.mbisim-wset-thread-ok2*)
apply(*fastforce simp add*: *FWbisimulation-base.mbisim-def flip-simps*)
apply(*frule FWbisimulation-base.mbisim-dom-eq*)
apply(*frule FWbisimulation-base.mbisim-wset-thread-ok2*)
apply(*fastforce simp add*: *FWbisimulation-base.mbisim-def flip-simps*)
done

lemma *mta-bisim-flip* [*flip-simps*]:
FWbisimulation-base.mta-bisim (λ*t*. *flip* (*bisim t*)) = *flip* (*FWbisimulation-base.mta-bisim bisim*)
by(*auto simp add*: *fun-eq-iff flip-simps FWbisimulation-base.mta-bisim-def*)

lemma *flip-const* [*simp*]: *flip* (λ*a b*. *c*) = (λ*a b*. *c*)
by(*rule flip-def*)

lemma *mbisim-K-flip* [*flip-simps*]:
FWbisimulation-base.mbisim (λ*t*. *flip* (*bisim t*)) (λ*x1 x2*. *c*) *s1 s2 =*
FWbisimulation-base.mbisim bisim (λ*x1 x2*. *c*) *s2 s1*
using *mbisim-flip*[*of bisim* λ*x1 x2*. *c s1 s2*]
unfolding *flip-const* **.**

context *FWbisimulation-base* **begin**

lemma *mbisim-actions-ok-bisim-no-join-12*:
assumes *mbisim*: *mbisim s1 s2*
and *collect-cond-actions* {|*ta1*|}$_c$ = {}
and *ta-bisim bisim ta1 ta2*
and *r1.actions-ok s1 t ta1*
shows *r2.actions-ok s2 t ta2*
using *assms mbisim-thrNone-eq*[*OF mbisim*]
by(*auto simp add*: *ta-bisim-def mbisim-def intro*: *thread-oks-bisim-inv*[*THEN iffD1*] *r2.may-join-cond-action-oks*)

lemma *mbisim-actions-ok-bisim-no-join-21*:
⟦ *mbisim s1 s2*; *collect-cond-actions* {|*ta2*|}$_c$ = {}; *ta-bisim bisim ta1 ta2*; *r2.actions-ok s2 t ta2* ⟧
⟹ *r1.actions-ok s1 t ta1*
using *FWbisimulation-base.mbisim-actions-ok-bisim-no-join-12*[**where** *bisim*=λ*t*. *flip* (*bisim t*) **and**

bisim-wait=flip bisim-wait]
unfolding *flip-simps* **.**

lemma *mbisim-actions-ok-bisim-no-join*:
⟦ *mbisim s1 s2*; *collect-cond-actions* ⦃*ta1*⦄$_C$ = {}; *ta-bisim bisim ta1 ta2* ⟧
⟹ *r1.actions-ok s1 t ta1* = *r2.actions-ok s2 t ta2*
apply(*rule iffI*)
apply(*erule* (*3*) *mbisim-actions-ok-bisim-no-join-12*)
apply(*erule mbisim-actions-ok-bisim-no-join-21*[**where** *?ta2.0* = *ta2*])
apply(*simp add*: *ta-bisim-def*)
apply *assumption*+
done

end

locale *FWbisimulation-base-aux* = *FWbisimulation-base* +
r1: *multithreaded final1 r1 convert-RA* +
r2: *multithreaded final2 r2 convert-RA* +
constrains *final1* :: *'x1* ⇒ *bool*
and *r1* :: (*'l*,*'t*,*'x1*,*'m1*,*'w*, *'o*) *semantics*
and *final2* :: *'x2* ⇒ *bool*
and *r2* :: (*'l*,*'t*,*'x2*,*'m2*,*'w*, *'o*) *semantics*
and *convert-RA* :: *'l released-locks* ⇒ *'o list*
and *bisim* :: *'t* ⇒ (*'x1* × *'m1*, *'x2* × *'m2*) *bisim*
and *bisim-wait* :: (*'x1*, *'x2*) *bisim*
begin

lemma *FWbisimulation-base-aux-flip*:
FWbisimulation-base-aux final2 r2 final1 r1
by(*unfold-locales*)

end

lemma *FWbisimulation-base-aux-flip-simps* [*flip-simps*]:
FWbisimulation-base-aux final2 r2 final1 r1 = *FWbisimulation-base-aux final1 r1 final2 r2*
by(*blast intro*: *FWbisimulation-base-aux.FWbisimulation-base-aux-flip*)

sublocale *FWbisimulation-base-aux* < *mthr*:
bisimulation-final-base
r1.redT
r2.redT
mbisim
mta-bisim
r1.mfinal
r2.mfinal
.

declare *split-paired-Ex* [*simp del*]

1.18.2 Lifting for delay bisimulations

locale *FWdelay-bisimulation-base* =
FWbisimulation-base - - - r2 convert-RA bisim bisim-wait +
r1: *τmultithreaded final1 r1 convert-RA τmove1* +

r2: *τmultithreaded final2 r2 convert-RA τmove2*
for *r2* :: (*'l*,*'t*,*'x2*,*'m2*,*'w*,*'o*) *semantics* (‹- ⊢ - −2−→ -›) [*50*,*0*,*0*,*50*] *80*)
and *convert-RA* :: *'l released-locks* ⇒ *'o list*
and *bisim* :: *'t* ⇒ (*'x1* × *'m1*, *'x2* × *'m2*) *bisim* (‹- ⊢ -/ ≈ -› [*50*, *50*, *50*] *60*)
and *bisim-wait* :: (*'x1*, *'x2*) *bisim* (‹-/ ≈*w* -› [*50*, *50*] *60*)
and *τmove1* :: (*'l*,*'t*,*'x1*,*'m1*,*'w*,*'o*) *τmoves*
and *τmove2* :: (*'l*,*'t*,*'x2*,*'m2*,*'w*,*'o*) *τmoves*
begin

abbreviation *τmred1* :: (*'l*,*'t*,*'x1*,*'m1*,*'w*) *state* ⇒ (*'l*,*'t*,*'x1*,*'m1*,*'w*) *state* ⇒ *bool*
where *τmred1* ≡ *r1.τmredT*

abbreviation *τmred2* :: (*'l*,*'t*,*'x2*,*'m2*,*'w*) *state* ⇒ (*'l*,*'t*,*'x2*,*'m2*,*'w*) *state* ⇒ *bool*
where *τmred2* ≡ *r2.τmredT*

abbreviation *mτmove1* :: ((*'l*,*'t*,*'x1*,*'m1*,*'w*) *state*, *'t* × (*'l*,*'t*,*'x1*,*'m1*,*'w*,*'o*) *thread-action*) *trsys*
where *mτmove1* ≡ *r1.mτmove*

abbreviation *mτmove2* :: ((*'l*,*'t*,*'x2*,*'m2*,*'w*) *state*, *'t* × (*'l*,*'t*,*'x2*,*'m2*,*'w*,*'o*) *thread-action*) *trsys*
where *mτmove2* ≡ *r2.mτmove*

abbreviation *τmRed1* :: (*'l*,*'t*,*'x1*,*'m1*,*'w*) *state* ⇒ (*'l*,*'t*,*'x1*,*'m1*,*'w*) *state* ⇒ *bool*
where *τmRed1* ≡ *τmred1*^**

abbreviation *τmRed2* :: (*'l*,*'t*,*'x2*,*'m2*,*'w*) *state* ⇒ (*'l*,*'t*,*'x2*,*'m2*,*'w*) *state* ⇒ *bool*
where *τmRed2* ≡ *τmred2*^**

abbreviation *τmtRed1* :: (*'l*,*'t*,*'x1*,*'m1*,*'w*) *state* ⇒ (*'l*,*'t*,*'x1*,*'m1*,*'w*) *state* ⇒ *bool*
where *τmtRed1* ≡ *τmred1*^++

abbreviation *τmtRed2* :: (*'l*,*'t*,*'x2*,*'m2*,*'w*) *state* ⇒ (*'l*,*'t*,*'x2*,*'m2*,*'w*) *state* ⇒ *bool*
where *τmtRed2* ≡ *τmred2*^++

lemma *bisim-inv-τs1-inv*:
assumes *inv*: *bisim-inv*
and *bisim*: *t* ⊢ *s1* ≈ *s2*
and *red*: *r1.silent-moves t s1 s1'*
obtains *s2'* **where** *t* ⊢ *s1'* ≈ *s2'*
proof(*atomize-elim*)
from *red bisim* **show** ∃ *s2'*. *t* ⊢ *s1'* ≈ *s2'*
by(*induct rule*: *rtranclp-induct*)(*fastforce elim*: *bisim-invD1*[*OF inv*])+
qed

lemma *bisim-inv-τs2-inv*:
assumes *inv*: *bisim-inv*
and *bisim*: *t* ⊢ *s1* ≈ *s2*
and *red*: *r2.silent-moves t s2 s2'*
obtains *s1'* **where** *t* ⊢ *s1'* ≈ *s2'*
proof(*atomize-elim*)
from *red bisim* **show** ∃ *s1'*. *t* ⊢ *s1'* ≈ *s2'*
by(*induct rule*: *rtranclp-induct*)(*fastforce elim*: *bisim-invD2*[*OF inv*])+
qed

primrec *activate-cond-action1* :: (*'l*,*'t*,*'x1*,*'m1*,*'w*) *state* ⇒ (*'l*,*'t*,*'x2*,*'m2*,*'w*) *state* ⇒

```
                        't conditional-action ⇒ ('l,'t,'x1,'m1,'w) state
where
  activate-cond-action1 s1 s2 (Join t) =
   (case thr s1 t of None ⇒ s1
          | ⌊(x1, ln1)⌋ ⇒ (case thr s2 t of None ⇒ s1
                                 | ⌊(x2, ln2)⌋ ⇒
  if final2 x2 ∧ ln2 = no-wait-locks
  then redT-upd-ε s1 t
                (SOME x1'. r1.silent-moves t (x1, shr s1) (x1', shr s1) ∧ final1 x1' ∧
                        t ⊢ (x1', shr s1) ≈ (x2, shr s2))
                (shr s1)
  else s1))
| activate-cond-action1 s1 s2 Yield = s1

primrec activate-cond-actions1 :: ('l,'t,'x1,'m1,'w) state ⇒ ('l,'t,'x2,'m2,'w) state
                        ⇒ ('t conditional-action) list ⇒ ('l,'t,'x1,'m1,'w) state
where
  activate-cond-actions1 s1 s2 [] = s1
| activate-cond-actions1 s1 s2 (ct # cts) = activate-cond-actions1 (activate-cond-action1 s1 s2 ct) s2
cts

primrec activate-cond-action2 :: ('l,'t,'x1,'m1,'w) state ⇒ ('l,'t,'x2,'m2,'w) state ⇒
                        't conditional-action ⇒ ('l,'t,'x2,'m2,'w) state
where
 activate-cond-action2 s1 s2 (Join t) =
   (case thr s2 t of None ⇒ s2
          | ⌊(x2, ln2)⌋ ⇒ (case thr s1 t of None ⇒ s2
                                 | ⌊(x1, ln1)⌋ ⇒
  if final1 x1 ∧ ln1 = no-wait-locks
  then redT-upd-ε s2 t
                (SOME x2'. r2.silent-moves t (x2, shr s2) (x2', shr s2) ∧ final2 x2' ∧
                        t ⊢ (x1, shr s1) ≈ (x2', shr s2))
                (shr s2)
  else s2))
| activate-cond-action2 s1 s2 Yield = s2

primrec activate-cond-actions2 :: ('l,'t,'x1,'m1,'w) state ⇒ ('l,'t,'x2,'m2,'w) state ⇒
                        ('t conditional-action) list ⇒ ('l,'t,'x2,'m2,'w) state
where
  activate-cond-actions2 s1 s2 [] = s2
| activate-cond-actions2 s1 s2 (ct # cts) = activate-cond-actions2 s1 (activate-cond-action2 s1 s2 ct)
cts

end

lemma activate-cond-action1-flip [flip-simps]:
  FWdelay-bisimulation-base.activate-cond-action1 final2 r2 final1 (λt. flip (bisim t)) τmove2 s2 s1 =
   FWdelay-bisimulation-base.activate-cond-action2 final1 final2 r2 bisim τmove2 s1 s2
apply(rule ext)
apply(case-tac x)
apply(simp-all only: FWdelay-bisimulation-base.activate-cond-action1.simps
                     FWdelay-bisimulation-base.activate-cond-action2.simps flip-simps)
done
```

lemma *activate-cond-actions1-flip* [*flip-simps*]:
FWdelay-bisimulation-base.activate-cond-actions1 final2 r2 final1 (λ*t*. *flip* (*bisim t*)) *τmove2 s2 s1*
=
FWdelay-bisimulation-base.activate-cond-actions2 final1 final2 r2 bisim τmove2 s1 s2
(**is** *?lhs* = *?rhs*)
proof(*rule ext*)
fix *xs*
show *?lhs xs* = *?rhs xs*
by(*induct xs arbitrary*: *s2*)
(*simp-all only*: *FWdelay-bisimulation-base.activate-cond-actions1.simps*
FWdelay-bisimulation-base.activate-cond-actions2.simps flip-simps)
qed

lemma *activate-cond-action2-flip* [*flip-simps*]:
FWdelay-bisimulation-base.activate-cond-action2 final2 final1 r1 (λ*t*. *flip* (*bisim t*)) *τmove1 s2 s1* =
FWdelay-bisimulation-base.activate-cond-action1 final1 r1 final2 bisim τmove1 s1 s2
apply(*rule ext*)
apply(*case-tac x*)
apply(*simp-all only*: *FWdelay-bisimulation-base.activate-cond-action1.simps*
FWdelay-bisimulation-base.activate-cond-action2.simps flip-simps)
done

lemma *activate-cond-actions2-flip* [*flip-simps*]:
FWdelay-bisimulation-base.activate-cond-actions2 final2 final1 r1 (λ*t*. *flip* (*bisim t*)) *τmove1 s2 s1*
=
FWdelay-bisimulation-base.activate-cond-actions1 final1 r1 final2 bisim τmove1 s1 s2
(**is** *?lhs* = *?rhs*)
proof(*rule ext*)
fix *xs*
show *?lhs xs* = *?rhs xs*
by(*induct xs arbitrary*: *s1*)
(*simp-all only*: *FWdelay-bisimulation-base.activate-cond-actions1.simps*
FWdelay-bisimulation-base.activate-cond-actions2.simps flip-simps)
qed

context *FWdelay-bisimulation-base* **begin**

lemma *shr-activate-cond-action1* [*simp*]: *shr* (*activate-cond-action1 s1 s2 ct*) = *shr s1*
by(*cases ct*) *simp-all*

lemma *shr-activate-cond-actions1* [*simp*]: *shr* (*activate-cond-actions1 s1 s2 cts*) = *shr s1*
by(*induct cts arbitrary*: *s1*) *auto*

lemma *shr-activate-cond-action2* [*simp*]: *shr* (*activate-cond-action2 s1 s2 ct*) = *shr s2*
by(*cases ct*) *simp-all*

lemma *shr-activate-cond-actions2* [*simp*]: *shr* (*activate-cond-actions2 s1 s2 cts*) = *shr s2*
by(*induct cts arbitrary*: *s2*) *auto*

lemma *locks-activate-cond-action1* [*simp*]: *locks* (*activate-cond-action1 s1 s2 ct*) = *locks s1*
by(*cases ct*) *simp-all*

lemma *locks-activate-cond-actions1* [*simp*]: *locks* (*activate-cond-actions1 s1 s2 cts*) = *locks s1*
by(*induct cts arbitrary*: *s1*) *auto*

lemma *locks-activate-cond-action2* [*simp*]: *locks* (*activate-cond-action2 s1 s2 ct*) = *locks s2*
by(*cases ct*) *simp-all*

lemma *locks-activate-cond-actions2* [*simp*]: *locks* (*activate-cond-actions2 s1 s2 cts*) = *locks s2*
by(*induct cts arbitrary*: *s2*) *auto*

lemma *wset-activate-cond-action1* [*simp*]: *wset* (*activate-cond-action1 s1 s2 ct*) = *wset s1*
by(*cases ct*) *simp-all*

lemma *wset-activate-cond-actions1* [*simp*]: *wset* (*activate-cond-actions1 s1 s2 cts*) = *wset s1*
by(*induct cts arbitrary*: *s1*) *auto*

lemma *wset-activate-cond-action2* [*simp*]: *wset* (*activate-cond-action2 s1 s2 ct*) = *wset s2*
by(*cases ct*) *simp-all*

lemma *wset-activate-cond-actions2* [*simp*]: *wset* (*activate-cond-actions2 s1 s2 cts*) = *wset s2*
by(*induct cts arbitrary*: *s2*) *auto*

lemma *interrupts-activate-cond-action1* [*simp*]: *interrupts* (*activate-cond-action1 s1 s2 ct*) = *interrupts s1*
by(*cases ct*) *simp-all*

lemma *interrupts-activate-cond-actions1* [*simp*]: *interrupts* (*activate-cond-actions1 s1 s2 cts*) = *interrupts s1*
by(*induct cts arbitrary*: *s1*) *auto*

lemma *interrupts-activate-cond-action2* [*simp*]: *interrupts* (*activate-cond-action2 s1 s2 ct*) = *interrupts s2*
by(*cases ct*) *simp-all*

lemma *interrupts-activate-cond-actions2* [*simp*]: *interrupts* (*activate-cond-actions2 s1 s2 cts*) = *interrupts s2*
by(*induct cts arbitrary*: *s2*) *auto*

end

locale *FWdelay-bisimulation-lift-aux* =
  FWdelay-bisimulation-base - - - - - - - τmove1 τmove2 +
  r1: *τmultithreaded-wf final1 r1 convert-RA τmove1* +
  r2: *τmultithreaded-wf final2 r2 convert-RA τmove2*
  for *τmove1* :: (*'l*,*'t*,*'x1*,*'m1*,*'w*,*'o*) *τmoves*
  and *τmove2* :: (*'l*,*'t*,*'x2*,*'m2*,*'w*,*'o*) *τmoves*
begin

lemma *FWdelay-bisimulation-lift-aux-flip*:
  FWdelay-bisimulation-lift-aux final2 r2 final1 r1 τmove2 τmove1
by *unfold-locales*

end

lemma *FWdelay-bisimulation-lift-aux-flip-simps* [*flip-simps*]:
  FWdelay-bisimulation-lift-aux final2 r2 final1 r1 τmove2 τmove1 =
   FWdelay-bisimulation-lift-aux final1 r1 final2 r2 τmove1 τmove2

by(*auto dest*: *FWdelay-bisimulation-lift-aux.FWdelay-bisimulation-lift-aux-flip simp only*: *flip-flip*)

context *FWdelay-bisimulation-lift-aux* **begin**

lemma *cond-actions-ok-τmred1-inv*:
  assumes *red*: *τmred1 s1 s1'*
  and *ct*: *r1.cond-action-ok s1 t ct*
  shows *r1.cond-action-ok s1' t ct*
using *ct*
proof(*cases ct*)
  case (*Join t'*)
  show *?thesis* **using** *red ct*
  proof(*cases thr s1 t'*)
    case *None* **with** *red ct Join* **show** *?thesis*
      by(*fastforce elim*!: *r1.mthr.silent-move.cases r1.redT.cases r1.mτmove.cases rtrancl3p-cases*
                dest: *r1.silent-tl split*: *if-split-asm*)
  next
    case (*Some a*) **with** *red ct Join* **show** *?thesis*
      by(*fastforce elim*!: *r1.mthr.silent-move.cases r1.redT.cases r1.mτmove.cases rtrancl3p-cases*
                dest: *r1.silent-tl r1.final-no-red split*: *if-split-asm simp add*: *redT-updWs-def*)
  qed
next
  case *Yield* **thus** *?thesis* **by** *simp*
qed

lemma *cond-actions-ok-τmred2-inv*:
  ⟦ *τmred2 s2 s2'*; *r2.cond-action-ok s2 t ct* ⟧ ⟹ *r2.cond-action-ok s2' t ct*
using *FWdelay-bisimulation-lift-aux.cond-actions-ok-τmred1-inv*[*OF FWdelay-bisimulation-lift-aux-flip*]
.

lemma *cond-actions-ok-τmRed1-inv*:
  ⟦ *τmRed1 s1 s1'*; *r1.cond-action-ok s1 t ct* ⟧ ⟹ *r1.cond-action-ok s1' t ct*
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *cond-actions-ok-τmred1-inv*)+

lemma *cond-actions-ok-τmRed2-inv*:
  ⟦ *τmRed2 s2 s2'*; *r2.cond-action-ok s2 t ct* ⟧ ⟹ *r2.cond-action-ok s2' t ct*
by(*rule FWdelay-bisimulation-lift-aux.cond-actions-ok-τmRed1-inv*[*OF FWdelay-bisimulation-lift-aux-flip*])

end

locale *FWdelay-bisimulation-lift* =
  FWdelay-bisimulation-lift-aux +
  constrains *final1* :: *'x1* ⇒ *bool*
  and *r1* :: (*'l*, *'t*, *'x1*, *'m1*, *'w*, *'o*) *semantics*
  and *final2* :: *'x2* ⇒ *bool*
  and *r2* :: (*'l*, *'t*, *'x2*, *'m2*, *'w*, *'o*) *semantics*
  and *convert-RA* :: *'l released-locks* ⇒ *'o list*
  and *bisim* :: *'t* ⇒ (*'x1* × *'m1*, *'x2* × *'m2*) *bisim*
  and *bisim-wait* :: (*'x1*, *'x2*) *bisim*
  and *τmove1* :: (*'l*, *'t*, *'x1*, *'m1*, *'w*, *'o*) *τmoves*
  and *τmove2* :: (*'l*, *'t*, *'x2*, *'m2*, *'w*, *'o*) *τmoves*
  assumes *τinv-locale*: *τinv* (*r1 t*) (*r2 t*) (*bisim t*) (*ta-bisim bisim*) *τmove1 τmove2*

sublocale *FWdelay-bisimulation-lift* < *τinv r1 t r2 t bisim t ta-bisim bisim τmove1 τmove2* **for** *t*

by(*rule τinv-locale*)

context *FWdelay-bisimulation-lift* **begin**

lemma *FWdelay-bisimulation-lift-flip*:
FWdelay-bisimulation-lift final2 r2 final1 r1 (λ*t*. *flip* (*bisim t*)) *τmove2 τmove1*
apply(*rule FWdelay-bisimulation-lift.intro*)
apply(*rule FWdelay-bisimulation-lift-aux-flip*)
apply(*rule FWdelay-bisimulation-lift-axioms.intro*)
apply(*unfold flip-simps*)
apply(*unfold-locales*)
done

end

lemma *FWdelay-bisimulation-lift-flip-simps* [*flip-simps*]:
FWdelay-bisimulation-lift final2 r2 final1 r1 (λ*t*. *flip* (*bisim t*)) *τmove2 τmove1* =
FWdelay-bisimulation-lift final1 r1 final2 r2 bisim τmove1 τmove2
by(*auto dest*: *FWdelay-bisimulation-lift.FWdelay-bisimulation-lift-flip simp only*: *flip-flip*)

context *FWdelay-bisimulation-lift* **begin**

lemma *τinv-lift*: *τinv r1.redT r2.redT mbisim mta-bisim mτmove1 mτmove2*
proof
fix *s1 s2 tl1 s1′ tl2 s2′*
assume *s1* ≈*m s2 s1′* ≈*m s2′ tl1* ∼*T tl2 r1.redT s1 tl1 s1′ r2.redT s2 tl2 s2′*
moreover obtain *t ta1* **where** *tl1*: *tl1* = (*t*, *ta1*) **by**(*cases tl1*)
moreover obtain *t′ ta2* **where** *tl2*: *tl2* = (*t′*, *ta2*) **by**(*cases tl2*)
moreover obtain *ls1 ts1 ws1 m1 is1* **where** *s1*: *s1* = (*ls1*, (*ts1*, *m1*), *ws1*, *is1*) **by**(*cases s1*) *fastforce*
moreover obtain *ls2 ts2 ws2 m2 is2* **where** *s2*: *s2* = (*ls2*, (*ts2*, *m2*), *ws2*, *is2*) **by**(*cases s2*) *fastforce*
moreover obtain *ls1′ ts1′ ws1′ m1′ is1′* **where** *s1′*: *s1′* = (*ls1′*, (*ts1′*, *m1′*), *ws1′*, *is1′*) **by**(*cases s1′*) *fastforce*
moreover obtain *ls2′ ts2′ ws2′ m2′ is2′* **where** *s2′*: *s2′* = (*ls2′*, (*ts2′*, *m2′*), *ws2′*, *is2′*) **by**(*cases s2′*) *fastforce*
ultimately have *mbisim*: (*ls1*, (*ts1*, *m1*), *ws1*, *is1*) ≈*m* (*ls2*, (*ts2*, *m2*), *ws2*, *is2*)
and *mbisim′*: (*ls1′*, (*ts1′*, *m1′*), *ws1′*, *is1′*) ≈*m* (*ls2′*, (*ts2′*, *m2′*), *ws2′*, *is2′*)
and *mred1*: (*ls1*, (*ts1*, *m1*), *ws1*, *is1*) −1−*t*▷*ta1*→ (*ls1′*, (*ts1′*, *m1′*), *ws1′*, *is1′*)
and *mred2*: (*ls2*, (*ts2*, *m2*), *ws2*, *is2*) −2−*t*▷*ta2*→ (*ls2′*, (*ts2′*, *m2′*), *ws2′*, *is2′*)
and *tasim*: *ta1* ∼*m ta2* **and** *tt′*: *t′* = *t* **by** *simp-all*
from *mbisim* **have** *ls*: *ls1* = *ls2* **and** *ws*: *ws1* = *ws2* **and** *is*: *is1* = *is2*
and *tbisim*: ⋀*t*. *tbisim* (*ws2 t* = *None*) *t* (*ts1 t*) *m1* (*ts2 t*) *m2* **by**(*simp-all add*: *mbisim-def*)
from *mbisim′* **have** *ls′*: *ls1′* = *ls2′* **and** *ws′*: *ws1′* = *ws2′* **and** *is′*: *is1′* = *is2′*
and *tbisim′*: ⋀*t*. *tbisim* (*ws2′ t* = *None*) *t* (*ts1′ t*) *m1′* (*ts2′ t*) *m2′* **by**(*simp-all add*: *mbisim-def*)
from *mred1 r1.redT-thread-not-disappear*[*OF mred1*]
obtain *x1 ln1 x1′ ln1′* **where** *tst1*: *ts1 t* = ⌊(*x1*, *ln1*)⌋
and *tst1′*: *ts1′ t* = ⌊(*x1′*, *ln1′*)⌋
by(*fastforce elim*!: *r1.redT.cases*)
from *mred2 r2.redT-thread-not-disappear*[*OF mred2*]
obtain *x2 ln2 x2′ ln2′* **where** *tst2*: *ts2 t* = ⌊(*x2*, *ln2*)⌋
and *tst2′*: *ts2′ t* = ⌊(*x2′*, *ln2′*)⌋ **by**(*fastforce elim*!: *r2.redT.cases*)
from *tbisim*[*of t*] *tst1 tst2 ws* **have** *bisim*: *t* ⊢ (*x1*, *m1*) ≈ (*x2*, *m2*)
and *ln*: *ln1* = *ln2* **by**(*auto simp add*: *tbisim-def*)

```
  from tbisim'[of t] tst1' tst2' have bisim': t ⊢ (x1', m1') ≈ (x2', m2')
    and ln': ln1' = ln2' by(auto simp add: tbisim-def)
  show mτmove1 s1 tl1 s1' = mτmove2 s2 tl2 s2' unfolding s1 s2 s1' s2' tt' tl1 tl2
  proof −
    show mτmove1 (ls1, (ts1, m1), ws1, is1) (t, ta1) (ls1', (ts1', m1'), ws1', is1') =
         mτmove2 (ls2, (ts2, m2), ws2, is2) (t, ta2) (ls2', (ts2', m2'), ws2', is2')
      (is ?lhs = ?rhs)
    proof
      assume mτ: ?lhs
      with tst1 tst1' obtain τ1: τmove1 (x1, m1) ta1 (x1', m1')
        and ln1: ln1 = no-wait-locks by(fastforce elim!: r1.mτmove.cases)
      from τ1 have ta1 = ε by(rule r1.silent-tl)
      with mred1 τ1 tst1 tst1' ln1 have red1: t ⊢ (x1, m1) −1−ta1→ (x1', m1')
        by(auto elim!: r1.redT.cases rtrancl3p-cases)
      from tasim ‹ta1 = ε› have [simp]: ta2 = ε by(simp)
      with mred2 ln1 ln tst2 tst2' have red2: t ⊢ (x2, m2) −2−ε→ (x2', m2')
        by(fastforce elim!: r2.redT.cases rtrancl3p-cases)
      from τ1 τinv[OF bisim red1 red2] bisim' tasim
      have τ2: τmove2 (x2, m2) ε (x2', m2') by simp
      with tst2 tst2' ln ln1 show ?rhs by −(rule r2.mτmove.intros, auto)
    next
      assume mτ: ?rhs
      with tst2 tst2' obtain τ2: τmove2 (x2, m2) ta2 (x2', m2')
        and ln2: ln2 = no-wait-locks by(fastforce elim!: r2.mτmove.cases)
      from τ2 have ta2 = ε by(rule r2.silent-tl)
      with mred2 τ2 tst2 tst2' ln2 have red2: t ⊢ (x2, m2) −2−ta2→ (x2', m2')
        by(auto elim!: r2.redT.cases rtrancl3p-cases)
      from tasim ‹ta2 = ε› have [simp]: ta1 = ε by simp
      with mred1 ln2 ln tst1 tst1' have red1: t ⊢ (x1, m1) −1−ε→ (x1', m1')
        by(fastforce elim!: r1.redT.cases rtrancl3p-cases)
      from τ2 τinv[OF bisim red1 red2] bisim' tasim
      have τ1: τmove1 (x1, m1) ε (x1', m1') by auto
      with tst1 tst1' ln ln2 show ?lhs unfolding ‹ta1 = ε›
        by−(rule r1.mτmove.intros, auto)
    qed
  qed
qed

end

sublocale FWdelay-bisimulation-lift < mthr: τinv r1.redT r2.redT mbisim mta-bisim mτmove1 mτmove2
by(rule τinv-lift)

locale FWdelay-bisimulation-final-base =
  FWdelay-bisimulation-lift-aux +
  constrains final1 :: 'x1 ⇒ bool
  and r1 :: ('l,'t,'x1,'m1,'w, 'o) semantics
  and final2 :: 'x2 ⇒ bool
  and r2 :: ('l,'t,'x2,'m2,'w, 'o) semantics
  and convert-RA :: 'l released-locks ⇒ 'o list
  and bisim :: 't ⇒ ('x1 × 'm1, 'x2 × 'm2) bisim
  and bisim-wait :: ('x1, 'x2) bisim
  and τmove1 :: ('l,'t,'x1,'m1,'w, 'o) τmoves
  and τmove2 :: ('l,'t,'x2,'m2,'w, 'o) τmoves
```

assumes *delay-bisim-locale*:
delay-bisimulation-final-base (*r1 t*) (*r2 t*) (*bisim t*) *τmove1 τmove2* (λ(*x1*, *m*). *final1 x1*) (λ(*x2*, *m*). *final2 x2*)

sublocale *FWdelay-bisimulation-final-base* <
delay-bisimulation-final-base r1 t r2 t bisim t ta-bisim bisim τmove1 τmove2
λ(*x1*, *m*). *final1 x1* λ(*x2*, *m*). *final2 x2*
for *t*
by(*rule delay-bisim-locale*)

context *FWdelay-bisimulation-final-base* **begin**

lemma *FWdelay-bisimulation-final-base-flip*:
FWdelay-bisimulation-final-base final2 r2 final1 r1 (λ*t*. *flip* (*bisim t*)) *τmove2 τmove1*
apply(*rule FWdelay-bisimulation-final-base.intro*)
apply(*rule FWdelay-bisimulation-lift-aux-flip*)
apply(*rule FWdelay-bisimulation-final-base-axioms.intro*)
apply(*rule delay-bisimulation-final-base-flip*)
done

end

lemma *FWdelay-bisimulation-final-base-flip-simps* [*flip-simps*]:
FWdelay-bisimulation-final-base final2 r2 final1 r1 (λ*t*. *flip* (*bisim t*)) *τmove2 τmove1* =
FWdelay-bisimulation-final-base final1 r1 final2 r2 bisim τmove1 τmove2
by(*auto dest*: *FWdelay-bisimulation-final-base.FWdelay-bisimulation-final-base-flip simp only*: *flip-flip*)

context *FWdelay-bisimulation-final-base* **begin**

lemma *cond-actions-ok-bisim-ex-τ1-inv*:
fixes *ls ts1 m1 ws is ts2 m2 ct*
defines *s1'* ≡ *activate-cond-action1* (*ls*, (*ts1*, *m1*), *ws*, *is*) (*ls*, (*ts2*, *m2*), *ws*, *is*) *ct*
assumes *mbisim*: ⋀*t'*. *t'* ≠ *t* ⟹ *tbisim* (*ws t'* = *None*) *t'* (*ts1 t'*) *m1* (*ts2 t'*) *m2*
and *ts1t*: *ts1 t* = *Some xln*
and *ts2t*: *ts2 t* = *Some xln'*
and *ct*: *r2.cond-action-ok* (*ls*, (*ts2*, *m2*), *ws*, *is*) *t ct*
shows *τmRed1* (*ls*, (*ts1*, *m1*), *ws*, *is*) *s1'*
and ⋀*t'*. *t'* ≠ *t* ⟹ *tbisim* (*ws t'* = *None*) *t'* (*thr s1' t'*) *m1* (*ts2 t'*) *m2*
and *r1.cond-action-ok s1' t ct*
and *thr s1' t* = *Some xln*
proof −
have *τmRed1* (*ls*, (*ts1*, *m1*), *ws*, *is*) *s1'* ∧
(∀ *t'*. *t'* ≠ *t* ⟶ *tbisim* (*ws t'* = *None*) *t'* (*thr s1' t'*) *m1* (*ts2 t'*) *m2*) ∧
r1.cond-action-ok s1' t ct ∧ *thr s1' t* = ⌊*xln*⌋
using *ct*
proof(*cases ct*)
case (*Join t'*)
show *?thesis*
proof(*cases ts1 t'*)
case *None*
with *mbisim ts1t* **have** *t* ≠ *t'* **by** *auto*
moreover from *None Join* **have** *s1'* = (*ls*, (*ts1*, *m1*), *ws*, *is*) **by**(*simp add*: *s1'-def*)
ultimately show *?thesis* **using** *mbisim Join ct None ts1t* **by**(*simp add*: *tbisim-def*)
next

```
    case (Some xln)
    moreover obtain x1 ln where xln = (x1, ln) by(cases xln)
    ultimately have ts1t': ts1 t' = ⌊(x1, ln)⌋ by simp
    from Join ct Some ts2t have tt': t' ≠ t by auto
    from mbisim[OF tt'] ts1t' obtain x2 where ts2t': ts2 t' = ⌊(x2, ln)⌋
      and bisim: t' ⊢ (x1, m1) ≈ (x2, m2) by(auto simp add: tbisim-def)
    from ct Join ts2t' have final2: final2 x2 and ln: ln = no-wait-locks
    and wst': ws t' = None by simp-all
    let ?x1' = SOME x. r1.silent-moves t' (x1, m1) (x, m1) ∧ final1 x ∧ t' ⊢ (x, m1) ≈ (x2, m2)
    { from final2-simulation[OF bisim] final2 obtain x1' m1'
        where r1.silent-moves t' (x1, m1) (x1', m1') and t' ⊢ (x1', m1') ≈ (x2, m2)
        and final1 x1' by auto
      moreover hence m1' = m1 using bisim by(auto dest: r1.red-rtrancl-τ-heapD-inv)
       ultimately have ∃x. r1.silent-moves t' (x1, m1) (x, m1) ∧ final1 x ∧ t' ⊢ (x, m1) ≈ (x2,
m2)
        by blast }
    from someI-ex[OF this] have red1: r1.silent-moves t' (x1, m1) (?x1', m1)
      and final1: final1 ?x1' and bisim': t' ⊢ (?x1', m1) ≈ (x2, m2) by blast+
    let ?S1' = redT-upd-ε (ls, (ts1, m1), ws, is) t' ?x1' m1
    from r1.silent-moves-into-RedT-τ-inv[where ?s=(ls, (ts1, m1), ws, is) and t=t', simplified, OF
red1]
      bisim ts1t' ln wst'
    have Red1: τmRed1 (ls, (ts1, m1), ws, is) ?S1' by auto
    moreover from Join ln ts1t' final1 wst' tt'
    have ct': r1.cond-action-ok ?S1' t ct by(auto intro: finfun-ext)
    { fix t''
      assume t ≠ t''
      with Join mbisim[OF this[symmetric]] bisim' ts1t' ts2t' wst' s1'-def
      have tbisim (ws t'' = None) t'' (thr s1' t'') m1 (ts2 t'') m2
        by(auto simp add: tbisim-def redT-updLns-def o-def finfun-Diag-const2) }
    moreover from Join ts1t' ts2t' final2 ln have s1' = ?S1' by(simp add: s1'-def)
    ultimately show ?thesis using Red1 ct' ts1t' tt' ts1t by(auto)
  qed
next
  case Yield thus ?thesis using mbisim ts1t by(simp add: s1'-def)
qed
thus τmRed1 (ls, (ts1, m1), ws, is) s1'
  and ⋀t'. t' ≠ t ⟹ tbisim (ws t' = None) t' (thr s1' t') m1 (ts2 t') m2
  and r1.cond-action-ok s1' t ct
  and thr s1' t = ⌊xln⌋ by blast+
qed

lemma cond-actions-oks-bisim-ex-τ1-inv:
  fixes ls ts1 m1 ws is ts2 m2 cts
  defines s1' ≡ activate-cond-actions1 (ls, (ts1, m1), ws, is) (ls, (ts2, m2), ws, is) cts
  assumes tbisim: ⋀t'. t' ≠ t ⟹ tbisim (ws t' = None) t' (ts1 t') m1 (ts2 t') m2
  and ts1t: ts1 t = Some xln
  and ts2t: ts2 t = Some xln'
  and ct: r2.cond-action-oks (ls, (ts2, m2), ws, is) t cts
  shows τmRed1 (ls, (ts1, m1), ws, is) s1'
  and ⋀t'. t' ≠ t ⟹ tbisim (ws t' = None) t' (thr s1' t') m1 (ts2 t') m2
  and r1.cond-action-oks s1' t cts
  and thr s1' t = Some xln
using tbisim ts1t ct unfolding s1'-def
```

proof(*induct cts arbitrary*: *ts1*)
case (*Cons ct cts*)
note *IH1* = ‹⋀*ts1*. ⟦⋀*t'*. *t'* ≠ *t* ⟹ *tbisim* (*ws t'* = *None*) *t'* (*ts1 t'*) *m1* (*ts2 t'*) *m2*; *ts1 t* = ⌊*xln*⌋;
r2.cond-action-oks (*ls*, (*ts2*, *m2*), *ws*, *is*) *t cts*⟧
⟹ *τmred1*** (*ls*, (*ts1*, *m1*), *ws*, *is*) (*activate-cond-actions1* (*ls*, (*ts1*, *m1*), *ws*, *is*) (*ls*, (*ts2*, *m2*), *ws*, *is*) *cts*)›
note *IH2* = ‹⋀*t' ts1*. ⟦*t'* ≠ *t*; ⋀*t'*. *t'* ≠ *t* ⟹ *tbisim* (*ws t'* = *None*) *t'* (*ts1 t'*) *m1* (*ts2 t'*) *m2*; *ts1 t* = ⌊*xln*⌋;
r2.cond-action-oks (*ls*, (*ts2*, *m2*), *ws*, *is*) *t cts*⟧
⟹ *tbisim* (*ws t'* = *None*) *t'* (*thr* (*activate-cond-actions1* (*ls*, (*ts1*, *m1*), *ws*, *is*) (*ls*, (*ts2*, *m2*), *ws*, *is*) *cts*) *t'*) *m1* (*ts2 t'*) *m2*›
note *IH3* = ‹⋀*ts1*. ⟦⋀*t'*. *t'* ≠ *t* ⟹ *tbisim* (*ws t'* = *None*) *t'* (*ts1 t'*) *m1* (*ts2 t'*) *m2*; *ts1 t* = ⌊*xln*⌋;
r2.cond-action-oks (*ls*, (*ts2*, *m2*), *ws*, *is*) *t cts*⟧
⟹ *r1.cond-action-oks* (*activate-cond-actions1* (*ls*, (*ts1*, *m1*), *ws*, *is*) (*ls*, (*ts2*, *m2*), *ws*, *is*) *cts*) *t cts*›
note *IH4* = ‹⋀*ts1*. ⟦⋀*t'*. *t'* ≠ *t* ⟹ *tbisim* (*ws t'* = *None*) *t'* (*ts1 t'*) *m1* (*ts2 t'*) *m2*; *ts1 t* = ⌊*xln*⌋;
r2.cond-action-oks (*ls*, (*ts2*, *m2*), *ws*, *is*) *t cts*⟧
⟹ *thr* (*activate-cond-actions1* (*ls*, (*ts1*, *m1*), *ws*, *is*) (*ls*, (*ts2*, *m2*), *ws*, *is*) *cts*) *t* = ⌊*xln*⌋›
{ fix *ts1*
assume *tbisim*: ⋀*t'*. *t'* ≠ *t* ⟹ *tbisim* (*ws t'* = *None*) *t'* (*ts1 t'*) *m1* (*ts2 t'*) *m2*
and *ts1t*: *ts1 t* = ⌊*xln*⌋
and *ct*: *r2.cond-action-oks* (*ls*, (*ts2*, *m2*), *ws*, *is*) *t* (*ct* # *cts*)
from *ct* **have** *1*: *r2.cond-action-ok* (*ls*, (*ts2*, *m2*), *ws*, *is*) *t ct*
and *2*: *r2.cond-action-oks* (*ls*, (*ts2*, *m2*), *ws*, *is*) *t cts* **by** *auto*
let *?s1'* = *activate-cond-action1* (*ls*, (*ts1*, *m1*), *ws*, *is*) (*ls*, (*ts2*, *m2*), *ws*, *is*) *ct*
from *cond-actions-ok-bisim-ex-τ1-inv*[*OF tbisim*, *OF - ts1t ts2t 1*]
have *tbisim'*: ⋀*t'*. *t'* ≠ *t* ⟹ *tbisim* (*ws t'* = *None*) *t'* (*thr ?s1' t'*) *m1* (*ts2 t'*) *m2*
and *red*: *τmRed1* (*ls*, (*ts1*, *m1*), *ws*, *is*) *?s1'* **and** *ct'*: *r1.cond-action-ok ?s1' t ct*
and *ts1't*: *thr ?s1' t* = ⌊*xln*⌋ **by** *blast+*
let *?s1''* = *activate-cond-actions1 ?s1'* (*ls*, (*ts2*, *m2*), *ws*, *is*) *cts*
have *locks ?s1'* = *ls shr ?s1'* = *m1 wset ?s1'* = *ws interrupts ?s1'* = *is* **by** *simp-all*
hence *s1'*: (*ls*, (*thr ?s1'*, *m1*), *ws*, *is*) = *?s1'* **by**(*cases ?s1'*) *auto*
from *IH1*[*OF tbisim'*, *OF - ts1't 2*] *s1'* **have** *red'*: *τmRed1 ?s1' ?s1''* **by** *simp*
with *red* **show** *τmRed1* (*ls*, (*ts1*, *m1*), *ws*, *is*) (*activate-cond-actions1* (*ls*, (*ts1*, *m1*), *ws*, *is*) (*ls*, (*ts2*, *m2*), *ws*, *is*) (*ct* # *cts*))
by *auto*
{ fix *t'*
assume *t't*: *t'* ≠ *t*
from *IH2*[*OF t't tbisim'*, *OF - ts1't 2*] *s1'*
show *tbisim* (*ws t'* = *None*) *t'* (*thr* (*activate-cond-actions1* (*ls*, (*ts1*, *m1*), *ws*, *is*) (*ls*, (*ts2*, *m2*), *ws*, *is*) (*ct* # *cts*)) *t'*) *m1* (*ts2 t'*) *m2*
by *auto* **}**
from *red' ct'* **have** *r1.cond-action-ok ?s1'' t ct* **by**(*rule cond-actions-ok-τmRed1-inv*)
with *IH3*[*OF tbisim'*, *OF - ts1't 2*] *s1'*
show *r1.cond-action-oks* (*activate-cond-actions1* (*ls*, (*ts1*, *m1*), *ws*, *is*) (*ls*, (*ts2*, *m2*), *ws*, *is*) (*ct* # *cts*)) *t* (*ct* # *cts*)
by *auto*
from *ts1't IH4*[*OF tbisim'*, *OF - ts1't 2*] *s1'*
show *thr* (*activate-cond-actions1* (*ls*, (*ts1*, *m1*), *ws*, *is*) (*ls*, (*ts2*, *m2*), *ws*, *is*) (*ct* # *cts*)) *t* = ⌊*xln*⌋ **by** *auto* **}**
qed(*auto*)

lemma *cond-actions-ok-bisim-ex-τ2-inv*:
fixes *ls ts1 m1 is ws ts2 m2 ct*

defines *s2'* ≡ *activate-cond-action2* (*ls*, (*ts1*, *m1*), *ws*, *is*) (*ls*, (*ts2*, *m2*), *ws*, *is*) *ct*
assumes *mbisim*: ⋀*t'*. *t'* ≠ *t* ⟹ *tbisim* (*ws t'* = *None*) *t'* (*ts1 t'*) *m1* (*ts2 t'*) *m2*
and *ts1t*: *ts1 t* = *Some xln*
and *ts2t*: *ts2 t* = *Some xln'*
and *ct*: *r1.cond-action-ok* (*ls*, (*ts1*, *m1*), *ws*, *is*) *t ct*
shows *τmRed2* (*ls*, (*ts2*, *m2*), *ws*, *is*) *s2'*
and ⋀*t'*. *t'* ≠ *t* ⟹ *tbisim* (*ws t'* = *None*) *t'* (*ts1 t'*) *m1* (*thr s2' t'*) *m2*
and *r2.cond-action-ok s2' t ct*
and *thr s2' t* = *Some xln'*
unfolding *s2'-def*
by(*blast intro*: *FWdelay-bisimulation-final-base.cond-actions-ok-bisim-ex-τ1-inv*[*OF FWdelay-bisimulation-final-base-flip*,
where *bisim-wait* = *flip bisim-wait*, *unfolded flip-simps*, *OF mbisim - - ct*, *OF - ts2t ts1t*])+

lemma *cond-actions-oks-bisim-ex-τ2-inv*:
fixes *ls ts1 m1 ws is ts2 m2 cts*
defines *s2'* ≡ *activate-cond-actions2* (*ls*, (*ts1*, *m1*), *ws*, *is*) (*ls*, (*ts2*, *m2*), *ws*, *is*) *cts*
assumes *tbisim*: ⋀*t'*. *t'* ≠ *t* ⟹ *tbisim* (*ws t'* = *None*) *t'* (*ts1 t'*) *m1* (*ts2 t'*) *m2*
and *ts1t*: *ts1 t* = *Some xln*
and *ts2t*: *ts2 t* = *Some xln'*
and *ct*: *r1.cond-action-oks* (*ls*, (*ts1*, *m1*), *ws*, *is*) *t cts*
shows *τmRed2* (*ls*, (*ts2*, *m2*), *ws*, *is*) *s2'*
and ⋀*t'*. *t'* ≠ *t* ⟹ *tbisim* (*ws t'* = *None*) *t'* (*ts1 t'*) *m1* (*thr s2' t'*) *m2*
and *r2.cond-action-oks s2' t cts*
and *thr s2' t* = *Some xln'*
unfolding *s2'-def*
by(*blast intro*: *FWdelay-bisimulation-final-base.cond-actions-oks-bisim-ex-τ1-inv*[*OF FWdelay-bisimulation-final-base-flip*,
where *bisim-wait* = *flip bisim-wait*, *unfolded flip-simps*, *OF tbisim - - ct*, *OF - ts2t ts1t*])+

lemma *mfinal1-inv-simulation*:
assumes *s1* ≈*m s2*
shows ∃*s2'*. *r2.mthr.silent-moves s2 s2'* ∧ *s1* ≈*m s2'* ∧ *r1.final-threads s1* ⊆ *r2.final-threads s2'*
∧ *shr s2'* = *shr s2*
proof −
from ‹*s1* ≈*m s2*› **have** *finite* (*dom* (*thr s1*)) **by**(*auto dest*: *mbisim-finite1*)
moreover have *r1.final-threads s1* ⊆ *dom* (*thr s1*) **by**(*auto simp add*: *r1.final-thread-def*)
ultimately have *finite* (*r1.final-threads s1*) **by**(*blast intro*: *finite-subset*)
thus *?thesis* **using** ‹*s1* ≈*m s2*›
proof(*induct A*≡*r1.final-threads s1 arbitrary*: *s1 s2 rule*: *finite-induct*)
case *empty*
from ‹{} = *r1.final-threads s1*›[*symmetric*] **have** ∀*t*. ¬ *r1.final-thread s1 t* **by**(*auto*)
with ‹*s1* ≈*m s2*› **show** *?case* **by** *blast*
next
case (*insert t A*)
define *s1'* **where** *s1'* = (*locks s1*, ((*thr s1*)(*t* := *None*)), *shr s1*), *wset s1*, *interrupts s1*)
define *s2'* **where** *s2'* = (*locks s2*, ((*thr s2*)(*t* := *None*)), *shr s2*), *wset s2*, *interrupts s2*)
from ‹*t* ∉ *A*› ‹*insert t A* = *r1.final-threads s1*› **have** *A* = *r1.final-threads s1'*
unfolding *s1'-def* **by**(*auto simp add*: *r1.final-thread-def r1.final-threads-def*)
moreover from ‹*insert t A* = *r1.final-threads s1*› **have** *r1.final-thread s1 t* **by** *auto*
hence *wset s1 t* = *None* **by**(*auto simp add*: *r1.final-thread-def*)
with ‹*s1* ≈*m s2*› **have** *s1'* ≈*m s2'* **unfolding** *s1'-def s2'-def*
by(*auto simp add*: *mbisim-def intro*: *tbisim-NoneI intro*!: *wset-thread-okI dest*: *wset-thread-okD*
split: *if-split-asm*)
ultimately have ∃*s2''*. *r2.mthr.silent-moves s2' s2''* ∧ *s1'* ≈*m s2''* ∧ *r1.final-threads s1'* ⊆
r2.final-threads s2'' ∧ *shr s2''* = *shr s2'* **by**(*rule insert*)

then obtain *s2''* **where** *reds*: *r2.mthr.silent-moves s2' s2''*
and *s1' ≈m s2''* **and** *fin*: $\bigwedge t$. *r1.final-thread s1' t* $\Longrightarrow$ *r2.final-thread s2'' t* **and** *shr s2'' = shr s2'* **by** *blast*
have *thr s2' t = None* **unfolding** *s2'-def* **by** *simp*
with ‹*r2.mthr.silent-moves s2' s2''*›
have *r2.mthr.silent-moves* (*locks s2'*, ((*thr s2'*)(*t* ↦ *the* (*thr s2 t*)), *shr s2'*), *wset s2'*, *interrupts s2'*)
(*locks s2''*, ((*thr s2''*)(*t* ↦ *the* (*thr s2 t*)), *shr s2''*), *wset s2''*, *interrupts s2''*)
by(*rule r2.τmRedT-add-thread-inv*)
also let *?s2''* = (*locks s2*, ((*thr s2''*)(*t* ↦ *the* (*thr s2 t*)), *shr s2*), *wset s2*, *interrupts s2*)
from ‹*shr s2'' = shr s2'*› ‹*s1' ≈m s2''*› ‹*s1 ≈m s2*›
have (*locks s2''*, ((*thr s2''*)(*t* ↦ *the* (*thr s2 t*)), *shr s2''*), *wset s2''*, *interrupts s2''*) = *?s2''*
unfolding *s2'-def s1'-def* **by**(*simp add*: *mbisim-def*)
also (*back-subst*) **from** ‹*s1 ≈m s2*› **have** *dom* (*thr s1*) = *dom* (*thr s2*) **by**(*rule mbisim-dom-eq*)
with ‹*r1.final-thread s1 t*› **have** *t* ∈ *dom* (*thr s2*) **by**(*auto simp add*: *r1.final-thread-def*)
then obtain *x2 ln* **where** *tst2*: *thr s2 t* = ⌊(*x2*, *ln*)⌋ **by** *auto*
hence (*locks s2'*, ((*thr s2'*)(*t* ↦ *the* (*thr s2 t*)), *shr s2'*), *wset s2'*, *interrupts s2'*) = *s2*
unfolding *s2'-def* **by**(*cases s2*)(*auto intro*!: *ext*)
also from ‹*s1 ≈m s2*› *tst2* **obtain** *x1*
where *tst1*: *thr s1 t* = ⌊(*x1*, *ln*)⌋
and *bisim*: *t* ⊢ (*x1*, *shr s1*) ≈ (*x2*, *shr s2*) **by**(*auto dest*: *mbisim-thrD2*)
from ‹*shr s2'' = shr s2'*› **have** *shr ?s2'' = shr s2* **by**(*simp add*: *s2'-def*)
from ‹*r1.final-thread s1 t*› *tst1*
have *final*: *final1 x1 ln = no-wait-locks wset s1 t = None* **by**(*auto simp add*: *r1.final-thread-def*)
with *final1-simulation*[*OF bisim*] ‹*shr ?s2'' = shr s2*› **obtain** *x2' m2'*
where *red*: *r2.silent-moves t* (*x2*, *shr ?s2''*) (*x2'*, *m2'*)
and *bisim'*: *t* ⊢ (*x1*, *shr s1*) ≈ (*x2'*, *m2'*) **and** *final2 x2'* **by** *auto*
from ‹*wset s1 t = None*› ‹*s1 ≈m s2*› **have** *wset s2 t = None* **by**(*simp add*: *mbisim-def*)
with *bisim r2.silent-moves-into-RedT-τ-inv*[*OF red*] *tst2* ‹*ln = no-wait-locks*›
have *r2.mthr.silent-moves ?s2''* (*redT-upd-ε ?s2'' t x2' m2'*) **unfolding** *s2'-def* **by** *auto*
also (*rtranclp-trans*)
from *bisim r2.red-rtrancl-τ-heapD-inv*[*OF red*] **have** *m2' = shr s2* **by** *auto*
hence *s1 ≈m* (*redT-upd-ε ?s2'' t x2' m2'*)
using ‹*s1' ≈m s2''*› ‹*s1 ≈m s2*› *tst1 tst2* ‹*shr ?s2'' = shr s2*› *bisim'* ‹*shr s2'' = shr s2'*› ‹*wset s2 t = None*›
unfolding *s1'-def s2'-def* **by**(*auto simp add*: *mbisim-def redT-updLns-def split*: *if-split-asm intro*: *tbisim-SomeI*)
moreover {
fix *t'*
assume *r1.final-thread s1 t'*
with *fin*[*of t'*] ‹*final2 x2'*› *tst2* ‹*ln = no-wait-locks*› ‹*wset s2 t = None*› ‹*s1' ≈m s2''*› ‹*s1 ≈m s2*›
have *r2.final-thread* (*redT-upd-ε ?s2'' t x2' m2'*) *t'* **unfolding** *s1'-def*
by(*fastforce split*: *if-split-asm simp add*: *r2.final-thread-def r1.final-thread-def redT-updLns-def finfun-Diag-const2 o-def mbisim-def*)
}
moreover have *shr* (*redT-upd-ε ?s2'' t x2' m2'*) = *shr s2* **using** ‹*m2' = shr s2*› **by** *simp*
ultimately show *?case* **by** *blast*
qed
qed

lemma *mfinal2-inv-simulation*:
s1 ≈m s2 $\Longrightarrow$ ∃ *s1'*. *r1.mthr.silent-moves s1 s1'* ∧ *s1' ≈m s2* ∧ *r2.final-threads s2* ⊆ *r1.final-threads s1'* ∧ *shr s1' = shr s1*

using *FWdelay-bisimulation-final-base.mfinal1-inv-simulation*[*OF FWdelay-bisimulation-final-base-flip*, **where** *bisim-wait=flip bisim-wait*]
by(*unfold flip-simps*)

lemma *mfinal1-simulation*:
assumes *s1* $\approx m$ *s2* **and** *r1.mfinal s1*
shows $\exists$ *s2'*. *r2.mthr.silent-moves s2 s2'* $\wedge$ *s1* $\approx m$ *s2'* $\wedge$ *r2.mfinal s2'* $\wedge$ *shr s2'* = *shr s2*
proof −
from *mfinal1-inv-simulation*[*OF* ‹*s1* $\approx m$ *s2*›]
obtain *s2'* **where** *1*: *r2.mthr.silent-moves s2 s2' s1* $\approx m$ *s2' shr s2'* = *shr s2*
and *fin*: $\bigwedge t$. *r1.final-thread s1 t* $\Longrightarrow$ *r2.final-thread s2' t* **by** *blast*
have *r2.mfinal s2'*
proof(*rule r2.mfinalI*)
fix *t x2 ln*
assume *thr s2' t* = ⌊(*x2*, *ln*)⌋
with ‹*s1* $\approx m$ *s2'*› **obtain** *x1* **where** *thr s1 t* = ⌊(*x1*, *ln*)⌋ *t* ⊢ (*x1*, *shr s1*) $\approx$ (*x2*, *shr s2'*)
by(*auto dest*: *mbisim-thrD2*)
from ‹*thr s1 t* = ⌊(*x1*, *ln*)⌋› ‹*r1.mfinal s1*› **have** *r1.final-thread s1 t*
by(*auto elim*!: *r1.mfinalE simp add*: *r1.final-thread-def*)
hence *r2.final-thread s2' t* **by**(*rule fin*)
thus *final2 x2* $\wedge$ *ln* = *no-wait-locks* $\wedge$ *wset s2' t* = *None*
using ‹*thr s2' t* = ⌊(*x2*, *ln*)⌋› **by**(*auto simp add*: *r2.final-thread-def*)
qed
with *1* **show** *?thesis* **by** *blast*
qed

lemma *mfinal2-simulation*:
⟦ *s1* $\approx m$ *s2*; *r2.mfinal s2* ⟧
$\Longrightarrow$ $\exists$ *s1'*. *r1.mthr.silent-moves s1 s1'* $\wedge$ *s1'* $\approx m$ *s2* $\wedge$ *r1.mfinal s1'* $\wedge$ *shr s1'* = *shr s1*
using *FWdelay-bisimulation-final-base.mfinal1-simulation*[*OF FWdelay-bisimulation-final-base-flip*, **where** *bisim-wait* = *flip bisim-wait*]
by(*unfold flip-simps*)

end

locale *FWdelay-bisimulation-obs* =
FWdelay-bisimulation-final-base - - - - - - - τ*move1* τ*move2*
for τ*move1* :: ('*l*,'*t*,'*x1*,'*m1*,'*w*, '*o*) τ*moves*
and τ*move2* :: ('*l*,'*t*,'*x2*,'*m2*,'*w*, '*o*) τ*moves* +
assumes *delay-bisimulation-obs-locale*: *delay-bisimulation-obs* (*r1 t*) (*r2 t*) (*bisim t*) (*ta-bisim bisim*) τ*move1* τ*move2*
and *bisim-inv-red-other*:
⟦ *t'* ⊢ (*x*, *m1*) $\approx$ (*xx*, *m2*); *t* ⊢ (*x1*, *m1*) $\approx$ (*x2*, *m2*);
r1.silent-moves t (*x1*, *m1*) (*x1'*, *m1*);
t ⊢ (*x1'*, *m1*) −1−*ta1*→ (*x1''*, *m1'*); ¬ τ*move1* (*x1'*, *m1*) *ta1* (*x1''*, *m1'*);
r2.silent-moves t (*x2*, *m2*) (*x2'*, *m2*);
t ⊢ (*x2'*, *m2*) −2−*ta2*→ (*x2''*, *m2'*); ¬ τ*move2* (*x2'*, *m2*) *ta2* (*x2''*, *m2'*);
t ⊢ (*x1''*, *m1'*) $\approx$ (*x2''*, *m2'*); *ta-bisim bisim ta1 ta2* ⟧
$\Longrightarrow$ *t'* ⊢ (*x*, *m1'*) $\approx$ (*xx*, *m2'*)
and *bisim-waitI*:
⟦ *t* ⊢ (*x1*, *m1*) $\approx$ (*x2*, *m2*); *r1.silent-moves t* (*x1*, *m1*) (*x1'*, *m1*);
t ⊢ (*x1'*, *m1*) −1−*ta1*→ (*x1''*, *m1'*); ¬ τ*move1* (*x1'*, *m1*) *ta1* (*x1''*, *m1'*);
r2.silent-moves t (*x2*, *m2*) (*x2'*, *m2*);
t ⊢ (*x2'*, *m2*) −2−*ta2*→ (*x2''*, *m2'*); ¬ τ*move2* (*x2'*, *m2*) *ta2* (*x2''*, *m2'*);

$t \vdash (x1'', m1') \approx (x2'', m2')$; *ta-bisim bisim ta1 ta2*;
Suspend $w \in$ *set* $\{\!|ta1|\!\}_w$; *Suspend* $w \in$ *set* $\{\!|ta2|\!\}_w$ $\rrbracket$
$\Longrightarrow x1'' \approx_w x2''$
and *simulation-Wakeup1*:
$[\![t \vdash (x1, m1) \approx (x2, m2)$; $x1 \approx_w x2$; $t \vdash (x1, m1) -1-ta1\rightarrow (x1', m1')$; *Notified* $\in$ *set* $\{\!|ta1|\!\}_w$
$\vee$ *WokenUp* $\in$ *set* $\{\!|ta1|\!\}_w$ $]\!]$
$\Longrightarrow \exists ta2\ x2'\ m2'.\ t \vdash (x2, m2) -2-ta2\rightarrow (x2', m2') \wedge t \vdash (x1', m1') \approx (x2', m2') \wedge$ *ta-bisim bisim ta1 ta2*
and *simulation-Wakeup2*:
$[\![t \vdash (x1, m1) \approx (x2, m2)$; $x1 \approx_w x2$; $t \vdash (x2, m2) -2-ta2\rightarrow (x2', m2')$; *Notified* $\in$ *set* $\{\!|ta2|\!\}_w$
$\vee$ *WokenUp* $\in$ *set* $\{\!|ta2|\!\}_w$ $]\!]$
$\Longrightarrow \exists ta1\ x1'\ m1'.\ t \vdash (x1, m1) -1-ta1\rightarrow (x1', m1') \wedge t \vdash (x1', m1') \approx (x2', m2') \wedge$ *ta-bisim bisim ta1 ta2*
and *ex-final1-conv-ex-final2*:
$(\exists x1.$ *final1* $x1) \longleftrightarrow (\exists x2.$ *final2* $x2)$

sublocale *FWdelay-bisimulation-obs* $<$
delay-bisimulation-obs r1 t r2 t bisim t ta-bisim bisim τ*move1* τ*move2* **for** *t*
by(*rule delay-bisimulation-obs-locale*)

context *FWdelay-bisimulation-obs* **begin**

lemma *FWdelay-bisimulation-obs-flip*:
FWdelay-bisimulation-obs final2 r2 final1 r1 $(\lambda t.$ *flip* (*bisim t*)) (*flip bisim-wait*) τ*move2* τ*move1*
apply(*rule FWdelay-bisimulation-obs.intro*)
apply(*rule FWdelay-bisimulation-final-base-flip*)
apply(*rule FWdelay-bisimulation-obs-axioms.intro*)
apply(*unfold flip-simps*)
apply(*rule delay-bisimulation-obs-axioms*)
apply(*erule* (*9*) *bisim-inv-red-other*)
apply(*erule* (*10*) *bisim-waitI*)
apply(*erule* (*3*) *simulation-Wakeup2*)
apply(*erule* (*3*) *simulation-Wakeup1*)
apply(*rule ex-final1-conv-ex-final2*[*symmetric*])
done

end

lemma *FWdelay-bisimulation-obs-flip-simps* [*flip-simps*]:
FWdelay-bisimulation-obs final2 r2 final1 r1 $(\lambda t.$ *flip* (*bisim t*)) (*flip bisim-wait*) τ*move2* τ*move1* $=$

FWdelay-bisimulation-obs final1 r1 final2 r2 bisim bisim-wait τ*move1* τ*move2*
by(*auto dest*: *FWdelay-bisimulation-obs.FWdelay-bisimulation-obs-flip simp only*: *flip-flip*)

context *FWdelay-bisimulation-obs* **begin**

lemma *mbisim-redT-upd*:
fixes *s1 t ta1 x1' m1' s2 ta2 x2' m2' ln*
assumes *s1'*: *redT-upd s1 t ta1 x1' m1' s1'*
and *s2'*: *redT-upd s2 t ta2 x2' m2' s2'*
and [*simp*]: *wset s1* $=$ *wset s2 locks s1* $=$ *locks s2*
and *wset*: *wset s1'* $=$ *wset s2'*
and *interrupts*: *interrupts s1'* $=$ *interrupts s2'*
and *fin1*: *finite* (*dom* (*thr s1*))

and *wsts*: *wset-thread-ok* (*wset s1*) (*thr s1*)
and *tst*: *thr s1 t* = ⌊(*x1*, *ln*)⌋
and *tst'*: *thr s2 t* = ⌊(*x2*, *ln*)⌋
and *aoe1*: *r1.actions-ok s1 t ta1*
and *aoe2*: *r2.actions-ok s2 t ta2*
and *tasim*: *ta-bisim bisim ta1 ta2*
and *bisim'*: *t* ⊢ (*x1'*, *m1'*) ≈ (*x2'*, *m2'*)
and *bisimw*: *wset s1' t* = *None* ∨ *x1'* ≈*w x2'*
and *τred1*: *r1.silent-moves t* (*x1''*, *shr s1*) (*x1*, *shr s1*)
and *red1*: *t* ⊢ (*x1*, *shr s1*) −*1*−*ta1*→ (*x1'*, *m1'*)
and *τred2*: *r2.silent-moves t* (*x2''*, *shr s2*) (*x2*, *shr s2*)
and *red2*: *t* ⊢ (*x2*, *shr s2*) −*2*−*ta2*→ (*x2'*, *m2'*)
and *bisim*: *t* ⊢ (*x1''*, *shr s1*) ≈ (*x2''*, *shr s2*)
and *τ1*: ¬ *τmove1* (*x1*, *shr s1*) *ta1* (*x1'*, *m1'*)
and *τ2*: ¬ *τmove2* (*x2*, *shr s2*) *ta2* (*x2'*, *m2'*)
and *tbisim*: ⋀*t'*. *t* ≠ *t'* ⟹ *tbisim* (*wset s1 t'* = *None*) *t'* (*thr s1 t'*) (*shr s1*) (*thr s2 t'*) (*shr s2*)
shows *s1'* ≈*m s2'*
proof(*rule mbisimI*)
from *fin1 s1'* **show** *finite* (*dom* (*thr s1'*))
by(*auto simp add*: *redT-updTs-finite-dom-inv*)
next
from *tasim s1' s2'* **show** *locks s1'* = *locks s2'*
by(*auto simp add*: *redT-updLs-def o-def ta-bisim-def*)
next
from *wset* **show** *wset s1'* = *wset s2'* **.**
next
from *interrupts* **show** *interrupts s1'* = *interrupts s2'* **.**
next
from *wsts s1' s2' wset* **show** *wset-thread-ok* (*wset s1'*) (*thr s1'*)
by(*fastforce intro*!: *wset-thread-okI split*: *if-split-asm dest*: *redT-updTs-None wset-thread-okD redT-updWs-None-implies-Non*
next
fix *T*
assume *thr s1' T* = *None*
moreover with *tst s1'* **have** [*simp*]: *t* ≠ *T* **by** *auto*
from *tbisim*[*OF this*] **have** (*thr s1 T* = *None*) = (*thr s2 T* = *None*)
by(*auto simp add*: *tbisim-def*)
hence (*redT-updTs* (*thr s1*) ⦃*ta1*⦄*t T* = *None*) = (*redT-updTs* (*thr s2*) ⦃*ta2*⦄*t T* = *None*)
using *tasim* **by** −(*rule redT-updTs-nta-bisim-inv*, *simp-all add*: *ta-bisim-def*)
ultimately show *thr s2' T* = *None* **using** *s2' s1'* **by**(*auto split*: *if-split-asm*)
next
fix *T X1 LN*
assume *tsT*: *thr s1' T* = ⌊(*X1*, *LN*)⌋
show ∃*x2*. *thr s2' T* = ⌊(*x2*, *LN*)⌋ ∧ *T* ⊢ (*X1*, *shr s1'*) ≈ (*x2*, *shr s2'*) ∧ (*wset s2' T* = *None* ∨
X1 ≈*w x2*)
proof(*cases thr s1 T*)
case *None*
with *tst* **have** *t* ≠ *T* **by** *auto*
with *tbisim*[*OF this*] *None* **have** *tsT'*: *thr s2 T* = *None* **by**(*simp add*: *tbisim-def*)
from *None* ‹*t* ≠ *T*› *tsT aoe1 s1'* **obtain** *M1*
where *ntset*: *NewThread T X1 M1* ∈ *set* ⦃*ta1*⦄*t* **and** [*simp*]: *LN* = *no-wait-locks*
by(*auto dest*!: *redT-updTs-new-thread*)
from *ntset* **obtain** *tas1 tas1'* **where** ⦃*ta1*⦄*t* = *tas1* @ *NewThread T X1 M1* # *tas1'*
by(*auto simp add*: *in-set-conv-decomp*)
with *tasim* **obtain** *tas2 X2 M2 tas2'* **where** ⦃*ta2*⦄*t* = *tas2* @ *NewThread T X2 M2* # *tas2'*

```
      length tas2 = length tas2 length tas1' = length tas2' and Bisim: T ⊢ (X1, M1) ≈ (X2, M2)
      by(auto simp add: list-all2-append1 list-all2-Cons1 ta-bisim-def)
    hence ntset': NewThread T X2 M2 ∈ set {|ta2|}_t by auto
    with tsT' ‹t ≠ T› aoe2 s2' have thr s2' T = ⌊(X2, no-wait-locks)⌋
      by(auto intro: redT-updTs-new-thread-ts)
    moreover from ntset' red2 have m2' = M2 by(auto dest: r2.new-thread-memory)
    moreover from ntset red1 have m1' = M1
      by(auto dest: r1.new-thread-memory)
    moreover from wsts None have wset s1 T = None by(rule wset-thread-okD)
    ultimately show ?thesis using Bisim ‹t ≠ T› s1' s2'
      by(auto simp add: redT-updWs-None-implies-None)
  next
    case (Some a)
    show ?thesis
    proof(cases t = T)
      case True
      with tst tsT s1' have [simp]: X1 = x1' LN = redT-updLns (locks s1) t ln {|ta1|}_l by(auto)
      show ?thesis using True bisim' bisimw tasim tst tst' s1' s2' wset
        by(auto simp add: redT-updLns-def ta-bisim-def)
    next
      case False
      with Some aoe1 tsT s1' have thr s1 T = ⌊(X1, LN)⌋ by(auto dest: redT-updTs-Some)
      with tbisim[OF False] obtain X2
        where tsT': thr s2 T = ⌊(X2, LN)⌋ and Bisim: T ⊢ (X1, shr s1) ≈ (X2, shr s2)
        and bisimw: wset s1 T = None ∨ X1 ≈w X2 by(auto simp add: tbisim-def)
      with aoe2 False s2' have tsT': thr s2' T = ⌊(X2, LN)⌋ by(auto simp add: redT-updTs-Some)
      moreover from Bisim bisim τred1 red1 τ1 τred2 red2 τ2 bisim' tasim
      have T ⊢ (X1, m1') ≈ (X2, m2') by(rule bisim-inv-red-other)
      ultimately show ?thesis using False bisimw s1' s2'
        by(auto simp add: redT-updWs-None-implies-None)
    qed
  qed
qed

theorem mbisim-simulation1:
  assumes mbisim: mbisim s1 s2 and ¬ mτmove1 s1 tl1 s1' r1.redT s1 tl1 s1'
  shows ∃ s2' s2'' tl2. r2.mthr.silent-moves s2 s2' ∧ r2.redT s2' tl2 s2'' ∧
                        ¬ mτmove2 s2' tl2 s2'' ∧ mbisim s1' s2'' ∧ mta-bisim tl1 tl2
proof −
  from assms obtain t ta1 where tl1 [simp]: tl1 = (t, ta1) and redT: s1 −1−t▷ta1→ s1'
    and mτ: ¬ mτmove1 s1 (t, ta1) s1' by(cases tl1) fastforce
  obtain ls1 ts1 m1 ws1 is1 where [simp]: s1 = (ls1, (ts1, m1), ws1, is1) by(cases s1) fastforce
  obtain ls1' ts1' m1' ws1' is1' where [simp]: s1' = (ls1', (ts1', m1'), ws1', is1') by(cases s1')
fastforce
  obtain ls2 ts2 m2 ws2 is2 where [simp]: s2 = (ls2, (ts2, m2), ws2, is2) by(cases s2) fastforce
  from mbisim have [simp]: ls2 = ls1 ws2 = ws1 is2 = is1 finite (dom ts1) by(auto simp add:
mbisim-def)
  from redT show ?thesis
  proof cases
    case (redT-normal x1 x1' M1')
    hence red: t ⊢ (x1, m1) −1−ta1→ (x1', M1')
      and tst: ts1 t = ⌊(x1, no-wait-locks)⌋
      and aoe: r1.actions-ok s1 t ta1
      and s1': redT-upd s1 t ta1 x1' M1' s1' by auto
```

from *mbisim tst* **obtain** *x2* **where** *tst'*: *ts2 t* = ⌊(*x2*, *no-wait-locks*)⌋
and *bisim*: *t* ⊢ (*x1*, *m1*) ≈ (*x2*, *m2*) **by**(*auto dest*: *mbisim-thrD1*)
from *mτ* **have** *τ*: ¬ *τmove1* (*x1*, *m1*) *ta1* (*x1'*, *M1'*)
proof(*rule contrapos-nn*)
assume *τ*: *τmove1* (*x1*, *m1*) *ta1* (*x1'*, *M1'*)
moreover hence [*simp*]: *ta1* = *ε* **by**(*rule r1.silent-tl*)
moreover have [*simp*]: *M1'* = *m1* **by**(*rule r1.τmove-heap*[*OF red τ*, *symmetric*])
ultimately show *mτmove1 s1* (*t*, *ta1*) *s1'* **using** *s1' tst s1'*
by(*auto simp add*: *redT-updLs-def o-def intro*: *r1.mτmove.intros elim*: *rtrancl3p-cases*)
qed
show *?thesis*
proof(*cases ws1 t*)
case *None*
note *wst* = *this*
from *simulation1*[*OF bisim red τ*] **obtain** *x2' M2' x2'' M2'' ta2*
where *red21*: *r2.silent-moves t* (*x2*, *m2*) (*x2'*, *M2'*)
and *red22*: *t* ⊢ (*x2'*, *M2'*) −*2*−*ta2*→ (*x2''*, *M2''*) **and** *τ2*: ¬ *τmove2* (*x2'*, *M2'*) *ta2* (*x2''*, *M2''*)
and *bisim'*: *t* ⊢ (*x1'*, *M1'*) ≈ (*x2''*, *M2''*)
and *tasim*: *ta-bisim bisim ta1 ta2* **by** *auto*
let *?s2'* = *redT-upd-ε s2 t x2' M2'*
let *?S2'* = *activate-cond-actions2 s1 ?s2'* ⦃*ta2*⦄$_c$
let *?s2''* = (*redT-updLs* (*locks ?S2'*) *t* ⦃*ta2*⦄$_l$, ((*redT-updTs* (*thr ?S2'*) ⦃*ta2*⦄$_t$)(*t* ↦ (*x2''*, *redT-updLns* (*locks ?S2'*) *t* (*snd* (*the* (*thr ?S2' t*))) ⦃*ta2*⦄$_l$)), *M2''*), *wset s1'*, *interrupts s1'*)
from *red21 tst' wst bisim* **have** *τmRed2 s2 ?s2'*
by −(*rule r2.silent-moves-into-RedT-τ-inv*, *auto*)
moreover from *red21 bisim* **have** [*simp*]: *M2'* = *m2* **by**(*auto dest*: *r2.red-rtrancl-τ-heapD-inv*)
from *tasim* **have** [*simp*]: ⦃ *ta1* ⦄$_l$ = ⦃ *ta2* ⦄$_l$ ⦃ *ta1* ⦄$_w$ = ⦃ *ta2* ⦄$_w$ ⦃ *ta1* ⦄$_c$ = ⦃ *ta2* ⦄$_c$ ⦃ *ta1* ⦄$_i$ = ⦃ *ta2* ⦄$_i$
and *nta*: *list-all2* (*nta-bisim bisim*) ⦃ *ta1* ⦄$_t$ ⦃ *ta2* ⦄$_t$ **by**(*auto simp add*: *ta-bisim-def*)
from *mbisim* **have** *tbisim*: ⋀*t*. *tbisim* (*ws1 t* = *None*) *t* (*ts1 t*) *m1* (*ts2 t*) *m2* **by**(*simp add*: *mbisim-def*)
hence *tbisim'*: ⋀*t'*. *t'* ≠ *t* ⟹ *tbisim* (*ws1 t'* = *None*) *t'* (*ts1 t'*) *m1* (*thr ?s2' t'*) *m2* **by**(*auto*)
from *aoe* **have** *cao1*: *r1.cond-action-oks* (*ls1*, (*ts1*, *m1*), *ws1*, *is1*) *t* ⦃*ta2*⦄$_c$ **by** *auto*
from *tst'* **have** *thr ?s2' t* = ⌊(*x2'*, *no-wait-locks*)⌋ **by**(*auto simp add*: *redT-updLns-def o-def finfun-Diag-const2*)
from *cond-actions-oks-bisim-ex-τ2-inv*[*OF tbisim'*, *OF - tst this cao1*]
have *red21'*: *τmRed2 ?s2' ?S2'* **and** *tbisim''*: ⋀*t'*. *t'* ≠ *t* ⟹ *tbisim* (*ws1 t'* = *None*) *t'* (*ts1 t'*) *m1* (*thr ?S2' t'*) *m2*
and *cao2*: *r2.cond-action-oks ?S2' t* ⦃*ta2*⦄$_c$ **and** *tst''*: *thr ?S2' t* = ⌊(*x2'*, *no-wait-locks*)⌋
by(*auto simp del*: *fun-upd-apply*)
note *red21'* **also** (*rtranclp-trans*)
from *tbisim'' tst'' tst* **have** ∀*t'*. *ts1 t'* = *None* ⟷ *thr ?S2' t'* = *None* **by**(*force simp add*: *tbisim-def*)
from *aoe thread-oks-bisim-inv*[*OF this nta*] **have** *thread-oks* (*thr ?S2'*) ⦃*ta2*⦄$_t$ **by** *simp*
with *cao2 aoe* **have** *aoe'*: *r2.actions-ok ?S2' t ta2* **by** *auto*
with *red22 tst'' s1'* **have** *?S2'* −*2*−*t*▷*ta2*→ *?s2''*
by −(*rule r2.redT.redT-normal*, *auto*)
moreover
from *τ2* **have** ¬ *mτmove2 ?S2'* (*t*, *ta2*) *?s2''*
proof(*rule contrapos-nn*)
assume *mτ*: *mτmove2 ?S2'* (*t*, *ta2*) *?s2''*
thus *τmove2* (*x2'*, *M2'*) *ta2* (*x2''*, *M2''*) **using** *tst'' tst'*
by *cases auto*

```
    qed
    moreover
    {
     note s1'
     moreover have redT-upd ?S2' t ta2 x2'' M2'' ?s2'' using s1' by auto
     moreover have wset s1 = wset ?S2' locks s1 = locks ?S2' by simp-all
     moreover have wset s1' = wset ?s2'' by simp
     moreover have interrupts s1' = interrupts ?s2'' by simp
     moreover have finite (dom (thr s1)) by simp
     moreover from mbisim have wset-thread-ok (wset s1) (thr s1) by(simp add: mbisim-def)
     moreover from tst have thr s1 t = ⌊(x1, no-wait-locks)⌋ by simp
     moreover note tst'' aoe aoe' tasim bisim'
     moreover have wset s1' t = None ∨ x1' ≈w x2''
     proof(cases wset s1' t)
      case None thus ?thesis ..
     next
      case (Some w)
      with wst s1' obtain w' where Suspend1: Suspend w' ∈ set {|ta1|}w
       by(auto dest: redT-updWs-None-SomeD)
      with tasim have Suspend2: Suspend w' ∈ set {|ta2|}w by(simp add: ta-bisim-def)
      from bisim-waitI[OF bisim rtranclp.rtrancl-refl red τ - - - bisim' tasim Suspend1 this, of x2']
red21 red22 τ2
      have x1' ≈w x2'' by auto
      thus ?thesis ..
     qed
     moreover note rtranclp.rtrancl-refl
     moreover from red have t ⊢ (x1, shr s1) −1−ta1→ (x1', M1') by simp
     moreover from red21 have r2.silent-moves t (x2, shr ?S2') (x2', shr ?S2') by simp
     moreover from red22 have t ⊢ (x2', shr ?S2') −2−ta2→ (x2'', M2'') by simp
     moreover from bisim have t ⊢ (x1, shr s1) ≈ (x2, shr ?S2') by simp
     moreover from τ have ¬ τmove1 (x1, shr s1) ta1 (x1', M1') by simp
     moreover from τ2 have ¬ τmove2 (x2', shr ?S2') ta2 (x2'', M2'') by simp
     moreover from tbisim''
     have ⋀t'. t ≠ t' ⟹ tbisim (wset s1 t' = None) t' (thr s1 t') (shr s1) (thr ?S2' t') (shr ?S2')
      by simp
     ultimately have mbisim s1' ?s2'' by(rule mbisim-redT-upd)
     }
    ultimately show ?thesis using tasim unfolding tl1 s1' by fastforce
   next
    case (Some w)
    with mbisim tst tst' have x1 ≈w x2
     by(auto dest: mbisim-thrD1)
    from aoe Some have wakeup: Notified ∈ set {|ta1|}w ∨ WokenUp ∈ set {|ta1|}w
     by(auto simp add: wset-actions-ok-def split: if-split-asm)
    from simulation-Wakeup1[OF bisim ‹x1 ≈w x2› red this]
    obtain ta2 x2' m2' where red2: t ⊢ (x2, m2) −2−ta2→ (x2', m2')
     and bisim': t ⊢ (x1', M1') ≈ (x2', m2')
     and tasim: ta1 ∼m ta2 by auto

    let ?S2' = activate-cond-actions2 s1 s2 {|ta2|}c

     let ?s2' = (redT-updLs (locks ?S2') t {|ta2|}l, ((redT-updTs (thr ?S2') {|ta2|}t)(t ↦ (x2',
redT-updLns (locks ?S2') t (snd (the (thr ?S2' t))) {|ta2|}l)), m2'), wset s1', interrupts s1')
```

from *tasim* **have** [*simp*]: $\{\!|\ ta1\ |\!\}_l = \{\!|\ ta2\ |\!\}_l\ \{\!|\ ta1\ |\!\}_w = \{\!|\ ta2\ |\!\}_w\ \{\!|\ ta1\ |\!\}_c = \{\!|\ ta2\ |\!\}_c\ \{\!|\ ta1\ |\!\}_i = \{\!|\ ta2\ |\!\}_i$
and *nta*: *list-all2* (*nta-bisim bisim*) $\{\!|\ ta1\ |\!\}_t\ \{\!|\ ta2\ |\!\}_t$ **by**(*auto simp add*: *ta-bisim-def*)
from *mbisim* **have** *tbisim*: $\bigwedge t.$ *tbisim* (*ws1 t* = *None*) *t* (*ts1 t*) *m1* (*ts2 t*) *m2* **by**(*simp add*: *mbisim-def*)
hence *tbisim'*: $\bigwedge t'.\ t' \neq t \Longrightarrow$ *tbisim* (*ws1 t'* = *None*) *t'* (*ts1 t'*) *m1* (*thr s2 t'*) *m2* **by**(*auto*)
from *aoe* **have** *cao1*: *r1.cond-action-oks* (*ls1*, (*ts1*, *m1*), *ws1*, *is1*) *t* $\{\!|ta2|\!\}_c$ **by** *auto*
from *tst'* **have** *thr s2 t* = $\lfloor$(*x2*, *no-wait-locks*)$\rfloor$
by(*auto simp add*: *redT-updLns-def o-def finfun-Diag-const2*)
from *cond-actions-oks-bisim-ex-τ2-inv*[*OF tbisim'*, *OF - tst this cao1*]
have *red21'*: *τmRed2 s2 ?S2'* **and** *tbisim''*: $\bigwedge t'.\ t' \neq t \Longrightarrow$ *tbisim* (*ws1 t'* = *None*) *t'* (*ts1 t'*) *m1* (*thr ?S2' t'*) *m2*
and *cao2*: *r2.cond-action-oks ?S2' t* $\{\!|ta2|\!\}_c$ **and** *tst''*: *thr ?S2' t* = $\lfloor$(*x2*, *no-wait-locks*)$\rfloor$
by(*auto simp del*: *fun-upd-apply*)
note *red21'* **moreover**
from *tbisim'' tst'' tst* **have** $\forall t'.$ *ts1 t'* = *None* $\longleftrightarrow$ *thr ?S2' t'* = *None* **by**(*force simp add*: *tbisim-def*)
from *aoe thread-oks-bisim-inv*[*OF this nta*] **have** *thread-oks* (*thr ?S2'*) $\{\!|ta2|\!\}_t$ **by** *simp*
with *cao2 aoe* **have** *aoe'*: *r2.actions-ok ?S2' t ta2* **by** *auto*
with *red2 tst'' s1' tasim* **have** *?S2' −2−t▷ta2→ ?s2'*
by −(*rule r2.redT-normal*, *auto simp add*: *ta-bisim-def*)
moreover from *wakeup tasim*
have *τ2*: ¬ *τmove2* (*x2*, *m2*) *ta2* (*x2'*, *m2'*) **by**(*auto dest*: *r2.silent-tl*)
hence ¬ *mτmove2 ?S2'* (*t*, *ta2*) *?s2'*
proof(*rule contrapos-nn*)
assume *mτ*: *mτmove2 ?S2'* (*t*, *ta2*) *?s2'*
thus *τmove2* (*x2*, *m2*) *ta2* (*x2'*, *m2'*) **using** *tst'' tst'*
by *cases auto*
qed
moreover **{**
note *s1'*
moreover have *redT-upd ?S2' t ta2 x2' m2' ?s2'* **using** *s1' tasim* **by**(*auto simp add*: *ta-bisim-def*)
moreover have *wset s1* = *wset ?S2' locks s1* = *locks ?S2'* **by** *simp-all*
moreover have *wset s1'* = *wset ?s2'* **by** *simp*
moreover have *interrupts s1'* = *interrupts ?s2'* **by** *simp*
moreover have *finite* (*dom* (*thr s1*)) **by** *simp*
moreover from *mbisim* **have** *wset-thread-ok* (*wset s1*) (*thr s1*) **by**(*rule mbisim-wset-thread-ok1*)
moreover from *tst* **have** *thr s1 t* = $\lfloor$(*x1*, *no-wait-locks*)$\rfloor$ **by** *simp*
moreover from *tst''* **have** *thr ?S2' t* = $\lfloor$(*x2*, *no-wait-locks*)$\rfloor$ **by** *simp*
moreover note *aoe aoe' tasim bisim'*
moreover have *wset s1' t* = *None* ∨ *x1'* ≈*w x2'*
proof(*cases wset s1' t*)
case *None* **thus** *?thesis* **..**
next
case (*Some w'*)
with *redT-updWs-WokenUp-SuspendD*[*OF - wakeup*, *of t wset s1 wset s1' w'*] *s1'*
obtain *w'* **where** *Suspend1*: *Suspend w'* ∈ *set* $\{\!|ta1|\!\}_w$ **by**(*auto*)
with *tasim* **have** *Suspend2*: *Suspend w'* ∈ *set* $\{\!|ta2|\!\}_w$ **by**(*simp add*: *ta-bisim-def*)
with *bisim rtranclp.rtrancl-refl red τ rtranclp.rtrancl-refl red2 τ2 bisim' tasim Suspend1*
have *x1'* ≈*w x2'* **by**(*rule bisim-waitI*)
thus *?thesis* **..**
qed
moreover note *rtranclp.rtrancl-refl*

```
      moreover from red have t ⊢ (x1, shr s1) −1−ta1→ (x1', M1') by simp
      moreover note rtranclp.rtrancl-refl
      moreover from red2 have t ⊢ (x2, shr ?S2') −2−ta2→ (x2', m2') by simp
      moreover from bisim have t ⊢ (x1, shr s1) ≈ (x2, shr ?S2') by simp
      moreover from τ have ¬ τmove1 (x1, shr s1) ta1 (x1', M1') by simp
      moreover from τ2 have ¬ τmove2 (x2, shr ?S2') ta2 (x2', m2') by simp
       moreover from tbisim'' have ⋀t'. t ≠ t' ⟹ tbisim (wset s1 t' = None) t' (thr s1 t') (shr
s1) (thr ?S2' t') (shr ?S2') by simp
      ultimately have s1' ≈m ?s2' by(rule mbisim-redT-upd) }
    moreover from tasim have tl1 ∼T (t, ta2) by simp
    ultimately show ?thesis unfolding s1' by blast
  qed
 next
  case (redT-acquire x1 n ln)
  hence [simp]: ta1 = (K$ [], [], [], [], [], convert-RA ln)
    and tst: thr s1 t = ⌊(x1, ln)⌋ and wst: ¬ waiting (wset s1 t)
    and maa: may-acquire-all (locks s1) t ln and ln: 0 < ln $ n
    and s1': s1' = (acquire-all ls1 t ln, (ts1(t ↦ (x1, no-wait-locks)), m1), ws1, is1) by auto
  from tst mbisim obtain x2 where tst': ts2 t = ⌊(x2, ln)⌋
    and bisim: t ⊢ (x1, m1) ≈ (x2, m2) by(auto dest: mbisim-thrD1)
  let ?s2' = (acquire-all ls1 t ln, (ts2(t ↦ (x2, no-wait-locks)), m2), ws1, is1)
  from tst' wst maa ln have s2 −2−t▷(K$ [], [], [], [], [], convert-RA ln)→ ?s2'
    by−(rule r2.redT.redT-acquire, auto)
  moreover from tst' ln have ¬ mτmove2 s2 (t, (K$ [], [], [], [], [], convert-RA ln)) ?s2'
    by(auto simp add: acquire-all-def fun-eq-iff elim!: r2.mτmove.cases)
  moreover have mbisim s1' ?s2'
  proof(rule mbisimI)
    from s1' show locks s1' = locks ?s2' by auto
  next
    from s1' show wset s1' = wset ?s2' by auto
  next
    from s1' show interrupts s1' = interrupts ?s2' by auto
  next
    fix t' assume thr s1' t' = None
    with s1' have thr s1 t' = None by(auto split: if-split-asm)
    with mbisim-thrNone-eq[OF mbisim] have ts2 t' = None by simp
    with tst' show thr ?s2' t' = None by auto
  next
    fix t' X1 LN
    assume ts't: thr s1' t' = ⌊(X1, LN)⌋
     show ∃x2. thr ?s2' t' = ⌊(x2, LN)⌋ ∧ t' ⊢ (X1, shr s1') ≈ (x2, shr ?s2') ∧ (wset ?s2' t' =
None ∨ X1 ≈w x2)
    proof(cases t' = t)
      case True
      with s1' tst ts't have [simp]: X1 = x1 LN = no-wait-locks by simp-all
      with mbisim-thrD1[OF mbisim tst] bisim tst tst' True s1' wst show ?thesis by(auto)
    next
      case False
      with ts't s1' have ts1 t' = ⌊(X1, LN)⌋ by auto
      with mbisim obtain X2 where ts2 t' = ⌊(X2, LN)⌋ t' ⊢ (X1, m1) ≈ (X2, m2) wset ?s2' t'
= None ∨ X1 ≈w X2
        by(auto dest: mbisim-thrD1)
      with False s1' show ?thesis by auto
    qed
```

```
    next
      from s1' show finite (dom (thr s1')) by auto
    next
      from mbisim-wset-thread-ok1[OF mbisim]
      show wset-thread-ok (wset s1') (thr s1') using s1' by(auto intro: wset-thread-ok-upd)
    qed
    moreover have (t, K$ [], [], [], [], [], convert-RA ln) ∼T (t, K$ [], [], [], [], [], convert-RA ln)
      by(simp add: ta-bisim-def)
    ultimately show ?thesis by fastforce
  qed
qed

theorem mbisim-simulation2:
  ⟦ mbisim s1 s2; r2.redT s2 tl2 s2'; ¬ mτmove2 s2 tl2 s2' ⟧
  ⟹ ∃ s1' s1'' tl1. r1.mthr.silent-moves s1 s1' ∧ r1.redT s1' tl1 s1'' ∧ ¬ mτmove1 s1' tl1 s1'' ∧
                mbisim s1'' s2' ∧ mta-bisim tl1 tl2
using FWdelay-bisimulation-obs.mbisim-simulation1[OF FWdelay-bisimulation-obs-flip]
unfolding flip-simps .

end

locale FWdelay-bisimulation-diverge =
  FWdelay-bisimulation-obs - - - - - - - τmove1 τmove2
  for τmove1 :: ('l,'t,'x1,'m1,'w,'o) τmoves
  and τmove2 :: ('l,'t,'x2,'m2,'w,'o) τmoves +
  assumes delay-bisimulation-diverge-locale: delay-bisimulation-diverge (r1 t) (r2 t) (bisim t) (ta-bisim
bisim) τmove1 τmove2

sublocale FWdelay-bisimulation-diverge <
  delay-bisimulation-diverge r1 t r2 t bisim t ta-bisim bisim τmove1 τmove2 for t
by(rule delay-bisimulation-diverge-locale)

context FWdelay-bisimulation-diverge begin

lemma FWdelay-bisimulation-diverge-flip:
  FWdelay-bisimulation-diverge final2 r2 final1 r1 (λt. flip (bisim t)) (flip bisim-wait) τmove2 τmove1
apply(rule FWdelay-bisimulation-diverge.intro)
 apply(rule FWdelay-bisimulation-obs-flip)
apply(rule FWdelay-bisimulation-diverge-axioms.intro)
apply(unfold flip-simps)
apply(rule delay-bisimulation-diverge-axioms)
done

end

lemma FWdelay-bisimulation-diverge-flip-simps [flip-simps]:
  FWdelay-bisimulation-diverge final2 r2 final1 r1 (λt. flip (bisim t)) (flip bisim-wait) τmove2 τmove1
=
   FWdelay-bisimulation-diverge final1 r1 final2 r2 bisim bisim-wait τmove1 τmove2
by(auto dest: FWdelay-bisimulation-diverge.FWdelay-bisimulation-diverge-flip simp only: flip-flip)

context FWdelay-bisimulation-diverge begin

lemma bisim-inv1:
```

assumes *bisim*: $t \vdash s1 \approx s2$
and *red*: $t \vdash s1\ -1-ta1\rightarrow\ s1'$
obtains *s2'* **where** $t \vdash s1' \approx s2'$
proof(*atomize-elim*)
show $\exists s2'.\ t \vdash s1' \approx s2'$
proof(*cases* τ*move1 s1 ta1 s1'*)
case *True*
with *red* **have** *r1.silent-move t s1 s1'* **by** *auto*
from *simulation-silent1*[*OF bisim this*]
show *?thesis* **by** *auto*
next
case *False*
from *simulation1*[*OF bisim red False*] **show** *?thesis* **by** *auto*
qed
qed

lemma *bisim-inv2*:
assumes $t \vdash s1 \approx s2\ t \vdash s2\ -2-ta2\rightarrow\ s2'$
obtains *s1'* **where** $t \vdash s1' \approx s2'$
using *assms FWdelay-bisimulation-diverge.bisim-inv1*[*OF FWdelay-bisimulation-diverge-flip*]
unfolding *flip-simps* **by** *blast*

lemma *bisim-inv*: *bisim-inv*
by(*blast intro*!: *bisim-invI elim*: *bisim-inv1 bisim-inv2*)

lemma *bisim-inv-*τ*s1*:
assumes $t \vdash s1 \approx s2$ **and** *r1.silent-moves t s1 s1'*
obtains *s2'* **where** $t \vdash s1' \approx s2'$
using *assms* **by**(*rule bisim-inv-*τ*s1-inv*[*OF bisim-inv*])

lemma *bisim-inv-*τ*s2*:
assumes $t \vdash s1 \approx s2$ **and** *r2.silent-moves t s2 s2'*
obtains *s1'* **where** $t \vdash s1' \approx s2'$
using *assms* **by**(*rule bisim-inv-*τ*s2-inv*[*OF bisim-inv*])

lemma *red1-rtrancl-*τ*-into-RedT-*τ:
assumes *r1.silent-moves t* (*x1*, *shr s1*) (*x1'*, *m1'*) $t \vdash$ (*x1*, *shr s1*) $\approx$ (*x2*, *m2*)
and *thr s1 t* $=$ $\lfloor$(*x1*, *no-wait-locks*)$\rfloor$ *wset s1 t* $=$ *None*
shows τ*mRed1 s1* (*redT-upd-*ε *s1 t x1' m1'*)
using *assms* **by**(*blast intro*: *r1.silent-moves-into-RedT-*τ*-inv*)

lemma *red2-rtrancl-*τ*-into-RedT-*τ:
assumes *r2.silent-moves t* (*x2*, *shr s2*) (*x2'*, *m2'*)
and $t \vdash$ (*x1*, *m1*) $\approx$ (*x2*, *shr s2*) *thr s2 t* $=$ $\lfloor$(*x2*, *no-wait-locks*)$\rfloor$ *wset s2 t* $=$ *None*
shows τ*mRed2 s2* (*redT-upd-*ε *s2 t x2' m2'*)
using *assms* **by**(*blast intro*: *r2.silent-moves-into-RedT-*τ*-inv*)

lemma *red1-rtrancl-*τ*-heapD*:
$[\![$ *r1.silent-moves t s1 s1'*; $t \vdash s1 \approx s2\ ]\!] \Longrightarrow$ *snd s1'* $=$ *snd s1*
by(*blast intro*: *r1.red-rtrancl-*τ*-heapD-inv*)

lemma *red2-rtrancl-*τ*-heapD*:
$[\![$ *r2.silent-moves t s2 s2'*; $t \vdash s1 \approx s2\ ]\!] \Longrightarrow$ *snd s2'* $=$ *snd s2*
by(*blast intro*: *r2.red-rtrancl-*τ*-heapD-inv*)

lemma *mbisim-simulation-silent1*:
assumes *mτ'*: *r1.mthr.silent-move s1 s1'* **and** *mbisim*: *s1* ≈*m s2*
shows ∃ *s2'*. *r2.mthr.silent-moves s2 s2'* ∧ *s1'* ≈*m s2'*
proof −
from *mτ'* **obtain** *tl1* **where** *mτ*: *mτmove1 s1 tl1 s1' r1.redT s1 tl1 s1'* **by** *auto*
obtain *ls1 ts1 m1 ws1 is1* **where** [*simp*]: *s1* = (*ls1*, (*ts1*, *m1*), *ws1*, *is1*) **by**(*cases s1*) *fastforce*
obtain *ls1' ts1' m1' ws1' is1'* **where** [*simp*]: *s1'* = (*ls1'*, (*ts1'*, *m1'*), *ws1'*, *is1'*) **by**(*cases s1'*) *fastforce*
obtain *ls2 ts2 m2 ws2 is2* **where** [*simp*]: *s2* = (*ls2*, (*ts2*, *m2*), *ws2*, *is2*) **by**(*cases s2*) *fastforce*
from *mτ* **obtain** *t* **where** *tl1* = (*t*, *ε*) **by**(*auto elim*!: *r1.mτmove.cases dest*: *r1.silent-tl*)
with *mτ* **have** *mτ*: *mτmove1 s1* (*t*, *ε*) *s1'* **and** *redT1*: *s1* −1−*t*▷*ε*→ *s1'* **by** *simp-all*
from *mτ* **obtain** *x x' ln'* **where** *tst*: *ts1 t* = ⌊(*x*, *no-wait-locks*)⌋
and *ts't*: *ts1' t* = ⌊(*x'*, *ln'*)⌋ **and** *τ*: *τmove1* (*x*, *m1*) *ε* (*x'*, *m1'*)
by(*fastforce elim*: *r1.mτmove.cases*)
from *mbisim* **have** [*simp*]: *ls2* = *ls1 ws2* = *ws1 is2* = *is1 finite* (*dom ts1*) **by**(*auto simp add*: *mbisim-def*)
from *redT1* **show** *?thesis*
proof *cases*
case (*redT-normal x1 x1' M'*)
with *tst ts't* **have** [*simp*]: *x* = *x1 x'* = *x1'*
and *red*: *t* ⊢ (*x1*, *m1*) −1−*ε*→ (*x1'*, *M'*)
and *tst*: *thr s1 t* = ⌊(*x1*, *no-wait-locks*)⌋
and *wst*: *wset s1 t* = *None*
and *s1'*: *redT-upd s1 t ε x1' M' s1'* **by**(*auto*)
from *s1' tst* **have** [*simp*]: *ls1'* = *ls1 ws1'* = *ws1 is1'* = *is1 M'* = *m1' ts1'* = *ts1*(*t* ↦ (*x1'*, *no-wait-locks*))
by(*auto simp add*: *redT-updLs-def redT-updLns-def o-def redT-updWs-def elim*!: *rtrancl3p-cases*)
from *mbisim tst* **obtain** *x2* **where** *tst'*: *ts2 t* = ⌊(*x2*, *no-wait-locks*)⌋
and *bisim*: *t* ⊢ (*x1*, *m1*) ≈ (*x2*, *m2*) **by**(*auto dest*: *mbisim-thrD1*)
from *r1.τmove-heap*[*OF red*] *τ* **have** [*simp*]: *m1* = *M'* **by** *simp*
from *red τ* **have** *r1.silent-move t* (*x1*, *m1*) (*x1'*, *M'*) **by** *auto*
from *simulation-silent1*[*OF bisim this*]
obtain *x2' m2'* **where** *red*: *r2.silent-moves t* (*x2*, *m2*) (*x2'*, *m2'*)
and *bisim'*: *t* ⊢ (*x1'*, *m1*) ≈ (*x2'*, *m2'*) **by** *auto*
from *red bisim* **have** [*simp*]: *m2'* = *m2*
by(*auto dest*: *red2-rtrancl-τ-heapD*)
let *?s2'* = *redT-upd-ε s2 t x2' m2'*
from *red tst' wst bisim* **have** *τmRed2 s2 ?s2'*
by −(*rule red2-rtrancl-τ-into-RedT-τ*, *auto*)
moreover have *mbisim s1' ?s2'*
proof(*rule mbisimI*)
show *locks s1'* = *locks ?s2' wset s1'* = *wset ?s2' interrupts s1'* = *interrupts ?s2'* **by** *auto*
next
fix *t'*
assume *thr s1' t'* = *None*
hence *ts1 t'* = *None* **by**(*auto split*: *if-split-asm*)
with *mbisim-thrNone-eq*[*OF mbisim*] **have** *ts2 t'* = *None* **by** *simp*
with *tst'* **show** *thr ?s2' t'* = *None* **by** *auto*
next
fix *t' X1 LN*
assume *ts't'*: *thr s1' t'* = ⌊(*X1*, *LN*)⌋
show ∃ *x2*. *thr ?s2' t'* = ⌊(*x2*, *LN*)⌋ ∧ *t'* ⊢ (*X1*, *shr s1'*) ≈ (*x2*, *shr ?s2'*) ∧ (*wset ?s2' t'* = *None* ∨ *X1* ≈*w x2*)

```
      proof(cases t' = t)
        case True
        note this[simp]
        with s1' tst ts't' have [simp]: X1 = x1' LN = no-wait-locks
          by(simp-all)(auto simp add: redT-updLns-def o-def finfun-Diag-const2)
       with bisim' tst' wst show ?thesis by(auto simp add: redT-updLns-def o-def finfun-Diag-const2)
      next
        case False
        with ts't' have ts1 t' = ⌊(X1, LN)⌋ by auto
         with mbisim obtain X2 where ts2 t' = ⌊(X2, LN)⌋ t' ⊢ (X1, m1) ≈ (X2, m2) ws1 t' =
None ∨ X1 ≈w X2
          by(auto dest: mbisim-thrD1)
        with False show ?thesis by auto
      qed
    next
      show finite (dom (thr s1')) by simp
    next
      from mbisim-wset-thread-ok1[OF mbisim]
      show wset-thread-ok (wset s1') (thr s1') by(auto intro: wset-thread-ok-upd)
    qed
    ultimately show ?thesis by(auto)
  next
    case redT-acquire
    with tst have False by auto
    thus ?thesis ..
  qed
qed

lemma mbisim-simulation-silent2:
  ⟦ mbisim s1 s2; r2.mthr.silent-move s2 s2' ⟧
  ⟹ ∃ s1'. r1.mthr.silent-moves s1 s1' ∧ mbisim s1' s2'
using FWdelay-bisimulation-diverge.mbisim-simulation-silent1[OF FWdelay-bisimulation-diverge-flip]
unfolding flip-simps .

lemma mbisim-simulation1':
  assumes mbisim: mbisim s1 s2 and ¬ mτmove1 s1 tl1 s1' r1.redT s1 tl1 s1'
  shows ∃ s2' s2'' tl2. r2.mthr.silent-moves s2 s2' ∧ r2.redT s2' tl2 s2'' ∧
                        ¬ mτmove2 s2' tl2 s2'' ∧ mbisim s1' s2'' ∧ mta-bisim tl1 tl2
using mbisim-simulation1 assms .

lemma mbisim-simulation2':
  ⟦ mbisim s1 s2; r2.redT s2 tl2 s2'; ¬ mτmove2 s2 tl2 s2' ⟧
  ⟹ ∃ s1' s1'' tl1. r1.mthr.silent-moves s1 s1' ∧ r1.redT s1' tl1 s1'' ∧ ¬ mτmove1 s1' tl1 s1'' ∧
                    mbisim s1'' s2' ∧ mta-bisim tl1 tl2
using FWdelay-bisimulation-diverge.mbisim-simulation1'[OF FWdelay-bisimulation-diverge-flip]
unfolding flip-simps .

lemma mτdiverge-simulation1:
  assumes s1 ≈m s2
  and r1.mthr.τdiverge s1
  shows r2.mthr.τdiverge s2
proof −
  from ‹s1 ≈m s2› have finite (dom (thr s1))
    by(rule mbisim-finite1)+
```

from *r1.τdiverge-τmredTD*[*OF* ‹*r1.mthr.τdiverge s1*› *this*]
obtain *t x* **where** *thr s1 t* = ⌊(*x*, *no-wait-locks*)⌋ *wset s1 t* = *None r1.τdiverge t* (*x*, *shr s1*) **by** *blast*
from ‹*s1* ≈*m s2*› ‹*thr s1 t* = ⌊(*x*, *no-wait-locks*)⌋› **obtain** *x′*
where *thr s2 t* = ⌊(*x′*, *no-wait-locks*)⌋ *t* ⊢ (*x*, *shr s1*) ≈ (*x′*, *shr s2*)
by(*auto dest*: *mbisim-thrD1*)
from ‹*s1* ≈*m s2*› ‹*wset s1 t* = *None*› **have** *wset s2 t* = *None* **by**(*simp add*: *mbisim-def*)
from ‹*t* ⊢ (*x*, *shr s1*) ≈ (*x′*, *shr s2*)› ‹*r1.τdiverge t* (*x*, *shr s1*)›
have *r2.τdiverge t* (*x′*, *shr s2*) **by**(*simp add*: *τdiverge-bisim-inv*)
thus *?thesis* **using** ‹*thr s2 t* = ⌊(*x′*, *no-wait-locks*)⌋› ‹*wset s2 t* = *None*›
by(*rule r2.τdiverge-into-τmredT*)
qed

lemma *τdiverge-mbisim-inv*:
s1 ≈*m s2* ⟹ *r1.mthr.τdiverge s1* ⟷ *r2.mthr.τdiverge s2*
apply(*rule iffI*)
apply(*erule* (*1*) *mτdiverge-simulation1*)
by(*rule FWdelay-bisimulation-diverge.mτdiverge-simulation1*[*OF FWdelay-bisimulation-diverge-flip*, *unfolded flip-simps*])

lemma *mbisim-delay-bisimulation*:
delay-bisimulation-diverge r1.redT r2.redT mbisim mta-bisim mτmove1 mτmove2
apply(*unfold-locales*)
apply(*rule mbisim-simulation1 mbisim-simulation2 mbisim-simulation-silent1 mbisim-simulation-silent2 τdiverge-mbisim-inv*|*assumption*)+
done

theorem *mdelay-bisimulation-final-base*:
delay-bisimulation-final-base r1.redT r2.redT mbisim mτmove1 mτmove2 r1.mfinal r2.mfinal
apply(*unfold-locales*)
apply(*blast dest*: *mfinal1-simulation mfinal2-simulation*)+
done

end

sublocale *FWdelay-bisimulation-diverge* < *mthr*: *delay-bisimulation-diverge r1.redT r2.redT mbisim mta-bisim mτmove1 mτmove2*
by(*rule mbisim-delay-bisimulation*)

sublocale *FWdelay-bisimulation-diverge* <
mthr: *delay-bisimulation-final-base r1.redT r2.redT mbisim mta-bisim mτmove1 mτmove2 r1.mfinal r2.mfinal*
by(*rule mdelay-bisimulation-final-base*)

context *FWdelay-bisimulation-diverge* **begin**

lemma *mthr-delay-bisimulation-diverge-final*:
delay-bisimulation-diverge-final r1.redT r2.redT mbisim mta-bisim mτmove1 mτmove2 r1.mfinal r2.mfinal
by(*unfold-locales*)

end

sublocale *FWdelay-bisimulation-diverge* <
mthr: *delay-bisimulation-diverge-final r1.redT r2.redT mbisim mta-bisim mτmove1 mτmove2 r1.mfinal*

r2.mfinal
by(*rule mthr-delay-bisimulation-diverge-final*)

1.18.3 Strong bisimulation as corollary

locale *FWbisimulation* = *FWbisimulation-base* - - - *r2 convert-RA bisim* λ*x1 x2*. *True* +
r1: *multithreaded final1 r1 convert-RA* +
r2: *multithreaded final2 r2 convert-RA*
for *r2* :: ('*l*,'*t*,'*x2*,'*m2*,'*w*,'*o*) *semantics* (‹- ⊢ - −2−-→ -› [*50*,*0*,*0*,*50*] *80*)
and *convert-RA* :: '*l released-locks* ⇒ '*o list*
and *bisim* :: '*t* ⇒ ('*x1* × '*m1*, '*x2* × '*m2*) *bisim* (‹- ⊢ -/ ≈ -› [*50*, *50*, *50*] *60*) +
assumes *bisimulation-locale*: *bisimulation* (*r1 t*) (*r2 t*) (*bisim t*) (*ta-bisim bisim*)
and *bisim-final*: *t* ⊢ (*x1*, *m1*) ≈ (*x2*, *m2*) ⟹ *final1 x1* ⟷ *final2 x2*
and *bisim-inv-red-other*:
⟦ *t'* ⊢ (*x*, *m1*) ≈ (*xx*, *m2*); *t* ⊢ (*x1*, *m1*) ≈ (*x2*, *m2*);
t ⊢ (*x1*, *m1*) −*1*−*ta1*→ (*x1'*, *m1'*); *t* ⊢ (*x2*, *m2*) −*2*−*ta2*→ (*x2'*, *m2'*);
t ⊢ (*x1'*, *m1'*) ≈ (*x2'*, *m2'*); *ta-bisim bisim ta1 ta2* ⟧
⟹ *t'* ⊢ (*x*, *m1'*) ≈ (*xx*, *m2'*)
and *ex-final1-conv-ex-final2*:
(∃ *x1*. *final1 x1*) ⟷ (∃ *x2*. *final2 x2*)

sublocale *FWbisimulation* < *bisim?*: *bisimulation r1 t r2 t bisim t ta-bisim bisim* **for** *t*
by(*rule bisimulation-locale*)

sublocale *FWbisimulation* < *bisim-diverge?*:
FWdelay-bisimulation-diverge final1 r1 final2 r2 convert-RA bisim λ*x1 x2*. *True* λ*s ta s'*. *False* λ*s ta s'*. *False*
proof −
interpret *biw*: *bisimulation-into-delay r1 t r2 t bisim t ta-bisim bisim* λ*s ta s'*. *False* λ*s ta s'*. *False*
for *t*
by(*unfold-locales*) *simp*
show *FWdelay-bisimulation-diverge final1 r1 final2 r2 bisim* (λ*x1 x2*. *True*) (λ*s ta s'*. *False*) (λ*s ta s'*. *False*)
proof(*unfold-locales*)
fix *t' x m1 xx m2 x1 x2 t x1' ta1 x1'' m1' x2' ta2 x2'' m2'*
assume *bisim*: *t'* ⊢ (*x*, *m1*) ≈ (*xx*, *m2*) **and** *bisim12*: *t* ⊢ (*x1*, *m1*) ≈ (*x2*, *m2*)
and *τ1*: *τtrsys.silent-moves* (*r1 t*) (λ*s ta s'*. *False*) (*x1*, *m1*) (*x1'*, *m1*)
and *red1*: *t* ⊢ (*x1'*, *m1*) −*1*−*ta1*→ (*x1''*, *m1'*)
and *τ2*: *τtrsys.silent-moves* (*r2 t*) (λ*s ta s'*. *False*) (*x2*, *m2*) (*x2'*, *m2*)
and *red2*: *t* ⊢ (*x2'*, *m2*) −*2*−*ta2*→ (*x2''*, *m2'*)
and *bisim12'*: *t* ⊢ (*x1''*, *m1'*) ≈ (*x2''*, *m2'*) **and** *tasim*: *ta1* ∼*m ta2*
from *τ1 τ2* **have** [*simp*]: *x1'* = *x1 x2'* = *x2* **by**(*simp-all add*: *rtranclp-False τmoves-False*)
from *bisim12 bisim-inv-red-other*[*OF bisim - red1 red2 bisim12' tasim*]
show *t'* ⊢ (*x*, *m1'*) ≈ (*xx*, *m2'*) **by** *simp*
next
fix *t x1 m1 x2 m2 ta1 x1' m1'*
assume *t* ⊢ (*x1*, *m1*) ≈ (*x2*, *m2*) *t* ⊢ (*x1*, *m1*) −*1*−*ta1*→ (*x1'*, *m1'*)
from *simulation1*[*OF this*]
show ∃ *ta2 x2' m2'*. *t* ⊢ (*x2*, *m2*) −*2*−*ta2*→ (*x2'*, *m2'*) ∧ *t* ⊢ (*x1'*, *m1'*) ≈ (*x2'*, *m2'*) ∧ *ta1* ∼*m ta2*
by *auto*
next
fix *t x1 m1 x2 m2 ta2 x2' m2'*
assume *t* ⊢ (*x1*, *m1*) ≈ (*x2*, *m2*) *t* ⊢ (*x2*, *m2*) −*2*−*ta2*→ (*x2'*, *m2'*)

from *simulation2*[*OF this*]
show ∃ *ta1 x1′ m1′*. *t* ⊢ (*x1*, *m1*) −*1*−*ta1*→ (*x1′*, *m1′*) ∧ *t* ⊢ (*x1′*, *m1′*) ≈ (*x2′*, *m2′*) ∧ *ta1* ∼*m ta2*
by *auto*
next
show (∃ *x1*. *final1 x1*) ⟷ (∃ *x2*. *final2 x2*) **by**(*rule ex-final1-conv-ex-final2*)
qed(*fastforce simp add*: *bisim-final*)+
qed

context *FWbisimulation* **begin**

lemma *FWbisimulation-flip*: *FWbisimulation final2 r2 final1 r1* (λ*t*. *flip* (*bisim t*))
apply(*rule FWbisimulation.intro*)
apply(*rule r2.multithreaded-axioms*)
apply(*rule r1.multithreaded-axioms*)
apply(*rule FWbisimulation-axioms.intro*)
apply(*unfold flip-simps*)
apply(*rule bisimulation-axioms*)
apply(*erule bisim-final*[*symmetric*])
apply(*erule* (*5*) *bisim-inv-red-other*)
apply(*rule ex-final1-conv-ex-final2*[*symmetric*])
done

end

lemma *FWbisimulation-flip-simps* [*flip-simps*]:
FWbisimulation final2 r2 final1 r1 (λ*t*. *flip* (*bisim t*)) = *FWbisimulation final1 r1 final2 r2 bisim*
by(*auto dest*: *FWbisimulation.FWbisimulation-flip simp only*: *flip-flip*)

context *FWbisimulation* **begin**

The notation for mbisim is lost because *bisim-wait* is instantiated to λ*x1 x2*. *True*. This reintroduces the syntax, but it does not work for output mode. This would require a new abbreviation.

notation *mbisim* (‹- ≈*m* -› [*50*, *50*] *60*)

theorem *mbisim-bisimulation*:
bisimulation r1.redT r2.redT mbisim mta-bisim
proof
fix *s1 s2 tta1 s1′*
assume *mbisim*: *s1* ≈*m s2* **and** *r1.redT s1 tta1 s1′*
from *mthr.simulation1*[*OF this*]
show ∃ *s2′ tta2*. *r2.redT s2 tta2 s2′* ∧ *s1′* ≈*m s2′* ∧ *tta1* ∼*T tta2*
by(*auto simp add*: τ*moves-False m*τ*move-False*)
next
fix *s2 s1 tta2 s2′*
assume *s1* ≈*m s2* **and** *r2.redT s2 tta2 s2′*
from *mthr.simulation2*[*OF this*]
show ∃ *s1′ tta1*. *r1.redT s1 tta1 s1′* ∧ *s1′* ≈*m s2′* ∧ *tta1* ∼*T tta2*
by(*auto simp add*: τ*moves-False m*τ*move-False*)
qed

lemma *mbisim-wset-eq*:
s1 ≈*m s2* ⟹ *wset s1* = *wset s2*

by(*simp add*: *mbisim-def*)

lemma *mbisim-mfinal*:
s1 ≈*m s2* ⟹ *r1.mfinal s1* ⟷ *r2.mfinal s2*
apply(*auto intro*!: *r2.mfinalI r1.mfinalI dest*: *mbisim-thrD2 mbisim-thrD1 bisim-final elim*: *r1.mfinalE r2.mfinalE*)
apply(*frule* (*1*) *mbisim-thrD2*, *drule mbisim-wset-eq*, *auto elim*: *r1.mfinalE*)
apply(*frule* (*1*) *mbisim-thrD1*, *drule mbisim-wset-eq*, *auto elim*: *r2.mfinalE*)
done

end

sublocale *FWbisimulation* < *mthr*: *bisimulation r1.redT r2.redT mbisim mta-bisim*
by(*rule mbisim-bisimulation*)

sublocale *FWbisimulation* < *mthr*: *bisimulation-final r1.redT r2.redT mbisim mta-bisim r1.mfinal r2.mfinal*
by(*unfold-locales*)(*rule mbisim-mfinal*)

end

1.19 Preservation of deadlock across bisimulations

theory *FWBisimDeadlock*
imports
FWBisimulation
FWDeadlock
begin

context *FWdelay-bisimulation-obs* **begin**

lemma *actions-ok1-ex-actions-ok2*:
assumes *r1.actions-ok s1 t ta1*
and *ta1* ∼*m ta2*
obtains *s2* **where** *r2.actions-ok s2 t ta2*
proof −
let *?s2* = (*locks s1*, (λ*t*. *map-option* (λ(*x1*, *ln*). (*SOME x2*. *if final1 x1 then final2 x2 else* ¬ *final2 x2*, *ln*)) (*thr s1 t*), *undefined*), *wset s1*, *interrupts s1*)
from ‹*ta1* ∼*m ta2*› **have** ⦃*ta1*⦄*c* = ⦃*ta2*⦄*c* **by**(*simp add*: *ta-bisim-def*)
with ‹*r1.actions-ok s1 t ta1*› **have** *cao1*: *r1.cond-action-oks s1 t* ⦃*ta2*⦄*c* **by** *auto*
have *r2.cond-action-oks ?s2 t* ⦃*ta2*⦄*c* **unfolding** *r2.cond-action-oks-conv-set*
proof
fix *ct*
assume *ct* ∈ *set* ⦃*ta2*⦄*c*
with *cao1* **have** *r1.cond-action-ok s1 t ct*
unfolding *r1.cond-action-oks-conv-set* **by** *auto*
thus *r2.cond-action-ok ?s2 t ct* **using** *ex-final1-conv-ex-final2*
by(*cases ct*)(*fastforce intro*: *someI-ex*[**where** *P*=*final2*])+
qed
hence *r2.actions-ok ?s2 t ta2*
using *assms* **by**(*auto simp add*: *ta-bisim-def split del*: *if-split elim*: *rev-iffD1*[*OF - thread-oks-bisim-inv*])
thus *thesis* **by**(*rule that*)
qed

lemma *actions-ok2-ex-actions-ok1*:
assumes *r2.actions-ok s2 t ta2*
and *ta1* $\sim m$ *ta2*
obtains *s1* **where** *r1.actions-ok s1 t ta1*
using *FWdelay-bisimulation-obs.actions-ok1-ex-actions-ok2*[*OF FWdelay-bisimulation-obs-flip*] *assms*
unfolding *flip-simps* **.**

lemma *ex-actions-ok1-conv-ex-actions-ok2*:
ta1 $\sim m$ *ta2* $\Longrightarrow$ $(\exists\, s1.\ r1.actions\text{-}ok\ s1\ t\ ta1) \longleftrightarrow (\exists\, s2.\ r2.actions\text{-}ok\ s2\ t\ ta2)$
by(*metis actions-ok1-ex-actions-ok2 actions-ok2-ex-actions-ok1*)

end

context *FWdelay-bisimulation-diverge* **begin**

lemma *no-*τ*Move1-*τ*s-to-no-*τ*Move2*:
fixes *no-*τ*moves1 no-*τ*moves2*
defines *no-*τ*moves1* $\equiv \lambda s1\ t.\ wset\ s1\ t = None \wedge (\exists\, x.\ thr\ s1\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor \wedge (\forall\, x'\ m'.$
$\neg\ r1.silent\text{-}move\ t\ (x,\ shr\ s1)\ (x',\ m')))$
defines *no-*τ*moves2* $\equiv \lambda s2\ t.\ wset\ s2\ t = None \wedge (\exists\, x.\ thr\ s2\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor \wedge (\forall\, x'\ m'.$
$\neg\ r2.silent\text{-}move\ t\ (x,\ shr\ s2)\ (x',\ m')))$
assumes *mbisim*: *s1* $\approx m$ $(ls2,\ (ts2,\ m2),\ ws2,\ is2)$

shows $\exists\, ts2'.\ r2.mthr.silent\text{-}moves\ (ls2,\ (ts2,\ m2),\ ws2,\ is2)\ (ls2,\ (ts2',\ m2),\ ws2,\ is2) \wedge$
$(\forall\, t.\ no\text{-}\tau moves1\ s1\ t \longrightarrow no\text{-}\tau moves2\ (ls2,\ (ts2',\ m2),\ ws2,\ is2)\ t) \wedge s1 \approx m\ (ls2,\ (ts2',$
$m2),\ ws2,\ is2)$
proof −
from *mbisim* **have** *finite* (*dom* (*thr s1*)) **by**(*simp add*: *mbisim-def*)
hence *finite* $\{t.\ no\text{-}\tau moves1\ s1\ t\}$ **unfolding** *no-*τ*moves1-def*
by−(*rule finite-subset*, *auto*)
thus *?thesis* **using** ‹*s1* $\approx m$ $(ls2,\ (ts2,\ m2),\ ws2,\ is2)$›
proof(*induct A*≡$\{t.\ no\text{-}\tau moves1\ s1\ t\}$ *arbitrary*: *s1 ts2 rule*: *finite-induct*)
case *empty*
from ‹$\{\} = \{t.\ no\text{-}\tau moves1\ s1\ t\}$›[*symmetric*] **have** *no-*τ*moves1 s1* = $(\lambda t.\ False)$
by(*auto intro*: *ext*)
thus *?case* **using** ‹*s1* $\approx m$ $(ls2,\ (ts2,\ m2),\ ws2,\ is2)$› **by** *auto*
next
case (*insert t A*)
note *mbisim* = ‹*s1* $\approx m$ $(ls2,\ (ts2,\ m2),\ ws2,\ is2)$›
from ‹*insert t A* = $\{t.\ no\text{-}\tau moves1\ s1\ t\}$›
have *no-*τ*moves1 s1 t* **by** *auto*
then obtain *x1* **where** *ts1t*: *thr s1 t* = $\lfloor(x1,\ no\text{-}wait\text{-}locks)\rfloor$
and *ws1t*: *wset s1 t* = *None*
and $\tau 1$: $\bigwedge x1m1'.\ \neg\ r1.silent\text{-}move\ t\ (x1,\ shr\ s1)\ x1m1'$
by(*auto simp add*: *no-*τ*moves1-def*)

from *ts1t mbisim* **obtain** *x2* **where** *ts2t*: *ts2 t* = $\lfloor(x2,\ no\text{-}wait\text{-}locks)\rfloor$
and $t \vdash (x1,\ shr\ s1) \approx (x2,\ m2)$ **by**(*auto dest*: *mbisim-thrD1*)
from *mbisim ws1t* **have** *ws2 t* = *None* **by**(*simp add*: *mbisim-def*)

let *?s1* = $(locks\ s1,\ ((thr\ s1)(t := None),\ shr\ s1),\ wset\ s1,\ interrupts\ s1)$
let *?s2* = $(ls2,\ (ts2(t := None),\ m2),\ ws2,\ is2)$
from ‹*insert t A* = $\{t.\ no\text{-}\tau moves1\ s1\ t\}$› ‹$t \notin A$›

have *A*: *A* = {*t*. *no-τmoves1 ?s1 t*} **by**(*auto simp add*: *no-τmoves1-def*)
have *?s1* ≈*m ?s2*
proof(*rule mbisimI*)
from *mbisim*
show *finite* (*dom* (*thr ?s1*)) *locks ?s1* = *locks ?s2 wset ?s1* = *wset ?s2 interrupts ?s1* = *interrupts ?s2*
by(*simp-all add*: *mbisim-def*)
next
from *mbisim-wset-thread-ok1*[*OF mbisim*] *ws1t* **show** *wset-thread-ok* (*wset ?s1*) (*thr ?s1*)
by(*auto intro*!: *wset-thread-okI dest*: *wset-thread-okD split*: *if-split-asm*)
next
fix *t′*
assume *thr ?s1 t′* = *None*
with *mbisim-thrNone-eq*[*OF mbisim*, *of t′*]
show *thr ?s2 t′* = *None* **by** *auto*
next
fix *t′ x1 ln*
assume *thr ?s1 t′* = ⌊(*x1*, *ln*)⌋
hence *thr s1 t′* = ⌊(*x1*, *ln*)⌋ *t′* ≠ *t* **by**(*auto split*: *if-split-asm*)
with *mbisim-thrD1*[*OF mbisim* ‹*thr s1 t′* = ⌊(*x1*, *ln*)⌋›] *mbisim*
show ∃ *x2*. *thr ?s2 t′* = ⌊(*x2*, *ln*)⌋ ∧ *t′* ⊢ (*x1*, *shr ?s1*) ≈ (*x2*, *shr ?s2*) ∧ (*wset ?s2 t′* = *None* ∨ *x1* ≈*w x2*)
by(*auto simp add*: *mbisim-def*)
qed
with *A* **have** ∃ *ts2′*. *r2.mthr.silent-moves ?s2* (*ls2*, (*ts2′*, *m2*), *ws2*, *is2*) ∧ (∀ *t*. *no-τmoves1 ?s1 t* ⟶ *no-τmoves2* (*ls2*, (*ts2′*, *m2*), *ws2*, *is2*) *t*) ∧ *?s1* ≈*m* (*ls2*, (*ts2′*, *m2*), *ws2*, *is2*) **by**(*rule insert*)
then obtain *ts2′* **where** *r2.mthr.silent-moves ?s2* (*ls2*, (*ts2′*, *m2*), *ws2*, *is2*)
and *no-τ*: ⋀*t*. *no-τmoves1 ?s1 t* ⟹ *no-τmoves2* (*ls2*, (*ts2′*, *m2*), *ws2*, *is2*) *t*
and *?s1* ≈*m* (*ls2*, (*ts2′*, *m2*), *ws2*, *is2*) **by** *auto*
let *?s2′* = (*ls2*, (*ts2′*(*t* ↦ (*x2*, *no-wait-locks*)), *m2*), *ws2*, *is2*)
from *ts2t* **have** *ts2*(*t* ↦ (*x2*, *no-wait-locks*)) = *ts2* **by**(*auto intro*: *ext*)
with *r2.τmRedT-add-thread-inv*[*OF* ‹*r2.mthr.silent-moves ?s2* (*ls2*, (*ts2′*, *m2*), *ws2*, *is2*)›, *of t* (*x2*, *no-wait-locks*)]
have *r2.mthr.silent-moves* (*ls2*, (*ts2*, *m2*), *ws2*, *is2*) *?s2′* **by** *simp*
from *no-τmove1-τs-to-no-τmove2*[*OF* ‹*t* ⊢ (*x1*, *shr s1*) ≈ (*x2*, *m2*)› *τ1*]
obtain *x2′ m2′* **where** *r2.silent-moves t* (*x2*, *m2*) (*x2′*, *m2′*)
and ⋀*x2″ m2″*. ¬ *r2.silent-move t* (*x2′*, *m2′*) (*x2″*, *m2″*)
and *t* ⊢ (*x1*, *shr s1*) ≈ (*x2′*, *m2′*) **by** *auto*
let *?s2″* = (*ls2*, (*ts2′*(*t* ↦ (*x2′*, *no-wait-locks*)), *m2′*), *ws2*, *is2*)
from *red2-rtrancl-τ-heapD*[*OF* ‹*r2.silent-moves t* (*x2*, *m2*) (*x2′*, *m2′*)› ‹*t* ⊢ (*x1*, *shr s1*) ≈ (*x2*, *m2*)›]
have *m2′* = *m2* **by** *simp*
with ‹*r2.silent-moves t* (*x2*, *m2*) (*x2′*, *m2′*)› **have** *r2.silent-moves t* (*x2*, *shr ?s2′*) (*x2′*, *m2*) **by** *simp*
hence *r2.mthr.silent-moves ?s2′* (*redT-upd-ε ?s2′ t x2′ m2*)
by(*rule red2-rtrancl-τ-into-RedT-τ*)(*auto simp add*: ‹*ws2 t* = *None*› *intro*: ‹*t* ⊢ (*x1*, *shr s1*) ≈ (*x2*, *m2*)›)
also have *redT-upd-ε ?s2′ t x2′ m2* = *?s2″* **using** ‹*m2′* = *m2*›
by(*auto simp add*: *fun-eq-iff redT-updLns-def finfun-Diag-const2 o-def*)
finally (*back-subst*) **have** *r2.mthr.silent-moves* (*ls2*, (*ts2*, *m2*), *ws2*, *is2*) *?s2″*
using ‹*r2.mthr.silent-moves* (*ls2*, (*ts2*, *m2*), *ws2*, *is2*) *?s2′*› **by**−(*rule rtranclp-trans*)
moreover {
fix *t′*
assume *no-τ1*: *no-τmoves1 s1 t′*

```
      have no-τmoves2 ?s2'' t'
      proof(cases t' = t)
        case True thus ?thesis
          using ‹ws2 t = None› ‹⋀x2'' m2''. ¬ r2.silent-move t (x2', m2') (x2'', m2'')› by(simp add:
no-τmoves2-def)
      next
        case False
        with no-τ1 have no-τmoves1 ?s1 t' by(simp add: no-τmoves1-def)
        hence no-τmoves2 (ls2, (ts2', m2), ws2, is2) t'
          by(rule ‹no-τmoves1 ?s1 t' ⟹ no-τmoves2 (ls2, (ts2', m2), ws2, is2) t'›)
        with False ‹m2' = m2› show ?thesis by(simp add: no-τmoves2-def)
      qed }
    moreover have s1 ≈m ?s2''
    proof(rule mbisimI)
      from mbisim
     show finite (dom (thr s1)) locks s1 = locks ?s2'' wset s1 = wset ?s2'' interrupts s1 = interrupts
?s2''
        by(simp-all add: mbisim-def)
    next
      from mbisim show wset-thread-ok (wset s1) (thr s1) by(rule mbisim-wset-thread-ok1)
    next
      fix t'
      assume thr s1 t' = None
      hence thr ?s1 t' = None t' ≠ t using ts1t by auto
      with mbisim-thrNone-eq[OF ‹?s1 ≈m (ls2, (ts2', m2), ws2, is2)›, of t']
      show thr ?s2'' t' = None by simp
    next
      fix t' x1' ln'
      assume thr s1 t' = ⌊(x1', ln')⌋
      show ∃x2. thr ?s2'' t' = ⌊(x2, ln')⌋ ∧ t' ⊢ (x1', shr s1) ≈ (x2, shr ?s2'') ∧ (wset ?s2'' t' =
None ∨ x1' ≈w x2)
      proof(cases t = t')
        case True
        with ‹thr s1 t' = ⌊(x1', ln')⌋› ts1t ‹t ⊢ (x1, shr s1) ≈ (x2', m2')› ‹m2' = m2› ‹ws2 t = None›
        show ?thesis by auto
      next
        case False
        with mbisim-thrD1[OF ‹?s1 ≈m (ls2, (ts2', m2), ws2, is2)›, of t' x1' ln'] ‹thr s1 t' = ⌊(x1',
ln')⌋› ‹m2' = m2› mbisim
        show ?thesis by(auto simp add: mbisim-def)
      qed
    qed
    ultimately show ?case unfolding ‹m2' = m2› by blast
  qed
qed

lemma no-τMove2-τs-to-no-τMove1:
  fixes no-τmoves1 no-τmoves2
  defines no-τmoves1 ≡ λs1 t. wset s1 t = None ∧ (∃x. thr s1 t = ⌊(x, no-wait-locks)⌋ ∧ (∀x' m'.
¬ r1.silent-move t (x, shr s1) (x', m')))
  defines no-τmoves2 ≡ λs2 t. wset s2 t = None ∧ (∃x. thr s2 t = ⌊(x, no-wait-locks)⌋ ∧ (∀x' m'.
¬ r2.silent-move t (x, shr s2) (x', m')))
  assumes (ls1, (ts1, m1), ws1, is1) ≈m s2
```

shows $\exists ts1'.\ r1.mthr.silent\text{-}moves\ (ls1,\ (ts1,\ m1),\ ws1,\ is1)\ (ls1,\ (ts1',\ m1),\ ws1,\ is1) \wedge$
$(\forall t.\ no\text{-}\tau moves2\ s2\ t \longrightarrow no\text{-}\tau moves1\ (ls1,\ (ts1',\ m1),\ ws1,\ is1)\ t) \wedge (ls1,\ (ts1',\ m1),\ ws1,\ is1) \approx m\ s2$
using *assms FWdelay-bisimulation-diverge.no-τMove1-τs-to-no-τMove2*[*OF FWdelay-bisimulation-diverge-flip*]
unfolding *flip-simps* **by** *blast*

lemma *deadlock-mbisim-not-final-thread-pres*:
assumes *dead*: $t \in r1.deadlocked\ s1 \vee r1.deadlock\ s1$
and *nfin*: $r1.not\text{-}final\text{-}thread\ s1\ t$
and *fin*: $r1.final\text{-}thread\ s1\ t \Longrightarrow r2.final\text{-}thread\ s2\ t$
and *mbisim*: $s1 \approx m\ s2$
shows $r2.not\text{-}final\text{-}thread\ s2\ t$
proof −
from *nfin* **obtain** *x1 ln* **where** $thr\ s1\ t = \lfloor (x1,\ ln) \rfloor$ **by** *cases auto*
with *mbisim* **obtain** *x2* **where** $thr\ s2\ t = \lfloor (x2,\ ln) \rfloor\ t \vdash (x1,\ shr\ s1) \approx (x2,\ shr\ s2)\ wset\ s1\ t = None \vee x1 \approx w\ x2$
by(*auto dest*: *mbisim-thrD1*)
show $r2.not\text{-}final\text{-}thread\ s2\ t$
proof(*cases wset s1 t = None ∧ ln = no-wait-locks*)
case *False*
with ‹$r1.not\text{-}final\text{-}thread\ s1\ t$› ‹$thr\ s1\ t = \lfloor (x1,\ ln) \rfloor$› ‹$thr\ s2\ t = \lfloor (x2,\ ln) \rfloor$› *mbisim*
show *?thesis* **by** *cases*(*auto simp add*: *mbisim-def r2.not-final-thread-iff*)
next
case *True*
with ‹$r1.not\text{-}final\text{-}thread\ s1\ t$› ‹$thr\ s1\ t = \lfloor (x1,\ ln) \rfloor$› **have** ¬ *final1 x1* **by**(*cases*) *auto*
have ¬ *final2 x2*
proof
assume *final2 x2*
with *final2-simulation*[*OF* ‹$t \vdash (x1,\ shr\ s1) \approx (x2,\ shr\ s2)$›]
obtain $x1'\ m1'$ **where** $r1.silent\text{-}moves\ t\ (x1,\ shr\ s1)\ (x1',\ m1')\ t \vdash (x1',\ m1') \approx (x2,\ shr\ s2)\ final1\ x1'$ **by** *auto*
from ‹$r1.silent\text{-}moves\ t\ (x1,\ shr\ s1)\ (x1',\ m1')$› **have** $x1' = x1$
proof(*cases rule*: *converse-rtranclpE2*[*consumes 1*, *case-names refl step*])
case (*step x1'' m1''*)
from ‹$r1.silent\text{-}move\ t\ (x1,\ shr\ s1)\ (x1'',\ m1'')$›
have $t \vdash (x1,\ shr\ s1)\ -1-\varepsilon\rightarrow\ (x1'',\ m1'')$ **by**(*auto dest*: *r1.silent-tl*)
hence $r1.redT\ s1\ (t,\ \varepsilon)\ (redT\text{-}upd\text{-}\varepsilon\ s1\ t\ x1''\ m1'')$
using ‹$thr\ s1\ t = \lfloor (x1,\ ln) \rfloor$› *True*
by −(*erule r1.redT-normal*, *auto simp add*: *redT-updLns-def finfun-Diag-const2 o-def redT-updWs-def*)
hence *False* **using** *dead* **by**(*auto intro*: *r1.deadlock-no-red r1.red-no-deadlock*)
thus *?thesis* **..**
qed *simp*
with ‹¬ *final1 x1*› ‹*final1 x1'*› **show** *False* **by** *simp*
qed
thus *?thesis* **using** ‹$thr\ s2\ t = \lfloor (x2,\ ln) \rfloor$› **by**(*auto simp add*: *r2.not-final-thread-iff*)
qed
qed

lemma *deadlocked1-imp-τs-deadlocked2*:
assumes *mbisim*: $s1 \approx m\ s2$
and *dead*: $t \in r1.deadlocked\ s1$
shows $\exists s2'.\ r2.mthr.silent\text{-}moves\ s2\ s2' \wedge t \in r2.deadlocked\ s2' \wedge s1 \approx m\ s2'$
proof −

from *mfinal1-inv-simulation*[*OF mbisim*]
obtain *ls2 ts2 m2 ws2 is2* **where** *red1*: *r2.mthr.silent-moves s2* (*ls2*, (*ts2*, *m2*), *ws2*, *is2*)
and *s1* ≈*m* (*ls2*, (*ts2*, *m2*), *ws2*, *is2*) **and** *m2* = *shr s2*
and *fin*: ⋀*t*. *r1.final-thread s1 t* ⟹ *r2.final-thread* (*ls2*, (*ts2*, *m2*), *ws2*, *is2*) *t* **by** *fastforce*
from *no-τMove1-τs-to-no-τMove2*[*OF* ‹*s1* ≈*m* (*ls2*, (*ts2*, *m2*), *ws2*, *is2*)›]
obtain *ts2′* **where** *red2*: *r2.mthr.silent-moves* (*ls2*, (*ts2*, *m2*), *ws2*, *is2*) (*ls2*, (*ts2′*, *m2*), *ws2*, *is2*)
and *no-τ*: ⋀*t x1 x2 x2′ m2′*. ⟦ *wset s1 t* = *None*; *thr s1 t* = ⌊(*x1*, *no-wait-locks*)⌋; *ts2′ t* = ⌊(*x2*, *no-wait-locks*)⌋;
⋀*x′ m′*. *r1.silent-move t* (*x1*, *shr s1*) (*x′*, *m′*) ⟹ *False* ⟧
⟹ ¬ *r2.silent-move t* (*x2*, *m2*) (*x2′*, *m2′*)
and *mbisim*: *s1* ≈*m* (*ls2*, (*ts2′*, *m2*), *ws2*, *is2*) **by** *fastforce*
from *mbisim* **have** *mbisim-eqs*: *ls2* = *locks s1 ws2* = *wset s1 is2* = *interrupts s1*
by(*simp-all add*: *mbisim-def*)
let *?s2* = (*ls2*, (*ts2′*, *m2*), *ws2*, *is2*)
from *red2* **have** *fin′*: ⋀*t*. *r1.final-thread s1 t* ⟹ *r2.final-thread ?s2 t*
by(*rule r2.τmRedT-preserves-final-thread*)(*rule fin*)
from *dead*
have *t* ∈ *r2.deadlocked ?s2*
proof(*coinduct*)
case (*deadlocked t*)
thus *?case*
proof(*cases rule*: *r1.deadlocked-elims*)
case (*lock x1*)
hence *csmw*: ⋀*LT*. *r1.can-sync t x1* (*shr s1*) *LT* ⟹
∃ *lt*∈*LT*. *r1.must-wait s1 t lt* (*r1.deadlocked s1* ∪ *r1.final-threads s1*)
by *blast*
from ‹*thr s1 t* = ⌊(*x1*, *no-wait-locks*)⌋› *mbisim* **obtain** *x2*
where *ts2′ t* = ⌊(*x2*, *no-wait-locks*)⌋ **and** *bisim*: *t* ⊢ (*x1*, *shr s1*) ≈ (*x2*, *m2*)
by(*auto dest*: *mbisim-thrD1*)
note ‹*ts2′ t* = ⌊(*x2*, *no-wait-locks*)⌋› **moreover**
{ from ‹*r1.must-sync t x1* (*shr s1*)› **obtain** *ta1 x1′ m1′*
where *r1*: *t* ⊢ (*x1*, *shr s1*) −1−*ta1*→ (*x1′*, *m1′*)
and *s1′*: ∃ *s1′*. *r1.actions-ok s1′ t ta1* **by**(*fastforce elim*: *r1.must-syncE*)
have ¬ *τmove1* (*x1*, *shr s1*) *ta1* (*x1′*, *m1′*) (**is** ¬ *?τ*)
proof
assume *?τ*
hence *ta1* = *ε* **by**(*rule r1.silent-tl*)
with *r1* **have** *r1.can-sync t x1* (*shr s1*) {}
by(*auto intro*!: *r1.can-syncI simp add*: *collect-locks-def collect-interrupts-def*)
from *csmw*[*OF this*] **show** *False* **by** *blast*
qed
from *simulation1*[*OF bisim r1 this*]
obtain *x2′ m2′ x2″ m2″ ta2* **where** *r2*: *r2.silent-moves t* (*x2*, *m2*) (*x2′*, *m2′*)
and *r2′*: *t* ⊢ (*x2′*, *m2′*) −2−*ta2*→ (*x2″*, *m2″*)
and *τ2*: ¬ *τmove2* (*x2′*, *m2′*) *ta2* (*x2″*, *m2″*)
and *bisim′*: *t* ⊢ (*x1′*, *m1′*) ≈ (*x2″*, *m2″*) **and** *tasim*: *ta1* ∼*m ta2* **by** *auto*
from *r2*
have ∃ *ta2 x2′ m2′ s2′*. *t* ⊢ (*x2*, *m2*) −2−*ta2*→ (*x2′*, *m2′*) ∧ *r2.actions-ok s2′ t ta2*
proof(*cases rule*: *converse-rtranclpE2*[*consumes 1*, *case-names base step*])
case *base*
from *r2′*[*folded base*] *s1′*[*unfolded ex-actions-ok1-conv-ex-actions-ok2*[*OF tasim*]]
show *?thesis* **by** *blast*
next
case (*step x2‴ m2‴*)

hence $t \vdash (x2,\ m2)\ -2-\varepsilon\rightarrow\ (x2''',\ m2''')$ **by**(*auto dest*: *r2.silent-tl*)
moreover have *r2.actions-ok* (*undefined*, (*undefined*, *undefined*), *Map.empty*, *undefined*) $t\ \varepsilon$ **by** *auto*
ultimately show *?thesis* **by**−(*rule exI conjI*|*assumption*)+
qed
hence *r2.must-sync t x2 m2* **unfolding** *r2.must-sync-def2* **. }**
moreover
{ fix *LT*
assume *r2.can-sync t x2 m2 LT*
then obtain *ta2 x2' m2'* **where** *r2*: $t \vdash (x2,\ m2)\ -2-ta2\rightarrow\ (x2',\ m2')$
and *LT*: *LT* = *collect-locks* $\{\!|ta2|\!\}_l$ <+> *collect-cond-actions* $\{\!|ta2|\!\}_c$ <+> *collect-interrupts* $\{\!|ta2|\!\}_i$
by(*auto elim*: *r2.can-syncE*)
from ‹*wset s1 t* = *None*› ‹*thr s1 t* = $\lfloor$(*x1*, *no-wait-locks*)$\rfloor$› ‹*ts2' t* = $\lfloor$(*x2*, *no-wait-locks*)$\rfloor$›
have ¬ *r2.silent-move t* (*x2*, *m2*) (*x2'*, *m2'*)
proof(*rule no-τ*)
fix *x1' m1'*
assume *r1.silent-move t* (*x1*, *shr s1*) (*x1'*, *m1'*)
hence $t \vdash (x1,\ shr\ s1)\ -1-\varepsilon\rightarrow\ (x1',\ m1')$ **by**(*auto dest*: *r1.silent-tl*)
hence *r1.can-sync t x1* (*shr s1*) {}
by(*auto intro*: *r1.can-syncI simp add*: *collect-locks-def collect-interrupts-def*)
with *csmw*[*OF this*] **show** *False* **by** *blast*
qed
with *r2* **have** ¬ *τmove2* (*x2*, *m2*) *ta2* (*x2'*, *m2'*) **by** *auto*
from *simulation2*[*OF bisim r2 this*] **obtain** *x1' m1' x1'' m1'' ta1*
where *τr1*: *r1.silent-moves t* (*x1*, *shr s1*) (*x1'*, *m1'*)
and *r1*: $t \vdash (x1',\ m1')\ -1-ta1\rightarrow\ (x1'',\ m1'')$
and *nτ1*: ¬ *τmove1* (*x1'*, *m1'*) *ta1* (*x1''*, *m1''*)
and *bisim'*: $t \vdash (x1'',\ m1'') \approx (x2',\ m2')$
and *tlsim*: *ta1* ∼*m ta2* **by** *auto*
from *τr1* **obtain** [*simp*]: *x1' = x1 m1' = shr s1*
proof(*cases rule*: *converse-rtranclpE2*[*consumes 1*, *case-names refl step*])
case (*step X M*)
from ‹*r1.silent-move t* (*x1*, *shr s1*) (*X*, *M*)›
have $t \vdash (x1,\ shr\ s1)\ -1-\varepsilon\rightarrow\ (X,\ M)$ **by**(*auto dest*: *r1.silent-tl*)
hence *r1.can-sync t x1* (*shr s1*) {}
by(*auto intro*: *r1.can-syncI simp add*: *collect-locks-def collect-interrupts-def*)
with *csmw*[*OF this*] **have** *False* **by** *blast*
thus *?thesis* **..**
qed *blast*
from *tlsim LT* **have** *LT* = *collect-locks* $\{\!|ta1|\!\}_l$ <+> *collect-cond-actions* $\{\!|ta1|\!\}_c$ <+> *collect-interrupts* $\{\!|ta1|\!\}_i$
by(*auto simp add*: *ta-bisim-def*)
with *r1* **have** *r1.can-sync t x1* (*shr s1*) *LT* **by**(*auto intro*: *r1.can-syncI*)
from *csmw*[*OF this*] **obtain** *lt*
where *lt*: *lt* ∈ *LT* **and** *mw*: *r1.must-wait s1 t lt* (*r1.deadlocked s1* ∪ *r1.final-threads s1*) **by** *blast*
have *subset*: *r1.deadlocked s1* ∪ *r1.final-threads s1* ⊆ *r1.deadlocked s1* ∪ *r2.deadlocked s2* ∪ *r2.final-threads ?s2*
by(*auto dest*: *fin'*)
from *mw* **have** *r2.must-wait ?s2 t lt* (*r1.deadlocked s1* ∪ *r2.deadlocked ?s2* ∪ *r2.final-threads ?s2*)
proof(*cases rule*: *r1.must-wait-elims*)
case *lock* **thus** *?thesis* **by**(*auto simp add*: *mbisim-eqs dest*!: *fin'*)

```
      next
        case (join t')
        from ‹r1.not-final-thread s1 t'› obtain x1 ln
          where thr s1 t' = ⌊(x1, ln)⌋ by cases auto
         with mbisim obtain x2 where ts2' t' = ⌊(x2, ln)⌋ t' ⊢ (x1, shr s1) ≈ (x2, m2) by(auto
dest: mbisim-thrD1)
        show ?thesis
        proof(cases wset s1 t' = None ∧ ln = no-wait-locks)
          case False
          with ‹r1.not-final-thread s1 t'› ‹thr s1 t' = ⌊(x1, ln)⌋› ‹ts2' t' = ⌊(x2, ln)⌋› ‹lt = Inr (Inl
t')› join
          show ?thesis by(auto simp add: mbisim-eqs r2.not-final-thread-iff r1.final-thread-def)
        next
          case True
          with ‹r1.not-final-thread s1 t'› ‹thr s1 t' = ⌊(x1, ln)⌋› have ¬ final1 x1 by(cases) auto
               with join ‹thr s1 t' = ⌊(x1, ln)⌋› have t' ∈ r1.deadlocked s1 by(auto simp add:
r1.final-thread-def)
          have ¬ final2 x2
          proof
            assume final2 x2
            with final2-simulation[OF ‹t' ⊢ (x1, shr s1) ≈ (x2, m2)›]
            obtain x1' m1' where r1.silent-moves t' (x1, shr s1) (x1', m1')
              and t' ⊢ (x1', m1') ≈ (x2, m2) final1 x1' by auto
            from ‹r1.silent-moves t' (x1, shr s1) (x1', m1')› have x1' = x1
            proof(cases rule: converse-rtranclpE2[consumes 1, case-names refl step])
              case (step x1'' m1'')
              from ‹r1.silent-move t' (x1, shr s1) (x1'', m1'')›
              have t' ⊢ (x1, shr s1) −1−ε→ (x1'', m1'') by(auto dest: r1.silent-tl)
              hence r1.redT s1 (t', ε) (redT-upd-ε s1 t' x1'' m1'')
                using ‹thr s1 t' = ⌊(x1, ln)⌋› True
                    by −(erule r1.redT-normal, auto simp add: redT-updLns-def redT-updWs-def fin-
fun-Diag-const2 o-def)
              hence False using ‹t' ∈ r1.deadlocked s1› by(rule r1.red-no-deadlock)
              thus ?thesis ..
            qed simp
            with ‹¬ final1 x1› ‹final1 x1'› show False by simp
          qed
          thus ?thesis using ‹ts2' t' = ⌊(x2, ln)⌋› join
            by(auto simp add: r2.not-final-thread-iff r1.final-thread-def)
        qed
      next
        case (interrupt t')
        have r2.all-final-except ?s2 (r1.deadlocked s1 ∪ r2.deadlocked ?s2 ∪ r2.final-threads ?s2)
        proof(rule r2.all-final-exceptI)
          fix t''
          assume r2.not-final-thread ?s2 t''
          then obtain x2 ln where thr ?s2 t'' = ⌊(x2, ln)⌋
            and fin: ¬ final2 x2 ∨ ln ≠ no-wait-locks ∨ wset ?s2 t'' ≠ None
            by(auto simp add: r2.not-final-thread-iff)
          from ‹thr ?s2 t'' = ⌊(x2, ln)⌋› mbisim
          obtain x1 where ts1t'': thr s1 t'' = ⌊(x1, ln)⌋
            and bisim'': t'' ⊢ (x1, shr s1) ≈ (x2, shr ?s2)
            by(auto dest: mbisim-thrD2)
          have r1.not-final-thread s1 t''
```

```
          proof(cases wset ?s2 t'' = None ∧ ln = no-wait-locks)
            case True
            with fin have ¬ final2 x2 by simp
            hence ¬ final1 x1
            proof(rule contrapos-nn)
              assume final1 x1
              with final1-simulation[OF bisim''] 
              obtain x2' m2' where τs2: r2.silent-moves t'' (x2, shr ?s2) (x2', m2')
                and bisim''': t'' ⊢ (x1, shr s1) ≈ (x2', m2')
                and final2 x2' by auto
              from τs2 have x2' = x2
              proof(cases rule: converse-rtranclpE2[consumes 1, case-names refl step])
                case refl thus ?thesis by simp
              next
                case (step x2'' m2'')
                from True have wset s1 t'' = None thr s1 t'' = ⌊(x1, no-wait-locks)⌋ ts2' t'' = ⌊(x2,
no-wait-locks)⌋
                  using ts1t'' ‹thr ?s2 t'' = ⌊(x2, ln)⌋› mbisim by(simp-all add: mbisim-def)
                hence no-τ2: ¬ r2.silent-move t'' (x2, m2) (x2'', m2'')
                proof(rule no-τ)
                  fix x1' m1'
                  assume r1.silent-move t'' (x1, shr s1) (x1', m1')
                  with ‹final1 x1› show False by(auto dest: r1.final-no-red)
                qed
                with ‹r2.silent-move t'' (x2, shr ?s2) (x2'', m2'')› have False by simp
                thus ?thesis ..
              qed
              with ‹final2 x2'› show final2 x2 by simp
            qed
            with ts1t'' show ?thesis ..
          next
            case False
            with ts1t'' mbisim show ?thesis by(auto simp add: r1.not-final-thread-iff mbisim-def)
          qed
          with ‹r1.all-final-except s1 (r1.deadlocked s1 ∪ r1.final-threads s1)›
          have t'' ∈ r1.deadlocked s1 ∪ r1.final-threads s1 by(rule r1.all-final-exceptD)
          thus t'' ∈ r1.deadlocked s1 ∪ r2.deadlocked ?s2 ∪ r2.final-threads ?s2
            by(auto dest: fin' simp add: mbisim-eqs)
        qed
        thus ?thesis using interrupt mbisim by(auto simp add: mbisim-def)
      qed
      hence ∃lt∈LT. r2.must-wait ?s2 t lt (r1.deadlocked s1 ∪ r2.deadlocked ?s2 ∪ r2.final-threads
?s2)
        using ‹lt ∈ LT› by blast }
    moreover from mbisim ‹wset s1 t = None› have wset ?s2 t = None by(simp add: mbisim-def)
    ultimately have ?Lock by simp
    thus ?thesis ..
  next
    case (wait x1 ln)
    from mbisim ‹thr s1 t = ⌊(x1, ln)⌋›
    obtain x2 where ts2' t = ⌊(x2, ln)⌋ by(auto dest: mbisim-thrD1)
    moreover
    have r2.all-final-except ?s2 (r1.deadlocked s1)
    proof(rule r2.all-final-exceptI)
```

fix *t*
assume *r2.not-final-thread ?s2 t*
then obtain *x2 ln* **where** *ts2′ t* = ⌊(*x2*, *ln*)⌋ **by**(*auto simp add*: *r2.not-final-thread-iff*)
with *mbisim* **obtain** *x1* **where** *thr s1 t* = ⌊(*x1*, *ln*)⌋ *t* ⊢ (*x1*, *shr s1*) ≈ (*x2*, *m2*) **by**(*auto dest*: *mbisim-thrD2*)
hence *r1.not-final-thread s1 t* **using** ‹*r2.not-final-thread ?s2 t*› ‹*ts2′ t* = ⌊(*x2*, *ln*)⌋› *mbisim fin′*[*of t*]
by(*cases wset s1 t*)(*auto simp add*: *r1.not-final-thread-iff r2.not-final-thread-iff mbisim-def r1.final-thread-def r2.final-thread-def*)
with ‹*r1.all-final-except s1* (*r1.deadlocked s1*)›
show *t* ∈ *r1.deadlocked s1* **by**(*rule r1.all-final-exceptD*)
qed
hence *r2.all-final-except ?s2* (*r1.deadlocked s1* ∪ *r2.deadlocked ?s2*)
by(*rule r2.all-final-except-mono′*) *blast*
moreover
from ‹*waiting* (*wset s1 t*)› *mbisim*
have *waiting* (*wset ?s2 t*) **by**(*simp add*: *mbisim-def*)
ultimately have *?Wait* **by** *simp*
thus *?thesis* **by** *blast*
next
case (*acquire x1 ln l t′*)
from *mbisim* ‹*thr s1 t* = ⌊(*x1*, *ln*)⌋›
obtain *x2* **where** *ts2′ t* = ⌊(*x2*, *ln*)⌋ **by**(*auto dest*: *mbisim-thrD1*)
moreover
from ‹*t′* ∈ *r1.deadlocked s1* ∨ *r1.final-thread s1 t′*›
have (*t′* ∈ *r1.deadlocked s1* ∨ *t′* ∈ *r2.deadlocked ?s2*) ∨ *r2.final-thread ?s2 t′* **by**(*blast dest*: *fin′*)
moreover
from *mbisim* ‹*has-lock* (*locks s1* $ *l*) *t′*›
have *has-lock* (*locks ?s2* $ *l*) *t′* **by**(*simp add*: *mbisim-def*)
ultimately have *?Acquire*
using ‹*0* < *ln* $ *l*› ‹*t* ≠ *t′*› ‹¬ *waiting* (*wset s1 t*)› *mbisim*
by(*auto simp add*: *mbisim-def*)
thus *?thesis* **by** *blast*
qed
qed
with *red1 red2 mbisim* **show** *?thesis* **by**(*blast intro*: *rtranclp-trans*)
qed

lemma *deadlocked2-imp-τs-deadlocked1*:
⟦ *s1* ≈*m s2*; *t* ∈ *r2.deadlocked s2* ⟧
⟹ ∃*s1′*. *r1.mthr.silent-moves s1 s1′* ∧ *t* ∈ *r1.deadlocked s1′* ∧ *s1′* ≈*m s2*
using *FWdelay-bisimulation-diverge.deadlocked1-imp-τs-deadlocked2*[*OF FWdelay-bisimulation-diverge-flip*]
unfolding *flip-simps* **.**

lemma *deadlock1-imp-τs-deadlock2*:
assumes *mbisim*: *s1* ≈*m s2*
and *dead*: *r1.deadlock s1*
shows ∃*s2′*. *r2.mthr.silent-moves s2 s2′* ∧ *r2.deadlock s2′* ∧ *s1* ≈*m s2′*
proof(*cases* ∃*t*. *r1.not-final-thread s1 t*)
case *True*
then obtain *t* **where** *nfin*: *r1.not-final-thread s1 t* **..**
from *mfinal1-inv-simulation*[*OF mbisim*]
obtain *ls2 ts2 m2 ws2 is2* **where** *red1*: *r2.mthr.silent-moves s2* (*ls2*, (*ts2*, *m2*), *ws2*, *is2*)
and *s1* ≈*m* (*ls2*, (*ts2*, *m2*), *ws2*, *is2*) **and** *m2* = *shr s2*

```
  and fin: ⋀t. r1.final-thread s1 t ⟹ r2.final-thread (ls2, (ts2, m2), ws2, is2) t by fastforce
 from no-τMove1-τs-to-no-τMove2[OF ‹s1 ≈m (ls2, (ts2, m2), ws2, is2)›]
 obtain ts2' where red2: r2.mthr.silent-moves (ls2, (ts2, m2), ws2, is2) (ls2, (ts2', m2), ws2, is2)
  and no-τ: ⋀t x1 x2 x2' m2'. ⟦ wset s1 t = None; thr s1 t = ⌊(x1, no-wait-locks)⌋; ts2' t = ⌊(x2,
no-wait-locks)⌋;
                        ⋀x' m'. r1.silent-move t (x1, shr s1) (x', m') ⟹ False ⟧
        ⟹ ¬ r2.silent-move t (x2, m2) (x2', m2')
  and mbisim: s1 ≈m (ls2, (ts2', m2), ws2, is2) by fastforce
 from mbisim have mbisim-eqs: ls2 = locks s1 ws2 = wset s1 is2 = interrupts s1
  by(simp-all add: mbisim-def)
 let ?s2 = (ls2, (ts2', m2), ws2, is2)
 from red2 have fin': ⋀t. r1.final-thread s1 t ⟹ r2.final-thread ?s2 t
  by(rule r2.τmRedT-preserves-final-thread)(rule fin)
 have r2.deadlock ?s2
 proof(rule r2.deadlockI, goal-cases)
  case (1 t x2)
  note ts2t = ‹thr ?s2 t = ⌊(x2, no-wait-locks)⌋›
  with mbisim obtain x1 where ts1t: thr s1 t = ⌊(x1, no-wait-locks)⌋
   and bisim: t ⊢ (x1, shr s1) ≈ (x2, m2) by(auto dest: mbisim-thrD2)
  from ‹wset ?s2 t = None› mbisim have ws1t: wset s1 t = None by(simp add: mbisim-def)
  have ¬ final1 x1
  proof
   assume final1 x1
   with ts1t ws1t have r1.final-thread s1 t by(simp add: r1.final-thread-def)
   hence r2.final-thread ?s2 t by(rule fin')
   with ‹¬ final2 x2› ts2t ‹wset ?s2 t = None› show False by(simp add: r2.final-thread-def)
  qed
  from r1.deadlockD1[OF dead ts1t this ‹wset s1 t = None›]
  have ms: r1.must-sync t x1 (shr s1)
   and csmw: ⋀LT. r1.can-sync t x1 (shr s1) LT ⟹ ∃lt∈LT. r1.must-wait s1 t lt (dom (thr s1))
   by blast+
  {
   from ‹r1.must-sync t x1 (shr s1)› obtain ta1 x1' m1'
    where r1: t ⊢ (x1, shr s1) −1−ta1→ (x1', m1')
    and s1': ∃s1'. r1.actions-ok s1' t ta1 by(fastforce elim: r1.must-syncE)
   have ¬ τmove1 (x1, shr s1) ta1 (x1', m1') (is ¬ ?τ)
   proof
    assume ?τ
    hence ta1 = ε by(rule r1.silent-tl)
    with r1 have r1.can-sync t x1 (shr s1) {}
     by(auto intro!: r1.can-syncI simp add: collect-locks-def collect-interrupts-def)
    from csmw[OF this] show False by blast
   qed
   from simulation1[OF bisim r1 this]
   obtain x2' m2' x2'' m2'' ta2 where r2: r2.silent-moves t (x2, m2) (x2', m2')
    and r2': t ⊢ (x2', m2') −2−ta2→ (x2'', m2'')
    and bisim': t ⊢ (x1', m1') ≈ (x2'', m2'') and tasim: ta1 ∼m ta2 by auto
   from r2
   have ∃ta2 x2' m2' s2'. t ⊢ (x2, m2) −2−ta2→ (x2', m2') ∧ r2.actions-ok s2' t ta2
   proof(cases rule: converse-rtranclpE2[consumes 1, case-names base step])
    case base
    from r2'[folded base] s1'[unfolded ex-actions-ok1-conv-ex-actions-ok2[OF tasim]]
    show ?thesis by blast
   next
```

```
      case (step x2''' m2''')
      hence t ⊢ (x2, m2) −2−ε→ (x2''', m2''') by(auto dest: r2.silent-tl)
       moreover have r2.actions-ok (undefined, (undefined, undefined), Map.empty, undefined) t ε
by auto
      ultimately show ?thesis by−(rule exI conjI|assumption)+
    qed
    hence r2.must-sync t x2 m2 unfolding r2.must-sync-def2 . }
  moreover
  { fix LT
    assume r2.can-sync t x2 m2 LT
    then obtain ta2 x2' m2' where r2: t ⊢ (x2, m2) −2−ta2→ (x2', m2')
      and LT: LT = collect-locks ⦃ta2⦄l <+> collect-cond-actions ⦃ta2⦄c <+> collect-interrupts
⦃ta2⦄i
      by(auto elim: r2.can-syncE)
    from ts2t have ts2' t = ⌊(x2, no-wait-locks)⌋ by simp
    with ws1t ts1t have ¬ r2.silent-move t (x2, m2) (x2', m2')
    proof(rule no-τ)
      fix x1' m1'
      assume r1.silent-move t (x1, shr s1) (x1', m1')
      hence t ⊢ (x1, shr s1) −1−ε→ (x1', m1') by(auto dest: r1.silent-tl)
      hence r1.can-sync t x1 (shr s1) {}
        by(auto intro: r1.can-syncI simp add: collect-locks-def collect-interrupts-def)
      with csmw[OF this] show False by blast
    qed
    with r2 have ¬ τmove2 (x2, m2) ta2 (x2', m2') by auto
    from simulation2[OF bisim r2 this] obtain x1' m1' x1'' m1'' ta1
      where τr1: r1.silent-moves t (x1, shr s1) (x1', m1')
      and r1: t ⊢ (x1', m1') −1−ta1→ (x1'', m1'')
      and nτ1: ¬ τmove1 (x1', m1') ta1 (x1'', m1'')
      and bisim': t ⊢ (x1'', m1'') ≈ (x2', m2')
      and tlsim: ta1 ∼m ta2 by auto
    from τr1 obtain [simp]: x1' = x1 m1' = shr s1
    proof(cases rule: converse-rtranclpE2[consumes 1, case-names refl step])
      case (step X M)
      from ‹r1.silent-move t (x1, shr s1) (X, M)›
      have t ⊢ (x1, shr s1) −1−ε→ (X, M) by(auto dest: r1.silent-tl)
      hence r1.can-sync t x1 (shr s1) {}
        by(auto intro: r1.can-syncI simp add: collect-locks-def collect-interrupts-def)
      with csmw[OF this] have False by blast
      thus ?thesis ..
    qed blast
    from tlsim LT have LT = collect-locks ⦃ta1⦄l <+> collect-cond-actions ⦃ta1⦄c <+> col-
lect-interrupts ⦃ta1⦄i
      by(auto simp add: ta-bisim-def)
    with r1 have r1.can-sync t x1 (shr s1) LT by(auto intro: r1.can-syncI)
    from csmw[OF this] obtain lt
      where lt: lt ∈ LT r1.must-wait s1 t lt (dom (thr s1)) by blast
    from ‹r1.must-wait s1 t lt (dom (thr s1))› have r2.must-wait ?s2 t lt (dom (thr ?s2))
    proof(cases rule: r1.must-wait-elims)
      case (lock l)
      with mbisim-dom-eq[OF mbisim] show ?thesis by(auto simp add: mbisim-eqs)
    next
      case (join t')
      from dead deadlock-mbisim-not-final-thread-pres[OF - ‹r1.not-final-thread s1 t'› fin' mbisim]
```

```
      have r2.not-final-thread ?s2 t' by auto
      thus ?thesis using join mbisim-dom-eq[OF mbisim] by auto
    next
      case (interrupt t')
      have r2.all-final-except ?s2 (dom (thr ?s2)) by(auto intro!: r2.all-final-exceptI)
      with interrupt show ?thesis by(auto simp add: mbisim-eqs)
    qed
    with lt have ∃ lt∈LT. r2.must-wait ?s2 t lt (dom (thr ?s2)) by blast }
  ultimately show ?case by fastforce
 next
  case (2 t x2 ln l)
  note dead moreover
  from mbisim ‹thr ?s2 t = ⌊(x2, ln)⌋›
  obtain x1 where thr s1 t = ⌊(x1, ln)⌋ by(auto dest: mbisim-thrD2)
  moreover note ‹0 < ln $ l›
  moreover from ‹¬ waiting (wset ?s2 t)› mbisim
  have ¬ waiting (wset s1 t) by(simp add: mbisim-def)
  ultimately obtain l' t' where 0 < ln $ l' t ≠ t' thr s1 t' ≠ None has-lock (locks s1 $ l') t'
    by(rule r1.deadlockD2)
  thus ?case using mbisim-thrNone-eq[OF mbisim, of t'] mbisim by(auto simp add: mbisim-def)
 next
  case (3 t x2 w)
  from mbisim-thrD2[OF mbisim this]
  obtain x1 where thr s1 t = ⌊(x1, no-wait-locks)⌋ by auto
  with dead have wset s1 t ≠ ⌊PostWS w⌋ by(rule r1.deadlockD3[rule-format])
  with mbisim show ?case by(simp add: mbisim-def)
 qed
 with red1 red2 mbisim show ?thesis by(blast intro: rtranclp-trans)
next
 case False
 hence r1.mfinal s1 by(auto intro: r1.mfinalI simp add: r1.not-final-thread-iff)
 from mfinal1-simulation[OF mbisim this]
 obtain s2' where τmRed2 s2 s2' s1 ≈m s2' r2.mfinal s2' shr s2' = shr s2 by blast
 thus ?thesis by(blast intro: r2.mfinal-deadlock)
qed

lemma deadlock2-imp-τs-deadlock1:
 ⟦ s1 ≈m s2; r2.deadlock s2 ⟧
 ⟹ ∃ s1'. r1.mthr.silent-moves s1 s1' ∧ r1.deadlock s1' ∧ s1' ≈m s2
using FWdelay-bisimulation-diverge.deadlock1-imp-τs-deadlock2[OF FWdelay-bisimulation-diverge-flip]
unfolding flip-simps .

lemma deadlocked'1-imp-τs-deadlocked'2:
 ⟦ s1 ≈m s2; r1.deadlocked' s1 ⟧
 ⟹ ∃ s2'. r2.mthr.silent-moves s2 s2' ∧ r2.deadlocked' s2' ∧ s1 ≈m s2'
unfolding r1.deadlock-eq-deadlocked'[symmetric] r2.deadlock-eq-deadlocked'[symmetric]
by(rule deadlock1-imp-τs-deadlock2)

lemma deadlocked'2-imp-τs-deadlocked'1:
 ⟦ s1 ≈m s2; r2.deadlocked' s2 ⟧ ⟹ ∃ s1'. r1.mthr.silent-moves s1 s1' ∧ r1.deadlocked' s1' ∧ s1'
≈m s2
unfolding r1.deadlock-eq-deadlocked'[symmetric] r2.deadlock-eq-deadlocked'[symmetric]
by(rule deadlock2-imp-τs-deadlock1)
```

end

context *FWbisimulation* **begin**

lemma *mbisim-final-thread-preserve1*:
assumes *mbisim*: *s1* $\approx m$ *s2* **and** *fin*: *r1.final-thread s1 t*
shows *r2.final-thread s2 t*
proof −
from *fin* **obtain** *x1* **where** *ts1t*: *thr s1 t* = $\lfloor$(*x1*, *no-wait-locks*)$\rfloor$
and *fin1*: *final1 x1* **and** *ws1t*: *wset s1 t* = *None*
by(*auto elim*: *r1.final-threadE*)
from *mbisim ts1t* **obtain** *x2*
where *ts2t*: *thr s2 t* = $\lfloor$(*x2*, *no-wait-locks*)$\rfloor$
and *bisim*: *t* ⊢ (*x1*, *shr s1*) ≈ (*x2*, *shr s2*) **by**(*auto dest*: *mbisim-thrD1*)
note *ts2t* **moreover from** *fin1 bisim* **have** *final2 x2* **by**(*auto dest*: *bisim-final*)
moreover from *mbisim ws1t* **have** *wset s2 t* = *None* **by**(*simp add*: *mbisim-def*)
ultimately show *?thesis* **by**(*rule r2.final-threadI*)
qed

lemma *mbisim-final-thread-preserve2*:
⟦ *s1* $\approx m$ *s2*; *r2.final-thread s2 t* ⟧ ⟹ *r1.final-thread s1 t*
using *FWbisimulation.mbisim-final-thread-preserve1*[*OF FWbisimulation-flip*]
unfolding *flip-simps* **.**

lemma *mbisim-final-thread-inv*:
s1 $\approx m$ *s2* ⟹ *r1.final-thread s1 t* ⟷ *r2.final-thread s2 t*
by(*blast intro*: *mbisim-final-thread-preserve1 mbisim-final-thread-preserve2*)

lemma *mbisim-not-final-thread-inv*:
assumes *bisim*: *mbisim s1 s2*
shows *r1.not-final-thread s1* = *r2.not-final-thread s2*
proof(*rule ext*)
fix *t*
show *r1.not-final-thread s1 t* = *r2.not-final-thread s2 t*
proof(*cases thr s1 t*)
case *None*
with *mbisim-thrNone-eq*[*OF bisim*, *of t*] **have** *thr s2 t* = *None* **by** *simp*
with *None* **show** *?thesis*
by(*auto elim*!: *r2.not-final-thread.cases r1.not-final-thread.cases*
intro: *r2.not-final-thread.intros r1.not-final-thread.intros*)
next
case (*Some a*)
then obtain *x1 ln* **where** *tst1*: *thr s1 t* = $\lfloor$(*x1*, *ln*)$\rfloor$ **by**(*cases a*) *auto*
from *mbisim-thrD1*[*OF bisim tst1*] **obtain** *x2*
where *tst2*: *thr s2 t* = $\lfloor$(*x2*, *ln*)$\rfloor$ **and** *bisimt*: *t* ⊢ (*x1*, *shr s1*) ≈ (*x2*, *shr s2*) **by** *blast*
from *bisim* **have** *wset s2* = *wset s1* **by**(*simp add*: *mbisim-def*)
with *tst2 tst1 bisim-final*[*OF bisimt*] **show** *?thesis*
by(*simp add*: *r1.not-final-thread-conv r2.not-final-thread-conv*)(*rule mbisim-final-thread-inv*[*OF bisim*])
qed
qed

lemma *mbisim-deadlocked-preserve1*:
assumes *mbisim*: *s1* $\approx m$ *s2* **and** *dead*: *t* ∈ *r1.deadlocked s1*

shows $t \in r2.deadlocked\ s2$
proof −
from *deadlocked1-imp-τs-deadlocked2*[*OF mbisim dead*]
obtain *s2'* **where** *r2.mthr.silent-moves s2 s2'*
and $t \in r2.deadlocked\ s2'$ **by** *blast*
from ‹*r2.mthr.silent-moves s2 s2'*› **have** $s2' = s2$
by(*rule converse-rtranclpE*)(*auto elim*: *r2.mτmove.cases*)
with ‹$t \in r2.deadlocked\ s2'$›
show *?thesis* **by** *simp*
qed

lemma *mbisim-deadlocked-preserve2*:
⟦ $s1 \approx m\ s2$; $t \in r2.deadlocked\ s2$ ⟧ ⟹ $t \in r1.deadlocked\ s1$
using *FWbisimulation.mbisim-deadlocked-preserve1*[*OF FWbisimulation-flip*]
unfolding *flip-simps* **.**

lemma *mbisim-deadlocked-inv*:
$s1 \approx m\ s2 \Longrightarrow r1.deadlocked\ s1 = r2.deadlocked\ s2$
by(*blast intro*!: *mbisim-deadlocked-preserve1 mbisim-deadlocked-preserve2*)

lemma *mbisim-deadlocked'-inv*:
$s1 \approx m\ s2 \Longrightarrow r1.deadlocked'\ s1 \longleftrightarrow r2.deadlocked'\ s2$
unfolding *r1.deadlocked'-def r2.deadlocked'-def*
by(*simp add*: *mbisim-not-final-thread-inv mbisim-deadlocked-inv*)

lemma *mbisim-deadlock-inv*:
$s1 \approx m\ s2 \Longrightarrow r1.deadlock\ s1 = r2.deadlock\ s2$
unfolding *r1.deadlock-eq-deadlocked' r2.deadlock-eq-deadlocked'*
by(*rule mbisim-deadlocked'-inv*)

end

context *FWbisimulation* **begin**

lemma *bisim-can-sync-preserve1*:
assumes *bisim*: $t \vdash (x1, m1) \approx (x2, m2)$ **and** *cs*: $t \vdash \langle x1, m1 \rangle\ LT\ \wr 1$
shows $t \vdash \langle x2, m2 \rangle\ LT\ \wr 2$
proof −
from *cs* **obtain** *ta1 x1' m1'* **where** *red1*: $t \vdash (x1, m1)\ -1-ta1\rightarrow\ (x1', m1')$
and *LT*: $LT = collect\text{-}locks\ \{\!|ta1|\!\}_l <+> collect\text{-}cond\text{-}actions\ \{\!|ta1|\!\}_c <+> collect\text{-}interrupts\ \{\!|ta1|\!\}_i$
by(*rule r1.can-syncE*)
from *bisimulation.simulation1*[*OF bisimulation-axioms*, *OF bisim red1*] **obtain** *x2' ta2 m2'*
where *red2*: $t \vdash (x2, m2)\ -2-ta2\rightarrow\ (x2', m2')$
and *tasim*: $ta1 \sim m\ ta2$ **by** *fastforce*
from *tasim LT* **have** $LT = collect\text{-}locks\ \{\!|ta2|\!\}_l <+> collect\text{-}cond\text{-}actions\ \{\!|ta2|\!\}_c <+> collect\text{-}interrupts\ \{\!|ta2|\!\}_i$
by(*auto simp add*: *ta-bisim-def*)
with *red2* **show** *?thesis* **by**(*rule r2.can-syncI*)
qed

lemma *bisim-can-sync-preserve2*:
⟦ $t \vdash (x1, m1) \approx (x2, m2)$; $t \vdash \langle x2, m2 \rangle\ LT\ \wr 2$ ⟧ ⟹ $t \vdash \langle x1, m1 \rangle\ LT\ \wr 1$
using *FWbisimulation.bisim-can-sync-preserve1*[*OF FWbisimulation-flip*]

unfolding *flip-simps* **.**

lemma *bisim-can-sync-inv*:
$t \vdash (x1,\ m1) \approx (x2,\ m2) \Longrightarrow t \vdash \langle x1,\ m1\rangle\ LT\ \wr 1 \longleftrightarrow t \vdash \langle x2,\ m2\rangle\ LT\ \wr 2$
by(*blast intro*: *bisim-can-sync-preserve1 bisim-can-sync-preserve2*)

lemma *bisim-must-sync-preserve1*:
assumes *bisim*: $t \vdash (x1,\ m1) \approx (x2,\ m2)$ **and** *ms*: $t \vdash \langle x1,\ m1\rangle\ \wr 1$
shows $t \vdash \langle x2,\ m2\rangle\ \wr 2$
proof −
from *ms* **obtain** *ta1 x1' m1'* **where** *red1*: $t \vdash (x1,\ m1)\ -1-ta1\rightarrow\ (x1',\ m1')$
and *s1'*: $\exists s1'.\ r1.actions\text{-}ok\ s1'\ t\ ta1$ **by**(*fastforce elim*: *r1.must-syncE*)
from *bisimulation.simulation1*[*OF bisimulation-axioms*, *OF bisim red1*] **obtain** *x2' ta2 m2'*
where *red2*: $t \vdash (x2,\ m2)\ -2-ta2\rightarrow\ (x2',\ m2')$
and *tasim*: $ta1 \sim m\ ta2$ **by** *fastforce*
from *ex-actions-ok1-conv-ex-actions-ok2*[*OF tasim*, *of t*] *s1' red2*
show *?thesis* **unfolding** *r2.must-sync-def2* **by** *blast*
qed

lemma *bisim-must-sync-preserve2*:
$[\![\ t \vdash (x1,\ m1) \approx (x2,\ m2);\ t \vdash \langle x2,\ m2\rangle\ \wr 2\ ]\!] \Longrightarrow t \vdash \langle x1,\ m1\rangle\ \wr 1$
using *FWbisimulation.bisim-must-sync-preserve1*[*OF FWbisimulation-flip*]
unfolding *flip-simps* **.**

lemma *bisim-must-sync-inv*:
$t \vdash (x1,\ m1) \approx (x2,\ m2) \Longrightarrow t \vdash \langle x1,\ m1\rangle\ \wr 1 \longleftrightarrow t \vdash \langle x2,\ m2\rangle\ \wr 2$
by(*blast intro*: *bisim-must-sync-preserve1 bisim-must-sync-preserve2*)

end

end

1.20 Semantic properties of lifted predicates

theory *FWLiftingSem*
imports
FWSemantics
FWLifting
begin

context *multithreaded-base* **begin**

lemma *redT-preserves-ts-inv-ok*:
$[\![\ s\ -t\triangleright ta\rightarrow\ s';\ ts\text{-}inv\text{-}ok\ (thr\ s)\ I\ ]\!]$
$\Longrightarrow ts\text{-}inv\text{-}ok\ (thr\ s')\ (upd\text{-}invs\ I\ P\ \{\!|ta|\!\}_t)$
by(*erule redT.cases*)(*fastforce intro*: *ts-inv-ok-upd-invs ts-inv-ok-upd-ts redT-updTs-Some*)+

lemma *RedT-preserves-ts-inv-ok*:
$[\![\ s\ -\triangleright ttas\rightarrow *\ s';\ ts\text{-}inv\text{-}ok\ (thr\ s)\ I\ ]\!]$
$\Longrightarrow ts\text{-}inv\text{-}ok\ (thr\ s')\ (upd\text{-}invs\ I\ Q\ (concat\ (map\ (thr\text{-}a \circ snd)\ ttas)))$
by(*induct rule*: *RedT-induct*)(*auto intro*: *redT-preserves-ts-inv-ok*)

lemma *redT-upd-inv-ext*:

fixes $I :: 't \rightharpoonup 'i$
shows $[\![s -t▷ta→ s'; ts\text{-}inv\text{-}ok\ (thr\ s)\ I]\!] \Longrightarrow I \subseteq_m upd\text{-}invs\ I\ P\ \{\!|ta|\!\}_t$
by(*erule redT.cases*, *auto intro*: *ts-inv-ok-inv-ext-upd-invs*)

lemma *RedT-upd-inv-ext*:
fixes $I :: 't \rightharpoonup 'i$
shows $[\![s -▷ttas→* s'; ts\text{-}inv\text{-}ok\ (thr\ s)\ I]\!]$
$\Longrightarrow I \subseteq_m upd\text{-}invs\ I\ P\ (concat\ (map\ (thr\text{-}a \circ snd)\ ttas))$
proof(*induct rule*: *RedT-induct*)
case *refl* **thus** *?case* **by** *simp*
next
case (*step S TTAS S' T TA S''*)
hence *ts-inv-ok* (*thr S'*) (*upd-invs I P* (*concat* (*map* (*thr-a* ∘ *snd*) *TTAS*)))
by −(*rule RedT-preserves-ts-inv-ok*)
hence *upd-invs I P* (*concat* (*map* (*thr-a* ∘ *snd*) *TTAS*)) ⊆$_m$ *upd-invs* (*upd-invs I P* (*concat* (*map* (*thr-a* ∘ *snd*) *TTAS*))) *P* $\{\!|TA|\!\}_t$
using *step* **by** −(*rule redT-upd-inv-ext*)
with *step* **show** *?case* **by**(*auto elim*!: *map-le-trans simp add*: *comp-def*)
qed

end

locale *lifting-inv* = *multithreaded final r convert-RA*
for *final* :: *'x* ⇒ *bool*
and *r* :: (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *semantics* (‹- ⊢ - −-→ -› [*50*,*0*,*0*,*50*] *80*)
and *convert-RA* :: *'l released-locks* ⇒ *'o list*
+
fixes *P* :: *'i* ⇒ *'t* ⇒ *'x* ⇒ *'m* ⇒ *bool*
assumes *invariant-red*: $[\![t \vdash \langle x, m\rangle -ta→ \langle x', m'\rangle; P\ i\ t\ x\ m]\!] \Longrightarrow P\ i\ t\ x'\ m'$
and *invariant-NewThread*: $[\![t \vdash \langle x, m\rangle -ta→ \langle x', m'\rangle; P\ i\ t\ x\ m; NewThread\ t''\ x''\ m' \in set\ \{\!|ta|\!\}_t]\!]$
$\Longrightarrow \exists i''.\ P\ i''\ t''\ x''\ m'$
and *invariant-other*: $[\![t \vdash \langle x, m\rangle -ta→ \langle x', m'\rangle; P\ i\ t\ x\ m; P\ i''\ t''\ x''\ m]\!] \Longrightarrow P\ i''\ t''\ x''\ m'$
begin

lemma *redT-updTs-invariant*:
fixes *ln*
assumes *tsiP*: *ts-inv P I ts m*
and *red*: $t \vdash \langle x, m\rangle -ta→ \langle x', m'\rangle$
and *tao*: *thread-oks ts* $\{\!|ta|\!\}_t$
and *tst*: *ts t* = ⌊(*x*, *ln*)⌋
shows *ts-inv P* (*upd-invs I P* $\{\!|ta|\!\}_t$) ((*redT-updTs ts* $\{\!|ta|\!\}_t$)(*t* ↦ (*x'*, *ln'*))) *m'*
proof(*rule ts-invI*)
fix *T X LN*
assume *XLN*: ((*redT-updTs ts* $\{\!|ta|\!\}_t$)(*t* ↦ (*x'*, *ln'*))) *T* = ⌊(*X*, *LN*)⌋
from *tsiP* ‹*ts t* = ⌊(*x*, *ln*)⌋› **obtain** *i* **where** *I t* = ⌊*i*⌋ *P i t x m*
by(*auto dest*: *ts-invD*)
show ∃ *i*. *upd-invs I P* $\{\!|ta|\!\}_t$ *T* = ⌊*i*⌋ ∧ *P i T X m'*
proof(*cases T* = *t*)
case *True*
from *red* ‹*P i t x m*› **have** *P i t x' m'* **by**(*rule invariant-red*)
moreover from ‹*I t* = ⌊*i*⌋› ‹*ts t* = ⌊(*x*, *ln*)⌋› *tao*
have *upd-invs I P* $\{\!|ta|\!\}_t$ *t* = ⌊*i*⌋
by(*simp add*: *upd-invs-Some*)

```
    ultimately show ?thesis using True XLN by simp
  next
    case False
    show ?thesis
    proof(cases ts T)
      case None
      with XLN tao False have ∃ m'. NewThread T X m' ∈ set {|ta|}_t
        by(auto dest: redT-updTs-new-thread)
      with red have nt: NewThread T X m' ∈ set {|ta|}_t by(auto dest: new-thread-memory)
      with red ‹P i t x m› have ∃ i''. P i'' T X m' by(rule invariant-NewThread)
      hence P (SOME i. P i T X m') T X m' by(rule someI-ex)
      with nt tao show ?thesis by(auto intro: SOME-new-thread-upd-invs)
    next
      case (Some a)
      obtain X' LN' where [simp]: a = (X', LN') by(cases a)
      with ‹ts T = ⌊a⌋› have esT: ts T = ⌊(X', LN')⌋ by simp
      hence redT-updTs ts {|ta|}_t T = ⌊(X', LN')⌋
        using ‹thread-oks ts {|ta|}_t› by(auto intro: redT-updTs-Some)
      moreover from esT tsiP obtain i' where I T = ⌊i'⌋ P i' T X' m
        by(auto dest: ts-invD)
      from red ‹P i t x m› ‹P i' T X' m›
      have P i' T X' m' by(rule invariant-other)
      moreover from ‹I T = ⌊i'⌋› esT tao have upd-invs I P {|ta|}_t T = ⌊i'⌋
        by(simp add: upd-invs-Some)
      ultimately show ?thesis using XLN False by simp
    qed
  qed
qed

theorem redT-invariant:
  assumes redT: s −t▷ta→ s'
  and esinvP: ts-inv P I (thr s) (shr s)
  shows ts-inv P (upd-invs I P {|ta|}_t) (thr s') (shr s')
using redT
proof(cases rule: redT-elims)
  case acquire thus ?thesis using esinvP
    by(auto intro!: ts-invI split: if-split-asm dest: ts-invD)
next
  case (normal x x' m')
  with esinvP
  have ts-inv P (upd-invs I P {|ta|}_t) ((redT-updTs (thr s) {|ta|}_t)(t ↦ (x', redT-updLns (locks s) t
no-wait-locks {|ta|}_l))) m'
    by(auto intro: redT-updTs-invariant)
  thus ?thesis using normal by simp
qed

theorem RedT-invariant:
  assumes RedT: s −▷ttas→* s'
  and esinvQ: ts-inv P I (thr s) (shr s)
  shows ts-inv P (upd-invs I P (concat (map (thr-a ∘ snd) ttas))) (thr s') (shr s')
using RedT esinvQ
proof(induct rule: RedT-induct)
  case refl thus ?case by(simp (no-asm))
next
```

case (*step S TTAS S′ T TA S″*)
note *IH* = ‹*ts-inv P I* (*thr S*) (*shr S*) ⟹ *ts-inv P* (*upd-invs I P* (*concat* (*map* (*thr-a* ∘ *snd*) *TTAS*))) (*thr S′*) (*shr S′*)›
with ‹*ts-inv P I* (*thr S*) (*shr S*)›
have *ts-inv P* (*upd-invs I P* (*concat* (*map* (*thr-a* ∘ *snd*) *TTAS*))) (*thr S′*) (*shr S′*) **by** *blast*
with ‹*S′* −*T*▷*TA*→ *S″*›
have *ts-inv P* (*upd-invs* (*upd-invs I P* (*concat* (*map* (*thr-a* ∘ *snd*) *TTAS*))) *P* {|*TA*|}$_t$) (*thr S″*) (*shr S″*)
by(*rule redT-invariant*)
thus *?case* **by**(*simp add*: *comp-def*)
qed

lemma *invariant3p-ts-inv*: *invariant3p redT* {*s*. ∃ *I*. *ts-inv P I* (*thr s*) (*shr s*)}
by(*auto intro*!: *invariant3pI dest*: *redT-invariant*)

end

locale *lifting-wf* = *multithreaded final r convert-RA*
for *final* :: ′*x* ⇒ *bool*
and *r* :: (′*l*,′*t*,′*x*,′*m*,′*w*,′*o*) *semantics* (‹- ⊢ - −-→ -› [*50*,*0*,*0*,*50*] *80*)
and *convert-RA* :: ′*l released-locks* ⇒ ′*o list*
+
fixes *P* :: ′*t* ⇒ ′*x* ⇒ ′*m* ⇒ *bool*
assumes *preserves-red*: ⟦ *t* ⊢ ⟨*x*, *m*⟩ −*ta*→ ⟨*x′*, *m′*⟩; *P t x m* ⟧ ⟹ *P t x′ m′*
and *preserves-NewThread*: ⟦ *t* ⊢ ⟨*x*, *m*⟩ −*ta*→ ⟨*x′*, *m′*⟩; *P t x m*; *NewThread t″ x″ m′* ∈ *set* {|*ta*|}$_t$ ⟧
⟹ *P t″ x″ m′*
and *preserves-other*: ⟦ *t* ⊢ ⟨*x*, *m*⟩ −*ta*→ ⟨*x′*, *m′*⟩; *P t x m*; *P t″ x″ m* ⟧ ⟹ *P t″ x″ m′*
begin

lemma *lifting-inv*: *lifting-inv final r* (λ- :: *unit*. *P*)
by(*unfold-locales*)(*blast intro*: *preserves-red preserves-NewThread preserves-other*)+

lemma *redT-updTs-preserves*:
fixes *ln*
assumes *esokQ*: *ts-ok P ts m*
and *red*: *t* ⊢ ⟨*x*, *m*⟩ −*ta*→ ⟨*x′*, *m′*⟩
and *ts t* = ⌊(*x*, *ln*)⌋
and *thread-oks ts* {|*ta*|}$_t$
shows *ts-ok P* ((*redT-updTs ts* {|*ta*|}$_t$)(*t* ↦ (*x′*, *ln′*))) *m′*
proof −
interpret *lifting-inv final r convert-RA* λ- :: *unit*. *P* **by**(*rule lifting-inv*)
from *esokQ* **obtain** *I* :: ′*t* ⇀ *unit* **where** *ts-inv* (λ-. *P*) *I ts m* **by**(*rule ts-ok-into-ts-inv-const*)
hence *ts-inv* (λ-. *P*) (*upd-invs I* (λ-. *P*) {|*ta*|}$_t$) ((*redT-updTs ts* {|*ta*|}$_t$)(*t* ↦ (*x′*, *ln′*))) *m′*
using *red* ‹*thread-oks ts* {|*ta*|}$_t$› ‹*ts t* = ⌊(*x*, *ln*)⌋› **by**(*rule redT-updTs-invariant*)
thus *?thesis* **by**(*rule ts-inv-const-into-ts-ok*)
qed

theorem *redT-preserves*:
assumes *redT*: *s* −*t*▷*ta*→ *s′*
and *esokQ*: *ts-ok P* (*thr s*) (*shr s*)
shows *ts-ok P* (*thr s′*) (*shr s′*)
proof −
interpret *lifting-inv final r convert-RA* λ- :: *unit*. *P* **by**(*rule lifting-inv*)
from *esokQ* **obtain** *I* :: ′*t* ⇀ *unit* **where** *ts-inv* (λ-. *P*) *I* (*thr s*) (*shr s*) **by**(*rule ts-ok-into-ts-inv-const*)

with *redT* **have** *ts-inv* (λ-. *P*) (*upd-invs I* (λ-. *P*) {|*ta*|}$_t$) (*thr s'*) (*shr s'*) **by**(*rule redT-invariant*)
thus *?thesis* **by**(*rule ts-inv-const-into-ts-ok*)
qed

theorem *RedT-preserves*:
⟦ *s* −▷*ttas*→* *s'*; *ts-ok P* (*thr s*) (*shr s*) ⟧ ⟹ *ts-ok P* (*thr s'*) (*shr s'*)
by(*erule* (*1*) *RedT-lift-preserveD*)(*fastforce elim*: *redT-preserves*)

lemma *invariant3p-ts-ok*: *invariant3p redT* {*s*. *ts-ok P* (*thr s*) (*shr s*)}
by(*auto intro*!: *invariant3pI intro*: *redT-preserves*)

end

lemma *lifting-wf-Const* [*intro*!]:
assumes *multithreaded final r*
shows *lifting-wf final r* (λ*t x m*. *k*)
proof −
interpret *multithreaded final r* **using** *assms* **.**
show *?thesis* **by** *unfold-locales blast*+
qed

end

1.21 Synthetic first and last actions for each thread

theory *FWInitFinLift*
imports
FWLTS
FWLiftingSem
begin

datatype *status* =
PreStart
| *Running*
| *Finished*

abbreviation *convert-TA-initial* :: ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* ⇒ ('*l*,'*t*,*status* × '*x*,'*m*,'*w*,'*o*) *thread-action*
where *convert-TA-initial* == *convert-extTA* (*Pair PreStart*)

lemma *convert-obs-initial-convert-TA-initial*:
convert-obs-initial (*convert-TA-initial ta*) = *convert-TA-initial* (*convert-obs-initial ta*)
by(*simp add*: *convert-obs-initial-def*)

lemma *convert-TA-initial-inject* [*simp*]:
convert-TA-initial ta = *convert-TA-initial ta'* ⟷ *ta* = *ta'*
by(*cases ta*)(*cases ta'*, *auto*)

context *final-thread* **begin**

primrec *init-fin-final* :: *status* × '*x* ⇒ *bool*
where *init-fin-final* (*status*, *x*) ⟷ *status* = *Finished* ∧ *final x*

end

context *multithreaded-base* **begin**

inductive *init-fin* :: ('*l*,'*t*,*status* × '*x*,'*m*,'*w*,'*o action*) *semantics* (‹- ⊢ - -→*i* -› [*50*,*0*,*0*,*51*] *51*)
where
NormalAction:
t ⊢ ⟨*x*, *m*⟩ −*ta*→ ⟨*x'*, *m'*⟩
⟹ *t* ⊢ ((*Running*, *x*), *m*) −*convert-TA-initial* (*convert-obs-initial ta*)→*i* ((*Running*, *x'*), *m'*)

| *InitialThreadAction*:
t ⊢ ((*PreStart*, *x*), *m*) −{|*InitialThreadAction*|}→*i* ((*Running*, *x*), *m*)

| *ThreadFinishAction*:
final x ⟹ *t* ⊢ ((*Running*, *x*), *m*) −{|*ThreadFinishAction*|}→*i* ((*Finished*, *x*), *m*)

end

declare *split-paired-Ex* [*simp del*]

inductive-simps (**in** *multithreaded-base*) *init-fin-simps* [*simp*]:
t ⊢ ((*Finished*, *x*), *m*) −*ta*→*i xm'*
t ⊢ ((*PreStart*, *x*), *m*) −*ta*→*i xm'*
t ⊢ ((*Running*, *x*), *m*) −*ta*→*i xm'*
t ⊢ *xm* −*ta*→*i* ((*Finished*, *x'*), *m'*)
t ⊢ *xm* −*ta*→*i* ((*Running*, *x'*), *m'*)
t ⊢ *xm* −*ta*→*i* ((*PreStart*, *x'*), *m'*)

declare *split-paired-Ex* [*simp*]

context *multithreaded* **begin**

lemma *multithreaded-init-fin*: *multithreaded init-fin-final init-fin*
by(*unfold-locales*)(*fastforce simp add*: *init-fin.simps convert-obs-initial-def ta-upd-simps dest*: *new-thread-memory*)+

end

locale *if-multithreaded-base* = *multithreaded-base* +
constrains *final* :: '*x* ⇒ *bool*
and *r* :: ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *semantics*
and *convert-RA* :: '*l released-locks* ⇒ '*o list*

sublocale *if-multithreaded-base* < *if*: *multithreaded-base*
init-fin-final
init-fin
map NormalAction ∘ *convert-RA*
.

locale *if-multithreaded* = *if-multithreaded-base* + *multithreaded* +
constrains *final* :: '*x* ⇒ *bool*
and *r* :: ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *semantics*
and *convert-RA* :: '*l released-locks* ⇒ '*o list*

sublocale *if-multithreaded* < *if*: *multithreaded*
init-fin-final

init-fin
map NormalAction ∘ *convert-RA*
by(*rule multithreaded-init-fin*)

context *τmultithreaded* **begin**

inductive *init-fin-τmove* :: ('*l*,'*t*,*status* × '*x*,'*m*,'*w*,'*o action*) *τmoves*
where
τmove (*x*, *m*) *ta* (*x'*, *m'*)
⟹ *init-fin-τmove* ((*Running*, *x*), *m*) (*convert-TA-initial* (*convert-obs-initial ta*)) ((*Running*, *x'*), *m'*)

lemma *init-fin-τmove-simps* [*simp*]:
init-fin-τmove ((*PreStart*, *x*), *m*) *ta x'm'* = *False*
init-fin-τmove xm ta ((*PreStart*, *x'*), *m'*) = *False*
init-fin-τmove ((*Running*, *x*), *m*) *ta* ((*s*, *x'*), *m'*) ⟷
(∃ *ta'*. *ta* = *convert-TA-initial* (*convert-obs-initial ta'*) ∧ *s* = *Running* ∧ *τmove* (*x*, *m*) *ta'* (*x'*, *m'*))
init-fin-τmove ((*s*, *x*), *m*) *ta* ((*Running*, *x'*), *m'*) ⟷
s = *Running* ∧ (∃ *ta'*. *ta* = *convert-TA-initial* (*convert-obs-initial ta'*) ∧ *τmove* (*x*, *m*) *ta'* (*x'*, *m'*))
init-fin-τmove ((*Finished*, *x*), *m*) *ta x'm'* = *False*
init-fin-τmove xm ta ((*Finished*, *x'*), *m'*) = *False*
by(*simp-all add*: *init-fin-τmove.simps*)

lemma *init-fin-silent-move-RunningI*:
assumes *silent-move t* (*x*, *m*) (*x'*, *m'*)
shows *τtrsys.silent-move* (*init-fin t*) *init-fin-τmove* ((*Running*, *x*), *m*) ((*Running*, *x'*), *m'*)
using *assms* **by**(*cases*)(*auto intro*: *τtrsys.silent-move.intros init-fin.NormalAction*)

lemma *init-fin-silent-moves-RunningI*:
assumes *silent-moves t* (*x*, *m*) (*x'*, *m'*)
shows *τtrsys.silent-moves* (*init-fin t*) *init-fin-τmove* ((*Running*, *x*), *m*) ((*Running*, *x'*), *m'*)
using *assms*
by(*induct rule*: *rtranclp-induct2*)(*auto elim*: *rtranclp.rtrancl-into-rtrancl intro*: *init-fin-silent-move-RunningI*)

lemma *init-fin-silent-moveD*:
assumes *τtrsys.silent-move* (*init-fin t*) *init-fin-τmove* ((*s*, *x*), *m*) ((*s'*, *x'*), *m'*)
shows *silent-move t* (*x*, *m*) (*x'*, *m'*) ∧ *s* = *s'* ∧ *s'* = *Running*
using *assms* **by**(*auto elim*!: *τtrsys.silent-move.cases init-fin.cases*)

lemma *init-fin-silent-movesD*:
assumes *τtrsys.silent-moves* (*init-fin t*) *init-fin-τmove* ((*s*, *x*), *m*) ((*s'*, *x'*), *m'*)
shows *silent-moves t* (*x*, *m*) (*x'*, *m'*) ∧ *s* = *s'*
using *assms*
by(*induct* ((*s*, *x*), *m*) ((*s'*, *x'*), *m'*) *arbitrary*: *s' x' m'*)
(*auto 7 2 simp only*: *dest*!: *init-fin-silent-moveD intro*: *rtranclp.rtrancl-into-rtrancl*)

lemma *init-fin-τdivergeD*:
assumes *τtrsys.τdiverge* (*init-fin t*) *init-fin-τmove* ((*status*, *x*), *m*)
shows *τdiverge t* (*x*, *m*) ∧ *status* = *Running*
proof
from *assms* **show** *status* = *Running*
by(*cases rule*: *τtrsys.τdiverge.cases*[*consumes 1*])(*auto dest*: *init-fin-silent-moveD*)
moreover define *xm* **where** *xm* = (*x*, *m*)
ultimately have ∃ *x m*. *xm* = (*x*, *m*) ∧ *τtrsys.τdiverge* (*init-fin t*) *init-fin-τmove* ((*Running*, *x*), *m*)

```
    using assms by blast
  thus τdiverge t xm
  proof(coinduct)
    case (τdiverge xm)
    then obtain x m
      where diverge: τtrsys.τdiverge (init-fin t) init-fin-τmove ((Running, x), m)
      and xm: xm = (x, m) by blast
    thus ?case
      by(cases rule:τtrsys.τdiverge.cases[consumes 1])(auto dest!: init-fin-silent-moveD)
  qed
qed

lemma init-fin-τdiverge-RunningI:
  assumes τdiverge t (x, m)
  shows τtrsys.τdiverge (init-fin t) init-fin-τmove ((Running, x), m)
proof −
  define sxm where sxm = ((Running, x), m)
  with assms have ∃ x m. τdiverge t (x, m) ∧ sxm = ((Running, x), m) by blast
  thus τtrsys.τdiverge (init-fin t) init-fin-τmove sxm
  proof(coinduct rule: τtrsys.τdiverge.coinduct[consumes 1, case-names τdiverge])
    case (τdiverge sxm)
    then obtain x m where τdiverge t (x, m) and sxm = ((Running, x), m) by blast
    thus ?case by(cases)(auto intro: init-fin-silent-move-RunningI)
  qed
qed

lemma init-fin-τdiverge-conv:
  τtrsys.τdiverge (init-fin t) init-fin-τmove ((status, x), m) ⟷
   τdiverge t (x, m) ∧ status = Running
by(blast intro: init-fin-τdiverge-RunningI dest: init-fin-τdivergeD)

end

lemma init-fin-τmoves-False:
  τmultithreaded.init-fin-τmove (λ- - -. False) = (λ- - -. False)
by(simp add: fun-eq-iff τmultithreaded.init-fin-τmove.simps)

locale if-τmultithreaded = if-multithreaded-base + τmultithreaded +
  constrains final :: 'x ⇒ bool
  and r :: ('l,'t,'x,'m,'w,'o) semantics
  and convert-RA :: 'l released-locks ⇒ 'o list
  and τmove :: ('l,'t,'x,'m,'w,'o) τmoves

sublocale if-τmultithreaded < if: τmultithreaded
  init-fin-final
  init-fin
  map NormalAction ∘ convert-RA
  init-fin-τmove
.

locale if-τmultithreaded-wf = if-multithreaded-base + τmultithreaded-wf +
  constrains final :: 'x ⇒ bool
  and r :: ('l,'t,'x,'m,'w,'o) semantics
  and convert-RA :: 'l released-locks ⇒ 'o list
```

and $\tau move :: ('l,'t,'x,'m,'w,'o)\ \tau moves$

sublocale *if-τmultithreaded-wf* < *if-multithreaded*
by *unfold-locales*

sublocale *if-τmultithreaded-wf* < *if-τmultithreaded* **.**

context *τmultithreaded-wf* **begin**

lemma *τmultithreaded-wf-init-fin*:
τmultithreaded-wf init-fin-final init-fin init-fin-τmove
proof −
interpret *if*: *multithreaded init-fin-final init-fin map NormalAction ∘ convert-RA*
by(*rule multithreaded-init-fin*)
show *?thesis*
proof(*unfold-locales*)
fix *t x m ta x' m'*
assume *init-fin-τmove* (x, m) *ta* (x', m') $t \vdash (x, m) -ta\rightarrow i\ (x', m')$
thus $m = m'$ **by**(*cases*)(*auto dest*: *τmove-heap*)
next
fix *s ta s'*
assume *init-fin-τmove s ta s'*
thus $ta = \varepsilon$ **by**(*cases*)(*auto dest*: *silent-tl*)
qed
qed

end

sublocale *if-τmultithreaded-wf* < *if*: *τmultithreaded-wf*
init-fin-final
init-fin
map NormalAction ∘ convert-RA
init-fin-τmove
by(*rule τmultithreaded-wf-init-fin*)

primrec *init-fin-lift-inv* :: $('i \Rightarrow 't \Rightarrow 'x \Rightarrow 'm \Rightarrow bool) \Rightarrow 'i \Rightarrow 't \Rightarrow status \times 'x \Rightarrow 'm \Rightarrow bool$
where *init-fin-lift-inv P I t* $(s, x) = P\ I\ t\ x$

context *lifting-inv* **begin**

lemma *lifting-inv-init-fin-lift-inv*:
lifting-inv init-fin-final init-fin (*init-fin-lift-inv P*)
proof −
interpret *if*: *multithreaded init-fin-final init-fin map NormalAction ∘ convert-RA*
by(*rule multithreaded-init-fin*)
show *?thesis*
by(*unfold-locales*)(*fastforce elim*!: *init-fin.cases dest*: *invariant-red invariant-NewThread invariant-other*)+
qed

end

locale *if-lifting-inv* =

if-multithreaded +
lifting-inv +
constrains *final* :: $'x \Rightarrow$ *bool*
and *r* :: $('l,'t,'x,'m,'w,'o)$ *semantics*
and *convert-RA* :: *'l released-locks* $\Rightarrow$ *'o list*
and *P* :: $'i \Rightarrow 't \Rightarrow 'x \Rightarrow 'm \Rightarrow$ *bool*

sublocale *if-lifting-inv* < *if*: *lifting-inv*
init-fin-final
init-fin
map NormalAction ∘ *convert-RA*
init-fin-lift-inv P
by(*rule lifting-inv-init-fin-lift-inv*)

primrec *init-fin-lift* :: $('t \Rightarrow 'x \Rightarrow 'm \Rightarrow$ *bool*$) \Rightarrow 't \Rightarrow$ *status* $\times 'x \Rightarrow 'm \Rightarrow$ *bool*
where *init-fin-lift P t* $(s, x) =$ *P t x*

context *lifting-wf* **begin**

lemma *lifting-wf-init-fin-lift*:
lifting-wf init-fin-final init-fin (*init-fin-lift P*)
proof −
interpret *if*: *multithreaded init-fin-final init-fin map NormalAction* ∘ *convert-RA*
by(*rule multithreaded-init-fin*)
show *?thesis*
by(*unfold-locales*)(*fastforce elim*!: *init-fin.cases dest*: *dest*: *preserves-red preserves-other preserves-NewThread*)+
qed

end

locale *if-lifting-wf* =
if-multithreaded +
lifting-wf +
constrains *final* :: $'x \Rightarrow$ *bool*
and *r* :: $('l,'t,'x,'m,'w,'o)$ *semantics*
and *convert-RA* :: *'l released-locks* $\Rightarrow$ *'o list*
and *P* :: $'t \Rightarrow 'x \Rightarrow 'm \Rightarrow$ *bool*

sublocale *if-lifting-wf* < *if*: *lifting-wf*
init-fin-final
init-fin
map NormalAction ∘ *convert-RA*
init-fin-lift P
by(*rule lifting-wf-init-fin-lift*)

lemma (**in** *if-lifting-wf*) *if-lifting-inv*:
if-lifting-inv final r (λ-::*unit*. *P*)
proof −
interpret *lifting-inv final r convert-RA* λ- :: *unit*. *P* **by**(*rule lifting-inv*)
show *?thesis* **by** *unfold-locales*
qed

locale τ*lifting-inv* = τ*multithreaded-wf* +
lifting-inv +

constrains *final* :: $'x \Rightarrow bool$
and r :: $('l,'t,'x,'m,'w,'o)$ *semantics*
and *convert-RA* :: $'l$ *released-locks* $\Rightarrow$ $'o$ *list*
and $\tau move$ :: $('l,'t,'x,'m,'w,'o)$ $\tau moves$
and P :: $'i \Rightarrow 't \Rightarrow 'x \Rightarrow 'm \Rightarrow bool$
begin

lemma *redT-silent-move-invariant*:
$[\![\tau mredT\ s\ s'; ts\text{-}inv\ P\ Is\ (thr\ s)\ (shr\ s)]\!] \Longrightarrow ts\text{-}inv\ P\ Is\ (thr\ s')\ (shr\ s')$
by(*auto dest*!: *redT-invariant mτmove-silentD*)

lemma *redT-silent-moves-invariant*:
$[\![mthr.silent\text{-}moves\ s\ s'; ts\text{-}inv\ P\ Is\ (thr\ s)\ (shr\ s)]\!] \Longrightarrow ts\text{-}inv\ P\ Is\ (thr\ s')\ (shr\ s')$
by(*induct rule*: *rtranclp-induct*)(*auto dest*: *redT-silent-move-invariant*)

lemma *redT-τrtrancl3p-invariant*:
$[\![mthr.\tau rtrancl3p\ s\ ttas\ s'; ts\text{-}inv\ P\ Is\ (thr\ s)\ (shr\ s)]\!]$
$\Longrightarrow ts\text{-}inv\ P\ (upd\text{-}invs\ Is\ P\ (concat\ (map\ (thr\text{-}a \circ snd)\ ttas)))\ (thr\ s')\ (shr\ s')$
proof(*induct arbitrary*: *Is rule*: *mthr.τrtrancl3p.induct*)
case *τrtrancl3p-refl* **thus** *?case* **by** *simp*
next
case (*τrtrancl3p-step s s' tls s'' tl*)
thus *?case* **by**(*cases tl*)(*force dest*: *redT-invariant*)
next
case (*τrtrancl3p-τstep s s' tls s'' tl*)
thus *?case* **by**(*cases tl*)(*force dest*: *redT-invariant mτmove-silentD*)
qed

end

locale *τlifting-wf* = *τmultithreaded* +
lifting-wf +
constrains *final* :: $'x \Rightarrow bool$
and r :: $('l,'t,'x,'m,'w,'o)$ *semantics*
and *convert-RA* :: $'l$ *released-locks* $\Rightarrow$ $'o$ *list*
and $\tau move$ :: $('l,'t,'x,'m,'w,'o)$ $\tau moves$
and P :: $'t \Rightarrow 'x \Rightarrow 'm \Rightarrow bool$
begin

lemma *redT-silent-move-preserves*:
$[\![\tau mredT\ s\ s'; ts\text{-}ok\ P\ (thr\ s)\ (shr\ s)]\!] \Longrightarrow ts\text{-}ok\ P\ (thr\ s')\ (shr\ s')$
by(*auto dest*: *redT-preserves*)

lemma *redT-silent-moves-preserves*:
$[\![mthr.silent\text{-}moves\ s\ s'; ts\text{-}ok\ P\ (thr\ s)\ (shr\ s)]\!] \Longrightarrow ts\text{-}ok\ P\ (thr\ s')\ (shr\ s')$
by(*induct rule*: *rtranclp.induct*)(*auto dest*: *redT-silent-move-preserves*)

lemma *redT-τrtrancl3p-preserves*:
$[\![mthr.\tau rtrancl3p\ s\ ttas\ s'; ts\text{-}ok\ P\ (thr\ s)\ (shr\ s)]\!] \Longrightarrow ts\text{-}ok\ P\ (thr\ s')\ (shr\ s')$
by(*induct rule*: *mthr.τrtrancl3p.induct*)(*auto dest*: *redT-silent-moves-preserves redT-preserves*)

end

definition *init-fin-lift-state* :: *status* $\Rightarrow$ $('l,'t,'x,'m,'w)$ *state* $\Rightarrow$ $('l,'t,status \times 'x,'m,'w)$ *state*

where *init-fin-lift-state s σ* = (*locks σ*, (λ*t*. *map-option* (λ(*x*, *ln*). ((*s*, *x*), *ln*)) (*thr σ t*), *shr σ*), *wset σ*, *interrupts σ*)

definition *init-fin-descend-thr* :: ('*l*,'*t*,'*status* × '*x*) *thread-info* ⇒ ('*l*,'*t*,'*x*) *thread-info*
where *init-fin-descend-thr ts* = *map-option* (λ((*s*, *x*), *ln*). (*x*, *ln*)) ∘ *ts*

definition *init-fin-descend-state* :: ('*l*,'*t*,'*status* × '*x*,'*m*,'*w*) *state* ⇒ ('*l*,'*t*,'*x*,'*m*,'*w*) *state*
where *init-fin-descend-state σ* = (*locks σ*, (*init-fin-descend-thr* (*thr σ*), *shr σ*), *wset σ*, *interrupts σ*)

lemma *ts-ok-init-fin-lift-init-fin-lift-state* [*simp*]:
ts-ok (*init-fin-lift P*) (*thr* (*init-fin-lift-state s σ*)) (*shr* (*init-fin-lift-state s σ*)) ⟷ *ts-ok P* (*thr σ*) (*shr σ*)
by(*auto simp add*: *init-fin-lift-state-def intro*!: *ts-okI dest*: *ts-okD*)

lemma *ts-inv-init-fin-lift-inv-init-fin-lift-state* [*simp*]:
ts-inv (*init-fin-lift-inv P*) *I* (*thr* (*init-fin-lift-state s σ*)) (*shr* (*init-fin-lift-state s σ*)) ⟷
ts-inv P I (*thr σ*) (*shr σ*)
by(*auto simp add*: *init-fin-lift-state-def intro*!: *ts-invI dest*: *ts-invD*)

lemma *init-fin-lift-state-conv-simps*:
shows *shr-init-fin-lift-state*: *shr* (*init-fin-lift-state s σ*) = *shr σ*
and *locks-init-fin-lift-state*: *locks* (*init-fin-lift-state s σ*) = *locks σ*
and *wset-init-fin-lift-state*: *wset* (*init-fin-lift-state s σ*) = *wset σ*
and *interrupts-init-fin-lift-stae*: *interrupts* (*init-fin-lift-state s σ*) = *interrupts σ*
and *thr-init-fin-list-state*:
thr (*init-fin-lift-state s σ*) *t* = *map-option* (λ(*x*, *ln*). ((*s*, *x*), *ln*)) (*thr σ t*)
by(*simp-all add*: *init-fin-lift-state-def*)

lemma *thr-init-fin-list-state'*:
thr (*init-fin-lift-state s σ*) = *map-option* (λ(*x*, *ln*). ((*s*, *x*), *ln*)) ∘ *thr σ*
by(*simp add*: *fun-eq-iff thr-init-fin-list-state*)

lemma *init-fin-descend-thr-Some-conv* [*simp*]:
⋀*ln*. *ts t* = ⌊((*status*, *x*), *ln*)⌋ ⟹ *init-fin-descend-thr ts t* = ⌊(*x*, *ln*)⌋
by(*simp add*: *init-fin-descend-thr-def*)

lemma *init-fin-descend-thr-None-conv* [*simp*]:
ts t = *None* ⟹ *init-fin-descend-thr ts t* = *None*
by(*simp add*: *init-fin-descend-thr-def*)

lemma *init-fin-descend-thr-eq-None* [*simp*]:
init-fin-descend-thr ts t = *None* ⟷ *ts t* = *None*
by(*simp add*: *init-fin-descend-thr-def*)

lemma *init-fin-descend-state-simps* [*simp*]:
init-fin-descend-state (*ls*, (*ts*, *m*), *ws*, *is*) = (*ls*, (*init-fin-descend-thr ts*, *m*), *ws*, *is*)
locks (*init-fin-descend-state s*) = *locks s*
thr (*init-fin-descend-state s*) = *init-fin-descend-thr* (*thr s*)
shr (*init-fin-descend-state s*) = *shr s*
wset (*init-fin-descend-state s*) = *wset s*
interrupts (*init-fin-descend-state s*) = *interrupts s*
by(*simp-all add*: *init-fin-descend-state-def*)

lemma *init-fin-descend-thr-update* [*simp*]:

init-fin-descend-thr (*ts*(*t* := *v*)) = (*init-fin-descend-thr ts*)(*t* := *map-option* (λ((*status*, *x*), *ln*). (*x*, *ln*)) *v*)
by(*simp add*: *init-fin-descend-thr-def fun-eq-iff*)

lemma *ts-ok-init-fin-descend-state*:
ts-ok P (*init-fin-descend-thr ts*) = *ts-ok* (*init-fin-lift P*) *ts*
by(*rule ext*)(*auto 4 3 intro*!: *ts-okI dest*: *ts-okD simp add*: *init-fin-descend-thr-def*)

lemma *free-thread-id-init-fin-descend-thr* [*simp*]:
free-thread-id (*init-fin-descend-thr ts*) = *free-thread-id ts*
by(*simp add*: *free-thread-id.simps fun-eq-iff*)

lemma *redT-updT'-init-fin-descend-thr-eq-None* [*simp*]:
redT-updT' (*init-fin-descend-thr ts*) *nt t* = *None* ⟷ *redT-updT' ts nt t* = *None*
by(*cases nt*) *simp-all*

lemma *thread-ok-init-fin-descend-thr* [*simp*]:
thread-ok (*init-fin-descend-thr ts*) *nta* = *thread-ok ts nta*
by(*cases nta*) *simp-all*

lemma *threads-ok-init-fin-descend-thr* [*simp*]:
thread-oks (*init-fin-descend-thr ts*) *ntas* = *thread-oks ts ntas*
by(*induct ntas arbitrary*: *ts*)(*auto elim*!: *thread-oks-ts-change*[*THEN iffD1*, *rotated 1*])

lemma *init-fin-descend-thr-redT-updT* [*simp*]:
init-fin-descend-thr (*redT-updT ts* (*convert-new-thread-action* (*Pair status*) *nt*)) =
redT-updT (*init-fin-descend-thr ts*) *nt*
by(*cases nt*) *simp-all*

lemma *init-fin-descend-thr-redT-updTs* [*simp*]:
init-fin-descend-thr (*redT-updTs ts* (*map* (*convert-new-thread-action* (*Pair status*)) *nts*)) =
redT-updTs (*init-fin-descend-thr ts*) *nts*
by(*induct nts arbitrary*: *ts*) *simp-all*

context *final-thread* **begin**

lemma *cond-action-ok-init-fin-descend-stateI* [*simp*]:
final-thread.cond-action-ok init-fin-final s t ct ⟹ *cond-action-ok* (*init-fin-descend-state s*) *t ct*
by(*cases ct*)(*auto simp add*: *final-thread.cond-action-ok.simps init-fin-descend-thr-def*)

lemma *cond-action-oks-init-fin-descend-stateI* [*simp*]:
final-thread.cond-action-oks init-fin-final s t cts ⟹ *cond-action-oks* (*init-fin-descend-state s*) *t cts*
by(*induct cts*)(*simp-all add*: *final-thread.cond-action-oks.simps cond-action-ok-init-fin-descend-stateI*)

end

definition *lift-start-obs* :: *'t* ⇒ *'o list* ⇒ (*'t* × *'o action*) *list*
where *lift-start-obs t obs* = (*t*, *InitialThreadAction*) # *map* (λ*ob*. (*t*, *NormalAction ob*)) *obs*

lemma *length-lift-start-obs* [*simp*]: *length* (*lift-start-obs t obs*) = *Suc* (*length obs*)
by(*simp add*: *lift-start-obs-def*)

lemma *set-lift-start-obs* [*simp*]:

set (*lift-start-obs t obs*) =
insert (*t*, *InitialThreadAction*) ((*Pair t* ∘ *NormalAction*) ‘ *set obs*)
by(*auto simp add*: *lift-start-obs-def o-def*)

lemma *distinct-lift-start-obs* [*simp*]: *distinct* (*lift-start-obs t obs*) = *distinct obs*
by(*auto simp add*: *lift-start-obs-def distinct-map intro*: *inj-onI*)

end
theory *FWBisimLift* **imports**
FWInitFinLift
FWBisimulation
begin

context *FWbisimulation-base* **begin**

inductive *init-fin-bisim* :: *'t* ⇒ ((*status* × *'x1*) × *'m1*, (*status* × *'x2*) × *'m2*) *bisim*
(‹- ⊢ - ≈*i* -›[*50*,*50*,*50*] *60*)
for *t* :: *'t*
where
PreStart: *t* ⊢ (*x1*, *m1*) ≈ (*x2*, *m2*) ⟹ *t* ⊢ ((*PreStart*, *x1*), *m1*) ≈*i* ((*PreStart*, *x2*), *m2*)
| *Running*: *t* ⊢ (*x1*, *m1*) ≈ (*x2*, *m2*) ⟹ *t* ⊢ ((*Running*, *x1*), *m1*) ≈*i* ((*Running*, *x2*), *m2*)
| *Finished*:
⟦ *t* ⊢ (*x1*, *m1*) ≈ (*x2*, *m2*); *final1 x1*; *final2 x2* ⟧
⟹ *t* ⊢ ((*Finished*, *x1*), *m1*) ≈*i* ((*Finished*, *x2*), *m2*)

definition *init-fin-bisim-wait* :: (*status* × *'x1*, *status* × *'x2*) *bisim* (‹- ≈*iw* -› [*50*,*50*] *60*)
where
init-fin-bisim-wait = (λ(*status1*, *x1*) (*status2*, *x2*). *status1* = *Running* ∧ *status2* = *Running* ∧ *x1* ≈*w x2*)

inductive-simps *init-fin-bisim-simps* [*simp*]:
t ⊢ ((*PreStart*, *x1*), *m1*) ≈*i* ((*s2*, *x2*), *m2*)
t ⊢ ((*Running*, *x1*), *m1*) ≈*i* ((*s2*, *x2*), *m2*)
t ⊢ ((*Finished*, *x1*), *m1*) ≈*i* ((*s2*, *x2*), *m2*)
t ⊢ ((*s1*, *x1*), *m1*) ≈*i* ((*PreStart*, *x2*), *m2*)
t ⊢ ((*s1*, *x1*), *m1*) ≈*i* ((*Running*, *x2*), *m2*)
t ⊢ ((*s1*, *x1*), *m1*) ≈*i* ((*Finished*, *x2*), *m2*)

lemma *init-fin-bisim-iff*:
t ⊢ ((*s1*, *x1*), *m1*) ≈*i* ((*s2*, *x2*), *m2*) ⟷
s1 = *s2* ∧ *t* ⊢ (*x1*, *m1*) ≈ (*x2*, *m2*) ∧ (*s2* = *Finished* ⟶ *final1 x1* ∧ *final2 x2*)
by(*cases s1*) *auto*

lemma *nta-bisim-init-fin-bisim* [*simp*]:
nta-bisim init-fin-bisim (*convert-new-thread-action* (*Pair PreStart*) *nt1*)
(*convert-new-thread-action* (*Pair PreStart*) *nt2*) =
nta-bisim bisim nt1 nt2
by(*cases nt1*) *simp-all*

lemma *ta-bisim-init-fin-bisim-convert* [*simp*]:
ta-bisim init-fin-bisim (*convert-TA-initial* (*convert-obs-initial ta1*)) (*convert-TA-initial* (*convert-obs-initial ta2*)) ⟷ *ta1* ∼*m ta2*
by(*auto simp add*: *ta-bisim-def list-all2-map1 list-all2-map2*)

lemma *ta-bisim-init-fin-bisim-InitialThreadAction* [*simp*]:
ta-bisim init-fin-bisim {|*InitialThreadAction*|} {|*InitialThreadAction*|}
by(*simp add*: *ta-bisim-def*)

lemma *ta-bisim-init-fin-bisim-ThreadFinishAction* [*simp*]:
ta-bisim init-fin-bisim {|*ThreadFinishAction*|} {|*ThreadFinishAction*|}
by(*simp add*: *ta-bisim-def*)

lemma *init-fin-bisim-wait-simps* [*simp*]:
(*status1*, *x1*) ≈*iw* (*status2*, *x2*) ⟷ *status1* = *Running* ∧ *status2* = *Running* ∧ *x1* ≈*w x2*
by(*simp add*: *init-fin-bisim-wait-def*)

lemma *init-fin-lift-state-mbisimI*:
s ≈*m s'* ⟹
FWbisimulation-base.mbisim init-fin-bisim init-fin-bisim-wait (*init-fin-lift-state Running s*) (*init-fin-lift-state Running s'*)
apply(*rule FWbisimulation-base.mbisimI*)
apply(*simp add*: *thr-init-fin-list-state' o-def dom-map-option mbisim-finite1*)
apply(*simp add*: *locks-init-fin-lift-state mbisim-def*)
apply(*simp add*: *wset-init-fin-lift-state mbisim-def*)
apply(*simp add*: *interrupts-init-fin-lift-stae mbisim-def*)
apply(*clarsimp simp add*: *wset-init-fin-lift-state mbisim-def thr-init-fin-list-state' o-def wset-thread-ok-conv-dom dom-map-option del*: *subsetI*)
apply(*drule-tac t*=*t* **in** *mbisim-thrNone-eq*)
apply(*simp add*: *thr-init-fin-list-state*)
apply(*clarsimp simp add*: *thr-init-fin-list-state shr-init-fin-lift-state wset-init-fin-lift-state init-fin-bisim-iff*)
apply(*frule* (*1*) *mbisim-thrD1*)
apply(*simp add*: *mbisim-def*)
done

end

context *FWdelay-bisimulation-base* **begin**

lemma *init-fin-delay-bisimulation-final-base*:
delay-bisimulation-final-base (*r1.init-fin t*) (*r2.init-fin t*) (*init-fin-bisim t*)
r1.init-fin-τmove r2.init-fin-τmove (λ(*x1*, *m*). *r1.init-fin-final x1*) (λ(*x2*, *m*). *r2.init-fin-final x2*)
by(*unfold-locales*)(*auto 4 3*)

end

lemma *init-fin-bisim-flip* [*flip-simps*]:
FWbisimulation-base.init-fin-bisim final2 final1 (λ*t*. *flip* (*bisim t*)) =
(λ*t*. *flip* (*FWbisimulation-base.init-fin-bisim final1 final2 bisim t*))
by(*auto simp only*: *FWbisimulation-base.init-fin-bisim-iff flip-simps fun-eq-iff split-paired-Ex*)

lemma *init-fin-bisim-wait-flip* [*flip-simps*]:
FWbisimulation-base.init-fin-bisim-wait (*flip bisim-wait*) =
flip (*FWbisimulation-base.init-fin-bisim-wait bisim-wait*)
by(*auto simp add*: *fun-eq-iff FWbisimulation-base.init-fin-bisim-wait-simps flip-simps*)

context *FWdelay-bisimulation-lift-aux* **begin**

lemma *init-fin-FWdelay-bisimulation-lift-aux*:

FWdelay-bisimulation-lift-aux r1.init-fin-final r1.init-fin r2.init-fin-final r2.init-fin r1.init-fin-τmove r2.init-fin-τmove
by(*intro FWdelay-bisimulation-lift-aux.intro r1.τmultithreaded-wf-init-fin r2.τmultithreaded-wf-init-fin*)

lemma *init-fin-FWdelay-bisimulation-final-base*:
FWdelay-bisimulation-final-base
r1.init-fin-final r1.init-fin r2.init-fin-final r2.init-fin
init-fin-bisim r1.init-fin-τmove r2.init-fin-τmove
by(*intro FWdelay-bisimulation-final-base.intro init-fin-FWdelay-bisimulation-lift-aux FWdelay-bisimulation-final-b*
init-fin-delay-bisimulation-final-base)

end

context *FWdelay-bisimulation-obs* **begin**

lemma *init-fin-simulation1*:
assumes *bisim*: $t \vdash s1 \approx i\ s2$
and *red1*: *r1.init-fin t s1 tl1 s1'*
and $\tau 1$: $\neg$ *r1.init-fin-τmove s1 tl1 s1'*
shows $\exists s2'\ s2''\ tl2.$ (τ*trsys.silent-move* (*r2.init-fin t*) *r2.init-fin-τmove*)** *s2 s2'* $\wedge$
r2.init-fin t s2' tl2 s2'' $\wedge$ $\neg$ *r2.init-fin-τmove s2' tl2 s2''* $\wedge$
$t \vdash s1' \approx i\ s2''$ $\wedge$ *ta-bisim init-fin-bisim tl1 tl2*
proof −
from *bisim* **obtain** *status x1 m1 x2 m2*
where *s1*: $s1 = ((status, x1), m1)$
and *s2*: $s2 = ((status, x2), m2)$
and *bisim*: $t \vdash (x1, m1) \approx (x2, m2)$
and *finished*: $status = Finished \Longrightarrow final1\ x1 \wedge final2\ x2$
by(*cases s1*)(*cases s2*, *fastforce simp add*: *init-fin-bisim-iff*)
from *red1* **show** *?thesis* **unfolding** *s1*
proof(*cases*)
case (*NormalAction ta1 x1' m1'*)
with $\tau 1$ *s1* **have** $\neg$ τ*move1* $(x1, m1)$ *ta1* $(x1', m1')$ **by**(*simp*)
from *simulation1*[*OF bisim* ‹$t \vdash (x1, m1)\ -1-ta1\rightarrow\ (x1', m1')$› *this*]
obtain *x2' m2' x2'' m2'' ta2*
where *red2*: *r2.silent-moves t* $(x2, m2)$ $(x2', m2')$
and *red2'*: $t \vdash (x2', m2')\ -2-ta2\rightarrow\ (x2'', m2'')$
and $\tau 2$: $\neg$ τ*move2* $(x2', m2')$ *ta2* $(x2'', m2'')$
and *bisim'*: $t \vdash (x1', m1') \approx (x2'', m2'')$
and *tasim*: $ta1 \sim m\ ta2$ **by** *auto*
let *?s2'* = $((Running, x2'), m2')$
let *?s2''* = $((Running, x2''), m2'')$
let *?ta2* = (*convert-TA-initial* (*convert-obs-initial ta2*))
from *red2* **have** τ*trsys.silent-moves* (*r2.init-fin t*) *r2.init-fin-τmove s2 ?s2'*
unfolding *s2* ‹*status* = *Running*› **by**(*rule r2.init-fin-silent-moves-RunningI*)
moreover from *red2'* **have** *r2.init-fin t ?s2' ?ta2 ?s2''* **by**(*rule r2.init-fin.NormalAction*)
moreover from $\tau 2$ **have** $\neg$ *r2.init-fin-τmove ?s2' ?ta2 ?s2''* **by** *simp*
moreover from *bisim'* **have** $t \vdash s1' \approx i$ *?s2''***using** ‹$s1' = ((Running, x1'), m1')$› **by** *simp*
moreover from *tasim* ‹*tl1* = *convert-TA-initial* (*convert-obs-initial ta1*)›
have *ta-bisim init-fin-bisim tl1 ?ta2* **by** *simp*
ultimately show *?thesis* **by** *blast*
next
case *InitialThreadAction*
with *s1 s2 bisim* **show** *?thesis* **by**(*auto simp del*: *split-paired-Ex*)

```
  next
    case ThreadFinishAction
    from final1-simulation[OF bisim] ‹final1 x1›
    obtain x2' m2' where red2: r2.silent-moves t (x2, m2) (x2', m2')
      and bisim': t ⊢ (x1, m1) ≈ (x2', m2')
      and fin2: final2 x2' by auto
    let ?s2' = ((Running, x2'), m2')
    let ?s2'' = ((Finished, x2'), m2')
    from red2 have τtrsys.silent-moves (r2.init-fin t) r2.init-fin-τmove s2 ?s2'
      unfolding s2 ‹status = Running› by(rule r2.init-fin-silent-moves-RunningI)
    moreover from fin2 have r2.init-fin t ?s2' ⦃ThreadFinishAction⦄ ?s2'' ..
    moreover have ¬ r2.init-fin-τmove ?s2' ⦃ThreadFinishAction⦄ ?s2'' by simp
    moreover have t ⊢ s1' ≈i ?s2''
      using ‹s1' = ((Finished, x1), m1)› fin2 ‹final1 x1› bisim' by simp
    ultimately show ?thesis unfolding ‹tl1 = ⦃ThreadFinishAction⦄›
      by(blast intro: ta-bisim-init-fin-bisim-ThreadFinishAction)
  qed
qed

lemma init-fin-simulation2:
  ⟦ t ⊢ s1 ≈i s2; r2.init-fin t s2 tl2 s2'; ¬ r2.init-fin-τmove s2 tl2 s2' ⟧
  ⟹ ∃ s1' s1'' tl1. (τtrsys.silent-move (r1.init-fin t) r1.init-fin-τmove)** s1 s1' ∧
          r1.init-fin t s1' tl1 s1'' ∧ ¬ r1.init-fin-τmove s1' tl1 s1'' ∧
          t ⊢ s1'' ≈i s2' ∧ ta-bisim init-fin-bisim tl1 tl2
using FWdelay-bisimulation-obs.init-fin-simulation1[OF FWdelay-bisimulation-obs-flip]
unfolding flip-simps .

lemma init-fin-simulation-Wakeup1:
  assumes bisim: t ⊢ (sx1, m1) ≈i (sx2, m2)
  and wait: sx1 ≈iw sx2
  and red1: r1.init-fin t (sx1, m1) ta1 (sx1', m1')
  and wakeup: Notified ∈ set ⦃ta1⦄w ∨ WokenUp ∈ set ⦃ta1⦄w
  shows ∃ ta2 sx2' m2'. r2.init-fin t (sx2, m2) ta2 (sx2', m2') ∧ t ⊢ (sx1', m1') ≈i (sx2', m2') ∧
                  ta-bisim init-fin-bisim ta1 ta2
proof −
  from bisim wait obtain status x1 x2
    where sx1: sx1 = (status, x1)
    and sx2: sx2 = (status, x2)
    and Bisim: t ⊢ (x1, m1) ≈ (x2, m2)
    and Wait: x1 ≈w x2 by cases auto
  from red1 wakeup sx1 obtain x1' ta1'
    where sx1': sx1' = (Running, x1')
    and status: status = Running
    and Red1: t ⊢ (x1, m1) −1−ta1'→ (x1', m1')
    and ta1: ta1 = convert-TA-initial (convert-obs-initial ta1')
    and Wakeup: Notified ∈ set ⦃ta1'⦄w ∨ WokenUp ∈ set ⦃ta1'⦄w
    by cases auto
  from simulation-Wakeup1[OF Bisim Wait Red1 Wakeup] obtain ta2' x2' m2'
    where red2: t ⊢ (x2, m2) −2−ta2'→ (x2', m2')
    and bisim': t ⊢ (x1', m1') ≈ (x2', m2')
    and tasim: ta1' ∼m ta2' by blast
  let ?sx2' = (Running, x2')
  let ?ta2 = convert-TA-initial (convert-obs-initial ta2')
  from red2 have r2.init-fin t (sx2, m2) ?ta2 (?sx2', m2') unfolding sx2 status ..
```

moreover from *bisim' sx1'* **have** $t \vdash (sx1', m1') \approx i$ (*?sx2'*, *m2'*) **by** *simp*
moreover from *tasim ta1* **have** *ta-bisim init-fin-bisim ta1 ?ta2* **by** *simp*
ultimately show *?thesis* **by** *blast*
qed

lemma *init-fin-simulation-Wakeup2*:
$[\![t \vdash (sx1, m1) \approx i (sx2, m2)$; *sx1* $\approx iw$ *sx2*; *r2.init-fin t* (*sx2*, *m2*) *ta2* (*sx2'*, *m2'*);
Notified $\in$ *set* $\{\!|ta2|\!\}_w \vee$ *WokenUp* $\in$ *set* $\{\!|ta2|\!\}_w]\!]$
$\Longrightarrow \exists$ *ta1 sx1' m1'*. *r1.init-fin t* (*sx1*, *m1*) *ta1* (*sx1'*, *m1'*) $\wedge$ $t \vdash (sx1', m1') \approx i (sx2', m2') \wedge$
ta-bisim init-fin-bisim ta1 ta2
using *FWdelay-bisimulation-obs.init-fin-simulation-Wakeup1*[*OF FWdelay-bisimulation-obs-flip*]
unfolding *flip-simps* **.**

lemma *init-fin-delay-bisimulation-obs*:
delay-bisimulation-obs (*r1.init-fin t*) (*r2.init-fin t*) (*init-fin-bisim t*) (*ta-bisim init-fin-bisim*)
r1.init-fin-τ*move r2.init-fin-*τ*move*
by(*unfold-locales*)(*erule* (*2*) *init-fin-simulation1 init-fin-simulation2*)+

lemma *init-fin-FWdelay-bisimulation-obs*:
FWdelay-bisimulation-obs r1.init-fin-final r1.init-fin r2.init-fin-final r2.init-fin init-fin-bisim init-fin-bisim-wait r1.init-fin-τ*move r2.init-fin-*τ*move*
proof(*intro FWdelay-bisimulation-obs.intro init-fin-FWdelay-bisimulation-final-base FWdelay-bisimulation-obs-axi init-fin-delay-bisimulation-obs*)
fix *t' sx m1 sxx m2 t sx1 sx2 sx1' ta1 sx1'' m1' sx2' ta2 sx2'' m2'*
assume *bisim*: $t' \vdash (sx, m1) \approx i (sxx, m2)$
and *bisim1*: $t \vdash (sx1, m1) \approx i (sx2, m2)$
and *red1*: τ*trsys.silent-moves* (*r1.init-fin t*) *r1.init-fin-*τ*move* (*sx1*, *m1*) (*sx1'*, *m1*)
and *red1'*: *r1.init-fin t* (*sx1'*, *m1*) *ta1* (*sx1''*, *m1'*)
and $\tau 1$: $\neg$ *r1.init-fin-*τ*move* (*sx1'*, *m1*) *ta1* (*sx1''*, *m1'*)
and *red2*: τ*trsys.silent-moves* (*r2.init-fin t*) *r2.init-fin-*τ*move* (*sx2*, *m2*) (*sx2'*, *m2*)
and *red2'*:*r2.init-fin t* (*sx2'*, *m2*) *ta2* (*sx2''*, *m2'*)
and $\tau 2$: $\neg$ *r2.init-fin-*τ*move* (*sx2'*, *m2*) *ta2* (*sx2''*, *m2'*)
and *bisim1'*: $t \vdash (sx1'', m1') \approx i (sx2'', m2')$
and *tasim*: *ta-bisim init-fin-bisim ta1 ta2*
from *bisim* **obtain** *status x xx*
where *sx*:*sx* = (*status*, *x*)
and *sxx*: *sxx* = (*status*, *xx*)
and *Bisim*: $t' \vdash (x, m1) \approx (xx, m2)$
and *Finish*: *status* = *Finished* $\Longrightarrow$ *final1 x* $\wedge$ *final2 xx*
by(*cases sx*)(*cases sxx*, *auto simp add*: *init-fin-bisim-iff*)
from *bisim1* **obtain** *status1 x1 x2*
where *sx1*: *sx1* = (*status1*, *x1*)
and *sx2*: *sx2* = (*status1*, *x2*)
and *Bisim1*: $t \vdash (x1, m1) \approx (x2, m2)$
by(*cases sx1*)(*cases sx2*, *auto simp add*: *init-fin-bisim-iff*)
from *bisim1'* **obtain** *status1' x1'' x2''*
where *sx1''*: *sx1''* = (*status1'*, *x1''*)
and *sx2''*: *sx2''* = (*status1'*, *x2''*)
and *Bisim1'*: $t \vdash (x1'', m1') \approx (x2'', m2')$
by(*cases sx1''*)(*cases sx2''*, *auto simp add*: *init-fin-bisim-iff*)
from *red1 sx1* **obtain** *x1'* **where** *sx1'*: *sx1'* = (*status1*, *x1'*)
and *Red1*: *r1.silent-moves t* (*x1*, *m1*) (*x1'*, *m1*)
by(*cases sx1'*)(*auto dest*: *r1.init-fin-silent-movesD*)
from *red2 sx2* **obtain** *x2'* **where** *sx2'*: *sx2'* = (*status1*, *x2'*)

```
    and Red2: r2.silent-moves t (x2, m2) (x2', m2)
    by(cases sx2')(auto dest: r2.init-fin-silent-movesD)
  show t' ⊢ (sx, m1') ≈i (sxx, m2')
  proof(cases status1 = Running ∧ status1' = Running)
    case True
    with red1' sx1' sx1'' obtain ta1'
      where Red1': t ⊢ (x1', m1) −1−ta1'→ (x1'', m1')
      and ta1: ta1 = convert-TA-initial (convert-obs-initial ta1')
      by cases auto
    from red2' sx2' sx2'' True obtain ta2'
      where Red2': t ⊢ (x2', m2) −2−ta2'→ (x2'', m2')
      and ta2: ta2 = convert-TA-initial (convert-obs-initial ta2')
      by cases auto
    from τ1 sx1' sx1'' ta1 True have τ1':¬ τmove1 (x1', m1) ta1' (x1'', m1') by simp
    from τ2 sx2' sx2'' ta2 True have τ2':¬ τmove2 (x2', m2) ta2' (x2'', m2') by simp
    from tasim ta1 ta2 have ta1' ∼m ta2' by simp
    with Bisim Bisim1 Red1 Red1' τ1' Red2 Red2' τ2' Bisim1'
    have t' ⊢ (x, m1') ≈ (xx, m2') by(rule bisim-inv-red-other)
    with True Finish show ?thesis unfolding sx sxx by(simp add: init-fin-bisim-iff)
  next
    case False
    with red1' sx1' sx1'' have m1' = m1 by cases auto
    moreover from red2' sx2' sx2'' False have m2' = m2 by cases auto
    ultimately show ?thesis using bisim by simp
  qed
next
  fix t sx1 m1 sx2 m2 sx1' ta1 sx1'' m1' sx2' ta2 sx2'' m2' w
  assume bisim: t ⊢ (sx1, m1) ≈i (sx2, m2)
    and red1: τtrsys.silent-moves (r1.init-fin t) r1.init-fin-τmove (sx1, m1) (sx1', m1)
    and red1': r1.init-fin t (sx1', m1) ta1 (sx1'', m1')
    and τ1: ¬ r1.init-fin-τmove (sx1', m1) ta1 (sx1'', m1')
    and red2: τtrsys.silent-moves (r2.init-fin t) r2.init-fin-τmove (sx2, m2) (sx2', m2)
    and red2': r2.init-fin t (sx2', m2) ta2 (sx2'', m2')
    and τ2: ¬ r2.init-fin-τmove (sx2', m2) ta2 (sx2'', m2')
    and bisim': t ⊢ (sx1'', m1') ≈i (sx2'', m2')
    and tasim: ta-bisim init-fin-bisim ta1 ta2
    and suspend1: Suspend w ∈ set {|ta1|}w
    and suspend2: Suspend w ∈ set {|ta2|}w
  from bisim obtain status x1 x2
    where sx1: sx1 = (status, x1)
    and sx2: sx2 = (status, x2)
    and Bisim: t ⊢ (x1, m1) ≈ (x2, m2)
    by(cases sx1)(cases sx2, auto simp add: init-fin-bisim-iff)
  from bisim' obtain status' x1'' x2''
    where sx1'': sx1'' = (status', x1'')
    and sx2'': sx2'' = (status', x2'')
    and Bisim': t ⊢ (x1'', m1') ≈ (x2'', m2')
     by(cases sx1'')(cases sx2'', auto simp add: init-fin-bisim-iff)
  from red1 sx1 obtain x1' where sx1': sx1' = (status, x1')
    and Red1: r1.silent-moves t (x1, m1) (x1', m1)
    by(cases sx1')(auto dest: r1.init-fin-silent-movesD)
  from red2 sx2 obtain x2' where sx2': sx2' = (status, x2')
    and Red2: r2.silent-moves t (x2, m2) (x2', m2)
    by(cases sx2')(auto dest: r2.init-fin-silent-movesD)
```

from *red1′ sx1′ sx1″ suspend1* **obtain** *ta1′*
where *Red1′*: $t \vdash (x1', m1) -1-ta1'\rightarrow (x1'', m1')$
and *ta1*: *ta1 = convert-TA-initial (convert-obs-initial ta1′)*
and *Suspend1*: *Suspend w ∈ set ⦃ta1′⦄w*
and *status*: *status = Running status′ = Running* **by** *cases auto*
from *red2′ sx2′ sx2″ suspend2* **obtain** *ta2′*
where *Red2′*: $t \vdash (x2', m2) -2-ta2'\rightarrow (x2'', m2')$
and *ta2*: *ta2 = convert-TA-initial (convert-obs-initial ta2′)*
and *Suspend2*: *Suspend w ∈ set ⦃ta2′⦄w* **by** *cases auto*
from *τ1 sx1′ sx1″ ta1 status* **have** *τ1′*:¬ *τmove1 (x1′, m1) ta1′ (x1″, m1′)* **by** *simp*
from *τ2 sx2′ sx2″ ta2 status* **have** *τ2′*:¬ *τmove2 (x2′, m2) ta2′ (x2″, m2′)* **by** *simp*
from *tasim ta1 ta2* **have** *ta1′ ∼m ta2′* **by** *simp*
with *Bisim Red1 Red1′ τ1′ Red2 Red2′ τ2′ Bisim′* **have** *x1″ ≈w x2″*
using *Suspend1 Suspend2* **by**(*rule bisim-waitI*)
thus *sx1″ ≈iw sx2″* **using** *sx1″ sx2″ status* **by** *simp*
next
fix *t sx1 m1 sx2 m2 ta1 sx1′ m1′*
assume *t ⊢ (sx1, m1) ≈i (sx2, m2)* **and** *sx1 ≈iw sx2*
and *r1.init-fin t (sx1, m1) ta1 (sx1′, m1′)*
and *Notified ∈ set ⦃ta1⦄w ∨ WokenUp ∈ set ⦃ta1⦄w*
thus *∃ ta2 sx2′ m2′. r2.init-fin t (sx2, m2) ta2 (sx2′, m2′) ∧ t ⊢ (sx1′, m1′) ≈i (sx2′, m2′) ∧ ta-bisim init-fin-bisim ta1 ta2*
by(*rule init-fin-simulation-Wakeup1*)
next
fix *t sx1 m1 sx2 m2 ta2 sx2′ m2′*
assume *t ⊢ (sx1, m1) ≈i (sx2, m2)* **and** *sx1 ≈iw sx2*
and *r2.init-fin t (sx2, m2) ta2 (sx2′, m2′)*
and *Notified ∈ set ⦃ta2⦄w ∨ WokenUp ∈ set ⦃ta2⦄w*
thus *∃ ta1 sx1′ m1′. r1.init-fin t (sx1, m1) ta1 (sx1′, m1′) ∧ t ⊢ (sx1′, m1′) ≈i (sx2′, m2′) ∧ ta-bisim init-fin-bisim ta1 ta2*
by(*rule init-fin-simulation-Wakeup2*)
next
show *(∃ sx1. r1.init-fin-final sx1) = (∃ sx2. r2.init-fin-final sx2)*
using *ex-final1-conv-ex-final2* **by**(*auto*)
qed

end

context *FWdelay-bisimulation-diverge* **begin**

lemma *init-fin-simulation-silent1*:
⟦ t ⊢ sxm1 ≈i sxm2; τtrsys.silent-move (r1.init-fin t) r1.init-fin-τmove sxm1 sxm1′ ⟧
⟹ ∃ sxm2′. τtrsys.silent-moves (r2.init-fin t) r2.init-fin-τmove sxm2 sxm2′ ∧ t ⊢ sxm1′ ≈i sxm2′
by(*cases sxm1′*)(*auto 4 4 elim*!: *init-fin-bisim.cases dest*!: *r1.init-fin-silent-moveD dest*: *simulation-silent1 intro*!: *r2.init-fin-silent-moves-RunningI*)

lemma *init-fin-simulation-silent2*:
⟦ t ⊢ sxm1 ≈i sxm2; τtrsys.silent-move (r2.init-fin t) r2.init-fin-τmove sxm2 sxm2′ ⟧
⟹ ∃ sxm1′. τtrsys.silent-moves (r1.init-fin t) r1.init-fin-τmove sxm1 sxm1′ ∧ t ⊢ sxm1′ ≈i sxm2′
using *FWdelay-bisimulation-diverge.init-fin-simulation-silent1*[*OF FWdelay-bisimulation-diverge-flip*]
unfolding *flip-simps* **.**

lemma *init-fin-τdiverge-bisim-inv*:
t ⊢ sxm1 ≈i sxm2

$\Longrightarrow \tau trsys.\tau diverge\ (r1.init\text{-}fin\ t)\ r1.init\text{-}fin\text{-}\tau move\ sxm1 =$
$\tau trsys.\tau diverge\ (r2.init\text{-}fin\ t)\ r2.init\text{-}fin\text{-}\tau move\ sxm2$
by(*cases sxm1*)(*cases sxm2*, *auto simp add*: *r1.init-fin-τdiverge-conv r2.init-fin-τdiverge-conv init-fin-bisim-iff τdiverge-bisim-inv*)

lemma *init-fin-delay-bisimulation-diverge*:
delay-bisimulation-diverge (*r1.init-fin t*) (*r2.init-fin t*) (*init-fin-bisim t*) (*ta-bisim init-fin-bisim*)
r1.init-fin-τmove r2.init-fin-τmove
by(*blast intro*: *delay-bisimulation-diverge.intro init-fin-delay-bisimulation-obs delay-bisimulation-diverge-axioms.intro init-fin-simulation-silent1 init-fin-simulation-silent2 init-fin-τdiverge-bisim-inv del*: *iffI*)+

lemma *init-fin-FWdelay-bisimulation-diverge*:
FWdelay-bisimulation-diverge r1.init-fin-final r1.init-fin r2.init-fin-final r2.init-fin init-fin-bisim init-fin-bisim-wait r1.init-fin-τmove r2.init-fin-τmove
by(*intro FWdelay-bisimulation-diverge.intro init-fin-FWdelay-bisimulation-obs FWdelay-bisimulation-diverge-axioms.intro init-fin-delay-bisimulation-diverge*)

end

context *FWbisimulation* **begin**

lemma *init-fin-simulation1*:
assumes $t \vdash s1 \approx i\ s2$ **and** *r1.init-fin t s1 tl1 s1'*
shows $\exists s2'\ tl2.\ r2.init\text{-}fin\ t\ s2\ tl2\ s2' \wedge t \vdash s1' \approx i\ s2' \wedge ta\text{-}bisim\ init\text{-}fin\text{-}bisim\ tl1\ tl2$
using *init-fin-simulation1*[*OF assms*] **by**(*auto simp add*: *τmoves-False init-fin-τmoves-False*)

lemma *init-fin-simulation2*:
$[\![\ t \vdash s1 \approx i\ s2;\ r2.init\text{-}fin\ t\ s2\ tl2\ s2'\ ]\!]$
$\Longrightarrow \exists s1'\ tl1.\ r1.init\text{-}fin\ t\ s1\ tl1\ s1' \wedge t \vdash s1' \approx i\ s2' \wedge ta\text{-}bisim\ init\text{-}fin\text{-}bisim\ tl1\ tl2$
using *FWbisimulation.init-fin-simulation1*[*OF FWbisimulation-flip*]
unfolding *flip-simps* **.**

lemma *init-fin-bisimulation*:
bisimulation (*r1.init-fin t*) (*r2.init-fin t*) (*init-fin-bisim t*) (*ta-bisim init-fin-bisim*)
by(*unfold-locales*)(*erule* (*1*) *init-fin-simulation1 init-fin-simulation2*)+

lemma *init-fin-FWbisimulation*:
FWbisimulation r1.init-fin-final r1.init-fin r2.init-fin-final r2.init-fin init-fin-bisim
proof(*intro FWbisimulation.intro r1.multithreaded-init-fin r2.multithreaded-init-fin FWbisimulation-axioms.intro init-fin-bisimulation*)
fix *t sx1 m1 sx2 m2*
assume $t \vdash (sx1,\ m1) \approx i\ (sx2,\ m2)$
thus *r1.init-fin-final sx1 = r2.init-fin-final sx2*
by *cases simp-all*
next
fix *t' sx m1 sxx m2 t sx1 sx2 ta1 sx1' m1' ta2 sx2' m2'*
assume $t' \vdash (sx,\ m1) \approx i\ (sxx,\ m2)$ $t \vdash (sx1,\ m1) \approx i\ (sx2,\ m2)$
and *r1.init-fin t* $(sx1,\ m1)$ *ta1* $(sx1',\ m1')$
and *r2.init-fin t* $(sx2,\ m2)$ *ta2* $(sx2',\ m2')$
and $t \vdash (sx1',\ m1') \approx i\ (sx2',\ m2')$
and *ta-bisim init-fin-bisim ta1 ta2*
from *FWdelay-bisimulation-obs.bisim-inv-red-other*
[*OF init-fin-FWdelay-bisimulation-obs*, *OF this*(*1−2*) *- this*(*3*) *- - this*(*4*) *- this*(*5−6*)]
show $t' \vdash (sx,\ m1') \approx i\ (sxx,\ m2')$ **by**(*simp add*: *init-fin-τmoves-False*)

next
show ($\exists$ *sx1*. *r1.init-fin-final sx1*) = ($\exists$ *sx2*. *r2.init-fin-final sx2*)
using *ex-final1-conv-ex-final2* **by**(*auto*)
qed

end

end

Chapter 2

Data Flow Analysis Framework

2.1 Semilattices

theory *Semilat*
imports *Main HOL−Library.While-Combinator*
begin

type-synonym *'a ord* = *'a ⇒ 'a ⇒ bool*
type-synonym *'a binop* = *'a ⇒ 'a ⇒ 'a*
type-synonym *'a sl* = *'a set × 'a ord × 'a binop*

definition *lesub* :: *'a ⇒ 'a ord ⇒ 'a ⇒ bool*
where *lesub x r y ⟷ r x y*

definition *lesssub* :: *'a ⇒ 'a ord ⇒ 'a ⇒ bool*
where *lesssub x r y ⟷ lesub x r y ∧ x ≠ y*

definition *plussub* :: *'a ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'b ⇒ 'c*
where *plussub x f y = f x y*

notation (*ASCII*)
lesub (‹(- /<='-- -)› [*50*, *1000*, *51*] *50*) **and**
lesssub (‹(- /<'-- -)› [*50*, *1000*, *51*] *50*) **and**
plussub (‹(- /+'-- -)› [*65*, *1000*, *66*] *65*)

notation
lesub (‹(- /⊑_ -)› [*50*, *0*, *51*] *50*) **and**
lesssub (‹(- /⊏_ -)› [*50*, *0*, *51*] *50*) **and**
plussub (‹(- /⊔_ -)› [*65*, *0*, *66*] *65*)

abbreviation (*input*)
lesub1 :: *'a ⇒ 'a ord ⇒ 'a ⇒ bool* (‹(- /⊑_ -)› [*50*, *1000*, *51*] *50*)
where *x ⊑r y == x ⊑r y*

abbreviation (*input*)
lesssub1 :: *'a ⇒ 'a ord ⇒ 'a ⇒ bool* (‹(- /⊏_ -)› [*50*, *1000*, *51*] *50*)
where *x ⊏r y == x ⊏r y*

abbreviation (*input*)

plussub1 :: $'a \Rightarrow ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b \Rightarrow 'c$ ($\langle$(- /$\sqcup$_ -)$\rangle$ [65, 1000, 66] 65)
where $x \sqcup_f y == x \sqcup_f y$

definition *ord* :: $('a \times 'a)$ *set* $\Rightarrow$ $'a$ *ord*
where
ord r = $(\lambda x\ y.\ (x,y) \in r)$

definition *order* :: $'a$ *ord* $\Rightarrow$ *bool*
where
order r $\longleftrightarrow$ $(\forall x.\ x \sqsubseteq_r x) \wedge (\forall x\ y.\ x \sqsubseteq_r y \wedge y \sqsubseteq_r x \longrightarrow x{=}y) \wedge (\forall x\ y\ z.\ x \sqsubseteq_r y \wedge y \sqsubseteq_r z \longrightarrow x \sqsubseteq_r z)$

definition *top* :: $'a$ *ord* $\Rightarrow$ $'a$ $\Rightarrow$ *bool*
where
top r T $\longleftrightarrow$ $(\forall x.\ x \sqsubseteq_r T)$

definition *acc* :: $'a$ *set* $\Rightarrow$ $'a$ *ord* $\Rightarrow$ *bool*
where
acc A r $\longleftrightarrow$ *wf* $\{(y,x).\ x \in A \wedge y \in A \wedge x \sqsubset_r y\}$

definition *closed* :: $'a$ *set* $\Rightarrow$ $'a$ *binop* $\Rightarrow$ *bool*
where
closed A f $\longleftrightarrow$ $(\forall x{\in}A.\ \forall y{\in}A.\ x \sqcup_f y \in A)$

definition *semilat* :: $'a$ *sl* $\Rightarrow$ *bool*
where
semilat = $(\lambda(A,r,f).$ *order r* $\wedge$ *closed A f* $\wedge$
$(\forall x{\in}A.\ \forall y{\in}A.\ x \sqsubseteq_r x \sqcup_f y) \wedge$
$(\forall x{\in}A.\ \forall y{\in}A.\ y \sqsubseteq_r x \sqcup_f y) \wedge$
$(\forall x{\in}A.\ \forall y{\in}A.\ \forall z{\in}A.\ x \sqsubseteq_r z \wedge y \sqsubseteq_r z \longrightarrow x \sqcup_f y \sqsubseteq_r z))$

definition *is-ub* :: $('a \times 'a)$ *set* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ $\Rightarrow$ *bool*
where
is-ub r x y u $\longleftrightarrow$ $(x,u){\in}r \wedge (y,u){\in}r$

definition *is-lub* :: $('a \times 'a)$ *set* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ $\Rightarrow$ *bool*
where
is-lub r x y u $\longleftrightarrow$ *is-ub r x y u* $\wedge$ $(\forall z.$ *is-ub r x y z* $\longrightarrow$ $(u,z){\in}r)$

definition *some-lub* :: $('a \times 'a)$ *set* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ $\Rightarrow$ $'a$
where
some-lub r x y = $(SOME\ z.$ *is-lub r x y z*$)$

locale *Semilat* =
fixes *A* :: $'a$ *set*
fixes *r* :: $'a$ *ord*
fixes *f* :: $'a$ *binop*
assumes *semilat*: *semilat* (*A*, *r*, *f*)

lemma *order-refl* [*simp*, *intro*]: *order r* $\Longrightarrow$ $x \sqsubseteq_r x$

lemma *order-antisym*: $[\![$ *order r*; $x \sqsubseteq_r y$; $y \sqsubseteq_r x$ $]\!] \Longrightarrow x = y$

lemma *order-trans*: $[\![$ *order r*; $x \sqsubseteq_r y$; $y \sqsubseteq_r z$ $]\!] \Longrightarrow x \sqsubseteq_r z$

lemma *order-less-irrefl* [*intro*, *simp*]: *order r* $\Longrightarrow \neg\ x \sqsubset_r x$

lemma *order-less-trans*: $[\![$ *order r*; $x \sqsubset_r y$; $y \sqsubset_r z$ $]\!] \Longrightarrow x \sqsubset_r z$

lemma *topD* [*simp*, *intro*]: *top r T* $\Longrightarrow x \sqsubseteq_r T$

lemma *top-le-conv* [*simp*]: $[\![$ *order r*; *top r T* $]\!] \Longrightarrow (T \sqsubseteq_r x) = (x = T)$

lemma *semilat-Def*:
semilat(*A*,*r*,*f*) $\longleftrightarrow$ *order r* $\wedge$ *closed A f* $\wedge$
$(\forall x{\in}A.\ \forall y{\in}A.\ x \sqsubseteq_r x \sqcup_f y) \wedge$
$(\forall x{\in}A.\ \forall y{\in}A.\ y \sqsubseteq_r x \sqcup_f y) \wedge$
$(\forall x{\in}A.\ \forall y{\in}A.\ \forall z{\in}A.\ x \sqsubseteq_r z \wedge y \sqsubseteq_r z \longrightarrow x \sqcup_f y \sqsubseteq_r z)$

lemma (**in** *Semilat*) *orderI* [*simp*, *intro*]: *order r*

lemma (**in** *Semilat*) *closedI* [*simp*, *intro*]: *closed A f*

lemma *closedD*: $[\![$ *closed A f*; $x{\in}A$; $y{\in}A$ $]\!] \Longrightarrow x \sqcup_f y \in A$

lemma *closed-UNIV* [*simp*]: *closed UNIV f*

lemma (**in** *Semilat*) *closed-f* [*simp*, *intro*]: $[\![x \in A; y \in A]\!] \Longrightarrow x \sqcup_f y \in A$

lemma (**in** *Semilat*) *refl-r* [*intro*, *simp*]: $x \sqsubseteq_r x$ **by** *simp*

lemma (**in** *Semilat*) *antisym-r* [*intro?*]: $[\![x \sqsubseteq_r y; y \sqsubseteq_r x]\!] \Longrightarrow x = y$

lemma (**in** *Semilat*) *trans-r* [*trans*, *intro?*]: $[\![x \sqsubseteq_r y; y \sqsubseteq_r z]\!] \Longrightarrow x \sqsubseteq_r z$

lemma (**in** *Semilat*) *ub1* [*simp*, *intro?*]: $[\![x \in A; y \in A]\!] \Longrightarrow x \sqsubseteq_r x \sqcup_f y$

lemma (**in** *Semilat*) *ub2* [*simp*, *intro?*]: $[\![x \in A; y \in A]\!] \Longrightarrow y \sqsubseteq_r x \sqcup_f y$

lemma (**in** *Semilat*) *lub* [*simp*, *intro?*]:
$[\![x \sqsubseteq_r z; y \sqsubseteq_r z; x \in A; y \in A; z \in A]\!] \Longrightarrow x \sqcup_f y \sqsubseteq_r z$

lemma (**in** *Semilat*) *plus-le-conv* [*simp*]:
$[\![x \in A; y \in A; z \in A]\!] \Longrightarrow (x \sqcup_f y \sqsubseteq_r z) = (x \sqsubseteq_r z \wedge y \sqsubseteq_r z)$

lemma (**in** *Semilat*) *le-iff-plus-unchanged*:
assumes $x \in A$ **and** $y \in A$
shows $x \sqsubseteq_r y \longleftrightarrow x \sqcup_f y = y$ (**is** *?P* $\longleftrightarrow$ *?Q*)
lemma (**in** *Semilat*) *le-iff-plus-unchanged2*:
assumes $x \in A$ **and** $y \in A$
shows $x \sqsubseteq_r y \longleftrightarrow y \sqcup_f x = y$ (**is** *?P* $\longleftrightarrow$ *?Q*)
lemma (**in** *Semilat*) *plus-assoc* [*simp*]:
assumes *a*: $a \in A$ **and** *b*: $b \in A$ **and** *c*: $c \in A$
shows $a \sqcup_f (b \sqcup_f c) = a \sqcup_f b \sqcup_f c$
lemma (**in** *Semilat*) *plus-com-lemma*:
$[\![a \in A; b \in A]\!] \Longrightarrow a \sqcup_f b \sqsubseteq_r b \sqcup_f a$
lemma (**in** *Semilat*) *plus-commutative*:
$[\![a \in A; b \in A]\!] \Longrightarrow a \sqcup_f b = b \sqcup_f a$

lemma *is-lubD*:
is-lub r x y u $\Longrightarrow$ *is-ub r x y u* $\wedge$ $(\forall z.$ *is-ub r x y z* $\longrightarrow$ $(u,z) \in r)$

lemma *is-ubI*:
$[\![(x,u) \in r;\ (y,u) \in r]\!] \Longrightarrow$ *is-ub r x y u*

lemma *is-ubD*:
is-ub r x y u $\Longrightarrow$ $(x,u) \in r \wedge (y,u) \in r$

lemma *is-lub-bigger1* [*iff*]:
is-lub $(r\widehat{}*)$ *x y y* = $((x,y) \in r\widehat{}*)$
lemma *is-lub-bigger2* [*iff*]:
is-lub $(r\widehat{}*)$ *x y x* = $((y,x) \in r\widehat{}*)$
lemma *extend-lub*:
assumes *single-valued r*
and *is-lub* (r^*) *x y u*
and $(x',\ x) \in r$
shows $\exists v.$ *is-lub* (r^*) x' *y v*
lemma *single-valued-has-lubs*:
assumes *single-valued r*
and *in-r*: $(x,\ u) \in r^*$ $(y,\ u) \in r^*$
shows $\exists z.$ *is-lub* (r^*) *x y z*
lemma *some-lub-conv*:
$[\![$ *acyclic r*; *is-lub* $(r\widehat{}*)$ *x y u* $]\!] \Longrightarrow$ *some-lub* $(r\widehat{}*)$ *x y* = *u*
lemma *is-lub-some-lub*:
$[\![$ *single-valued r*; *acyclic r*; $(x,u) \in r\widehat{}*$; $(y,u) \in r\widehat{}*$ $]\!]$
$\Longrightarrow$ *is-lub* $(r\widehat{}*)$ *x y* (*some-lub* $(r\widehat{}*)$ *x y*)

2.1.1 An executable lub-finder

definition *exec-lub* :: $('a * 'a)$ *set* $\Rightarrow$ $('a \Rightarrow 'a) \Rightarrow 'a$ *binop*
where
exec-lub r f x y = *while* $(\lambda z.$ $(x,z) \notin r^*)$ *f y*

lemma *exec-lub-refl*: *exec-lub r f T T* = *T*
by (*simp add*: *exec-lub-def while-unfold*)

lemma *acyclic-single-valued-finite*:
$[\![$*acyclic r*; *single-valued r*; $(x,y) \in r^*$$]\!]$
$\Longrightarrow$ *finite* $(r \cap \{a.\ (x,\ a) \in r^*\} \times \{b.\ (b,\ y) \in r^*\})$

lemma *exec-lub-conv*:
$[\![$ *acyclic r*; $\forall x\ y.\ (x,y) \in r \longrightarrow f\ x = y$; *is-lub* (r^*) *x y u* $]\!] \Longrightarrow$
exec-lub r f x y = *u*
lemma *is-lub-exec-lub*:
$[\![$ *single-valued r*; *acyclic r*; $(x,u):r\widehat{}*$; $(y,u):r\widehat{}*$; $\forall x\ y.\ (x,y) \in r \longrightarrow f\ x = y$ $]\!]$
$\Longrightarrow$ *is-lub* $(r\widehat{}*)$ *x y* (*exec-lub r f x y*)

end

2.2 The Error Type

theory *Err*
imports *Semilat*
begin

datatype *'a err* = *Err* | *OK 'a*

type-synonym *'a ebinop* = *'a* ⇒ *'a* ⇒ *'a err*
type-synonym *'a esl* = *'a set* × *'a ord* × *'a ebinop*

primrec *ok-val* :: *'a err* ⇒ *'a*
where
ok-val (*OK x*) = *x*

definition *lift* :: (*'a* ⇒ *'b err*) ⇒ (*'a err* ⇒ *'b err*)
where
lift f e = (*case e of Err* ⇒ *Err* | *OK x* ⇒ *f x*)

definition *lift2* :: (*'a* ⇒ *'b* ⇒ *'c err*) ⇒ *'a err* ⇒ *'b err* ⇒ *'c err*
where
lift2 f e_1 e_2 =
(*case* e_1 *of Err* ⇒ *Err* | *OK x* ⇒ (*case* e_2 *of Err* ⇒ *Err* | *OK y* ⇒ *f x y*))

definition *le* :: *'a ord* ⇒ *'a err ord*
where
le r e_1 e_2 =
(*case* e_2 *of Err* ⇒ *True* | *OK y* ⇒ (*case* e_1 *of Err* ⇒ *False* | *OK x* ⇒ *x* ⊑$_r$ *y*))

definition *sup* :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ (*'a err* ⇒ *'b err* ⇒ *'c err*)
where
sup f = *lift2* (λ*x y*. *OK* (*x* ⊔$_f$ *y*))

definition *err* :: *'a set* ⇒ *'a err set*
where
err A = *insert Err* {*OK x*|*x*. *x*∈*A*}

definition *esl* :: *'a sl* ⇒ *'a esl*
where
esl = (λ(*A*,*r*,*f*). (*A*, *r*, λ*x y*. *OK*(*f x y*)))

definition *sl* :: *'a esl* ⇒ *'a err sl*
where
sl = (λ(*A*,*r*,*f*). (*err A*, *le r*, *lift2 f*))

abbreviation
err-semilat :: *'a esl* ⇒ *bool* **where**
err-semilat L == *semilat*(*sl L*)

primrec *strict* :: (*'a* ⇒ *'b err*) ⇒ (*'a err* ⇒ *'b err*)
where
strict f Err = *Err*
| *strict f* (*OK x*) = *f x*

lemma *err-def'*:
err A = *insert Err* {*x*. ∃ *y*∈*A*. *x* = *OK y*}
lemma *strict-Some* [*simp*]:
(*strict f x* = *OK y*) = (∃ *z*. *x* = *OK z* ∧ *f z* = *OK y*)
lemma *not-Err-eq*: (*x* ≠ *Err*) = (∃ *a*. *x* = *OK a*)
lemma *not-OK-eq*: (∀ *y*. *x* ≠ *OK y*) = (*x* = *Err*)
lemma *unfold-lesub-err*: *e1* ⊑$_{le\ r}$ *e2* = *le r e1 e2*
lemma *le-err-refl*: ∀ *x*. *x* ⊑$_r$ *x* ⟹ *e* ⊑$_{le\ r}$ *e*
lemma *le-err-trans* [*rule-format*]:
order r ⟹ *e1* ⊑$_{le\ r}$ *e2* ⟶ *e2* ⊑$_{le\ r}$ *e3* ⟶ *e1* ⊑$_{le\ r}$ *e3*
lemma *le-err-antisym* [*rule-format*]:
order r ⟹ *e1* ⊑$_{le\ r}$ *e2* ⟶ *e2* ⊑$_{le\ r}$ *e1* ⟶ *e1*=*e2*
lemma *OK-le-err-OK*: (*OK x* ⊑$_{le\ r}$ *OK y*) = (*x* ⊑$_r$ *y*)
lemma *order-le-err* [*iff*]: *order*(*le r*) = *order r*
lemma *le-Err* [*iff*]: *e* ⊑$_{le\ r}$ *Err*
lemma *Err-le-conv* [*iff*]: *Err* ⊑$_{le\ r}$ *e* = (*e* = *Err*)
lemma *le-OK-conv* [*iff*]: *e* ⊑$_{le\ r}$ *OK x* = (∃ *y*. *e* = *OK y* ∧ *y* ⊑$_r$ *x*)
lemma *OK-le-conv*: *OK x* ⊑$_{le\ r}$ *e* = (*e* = *Err* ∨ (∃ *y*. *e* = *OK y* ∧ *x* ⊑$_r$ *y*))
lemma *top-Err* [*iff*]: *top* (*le r*) *Err*
lemma *OK-less-conv* [*rule-format*, *iff*]:
OK x ⊏$_{le\ r}$ *e* = (*e*=*Err* ∨ (∃ *y*. *e* = *OK y* ∧ *x* ⊏$_r$ *y*))
lemma *not-Err-less* [*rule-format*, *iff*]: ¬(*Err* ⊏$_{le\ r}$ *x*)
lemma *semilat-errI* [*intro*]: **assumes** *Semilat A r f*
shows *semilat*(*err A*, *le r*, *lift2*(λ*x y*. *OK*(*f x y*)))
lemma *err-semilat-eslI-aux*:
assumes *Semilat A r f* **shows** *err-semilat*(*esl*(*A*,*r*,*f*))
lemma *err-semilat-eslI* [*intro*, *simp*]:
semilat L ⟹ *err-semilat* (*esl L*)
lemma *acc-err* [*simp*, *intro*!]: *acc A r* ⟹ *acc* (*err A*) (*le r*)
lemma *Err-in-err* [*iff*]: *Err* : *err A*
lemma *Ok-in-err* [*iff*]: (*OK x* ∈ *err A*) = (*x*∈*A*)

2.2.1 lift

lemma *lift-in-errI*:
⟦ *e* ∈ *err S*; ∀ *x*∈*S*. *e* = *OK x* ⟶ *f x* ∈ *err S* ⟧ ⟹ *lift f e* ∈ *err S*
lemma *Err-lift2* [*simp*]: *Err* ⊔$_{lift2\ f}$ *x* = *Err*
lemma *lift2-Err* [*simp*]: *x* ⊔$_{lift2\ f}$ *Err* = *Err*
lemma *OK-lift2-OK* [*simp*]: *OK x* ⊔$_{lift2\ f}$ *OK y* = *x* ⊔$_f$ *y*

2.2.2 sup

lemma *Err-sup-Err* [*simp*]: *Err* ⊔$_{sup\ f}$ *x* = *Err*
lemma *Err-sup-Err2* [*simp*]: *x* ⊔$_{sup\ f}$ *Err* = *Err*
lemma *Err-sup-OK* [*simp*]: *OK x* ⊔$_{sup\ f}$ *OK y* = *OK* (*x* ⊔$_f$ *y*)
lemma *Err-sup-eq-OK-conv* [*iff*]:
(*sup f ex ey* = *OK z*) = (∃ *x y*. *ex* = *OK x* ∧ *ey* = *OK y* ∧ *f x y* = *z*)
lemma *Err-sup-eq-Err* [*iff*]: (*sup f ex ey* = *Err*) = (*ex*=*Err* ∨ *ey*=*Err*)

2.2.3 semilat (err A) (le r) f

lemma *semilat-le-err-Err-plus* [*simp*]:
⟦ *x*∈ *err A*; *semilat*(*err A*, *le r*, *f*) ⟧ ⟹ *Err* ⊔$_f$ *x* = *Err*
lemma *semilat-le-err-plus-Err* [*simp*]:

$[\![x{\in} \mathit{err}\ A;\ \mathit{semilat}(\mathit{err}\ A,\ \mathit{le}\ r,\ f)\ ]\!] \Longrightarrow x \sqcup_f \mathit{Err} = \mathit{Err}$
lemma *semilat-le-err-OK1*:
$[\![x{\in}A;\ y{\in}A;\ \mathit{semilat}(\mathit{err}\ A,\ \mathit{le}\ r,\ f);\ \mathit{OK}\ x \sqcup_f \mathit{OK}\ y = \mathit{OK}\ z\ ]\!]$
$\Longrightarrow x \sqsubseteq_r z$
lemma *semilat-le-err-OK2*:
$[\![x{\in}A;\ y{\in}A;\ \mathit{semilat}(\mathit{err}\ A,\ \mathit{le}\ r,\ f);\ \mathit{OK}\ x \sqcup_f \mathit{OK}\ y = \mathit{OK}\ z\ ]\!]$
$\Longrightarrow y \sqsubseteq_r z$
lemma *eq-order-le*:
$[\![x{=}y;\ \mathit{order}\ r\ ]\!] \Longrightarrow x \sqsubseteq_r y$
lemma *OK-plus-OK-eq-Err-conv* [*simp*]:
assumes $x{\in}A \quad y{\in}A \quad \mathit{semilat}(\mathit{err}\ A,\ \mathit{le}\ r,\ \mathit{fe})$
shows $(\mathit{OK}\ x \sqcup_{\mathit{fe}} \mathit{OK}\ y = \mathit{Err}) = (\neg(\exists\, z{\in}A.\ x \sqsubseteq_r z \wedge y \sqsubseteq_r z))$

2.2.4 semilat (err(Union AS))

lemma *all-bex-swap-lemma* [*iff*]:
$(\forall x.\ (\exists\, y{\in}A.\ x = f\ y) \longrightarrow P\ x) = (\forall\, y{\in}A.\ P(f\ y))$
lemma *closed-err-Union-lift2I*:
$[\![\forall\, A{\in}AS.\ \mathit{closed}\ (\mathit{err}\ A)\ (\mathit{lift2}\ f);\ AS \neq \{\};$
$\forall\, A{\in}AS.\forall\, B{\in}AS.\ A{\neq}B \longrightarrow (\forall\, a{\in}A.\forall\, b{\in}B.\ a \sqcup_f b = \mathit{Err})\ ]\!]$
$\Longrightarrow \mathit{closed}\ (\mathit{err}(\mathit{Union}\ AS))\ (\mathit{lift2}\ f)$

If $AS = \{\}$ the thm collapses to $\mathit{order}\ r \wedge \mathit{closed}\ \{\mathit{Err}\}\ f \wedge \mathit{Err} \sqcup_f \mathit{Err} = \mathit{Err}$ which may not hold

lemma *err-semilat-UnionI*:
$[\![\forall\, A{\in}AS.\ \mathit{err\text{-}semilat}(A,\ r,\ f);\ AS \neq \{\};$
$\forall\, A{\in}AS.\forall\, B{\in}AS.\ A{\neq}B \longrightarrow (\forall\, a{\in}A.\forall\, b{\in}B.\ \neg a \sqsubseteq_r b \wedge a \sqcup_f b = \mathit{Err})\ ]\!]$
$\Longrightarrow \mathit{err\text{-}semilat}(\mathit{Union}\ AS,\ r,\ f)$
end

2.3 More about Options

theory *Opt*
imports
Err
begin

definition *le* :: *'a ord* ⇒ *'a option ord*
where
le r o$_1$ *o*$_2$ =
(*case o*$_2$ *of None* ⇒ *o*$_1$=*None* | *Some y* ⇒ (*case o*$_1$ *of None* ⇒ *True* | *Some x* ⇒ *x* $\sqsubseteq_r$ *y*))

definition *opt* :: *'a set* ⇒ *'a option set*
where
opt A = *insert None* {*Some y* |*y*. *y* ∈ *A*}

definition *sup* :: *'a ebinop* ⇒ *'a option ebinop*
where
sup f o$_1$ *o*$_2$ =
(*case o*$_1$ *of None* ⇒ *OK o*$_2$
| *Some x* ⇒ (*case o*$_2$ *of None* ⇒ *OK o*$_1$
| *Some y* ⇒ (*case f x y of Err* ⇒ *Err* | *OK z* ⇒ *OK* (*Some z*))))

definition *esl* :: *'a esl* ⇒ *'a option esl*

where
esl = (λ(*A*,*r*,*f*). (*opt A*, *le r*, *sup f*))

lemma *unfold-le-opt*:
$o_1 \sqsubseteq_{le\ r}\ o_2 =$
(*case* o_2 *of None* ⇒ o_1=*None* |
Some y ⇒ (*case* o_1 *of None* ⇒ *True* | *Some x* ⇒ $x \sqsubseteq_r\ y$))
lemma *le-opt-refl*: *order r* ⟹ $x \sqsubseteq_{le\ r}\ x$

2.4 Products as Semilattices

theory *Product*
imports *Err*
begin

definition *le* :: *'a ord* ⇒ *'b ord* ⇒ (*'a* × *'b*) *ord*
where
le $r_A\ r_B$ = (λ(a_1,b_1) (a_2,b_2). $a_1 \sqsubseteq_{r_A}\ a_2 \wedge b_1 \sqsubseteq_{r_B}\ b_2$)

definition *sup* :: *'a ebinop* ⇒ *'b ebinop* ⇒ (*'a* × *'b*) *ebinop*
where
sup f g = (λ(a_1,b_1)(a_2,b_2). *Err.sup Pair* ($a_1 \sqcup_f\ a_2$) ($b_1 \sqcup_g\ b_2$))

definition *esl* :: *'a esl* ⇒ *'b esl* ⇒ (*'a* × *'b*) *esl*
where
esl = (λ(*A*,r_A,f_A) (*B*,r_B,f_B). (*A* × *B*, *le* $r_A\ r_B$, *sup* $f_A\ f_B$))

abbreviation
lesubprod :: *'a* × *'b* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'b* ⇒ *'b* ⇒ *bool*) ⇒ *'a* × *'b* ⇒ *bool*
(‹(- /⊑'(-,-') -)› [*50*, *0*, *0*, *51*] *50*) **where**
p ⊑(*rA*,*rB*) *q* == *p* ⊑*Product.le rA rB q*

lemma *unfold-lesub-prod*: $x \sqsubseteq(r_A,r_B)\ y$ = *le* $r_A\ r_B\ x\ y$
lemma *le-prod-Pair-conv* [*iff*]: $((a_1,b_1) \sqsubseteq(r_A,r_B)\ (a_2,b_2)) = (a_1 \sqsubseteq_{r_A}\ a_2\ \&\ b_1 \sqsubseteq_{r_B}\ b_2)$
lemma *less-prod-Pair-conv*:
$((a_1,b_1) \sqsubset_{Product.le\ r_A\ r_B}\ (a_2,b_2)) =$
$(a_1 \sqsubset_{r_A}\ a_2\ \&\ b_1 \sqsubseteq_{r_B}\ b_2 \mid a_1 \sqsubseteq_{r_A}\ a_2\ \&\ b_1 \sqsubset_{r_B}\ b_2)$
lemma *order-le-prod* [*iff*]: *order*(*Product.le* $r_A\ r_B$) = (*order* r_A & *order* r_B)

lemma *acc-le-prodI* [*intro*!]:
⟦ *acc A* r_A; *acc B* r_B ⟧ ⟹ *acc* (*A* × *B*) (*Product.le* $r_A\ r_B$)

lemma *closed-lift2-sup*:
⟦ *closed* (*err A*) (*lift2 f*); *closed* (*err B*) (*lift2 g*) ⟧ ⟹
closed (*err*(*A*×*B*)) (*lift2*(*sup f g*))
lemma *unfold-plussub-lift2*: $e_1 \sqcup_{lift2\ f}\ e_2$ = *lift2 f* $e_1\ e_2$

lemma *plus-eq-Err-conv* [*simp*]:
assumes *x*∈*A* *y*∈*A* *semilat*(*err A*, *Err.le r*, *lift2 f*)
shows ($x \sqcup_f\ y$ = *Err*) = (¬(∃ *z*∈*A*. $x \sqsubseteq_r\ z \wedge y \sqsubseteq_r\ z$))
lemma *err-semilat-Product-esl*:
⋀$L_1\ L_2$. ⟦ *err-semilat* L_1; *err-semilat* L_2 ⟧ ⟹ *err-semilat*(*Product.esl* $L_1\ L_2$)

end

2.5 Fixed Length Lists

theory *Listn*
imports *Err*
begin

definition *list* :: *nat* ⇒ *'a set* ⇒ *'a list set*
where
list n A = {*xs*. *size xs* = *n* ∧ *set xs* ⊆ *A*}

definition *le* :: *'a ord* ⇒ (*'a list*)*ord*
where
le r = *list-all2* (λ*x y*. *x* ⊑*r* *y*)

abbreviation
lesublist :: *'a list* ⇒ *'a ord* ⇒ *'a list* ⇒ *bool* (‹(- /[⊑-] -)› [*50*, *0*, *51*] *50*) **where**
x [⊑*r*] *y* == *x* <=-(*Listn.le r*) *y*

abbreviation
lesssublist :: *'a list* ⇒ *'a ord* ⇒ *'a list* ⇒ *bool* (‹(- /[⊏-] -)› [*50*, *0*, *51*] *50*) **where**
x [⊏*r*] *y* == *x* <-(*Listn.le r*) *y*

abbreviation
plussublist :: *'a list* ⇒ (*'a* ⇒ *'b* ⇒ *'c*) ⇒ *'b list* ⇒ *'c list*
(‹(- /[⊔-] -)› [*65*, *0*, *66*] *65*) **where**
x [⊔*f*] *y* == *x* ⊔*map2 f* *y*

primrec *coalesce* :: *'a err list* ⇒ *'a list err*
where
coalesce [] = *OK*[]
| *coalesce* (*ex*#*exs*) = *Err.sup* (#) *ex* (*coalesce exs*)

definition *sl* :: *nat* ⇒ *'a sl* ⇒ *'a list sl*
where
sl n = (λ(*A*,*r*,*f*). (*list n A*, *le r*, *map2 f*))

definition *sup* :: (*'a* ⇒ *'b* ⇒ *'c err*) ⇒ *'a list* ⇒ *'b list* ⇒ *'c list err*
where
sup f = (λ*xs ys*. *if size xs* = *size ys then coalesce*(*xs* [⊔*f*] *ys*) *else Err*)

definition *upto-esl* :: *nat* ⇒ *'a esl* ⇒ *'a list esl*
where
upto-esl m = (λ(*A*,*r*,*f*). (*Union*{*list n A* |*n*. *n* ≤ *m*}, *le r*, *sup f*))

lemmas [*simp*] = *set-update-subsetI*

lemma *unfold-lesub-list*: *xs* [⊑*r*] *ys* = *Listn.le r xs ys*
lemma *Nil-le-conv* [*iff*]: ([] [⊑*r*] *ys*) = (*ys* = [])
lemma *Cons-notle-Nil* [*iff*]: ¬ *x*#*xs* [⊑*r*] []

lemma *Cons-le-Cons* [*iff*]: $x\#xs\ [\sqsubseteq_r]\ y\#ys = (x \sqsubseteq_r y \wedge xs\ [\sqsubseteq_r]\ ys)$
lemma *Cons-less-Conss* [*simp*]:
$order\ r \Longrightarrow\ x\#xs\ [\sqsubset_r]\ y\#ys = (x \sqsubset_r y \wedge xs\ [\sqsubseteq_r]\ ys \vee x = y \wedge xs\ [\sqsubset_r]\ ys)$
lemma *list-update-le-cong*:
$[\![\ i{<}size\ xs;\ xs\ [\sqsubseteq_r]\ ys;\ x \sqsubseteq_r y\ ]\!] \Longrightarrow xs[i{:=}x]\ [\sqsubseteq_r]\ ys[i{:=}y]$

lemma *le-listD*: $[\![\ xs\ [\sqsubseteq_r]\ ys;\ p < size\ xs\ ]\!] \Longrightarrow xs!p \sqsubseteq_r ys!p$
lemma *le-list-refl*: $\forall x.\ x \sqsubseteq_r x \Longrightarrow xs\ [\sqsubseteq_r]\ xs$
lemma *le-list-trans*: $[\![\ order\ r;\ xs\ [\sqsubseteq_r]\ ys;\ ys\ [\sqsubseteq_r]\ zs\ ]\!] \Longrightarrow xs\ [\sqsubseteq_r]\ zs$
lemma *le-list-antisym*: $[\![\ order\ r;\ xs\ [\sqsubseteq_r]\ ys;\ ys\ [\sqsubseteq_r]\ xs\ ]\!] \Longrightarrow xs = ys$
lemma *order-listI* [*simp*, *intro*!]: $order\ r \Longrightarrow order(Listn.le\ r)$
lemma *lesub-list-impl-same-size* [*simp*]: $xs\ [\sqsubseteq_r]\ ys \Longrightarrow size\ ys = size\ xs$
lemma *lesssub-lengthD*: $xs\ [\sqsubset_r]\ ys \Longrightarrow size\ ys = size\ xs$
lemma *le-list-appendI*: $a\ [\sqsubseteq_r]\ b \Longrightarrow c\ [\sqsubseteq_r]\ d \Longrightarrow a@c\ [\sqsubseteq_r]\ b@d$
lemma *le-listI*:
assumes $length\ a = length\ b$
assumes $\bigwedge n.\ n < length\ a \Longrightarrow a!n \sqsubseteq_r b!n$
shows $a\ [\sqsubseteq_r]\ b$
lemma *listI*: $[\![\ size\ xs = n;\ set\ xs \subseteq A\ ]\!] \Longrightarrow xs \in list\ n\ A$

lemma *listE-length* [*simp*]: $xs \in list\ n\ A \Longrightarrow size\ xs = n$
lemma *less-lengthI*: $[\![\ xs \in list\ n\ A;\ p < n\ ]\!] \Longrightarrow p < size\ xs$
lemma *listE-set* [*simp*]: $xs \in list\ n\ A \Longrightarrow set\ xs \subseteq A$
lemma *list-0* [*simp*]: $list\ 0\ A = \{[]\}$
lemma *in-list-Suc-iff*:
$(xs \in list\ (Suc\ n)\ A) = (\exists y{\in}A.\ \exists ys \in list\ n\ A.\ xs = y\#ys)$
lemma *Cons-in-list-Suc* [*iff*]:
$(x\#xs \in list\ (Suc\ n)\ A) = (x{\in}A \wedge xs \in list\ n\ A)$
lemma *list-not-empty*:
$\exists a.\ a{\in}A \Longrightarrow \exists xs.\ xs \in list\ n\ A$

lemma *nth-in* [*rule-format*, *simp*]:
$\forall i\ n.\ size\ xs = n \longrightarrow set\ xs \subseteq A \longrightarrow i < n \longrightarrow (xs!i) \in A$
lemma *listE-nth-in*: $[\![\ xs \in list\ n\ A;\ i < n\ ]\!] \Longrightarrow xs!i \in A$
lemma *listn-Cons-Suc* [*elim*!]:
$l\#xs \in list\ n\ A \Longrightarrow (\bigwedge n'.\ n = Suc\ n' \Longrightarrow l \in A \Longrightarrow xs \in list\ n'\ A \Longrightarrow P) \Longrightarrow P$
lemma *listn-appendE* [*elim*!]:
$a@b \in list\ n\ A \Longrightarrow (\bigwedge n1\ n2.\ n{=}n1{+}n2 \Longrightarrow a \in list\ n1\ A \Longrightarrow b \in list\ n2\ A \Longrightarrow P) \Longrightarrow P$

lemma *listt-update-in-list* [*simp*, *intro*!]:
$[\![\ xs \in list\ n\ A;\ x{\in}A\ ]\!] \Longrightarrow xs[i := x] \in list\ n\ A$
lemma *list-appendI* [*intro?*]:
$[\![\ a \in list\ n\ A;\ b \in list\ m\ A\ ]\!] \Longrightarrow a\ @\ b \in list\ (n{+}m)\ A$
lemma *list-map* [*simp*]: $(map\ f\ xs \in list\ (size\ xs)\ A) = (f\ `\ set\ xs \subseteq A)$
lemma *list-replicateI* [*intro*]: $x \in A \Longrightarrow replicate\ n\ x \in list\ n\ A$
lemma *plus-list-Nil* [*simp*]: $[]\ [\sqcup_f]\ xs = []$
lemma *plus-list-Cons* [*simp*]:
$(x\#xs)\ [\sqcup_f]\ ys = (case\ ys\ of\ []\ \Rightarrow\ []\ |\ y\#ys \Rightarrow (x \sqcup_f y)\#(xs\ [\sqcup_f]\ ys))$
lemma *length-plus-list* [*rule-format*, *simp*]:
$\forall ys.\ size(xs\ [\sqcup_f]\ ys) = min(size\ xs)\ (size\ ys)$
lemma *nth-plus-list* [*rule-format*, *simp*]:
$\forall xs\ ys\ i.\ size\ xs = n \longrightarrow size\ ys = n \longrightarrow i{<}n \longrightarrow (xs\ [\sqcup_f]\ ys)!i = (xs!i) \sqcup_f (ys!i)$

lemma (**in** *Semilat*) *plus-list-ub1* [*rule-format*]:

⟦ *set xs* ⊆ *A*; *set ys* ⊆ *A*; *size xs* = *size ys* ⟧
⟹ *xs* [⊑$_r$] *xs* [⊔$_f$] *ys*
lemma (**in** *Semilat*) *plus-list-ub2*:
⟦*set xs* ⊆ *A*; *set ys* ⊆ *A*; *size xs* = *size ys* ⟧ ⟹ *ys* [⊑$_r$] *xs* [⊔$_f$] *ys*
lemma (**in** *Semilat*) *plus-list-lub* [*rule-format*]:
shows ∀ *xs ys zs*. *set xs* ⊆ *A* ⟶ *set ys* ⊆ *A* ⟶ *set zs* ⊆ *A*
⟶ *size xs* = *n* ∧ *size ys* = *n* ⟶
xs [⊑$_r$] *zs* ∧ *ys* [⊑$_r$] *zs* ⟶ *xs* [⊔$_f$] *ys* [⊑$_r$] *zs*
lemma (**in** *Semilat*) *list-update-incr* [*rule-format*]:
x∈*A* ⟹ *set xs* ⊆ *A* ⟶
(∀ *i*. *i*<*size xs* ⟶ *xs* [⊑$_r$] *xs*[*i* := *x* ⊔$_f$ *xs*!*i*])
lemma *acc-le-listI'* [*intro*!]:
⟦ *order r*; *acc A r* ⟧ ⟹ *acc* (⋃ *n*. *list n A*) (*Listn.le r*)

2.6 Typing and Dataflow Analysis Framework

theory *Typing-Framework*
imports
Semilattices
begin

The relationship between dataflow analysis and a welltyped-instruction predicate.

type-synonym
's step-type = *nat* ⇒ *'s* ⇒ (*nat* × *'s*) *list*

definition *stable* :: *'s ord* ⇒ *'s step-type* ⇒ *'s list* ⇒ *nat* ⇒ *bool*
where
stable r step τs p ⟷ (∀ (*q*,*τ*) ∈ *set* (*step p* (*τs*!*p*)). *τ* ⊑$_r$ *τs*!*q*)

definition *stables* :: *'s ord* ⇒ *'s step-type* ⇒ *'s list* ⇒ *bool*
where
stables r step τs ⟷ (∀ *p* < *size τs*. *stable r step τs p*)

definition *wt-step* :: *'s ord* ⇒ *'s* ⇒ *'s step-type* ⇒ *'s list* ⇒ *bool*
where
wt-step r T step τs ⟷ (∀ *p*<*size τs*. *τs*!*p* ≠ *T* ∧ *stable r step τs p*)

definition *is-bcv* :: *'s ord* ⇒ *'s* ⇒ *'s step-type* ⇒ *nat* ⇒ *'s set* ⇒ (*'s list* ⇒ *'s list*) ⇒ *bool*
where
is-bcv r T step n A bcv ⟷ (∀ *τs*$_0$ ∈ *list n A*.
(∀ *p*<*n*. (*bcv τs*$_0$)!*p* ≠ *T*) = (∃ *τs* ∈ *list n A*. *τs*$_0$ [⊑$_r$] *τs* ∧ *wt-step r T step τs*))

end

2.7 More on Semilattices

theory *SemilatAlg*
imports *Typing-Framework*
begin

definition *lesubstep-type* :: (*nat* × *'s*) *set* ⇒ *'s ord* ⇒ (*nat* × *'s*) *set* ⇒ *bool*
(‹(- /{⊑$_-$} -)› [*50*, *0*, *51*] *50*)
where *A* {⊑$_r$} *B* ≡ ∀ (*p*,*τ*) ∈ *A*. ∃ *τ'*. (*p*,*τ'*) ∈ *B* ∧ *τ* ⊑$_r$ *τ'*

notation (*ASCII*)
lesubstep-type (‹(- /{<='--} -)› [*50*, *0*, *51*] *50*)

primrec *pluslussub* :: *'a list* ⇒ (*'a* ⇒ *'a* ⇒ *'a*) ⇒ *'a* ⇒ *'a* (‹(- /⨆_ -)› [*65*, *0*, *66*] *65*)
where
pluslussub [] *f y* = *y*
| *pluslussub* (*x*#*xs*) *f y* = *pluslussub xs f* (*x* ⊔*f y*)
definition *bounded* :: *'s step-type* ⇒ *nat* ⇒ *bool*
where
bounded step n ⟷ (∀ *p*<*n*. ∀ *τ*. ∀ (*q*,*τ'*) ∈ *set* (*step p τ*). *q*<*n*)

definition *pres-type* :: *'s step-type* ⇒ *nat* ⇒ *'s set* ⇒ *bool*
where
pres-type step n A ⟷ (∀ *τ*∈*A*. ∀ *p*<*n*. ∀ (*q*,*τ'*) ∈ *set* (*step p τ*). *τ'* ∈ *A*)

definition *mono* :: *'s ord* ⇒ *'s step-type* ⇒ *nat* ⇒ *'s set* ⇒ *bool*
where
mono r step n A ⟷
(∀ *τ p τ'*. *τ* ∈ *A* ∧ *p*<*n* ∧ *τ* ⊑*r τ'* ⟶ *set* (*step p τ*) {⊑*r*} *set* (*step p τ'*))

lemma [*iff*]: {} {⊑*r*} *B*

lemma [*iff*]: (*A* {⊑*r*} {}) = (*A* = {})

lemma *lesubstep-union*:
⟦ *A*1 {⊑*r*} *B*1; *A*2 {⊑*r*} *B*2 ⟧ ⟹ *A*1 ∪ *A*2 {⊑*r*} *B*1 ∪ *B*2

lemma *pres-typeD*:
⟦ *pres-type step n A*; *s*∈*A*; *p*<*n*; (*q*,*s'*)∈*set* (*step p s*) ⟧ ⟹ *s'* ∈ *A*
lemma *monoD*:
⟦ *mono r step n A*; *p* < *n*; *s*∈*A*; *s* ⊑*r t* ⟧ ⟹ *set* (*step p s*) {⊑*r*} *set* (*step p t*)
lemma *boundedD*:
⟦ *bounded step n*; *p* < *n*; (*q*,*t*) ∈ *set* (*step p xs*) ⟧ ⟹ *q* < *n*
lemma *lesubstep-type-refl* [*simp*, *intro*]:
(⋀*x*. *x* ⊑*r x*) ⟹ *A* {⊑*r*} *A*
lemma *lesub-step-typeD*:
A {⊑*r*} *B* ⟹ (*x*,*y*) ∈ *A* ⟹ ∃ *y'*. (*x*, *y'*) ∈ *B* ∧ *y* ⊑*r y'*

lemma *list-update-le-listI* [*rule-format*]:
set xs ⊆ *A* ⟶ *set ys* ⊆ *A* ⟶ *xs* [⊑*r*] *ys* ⟶ *p* < *size xs* ⟶
x ⊑*r ys*!*p* ⟶ *semilat*(*A*,*r*,*f*) ⟶ *x*∈*A* ⟶
xs[*p* := *x* ⊔*f xs*!*p*] [⊑*r*] *ys*
lemma *plusplus-closed*: **assumes** *Semilat A r f* **shows**
⋀*y*. ⟦ *set x* ⊆ *A*; *y* ∈ *A*⟧ ⟹ *x* ⨆*f y* ∈ *A*
lemma (**in** *Semilat*) *pp-ub2*:
⋀*y*. ⟦ *set x* ⊆ *A*; *y* ∈ *A*⟧ ⟹ *y* ⊑*r x* ⨆*f y*

lemma (**in** *Semilat*) *pp-ub1*:
shows ⋀*y*. ⟦*set ls* ⊆ *A*; *y* ∈ *A*; *x* ∈ *set ls*⟧ ⟹ *x* ⊑*r ls* ⨆*f y*

lemma (**in** *Semilat*) *pp-lub*:
assumes *z*: *z* ∈ *A*
shows

$\bigwedge y.\ y \in A \Longrightarrow set\ xs \subseteq A \Longrightarrow \forall x \in set\ xs.\ x \sqsubseteq_r z \Longrightarrow y \sqsubseteq_r z \Longrightarrow xs \bigsqcup_f y \sqsubseteq_r z$

lemma *ub1'*: **assumes** *Semilat A r f*
shows $[\![\forall (p,s) \in set\ S.\ s \in A;\ y \in A;\ (a,b) \in set\ S]\!]$
$\Longrightarrow b \sqsubseteq_r map\ snd\ [(p',\ t') \leftarrow S.\ p' = a] \bigsqcup_f y$

lemma *plusplus-empty*:
$\forall s'.\ (q,\ s') \in set\ S \longrightarrow s' \sqcup_f ss\ !\ q = ss\ !\ q \Longrightarrow$
$(map\ snd\ [(p',\ t') \leftarrow S.\ p' = q] \bigsqcup_f ss\ !\ q) = ss\ !\ q$

end

2.8 Lifting the Typing Framework to err, app, and eff

theory *Typing-Framework-err*
imports
Typing-Framework
SemilatAlg
begin

definition *wt-err-step* :: *'s ord* ⇒ *'s err step-type* ⇒ *'s err list* ⇒ *bool*
where
wt-err-step r step τs ⟷ *wt-step* (*Err.le r*) *Err step τs*

definition *wt-app-eff* :: *'s ord* ⇒ (*nat* ⇒ *'s* ⇒ *bool*) ⇒ *'s step-type* ⇒ *'s list* ⇒ *bool*
where
wt-app-eff r app step τs ⟷
(∀ *p* < *size τs*. *app p* (*τs*!*p*) ∧ (∀ (*q*,*τ*) ∈ *set* (*step p* (*τs*!*p*)). *τ* <=-*r τs*!*q*))

definition *map-snd* :: (*'b* ⇒ *'c*) ⇒ (*'a* × *'b*) *list* ⇒ (*'a* × *'c*) *list*
where
map-snd f = *map* (λ(*x*,*y*). (*x*, *f y*))

definition *error* :: *nat* ⇒ (*nat* × *'a err*) *list*
where
error n = *map* (λ*x*. (*x*,*Err*)) [*0*..<*n*]

definition *err-step* :: *nat* ⇒ (*nat* ⇒ *'s* ⇒ *bool*) ⇒ *'s step-type* ⇒ *'s err step-type*
where
err-step n app step p t =
(*case t of*
Err ⇒ *error n*
| *OK τ* ⇒ *if app p τ then map-snd OK* (*step p τ*) *else error n*)

definition *app-mono* :: *'s ord* ⇒ (*nat* ⇒ *'s* ⇒ *bool*) ⇒ *nat* ⇒ *'s set* ⇒ *bool*
where
app-mono r app n A ⟷
(∀ *s p t*. *s* ∈ *A* ∧ *p* < *n* ∧ *s* ⊑$_r$ *t* ⟶ *app p t* ⟶ *app p s*)

lemmas *err-step-defs* = *err-step-def map-snd-def error-def*

lemma *bounded-err-stepD*:
⟦ *bounded* (*err-step n app step*) *n*;
p < *n*; *app p a*; (*q*,*b*) ∈ *set* (*step p a*) ⟧ ⟹ *q* < *n*

lemma *in-map-sndD*: (*a*,*b*) ∈ *set* (*map-snd f xs*) ⟹ ∃ *b'*. (*a*,*b'*) ∈ *set xs*

lemma *bounded-err-stepI*:
∀ *p*. *p* < *n* ⟶ (∀ *s*. *ap p s* ⟶ (∀ (*q*,*s'*) ∈ *set* (*step p s*). *q* < *n*))
⟹ *bounded* (*err-step n ap step*) *n*

lemma *bounded-lift*:
bounded step n ⟹ *bounded* (*err-step n app step*) *n*

lemma *le-list-map-OK* [*simp*]:
⋀*b*. (*map OK a* [⊑$_{Err.le\ r}$] *map OK b*) = (*a* [⊑$_r$] *b*)

lemma *map-snd-lessI*:
set xs {⊑$_r$} *set ys* ⟹ *set* (*map-snd OK xs*) {⊑$_{Err.le\ r}$} *set* (*map-snd OK ys*)

lemma *mono-lift*:
⟦ *order r*; *app-mono r app n A*; *bounded* (*err-step n app step*) *n*;
∀ *s p t*. *s* ∈ *A* ∧ *p* < *n* ∧ *s* ⊑$_r$ *t* ⟶ *app p t* ⟶ *set* (*step p s*) {⊑$_r$} *set* (*step p t*) ⟧
⟹ *mono* (*Err.le r*) (*err-step n app step*) *n* (*err A*)
lemma *in-errorD*: (*x*,*y*) ∈ *set* (*error n*) ⟹ *y* = *Err*
lemma *pres-type-lift*:
∀ *s*∈*A*. ∀ *p*. *p* < *n* ⟶ *app p s* ⟶ (∀ (*q*, *s'*)∈*set* (*step p s*). *s'* ∈ *A*)
⟹ *pres-type* (*err-step n app step*) *n* (*err A*)

lemma *wt-err-imp-wt-app-eff*:
assumes *wt*: *wt-err-step r* (*err-step* (*size ts*) *app step*) *ts*
assumes *b*: *bounded* (*err-step* (*size ts*) *app step*) (*size ts*)
shows *wt-app-eff r app step* (*map ok-val ts*)

lemma *wt-app-eff-imp-wt-err*:
assumes *app-eff*: *wt-app-eff r app step ts*
assumes *bounded*: *bounded* (*err-step* (*size ts*) *app step*) (*size ts*)
shows *wt-err-step r* (*err-step* (*size ts*) *app step*) (*map OK ts*)
end

2.9 Kildall's Algorithm

theory *Kildall*
imports *SemilatAlg* *../Basic/Auxiliary*
begin

locale *Kildall-base* =
fixes *s-α* :: *'w* ⇒ *nat set*
and *s-empty* :: *'w*
and *s-is-empty* :: *'w* ⇒ *bool*
and *s-choose* :: *'w* ⇒ *nat*
and *s-remove* :: *nat* ⇒ *'w* ⇒ *'w*
and *s-insert* :: *nat* ⇒ *'w* ⇒ *'w*
begin

primrec *propa* :: *'s binop* ⇒ (*nat* × *'s*) *list* ⇒ *'s list* ⇒ *'w* ⇒ *'s list* ∗ *'w*
where
  propa f [] *τs w* = (*τs*,*w*)
| *propa f* (*q'*#*qs*) *τs w* = (*let* (*q*,*τ*) = *q'*;
                    u = *τ* ⊔*f* *τs*!*q*;
                    w' = (*if u* = *τs*!*q then w else s-insert q w*)
                  in propa f qs (*τs*[*q* := *u*]) *w'*)

definition *iter* :: *'s binop* ⇒ *'s step-type* ⇒ *'s list* ⇒ *'w* ⇒ *'s list* × *'w*
where
  iter f step τs w =
  while (λ(*τs*,*w*). ¬ *s-is-empty w*)
      (λ(*τs*,*w*). *let p* = *s-choose w in propa f* (*step p* (*τs*!*p*)) *τs* (*s-remove p w*))
      (*τs*,*w*)

definition *unstables* :: *'s ord* ⇒ *'s step-type* ⇒ *'s list* ⇒ *'w*
where
  unstables r step τs = *foldr s-insert* (*filter* (λ*p*. ¬*stable r step τs p*) [*0*..<*size τs*]) *s-empty*

definition *kildall* :: *'s ord* ⇒ *'s binop* ⇒ *'s step-type* ⇒ *'s list* ⇒ *'s list*
where *kildall r f step τs* ≡ *fst*(*iter f step τs* (*unstables r step τs*))

primrec *t-α* :: *'s list* × *'w* ⇒ *'s list* × *nat set*
where *t-α* (*τs*, *w*) = (*τs*, *s-α w*)

end

primrec *merges* :: *'s binop* ⇒ (*nat* × *'s*) *list* ⇒ *'s list* ⇒ *'s list*
where
  merges f [] *τs* = *τs*
| *merges f* (*p'*#*ps*) *τs* = (*let* (*p*,*τ*) = *p' in merges f ps* (*τs*[*p* := *τ* ⊔*f* *τs*!*p*]))

locale *Kildall* =
  Kildall-base +
  assumes *empty-spec* [*simp*]: *s-α s-empty* = {}
  and *is-empty-spec* [*simp*]: *s-is-empty A* ⟷ *s-α A* = {}
  and *choose-spec*: *s-α A* ≠ {} ⟹ *s-choose A* ∈ *s-α A*
  and *remove-spec* [*simp*]: *s-α* (*s-remove n A*) = *s-α A* − {*n*}
  and *insert-spec* [*simp*]: *s-α* (*s-insert n A*) = *insert n* (*s-α A*)
begin

lemma *s-α-foldr-s-insert*:
  s-α (*foldr s-insert xs A*) = *foldr insert xs* (*s-α A*)
by(*induct xs arbitrary*: *A*) *simp-all*

lemma *unstables-spec* [*simp*]: *s-α* (*unstables r step τs*) = {*p*. *p* < *size τs* ∧ ¬*stable r step τs p*}
proof −
  have {*p*. *p* < *size τs* ∧ ¬*stable r step τs p*} = *foldr insert* (*filter* (λ*p*. ¬*stable r step τs p*) [*0*..<*size τs*]) {}
    unfolding *foldr-insert-conv-set* **by** *auto*
  thus *?thesis* **by**(*simp add*: *unstables-def s-α-foldr-s-insert*)
qed

end

lemmas [*simp*] = *Let-def Semilat.le-iff-plus-unchanged* [*OF Semilat.intro*, *symmetric*]

lemma (**in** *Semilat*) *nth-merges*:
⋀*ss*. ⟦*p* < *length ss*; *ss* ∈ *list n A*; ∀(*p*,*t*)∈*set ps*. *p*<*n* ∧ *t*∈*A* ⟧ ⟹
(*merges f ps ss*)!*p* = *map snd* [(*p*′,*t*′) ← *ps*. *p*′=*p*] ⨆$_f$ *ss*!*p*
(**is** ⋀*ss*. ⟦-; -; *?steptype ps*⟧ ⟹ *?P ss ps*)

lemma *length-merges* [*simp*]:
⋀*ss*. *size*(*merges f ps ss*) = *size ss*
lemma (**in** *Semilat*) *merges-preserves-type-lemma*:
shows ∀ *xs*. *xs* ∈ *list n A* ⟶ (∀(*p*,*x*) ∈ *set ps*. *p*<*n* ∧ *x*∈*A*)
⟶ *merges f ps xs* ∈ *list n A*
lemma (**in** *Semilat*) *merges-preserves-type* [*simp*]:
⟦ *xs* ∈ *list n A*; ∀(*p*,*x*) ∈ *set ps*. *p*<*n* ∧ *x*∈*A* ⟧
⟹ *merges f ps xs* ∈ *list n A*
by (*simp add*: *merges-preserves-type-lemma*)

lemma (**in** *Semilat*) *merges-incr-lemma*:
∀ *xs*. *xs* ∈ *list n A* ⟶ (∀(*p*,*x*)∈*set ps*. *p*<*size xs* ∧ *x* ∈ *A*) ⟶ *xs* [⊑$_r$] *merges f ps xs*
lemma (**in** *Semilat*) *merges-incr*:
⟦ *xs* ∈ *list n A*; ∀(*p*,*x*)∈*set ps*. *p*<*size xs* ∧ *x* ∈ *A* ⟧
⟹ *xs* [⊑$_r$] *merges f ps xs*
by (*simp add*: *merges-incr-lemma*)

lemma (**in** *Semilat*) *merges-same-conv* [*rule-format*]:
(∀ *xs*. *xs* ∈ *list n A* ⟶ (∀(*p*,*x*)∈*set ps*. *p*<*size xs* ∧ *x*∈*A*) ⟶
(*merges f ps xs* = *xs*) = (∀(*p*,*x*)∈*set ps*. *x* ⊑$_r$ *xs*!*p*))

lemma (**in** *Semilat*) *list-update-le-listI* [*rule-format*]:
set xs ⊆ *A* ⟶ *set ys* ⊆ *A* ⟶ *xs* [⊑$_r$] *ys* ⟶ *p* < *size xs* ⟶
x ⊑$_r$ *ys*!*p* ⟶ *x*∈*A* ⟶ *xs*[*p* := *x* ⨆$_f$ *xs*!*p*] [⊑$_r$] *ys*
lemma (**in** *Semilat*) *merges-pres-le-ub*:
assumes *set ts* ⊆ *A* *set ss* ⊆ *A*
∀(*p*,*t*)∈*set ps*. *t* ⊑$_r$ *ts*!*p* ∧ *t* ∈ *A* ∧ *p* < *size ts* *ss* [⊑$_r$] *ts*
shows *merges f ps ss* [⊑$_r$] *ts*
context *Kildall* **begin**

2.9.1 *propa*

lemma *decomp-propa*:
⋀*ss w*. (∀(*q*,*t*)∈*set qs*. *q* < *size ss*) ⟹
t-α (*propa f qs ss w*) =
(*merges f qs ss*, {*q*. ∃ *t*.(*q*,*t*)∈*set qs* ∧ *t* ⨆$_f$ *ss*!*q* ≠ *ss*!*q*} ∪ *s-α w*)
apply (*induct qs*)
apply *simp*
apply (*simp* (*no-asm*))
apply *clarify*
apply *simp*

apply (*rule conjI*)
apply *blast*
apply (*simp add*: *nth-list-update*)
apply *blast*
done

end

lemma (**in** *Semilat*) *stable-pres-lemma*:
shows ⟦*pres-type step n A*; *bounded step n*;
ss ∈ *list n A*; *p* ∈ *w*; ∀ *q*∈*w*. *q* < *n*;
∀ *q*. *q* < *n* ⟶ *q* ∉ *w* ⟶ *stable r step ss q*; *q* < *n*;
∀ *s'*. (*q*,*s'*) ∈ *set* (*step p* (*ss*!*p*)) ⟶ *s'* ⊔$_f$ *ss*!*q* = *ss*!*q*;
q ∉ *w* ∨ *q* = *p* ⟧
⟹ *stable r step* (*merges f* (*step p* (*ss*!*p*)) *ss*) *q*

lemma (**in** *Semilat*) *merges-bounded-lemma*:
⟦ *mono r step n A*; *bounded step n*;
∀ (*p'*,*s'*) ∈ *set* (*step p* (*ss*!*p*)). *s'* ∈ *A*; *ss* ∈ *list n A*; *ts* ∈ *list n A*; *p* < *n*;
ss [⊑$_r$] *ts*; ∀ *p*. *p* < *n* ⟶ *stable r step ts p* ⟧
⟹ *merges f* (*step p* (*ss*!*p*)) *ss* [⊑$_r$] *ts*

lemma *termination-lemma*: **assumes** *Semilat A r f*
shows ⟦ *ss* ∈ *list n A*; ∀ (*q*,*t*)∈*set qs*. *q*<*n* ∧ *t*∈*A*; *p*∈*w* ⟧ ⟹
ss [⊏$_r$] *merges f qs ss* ∨
merges f qs ss = *ss* ∧ {*q*. ∃ *t*. (*q*,*t*)∈*set qs* ∧ *t* ⊔$_f$ *ss*!*q* ≠ *ss*!*q*} ∪ (*w*−{*p*}) ⊂ *w*
context *Kildall-base* **begin**

definition *s-finite-psubset* :: ('*w* ∗ '*w*) *set*
where *s-finite-psubset* == {(*A*,*B*). *s-α A* < *s-α B* & *finite* (*s-α B*)}

lemma *s-finite-psubset-inv-image*:
s-finite-psubset = *inv-image finite-psubset s-α*
by(*auto simp add*: *s-finite-psubset-def finite-psubset-def*)

lemma *wf-s-finite-psubset* [*simp*]: *wf s-finite-psubset*
unfolding *s-finite-psubset-inv-image* **by** *simp*

end

context *Kildall* **begin**

2.9.2 *iter*

lemma *iter-properties*[*rule-format*]: **assumes** *Semilat A r f*
shows ⟦ *acc A r*; *pres-type step n A*; *mono r step n A*;
bounded step n; ∀ *p*∈*s-α w0*. *p* < *n*; *ss0* ∈ *list n A*;
∀ *p*<*n*. *p* ∉ *s-α w0* ⟶ *stable r step ss0 p* ⟧ ⟹
t-α (*iter f step ss0 w0*) = (*ss'*,*w'*)
⟶
ss' ∈ *list n A* ∧ *stables r step ss'* ∧ *ss0* [⊑$_r$] *ss'* ∧
(∀ *ts*∈*list n A*. *ss0* [⊑$_r$] *ts* ∧ *stables r step ts* ⟶ *ss'* [⊑$_r$] *ts*)
lemma *kildall-properties*: **assumes** *Semilat A r f*
shows ⟦ *acc A r*; *pres-type step n A*; *mono r step n A*;

bounded step n; ss0 ∈ list n A ⟧ ⟹
kildall r f step ss0 ∈ list n A ∧
stables r step (kildall r f step ss0) ∧
ss0 [⊑$_r$] kildall r f step ss0 ∧
(∀ ts∈list n A. ss0 [⊑$_r$] ts ∧ stables r step ts ⟶
kildall r f step ss0 [⊑$_r$] ts)
end

interpretation *Kildall set [] λxs. xs = [] hd removeAll Cons*
by(*unfold-locales*) *auto*

lemmas *kildall-code* [*code*] =
kildall-def
Kildall-base.propa.simps
Kildall-base.iter-def
Kildall-base.unstables-def
Kildall-base.kildall-def

end

2.10 The Lightweight Bytecode Verifier

theory *LBVSpec*
imports *SemilatAlg Opt*
begin

type-synonym
's certificate = 's list

primrec *merge :: 's certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒ nat ⇒ (nat × 's) list ⇒ 's ⇒ 's*
where
merge cert f r T pc [] x = x
| merge cert f r T pc (s#ss) x = merge cert f r T pc ss (let (pc',s') = s in
if pc'=pc+1 then s' ⊔$_f$ x
else if s' ⊑$_r$ cert!pc' then x
else T)

definition *wtl-inst :: 's certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒*
's step-type ⇒ nat ⇒ 's ⇒ 's
where
wtl-inst cert f r T step pc s = merge cert f r T pc (step pc s) (cert!(pc+1))

definition *wtl-cert :: 's certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒ 's ⇒*
's step-type ⇒ nat ⇒ 's ⇒ 's
where
wtl-cert cert f r T B step pc s =
(if cert!pc = B then
wtl-inst cert f r T step pc s
else
if s ⊑$_r$ cert!pc then wtl-inst cert f r T step pc (cert!pc) else T)

primrec *wtl-inst-list :: 'a list ⇒ 's certificate ⇒ 's binop ⇒ 's ord ⇒ 's ⇒ 's ⇒*
's step-type ⇒ nat ⇒ 's ⇒ 's

where
  wtl-inst-list [] *cert f r T B step pc s* = *s*
| *wtl-inst-list* (*i*#*is*) *cert f r T B step pc s* =
    (*let s'* = *wtl-cert cert f r T B step pc s in*
      if s' = *T* ∨ *s* = *T then T else wtl-inst-list is cert f r T B step* (*pc*+*1*) *s'*)

definition *cert-ok* :: *'s certificate* ⇒ *nat* ⇒ *'s* ⇒ *'s* ⇒ *'s set* ⇒ *bool*
where
  cert-ok cert n T B A ⟷ (∀ *i* < *n*. *cert*!*i* ∈ *A* ∧ *cert*!*i* ≠ *T*) ∧ (*cert*!*n* = *B*)

definition *bottom* :: *'a ord* ⇒ *'a* ⇒ *bool*
where
  bottom r B ⟷ (∀ *x*. *B* ⊑$_r$ *x*)

locale *lbv* = *Semilat* +
  fixes *T* :: *'a* (‹⊤›)
  fixes *B* :: *'a* (‹⊥›)
  fixes *step* :: *'a step-type*
  assumes *top*: *top r* ⊤
  assumes *T-A*: ⊤ ∈ *A*
  assumes *bot*: *bottom r* ⊥
  assumes *B-A*: ⊥ ∈ *A*

  fixes *merge* :: *'a certificate* ⇒ *nat* ⇒ (*nat* × *'a*) *list* ⇒ *'a* ⇒ *'a*
  defines *mrg-def*: *merge cert* ≡ *LBVSpec.merge cert f r* ⊤

  fixes *wti* :: *'a certificate* ⇒ *nat* ⇒ *'a* ⇒ *'a*
  defines *wti-def*: *wti cert* ≡ *wtl-inst cert f r* ⊤ *step*

  fixes *wtc* :: *'a certificate* ⇒ *nat* ⇒ *'a* ⇒ *'a*
  defines *wtc-def*: *wtc cert* ≡ *wtl-cert cert f r* ⊤ ⊥ *step*

  fixes *wtl* :: *'b list* ⇒ *'a certificate* ⇒ *nat* ⇒ *'a* ⇒ *'a*
  defines *wtl-def*: *wtl ins cert* ≡ *wtl-inst-list ins cert f r* ⊤ ⊥ *step*

lemma (**in** *lbv*) *wti*:
  wti c pc s = *merge c pc* (*step pc s*) (*c*!(*pc*+*1*))

lemma (**in** *lbv*) *wtc*:
  wtc c pc s = (*if c*!*pc* = ⊥ *then wti c pc s else if s* ⊑$_r$ *c*!*pc then wti c pc* (*c*!*pc*) *else* ⊤)

lemma *cert-okD1* [*intro?*]:
  cert-ok c n T B A ⟹ *pc* < *n* ⟹ *c*!*pc* ∈ *A*

lemma *cert-okD2* [*intro?*]:
  cert-ok c n T B A ⟹ *c*!*n* = *B*

lemma *cert-okD3* [*intro?*]:
  cert-ok c n T B A ⟹ *B* ∈ *A* ⟹ *pc* < *n* ⟹ *c*!*Suc pc* ∈ *A*

lemma *cert-okD4* [*intro?*]:
  cert-ok c n T B A ⟹ *pc* < *n* ⟹ *c*!*pc* ≠ *T*

declare *Let-def* [*simp*]

2.10.1 more semilattice lemmas

lemma (**in** *lbv*) *sup-top* [*simp*, *elim*]:
assumes *x*: $x \in A$
shows $x \sqcup_f \top = \top$
lemma (**in** *lbv*) *plusplussup-top* [*simp*, *elim*]:
$set\ xs \subseteq A \Longrightarrow xs \bigsqcup_f \top = \top$
by (*induct xs*) *auto*

lemma (**in** *Semilat*) *pp-ub1'*:
assumes *S*: $snd`set\ S \subseteq A$
assumes *y*: $y \in A$ **and** *ab*: $(a, b) \in set\ S$
shows $b \sqsubseteq_r map\ snd\ [(p', t') \leftarrow S\ .\ p' = a] \bigsqcup_f y$
lemma (**in** *lbv*) *bottom-le* [*simp*, *intro*!]: $\bot \sqsubseteq_r x$
by (*insert bot*) (*simp add*: *bottom-def*)

lemma (**in** *lbv*) *le-bottom* [*simp*]: $x \sqsubseteq_r \bot = (x = \bot)$
by (*blast intro*: *antisym-r*)

2.10.2 merge

lemma (**in** *lbv*) *merge-Nil* [*simp*]:
$merge\ c\ pc\ []\ x = x$ **by** (*simp add*: *mrg-def*)

lemma (**in** *lbv*) *merge-Cons* [*simp*]:
$merge\ c\ pc\ (l\#ls)\ x = merge\ c\ pc\ ls\ (if\ fst\ l{=}pc{+}1\ then\ snd\ l\ +_f\ x$
$else\ if\ snd\ l \sqsubseteq_r c!fst\ l\ then\ x$
$else\ \top)$
by (*simp add*: *mrg-def split-beta*)

lemma (**in** *lbv*) *merge-Err* [*simp*]:
$snd`set\ ss \subseteq A \Longrightarrow merge\ c\ pc\ ss\ \top = \top$
by (*induct ss*) *auto*

lemma (**in** *lbv*) *merge-not-top*:
$\bigwedge x.\ snd`set\ ss \subseteq A \Longrightarrow merge\ c\ pc\ ss\ x \neq \top \Longrightarrow$
$\forall (pc',s') \in set\ ss.\ (pc' \neq pc{+}1 \longrightarrow s' \sqsubseteq_r c!pc')$
(**is** $\bigwedge x.\ ?set\ ss \Longrightarrow ?merge\ ss\ x \Longrightarrow ?P\ ss$)

lemma (**in** *lbv*) *merge-def*:
shows
$\bigwedge x.\ x \in A \Longrightarrow snd`set\ ss \subseteq A \Longrightarrow$
$merge\ c\ pc\ ss\ x =$
$(if\ \forall (pc',s') \in set\ ss.\ pc' \neq pc{+}1 \longrightarrow s' \sqsubseteq_r c!pc'\ then$
$map\ snd\ [(p',t') \leftarrow ss.\ p'{=}pc{+}1] \bigsqcup_f x$
$else\ \top)$
(**is** $\bigwedge x.\ \text{-} \Longrightarrow \text{-} \Longrightarrow ?merge\ ss\ x = ?if\ ss\ x$ **is** $\bigwedge x.\ \text{-} \Longrightarrow \text{-} \Longrightarrow ?P\ ss\ x$)
lemma (**in** *lbv*) *merge-not-top-s*:
assumes *x*: $x \in A$ **and** *ss*: $snd`set\ ss \subseteq A$
assumes *m*: $merge\ c\ pc\ ss\ x \neq \top$

shows *merge c pc ss x* = (*map snd* [(*p'*,*t'*) ← *ss*. *p'*=*pc+1*] $\bigsqcup_f$ *x*)

2.10.3 wtl-inst-list

lemmas [*iff*] = *not-Err-eq*

lemma (**in** *lbv*) *wtl-Nil* [*simp*]: *wtl* [] *c pc s* = *s*
by (*simp add*: *wtl-def*)

lemma (**in** *lbv*) *wtl-Cons* [*simp*]:
wtl (*i*#*is*) *c pc s* =
(*let s'* = *wtc c pc s in if s'* = ⊤ ∨ *s* = ⊤ *then* ⊤ *else wtl is c* (*pc+1*) *s'*)
by (*simp add*: *wtl-def wtc-def*)

lemma (**in** *lbv*) *wtl-Cons-not-top*:
wtl (*i*#*is*) *c pc s* ≠ ⊤ =
(*wtc c pc s* ≠ ⊤ ∧ *s* ≠ *T* ∧ *wtl is c* (*pc+1*) (*wtc c pc s*) ≠ ⊤)
by (*auto simp del*: *split-paired-Ex*)

lemma (**in** *lbv*) *wtl-top* [*simp*]: *wtl ls c pc* ⊤ = ⊤
by (*cases ls*) *auto*

lemma (**in** *lbv*) *wtl-not-top*:
wtl ls c pc s ≠ ⊤ ⟹ *s* ≠ ⊤
by (*cases s*=⊤) *auto*

lemma (**in** *lbv*) *wtl-append* [*simp*]:
⋀*pc s*. *wtl* (*a*@*b*) *c pc s* = *wtl b c* (*pc+length a*) (*wtl a c pc s*)
by (*induct a*) *auto*

lemma (**in** *lbv*) *wtl-take*:
wtl is c pc s ≠ ⊤ ⟹ *wtl* (*take pc' is*) *c pc s* ≠ ⊤
(**is** *?wtl is* ≠ - ⟹ -)
lemma *take-Suc*:
∀ *n*. *n* < *length l* ⟶ *take* (*Suc n*) *l* = (*take n l*)@[*l*!*n*] (**is** *?P l*)
lemma (**in** *lbv*) *wtl-Suc*:
assumes *suc*: *pc+1* < *length is*
assumes *wtl*: *wtl* (*take pc is*) *c 0 s* ≠ ⊤
shows *wtl* (*take* (*pc+1*) *is*) *c 0 s* = *wtc c pc* (*wtl* (*take pc is*) *c 0 s*)
lemma (**in** *lbv*) *wtl-all*:
assumes *all*: *wtl is c 0 s* ≠ ⊤ (**is** *?wtl is* ≠ -)
assumes *pc*: *pc* < *length is*
shows *wtc c pc* (*wtl* (*take pc is*) *c 0 s*) ≠ ⊤

2.10.4 preserves-type

lemma (**in** *lbv*) *merge-pres*:
assumes *s0*: *snd'set ss* ⊆ *A* **and** *x*: *x* ∈ *A*
shows *merge c pc ss x* ∈ *A*
lemma *pres-typeD2*:
pres-type step n A ⟹ *s* ∈ *A* ⟹ *p* < *n* ⟹ *snd'set* (*step p s*) ⊆ *A*
by *auto* (*drule pres-typeD*)

lemma (**in** *lbv*) *wti-pres* [*intro?*]:

assumes *pres*: *pres-type step n A*
assumes *cert*: $c!(pc{+}1) \in A$
assumes *s-pc*: $s \in A$ $pc < n$
shows *wti c pc s* $\in A$
lemma (**in** *lbv*) *wtc-pres*:
assumes *pres-type step n A*
assumes $c!pc \in A$ **and** $c!(pc{+}1) \in A$
assumes $s \in A$ **and** $pc < n$
shows *wtc c pc s* $\in A$
lemma (**in** *lbv*) *wtl-pres*:
assumes *pres*: *pres-type step* (*length is*) *A*
assumes *cert*: *cert-ok c* (*length is*) $\top$ $\bot$ *A*
assumes *s*: $s \in A$
assumes *all*: *wtl is c 0 s* $\neq \top$
shows *pc* $<$ *length is* $\Longrightarrow$ *wtl* (*take pc is*) *c 0 s* $\in A$
(**is** *?len pc* $\Longrightarrow$ *?wtl pc* $\in A$)
end

2.11 Correctness of the LBV

theory *LBVCorrect*
imports *LBVSpec Typing-Framework*
begin

locale *lbvs* = *lbv* +
fixes s_0 :: *'a*
fixes *c* :: *'a list*
fixes *ins* :: *'b list*
fixes τs :: *'a list*
defines *phi-def*:
$\tau s \equiv$ *map* (λpc. *if* $c!pc = \bot$ *then wtl* (*take pc ins*) *c 0* s_0 *else* $c!pc$)
[*0..<size ins*]

assumes *bounded*: *bounded step* (*size ins*)
assumes *cert*: *cert-ok c* (*size ins*) $\top$ $\bot$ *A*
assumes *pres*: *pres-type step* (*size ins*) *A*

lemma (**in** *lbvs*) *phi-None* [*intro?*]:
$[\![$ *pc* $<$ *size ins*; $c!pc = \bot$ $]\!] \Longrightarrow \tau s!pc =$ *wtl* (*take pc ins*) *c 0* s_0
lemma (**in** *lbvs*) *phi-Some* [*intro?*]:
$[\![$ *pc* $<$ *size ins*; $c!pc \neq \bot$ $]\!] \Longrightarrow \tau s!pc = c!pc$
lemma (**in** *lbvs*) *phi-len* [*simp*]: *size* $\tau s =$ *size ins*
lemma (**in** *lbvs*) *wtl-suc-pc*:
assumes *all*: *wtl ins c 0* $s_0 \neq \top$
assumes *pc*: *pc+1* $<$ *size ins*
shows *wtl* (*take* (*pc+1*) *ins*) *c 0* $s_0 \sqsubseteq_r \tau s!(pc{+}1)$
lemma (**in** *lbvs*) *wtl-stable*:
assumes *wtl*: *wtl ins c 0* $s_0 \neq \top$
assumes s_0: $s_0 \in A$ **and** *pc*: *pc* $<$ *size ins*
shows *stable r step* τs *pc*
lemma (**in** *lbvs*) *phi-not-top*:
assumes *wtl*: *wtl ins c 0* $s_0 \neq \top$ **and** *pc*: *pc* $<$ *size ins*
shows $\tau s!pc \neq \top$

lemma (**in** *lbvs*) *phi-in-A*:
assumes *wtl*: *wtl ins c 0* $s_0 \neq \top$ **and** s_0: $s_0 \in A$
shows $\tau s \in$ *list* (*size ins*) *A*
lemma (**in** *lbvs*) *phi0*:
assumes *wtl*: *wtl ins c 0* $s_0 \neq \top$ **and** *0*: *0* $<$ *size ins*
shows $s_0 \sqsubseteq_r \tau s!0$

theorem (**in** *lbvs*) *wtl-sound*:
assumes *wtl*: *wtl ins c 0* $s_0 \neq \top$ **and** s_0: $s_0 \in A$
shows $\exists \tau s.$ *wt-step r* $\top$ *step* τs

theorem (**in** *lbvs*) *wtl-sound-strong*:
assumes *wtl*: *wtl ins c 0* $s_0 \neq \top$
assumes s_0: $s_0 \in A$ **and** *ins*: *0* $<$ *size ins*
shows $\exists \tau s \in$ *list* (*size ins*) *A*. *wt-step r* $\top$ *step* $\tau s \wedge s_0 \sqsubseteq_r \tau s!0$
end

2.12 Completeness of the LBV

theory *LBVComplete*
imports *LBVSpec Typing-Framework*
begin

definition *is-target* :: *'s step-type* $\Rightarrow$ *'s list* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
is-target step τs *pc'* $\longleftrightarrow$ ($\exists$ *pc s'*. *pc'* $\neq$ *pc+1* $\wedge$ *pc* $<$ *size* $\tau s \wedge$ (*pc'*,*s'*) $\in$ *set* (*step pc* ($\tau s!pc$)))

definition *make-cert* :: *'s step-type* $\Rightarrow$ *'s list* $\Rightarrow$ *'s* $\Rightarrow$ *'s certificate* **where**
make-cert step τs *B* = *map* (λ*pc*. *if is-target step* τs *pc then* $\tau s!pc$ *else B*) [*0..<size* τs] @ [*B*]

lemma [*code*]:
is-target step τs *pc'* =
list-ex (λ*pc*. *pc'* $\neq$ *pc+1* $\wedge$ *List.member* (*map fst* (*step pc* ($\tau s!pc$))) *pc'*) [*0..<size* τs]
locale *lbvc* = *lbv* +
fixes τs :: *'a list*
fixes *c* :: *'a list*
defines *cert-def*: *c* $\equiv$ *make-cert step* τs $\bot$

assumes *mono*: *mono r step* (*size* τs) *A*
assumes *pres*: *pres-type step* (*size* τs) *A*
assumes τs: $\forall$ *pc* $<$ *size* τs. $\tau s!pc \in A \wedge \tau s!pc \neq \top$
assumes *bounded*: *bounded step* (*size* τs)

assumes *B-neq-T*: $\bot \neq \top$

lemma (**in** *lbvc*) *cert*: *cert-ok c* (*size* τs) $\top$ $\bot$ *A*
lemmas [*simp del*] = *split-paired-Ex*

lemma (**in** *lbvc*) *cert-target* [*intro?*]:
⟦ (*pc'*,*s'*) $\in$ *set* (*step pc* ($\tau s!pc$));
pc' $\neq$ *pc+1*; *pc* $<$ *size* τs; *pc'* $<$ *size* τs ⟧
$\Longrightarrow$ *c*!*pc'* = $\tau s!pc'$
lemma (**in** *lbvc*) *cert-approx* [*intro?*]:

$[\![\ pc < size\ \tau s;\ c!pc \neq \bot\ ]\!] \Longrightarrow c!pc = \tau s!pc$
lemma (**in** *lbv*) *le-top* [*simp*, *intro*]: x *<=-r* $\top$
lemma (**in** *lbv*) *merge-mono*:
assumes *less*: *set* ss_2 $\{\sqsubseteq_r\}$ *set* ss_1
assumes x: $x \in A$
assumes ss_1: *snd'set* $ss_1 \subseteq A$
assumes ss_2: *snd'set* $ss_2 \subseteq A$
shows *merge c pc* ss_2 $x \sqsubseteq_r$ *merge c pc* ss_1 x (**is** $?s_2 \sqsubseteq_r ?s_1$)
lemma (**in** *lbvc*) *wti-mono*:
assumes *less*: $s_2 \sqsubseteq_r s_1$
assumes *pc*: $pc < size\ \tau s$ **and** s_1: $s_1 \in A$ **and** s_2: $s_2 \in A$
shows *wti c pc* $s_2 \sqsubseteq_r$ *wti c pc* s_1 (**is** $?s_2' \sqsubseteq_r ?s_1'$)
lemma (**in** *lbvc*) *wtc-mono*:
assumes *less*: $s_2 \sqsubseteq_r s_1$
assumes *pc*: $pc < size\ \tau s$ **and** s_1: $s_1 \in A$ **and** s_2: $s_2 \in A$
shows *wtc c pc* $s_2 \sqsubseteq_r$ *wtc c pc* s_1 (**is** $?s_2' \sqsubseteq_r ?s_1'$)
lemma (**in** *lbv*) *top-le-conv* [*simp*]: $\top \sqsubseteq_r x = (x = \top)$
lemma (**in** *lbv*) *neq-top* [*simp*, *elim*]: $[\![\ x \sqsubseteq_r y;\ y \neq \top\ ]\!] \Longrightarrow x \neq \top$
lemma (**in** *lbvc*) *stable-wti*:
assumes *stable*: *stable r step* τs *pc* **and** *pc*: $pc < size\ \tau s$
shows *wti c pc* $(\tau s!pc) \neq \top$
lemma (**in** *lbvc*) *wti-less*:
assumes *stable*: *stable r step* τs *pc* **and** *suc-pc*: *Suc pc* $< size\ \tau s$
shows *wti c pc* $(\tau s!pc) \sqsubseteq_r \tau s!Suc\ pc$ (**is** *?wti* $\sqsubseteq_r$ -)
lemma (**in** *lbvc*) *stable-wtc*:
assumes *stable*: *stable r step* τs *pc* **and** *pc*: $pc < size\ \tau s$
shows *wtc c pc* $(\tau s!pc) \neq \top$
lemma (**in** *lbvc*) *wtc-less*:
assumes *stable*: *stable r step* τs *pc* **and** *suc-pc*: *Suc pc* $< size\ \tau s$
shows *wtc c pc* $(\tau s!pc) \sqsubseteq_r \tau s!Suc\ pc$ (**is** *?wtc* $\sqsubseteq_r$ -)
lemma (**in** *lbvc*) *wt-step-wtl-lemma*:
assumes *wt-step*: *wt-step r* $\top$ *step* τs
shows $\bigwedge pc\ s.\ pc+size\ ls = size\ \tau s \Longrightarrow s \sqsubseteq_r \tau s!pc \Longrightarrow s \in A \Longrightarrow s \neq \top \Longrightarrow$
wtl ls c pc s $\neq \top$
(**is** $\bigwedge pc\ s.$ - $\Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ *?wtl ls pc s* $\neq$ -)
theorem (**in** *lbvc*) *wtl-complete*:
assumes *wt*: *wt-step r* $\top$ *step* τs
assumes *s*: $s \sqsubseteq_r \tau s!0\ s \in A\ s \neq \top$ **and** *eq*: *size ins* $= size\ \tau s$
shows *wtl ins c 0 s* $\neq \top$
end

Chapter 3

Concepts for all JinjaThreads Languages

3.1 JinjaThreads types

theory *Type*
imports
  ../Basic/Auxiliary
begin

type-synonym *cname* = *String.literal* — class names
type-synonym *mname* = *String.literal* — method name
type-synonym *vname* = *String.literal* — names for local/field variables

definition *Object* :: *cname*
where *Object* ≡ *STR ''java/lang/Object''*

definition *Thread* :: *cname*
where *Thread* ≡ *STR ''java/lang/Thread''*

definition *Throwable* :: *cname*
where *Throwable* ≡ *STR ''java/lang/Throwable''*

definition *this* :: *vname*
where *this* ≡ *STR ''this''*

definition *run* :: *mname*
where *run* ≡ *STR ''run() V''*

definition *start* :: *mname*
where *start* ≡ *STR ''start() V''*

definition *wait* :: *mname*
where *wait* ≡ *STR ''wait() V''*

definition *notify* :: *mname*
where *notify* ≡ *STR ''notify() V''*

definition *notifyAll* :: *mname*

where *notifyAll* ≡ *STR ''notifyAll()V''*

definition *join* :: *mname*
where *join* ≡ *STR ''join()V''*

definition *interrupt* :: *mname*
where *interrupt* ≡ *STR ''interrupt()V''*

definition *isInterrupted* :: *mname*
where *isInterrupted* ≡ *STR ''isInterrupted()Z''*

definition *hashcode* :: *mname*
where *hashcode* = *STR ''hashCode()I''*

definition *clone* :: *mname*
where *clone* = *STR ''clone()Ljava/lang/Object;''*

definition *print* :: *mname*
where *print* = *STR ''~print(I)V''*

definition *currentThread* :: *mname*
where *currentThread* = *STR ''~Thread.currentThread()Ljava/lang/Thread;''*

definition *interrupted* :: *mname*
where *interrupted* = *STR ''~Thread.interrupted()Z''*

definition *yield* :: *mname*
where *yield* = *STR ''~Thread.yield()V''*

lemmas *identifier-name-defs* [*code-unfold*] =
this-def run-def start-def wait-def notify-def notifyAll-def join-def interrupt-def isInterrupted-def
hashcode-def clone-def print-def currentThread-def interrupted-def yield-def

lemma *Object-Thread-Throwable-neq* [*simp*]:
Thread ≠ *Object Object* ≠ *Thread*
Object ≠ *Throwable Throwable* ≠ *Object*
Thread ≠ *Throwable Throwable* ≠ *Thread*
by(*auto simp add*: *Thread-def Object-def Throwable-def*)

lemma *synth-method-names-neq-aux*:
start ≠ *wait start* ≠ *notify start* ≠ *notifyAll start* ≠ *join start* ≠ *interrupt start* ≠ *isInterrupted*
start ≠ *hashcode start* ≠ *clone start* ≠ *print start* ≠ *currentThread*
start ≠ *interrupted start* ≠ *yield start* ≠ *run*
wait ≠ *notify wait* ≠ *notifyAll wait* ≠ *join wait* ≠ *interrupt wait* ≠ *isInterrupted*
wait ≠ *hashcode wait* ≠ *clone wait* ≠ *print wait* ≠ *currentThread*
wait ≠ *interrupted wait* ≠ *yield wait* ≠ *run*
notify ≠ *notifyAll notify* ≠ *join notify* ≠ *interrupt notify* ≠ *isInterrupted*
notify ≠ *hashcode notify* ≠ *clone notify* ≠ *print notify* ≠ *currentThread*
notify ≠ *interrupted notify* ≠ *yield notify* ≠ *run*
notifyAll ≠ *join notifyAll* ≠ *interrupt notifyAll* ≠ *isInterrupted*
notifyAll ≠ *hashcode notifyAll* ≠ *clone notifyAll* ≠ *print notifyAll* ≠ *currentThread*
notifyAll ≠ *interrupted notifyAll* ≠ *yield notifyAll* ≠ *run*

join ≠ *interrupt join* ≠ *isInterrupted*
join ≠ *hashcode join* ≠ *clone join* ≠ *print join* ≠ *currentThread*
join ≠ *interrupted join* ≠ *yield join* ≠ *run*
interrupt ≠ *isInterrupted*
interrupt ≠ *hashcode interrupt* ≠ *clone interrupt* ≠ *print interrupt* ≠ *currentThread*
interrupt ≠ *interrupted interrupt* ≠ *yield interrupt* ≠ *run*
isInterrupted ≠ *hashcode isInterrupted* ≠ *clone isInterrupted* ≠ *print isInterrupted* ≠ *currentThread*

isInterrupted ≠ *interrupted isInterrupted* ≠ *yield isInterrupted* ≠ *run*
hashcode ≠ *clone hashcode* ≠ *print hashcode* ≠ *currentThread*
hashcode ≠ *interrupted hashcode* ≠ *yield hashcode* ≠ *run*
clone ≠ *print clone* ≠ *currentThread*
clone ≠ *interrupted clone* ≠ *yield clone* ≠ *run*
print ≠ *currentThread*
print ≠ *interrupted print* ≠ *yield print* ≠ *run*
currentThread ≠ *interrupted currentThread* ≠ *yield currentThread* ≠ *run*
interrupted ≠ *yield interrupted* ≠ *run*
yield ≠ *run*
by(*simp-all add*: *identifier-name-defs*)

lemmas *synth-method-names-neq* [*simp*] = *synth-method-names-neq-aux synth-method-names-neq-aux*[*symmetric*]

— types
datatype *ty*
= *Void* — type of statements
| *Boolean*
| *Integer*
| *NT* — null type
| *Class cname* — class type
| *Array ty* (‹-⌊⌋› *95*) — array type

context
notes [[*inductive-internals*]]
begin

inductive *is-refT* :: *ty* ⇒ *bool* **where**
is-refT NT
| *is-refT* (*Class C*)
| *is-refT* (*A*⌊⌋)

declare *is-refT.intros*[*iff*]

end

lemmas *refTE* [*consumes 1*, *case-names NT Class Array*] = *is-refT.cases*

lemma *not-refTE* [*consumes 1*, *case-names Void Boolean Integer*]:
⟦ ¬*is-refT T*; *T* = *Void* ⟹ *P*; *T* = *Boolean* ⟹ *P*; *T* = *Integer* ⟹ *P* ⟧ ⟹ *P*
by (*cases T*, *auto*)

fun *ground-type* :: *ty* ⇒ *ty* **where**
ground-type (*Array T*) = *ground-type T*
| *ground-type T* = *T*

abbreviation *is-NT-Array* :: *ty* ⇒ *bool* **where**
is-NT-Array T ≡ *ground-type T* = *NT*

primrec *the-Class* :: *ty* ⇒ *cname*
where
the-Class (*Class C*) = *C*

primrec *the-Array* :: *ty* ⇒ *ty*
where
the-Array (*T*⌊⌋) = *T*

datatype *htype* =
Class-type cname
| *Array-type ty nat*

primrec *ty-of-htype* :: *htype* ⇒ *ty*
where
ty-of-htype (*Class-type C*) = *Class C*
| *ty-of-htype* (*Array-type T n*) = *Array T*

primrec *alen-of-htype* :: *htype* ⇒ *nat*
where
alen-of-htype (*Array-type T n*) = *n*

primrec *class-type-of* :: *htype* ⇒ *cname*
where
class-type-of (*Class-type C*) = *C*
| *class-type-of* (*Array-type T n*) = *Object*

fun *class-type-of′* :: *ty* ⇒ *cname option*
where
class-type-of′ (*Class C*) = ⌊*C*⌋
| *class-type-of′* (*Array T*) = ⌊*Object*⌋
| *class-type-of′* - = *None*

lemma *rec-htype-is-case* [*simp*]: *rec-htype* = *case-htype*
by(*auto simp add*: *fun-eq-iff split*: *htype.split*)

lemma *ty-of-htype-eq-convs* [*simp*]:
shows *ty-of-htype-eq-Boolean*: *ty-of-htype hT* ≠ *Boolean*
and *ty-of-htype-eq-Void*: *ty-of-htype hT* ≠ *Void*
and *ty-of-htype-eq-Integer*: *ty-of-htype hT* ≠ *Integer*
and *ty-of-htype-eq-NT*: *ty-of-htype hT* ≠ *NT*
and *ty-of-htype-eq-Class*: *ty-of-htype hT* = *Class C* ⟷ *hT* = *Class-type C*
and *ty-of-htype-eq-Array*: *ty-of-htype hT* = *Array T* ⟷ (∃ *n*. *hT* = *Array-type T n*)
by(*case-tac* [!] *hT*) *simp-all*

lemma *class-type-of-eq*:
class-type-of hT =
(*case hT of Class-type C* ⇒ *C* | *Array-type T n* ⇒ *Object*)
by(*simp split*: *htype.split*)

lemma *class-type-of′-ty-of-htype* [*simp*]:

class-type-of' (*ty-of-htype hT*) = ⌊*class-type-of hT*⌋
by(*cases hT*) *simp-all*

fun *is-Array* :: *ty* ⇒ *bool*
where
is-Array (*Array T*) = *True*
| *is-Array* - = *False*

lemma *is-Array-conv* [*simp*]: *is-Array T* ⟷ (∃ *U*. *T* = *Array U*)
by(*cases T*) *simp-all*

fun *is-Class* :: *ty* ⇒ *bool*
where
is-Class (*Class C*) = *True*
| *is-Class* - = *False*

lemma *is-Class-conv* [*simp*]: *is-Class T* ⟷ (∃ *C*. *T* = *Class C*)
by(*cases T*) *simp-all*

3.1.1 Code generator setup

code-pred *is-refT* .

end

3.2 Class Declarations and Programs

theory *Decl*
imports
Type
begin

type-synonym *volatile* = *bool*

record *fmod* =
volatile :: *volatile*

type-synonym *fdecl* = *vname* × *ty* × *fmod* — field declaration
type-synonym *'m mdecl* = *mname* × *ty list* × *ty* × *'m* — method = name, arg. types, return type, body
type-synonym *'m mdecl'* = *mname* × *ty list* × *ty* × *'m option* — method = name, arg. types, return type, possible body
type-synonym *'m class* = *cname* × *fdecl list* × *'m mdecl' list* — class = superclass, fields, methods
type-synonym *'m cdecl* = *cname* × *'m class* — class declaration

datatype
'm prog = *Program 'm cdecl list*

translations
(*type*) *fdecl* <= (*type*) *String.literal* × *ty* × *fmod*
(*type*) *'c mdecl* <= (*type*) *String.literal* × *ty list* × *ty* × *'c*
(*type*) *'c mdecl'* <= (*type*) *String.literal* × *ty list* × *ty* × *'c option*
(*type*) *'c class* <= (*type*) *String.literal* × *fdecl list* × (*'c mdecl*) *list*

(*type*) *'c cdecl* <= (*type*) *String.literal* × (*'c class*)

notation (*input*) *None* (‹*Native*›)

primrec *classes* :: *'m prog* ⇒ *'m cdecl list*
where
classes (*Program P*) = *P*

primrec *class* :: *'m prog* ⇒ *cname* ⇀ *'m class*
where
class (*Program p*) = *map-of p*

locale *prog* =
fixes *P* :: *'m prog*

definition *is-class* :: *'m prog* ⇒ *cname* ⇒ *bool*
where
is-class P C ≡ *class P C* ≠ *None*

lemma *finite-is-class*: *finite* {*C*. *is-class P C*}
primrec *is-type* :: *'m prog* ⇒ *ty* ⇒ *bool*
where
is-type-void: *is-type P Void* = *True*
| *is-type-bool*: *is-type P Boolean* = *True*
| *is-type-int*: *is-type P Integer* = *True*
| *is-type-nt*: *is-type P NT* = *True*
| *is-type-class*: *is-type P* (*Class C*) = *is-class P C*
| *is-type-array*: *is-type P* (*A*⌊⌉) = (*case ground-type A of NT* ⇒ *False* | *Class C* ⇒ *is-class P C* | - ⇒ *True*)

lemma *is-type-ArrayD*: *is-type P* (*T*⌊⌉) ⟹ *is-type P T*
by(*induct T*) *auto*

lemma *is-type-ground-type*:
is-type P T ⟹ *is-type P* (*ground-type T*)
by(*induct T*)(*auto*, *metis is-type-ArrayD is-type-array*)

abbreviation *types* :: *'m prog* ⇒ *ty set*
where *types P* ≡ {*T*. *is-type P T*}

abbreviation *is-htype* :: *'m prog* ⇒ *htype* ⇒ *bool*
where *is-htype P hT* ≡ *is-type P* (*ty-of-htype hT*)

3.2.1 Code generation

lemma *is-class-intros* [*code-pred-intro*]:
class P C ≠ *None* ⟹ *is-class P C*
by(*auto simp add*: *is-class-def*)

code-pred
(*modes*: *i* ⇒ *i* ⇒ *bool*)
is-class
unfolding *is-class-def* **by** *simp*

declare *is-class-def*[*code*]

end

3.3 Relations between Jinja Types

theory *TypeRel*
imports
Decl
begin

3.3.1 The subclass relations

inductive *subcls1* :: *'m prog* ⇒ *cname* ⇒ *cname* ⇒ *bool* (‹- ⊢ - ≺¹ -› [*71*, *71*, *71*] *70*)
for *P* :: *'m prog*
where *subcls1I*: ⟦ *class P C* = *Some* (*D*, *rest*); *C* ≠ *Object* ⟧ ⟹ *P* ⊢ *C* ≺¹ *D*

abbreviation *subcls* :: *'m prog* ⇒ *cname* ⇒ *cname* ⇒ *bool* (‹- ⊢ - ⪯* -› [*71*,*71*,*71*] *70*)
where *P* ⊢ *C* ⪯* *D* ≡ (*subcls1 P*)** *C D*

lemma *subcls1D*:
P ⊢ *C* ≺¹ *D* ⟹ *C* ≠ *Object* ∧ (∃*fs ms*. *class P C* = *Some* (*D*,*fs*,*ms*))
by(*auto elim*: *subcls1.cases*)

lemma *Object-subcls1* [*iff*]: ¬ *P* ⊢ *Object* ≺¹ *C*
by(*simp add*: *subcls1.simps*)

lemma *Object-subcls-conv* [*iff*]: (*P* ⊢ *Object* ⪯* *C*) = (*C* = *Object*)
by(*auto elim*: *converse-rtranclpE*)

lemma *finite-subcls1*: *finite* {(*C*, *D*). *P* ⊢ *C* ≺¹ *D*}
proof −
let *?A* = *SIGMA C*:{*C*. *is-class P C*}. {*D*. *C*≠*Object* ∧ *fst* (*the* (*class P C*))=*D*}
have *finite ?A* **by**(*rule finite-SigmaI* [*OF finite-is-class*]) *auto*
also have *?A* = {(*C*, *D*). *P* ⊢ *C* ≺¹ *D*}
by(*fastforce simp*:*is-class-def dest*: *subcls1D elim*: *subcls1I*)
finally show *?thesis* **.**
qed

lemma *finite-subcls1'*:
finite ({(*D*, *C*). *P* ⊢ *C* ≺¹ *D*})
by(*subst finite-converse*[*symmetric*])
(*simp add*: *converse-unfold finite-subcls1 del*: *finite-converse*)

lemma *subcls-is-class*: (*subcls1 P*)++ *C D* ⟹ *is-class P C*
by(*auto elim*: *converse-tranclpE dest*!: *subcls1D simp add*: *is-class-def*)

lemma *subcls-is-class1*: ⟦ *P* ⊢ *C* ⪯* *D*; *is-class P D* ⟧ ⟹ *is-class P C*
by(*auto elim*: *converse-rtranclpE dest*!: *subcls1D simp add*: *is-class-def*)

3.3.2 The subtype relations

inductive *widen* :: *'m prog* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹- ⊢ - ≤ -› [*71*,*71*,*71*] *70*)
for *P* :: *'m prog*

where
  widen-refl[*iff*]: $P \vdash T \leq T$
| *widen-subcls*: $P \vdash C \preceq^* D \Longrightarrow P \vdash Class\ C \leq Class\ D$
| *widen-null*[*iff*]: $P \vdash NT \leq Class\ C$
| *widen-null-array*[*iff*]: $P \vdash NT \leq Array\ A$
| *widen-array-object*: $P \vdash Array\ A \leq Class\ Object$
| *widen-array-array*: $P \vdash A \leq B \Longrightarrow P \vdash Array\ A \leq Array\ B$

abbreviation
  widens :: *'m prog* ⇒ *ty list* ⇒ *ty list* ⇒ *bool* (‹- ⊢ - [≤] -› [*71*,*71*,*71*] *70*)
where
  $P \vdash Ts\ [\leq]\ Ts' == list\text{-}all2\ (widen\ P)\ Ts\ Ts'$

lemma [*iff*]: $(P \vdash T \leq Void) = (T = Void)$
lemma [*iff*]: $(P \vdash T \leq Boolean) = (T = Boolean)$
lemma [*iff*]: $(P \vdash T \leq Integer) = (T = Integer)$
lemma [*iff*]: $(P \vdash Void \leq T) = (T = Void)$
lemma [*iff*]: $(P \vdash Boolean \leq T) = (T = Boolean)$
lemma [*iff*]: $(P \vdash Integer \leq T) = (T = Integer)$
lemma *Class-widen*: $P \vdash Class\ C \leq T \Longrightarrow \exists D.\ T = Class\ D$
by(*erule widen.cases*, *auto*)

lemma *Array-Array-widen*:
  $P \vdash Array\ T \leq Array\ U \Longrightarrow P \vdash T \leq U$
by(*auto elim*: *widen.cases*)

lemma *widen-Array*: $(P \vdash T \leq U\lfloor\rceil) \longleftrightarrow (T = NT \vee (\exists V.\ T = V\lfloor\rceil \wedge P \vdash V \leq U))$
by(*induct T*)(*auto dest*: *Array-Array-widen elim*: *widen.cases intro*: *widen-array-array*)

lemma *Array-widen*: $P \vdash Array\ A \leq T \Longrightarrow (\exists B.\ T = Array\ B \wedge P \vdash A \leq B) \vee T = Class\ Object$
by(*auto elim*: *widen.cases*)

lemma [*iff*]: $(P \vdash T \leq NT) = (T = NT)$
by(*induct T*)(*auto dest*:*Class-widen Array-widen*)

lemma *Class-widen-Class* [*iff*]: $(P \vdash Class\ C \leq Class\ D) = (P \vdash C \preceq^* D)$
by (*auto elim*: *widen-subcls widen.cases*)

lemma *widen-Class*: $(P \vdash T \leq Class\ C) = (T = NT \vee (\exists D.\ T = Class\ D \wedge P \vdash D \preceq^* C) \vee (C = Object \wedge (\exists A.\ T = Array\ A)))$
by(*induct T*)(*auto dest*: *Array-widen intro*: *widen-array-object*)

lemma *NT-widen*:
  $P \vdash NT \leq T = (T = NT \vee (\exists C.\ T = Class\ C) \vee (\exists U.\ T = U\lfloor\rceil))$
by(*cases T*) *auto*

lemma *Class-widen2*: $P \vdash Class\ C \leq T = (\exists D.\ T = Class\ D \wedge P \vdash C \preceq^* D)$
by (*cases T*, *auto elim*: *widen.cases*)

lemma *Object-widen*: $P \vdash Class\ Object \leq T \Longrightarrow T = Class\ Object$
by(*cases T*, *auto elim*: *widen.cases*)

lemma *NT-Array-widen-Object*:
  is-NT-Array T ⟹ $P \vdash T \leq Class\ Object$

by(*induct T*, *auto intro*: *widen-array-object*)

lemma *widen-trans*[*trans*]:
  assumes $P \vdash S \leq U$ $P \vdash U \leq T$
  shows $P \vdash S \leq T$
using *assms*
proof(*induct arbitrary*: *T*)
  case (*widen-refl T T'*) **thus** $P \vdash T \leq T'$ **.**
next
  case (*widen-subcls C D T*)
  then obtain *E* **where** $T = Class\ E$ **by** (*blast dest*: *Class-widen*)
  with *widen-subcls* **show** $P \vdash Class\ C \leq T$ **by** (*auto elim*: *rtrancl-trans*)
next
  case (*widen-null C RT*)
  then obtain *D* **where** $RT = Class\ D$ **by** (*blast dest*: *Class-widen*)
  thus $P \vdash NT \leq RT$ **by** *auto*
next
  case *widen-null-array* **thus** *?case* **by**(*auto dest*: *Array-widen*)
next
  case (*widen-array-object A T*)
  hence $T = Class\ Object$ **by**(*rule Object-widen*)
  with *widen-array-object* **show** $P \vdash A\lfloor\rceil \leq T$
    by(*auto intro*: *widen.widen-array-object*)
next
  case *widen-array-array* **thus** *?case*
    by(*auto dest*!: *Array-widen intro*: *widen.widen-array-array widen-array-object*)
qed

lemma *widens-trans*: $[\![P \vdash Ss\ [\leq]\ Ts;\ P \vdash Ts\ [\leq]\ Us]\!] \Longrightarrow P \vdash Ss\ [\leq]\ Us$
by (*rule list-all2-trans*)(*rule widen-trans*)

lemma *class-type-of'-widenD*:
  $class\text{-}type\text{-}of'\ T = \lfloor C \rfloor \Longrightarrow P \vdash T \leq Class\ C$
by(*cases T*)(*auto intro*: *widen-array-object*)

lemma *widen-is-class-type-of*:
  assumes $class\text{-}type\text{-}of'\ T = \lfloor C \rfloor$ $P \vdash T' \leq T$ $T' \neq NT$
  obtains C' **where** $class\text{-}type\text{-}of'\ T' = \lfloor C' \rfloor$ $P \vdash C' \preceq^* C$
using *assms* **by**(*cases T*)(*auto simp add*: *widen-Class widen-Array*)

lemma *widens-refl*: $P \vdash Ts\ [\leq]\ Ts$
by(*rule list-all2-refl*[*OF widen-refl*])

lemma *widen-append1*:
  $P \vdash (xs\ @\ ys)\ [\leq]\ Ts = (\exists\ Ts1\ Ts2.\ Ts = Ts1\ @\ Ts2 \wedge length\ xs = length\ Ts1 \wedge length\ ys = length\ Ts2 \wedge P \vdash xs\ [\leq]\ Ts1 \wedge P \vdash ys\ [\leq]\ Ts2)$
unfolding *list-all2-append1* **by** *fastforce*

lemmas *widens-Cons* [*iff*] = *list-all2-Cons1* [*of widen P*] **for** *P*

lemma *widens-lengthD*:
  $P \vdash xs\ [\leq]\ ys \Longrightarrow length\ xs = length\ ys$
by(*rule list-all2-lengthD*)

lemma *widen-refT*: ⟦ *is-refT T*; *P* ⊢ *U* ≤ *T* ⟧ ⟹ *is-refT U*
by(*erule refTE*)(*auto simp add*: *widen-Class widen-Array*)

lemma *refT-widen*: ⟦ *is-refT T*; *P* ⊢ *T* ≤ *U* ⟧ ⟹ *is-refT U*
by(*erule widen.cases*) *auto*

inductive *is-lub* :: *'m prog* ⇒ *ty* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹- ⊢ *lub'*((-,/ -)') = -› [*51*,*51*,*51*,*51*] *50*)
for *P* :: *'m prog* **and** *U* :: *ty* **and** *V* :: *ty* **and** *T* :: *ty*
where
⟦ *P* ⊢ *U* ≤ *T*; *P* ⊢ *V* ≤ *T*;
⋀*T'*. ⟦ *P* ⊢ *U* ≤ *T'*; *P* ⊢ *V* ≤ *T'* ⟧ ⟹ *P* ⊢ *T* ≤ *T'* ⟧
⟹ *P* ⊢ *lub*(*U*, *V*) = *T*

lemma *is-lub-upper*:
P ⊢ *lub*(*U*, *V*) = *T* ⟹ *P* ⊢ *U* ≤ *T* ∧ *P* ⊢ *V* ≤ *T*
by(*auto elim*: *is-lub.cases*)

lemma *is-lub-least*:
⟦ *P* ⊢ *lub*(*U*, *V*) = *T*; *P* ⊢ *U* ≤ *T'*; *P* ⊢ *V* ≤ *T'* ⟧ ⟹ *P* ⊢ *T* ≤ *T'*
by(*auto elim*: *is-lub.cases*)

lemma *is-lub-Void* [*iff*]:
P ⊢ *lub*(*Void*, *Void*) = *T* ⟷ *T* = *Void*
by(*auto intro*: *is-lub.intros elim*: *is-lub.cases*)

lemma *is-lubI* [*code-pred-intro*]:
⟦*P* ⊢ *U* ≤ *T*; *P* ⊢ *V* ≤ *T*; ∀ *T'*. *P* ⊢ *U* ≤ *T'* ⟶ *P* ⊢ *V* ≤ *T'* ⟶ *P* ⊢ *T* ≤ *T'*⟧ ⟹ *P* ⊢ *lub*(*U*, *V*) = *T*
by(*blast intro*: *is-lub.intros*)

3.3.3 Method lookup

inductive *Methods* :: *'m prog* ⇒ *cname* ⇒ (*mname* ⇀ (*ty list* × *ty* × *'m option*) × *cname*) ⇒ *bool*
(‹- ⊢ - *sees'-methods* -› [*51*,*51*,*51*] *50*)
for *P* :: *'m prog*
where
sees-methods-Object:
⟦ *class P Object* = *Some*(*D*,*fs*,*ms*); *Mm* = *map-option* (λ*m*. (*m*,*Object*)) ∘ *map-of ms* ⟧
⟹ *P* ⊢ *Object sees-methods Mm*
| *sees-methods-rec*:
⟦ *class P C* = *Some*(*D*,*fs*,*ms*); *C* ≠ *Object*; *P* ⊢ *D sees-methods Mm*;
Mm' = *Mm* ++ (*map-option* (λ*m*. (*m*,*C*)) ∘ *map-of ms*) ⟧
⟹ *P* ⊢ *C sees-methods Mm'*

lemma *sees-methods-fun*:
assumes *P* ⊢ *C sees-methods Mm*
shows *P* ⊢ *C sees-methods Mm'* ⟹ *Mm'* = *Mm*
using *assms*
proof(*induction arbitrary*: *Mm'*)
case *sees-methods-Object* **thus** *?case* **by**(*auto elim*: *Methods.cases*)
next
case (*sees-methods-rec C D fs ms Dres Cres Cres'*)
from ‹*P* ⊢ *C sees-methods Cres'*› ‹*C* ≠ *Object*› ‹*class P C* = ⌊(*D*, *fs*, *ms*)⌋›
obtain *Dres'* **where** *Dmethods'*: *P* ⊢ *D sees-methods Dres'*

and *Cres'*: *Cres'* = *Dres'* ++ (*map-option* (λm. (m,C)) ∘ *map-of ms*)
by *cases auto*
from *sees-methods-rec.IH*[*OF Dmethods'*] ‹*Cres* = *Dres* ++ (*map-option* (λm. (m,C)) ∘ *map-of ms*)› *Cres'*
show *?case* **by** *simp*
qed

lemma *visible-methods-exist*:
P ⊢ *C sees-methods Mm* ⟹ *Mm M* = *Some*(m,D) ⟹
(∃ *D' fs ms*. *class P D* = *Some*(*D'*,*fs*,*ms*) ∧ *map-of ms M* = *Some m*)
by(*induct rule*:*Methods.induct*) *auto*

lemma *sees-methods-decl-above*:
assumes *P* ⊢ *C sees-methods Mm*
shows *Mm M* = *Some*(m,D) ⟹ *P* ⊢ *C* ≼* *D*
using *assms*
by *induct*(*auto elim*: *converse-rtranclp-into-rtranclp*[**where** *r* = *subcls1 P*, *OF subcls1I*])

lemma *sees-methods-idemp*:
assumes *P* ⊢ *C sees-methods Mm* **and** *Mm M* = *Some*(m,D)
shows ∃ *Mm'*. (*P* ⊢ *D sees-methods Mm'*) ∧ *Mm' M* = *Some*(m,D)
using *assms*
by(*induct arbitrary*: *m D*)(*fastforce dest*: *Methods.intros*)+

lemma *sees-methods-decl-mono*:
assumes *sub*: *P* ⊢ *C'* ≼* *C* **and** *P* ⊢ *C sees-methods Mm*
shows ∃ *Mm' Mm*$_2$. *P* ⊢ *C' sees-methods Mm'* ∧ *Mm'* = *Mm* ++ *Mm*$_2$ ∧ (∀ *M m D*. *Mm*$_2$ *M* = *Some*(m,D) ⟶ *P* ⊢ *D* ≼* *C*)
(**is** ∃ *Mm' Mm2*. *?Q C' C Mm' Mm2*)
using *assms*
proof (*induction rule*: *converse-rtranclp-induct*)
case *base*
hence *?Q C C Mm Map.empty* **by** *simp*
thus ∃ *Mm' Mm2*. *?Q C C Mm' Mm2* **by** *blast*
next
case (*step C'' C'*)
note *sub1* = ‹*P* ⊢ *C''* ≺^{1} *C'*› **and** *sub* = ‹*P* ⊢ *C'* ≼* *C*›
and *Csees* = ‹*P* ⊢ *C sees-methods Mm*›
from *step.IH*[*OF Csees*] **obtain** *Mm' Mm2* **where** *C'sees*: *P* ⊢ *C' sees-methods Mm'*
and *Mm'*: *Mm'* = *Mm* ++ *Mm2*
and *subC*: ∀ *M m D*. *Mm2 M* = *Some*(m,D) ⟶ *P* ⊢ *D* ≼* *C* **by** *blast*
obtain *fs ms* **where** *class*: *class P C''* = *Some*(*C'*,*fs*,*ms*) *C''* ≠ *Object*
using *subcls1D*[*OF sub1*] **by** *blast*
let *?Mm3* = *map-option* (λm. (m,*C''*)) ∘ *map-of ms*
have *P* ⊢ *C''* *sees-methods* (*Mm* ++ *Mm2*) ++ *?Mm3*
using *sees-methods-rec*[*OF class C'sees refl*] *Mm'* **by** *simp*
hence *?Q C''* *C* ((*Mm* ++ *Mm2*) ++ *?Mm3*) (*Mm2*++*?Mm3*)
using *converse-rtranclp-into-rtranclp*[*OF sub1 sub*]
by *simp* (*simp add*:*map-add-def subC split*:*option.split*)
thus ∃ *Mm' Mm2*. *?Q C''* *C Mm' Mm2* **by** *blast*
qed

definition *Method* :: *'m prog* ⇒ *cname* ⇒ *mname* ⇒ *ty list* ⇒ *ty* ⇒ *'m option* ⇒ *cname* ⇒ *bool*
(‹- ⊢ - *sees* -: -→- = - *in* -› [*51*,*51*,*51*,*51*,*51*,*51*,*51*] *50*)

where
$P \vdash C$ *sees* M: $Ts \to T = m$ *in* $D \equiv$
$\exists Mm.\ P \vdash C$ *sees-methods* $Mm \wedge Mm\ M = Some((Ts,T,m),D)$

Output translation to replace *None* with its notation *Native* when used as method body in *Method*.

abbreviation (**output**)
Method-native :: *'m prog* ⇒ *cname* ⇒ *mname* ⇒ *ty list* ⇒ *ty* ⇒ *cname* ⇒ *bool*
(‹- ⊢ - *sees* -: -→- = *Native in* -› [*51*,*51*,*51*,*51*,*51*,*51*] *50*)
where *Method-native P C M Ts T D* ≡ *Method P C M Ts T Native D*

definition *has-method* :: *'m prog* ⇒ *cname* ⇒ *mname* ⇒ *bool* (‹- ⊢ - *has* -› [*51*,*0*,*51*] *50*)
where
$P \vdash C$ *has* $M \equiv \exists\ Ts\ T\ m\ D.\ P \vdash C$ *sees* M:$Ts \to T = m$ *in* D

lemma *has-methodI*:
$P \vdash C$ *sees* M:$Ts \to T = m$ *in* $D \Longrightarrow P \vdash C$ *has* M
by (*unfold has-method-def*) *blast*

lemma *sees-method-fun*:
$[\![P \vdash C$ *sees* M:$TS \to T = m$ *in* D; $P \vdash C$ *sees* M:$TS' \to T' = m'$ *in* $D']\!]$
$\Longrightarrow TS' = TS \wedge T' = T \wedge m' = m \wedge D' = D$

lemma *sees-method-decl-above*:
$P \vdash C$ *sees* M:$Ts \to T = m$ *in* $D \Longrightarrow P \vdash C \preceq^* D$

lemma *visible-method-exists*:
$P \vdash C$ *sees* M:$Ts \to T = m$ *in* $D \Longrightarrow$
$\exists D'\ fs\ ms.$ *class* $P\ D = Some(D',fs,ms) \wedge$ *map-of* $ms\ M = Some(Ts,T,m)$

lemma *sees-method-idemp*:
$P \vdash C$ *sees* M:$Ts \to T{=}m$ *in* $D \Longrightarrow P \vdash D$ *sees* M:$Ts \to T{=}m$ *in* D

lemma *sees-method-decl-mono*:
$[\![\ P \vdash C' \preceq^* C$; $P \vdash C$ *sees* M:$Ts \to T = m$ *in* D;
$P \vdash C'$ *sees* M:$Ts' \to T' = m'$ *in* $D'\ ]\!] \Longrightarrow P \vdash D' \preceq^* D$
apply(*frule sees-method-decl-above*)
apply(*unfold Method-def*)
apply *clarsimp*
apply(*drule* (*1*) *sees-methods-decl-mono*)
apply *clarsimp*
apply(*drule* (*1*) *sees-methods-fun*)
apply *clarsimp*
apply(*blast intro*:*rtranclp-trans*)
done

lemma *sees-method-is-class*:
$P \vdash C$ *sees* M:$Ts \to T = m$ *in* $D \Longrightarrow$ *is-class* $P\ C$
by (*auto simp add*: *is-class-def Method-def elim*: *Methods.cases*)

3.3.4 Field lookup

inductive *Fields* :: *'m prog* ⇒ *cname* ⇒ ((*vname* × *cname*) × (*ty* × *fmod*)) *list* ⇒ *bool*
(‹- ⊢ - *has'-fields* -› [*51*,*51*,*51*] *50*)

```
  for P :: 'm prog
where
  has-fields-rec:
  ⟦ class P C = Some(D,fs,ms); C ≠ Object; P ⊢ D has-fields FDTs;
     FDTs' = map (λ(F,Tm). ((F,C),Tm)) fs @ FDTs ⟧
   ⟹ P ⊢ C has-fields FDTs'

| has-fields-Object:
  ⟦ class P Object = Some(D,fs,ms); FDTs = map (λ(F,T). ((F,Object),T)) fs ⟧
   ⟹ P ⊢ Object has-fields FDTs

lemma has-fields-fun:
  assumes P ⊢ C has-fields FDTs and P ⊢ C has-fields FDTs'
  shows FDTs' = FDTs
using assms
proof(induction arbitrary: FDTs')
  case has-fields-Object thus ?case by(auto elim: Fields.cases)
next
  case (has-fields-rec C D fs ms Dres Cres Cres')
  from ‹P ⊢ C has-fields Cres'› ‹C ≠ Object› ‹class P C = Some (D, fs, ms)›
  obtain Dres' where DFields': P ⊢ D has-fields Dres'
    and Cres': Cres' = map (λ(F,Tm). ((F,C),Tm)) fs @ Dres'
    by cases auto
  from has-fields-rec.IH[OF DFields'] ‹Cres = map (λ(F,Tm). ((F,C),Tm)) fs @ Dres› Cres'
  show ?case by simp
qed

lemma all-fields-in-has-fields:
  assumes P ⊢ C has-fields FDTs
  and P ⊢ C ≼* D class P D = Some(D',fs,ms) (F,Tm) ∈ set fs
  shows ((F,D),Tm) ∈ set FDTs
using assms
by induct (auto 4 3 elim: converse-rtranclpE dest: subcls1D)

lemma has-fields-decl-above:
  assumes P ⊢ C has-fields FDTs ((F,D),Tm) ∈ set FDTs
  shows P ⊢ C ≼* D
using assms
by induct (auto intro: converse-rtranclp-into-rtranclp subcls1I)

lemma subcls-notin-has-fields:
  assumes P ⊢ C has-fields FDTs ((F,D),Tm) ∈ set FDTs
  shows ¬ (subcls1 P)^++ D C
using assms apply(induct)
 prefer 2 apply(fastforce dest: tranclpD)
apply clarsimp
apply(erule disjE)
 apply(clarsimp simp add:image-def)
 apply(drule tranclpD)
 apply clarify
 apply(frule subcls1D)
 apply(fastforce dest:tranclpD all-fields-in-has-fields)
apply(blast dest:subcls1I tranclp.trancl-into-trancl)
done
```

lemma *has-fields-mono-lem*:
  assumes $P \vdash D \preceq^* C$ $P \vdash C$ *has-fields FDTs*
  shows $\exists$ *pre*. $P \vdash D$ *has-fields pre@FDTs* $\wedge$ *dom*(*map-of pre*) $\cap$ *dom*(*map-of FDTs*) = {}
using *assms*
apply(*induct rule*:*converse-rtranclp-induct*)
 apply(*rule-tac x* = [] **in** *exI*)
 apply *simp*
apply *clarsimp*
apply(*rename-tac D*′ *D pre*)
apply(*subgoal-tac* (*subcls1 P*)^++ *D*′ *C*)
 prefer *2* **apply**(*erule* (*1*) *rtranclp-into-tranclp2*)
apply(*drule subcls1D*)
apply *clarsimp*
apply(*rename-tac fs ms*)
apply(*drule* (*2*) *has-fields-rec*)
 apply(*rule refl*)
apply(*rule-tac x* = *map* (λ(*F*,*Tm*). ((*F*,*D*′),*Tm*)) *fs* @ *pre* **in** *exI*)
apply *simp*
apply(*simp add*:*Int-Un-distrib2*)
apply(*rule equals0I*)
apply(*auto dest*: *subcls-notin-has-fields simp*:*dom-map-of-conv-image-fst image-def*)
done

lemma *has-fields-is-class*:
  $P \vdash C$ *has-fields FDTs* $\Longrightarrow$ *is-class P C*
by (*auto simp add*: *is-class-def elim*: *Fields.cases*)

lemma *Object-has-fields-Object*:
  assumes $P \vdash$ *Object has-fields FDTs*
  shows *snd* ‘ *fst* ‘ *set FDTs* $\subseteq$ {*Object*}
using *assms* **by** *cases auto*

definition
  has-field :: ′*m prog* ⇒ *cname* ⇒ *vname* ⇒ *ty* ⇒ *fmod* ⇒ *cname* ⇒ *bool*
                (‹- ⊢ - *has* -:- ′(-′) *in* -› [*51*,*51*,*51*,*51*,*51*,*51] *50*)
where
  $P \vdash C$ *has F*:*T* (*fm*) *in D* ≡
  $\exists$ *FDTs*. $P \vdash C$ *has-fields FDTs* $\wedge$ *map-of FDTs* (*F*,*D*) = *Some* (*T*, *fm*)

lemma *has-field-mono*:
  ⟦ $P \vdash C$ *has F*:*T* (*fm*) *in D*; $P \vdash C' \preceq^* C$ ⟧ $\Longrightarrow$ $P \vdash C'$ *has F*:*T* (*fm*) *in D*
by(*fastforce simp*:*has-field-def map-add-def dest*: *has-fields-mono-lem*)

lemma *has-field-is-class*:
  $P \vdash C$ *has M*:*T* (*fm*) *in D* $\Longrightarrow$ *is-class P C*
by (*auto simp add*: *is-class-def has-field-def elim*: *Fields.cases*)

lemma *has-field-decl-above*:
  $P \vdash C$ *has F*:*T* (*fm*) *in D* $\Longrightarrow$ $P \vdash C \preceq^* D$
unfolding *has-field-def*
by(*auto dest*: *map-of-SomeD has-fields-decl-above*)

lemma *has-field-fun*:

$[\![P \vdash C \text{ has } F{:}T\ (fm)\ \text{in } D;\ P \vdash C \text{ has } F{:}T'\ (fm')\ \text{in } D]\!] \Longrightarrow T' = T \wedge fm = fm'$
by(*auto simp*:*has-field-def dest*:*has-fields-fun*)

definition
$sees\text{-}field :: 'm\ prog \Rightarrow cname \Rightarrow vname \Rightarrow ty \Rightarrow fmod \Rightarrow cname \Rightarrow bool$
$(\langle - \vdash - \text{ sees } -{:}- '(-') \text{ in } - \rangle\ [51,51,51,51,51,51]\ 50)$
where
$P \vdash C \text{ sees } F{:}T\ (fm) \text{ in } D \equiv$
$\exists FDTs.\ P \vdash C \text{ has-fields } FDTs \wedge$
$map\text{-}of\ (map\ (\lambda((F,D),Tm).\ (F,(D,Tm)))\ FDTs)\ F = Some(D,T,fm)$

lemma *map-of-remap-SomeD*:
$map\text{-}of\ (map\ (\lambda((k,k'),x).\ (k,(k',x)))\ t)\ k = Some\ (k',x) \Longrightarrow map\text{-}of\ t\ (k,\ k') = Some\ x$
by (*induct t*) (*auto simp*:*fun-upd-apply split*: *if-split-asm*)

lemma *has-visible-field*:
$P \vdash C \text{ sees } F{:}T\ (fm) \text{ in } D \Longrightarrow P \vdash C \text{ has } F{:}T\ (fm) \text{ in } D$
by(*auto simp add*:*has-field-def sees-field-def map-of-remap-SomeD*)

lemma *sees-field-fun*:
$[\![P \vdash C \text{ sees } F{:}T\ (fm) \text{ in } D;\ P \vdash C \text{ sees } F{:}T'\ (fm') \text{ in } D']\!] \Longrightarrow T' = T \wedge D' = D \wedge fm = fm'$
by(*fastforce simp*:*sees-field-def dest*:*has-fields-fun*)

lemma *sees-field-decl-above*:
$P \vdash C \text{ sees } F{:}T\ (fm) \text{ in } D \Longrightarrow P \vdash C \preceq^* D$
by(*clarsimp simp add*: *sees-field-def*)
(*blast intro*: *has-fields-decl-above map-of-SomeD map-of-remap-SomeD*)

lemma *sees-field-idemp*:
assumes $P \vdash C \text{ sees } F{:}T\ (fm) \text{ in } D$
shows $P \vdash D \text{ sees } F{:}T\ (fm) \text{ in } D$
proof −
from *assms* **obtain** *FDTs* **where** *has*: $P \vdash C \text{ has-fields } FDTs$
and *F*: $map\text{-}of\ (map\ (\lambda((F,\ D),\ Tm).\ (F,\ D,\ Tm))\ FDTs)\ F = \lfloor(D,\ T,\ fm)\rfloor$
unfolding *sees-field-def* **by** *blast*
thus *?thesis*
proof *induct*
case *has-fields-rec* **thus** *?case* **unfolding** *sees-field-def*
by(*auto*)(*fastforce dest*: *map-of-SomeD intro*!: *exI intro*: *Fields.has-fields-rec*)
next
case *has-fields-Object* **thus** *?case* **unfolding** *sees-field-def*
by(*fastforce dest*: *map-of-SomeD intro*: *Fields.has-fields-Object intro*!: *exI*)
qed
qed

3.3.5 Functional lookup

definition $method :: 'm\ prog \Rightarrow cname \Rightarrow mname \Rightarrow cname \times ty\ list \times ty \times 'm\ option$
where $method\ P\ C\ M \equiv THE\ (D,Ts,T,m).\ P \vdash C \text{ sees } M{:}Ts \to T = m \text{ in } D$

definition $field :: 'm\ prog \Rightarrow cname \Rightarrow vname \Rightarrow cname \times ty \times fmod$
where $field\ P\ C\ F \equiv THE\ (D,T,fm).\ P \vdash C \text{ sees } F{:}T\ (fm) \text{ in } D$

definition *fields* :: *'m prog* ⇒ *cname* ⇒ ((*vname* × *cname*) × (*ty* × *fmod*)) *list*
where *fields P C* ≡ *THE FDTs. P* ⊢ *C has-fields FDTs*

lemma [*simp*]: *P* ⊢ *C has-fields FDTs* ⟹ *fields P C* = *FDTs*
lemma *field-def2* [*simp*]: *P* ⊢ *C sees F*:*T* (*fm*) *in D* ⟹ *field P C F* = (*D*,*T*,*fm*)
lemma *method-def2* [*simp*]: *P* ⊢ *C sees M*: *Ts*→*T* = *m in D* ⟹ *method P C M* = (*D*,*Ts*,*T*,*m*)
lemma *has-fields-b-fields*:
P ⊢ *C has-fields FDTs* ⟹ *fields P C* = *FDTs*
unfolding *fields-def*
by (*blast intro*: *the-equality has-fields-fun*)

lemma *has-field-map-of-fields* [*simp*]:
P ⊢ *C has F*:*T* (*fm*) *in D* ⟹ *map-of* (*fields P C*) (*F*, *D*) = ⌊(*T*, *fm*)⌋
by(*auto simp add*: *has-field-def*)

3.3.6 Code generation

New introduction rules for subcls1

code-pred
— Disallow mode *i-o-o* to force *code-pred* in subsequent predicates not to use this inefficient mode
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *bool*, *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
subcls1
.

Introduce proper constant *subcls'* for *subcls* and generate executable equation for *subcls'*

definition *subcls'* **where** *subcls'* = *subcls*

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *bool*, *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
[*inductify*]
subcls'
.

lemma *subcls-conv-subcls'* [*code-unfold*]:
(*subcls1 P*)^**** = *subcls' P*
by(*simp add*: *subcls'-def*)

Change rule *?P* ⊢ *?A*⌊⌋ ≤ *Class Object* such that predicate compiler tests on class *Object* first. Otherwise *widen-i-o-i* never terminates.

lemma *widen-array-object-code*:
C = *Object* ⟹ *P* ⊢ *Array A* ≤ *Class C*
by(*auto intro*: *widen.intros*)

lemmas [*code-pred-intro*] =
widen-refl widen-subcls widen-null widen-null-array widen-array-object-code widen-array-array
code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *bool*)
widen
by(*erule widen.cases*) *auto*

Readjust the code equations for *widen* such that *widen-i-i-i* is guaranteed to contain () at most once (even in the code representation!). This is important for the scheduler and the small-step semantics because of the weaker code equations for *the*.

A similar problem cannot hit the subclass relation because, for acyclic subclass hierarchies, the paths in the hieararchy are unique and cycle-free.

definition *widen-i-i-i′* **where** *widen-i-i-i′ = widen-i-i-i*

declare *widen.equation* [*code del*]
lemmas *widen-i-i-i′-equation* [*code*] = *widen.equation*[*folded widen-i-i-i′-def*]

lemma *widen-i-i-i-code* [*code*]:
widen-i-i-i P T T′ = (if P ⊢ T ≤ T′ then Predicate.single () else bot)
by(*auto intro*!: *pred-eqI intro*: *widen-i-i-iI elim*: *widen-i-i-iE*)

code-pred
(*modes*: *i ⇒ i ⇒ i ⇒ bool*, *i ⇒ i ⇒ o ⇒ bool*)
Methods
.

code-pred
(*modes*: *i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ o ⇒ o ⇒ bool*, *i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ o ⇒ i ⇒ bool*)
[*inductify*]
Method
.

code-pred
(*modes*: *i ⇒ i ⇒ i ⇒ bool*)
[*inductify*]
has-method
.

declare *fun-upd-def* [*code-pred-inline*]

code-pred
(*modes*: *i ⇒ i ⇒ o ⇒ bool*)
Fields
.

code-pred
(*modes*: *i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ i ⇒ bool*)
[*inductify*, *skip-proof*]
has-field
.

code-pred
(*modes*: *i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ o ⇒ bool*, *i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ i ⇒ bool*)
[*inductify*, *skip-proof*]
sees-field
.

lemma *eval-Method-i-i-i-o-o-o-o-conv*:
Predicate.eval (*Method-i-i-i-o-o-o-o P C M*) = (*λ*(*Ts*, *T*, *m*, *D*). *P ⊢ C sees M*:*Ts→T*=*m in D*)
by(*auto intro*: *Method-i-i-i-o-o-o-oI elim*: *Method-i-i-i-o-o-o-oE intro*!: *ext*)

lemma *method-code* [*code*]:
method P C M =

Predicate.the (*Predicate.bind* (*Method-i-i-i-o-o-o-o P C M*) (λ(*Ts*, *T*, *m*, *D*). *Predicate.single* (*D*, *Ts*, *T*, *m*)))
apply (*rule sym*, *rule the-eqI*)
apply (*simp add*: *method-def eval-Method-i-i-i-o-o-o-o-conv*)
apply (*rule arg-cong* [**where** *f*=*The*])
apply (*auto simp add*: *Sup-fun-def Sup-bool-def fun-eq-iff*)
done

lemma *eval-sees-field-i-i-i-o-o-o-conv*:
Predicate.eval (*sees-field-i-i-i-o-o-o P C F*) = (λ(*T*, *fm*, *D*). *P* ⊢ *C sees F*:*T* (*fm*) *in D*)
by(*auto intro*!: *ext intro*: *sees-field-i-i-i-o-o-oI elim*: *sees-field-i-i-i-o-o-oE*)

lemma *eval-sees-field-i-i-i-o-i-conv*:
Predicate.eval (*sees-field-i-i-i-o-o-i P C F D*) = (λ(*T*, *fm*). *P* ⊢ *C sees F*:*T* (*fm*) *in D*)
by(*auto intro*!: *ext intro*: *sees-field-i-i-i-o-o-iI elim*: *sees-field-i-i-i-o-o-iE*)

lemma *field-code* [*code*]:
field P C F = *Predicate.the* (*Predicate.bind* (*sees-field-i-i-i-o-o-o P C F*) (λ(*T*, *fm*, *D*). *Predicate.single* (*D*, *T*, *fm*)))
apply (*rule sym*, *rule the-eqI*)
apply (*simp add*: *field-def eval-sees-field-i-i-i-o-o-o-conv*)
apply (*rule arg-cong* [**where** *f*=*The*])
apply (*auto simp add*: *Sup-fun-def Sup-bool-def fun-eq-iff*)
done

lemma *eval-Fields-conv*:
Predicate.eval (*Fields-i-i-o P C*) = (λ*FDTs*. *P* ⊢ *C has-fields FDTs*)
by(*auto intro*: *Fields-i-i-oI elim*: *Fields-i-i-oE intro*!: *ext*)

lemma *fields-code* [*code*]:
fields P C = *Predicate.the* (*Fields-i-i-o P C*)
by(*simp add*: *fields-def Predicate.the-def eval-Fields-conv*)

code-identifier
code-module *TypeRel* ⇀
(*SML*) *TypeRel* **and** (*Haskell*) *TypeRel* **and** (*OCaml*) *TypeRel*
| **code-module** *Decl* ⇀
(*SML*) *TypeRel* **and** (*Haskell*) *TypeRel* **and** (*OCaml*) *TypeRel*

end

3.4 Jinja Values

theory *Value*
imports
TypeRel
HOL−Library.Word
begin

unbundle *no floor-ceiling-syntax*

type-synonym *word32* = *32 word*

datatype *'addr val*
  = *Unit* — dummy result value of void expressions
  | *Null* — null reference
  | *Bool bool* — Boolean value
  | *Intg word32* — integer value
  | *Addr 'addr* — addresses of objects, arrays and threads in the heap

primrec *default-val* :: *ty* ⇒ *'addr val* — default value for all types
where
  default-val Void = *Unit*
| *default-val Boolean* = *Bool False*
| *default-val Integer* = *Intg 0*
| *default-val NT* = *Null*
| *default-val* (*Class C*) = *Null*
| *default-val* (*Array A*) = *Null*

lemma *default-val-not-Addr*: *default-val T* ≠ *Addr a*
by(*cases T*)(*simp-all*)

lemma *Addr-not-default-val*: *Addr a* ≠ *default-val T*
by(*cases T*)(*simp-all*)

primrec *the-Intg* :: *'addr val* ⇒ *word32*
where
  the-Intg (*Intg i*) = *i*

primrec *the-Addr* :: *'addr val* ⇒ *'addr*
where
  the-Addr (*Addr a*) = *a*

fun *is-Addr* :: *'addr val* ⇒ *bool*
where
  is-Addr (*Addr a*) = *True*
| *is-Addr* - = *False*

lemma *is-AddrE* [*elim*!]:
  ⟦ *is-Addr v*; ⋀*a*. *v* = *Addr a* ⟹ *thesis* ⟧ ⟹ *thesis*
by(*cases v*, *auto*)

fun *is-Intg* :: *'addr val* ⇒ *bool*
where
  is-Intg (*Intg i*) = *True*
| *is-Intg* - = *False*

lemma *is-IntgE* [*elim*!]:
  ⟦ *is-Intg v*; ⋀*i*. *v* = *Intg i* ⟹ *thesis* ⟧ ⟹ *thesis*
by(*cases v*, *auto*)

fun *is-Bool* :: *'addr val* ⇒ *bool*
where
  is-Bool (*Bool b*) = *True*
| *is-Bool* - = *False*

lemma *is-BoolE* [*elim*!]:

⟦ *is-Bool v*; ⋀*a*. *v* = *Bool a* ⟹ *thesis* ⟧ ⟹ *thesis*
by(*cases v*, *auto*)

definition *is-Ref* :: *'addr val* ⇒ *bool*
where *is-Ref v* ≡ *v* = *Null* ∨ *is-Addr v*

lemma *is-Ref-def2*:
is-Ref v = (*v* = *Null* ∨ (∃ *a*. *v* = *Addr a*))
by (*cases v*) (*auto simp add*: *is-Ref-def*)

lemma [*iff*]: *is-Ref Null* **by** (*simp add*: *is-Ref-def2*)

definition *undefined-value* :: *'addr val* **where** *undefined-value* = *Unit*

lemma *undefined-value-not-Addr*:
undefined-value ≠ *Addr a Addr a* ≠ *undefined-value*
by(*simp-all add*: *undefined-value-def*)

class *addr* =
fixes *hash-addr* :: *'a* ⇒ *int*
and *monitor-finfun-to-list* :: (*'a* ⇒*f nat*) ⇒ *'a list*
assumes *set* (*monitor-finfun-to-list f*) = *Collect* (($) (*finfun-dom f*))

locale *addr-base* =
fixes *addr2thread-id* :: *'addr* ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*

end

3.5 Exceptions

theory *Exceptions*
imports
Value
begin

definition *NullPointer* :: *cname*
where [*code-unfold*]: *NullPointer* = *STR ''java/lang/NullPointerException''*

definition *ClassCast* :: *cname*
where [*code-unfold*]: *ClassCast* = *STR ''java/lang/ClassCastException''*

definition *OutOfMemory* :: *cname*
where [*code-unfold*]: *OutOfMemory* = *STR ''java/lang/OutOfMemoryError''*

definition *ArrayIndexOutOfBounds* :: *cname*
where [*code-unfold*]: *ArrayIndexOutOfBounds* = *STR ''java/lang/ArrayIndexOutOfBoundsException''*

definition *ArrayStore* :: *cname*
where [*code-unfold*]: *ArrayStore* = *STR ''java/lang/ArrayStoreException''*

definition *NegativeArraySize* :: *cname*
where [*code-unfold*]: *NegativeArraySize* = *STR ''java/lang/NegativeArraySizeException''*

definition *ArithmeticException* :: *cname*
where [*code-unfold*]: *ArithmeticException* = *STR ''java/lang/ArithmeticException''*

definition *IllegalMonitorState* :: *cname*
where [*code-unfold*]: *IllegalMonitorState* = *STR ''java/lang/IllegalMonitorStateException''*

definition *IllegalThreadState* :: *cname*
where [*code-unfold*]: *IllegalThreadState* = *STR ''java/lang/IllegalThreadStateException''*

definition *InterruptedException* :: *cname*
where [*code-unfold*]: *InterruptedException* = *STR ''java/lang/InterruptedException''*

definition *sys-xcpts-list* :: *cname list*
where
sys-xcpts-list =
[*NullPointer*, *ClassCast*, *OutOfMemory*, *ArrayIndexOutOfBounds*, *ArrayStore*, *NegativeArraySize*, *ArithmeticException*,
IllegalMonitorState, *IllegalThreadState*, *InterruptedException*]

definition *sys-xcpts* :: *cname set*
where [*code-unfold*]: *sys-xcpts* = *set sys-xcpts-list*

definition *wf-syscls* :: *'m prog* ⇒ *bool*
where *wf-syscls P* ≡ (∀ *C* ∈ {*Object*, *Throwable*, *Thread*}. *is-class P C*) ∧ (∀ *C* ∈ *sys-xcpts*. *P* ⊢ *C* ≼* *Throwable*)

3.5.1 System exceptions

lemma [*simp*]:
NullPointer ∈ *sys-xcpts* ∧
OutOfMemory ∈ *sys-xcpts* ∧
ClassCast ∈ *sys-xcpts* ∧
ArrayIndexOutOfBounds ∈ *sys-xcpts* ∧
ArrayStore ∈ *sys-xcpts* ∧
NegativeArraySize ∈ *sys-xcpts* ∧
IllegalMonitorState ∈ *sys-xcpts* ∧
IllegalThreadState ∈ *sys-xcpts* ∧
InterruptedException ∈ *sys-xcpts* ∧
ArithmeticException ∈ *sys-xcpts*
by(*simp add*: *sys-xcpts-def sys-xcpts-list-def*)

lemma *sys-xcpts-cases* [*consumes 1*, *cases set*]:
⟦ *C* ∈ *sys-xcpts*; *P NullPointer*; *P OutOfMemory*; *P ClassCast*;
P ArrayIndexOutOfBounds; *P ArrayStore*; *P NegativeArraySize*;
P ArithmeticException;
P IllegalMonitorState; *P IllegalThreadState*; *P InterruptedException* ⟧
⟹ *P C*
by (*auto simp add*: *sys-xcpts-def sys-xcpts-list-def*)

lemma *OutOfMemory-not-Object*[*simp*]: *OutOfMemory* ≠ *Object*
by(*simp add*: *OutOfMemory-def Object-def*)

lemma *ClassCast-not-Object*[*simp*]: *ClassCast* ≠ *Object*

by(*simp add*: *ClassCast-def Object-def*)

lemma *NullPointer-not-Object*[*simp*]: *NullPointer* ≠ *Object*
by(*simp add*: *NullPointer-def Object-def*)

lemma *ArrayIndexOutOfBounds-not-Object*[*simp*]: *ArrayIndexOutOfBounds* ≠ *Object*
by(*simp add*: *ArrayIndexOutOfBounds-def Object-def*)

lemma *ArrayStore-not-Object*[*simp*]: *ArrayStore* ≠ *Object*
by(*simp add*: *ArrayStore-def Object-def*)

lemma *NegativeArraySize-not-Object*[*simp*]: *NegativeArraySize* ≠ *Object*
by(*simp add*: *NegativeArraySize-def Object-def*)

lemma *ArithmeticException-not-Object*[*simp*]: *ArithmeticException* ≠ *Object*
by(*simp add*: *ArithmeticException-def Object-def*)

lemma *IllegalMonitorState-not-Object*[*simp*]: *IllegalMonitorState* ≠ *Object*
by(*simp add*: *IllegalMonitorState-def Object-def*)

lemma *IllegalThreadState-not-Object*[*simp*]: *IllegalThreadState* ≠ *Object*
by(*simp add*: *IllegalThreadState-def Object-def*)

lemma *InterruptedException-not-Object*[*simp*]: *InterruptedException* ≠ *Object*
by(*simp add*: *InterruptedException-def Object-def*)

lemma *sys-xcpts-neqs-aux*:
NullPointer ≠ *ClassCast NullPointer* ≠ *OutOfMemory NullPointer* ≠ *ArrayIndexOutOfBounds*
NullPointer ≠ *ArrayStore NullPointer* ≠ *NegativeArraySize NullPointer* ≠ *IllegalMonitorState*
NullPointer ≠ *IllegalThreadState NullPointer* ≠ *InterruptedException NullPointer* ≠ *ArithmeticException*
ClassCast ≠ *OutOfMemory ClassCast* ≠ *ArrayIndexOutOfBounds*
ClassCast ≠ *ArrayStore ClassCast* ≠ *NegativeArraySize ClassCast* ≠ *IllegalMonitorState*
ClassCast ≠ *IllegalThreadState ClassCast* ≠ *InterruptedException ClassCast* ≠ *ArithmeticException*
OutOfMemory ≠ *ArrayIndexOutOfBounds*
OutOfMemory ≠ *ArrayStore OutOfMemory* ≠ *NegativeArraySize OutOfMemory* ≠ *IllegalMonitorState*
OutOfMemory ≠ *IllegalThreadState OutOfMemory* ≠ *InterruptedException*
OutOfMemory ≠ *ArithmeticException*
ArrayIndexOutOfBounds ≠ *ArrayStore ArrayIndexOutOfBounds* ≠ *NegativeArraySize ArrayIndexOutOfBounds* ≠ *IllegalMonitorState*
ArrayIndexOutOfBounds ≠ *IllegalThreadState ArrayIndexOutOfBounds* ≠ *InterruptedException ArrayIndexOutOfBounds* ≠ *ArithmeticException*
ArrayStore ≠ *NegativeArraySize ArrayStore* ≠ *IllegalMonitorState*
ArrayStore ≠ *IllegalThreadState ArrayStore* ≠ *InterruptedException*
ArrayStore ≠ *ArithmeticException*
NegativeArraySize ≠ *IllegalMonitorState*
NegativeArraySize ≠ *IllegalThreadState NegativeArraySize* ≠ *InterruptedException*
NegativeArraySize ≠ *ArithmeticException*
IllegalMonitorState ≠ *IllegalThreadState IllegalMonitorState* ≠ *InterruptedException*
IllegalMonitorState ≠ *ArithmeticException*
IllegalThreadState ≠ *InterruptedException*
IllegalThreadState ≠ *ArithmeticException*
InterruptedException ≠ *ArithmeticException*

by(*simp-all add*: *NullPointer-def ClassCast-def OutOfMemory-def ArrayIndexOutOfBounds-def ArrayStore-def NegativeArraySize-def IllegalMonitorState-def IllegalThreadState-def InterruptedException-def ArithmeticException-def*)

lemmas *sys-xcpts-neqs* = *sys-xcpts-neqs-aux sys-xcpts-neqs-aux*[*symmetric*]

lemma *Thread-neq-sys-xcpts-aux*:
Thread ≠ *NullPointer*
Thread ≠ *ClassCast*
Thread ≠ *OutOfMemory*
Thread ≠ *ArrayIndexOutOfBounds*
Thread ≠ *ArrayStore*
Thread ≠ *NegativeArraySize*
Thread ≠ *ArithmeticException*
Thread ≠ *IllegalMonitorState*
Thread ≠ *IllegalThreadState*
Thread ≠ *InterruptedException*
by(*simp-all add*: *Thread-def NullPointer-def ClassCast-def OutOfMemory-def ArrayIndexOutOfBounds-def ArrayStore-def NegativeArraySize-def IllegalMonitorState-def IllegalThreadState-def InterruptedException-def ArithmeticException-def*)

lemmas *Thread-neq-sys-xcpts* = *Thread-neq-sys-xcpts-aux Thread-neq-sys-xcpts-aux*[*symmetric*]

3.5.2 Well-formedness for system classes and exceptions

lemma
assumes *wf-syscls P*
shows *wf-syscls-class-Object*: ∃ *C fs ms. class P Object* = *Some* (*C*,*fs*,*ms*)
and *wf-syscls-class-Thread*: ∃ *C fs ms. class P Thread* = *Some* (*C*,*fs*,*ms*)
using *assms*
by(*auto simp*: *map-of-SomeI wf-syscls-def is-class-def*)

lemma [*simp*]:
assumes *wf-syscls P*
shows *wf-syscls-is-class-Object*: *is-class P Object*
and *wf-syscls-is-class-Thread*: *is-class P Thread*
using *assms* **by**(*simp-all add*: *is-class-def wf-syscls-class-Object wf-syscls-class-Thread*)

lemma *wf-syscls-xcpt-subcls-Throwable*:
⟦ *C* ∈ *sys-xcpts*; *wf-syscls P* ⟧ ⟹ *P* ⊢ *C* ≼* *Throwable*
by(*simp add*: *wf-syscls-def is-class-def class-def*)

lemma *wf-syscls-is-class-Throwable*:
wf-syscls P ⟹ *is-class P Throwable*
by(*auto simp add*: *wf-syscls-def is-class-def class-def map-of-SomeI*)

lemma *wf-syscls-is-class-sub-Throwable*:
⟦ *wf-syscls P*; *P* ⊢ *C* ≼* *Throwable* ⟧ ⟹ *is-class P C*
by(*erule subcls-is-class1*)(*erule wf-syscls-is-class-Throwable*)

lemma *wf-syscls-is-class-xcpt*:
⟦ *C* ∈ *sys-xcpts*; *wf-syscls P* ⟧ ⟹ *is-class P C*
by(*blast intro*: *wf-syscls-is-class-sub-Throwable wf-syscls-xcpt-subcls-Throwable*)

lemma *wf-syscls-code* [*code*]:
wf-syscls P $\longleftrightarrow$
($\forall$ *C* $\in$ *set* [*Object*, *Throwable*, *Thread*]. *is-class P C*) $\wedge$ ($\forall$ *C* $\in$ *sys-xcpts*. *P* $\vdash$ *C* $\preceq^*$ *Throwable*)
by(*simp only*: *wf-syscls-def*) *simp*

end

3.6 System Classes

theory *SystemClasses*
imports
Exceptions
begin

This theory provides definitions for the *Object* class, and the system exceptions.

Inline SystemClasses definition because they are polymorphic values that violate ML's value restriction.

Object has actually superclass, but we set it to the empty string for code generation. Any other class name (like *undefined*) would do as well except for code generation.

definition *ObjectC* :: *'m cdecl*
where [*code-unfold*]:
ObjectC =
(*Object*, (*STR ''''*,[],
[(*wait*,[],*Void*,*Native*),
(*notify*,[],*Void*,*Native*),
(*notifyAll*,[],*Void*,*Native*),
(*hashcode*,[],*Integer*,*Native*),
(*clone*,[],*Class Object*,*Native*),
(*print*,[*Integer*],*Void*,*Native*),
(*currentThread*,[],*Class Thread*,*Native*),
(*interrupted*,[],*Boolean*,*Native*),
(*yield*,[],*Void*,*Native*)
]))

definition *ThrowableC* :: *'m cdecl*
where [*code-unfold*]: *ThrowableC* $\equiv$ (*Throwable*, (*Object*, [], []))

definition *NullPointerC* :: *'m cdecl*
where [*code-unfold*]: *NullPointerC* $\equiv$ (*NullPointer*, (*Throwable*,[],[]))

definition *ClassCastC* :: *'m cdecl*
where [*code-unfold*]: *ClassCastC* $\equiv$ (*ClassCast*, (*Throwable*,[],[]))

definition *OutOfMemoryC* :: *'m cdecl*
where [*code-unfold*]: *OutOfMemoryC* $\equiv$ (*OutOfMemory*, (*Throwable*,[],[]))

definition *ArrayIndexOutOfBoundsC* :: *'m cdecl*
where [*code-unfold*]: *ArrayIndexOutOfBoundsC* $\equiv$ (*ArrayIndexOutOfBounds*, (*Throwable*,[],[]))

definition *ArrayStoreC* :: *'m cdecl*
where [*code-unfold*]: *ArrayStoreC* $\equiv$ (*ArrayStore*, (*Throwable*, [], []))

definition *NegativeArraySizeC* :: *'m cdecl*

where [*code-unfold*]: *NegativeArraySizeC* ≡ (*NegativeArraySize*, (*Throwable*,[],[]))

definition *ArithmeticExceptionC* :: *'m cdecl*
where [*code-unfold*]: *ArithmeticExceptionC* ≡ (*ArithmeticException*, (*Throwable*,[],[]))

definition *IllegalMonitorStateC* :: *'m cdecl*
where [*code-unfold*]: *IllegalMonitorStateC* ≡ (*IllegalMonitorState*, (*Throwable*,[],[]))

definition *IllegalThreadStateC* :: *'m cdecl*
where [*code-unfold*]: *IllegalThreadStateC* ≡ (*IllegalThreadState*, (*Throwable*,[],[]))

definition *InterruptedExceptionC* :: *'m cdecl*
where [*code-unfold*]: *InterruptedExceptionC* ≡ (*InterruptedException*, (*Throwable*,[],[]))

definition *SystemClasses* :: *'m cdecl list*
where [*code-unfold*]:
SystemClasses ≡
[*ObjectC*, *ThrowableC*, *NullPointerC*, *ClassCastC*, *OutOfMemoryC*,
ArrayIndexOutOfBoundsC, *ArrayStoreC*, *NegativeArraySizeC*,
ArithmeticExceptionC,
IllegalMonitorStateC, *IllegalThreadStateC*, *InterruptedExceptionC*]

end

3.7 An abstract heap model

theory *Heap*
imports
Value
begin

primrec *typeof* :: *'addr val* ⇀ *ty*
where
typeof Unit = *Some Void*
| *typeof Null* = *Some NT*
| *typeof* (*Bool b*) = *Some Boolean*
| *typeof* (*Intg i*) = *Some Integer*
| *typeof* (*Addr a*) = *None*

datatype *addr-loc* =
CField cname vname
| *ACell nat*

lemma *rec-addr-loc* [*simp*]: *rec-addr-loc* = *case-addr-loc*
by(*auto simp add*: *fun-eq-iff split*: *addr-loc.splits*)

primrec *is-volatile* :: *'m prog* ⇒ *addr-loc* ⇒ *bool*
where
is-volatile P (*ACell n*) = *False*
| *is-volatile P* (*CField D F*) = *volatile* (*snd* (*snd* (*field P D F*)))

locale *heap-base* =
addr-base addr2thread-id thread-id2addr

for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
+
fixes *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
begin

fun *typeof-h* :: *'heap* ⇒ *'addr val* ⇒ *ty option* (‹*typeof*_›)
where
typeof$_h$ (*Addr a*) = *map-option ty-of-htype* (*typeof-addr h a*)
| *typeof*$_h$ *v* = *typeof v*

definition *cname-of* :: *'heap* ⇒ *'addr* ⇒ *cname*
where *cname-of h a* = *the-Class* (*ty-of-htype* (*the* (*typeof-addr h a*)))

definition *hext* :: *'heap* ⇒ *'heap* ⇒ *bool* (‹- ⊴ -› [*51*,*51*] *50*)
where
h ⊴ *h'* ≡ *typeof-addr h* ⊆$_m$ *typeof-addr h'*

context
notes [[*inductive-internals*]]
begin

inductive *addr-loc-type* :: *'m prog* ⇒ *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *ty* ⇒ *bool*
(‹-,- ⊢ -@- : -› [*50*, *50*, *50*, *50*, *50*] *51*)
for *P* :: *'m prog* **and** *h* :: *'heap* **and** *a* :: *'addr*
where
addr-loc-type-field:
⟦ *typeof-addr h a* = ⌊*U*⌋; *P* ⊢ *class-type-of U has F*:*T* (*fm*) *in D* ⟧
⟹ *P*,*h* ⊢ *a*@*CField D F* : *T*

| *addr-loc-type-cell*:
⟦ *typeof-addr h a* = ⌊*Array-type T n'*⌋; *n* < *n'* ⟧
⟹ *P*,*h* ⊢ *a*@*ACell n* : *T*

end

definition *typeof-addr-loc* :: *'m prog* ⇒ *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *ty*
where *typeof-addr-loc P h a al* = (*THE T*. *P*,*h* ⊢ *a*@*al* : *T*)

definition *deterministic-heap-ops* :: *bool*
where
deterministic-heap-ops ⟷
(∀ *h ad al v v'*. *heap-read h ad al v* ⟶ *heap-read h ad al v'* ⟶ *v* = *v'*) ∧
(∀ *h ad al v h' h''*. *heap-write h ad al v h'* ⟶ *heap-write h ad al v h''* ⟶ *h'* = *h''*) ∧
(∀ *h hT h' a h'' a'*. (*h'*, *a*) ∈ *allocate h hT* ⟶ (*h''*, *a'*) ∈ *allocate h hT* ⟶ *h'* = *h''* ∧ *a* = *a'*) ∧
¬ *spurious-wakeups*

end

lemma *typeof-lit-eq-Boolean* [*simp*]: (*typeof v* = *Some Boolean*) = (∃ *b*. *v* = *Bool b*)
by(*cases v*)(*auto*)

lemma *typeof-lit-eq-Integer* [*simp*]: (*typeof v* = *Some Integer*) = (∃ *i*. *v* = *Intg i*)
by(*cases v*)(*auto*)

lemma *typeof-lit-eq-NT* [*simp*]: (*typeof v* = *Some NT*) = (*v* = *Null*)
by(*cases v*)(*auto*)

lemma *typeof-lit-eq-Void* [*simp*]: *typeof v* = *Some Void* ⟷ *v* = *Unit*
by(*cases v*)(*auto*)

lemma *typeof-lit-neq-Class* [*simp*]: *typeof v* ≠ *Some* (*Class C*)
by(*cases v*) *auto*

lemma *typeof-lit-neq-Array* [*simp*]: *typeof v* ≠ *Some* (*Array T*)
by(*cases v*) *auto*

lemma *typeof-NoneD* [*simp*,*dest*]:
typeof v = *Some x* ⟹ ¬ *is-Addr v*
by (*cases v*) *auto*

lemma *typeof-lit-is-type*:
typeof v = *Some T* ⟹ *is-type P T*
by(*cases v*) *auto*

context *heap-base* **begin**

lemma *typeof-h-eq-Boolean* [*simp*]: (*typeof*$_h$ *v* = *Some Boolean*) = (∃ *b*. *v* = *Bool b*)
by(*cases v*)(*auto*)

lemma *typeof-h-eq-Integer* [*simp*]: (*typeof*$_h$ *v* = *Some Integer*) = (∃ *i*. *v* = *Intg i*)
by(*cases v*)(*auto*)

lemma *typeof-h-eq-NT* [*simp*]: (*typeof*$_h$ *v* = *Some NT*) = (*v* = *Null*)
by(*cases v*)(*auto*)

lemma *hextI*:
⟦ ⋀*a C*. *typeof-addr h a* = ⌊*Class-type C*⌋ ⟹ *typeof-addr h*′ *a* = ⌊*Class-type C*⌋;
⋀*a T n*. *typeof-addr h a* = ⌊*Array-type T n*⌋ ⟹ *typeof-addr h*′ *a* = ⌊*Array-type T n*⌋ ⟧
⟹ *h* ⊴ *h*′
unfolding *hext-def*
by(*rule map-leI*)(*case-tac v*, *simp-all*)

lemma *hext-objD*:
assumes *h* ⊴ *h*′
and *typeof-addr h a* = ⌊*Class-type C*⌋
shows *typeof-addr h*′ *a* = ⌊*Class-type C*⌋
using *assms* **unfolding** *hext-def* **by**(*auto dest*: *map-le-SomeD*)

lemma *hext-arrD*:
assumes *h* ⊴ *h*′ *typeof-addr h a* = ⌊*Array-type T n*⌋
shows *typeof-addr h*′ *a* = ⌊*Array-type T n*⌋

using *assms* **unfolding** *hext-def* **by**(*blast dest*: *map-le-SomeD*)

lemma *hext-refl* [*iff*]: $h \trianglelefteq h$
by (*rule hextI*) *blast+*

lemma *hext-trans* [*trans*]: $[\![h \trianglelefteq h'; h' \trianglelefteq h'']\!] \Longrightarrow h \trianglelefteq h''$
unfolding *hext-def* **by**(*rule map-le-trans*)

lemma *typeof-lit-typeof*:
$typeof\ v = \lfloor T \rfloor \Longrightarrow typeof_h\ v = \lfloor T \rfloor$
by(*cases v*)(*simp-all*)

lemma *addr-loc-type-fun*:
$[\![P,h \vdash a@al : T; P,h \vdash a@al : T']\!] \Longrightarrow T = T'$
by(*auto elim*!: *addr-loc-type.cases dest*: *has-field-fun*)

lemma *THE-addr-loc-type*:
$P,h \vdash a@al : T \Longrightarrow (THE\ T.\ P,h \vdash a@al : T) = T$
by(*rule the-equality*)(*auto dest*: *addr-loc-type-fun*)

lemma *typeof-addr-locI* [*simp*]:
$P,h \vdash a@al : T \Longrightarrow typeof\text{-}addr\text{-}loc\ P\ h\ a\ al = T$
by(*auto simp add*: *typeof-addr-loc-def dest*: *addr-loc-type-fun*)

lemma *deterministic-heap-opsI*:
$[\![\bigwedge h\ ad\ al\ v\ v'.\ [\![heap\text{-}read\ h\ ad\ al\ v; heap\text{-}read\ h\ ad\ al\ v']\!] \Longrightarrow v = v';$
$\bigwedge h\ ad\ al\ v\ h'\ h''.\ [\![heap\text{-}write\ h\ ad\ al\ v\ h'; heap\text{-}write\ h\ ad\ al\ v\ h'']\!] \Longrightarrow h' = h'';$
$\bigwedge h\ hT\ h'\ a\ h''\ a'.\ [\![(h', a) \in allocate\ h\ hT; (h'', a') \in allocate\ h\ hT]\!] \Longrightarrow h' = h'' \wedge a = a';$
$\neg\ spurious\text{-}wakeups\ ]\!]$
$\Longrightarrow deterministic\text{-}heap\text{-}ops$
unfolding *deterministic-heap-ops-def* **by** *blast*

lemma *deterministic-heap-ops-readD*:
$[\![deterministic\text{-}heap\text{-}ops; heap\text{-}read\ h\ ad\ al\ v; heap\text{-}read\ h\ ad\ al\ v']\!] \Longrightarrow v = v'$
unfolding *deterministic-heap-ops-def* **by** *blast*

lemma *deterministic-heap-ops-writeD*:
$[\![deterministic\text{-}heap\text{-}ops; heap\text{-}write\ h\ ad\ al\ v\ h'; heap\text{-}write\ h\ ad\ al\ v\ h'']\!] \Longrightarrow h' = h''$
unfolding *deterministic-heap-ops-def* **by** *blast*

lemma *deterministic-heap-ops-allocateD*:
$[\![deterministic\text{-}heap\text{-}ops; (h', a) \in allocate\ h\ hT; (h'', a') \in allocate\ h\ hT]\!] \Longrightarrow h' = h'' \wedge a = a'$
unfolding *deterministic-heap-ops-def* **by** *blast*

lemma *deterministic-heap-ops-no-spurious-wakeups*:
$deterministic\text{-}heap\text{-}ops \Longrightarrow \neg\ spurious\text{-}wakeups$
unfolding *deterministic-heap-ops-def* **by** *blast*

end

locale *addr-conv* =
heap-base
addr2thread-id thread-id2addr
spurious-wakeups

empty-heap allocate typeof-addr heap-read heap-write
+
prog P
for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *P* :: *'m prog*
+
assumes *addr2thread-id-inverse*:
⟦ *typeof-addr h a* = ⌊*Class-type C*⌋; *P* ⊢ *C* ≼* *Thread* ⟧ ⟹ *thread-id2addr* (*addr2thread-id a*) = *a*
begin

lemma *typeof-addr-thread-id2-addr-addr2thread-id* [*simp*]:
⟦ *typeof-addr h a* = ⌊*Class-type C*⌋; *P* ⊢ *C* ≼* *Thread* ⟧ ⟹ *typeof-addr h* (*thread-id2addr* (*addr2thread-id a*)) = ⌊*Class-type C*⌋
by(*simp add*: *addr2thread-id-inverse*)

end

locale *heap* =
addr-conv
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
P
for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *P* :: *'m prog*
+
assumes *allocate-SomeD*: ⟦ (*h'*, *a*) ∈ *allocate h hT*; *is-htype P hT* ⟧ ⟹ *typeof-addr h' a* = *Some hT*

and *hext-allocate*: ⋀*a*. (*h'*, *a*) ∈ *allocate h hT* ⟹ *h* ⊴ *h'*

and *hext-heap-write*:
heap-write h a al v h' ⟹ *h* ⊴ *h'*

begin

lemmas *hext-heap-ops* = *hext-allocate hext-heap-write*

lemma *typeof-addr-hext-mono*:
⟦ *h* ⊴ *h'*; *typeof-addr h a* = ⌊*hT*⌋ ⟧ ⟹ *typeof-addr h' a* = ⌊*hT*⌋

unfolding *hext-def* **by**(*rule map-le-SomeD*)

lemma *hext-typeof-mono*:
⟦ $h \trianglelefteq h'$; $typeof_h\ v = Some\ T$ ⟧ ⟹ $typeof_{h'}\ v = Some\ T$
by (*cases v*)(*auto intro*: *typeof-addr-hext-mono*)

lemma *addr-loc-type-hext-mono*:
⟦ $P,h \vdash a@al : T$; $h \trianglelefteq h'$ ⟧ ⟹ $P,h' \vdash a@al : T$
by(*force elim*!: *addr-loc-type.cases intro*: *addr-loc-type.intros elim*: *typeof-addr-hext-mono dest*: *hext-arrD*)

lemma *type-of-hext-type-of*: — FIXME: What's this rule good for?
⟦ $typeof_h\ w = \lfloor T \rfloor$; *hext h h'* ⟧ ⟹ $typeof_{h'}\ w = \lfloor T \rfloor$
by(*rule hext-typeof-mono*)

lemma *hext-None*: ⟦ $h \trianglelefteq h'$; *typeof-addr h' a* = *None* ⟧ ⟹ *typeof-addr h a* = *None*
by(*rule ccontr*)(*auto dest*: *typeof-addr-hext-mono*)

lemma *map-typeof-hext-mono*:
⟦ $map\ typeof_h\ vs = map\ Some\ Ts$; $h \trianglelefteq h'$ ⟧ ⟹ $map\ typeof_{h'}\ vs = map\ Some\ Ts$
apply(*induct vs arbitrary*: *Ts*)
apply(*auto simp add*: *Cons-eq-map-conv intro*: *hext-typeof-mono*)
done

lemma *hext-typeof-addr-map-le*:
$h \trianglelefteq h'$ ⟹ *typeof-addr h* $\subseteq_m$ *typeof-addr h'*
by(*auto simp add*: *map-le-def dest*: *typeof-addr-hext-mono*)

lemma *hext-dom-typeof-addr-subset*:
$h \trianglelefteq h'$ ⟹ *dom* (*typeof-addr h*) ⊆ *dom* (*typeof-addr h'*)
by (*metis hext-typeof-addr-map-le map-le-implies-dom-le*)

end

declare *heap-base.typeof-h.simps* [*code*]
declare *heap-base.cname-of-def* [*code*]

end

3.8 Observable events in JinjaThreads

theory *Observable-Events*
imports
Heap
../Framework/FWState
begin

datatype (*'addr*,*'thread-id*) *obs-event* =
ExternalCall 'addr mname 'addr val list 'addr val
| *ReadMem 'addr addr-loc 'addr val*
| *WriteMem 'addr addr-loc 'addr val*
| *NewHeapElem 'addr htype*
| *ThreadStart 'thread-id*
| *ThreadJoin 'thread-id*

| *SyncLock 'addr*
| *SyncUnlock 'addr*
| *ObsInterrupt 'thread-id*
| *ObsInterrupted 'thread-id*

instance *obs-event* :: (*type*, *type*) *obs-action*
proof qed

type-synonym
('*addr*, '*thread-id*, '*x*, '*heap*) *Jinja-thread-action* =
('*addr*,'*thread-id*,'*x*,'*heap*,'*addr*,('*addr*, '*thread-id*) *obs-event*) *thread-action*

print-translation ‹
let
fun tr'
[*a1*, *t1*, *x*, *h*, *a2*
, *Const* (@{*type-syntax obs-event*}, -) $ *a3* $ *t2*] =
if a1 = a2 andalso a2 = a3 andalso t1 = t2 then Syntax.const @{*type-syntax Jinja-thread-action*}
$ *a1* $ *t1* $ *x* $ *h*
else raise Match;
in [(@{*type-syntax thread-action*}, *K tr'*)]
end
›
typ ('*addr*, '*thread-id*, '*x*, '*heap*) *Jinja-thread-action*

lemma *range-ty-of-htype*: *range ty-of-htype* ⊆ *range Class* ∪ *range Array*
apply(*rule subsetI*)
apply(*erule rangeE*)
apply(*rename-tac ht*)
apply(*case-tac ht*)
apply *auto*
done

lemma *some-choice*: (∃ *a*. ∀ *b*. *P b* (*a b*)) ⟷ (∀ *b*. ∃ *a*. *P b a*)
by *metis*

definition *convert-RA* :: '*addr released-locks* ⇒ ('*addr* :: *addr*, '*thread-id*) *obs-event list*
where ⋀*ln*. *convert-RA ln* = *concat* (*map* (λ*ad*. *replicate* (*ln* $ *ad*) (*SyncLock ad*)) (*monitor-finfun-to-list ln*))

lemma *set-convert-RA-not-New* [*simp*]:
⋀*ln*. *NewHeapElem a CTn* ∉ *set* (*convert-RA ln*)
by(*auto simp add*: *convert-RA-def*)

lemma *set-convert-RA-not-Read* [*simp*]:
⋀*ln*. *ReadMem ad al v* ∉ *set* (*convert-RA ln*)
by(*auto simp add*: *convert-RA-def*)

end

3.9 The initial configuration

theory *StartConfig*
imports
Exceptions
Observable-Events
begin

definition *initialization-list* :: *cname list*
where
initialization-list = *Thread* # *sys-xcpts-list*

context *heap-base* **begin**

definition *create-initial-object* :: *'heap* × *'addr list* × *bool* ⇒ *cname* ⇒ *'heap* × *'addr list* × *bool*
where
create-initial-object =
(λ(*h*, *ads*, *b*) *C*.
if b
then let HA = *allocate h* (*Class-type C*)
in if HA = {} *then* (*h*, *ads*, *False*)
else let (*h'*, *a''*) = *SOME ha*. *ha* ∈ *HA in* (*h'*, *ads* @ [*a''*], *True*)
else (*h*, *ads*, *False*))

definition *start-heap-data* :: *'heap* × *'addr list* × *bool*
where
start-heap-data = *foldl create-initial-object* (*empty-heap*, [], *True*) *initialization-list*

definition *start-heap* :: *'heap*
where *start-heap* = *fst start-heap-data*

definition *start-heap-ok* :: *bool*
where *start-heap-ok* = *snd* (*snd* (*start-heap-data*))

definition *start-heap-obs* :: (*'addr*, *'thread-id*) *obs-event list*
where
start-heap-obs =
map (λ(*C*, *a*). *NewHeapElem a* (*Class-type C*)) (*zip initialization-list* (*fst* (*snd start-heap-data*)))

definition *start-addrs* :: *'addr list*
where *start-addrs* = *fst* (*snd start-heap-data*)

definition *addr-of-sys-xcpt* :: *cname* ⇒ *'addr*
where *addr-of-sys-xcpt C* = *the* (*map-of* (*zip initialization-list start-addrs*) *C*)

definition *start-tid* :: *'thread-id*
where *start-tid* = *addr2thread-id* (*hd start-addrs*)

definition *start-state* :: (*cname* ⇒ *mname* ⇒ *ty list* ⇒ *ty* ⇒ *'m* ⇒ *'addr val list* ⇒ *'x*) ⇒ *'m prog*
⇒ *cname* ⇒ *mname* ⇒ *'addr val list* ⇒ (*'addr*,*'thread-id*,*'x*,*'heap*,*'addr*) *state*
where
start-state f P C M vs ≡
let (*D*, *Ts*, *T*, *m*) = *method P C M*
in (*K*$ *None*, ([*start-tid* ↦ (*f D M Ts T* (*the m*) *vs*, *no-wait-locks*)], *start-heap*), *Map.empty*, {})

lemma *create-initial-object-simps*:
create-initial-object (*h*, *ads*, *b*) *C* =
(*if b*
then let HA = *allocate h* (*Class-type C*)
in if HA = {} *then* (*h*, *ads*, *False*)
else let (*h*′, *a*′′) = *SOME ha*. *ha* ∈ *HA in* (*h*′, *ads* @ [*a*′′], *True*)
else (*h*, *ads*, *False*))
unfolding *create-initial-object-def* **by** *simp*

lemma *create-initial-object-False* [*simp*]:
create-initial-object (*h*, *ads*, *False*) *C* = (*h*, *ads*, *False*)
by(*simp add*: *create-initial-object-simps*)

lemma *foldl-create-initial-object-False* [*simp*]:
foldl create-initial-object (*h*, *ads*, *False*) *Cs* = (*h*, *ads*, *False*)
by(*induct Cs*) *simp-all*

lemma *NewHeapElem-start-heap-obs-start-addrsD*:
NewHeapElem a CTn ∈ *set start-heap-obs* ⟹ *a* ∈ *set start-addrs*
unfolding *start-heap-obs-def start-addrs-def*
by(*auto dest*: *set-zip-rightD*)

lemma *shr-start-state*: *shr* (*start-state f P C M vs*) = *start-heap*
by(*simp add*: *start-state-def split-beta*)

lemma *start-heap-obs-not-Read*:
ReadMem ad al v ∉ *set start-heap-obs*
unfolding *start-heap-obs-def* **by** *auto*

lemma *length-initialization-list-le-length-start-addrs*:
length initialization-list ≥ *length start-addrs*
proof −
{ fix *h ads xs*
have *length* (*fst* (*snd* (*foldl create-initial-object* (*h*, *ads*, *True*) *xs*))) ≤ *length ads* + *length xs*
proof(*induct xs arbitrary*: *h ads*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*)
from *this*[*of fst* (*SOME ha*. *ha* ∈ *allocate h* (*Class-type x*)) *ads* @ [*snd* (*SOME ha*. *ha* ∈ *allocate h* (*Class-type x*))]]
show *?case* **by**(*clarsimp simp add*: *create-initial-object-simps split-beta*)
qed }
from *this*[*of empty-heap* [] *initialization-list*]
show *?thesis* **unfolding** *start-heap-def start-addrs-def start-heap-data-def* **by** *simp*
qed

lemma (**in** −) *distinct-initialization-list*:
distinct initialization-list
by(*simp add*: *initialization-list-def sys-xcpts-list-def sys-xcpts-neqs Thread-neq-sys-xcpts*)

lemma (**in** −) *wf-syscls-initialization-list-is-class*:
⟦ *wf-syscls P*; *C* ∈ *set initialization-list* ⟧ ⟹ *is-class P C*
by(*auto simp add*: *initialization-list-def sys-xcpts-list-def wf-syscls-is-class-xcpt*)

lemma *start-addrs-NewHeapElem-start-heap-obsD*:
a ∈ *set start-addrs* ⟹ ∃ *CTn*. *NewHeapElem a CTn* ∈ *set start-heap-obs*
using *length-initialization-list-le-length-start-addrs*
unfolding *start-heap-obs-def start-addrs-def*
by(*force simp add*: *set-zip in-set-conv-nth intro*: *rev-image-eqI*)

lemma *in-set-start-addrs-conv-NewHeapElem*:
a ∈ *set start-addrs* ⟷ (∃ *CTn*. *NewHeapElem a CTn* ∈ *set start-heap-obs*)
by(*blast dest*: *start-addrs-NewHeapElem-start-heap-obsD intro*: *NewHeapElem-start-heap-obs-start-addrsD*)

3.9.1 *preallocated*

definition *preallocated* :: *'heap* ⇒ *bool*
where *preallocated h* ≡ ∀ *C* ∈ *sys-xcpts*. *typeof-addr h* (*addr-of-sys-xcpt C*) = ⌊*Class-type C*⌋

lemma *typeof-addr-sys-xcp*:
⟦ *preallocated h*; *C* ∈ *sys-xcpts* ⟧ ⟹ *typeof-addr h* (*addr-of-sys-xcpt C*) = ⌊*Class-type C*⌋
by(*simp add*: *preallocated-def*)

lemma *typeof-sys-xcp*:
⟦ *preallocated h*; *C* ∈ *sys-xcpts* ⟧ ⟹ *typeof*$_h$ (*Addr* (*addr-of-sys-xcpt C*)) = ⌊*Class C*⌋
by(*simp add*: *typeof-addr-sys-xcp*)

lemma *addr-of-sys-xcpt-start-addr*:
⟦ *start-heap-ok*; *C* ∈ *sys-xcpts* ⟧ ⟹ *addr-of-sys-xcpt C* ∈ *set start-addrs*
unfolding *start-heap-ok-def start-heap-data-def initialization-list-def sys-xcpts-list-def*
preallocated-def start-heap-def start-addrs-def
apply(*simp split*: *prod.split-asm if-split-asm add*: *create-initial-object-simps*)
apply(*erule sys-xcpts-cases*)
apply(*simp-all add*: *addr-of-sys-xcpt-def start-addrs-def start-heap-data-def initialization-list-def sys-xcpts-list-def create-initial-object-simps*)
done

lemma [*simp*]:
assumes *preallocated h*
shows *typeof-ClassCast*: *typeof-addr h* (*addr-of-sys-xcpt ClassCast*) = *Some*(*Class-type ClassCast*)
and *typeof-OutOfMemory*: *typeof-addr h* (*addr-of-sys-xcpt OutOfMemory*) = *Some*(*Class-type OutOfMemory*)
and *typeof-NullPointer*: *typeof-addr h* (*addr-of-sys-xcpt NullPointer*) = *Some*(*Class-type NullPointer*)

and *typeof-ArrayIndexOutOfBounds*:
typeof-addr h (*addr-of-sys-xcpt ArrayIndexOutOfBounds*) = *Some*(*Class-type ArrayIndexOutOfBounds*)

and *typeof-ArrayStore*: *typeof-addr h* (*addr-of-sys-xcpt ArrayStore*) = *Some*(*Class-type ArrayStore*)
and *typeof-NegativeArraySize*: *typeof-addr h* (*addr-of-sys-xcpt NegativeArraySize*) = *Some*(*Class-type NegativeArraySize*)
and *typeof-ArithmeticException*: *typeof-addr h* (*addr-of-sys-xcpt ArithmeticException*) = *Some*(*Class-type ArithmeticException*)
and *typeof-IllegalMonitorState*: *typeof-addr h* (*addr-of-sys-xcpt IllegalMonitorState*) = *Some*(*Class-type IllegalMonitorState*)
and *typeof-IllegalThreadState*: *typeof-addr h* (*addr-of-sys-xcpt IllegalThreadState*) = *Some*(*Class-type IllegalThreadState*)
and *typeof-InterruptedException*: *typeof-addr h* (*addr-of-sys-xcpt InterruptedException*) = *Some*(*Class-type*

InterruptedException)
using *assms*
by(*simp-all add*: *typeof-addr-sys-xcp*)

lemma *cname-of-xcp* [*simp*]:
⟦ *preallocated h*; *C* ∈ *sys-xcpts* ⟧ ⟹ *cname-of h* (*addr-of-sys-xcpt C*) = *C*
by(*drule* (*1*) *typeof-addr-sys-xcp*)(*simp add*: *cname-of-def*)

lemma *preallocated-hext*:
⟦ *preallocated h*; *h* ◁ *h'* ⟧ ⟹ *preallocated h'*
by(*auto simp add*: *preallocated-def dest*: *hext-objD*)

end

context *heap* **begin**

lemma *preallocated-heap-ops*:
assumes *preallocated h*
shows *preallocated-allocate*: ⋀*a*. (*h'*, *a*) ∈ *allocate h hT* ⟹ *preallocated h'*
and *preallocated-write-field*: *heap-write h a al v h'* ⟹ *preallocated h'*
using *preallocated-hext*[*OF assms*, *of h'*]
by(*blast intro*: *hext-heap-ops*)+

lemma *not-empty-pairE*: ⟦ *A* ≠ {}; ⋀*a b*. (*a*, *b*) ∈ *A* ⟹ *thesis* ⟧ ⟹ *thesis*
by *auto*

lemma *allocate-not-emptyI*: (*h'*, *a*) ∈ *allocate h hT* ⟹ *allocate h hT* ≠ {}
by *auto*

lemma *allocate-Eps*:
⟦ (*h''*, *a''*) ∈ *allocate h hT*; (*SOME ha*. *ha* ∈ *allocate h hT*) = (*h'*, *a'*) ⟧ ⟹ (*h'*, *a'*) ∈ *allocate h hT*
by(*drule sym*)(*auto intro*: *someI*)

lemma *preallocated-start-heap*:
⟦ *start-heap-ok*; *wf-syscls P* ⟧ ⟹ *preallocated start-heap*
unfolding *start-heap-ok-def start-heap-data-def initialization-list-def sys-xcpts-list-def*
preallocated-def start-heap-def start-addrs-def
apply(*clarsimp split*: *prod.split-asm if-split-asm simp add*: *create-initial-object-simps*)
apply(*erule not-empty-pairE*)+
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*rotate-tac 13*)
apply(*frule allocate-SomeD*, *simp add*: *wf-syscls-is-class-xcpt*, *frule hext-allocate*, *rotate-tac 1*)
apply(*frule allocate-SomeD*, *simp add*: *wf-syscls-is-class-xcpt*, *frule hext-allocate*, *rotate-tac 1*)
apply(*frule allocate-SomeD*, *simp add*: *wf-syscls-is-class-xcpt*, *frule hext-allocate*, *rotate-tac 1*)

apply(*frule allocate-SomeD*, *simp add*: *wf-syscls-is-class-xcpt*, *frule hext-allocate*, *rotate-tac 1*)
apply(*frule allocate-SomeD*, *simp add*: *wf-syscls-is-class-xcpt*, *frule hext-allocate*, *rotate-tac 1*)
apply(*frule allocate-SomeD*, *simp add*: *wf-syscls-is-class-xcpt*, *frule hext-allocate*, *rotate-tac 1*)
apply(*frule allocate-SomeD*, *simp add*: *wf-syscls-is-class-xcpt*, *frule hext-allocate*, *rotate-tac 1*)
apply(*frule allocate-SomeD*, *simp add*: *wf-syscls-is-class-xcpt*, *frule hext-allocate*, *rotate-tac 1*)
apply(*frule allocate-SomeD*, *simp add*: *wf-syscls-is-class-xcpt*, *frule hext-allocate*, *rotate-tac 1*)
apply(*frule allocate-SomeD*, *simp add*: *wf-syscls-is-class-xcpt*, *frule hext-allocate*, *rotate-tac 1*)
apply(*frule allocate-SomeD*, *simp add*: *wf-syscls-is-class-xcpt*, *frule hext-allocate*, *rotate-tac 1*)
apply(*erule sys-xcpts-cases*)
apply(*simp-all add*: *addr-of-sys-xcpt-def initialization-list-def sys-xcpts-list-def sys-xcpts-neqs Thread-neq-sys-xcpts*
start-heap-data-def start-addrs-def create-initial-object-simps allocate-not-emptyI split del: *if-split*)
apply(*assumption*|*erule typeof-addr-hext-mono*)+
done

lemma *start-tid-start-addrs*:
⟦ *wf-syscls P*; *start-heap-ok* ⟧ ⟹ *thread-id2addr start-tid* ∈ *set start-addrs*
unfolding *start-heap-ok-def start-heap-data-def initialization-list-def sys-xcpts-list-def*
preallocated-def start-heap-def start-addrs-def
apply(*simp split*: *prod.split-asm if-split-asm add*: *create-initial-object-simps addr-of-sys-xcpt-def start-addrs-def*
start-tid-def start-heap-data-def initialization-list-def sys-xcpts-list-def)
apply(*erule not-empty-pairE*)+
apply(*drule* (*1*) *allocate-Eps*)
apply(*rotate-tac* −*1*)
apply(*drule allocate-SomeD*, *simp*)
apply(*auto intro*: *addr2thread-id-inverse*)
done

lemma
assumes *wf-syscls P*
shows *dom-typeof-addr-start-heap*: *set start-addrs* ⊆ *dom* (*typeof-addr start-heap*)
and *distinct-start-addrs*: *distinct start-addrs*
proof −
{ fix *h ads b* **and** *Cs xs* :: *cname list*
assume *set ads* ⊆ *dom* (*typeof-addr h*) **and** *distinct* (*Cs* @ *xs*) **and** *length Cs* = *length ads*
and ⋀*C a*. (*C*, *a*) ∈ *set* (*zip Cs ads*) ⟹ *typeof-addr h a* = ⌊*Class-type C*⌋
and ⋀*C*. *C* ∈ *set xs* ⟹ *is-class P C*
hence *set* (*fst* (*snd* (*foldl create-initial-object* (*h*, *ads*, *b*) *xs*))) ⊆
dom (*typeof-addr* (*fst* (*foldl create-initial-object* (*h*, *ads*, *b*) *xs*))) ∧
(*distinct ads* ⟶ *distinct* (*fst* (*snd* (*foldl create-initial-object* (*h*, *ads*, *b*) *xs*))))
(**is** *?concl xs h ads b Cs*)
proof(*induct xs arbitrary*: *h ads b Cs*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons x xs*)
note *ads* = ‹*set ads* ⊆ *dom* (*typeof-addr h*)›
note *dist* = ‹*distinct* (*Cs* @ *x* # *xs*)›
note *len* = ‹*length Cs* = *length ads*›
note *type* = ‹⋀*C a*. (*C*, *a*) ∈ *set* (*zip Cs ads*) ⟹ *typeof-addr h a* = ⌊*Class-type C*⌋›
note *is-class* = ‹⋀*C*. *C* ∈ *set* (*x* # *xs*) ⟹ *is-class P C*›
show *?case*
proof(*cases b* ∧ *allocate h* (*Class-type x*) ≠ {})
case *False* **thus** *?thesis*
using *ads len* **by**(*auto simp add*: *create-initial-object-simps zip-append1*)
next

```
      case [simp]: True
      obtain h' a' where h'a': (SOME ha. ha ∈ allocate h (Class-type x)) = (h', a')
        by(cases SOME ha. ha ∈ allocate h (Class-type x))
      with True have new-obj: (h', a') ∈ allocate h (Class-type x)
        by(auto simp del: True intro: allocate-Eps)
      hence hext: h ◁ h' by(rule hext-allocate)
      with ads new-obj have ads': set ads ⊆ dom (typeof-addr h')
        by(auto dest: typeof-addr-hext-mono[OF hext-allocate])
      moreover {
        from new-obj ads' is-class[of x]
        have set (ads @ [a']) ⊆ dom (typeof-addr h')
          by(auto dest: allocate-SomeD)
        moreover from dist have distinct ((Cs @ [x]) @ xs) by simp
        moreover have length (Cs @ [x]) = length (ads @ [a']) using len by simp
        moreover {
          fix C a
          assume (C, a) ∈ set (zip (Cs @ [x]) (ads @ [a']))
          hence typeof-addr h' a = ⌊Class-type C⌋
            using hext new-obj type[of C a] len is-class
            by(auto dest: allocate-SomeD hext-objD) }
        note type' = this
        moreover have is-class': ⋀C. C ∈ set xs ⟹ is-class P C using is-class by simp
        ultimately have ?concl xs h' (ads @ [a']) True (Cs @ [x]) by(rule Cons)
        moreover have a' ∉ set ads
        proof
          assume a': a' ∈ set ads
          then obtain C where (C, a') ∈ set (zip Cs ads) C ∈ set Cs
            using len unfolding set-zip in-set-conv-nth by auto
          hence typeof-addr h a' = ⌊Class-type C⌋ by−(rule type)
          with hext have typeof-addr h' a' = ⌊Class-type C⌋ by(rule typeof-addr-hext-mono)
          moreover from new-obj is-class
          have typeof-addr h' a' = ⌊Class-type x⌋ by(auto dest: allocate-SomeD)
          ultimately have C = x by simp
          with dist ‹C ∈ set Cs› show False by simp
        qed
        moreover note calculation }
      ultimately show ?thesis by(simp add: create-initial-object-simps new-obj h'a')
    qed
  qed }
  from this[of [] empty-heap [] initialization-list True]
    distinct-initialization-list wf-syscls-initialization-list-is-class[OF assms]
  show set start-addrs ⊆ dom (typeof-addr start-heap)
    and distinct start-addrs
    unfolding start-heap-def start-addrs-def start-heap-data-def by auto
qed

lemma NewHeapElem-start-heap-obsD:
  assumes wf-syscls P
  and NewHeapElem a hT ∈ set start-heap-obs
  shows typeof-addr start-heap a = ⌊hT⌋
proof −
  show ?thesis
  proof(cases hT)
    case (Class-type C)
```

```
  { fix h ads b xs Cs
    assume (C, a) ∈ set (zip (Cs @ xs) (fst (snd (foldl create-initial-object (h, ads, b) xs))))
     and ∀(C, a) ∈ set (zip Cs ads). typeof-addr h a = ⌊Class-type C⌋
     and length Cs = length ads
     and ∀ C ∈ set xs. is-class P C
    hence typeof-addr (fst (foldl create-initial-object (h, ads, b) xs)) a = ⌊Class-type C⌋
    proof(induct xs arbitrary: h ads b Cs)
     case Nil thus ?case by auto
    next
     case (Cons x xs)
     note inv = ‹∀(C, a) ∈ set (zip Cs ads). typeof-addr h a = ⌊Class-type C⌋›
      and Ca = ‹(C, a) ∈ set (zip (Cs @ x # xs) (fst (snd (foldl create-initial-object (h, ads, b) (x
# xs)))))›
      and len = ‹length Cs = length ads›
      and is-class = ‹∀ C ∈ set (x # xs). is-class P C›
     show ?case
     proof(cases b ∧ allocate h (Class-type x) ≠ {})
      case False thus ?thesis
       using inv Ca len by(auto simp add: create-initial-object-simps zip-append1 split: if-split-asm)
     next
      case [simp]: True
      obtain h' a' where h'a': (SOME ha. ha ∈ allocate h (Class-type x)) = (h', a')
       by(cases SOME ha. ha ∈ allocate h (Class-type x))
      with True have new-obj: (h', a') ∈ allocate h (Class-type x)
       by(auto simp del: True intro: allocate-Eps)
      hence hext: h ≤| h' by(rule hext-allocate)

       have (C, a) ∈ set (zip ((Cs @ [x]) @ xs) (fst (snd (foldl create-initial-object (h', ads @ [a'],
True) xs))))
        using Ca new-obj by(simp add: create-initial-object-simps h'a')
      moreover have ∀(C, a)∈set (zip (Cs @ [x]) (ads @ [a'])). typeof-addr h' a = ⌊Class-type C⌋
      proof(clarify)
       fix C a
       assume (C, a) ∈ set (zip (Cs @ [x]) (ads @ [a']))
       thus typeof-addr h' a = ⌊Class-type C⌋
        using inv len hext new-obj is-class by(auto dest: allocate-SomeD typeof-addr-hext-mono)
      qed
      moreover have length (Cs @ [x]) = length (ads @ [a']) using len by simp
      moreover have ∀ C ∈ set xs. is-class P C using is-class by simp
       ultimately have typeof-addr (fst (foldl create-initial-object (h', ads @ [a'], True) xs)) a =
⌊Class-type C⌋
       by(rule Cons)
      thus ?thesis using new-obj by(simp add: create-initial-object-simps h'a')
     qed
    qed }
   from this[of [] initialization-list empty-heap [] True] assms wf-syscls-initialization-list-is-class[of P]

   show ?thesis by(auto simp add: start-heap-obs-def start-heap-data-def start-heap-def Class-type)
  next
   case Array-type thus ?thesis using assms
    by(auto simp add: start-heap-obs-def start-heap-data-def start-heap-def)
  qed
qed
```

end

3.9.2 Code generation

definition *pick-addr* :: (*'heap* × *'addr*) *set* ⇒ *'heap* × *'addr*
where *pick-addr HA* = (*SOME ha*. *ha* ∈ *HA*)

lemma *pick-addr-code* [*code*]:
pick-addr (*set* [*ha*]) = *ha*
by(*simp add*: *pick-addr-def*)

lemma (**in** *heap-base*) *start-heap-data-code*:
start-heap-data =
(*let*
(*h*, *ads*, *b*) = *foldl*
(λ(*h*, *ads*, *b*) *C*.
if b then
let HA = *allocate h* (*Class-type C*)
in if HA = {} *then* (*h*, *ads*, *False*)
else let (*h'*, *a''*) = *pick-addr HA in* (*h'*, *a''* # *ads*, *True*)
else (*h*, *ads*, *False*))
(*empty-heap*, [], *True*)
initialization-list
in (*h*, *rev ads*, *b*))
unfolding *start-heap-data-def create-initial-object-def pick-addr-def*
by(*rule rev-induct*)(*simp-all add*: *split-beta*)

lemmas [*code*] =
heap-base.start-heap-data-code
heap-base.start-heap-def
heap-base.start-heap-ok-def
heap-base.start-heap-obs-def
heap-base.start-addrs-def
heap-base.addr-of-sys-xcpt-def
heap-base.start-tid-def
heap-base.start-state-def

end

3.10 Conformance Relations for Type Soundness Proofs

theory *Conform*
imports
StartConfig
begin

context *heap-base* **begin**

definition *conf* :: *'m prog* ⇒ *'heap* ⇒ *'addr val* ⇒ *ty* ⇒ *bool* (‹-,- ⊢ - :≤ -› [*51*,*51*,*51*,*51*] *50*)
where *P*,*h* ⊢ *v* :≤ *T* ≡ ∃ *T'*. *typeof*$_h$ *v* = *Some T'* ∧ *P* ⊢ *T'* ≤ *T*

definition *lconf* :: *'m prog* ⇒ *'heap* ⇒ (*vname* ⇀ *'addr val*) ⇒ (*vname* ⇀ *ty*) ⇒ *bool* (‹-,- ⊢ - *'*(:≤*'*)
-› [*51*,*51*,*51*,*51*] *50*)
where *P*,*h* ⊢ *l* (:≤) *E* ≡ ∀ *V v*. *l V* = *Some v* ⟶ (∃ *T*. *E V* = *Some T* ∧ *P*,*h* ⊢ *v* :≤ *T*)

abbreviation *confs* :: *'m prog* ⇒ *'heap* ⇒ *'addr val list* ⇒ *ty list* ⇒ *bool* (‹-,- ⊢ - [:≤] -› [*51*,*51*,*51*,*51*] *50*)
where *P*,*h* ⊢ *vs* [:≤] *Ts* == *list-all2* (*conf P h*) *vs Ts*

definition *tconf* :: *'m prog* ⇒ *'heap* ⇒ *'thread-id* ⇒ *bool* (‹-,- ⊢ - √*t*› [*51*,*51*,*51*] *50*)
where *P*,*h* ⊢ *t* √*t* ≡ ∃ *C*. *typeof-addr h* (*thread-id2addr t*) = ⌊*Class-type C*⌋ ∧ *P* ⊢ *C* ⪯* *Thread*

end

locale *heap-conf-base* =
heap-base +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
fixes *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'m prog*

sublocale *heap-conf-base* < *prog P* **.**

locale *heap-conf* =
heap
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
P
+
heap-conf-base
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
hconf P
for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'m prog*
+
assumes *hconf-empty* [*iff*]: *hconf empty-heap*
and *typeof-addr-is-type*: ⟦ *typeof-addr h a* = ⌊*hT*⌋; *hconf h* ⟧ ⟹ *is-type P* (*ty-of-htype hT*)
and *hconf-allocate-mono*: ⋀*a*. ⟦ (*h'*, *a*) ∈ *allocate h hT*; *hconf h*; *is-htype P hT* ⟧ ⟹ *hconf h'*
and *hconf-heap-write-mono*:
⋀*T*. ⟦ *heap-write h a al v h'*; *hconf h*; *P*,*h* ⊢ *a*@*al* : *T*; *P*,*h* ⊢ *v* :≤ *T* ⟧ ⟹ *hconf h'*

locale *heap-progress* =
  heap-conf
    addr2thread-id thread-id2addr
    spurious-wakeups
    empty-heap allocate typeof-addr heap-read heap-write
    hconf P
  for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
  and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
  and *spurious-wakeups* :: *bool*
  and *empty-heap* :: *'heap*
  and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
  and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
  and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
  and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
  and *hconf* :: *'heap* ⇒ *bool*
  and *P* :: *'m prog*
  +
  assumes *heap-read-total*: ⟦ *hconf h*; *P*,*h* ⊢ *a*@*al* : *T* ⟧ ⟹ ∃ *v*. *heap-read h a al v* ∧ *P*,*h* ⊢ *v* :≤ *T*
  and *heap-write-total*: ⟦ *hconf h*; *P*,*h* ⊢ *a*@*al* : *T*; *P*,*h* ⊢ *v* :≤ *T* ⟧ ⟹ ∃ *h'*. *heap-write h a al v h'*

locale *heap-conf-read* =
  heap-conf
    addr2thread-id thread-id2addr
    spurious-wakeups
    empty-heap allocate typeof-addr heap-read heap-write
    hconf P
  for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
  and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
  and *spurious-wakeups* :: *bool*
  and *empty-heap* :: *'heap*
  and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
  and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
  and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
  and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
  and *hconf* :: *'heap* ⇒ *bool*
  and *P* :: *'m prog*
  +
  assumes *heap-read-conf*: ⟦ *heap-read h a al v*; *P*,*h* ⊢ *a*@*al* : *T*; *hconf h* ⟧ ⟹ *P*,*h* ⊢ *v* :≤ *T*

locale *heap-typesafe* =
  heap-conf-read +
  heap-progress +
  constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
  and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
  and *spurious-wakeups* :: *bool*
  and *empty-heap* :: *'heap*
  and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
  and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
  and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
  and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
  and *hconf* :: *'heap* ⇒ *bool*
  and *P* :: *'m prog*

context *heap-conf* **begin**

lemmas *hconf-heap-ops-mono* =
hconf-allocate-mono
hconf-heap-write-mono

end

3.10.1 Value conformance $:\leq$

context *heap-base* **begin**

lemma *conf-Null* [*simp*]: $P,h \vdash Null :\leq T = P \vdash NT \leq T$
unfolding *conf-def* **by**(*simp* (*no-asm*))

lemma *typeof-conf*[*simp*]: $typeof_h\ v = Some\ T \Longrightarrow P,h \vdash v :\leq T$
unfolding *conf-def* **by** (*cases v*) *auto*

lemma *typeof-lit-conf*[*simp*]: $typeof\ v = Some\ T \Longrightarrow P,h \vdash v :\leq T$
by (*rule typeof-conf*[*OF typeof-lit-typeof*])

lemma *defval-conf*[*simp*]: $P,h \vdash default\text{-}val\ T :\leq T$
unfolding *conf-def* **by** (*cases T*) *auto*

lemma *conf-widen*: $P,h \vdash v :\leq T \Longrightarrow P \vdash T \leq T' \Longrightarrow P,h \vdash v :\leq T'$
unfolding *conf-def* **by** (*cases v*) (*auto intro*: *widen-trans*)

lemma *conf-sys-xcpt*:
$[\![preallocated\ h;\ C \in sys\text{-}xcpts]\!] \Longrightarrow P,h \vdash Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ C) :\leq Class\ C$
by(*simp add*: *conf-def typeof-addr-sys-xcp*)

lemma *conf-NT* [*iff*]: $P,h \vdash v :\leq NT = (v = Null)$
by (*auto simp add*: *conf-def*)

lemma *is-IntgI*: $P,h \vdash v :\leq Integer \Longrightarrow is\text{-}Intg\ v$
by (*unfold conf-def*) *auto*

lemma *is-BoolI*: $P,h \vdash v :\leq Boolean \Longrightarrow is\text{-}Bool\ v$
by (*unfold conf-def*) *auto*

lemma *is-RefI*: $P,h \vdash v :\leq T \Longrightarrow is\text{-}refT\ T \Longrightarrow is\text{-}Ref\ v$
by(*cases v*)(*auto elim*: *is-refT.cases simp add*: *conf-def is-Ref-def*)

lemma *non-npD*:
$[\![\ v \neq Null;\ P,h \vdash v :\leq Class\ C;\ C \neq Object\ ]\!]$
$\Longrightarrow \exists a\ C'.\ v = Addr\ a \wedge typeof\text{-}addr\ h\ a = \lfloor Class\text{-}type\ C' \rfloor \wedge P \vdash C' \preceq^* C$
by(*cases v*)(*auto simp add*: *conf-def widen-Class*)

lemma *non-npD2*:
$[\![v \neq Null;\ P,h \vdash v :\leq Class\ C\ ]\!]$
$\Longrightarrow \exists a\ hT.\ v = Addr\ a \wedge typeof\text{-}addr\ h\ a = \lfloor hT \rfloor \wedge P \vdash class\text{-}type\text{-}of\ hT \preceq^* C$
by(*cases v*)(*auto simp add*: *conf-def widen-Class*)

end

context *heap* **begin**

lemma *conf-hext*: ⟦ $h \trianglelefteq h'$; $P,h \vdash v :\leq T$ ⟧ ⟹ $P,h' \vdash v :\leq T$
unfolding *conf-def* **by**(*cases v*)(*auto dest*: *typeof-addr-hext-mono*)

lemma *conf-heap-ops-mono*:
  assumes $P,h \vdash v :\leq T$
  shows *conf-allocate-mono*: $(h', a) \in$ *allocate h hT* ⟹ $P,h' \vdash v :\leq T$
  and *conf-heap-write-mono*: *heap-write h a al v' h'* ⟹ $P,h' \vdash v :\leq T$
using *assms*
by(*auto intro*: *conf-hext dest*: *hext-heap-ops*)

end

3.10.2 Value list conformance $[:\leq]$

context *heap-base* **begin**

lemma *confs-widens* [*trans*]: ⟦$P,h \vdash vs$ $[:\leq]$ *Ts*; $P \vdash Ts$ $[\leq]$ *Ts'*⟧ ⟹ $P,h \vdash vs$ $[:\leq]$ *Ts'*
by (*rule list-all2-trans*)(*rule conf-widen*)

lemma *confs-rev*: $P,h \vdash$ *rev s* $[:\leq]$ *t* = ($P,h \vdash s$ $[:\leq]$ *rev t*)
by(*rule list-all2-rev1*)

lemma *confs-conv-map*:
  $P,h \vdash vs$ $[:\leq]$ *Ts'* = (∃ *Ts*. *map typeof*$_h$ *vs* = *map Some Ts* ∧ $P \vdash Ts$ $[\leq]$ *Ts'*)
apply(*induct vs arbitrary*: *Ts'*)
 apply *simp*
apply(*case-tac Ts'*)
apply(*auto simp add*:*conf-def*)
apply(*rule-tac x*=*T'* # *Ts* **in** *exI*)
apply(*simp add*: *fun-of-def*)
done

lemma *confs-Cons2*: $P,h \vdash xs$ $[:\leq]$ *y*#*ys* = (∃ *z zs*. *xs* = *z*#*zs* ∧ $P,h \vdash z :\leq y$ ∧ $P,h \vdash zs$ $[:\leq]$ *ys*)
by (*rule list-all2-Cons2*)

end

context *heap* **begin**

lemma *confs-hext*: $P,h \vdash vs$ $[:\leq]$ *Ts* ⟹ $h \trianglelefteq h'$ ⟹ $P,h' \vdash vs$ $[:\leq]$ *Ts*
by (*erule list-all2-mono*, *erule conf-hext*, *assumption*)

end

3.10.3 Local variable conformance

context *heap-base* **begin**

lemma *lconf-upd*:
  ⟦ $P,h \vdash l$ $(:\leq)$ *E*; $P,h \vdash v :\leq T$; *E V* = *Some T* ⟧ ⟹ $P,h \vdash l(V \mapsto v)$ $(:\leq)$ *E*
unfolding *lconf-def* **by** *auto*

lemma *lconf-empty* [*iff*]: $P,h \vdash Map.empty\ (:\leq)\ E$
by(*simp add*:*lconf-def*)

lemma *lconf-upd2*: $[\![P,h \vdash l\ (:\leq)\ E;\ P,h \vdash v :\leq T]\!] \Longrightarrow P,h \vdash l(V \mapsto v)\ (:\leq)\ E(V \mapsto T)$
by(*simp add*:*lconf-def*)

end

context *heap* **begin**

lemma *lconf-hext*: $[\![\ P,h \vdash l\ (:\leq)\ E;\ h \trianglelefteq h'\ ]\!] \Longrightarrow P,h' \vdash l\ (:\leq)\ E$
unfolding *lconf-def* **by**(*fast elim*: *conf-hext*)

end

3.10.4 Thread object conformance

context *heap-base* **begin**

lemma *tconfI*: $[\![\ typeof\text{-}addr\ h\ (thread\text{-}id2addr\ t) = \lfloor Class\text{-}type\ C \rfloor;\ P \vdash C \preceq^* Thread\ ]\!] \Longrightarrow P,h \vdash t\ \surd t$
by(*simp add*: *tconf-def*)

lemma *tconfD*: $P,h \vdash t\ \surd t \Longrightarrow \exists C.\ typeof\text{-}addr\ h\ (thread\text{-}id2addr\ t) = \lfloor Class\text{-}type\ C \rfloor \wedge P \vdash C \preceq^* Thread$
by(*auto simp add*: *tconf-def*)

end

context *heap* **begin**

lemma *tconf-hext-mono*: $[\![\ P,h \vdash t\ \surd t;\ h \trianglelefteq h'\ ]\!] \Longrightarrow P,h' \vdash t\ \surd t$
by(*auto simp add*: *tconf-def dest*: *typeof-addr-hext-mono*)

lemma *tconf-heap-ops-mono*:
  assumes $P,h \vdash t\ \surd t$
  shows *tconf-allocate-mono*: $(h',\ a) \in allocate\ h\ hT \Longrightarrow P,h' \vdash t\ \surd t$
  and *tconf-heap-write-mono*: *heap-write h a al v h'* $\Longrightarrow P,h' \vdash t\ \surd t$
using *tconf-hext-mono*[*OF assms*, *of h'*]
by(*blast intro*: *hext-heap-ops*)+

lemma *tconf-start-heap-start-tid*:
  $[\![$ *start-heap-ok*; *wf-syscls P* $]\!] \Longrightarrow P,start\text{-}heap \vdash start\text{-}tid\ \surd t$
unfolding *start-tid-def start-heap-def start-heap-ok-def start-heap-data-def initialization-list-def addr-of-sys-xcpt-de*
start-addrs-def sys-xcpts-list-def
apply(*clarsimp split*: *prod.split-asm simp add*: *create-initial-object-simps split*: *if-split-asm*)
apply(*erule not-empty-pairE*)+
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)

apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule* (*1*) *allocate-Eps*)
apply(*drule allocate-SomeD*[**where** *hT*=*Class-type Thread*])
 apply *simp*
apply(*rule tconfI*)
 apply(*erule typeof-addr-hext-mono*[*OF hext-allocate*])+
 apply *simp*
apply *blast*
done

lemma *start-heap-write-typeable*:
  assumes *WriteMem ad al v* ∈ *set start-heap-obs*
  shows ∃ *T*. *P*,*start-heap* ⊢ *ad*@*al* : *T* ∧ *P*,*start-heap* ⊢ *v* :≤ *T*
using *assms*
unfolding *start-heap-obs-def start-heap-def*
by *clarsimp*

end

3.10.5 Well-formed start state

context *heap-base* **begin**

inductive *wf-start-state* :: *'m prog* ⇒ *cname* ⇒ *mname* ⇒ *'addr val list* ⇒ *bool*
for *P* :: *'m prog* **and** *C* :: *cname* **and** *M* :: *mname* **and** *vs* :: *'addr val list*
where
  wf-start-state:
  ⟦ *P* ⊢ *C sees M*:*Ts*→*T* = ⌊*meth*⌋ *in D*; *start-heap-ok*; *P*,*start-heap* ⊢ *vs* [:≤] *Ts* ⟧
  ⟹ *wf-start-state P C M vs*

end

end

3.11 Semantics of method calls that cannot be defined inside JinjaThreads

theory *ExternalCall*
imports
  ../Framework/FWSemantics
  Conform
begin

type-synonym
  (*'addr*,*'thread-id*,*'heap*) *external-thread-action* = (*'addr*, *'thread-id*, *cname* × *mname* × *'addr*,*'heap*)
Jinja-thread-action

print-translation ‹
  let
    fun tr'

```
      [a1, t
      , Const (@{type-syntax prod}, -) $ Const (@{type-syntax String.literal}, -) $
          (Const (@{type-syntax prod}, -) $ Const (@{type-syntax String.literal}, -) $ a2)
      , h] =
      if a1 = a2 then Syntax.const @{type-syntax external-thread-action} $ a1 $ t $ h
      else raise Match;
    in [(@{type-syntax Jinja-thread-action}, K tr')]
  end
›
typ ('addr,'thread-id,'heap) external-thread-action
```

3.11.1 Typing of external calls

```
inductive external-WT-defs :: cname ⇒ mname ⇒ ty list ⇒ ty ⇒ bool (‹(-•-'(-'))› :: -› [50, 0, 0, 50] 60)
where
  Thread•start([]) :: Void
| Thread•join([]) :: Void
| Thread•interrupt([]) :: Void
| Thread•isInterrupted([]) :: Boolean
| Object•wait([]) :: Void
| Object•notify([]) :: Void
| Object•notifyAll([]) :: Void
| Object•clone([]) :: Class Object
| Object•hashcode([]) :: Integer
| Object•print([Integer]) :: Void
| Object•currentThread([]) :: Class Thread
| Object•interrupted([]) :: Boolean
| Object•yield([]) :: Void

inductive-cases external-WT-defs-cases:
  a•start(vs) :: T
  a•join(vs) :: T
  a•interrupt(vs) :: T
  a•isInterrupted(vs) :: T
  a•wait(vs) :: T
  a•notify(vs) :: T
  a•notifyAll(vs) :: T
  a•clone(vs) :: T
  a•hashcode(vs) :: T
  a•print(vs) :: T
  a•currentThread(vs) :: T
  a•interrupted([]) :: T
  a•yield(vs) :: T

inductive is-native :: 'm prog ⇒ htype ⇒ mname ⇒ bool
  for P :: 'm prog and hT :: htype and M :: mname
where ⟦ P ⊢ class-type-of hT sees M:Ts→T = Native in D; D•M(Ts) :: T ⟧ ⟹ is-native P hT M

lemma is-nativeD: is-native P hT M ⟹ ∃ Ts T D. P ⊢ class-type-of hT sees M:Ts→T = Native in D ∧ D•M(Ts)::T
by(simp add: is-native.simps)

inductive (in heap-base) external-WT' :: 'm prog ⇒ 'heap ⇒ 'addr ⇒ mname ⇒ 'addr val list ⇒ ty
```

$\Rightarrow$ *bool*
  (‹-,- ⊢ (-•-'(-')) : -› [50,0,0,0,50] 60)
for P :: *'m prog* **and** h :: *'heap* **and** a :: *'addr* **and** M :: *mname* **and** vs :: *'addr val list* **and** U :: *ty*
where
  ⟦ *typeof-addr h a* = ⌊*hT*⌋; *map typeof*$_h$ *vs* = *map Some Ts*; P ⊢ *class-type-of hT sees M*:*Ts'*→*U* = *Native in D*;
    P ⊢ *Ts* [≤] *Ts'* ⟧
  ⟹ P,h ⊢ *a*•*M*(*vs*) : *U*

context *heap-base* **begin**

lemma *external-WT'-iff*:
  P,h ⊢ *a*•*M*(*vs*) : *U* ⟷
  (∃ *hT Ts Ts' D*. *typeof-addr h a* = ⌊*hT*⌋ ∧ *map typeof*$_h$ *vs* = *map Some Ts* ∧ P ⊢ *class-type-of hT sees M*:*Ts'*→*U*=*Native in D* ∧ P ⊢ *Ts* [≤] *Ts'*)
by(*simp add*: *external-WT'.simps*)

end

context *heap* **begin**

lemma *external-WT'-hext-mono*:
  ⟦ P,h ⊢ *a*•*M*(*vs*) : *T*; h ⊴ h' ⟧ ⟹ P,h' ⊢ *a*•*M*(*vs*) : *T*
by(*auto 5 2 simp add*: *external-WT'-iff dest*: *typeof-addr-hext-mono map-typeof-hext-mono*)

end

3.11.2 Semantics of external calls

datatype *'addr extCallRet* =
    RetVal 'addr val
  | *RetExc 'addr*
  | *RetStaySame*

lemma *rec-extCallRet* [*simp*]: *rec-extCallRet* = *case-extCallRet*
by(*auto simp add*: *fun-eq-iff split*: *extCallRet.split*)

context *heap-base* **begin**

abbreviation *RetEXC* :: *cname* ⇒ *'addr extCallRet*
where *RetEXC C* ≡ *RetExc* (*addr-of-sys-xcpt C*)

inductive *heap-copy-loc* :: *'addr* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'heap* ⇒ (*'addr*, *'thread-id*) *obs-event list* ⇒ *'heap* ⇒ *bool*
for a :: *'addr* **and** a' :: *'addr* **and** *al* :: *addr-loc* **and** h :: *'heap*
where
  ⟦ *heap-read h a al v*; *heap-write h a' al v h'* ⟧
  ⟹ *heap-copy-loc a a' al h* ([*ReadMem a al v*, *WriteMem a' al v*]) *h'*

inductive *heap-copies* :: *'addr* ⇒ *'addr* ⇒ *addr-loc list* ⇒ *'heap* ⇒ (*'addr*, *'thread-id*) *obs-event list* ⇒ *'heap* ⇒ *bool*
for a :: *'addr* **and** a' :: *'addr*
where
  Nil: *heap-copies a a'* [] h [] h

| *Cons*:
⟦ *heap-copy-loc a a′ al h ob h′*; *heap-copies a a′ als h′ obs h″* ⟧
⟹ *heap-copies a a′* (*al* # *als*) *h* (*ob* @ *obs*) *h″*

inductive-cases *heap-copies-cases*:
heap-copies a a′ [] *h ops h′*
heap-copies a a′ (*al*#*als*) *h ops h′*

Contrary to Sun's JVM 1.6.0_07, cloning an interrupted thread does not yield an interrupted thread, because the interrupt flag is not stored inside the thread object. Starting a clone of a started thread with Sun JVM 1.6.0_07 raises an illegal thread state exception, we just start another thread. The thread at http://mail.openjdk.java.net/pipermail/core-libs-dev/2010-August/004715.html discusses the general problem of thread cloning and argues against that. The bug report http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6968584 changes the Thread class implementation such that `Object.clone()` can no longer be accessed for Thread and subclasses in Java 7.

Array cells are never volatile themselves.

inductive *heap-clone* :: *′m prog* ⇒ *′heap* ⇒ *′addr* ⇒ *′heap* ⇒ ((*′addr*, *′thread-id*) *obs-event list* × *′addr*) *option* ⇒ *bool*
for *P* :: *′m prog* **and** *h* :: *′heap* **and** *a* :: *′addr*
where
CloneFail:
⟦ *typeof-addr h a* = ⌊*hT*⌋; *allocate h hT* = {} ⟧
⟹ *heap-clone P h a h None*
| *ObjClone*:
⟦ *typeof-addr h a* = ⌊*Class-type C*⌋; (*h′*, *a′*) ∈ *allocate h* (*Class-type C*);
P ⊢ *C has-fields FDTs*; *heap-copies a a′* (*map* (λ((*F*, *D*), *Tfm*). *CField D F*) *FDTs*) *h′ obs h″* ⟧
⟹ *heap-clone P h a h″* ⌊(*NewHeapElem a′* (*Class-type C*) # *obs*, *a′*)⌋
| *ArrClone*:
⟦ *typeof-addr h a* = ⌊*Array-type T n*⌋; (*h′*, *a′*) ∈ *allocate h* (*Array-type T n*); *P* ⊢ *Object has-fields FDTs*;
heap-copies a a′ (*map* (λ((*F*, *D*), *Tfm*). *CField D F*) *FDTs* @ *map ACell* [*0*..<*n*]) *h′ obs h″* ⟧
⟹ *heap-clone P h a h″* ⌊(*NewHeapElem a′* (*Array-type T n*) # *obs*, *a′*)⌋

inductive *red-external* ::
′m prog ⇒ *′thread-id* ⇒ *′heap* ⇒ *′addr* ⇒ *mname* ⇒ *′addr val list*
⇒ (*′addr*, *′thread-id*, *′heap*) *external-thread-action* ⇒ *′addr extCallRet* ⇒ *′heap* ⇒ *bool*
and *red-external-syntax* ::
′m prog ⇒ *′thread-id* ⇒ *′addr* ⇒ *mname* ⇒ *′addr val list* ⇒ *′heap*
⇒ (*′addr*, *′thread-id*, *′heap*) *external-thread-action* ⇒ *′addr extCallRet* ⇒ *′heap* ⇒ *bool*
(‹-,- ⊢ (⟨(-•-′(-′)),/-⟩) −-→*ext* (⟨(-),/(-)⟩)› [*50*, *0*, *0*, *0*, *0*, *0*, *0*, *0*, *0*] *51*)
for *P* :: *′m prog* **and** *t* :: *′thread-id* **and** *h* :: *′heap* **and** *a* :: *′addr*
where
P,*t* ⊢ ⟨*a*•*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h′*⟩ ≡ *red-external P t h a M vs ta va h′*

| *RedNewThread*:
⟦ *typeof-addr h a* = ⌊*Class-type C*⌋; *P* ⊢ *C* ≼* *Thread* ⟧
⟹ *P*,*t* ⊢ ⟨*a*•*start*([]), *h*⟩ −{⟦*NewThread* (*addr2thread-id a*) (*C*, *run*, *a*) *h*, *ThreadStart* (*addr2thread-id a*)⟧}→*ext* ⟨*RetVal Unit*, *h*⟩

| *RedNewThreadFail*:
⟦ *typeof-addr h a* = ⌊*Class-type C*⌋; *P* ⊢ *C* ≼* *Thread* ⟧
⟹ *P*,*t* ⊢ ⟨*a*•*start*([]), *h*⟩ −{⟦*ThreadExists* (*addr2thread-id a*) *True*⟧}→*ext* ⟨*RetEXC IllegalThreadState*,

$h\rangle$

| *RedJoin*:
$[\![$ *typeof-addr* $h\ a = \lfloor$*Class-type* $C\rfloor$; $P \vdash C \preceq^*$ *Thread* $]\!]$
$\Longrightarrow P,t \vdash \langle a{\cdot}join([]),\ h\rangle\ -\{\!|$*Join* (*addr2thread-id* a), *IsInterrupted* t *False*, *ThreadJoin* (*addr2thread-id* a)$|\!\}\rightarrow ext\ \langle$*RetVal Unit*, $h\rangle$

| *RedJoinInterrupt*:
$[\![$ *typeof-addr* $h\ a = \lfloor$*Class-type* $C\rfloor$; $P \vdash C \preceq^*$ *Thread* $]\!]$
$\Longrightarrow P,t \vdash \langle a{\cdot}join([]),\ h\rangle\ -\{\!|$*IsInterrupted* t *True*, *ClearInterrupt* t, *ObsInterrupted* $t|\!\}\rightarrow ext\ \langle$*RetEXC InterruptedException*, $h\rangle$

— Interruption should produce inter-thread actions (JLS 17.4.4) for the synchronizes-with order. They should synchronize with the inter-thread actions that determine whether a thread has been interrupted. Hence, interruption generates an *ObsInterrupt* action.
Although *WakeUp* causes the interrupted thread to raise an *InterruptedException* independent of the interrupt status, the interrupt flag must be set with *Interrupt* such that other threads observe the interrupted thread as interrupted while it competes for the monitor lock again.
Interrupting a thread which has not yet been started does not set the interrupt flag (tested with Sun HotSpot JVM 1.6.0_07).

| *RedInterrupt*:
$[\![$ *typeof-addr* $h\ a = \lfloor$*Class-type* $C\rfloor$; $P \vdash C \preceq^*$ *Thread* $]\!]$
$\Longrightarrow P,t \vdash \langle a{\cdot}interrupt([]),\ h\rangle$
$-\{\!|$*ThreadExists* (*addr2thread-id* a) *True*, *WakeUp* (*addr2thread-id* a),
Interrupt (*addr2thread-id* a), *ObsInterrupt* (*addr2thread-id* a)$|\!\}\rightarrow ext$
$\langle$*RetVal Unit*, $h\rangle$

| *RedInterruptInexist*:
$[\![$ *typeof-addr* $h\ a = \lfloor$*Class-type* $C\rfloor$; $P \vdash C \preceq^*$ *Thread* $]\!]$
$\Longrightarrow P,t \vdash \langle a{\cdot}interrupt([]),\ h\rangle$
$-\{\!|$*ThreadExists* (*addr2thread-id* a) *False*$|\!\}\rightarrow ext$
$\langle$*RetVal Unit*, $h\rangle$

| *RedIsInterruptedTrue*:
$[\![$ *typeof-addr* $h\ a = \lfloor$*Class-type* $C\rfloor$; $P \vdash C \preceq^*$ *Thread* $]\!]$
$\Longrightarrow P,t \vdash \langle a{\cdot}isInterrupted([]),\ h\rangle\ -\{\!|$ *IsInterrupted* (*addr2thread-id* a) *True*, *ObsInterrupted* (*addr2thread-id* a)$|\!\}\rightarrow ext$
$\langle$*RetVal* (*Bool True*), $h\rangle$

| *RedIsInterruptedFalse*:
$[\![$ *typeof-addr* $h\ a = \lfloor$*Class-type* $C\rfloor$; $P \vdash C \preceq^*$ *Thread* $]\!]$
$\Longrightarrow P,t \vdash \langle a{\cdot}isInterrupted([]),\ h\rangle\ -\{\!|$*IsInterrupted* (*addr2thread-id* a) *False*$|\!\}\rightarrow ext\ \langle$*RetVal* (*Bool False*), $h\rangle$

— The JLS leaves unspecified whether *wait* first checks for the monitor state (whether the thread holds a lock on the monitor) or for the interrupt flag of the current thread. Sun Hotspot JVM 1.6.0_07 seems to check for the monitor state first, so we do it here, too.
| *RedWaitInterrupt*:
$P,t \vdash \langle a{\cdot}wait([]),\ h\rangle\ -\{\!|$*Unlock*$\rightarrow a$, *Lock*$\rightarrow a$, *IsInterrupted* t *True*, *ClearInterrupt* t, *ObsInterrupted* $t|\!\}\rightarrow ext$
$\langle$*RetEXC InterruptedException*, $h\rangle$

| *RedWait*:

$P,t \vdash \langle a{\cdot}wait([]),\ h\rangle\ -\{\!|Suspend\ a,\ Unlock{\rightarrow}a,\ Lock{\rightarrow}a,\ ReleaseAcquire{\rightarrow}a,\ IsInterrupted\ t\ False,\ SyncUnlock\ a\ |\!\}{\rightarrow}ext$
$\langle RetStaySame,\ h\rangle$

| *RedWaitFail*:
$P,t \vdash \langle a{\cdot}wait([]),\ h\rangle\ -\{\!|UnlockFail{\rightarrow}a|\!\}{\rightarrow}ext\ \langle RetEXC\ IllegalMonitorState,\ h\rangle$

| *RedWaitNotified*:
$P,t \vdash \langle a{\cdot}wait([]),\ h\rangle\ -\{\!|Notified|\!\}{\rightarrow}ext\ \langle RetVal\ Unit,\ h\rangle$

— This rule does NOT check that the interrupted flag is set, but still clears it. The semantics will be that only the executing thread clears its interrupt.
| *RedWaitInterrupted*:
$P,t \vdash \langle a{\cdot}wait([]),\ h\rangle\ -\{\!|WokenUp,\ ClearInterrupt\ t,\ ObsInterrupted\ t|\!\}{\rightarrow}ext\ \langle RetEXC\ InterruptedException,\ h\rangle$

— Calls to wait may decide to immediately wake up spuriously. This is indistinguishable from waking up spuriously any time before being notified or interrupted. Spurious wakeups are configured by the *spurious-wakeup* parameter of the *heap-base* locale.
| *RedWaitSpurious*:
spurious-wakeups $\Longrightarrow$
$P,t \vdash \langle a{\cdot}wait([]),\ h\rangle\ -\{\!|Unlock{\rightarrow}a,\ Lock{\rightarrow}a,\ ReleaseAcquire{\rightarrow}a,\ IsInterrupted\ t\ False,\ SyncUnlock\ a|\!\}\ {\rightarrow}ext$
$\langle RetVal\ Unit,\ h\rangle$

— *notify* and *notifyAll* do not perform synchronization inter-thread actions because they only tests whether the thread holds a lock, but do not change the lock state.

| *RedNotify*:
$P,t \vdash \langle a{\cdot}notify([]),\ h\rangle\ -\{\!|Notify\ a,\ Unlock{\rightarrow}a,\ Lock{\rightarrow}a|\!\}{\rightarrow}ext\ \langle RetVal\ Unit,\ h\rangle$

| *RedNotifyFail*:
$P,t \vdash \langle a{\cdot}notify([]),\ h\rangle\ -\{\!|UnlockFail{\rightarrow}a|\!\}{\rightarrow}ext\ \langle RetEXC\ IllegalMonitorState,\ h\rangle$

| *RedNotifyAll*:
$P,t \vdash \langle a{\cdot}notifyAll([]),\ h\rangle\ -\{\!|NotifyAll\ a,\ Unlock{\rightarrow}a,\ Lock{\rightarrow}a|\!\}{\rightarrow}ext\ \langle RetVal\ Unit,\ h\rangle$

| *RedNotifyAllFail*:
$P,t \vdash \langle a{\cdot}notifyAll([]),\ h\rangle\ -\{\!|UnlockFail{\rightarrow}a|\!\}{\rightarrow}ext\ \langle RetEXC\ IllegalMonitorState,\ h\rangle$

| *RedClone*:
heap-clone $P\ h\ a\ h'\ \lfloor(obs,\ a')\rfloor$
$\Longrightarrow P,t \vdash \langle a{\cdot}clone([]),\ h\rangle\ -(K\$\ [],\ [],\ [],\ [],\ [],\ obs){\rightarrow}ext\ \langle RetVal\ (Addr\ a'),\ h'\rangle$

| *RedCloneFail*:
heap-clone $P\ h\ a\ h'\ None \Longrightarrow P,t \vdash \langle a{\cdot}clone([]),\ h\rangle\ -\varepsilon{\rightarrow}ext\ \langle RetEXC\ OutOfMemory,\ h'\rangle$

| *RedHashcode*:
$P,t \vdash \langle a{\cdot}hashcode([]),\ h\rangle\ -\{\!|\!|\!\}{\rightarrow}ext\ \langle RetVal\ (Intg\ (word\text{-}of\text{-}int\ (hash\text{-}addr\ a))),\ h\rangle$

| *RedPrint*:
$P,t \vdash \langle a{\cdot}print(vs),\ h\rangle\ -\{\!|ExternalCall\ a\ print\ vs\ Unit|\!\}{\rightarrow}ext\ \langle RetVal\ Unit,\ h\rangle$

| *RedCurrentThread*:

$P,t \vdash \langle a{\cdot}currentThread([]),\ h\rangle\ -\{\!|\!\}\!\rightarrow ext\ \langle RetVal\ (Addr\ (thread\text{-}id2addr\ t)),\ h\rangle$

| *RedInterruptedTrue*:
$P,t \vdash \langle a{\cdot}interrupted([]),\ h\rangle\ -\{\!|IsInterrupted\ t\ True,\ ClearInterrupt\ t,\ ObsInterrupted\ t|\!\}\!\rightarrow ext\ \langle RetVal\ (Bool\ True),\ h\rangle$

| *RedInterruptedFalse*:
$P,t \vdash \langle a{\cdot}interrupted([]),\ h\rangle\ -\{\!|IsInterrupted\ t\ False|\!\}\!\rightarrow ext\ \langle RetVal\ (Bool\ False),\ h\rangle$

| *RedYield*:
$P,t \vdash \langle a{\cdot}yield([]),\ h\rangle\ -\{\!|Yield|\!\}\!\rightarrow ext\ \langle RetVal\ Unit,\ h\rangle$

3.11.3 Aggressive formulation for external cals

definition *red-external-aggr* ::
$'m\ prog \Rightarrow 'thread\text{-}id \Rightarrow 'addr \Rightarrow mname \Rightarrow 'addr\ val\ list \Rightarrow 'heap \Rightarrow$
$('addr,\ 'thread\text{-}id,\ 'heap)\ external\text{-}thread\text{-}action \times 'addr\ extCallRet \times 'heap)\ set$
where
red-external-aggr P t a M vs h =
(*if M* = *wait then*
let ad-t = *thread-id2addr t*
in {(\{|*Unlock→a*, *Lock→a*, *IsInterrupted t True*, *ClearInterrupt t*, *ObsInterrupted t*|\}, *RetEXC InterruptedException*, *h*),
(\{|*Suspend a*, *Unlock→a*, *Lock→a*, *ReleaseAcquire→a*, *IsInterrupted t False*, *SyncUnlock a*|\}, *RetStaySame*, *h*),
(\{|*UnlockFail→a*|\}, *RetEXC IllegalMonitorState*, *h*),
(\{|*Notified*|\}, *RetVal Unit*, *h*),
(\{|*WokenUp*, *ClearInterrupt t*, *ObsInterrupted t*|\}, *RetEXC InterruptedException*, *h*)} ∪
(*if spurious-wakeups then* {(\{|*Unlock→a*, *Lock→a*, *ReleaseAcquire→a*, *IsInterrupted t False*, *SyncUnlock a*|\}, *RetVal Unit*, *h*)} *else* {})
else if M = *notify then* {(\{|*Notify a*, *Unlock→a*, *Lock→a*|\}, *RetVal Unit*, *h*),
(\{|*UnlockFail→a*|\}, *RetEXC IllegalMonitorState*, *h*)}
else if M = *notifyAll then* {(\{|*NotifyAll a*, *Unlock→a*, *Lock→a* |\}, *RetVal Unit*, *h*),
(\{|*UnlockFail→a*|\}, *RetEXC IllegalMonitorState*, *h*)}
else if M = *clone then*
{((*K$* [], [], [], [], [], *obs*), *RetVal* (*Addr a'*), *h'*)|*obs a' h'*. *heap-clone P h a h'* ⌊(*obs*, *a'*)⌋}
∪ {(\{|\|\}, *RetEXC OutOfMemory*, *h'*)|*h'*. *heap-clone P h a h' None*}
else if M = *hashcode then* {(\{|\|\}, *RetVal* (*Intg* (*word-of-int* (*hash-addr a*))), *h*)}
else if M = *print then* {(\{|*ExternalCall a M vs Unit*|\}, *RetVal Unit*, *h*)}
else if M = *currentThread then* {(\{|\|\}, *RetVal* (*Addr* (*thread-id2addr t*)), *h*)}
else if M = *interrupted then* {(\{|*IsInterrupted t True*, *ClearInterrupt t*, *ObsInterrupted t*|\}, *RetVal* (*Bool True*), *h*),
(\{|*IsInterrupted t False*|\}, *RetVal* (*Bool False*), *h*)}
else if M = *yield then* {(\{|*Yield*|\}, *RetVal Unit*, *h*)}
else
let hT = *the* (*typeof-addr h a*)
in if P ⊢ *ty-of-htype hT* ≤ *Class Thread then*
let t-a = *addr2thread-id a*
in if M = *start then*
{(\{|*NewThread t-a* (*the-Class* (*ty-of-htype hT*), *run*, *a*) *h*, *ThreadStart t-a*|\}, *RetVal Unit*, *h*),
(\{|*ThreadExists t-a True*|\}, *RetEXC IllegalThreadState*, *h*)}
else if M = *join then*
{(\{|*Join t-a*, *IsInterrupted t False*, *ThreadJoin t-a*|\}, *RetVal Unit*, *h*),

```
            ({|IsInterrupted t True, ClearInterrupt t, ObsInterrupted t|}, RetEXC InterruptedException,
h)}
          else if M = interrupt then
            {({|ThreadExists t-a True, WakeUp t-a, Interrupt t-a, ObsInterrupt t-a|}, RetVal Unit, h),
             ({|ThreadExists t-a False|}, RetVal Unit, h)}
          else if M = isInterrupted then
            {({|IsInterrupted t-a False|}, RetVal (Bool False), h),
             ({|IsInterrupted t-a True, ObsInterrupted t-a|}, RetVal (Bool True), h)}
        else {({||}, undefined)}
      else {({||}, undefined)})
```

lemma *red-external-imp-red-external-aggr*:
  $P,t \vdash \langle a{\cdot}M(vs),\ h\rangle\ -ta\rightarrow ext\ \langle va,\ h'\rangle \Longrightarrow (ta,\ va,\ h') \in$ *red-external-aggr P t a M vs h*
unfolding *red-external-aggr-def*
by(*auto elim*!: *red-external.cases split del*: *if-split simp add*: *split-beta*)

end

context *heap* **begin**

lemma *hext-heap-copy-loc*:
  heap-copy-loc a a' al h obs h' $\Longrightarrow$ $h \trianglelefteq h'$
by(*blast elim*: *heap-copy-loc.cases dest*: *hext-heap-ops*)

lemma *hext-heap-copies*:
  assumes *heap-copies a a' als h obs h'*
  shows $h \trianglelefteq h'$
using *assms* **by** *induct*(*blast intro*: *hext-heap-copy-loc hext-trans*)+

lemma *hext-heap-clone*:
  assumes *heap-clone P h a h' res*
  shows $h \trianglelefteq h'$
using *assms* **by**(*blast elim*: *heap-clone.cases dest*: *hext-heap-ops hext-heap-copies intro*: *hext-trans*)

theorem *red-external-hext*:
  assumes $P,t \vdash \langle a{\cdot}M(vs),\ h\rangle\ -ta\rightarrow ext\ \langle va,\ h'\rangle$
  shows *hext h h'*
using *assms*
by(*cases*)(*blast intro*: *hext-heap-ops hext-heap-clone*)+

lemma *red-external-preserves-tconf*:
  $[\![\ P,t \vdash \langle a{\cdot}M(vs),\ h\rangle\ -ta\rightarrow ext\ \langle va,\ h'\rangle;\ P,h \vdash t'\ \surd t\ ]\!] \Longrightarrow P,h' \vdash t'\ \surd t$
by(*drule red-external-hext*)(*rule tconf-hext-mono*)

end

context *heap-conf* **begin**

lemma *typeof-addr-heap-clone*:
  assumes *heap-clone P h a h'* $\lfloor(obs,\ a')\rfloor$
  and *hconf h*
  shows *typeof-addr h' a' = typeof-addr h a*
using *assms*
by *cases* (*auto dest*!: *allocate-SomeD hext-heap-copies dest*: *typeof-addr-hext-mono typeof-addr-is-type*

is-type-ArrayD)

end

context *heap-base* **begin**

lemma *red-ext-new-thread-heap*:
⟦ *P*,*t* ⊢ ⟨*a*•*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h'*⟩; *NewThread t' ex h''* ∈ *set* {|*ta*|}$_t$ ⟧ ⟹ *h''* = *h'*
by(*auto elim*: *red-external.cases simp add*: *ta-upd-simps*)

lemma *red-ext-aggr-new-thread-heap*:
⟦ (*ta*, *va*, *h'*) ∈ *red-external-aggr P t a M vs h*; *NewThread t' ex h''* ∈ *set* {|*ta*|}$_t$ ⟧ ⟹ *h''* = *h'*
by(*auto simp add*: *red-external-aggr-def is-native.simps split-beta ta-upd-simps split*: *if-split-asm*)

end

context *addr-conv* **begin**

lemma *red-external-new-thread-exists-thread-object*:
⟦ *P*,*t* ⊢ ⟨*a*•*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h'*⟩; *NewThread t' x h''* ∈ *set* {|*ta*|}$_t$ ⟧
⟹ ∃ *C*. *typeof-addr h'* (*thread-id2addr t'*) = ⌊*Class-type C*⌋ ∧ *P* ⊢ *C* ≼* *Thread*
by(*auto elim*!: *red-external.cases dest*!: *Array-widen simp add*: *ta-upd-simps*)

lemma *red-external-aggr-new-thread-exists-thread-object*:
⟦ (*ta*, *va*, *h'*) ∈ *red-external-aggr P t a M vs h*; *typeof-addr h a* ≠ *None*;
NewThread t' x h'' ∈ *set* {|*ta*|}$_t$ ⟧
⟹ ∃ *C*. *typeof-addr h'* (*thread-id2addr t'*) = ⌊*Class-type C*⌋ ∧ *P* ⊢ *C* ≼* *Thread*
by(*auto simp add*: *red-external-aggr-def is-native.simps split-beta ta-upd-simps widen-Class split*: *if-split-asm dest*!: *Array-widen*)

end

context *heap* **begin**

lemma *red-external-aggr-hext*:
⟦ (*ta*, *va*, *h'*) ∈ *red-external-aggr P t a M vs h*; *is-native P* (*the* (*typeof-addr h a*)) *M* ⟧ ⟹ *h* ⊴ *h'*
apply(*auto simp add*: *red-external-aggr-def split-beta is-native.simps elim*!: *external-WT-defs-cases hext-heap-clone split*: *if-split-asm*)
apply(*auto elim*!: *external-WT-defs.cases dest*!: *sees-method-decl-above intro*: *widen-trans simp add*: *class-type-of-eq split*: *htype.split-asm*)
done

lemma *red-external-aggr-preserves-tconf*:
⟦ (*ta*, *va*, *h'*) ∈ *red-external-aggr P t a M vs h*; *is-native P* (*the* (*typeof-addr h a*)) *M*; *P*,*h* ⊢ *t'* √*t* ⟧
⟹ *P*,*h'* ⊢ *t'* √*t*
by(*blast dest*: *red-external-aggr-hext intro*: *tconf-hext-mono*)

end

context *heap-base* **begin**

lemma *red-external-Wakeup-no-Join-no-Lock-no-Interrupt*:
⟦ *P*,*t* ⊢ ⟨*a*•*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h'*⟩; *Notified* ∈ *set* {|*ta*|}$_w$ ∨ *WokenUp* ∈ *set* {|*ta*|}$_w$ ⟧ ⟹
collect-locks {|*ta*|}$_l$ = {} ∧ *collect-cond-actions* {|*ta*|}$_c$ = {} ∧ *collect-interrupts* {|*ta*|}$_i$ = {}

by(*auto elim*!: *red-external.cases simp add*: *ta-upd-simps collect-locks-def collect-interrupts-def*)

lemma *red-external-aggr-Wakeup-no-Join*:
⟦ (*ta*, *va*, *h'*) ∈ *red-external-aggr P t a M vs h*;
Notified ∈ *set* {|*ta*|}$_w$ ∨ *WokenUp* ∈ *set* {|*ta*|}$_w$ ⟧
⟹ *collect-locks* {|*ta*|}$_l$ = {} ∧ *collect-cond-actions* {|*ta*|}$_c$ = {} ∧ *collect-interrupts* {|*ta*|}$_i$ = {}
by(*auto simp add*: *red-external-aggr-def split-beta ta-upd-simps collect-locks-def collect-interrupts-def split*: *if-split-asm*)

lemma *red-external-Suspend-StaySame*:
⟦ *P*,*t* ⊢ ⟨*a*·*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h'*⟩; *Suspend w* ∈ *set* {|*ta*|}$_w$ ⟧ ⟹ *va* = *RetStaySame*
by(*auto elim*!: *red-external.cases simp add*: *ta-upd-simps*)

lemma *red-external-aggr-Suspend-StaySame*:
⟦ (*ta*, *va*, *h'*) ∈ *red-external-aggr P t a M vs h*; *Suspend w* ∈ *set* {|*ta*|}$_w$ ⟧ ⟹ *va* = *RetStaySame*
by(*auto simp add*: *red-external-aggr-def split-beta ta-upd-simps split*: *if-split-asm*)

lemma *red-external-Suspend-waitD*:
⟦ *P*,*t* ⊢ ⟨*a*·*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h'*⟩; *Suspend w* ∈ *set* {|*ta*|}$_w$ ⟧ ⟹ *M* = *wait*
by(*auto elim*!: *red-external.cases simp add*: *ta-upd-simps*)

lemma *red-external-aggr-Suspend-waitD*:
⟦ (*ta*, *va*, *h'*) ∈ *red-external-aggr P t a M vs h*; *Suspend w* ∈ *set* {|*ta*|}$_w$ ⟧ ⟹ *M* = *wait*
by(*auto simp add*: *red-external-aggr-def split-beta ta-upd-simps split*: *if-split-asm*)

lemma *red-external-new-thread-sub-thread*:
⟦ *P*,*t* ⊢ ⟨*a*·*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h'*⟩; *NewThread t'* (*C*, *M'*, *a'*) *h''* ∈ *set* {|*ta*|}$_t$ ⟧
⟹ *typeof-addr h' a'* = ⌊*Class-type C*⌋ ∧ *P* ⊢ *C* ≼* *Thread* ∧ *M'* = *run*
by(*auto elim*!: *red-external.cases simp add*: *widen-Class ta-upd-simps*)

lemma *red-external-aggr-new-thread-sub-thread*:
⟦ (*ta*, *va*, *h'*) ∈ *red-external-aggr P t a M vs h*; *typeof-addr h a* ≠ *None*;
NewThread t' (*C*, *M'*, *a'*) *h''* ∈ *set* {|*ta*|}$_t$ ⟧
⟹ *typeof-addr h' a'* = ⌊*Class-type C*⌋ ∧ *P* ⊢ *C* ≼* *Thread* ∧ *M'* = *run*
by(*auto simp add*: *red-external-aggr-def split-beta ta-upd-simps widen-Class split*: *if-split-asm dest*!: *Array-widen*)

lemma *heap-copy-loc-length*:
assumes *heap-copy-loc a a' al h obs h'*
shows *length obs* = *2*
using *assms* **by**(*cases*) *simp*

lemma *heap-copies-length*:
assumes *heap-copies a a' als h obs h'*
shows *length obs* = *2* ∗ *length als*
using *assms* **by**(*induct*)(*auto dest*!: *heap-copy-loc-length*)

end

3.11.4 τ-moves

inductive τ*external-defs* :: *cname* ⇒ *mname* ⇒ *bool*
where

τexternal-defs Object hashcode
| *τexternal-defs Object currentThread*

definition *τexternal* :: *'m prog ⇒ htype ⇒ mname ⇒ bool*
where *τexternal P hT M ⟷ (∃ Ts Tr D. P ⊢ class-type-of hT sees M:Ts→Tr = Native in D ∧ τexternal-defs D M)*

context *heap-base* **begin**

definition *τexternal'* :: *'m prog ⇒ 'heap ⇒ 'addr ⇒ mname ⇒ bool*
where *τexternal' P h a M ⟷ (∃ hT. typeof-addr h a = Some hT ∧ τexternal P hT M)*

lemma *τexternal'-red-external-heap-unchanged*:
⟦ P,t ⊢ ⟨a·M(vs), h⟩ −ta→ext ⟨va, h'⟩; τexternal' P h a M ⟧ ⟹ h' = h
by(*auto elim*!: *red-external.cases τexternal-defs.cases simp add*: *τexternal-def τexternal'-def*)

lemma *τexternal'-red-external-aggr-heap-unchanged*:
⟦ (ta, va, h') ∈ red-external-aggr P t a M vs h; τexternal' P h a M ⟧ ⟹ h' = h
by(*auto elim*!: *τexternal-defs.cases simp add*: *τexternal-def τexternal'-def red-external-aggr-def*)

lemma *τexternal'-red-external-TA-empty*:
⟦ P,t ⊢ ⟨a·M(vs), h⟩ −ta→ext ⟨va, h'⟩; τexternal' P h a M ⟧ ⟹ ta = ε
by(*auto elim*!: *red-external.cases τexternal-defs.cases simp add*: *τexternal-def τexternal'-def*)

lemma *τexternal'-red-external-aggr-TA-empty*:
⟦ (ta, va, h') ∈ red-external-aggr P t a M vs h; τexternal' P h a M ⟧ ⟹ ta = ε
by(*auto elim*!: *τexternal-defs.cases simp add*: *τexternal-def τexternal'-def red-external-aggr-def*)

lemma *red-external-new-thread-addr-conf*:
⟦ P,t ⊢ ⟨a·M(vs),h⟩ −ta→ext ⟨va,h'⟩; NewThread t (C, M, a') h'' ∈ set {|ta|}_t ⟧
⟹ P,h' ⊢ Addr a :≤ Class Thread
by(*auto elim*!: *red-external.cases simp add*: *conf-def ta-upd-simps*)

lemma *τexternal-red-external-aggr-heap-unchanged*:
⟦ (ta, va, h') ∈ red-external-aggr P t a M vs h; τexternal P (the (typeof-addr h a)) M ⟧ ⟹ h' = h
by(*auto elim*!: *τexternal-defs.cases simp add*: *τexternal-def red-external-aggr-def*)

lemma *τexternal-red-external-aggr-TA-empty*:
⟦ (ta, va, h') ∈ red-external-aggr P t a M vs h; τexternal P (the (typeof-addr h a)) M ⟧ ⟹ ta = ε
by(*auto elim*!: *τexternal-defs.cases simp add*: *τexternal-def red-external-aggr-def*)

end

3.11.5 Code generation

code-pred
(*modes*:
i ⇒ i ⇒ i ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ o ⇒ bool,
i ⇒ i ⇒ o ⇒ o ⇒ bool,
o ⇒ i ⇒ o ⇒ o ⇒ bool)
external-WT-defs
.

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$)
[*inductify*, *skip-proof*]
is-native
.

declare *heap-base.heap-copy-loc.intros*[*code-pred-intro*]

code-pred
(*modes*: $(i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool) \Rightarrow (i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool) \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool$)
heap-base.heap-copy-loc
proof −
case *heap-copy-loc*
from *heap-copy-loc.prems* **show** *thesis*
by(*rule heap-base.heap-copy-loc.cases*)(*rule heap-copy-loc.that*[*OF refl refl refl refl refl refl*])
qed

declare *heap-base.heap-copies.intros* [*code-pred-intro*]

code-pred
(*modes*: $(i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool) => (i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool) \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool$)
heap-base.heap-copies
proof −
case *heap-copies*
from *heap-copies.prems* **show** *thesis*
by(*rule heap-base.heap-copies.cases*)(*erule* (*3*) *heap-copies.that*[*OF refl refl refl refl*]|*assumption*)+
qed

declare *heap-base.heap-clone.intros* [*folded Predicate-Compile.contains-def*, *code-pred-intro*]

code-pred
(*modes*: $i \Rightarrow i \Rightarrow (i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool) \Rightarrow (i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool) \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow bool$)
heap-base.heap-clone
proof −
case *heap-clone*
from *heap-clone.prems* **show** *thesis*
by(*rule heap-base.heap-clone.cases*[*folded Predicate-Compile.contains-def*])(*erule* (*3*) *heap-clone.that*[*OF refl refl refl refl refl refl refl*]|*assumption*)+
qed

code_pred in Isabelle2012 cannot handle boolean parameters as premises properly, so this replacement rule explicitly tests for *True*

lemma (**in** *heap-base*) *RedWaitSpurious-Code*:
spurious-wakeups = *True* ⟹
P,*t* ⊢ ⟨*a*·*wait*([]),*h*⟩ −{|*Unlock*→*a*, *Lock*→*a*, *ReleaseAcquire*→*a*, *IsInterrupted t False*, *SyncUnlock a*|}→*ext* ⟨*RetVal Unit*,*h*⟩
by(*rule RedWaitSpurious*) *simp*

lemmas [*code-pred-intro*] =
heap-base.RedNewThread heap-base.RedNewThreadFail
heap-base.RedJoin heap-base.RedJoinInterrupt

heap-base.RedInterrupt heap-base.RedInterruptInexist heap-base.RedIsInterruptedTrue heap-base.RedIsInterruptedFalse
heap-base.RedWaitInterrupt heap-base.RedWait heap-base.RedWaitFail heap-base.RedWaitNotified heap-base.RedWaitInterrup
declare *heap-base.RedWaitSpurious-Code* [*code-pred-intro RedWaitSpurious*]
lemmas [*code-pred-intro*] =
heap-base.RedNotify heap-base.RedNotifyFail heap-base.RedNotifyAll heap-base.RedNotifyAllFail
heap-base.RedClone heap-base.RedCloneFail
heap-base.RedHashcode heap-base.RedPrint heap-base.RedCurrentThread
heap-base.RedInterruptedTrue heap-base.RedInterruptedFalse
heap-base.RedYield

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow (i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool) \Rightarrow (i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool) \Rightarrow i$
$\Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$)
heap-base.red-external
proof −
case *red-external*
from *red-external.prems* **show** *?thesis*
apply(*rule heap-base.red-external.cases*)
apply(*erule (4) red-external.that*[*OF refl refl refl refl refl refl refl refl refl refl refl refl*]|*assumption*|*erule eqTrueI*)+
done
qed

end

3.12 Generic Well-formedness of programs

theory *WellForm*
imports
SystemClasses
ExternalCall
begin

This theory defines global well-formedness conditions for programs but does not look inside method bodies. Hence it works for both Jinja and JVM programs. Well-typing of expressions is defined elsewhere (in theory *WellType*).

Because JinjaThreads does not have method overloading, its policy for method overriding is the classical one: *covariant in the result type but contravariant in the argument types.* This means the result type of the overriding method becomes more specific, the argument types become more general.

type-synonym *'m wf-mdecl-test* = *'m prog* ⇒ *cname* ⇒ *'m mdecl* ⇒ *bool*

definition *wf-fdecl* :: *'m prog* ⇒ *fdecl* ⇒ *bool*
where *wf-fdecl P* ≡ λ(*F*,*T*,*fm*). *is-type P T*

definition *wf-mdecl* :: *'m wf-mdecl-test* ⇒ *'m prog* ⇒ *cname* ⇒ *'m mdecl'* ⇒ *bool* **where**
wf-mdecl wf-md P C ≡
λ(*M*,*Ts*,*T*,*m*). (∀ *T*∈*set Ts*. *is-type P T*) ∧ *is-type P T* ∧
(*case m of Native* ⇒ *C*·*M*(*Ts*) :: *T* | ⌊*mb*⌋ ⇒ *wf-md P C* (*M*,*Ts*,*T*,*mb*))

fun *wf-overriding* :: *'m prog* ⇒ *cname* ⇒ *'m mdecl'* ⇒ *bool*
where
wf-overriding P D (*M*,*Ts*,*T*,*m*) =

$(\forall\, D'\ Ts'\ T'\ m'.\ P \vdash D\ sees\ M{:}Ts' \to T' = m'\ in\ D' \longrightarrow P \vdash Ts'\ [\leq]\ Ts \wedge P \vdash T \leq T')$

definition *wf-cdecl* :: *'m wf-mdecl-test* ⇒ *'m prog* ⇒ *'m cdecl* ⇒ *bool*
where
wf-cdecl wf-md P ≡ λ(*C*,(*D*,*fs*,*ms*)).
(∀ *f*∈*set fs*. *wf-fdecl P f*) ∧ *distinct-fst fs* ∧
(∀ *m*∈*set ms*. *wf-mdecl wf-md P C m*) ∧
distinct-fst ms ∧
(*C* ≠ *Object* ⟶
is-class P D ∧ ¬ *P* ⊢ *D* ≼* *C* ∧
(∀ *m*∈*set ms*. *wf-overriding P D m*)) ∧
(*C* = *Thread* ⟶ (∃ *m*. (*run*, [], *Void*, *m*) ∈ *set ms*))

definition *wf-prog* :: *'m wf-mdecl-test* ⇒ *'m prog* ⇒ *bool*
where
wf-prog wf-md P ⟷ *wf-syscls P* ∧ *distinct-fst* (*classes P*) ∧ (∀ *c* ∈ *set* (*classes P*). *wf-cdecl wf-md P c*)

lemma *wf-prog-def2*:
wf-prog wf-md P ⟷ *wf-syscls P* ∧ (∀ *C rest*. *class P C* = ⌊*rest*⌋ ⟶ *wf-cdecl wf-md P* (*C*, *rest*)) ∧ *distinct-fst* (*classes P*)
by(*cases P*)(*auto simp add*: *wf-prog-def dest*: *map-of-SomeD map-of-SomeI*)

3.12.1 Well-formedness lemmas

lemma *wf-prog-wf-syscls*: *wf-prog wf-md P* ⟹ *wf-syscls P*
by(*simp add*: *wf-prog-def*)

lemma *class-wf*:
⟦*class P C* = *Some c*; *wf-prog wf-md P*⟧ ⟹ *wf-cdecl wf-md P* (*C*,*c*)
by (*cases P*) (*fastforce dest*: *map-of-SomeD simp add*: *wf-prog-def*)

lemma [*simp*]:
assumes *wf-prog wf-md P*
shows *class-Object*: ∃ *C fs ms*. *class P Object* = *Some* (*C*,*fs*,*ms*)
and *class-Thread*: ∃ *C fs ms*. *class P Thread* = *Some* (*C*,*fs*,*ms*)
using *wf-prog-wf-syscls*[*OF assms*]
by(*rule wf-syscls-class-Object wf-syscls-class-Thread*)+

lemma [*simp*]:
assumes *wf-prog wf-md P*
shows *is-class-Object*: *is-class P Object*
and *is-class-Thread*: *is-class P Thread*
using *wf-prog-wf-syscls*[*OF assms*] **by** *simp-all*

lemma *xcpt-subcls-Throwable*:
⟦ *C* ∈ *sys-xcpts*; *wf-prog wf-md P* ⟧ ⟹ *P* ⊢ *C* ≼* *Throwable*
by(*simp add*: *wf-prog-wf-syscls wf-syscls-xcpt-subcls-Throwable*)

lemma *is-class-Throwable*:
wf-prog wf-md P ⟹ *is-class P Throwable*
by(*rule wf-prog-wf-syscls wf-syscls-is-class-Throwable*)+

lemma *is-class-sub-Throwable*:

⟦ *wf-prog wf-md P*; $P \vdash C \preceq^* Throwable$ ⟧ ⟹ *is-class P C*
by(*rule wf-syscls-is-class-sub-Throwable*[*OF wf-prog-wf-syscls*])

lemma *is-class-xcpt*:
⟦ $C \in$ *sys-xcpts*; *wf-prog wf-md P* ⟧ ⟹ *is-class P C*
by(*rule wf-syscls-is-class-xcpt*[*OF - wf-prog-wf-syscls*])

context *heap-base* **begin**
lemma *wf-preallocatedE*:
assumes *wf-prog wf-md P*
and *preallocated h*
and $C \in$ *sys-xcpts*
obtains *typeof-addr h* (*addr-of-sys-xcpt C*) = ⌊*Class-type C*⌋ $P \vdash C \preceq^* Throwable$
proof −
from ‹*preallocated h*› ‹$C \in$ *sys-xcpts*› **have** *typeof-addr h* (*addr-of-sys-xcpt C*) = ⌊*Class-type C*⌋
by(*rule typeof-addr-sys-xcp*)
moreover from ‹$C \in$ *sys-xcpts*› ‹*wf-prog wf-md P*› **have** $P \vdash C \preceq^* Throwable$ **by**(*rule xcpt-subcls-Throwable*)
ultimately show *thesis* **by**(*rule that*)
qed

lemma *wf-preallocatedD*:
assumes *wf-prog wf-md P*
and *preallocated h*
and $C \in$ *sys-xcpts*
shows *typeof-addr h* (*addr-of-sys-xcpt C*) = ⌊*Class-type C*⌋ ∧ $P \vdash C \preceq^* Throwable$
using *assms*
by(*rule wf-preallocatedE*) *blast*

end

lemma (**in** *heap-conf*) *hconf-start-heap*:
wf-prog wf-md P ⟹ *hconf start-heap*
unfolding *start-heap-def start-heap-data-def initialization-list-def sys-xcpts-list-def*
using *hconf-empty*
by −(*simp add*: *create-initial-object-simps del*: *hconf-empty*, *clarsimp split*: *prod.split elim*!: *not-empty-pairE*
simp del: *hconf-empty*, *drule* (*1*) *allocate-Eps*, *drule* (*1*) *hconf-allocate-mono*, *simp add*: *is-class-xcpt*)+

lemma *subcls1-wfD*:
⟦ $P \vdash C \prec^1 D$; *wf-prog wf-md P* ⟧ ⟹ $D \neq C \wedge \neg$ (*subcls1 P*)$^{++}$ *D C*
apply(*frule tranclp.r-into-trancl*[**where** *r*=*subcls1 P*])
apply(*drule subcls1D*)
apply(*clarify*)
apply(*drule* (*1*) *class-wf*)
apply(*unfold wf-cdecl-def*)
apply(*rule conjI*)
apply(*force*)
apply(*unfold reflclp-tranclp*[*symmetric*, **where** *r*=*subcls1 P*])
apply(*blast*)
done

lemma *wf-cdecl-supD*:
⟦*wf-cdecl wf-md P* (*C*,*D*,*r*); $C \neq Object$⟧ ⟹ *is-class P D*

lemma *subcls-asym*:

⟦ *wf-prog wf-md P*; (*subcls1 P*)$^{++}$ *C D* ⟧ ⟹ ¬ (*subcls1 P*)$^{++}$ *D C*

lemma *subcls-irrefl*:
⟦ *wf-prog wf-md P*; (*subcls1 P*)$^{++}$ *C D*⟧ ⟹ *C* ≠ *D*
lemma *acyclicP-def*:
acyclicP r ⟷ (∀ *x*. ¬ *r*^++ *x x*)
unfolding *acyclic-def trancl-def*
by(*auto*)

lemma *acyclic-subcls1*:
wf-prog wf-md P ⟹ *acyclicP* (*subcls1 P*)
by(*unfold acyclicP-def*)(*fast dest*: *subcls-irrefl*)

lemma *finite-conversep*: *finite* {(*x*, *y*). *r*$^{-1-1}$ *x y*} = *finite* {(*x*, *y*). *r x y*}
by(*subst finite-converse*[*unfolded converse-unfold*, *symmetric*]) *simp*

lemma *acyclicP-wf-subcls1*:
acyclicP (*subcls1 P*) ⟹ *wfP* ((*subcls1 P*)$^{-1-1}$)
unfolding *wfp-def*
by(*rule finite-acyclic-wf*)(*simp-all only*: *finite-conversep finite-subcls1 acyclicP-converse*)

lemma *wf-subcls1*:
wf-prog wf-md P ⟹ *wfP* ((*subcls1 P*)$^{-1-1}$)
by(*rule acyclicP-wf-subcls1*)(*rule acyclic-subcls1*)

lemma *single-valued-subcls1*:
wf-prog wf-md G ⟹ *single-valuedp* (*subcls1 G*)

lemma *subcls-induct*:
⟦ *wf-prog wf-md P*; ⋀*C*. ∀ *D*. (*subcls1 P*)$^{++}$ *C D* ⟶ *Q D* ⟹ *Q C* ⟧ ⟹ *Q C*

lemma *subcls1-induct-aux*:
⟦ *is-class P C*; *wf-prog wf-md P*; *Q Object*;
⋀*C D fs ms*.
⟦ *C* ≠ *Object*; *is-class P C*; *class P C* = *Some* (*D*,*fs*,*ms*) ∧
wf-cdecl wf-md P (*C*,*D*,*fs*,*ms*) ∧ *P* ⊢ *C* ≺^{1} *D* ∧ *is-class P D* ∧ *Q D*⟧ ⟹ *Q C* ⟧
⟹ *Q C*
lemma *subcls1-induct* [*consumes 2*, *case-names Object Subcls*]:
⟦ *wf-prog wf-md P*; *is-class P C*; *Q Object*;
⋀*C D*. ⟦*C* ≠ *Object*; *P* ⊢ *C* ≺^{1} *D*; *is-class P D*; *Q D*⟧ ⟹ *Q C* ⟧
⟹ *Q C*

lemma *subcls-C-Object*:
⟦ *is-class P C*; *wf-prog wf-md P* ⟧ ⟹ *P* ⊢ *C* ≼* *Object*
lemma *converse-subcls-is-class*:
assumes *wf*: *wf-prog wf-md P*
shows ⟦ *P* ⊢ *C* ≼* *D*; *is-class P C* ⟧ ⟹ *is-class P D*
proof(*induct rule*: *rtranclp-induct*)
assume *is-class P C*
thus *is-class P C* **.**
next
fix *D E*
assume *PDE*: *P* ⊢ *D* ≺^{1} *E*
and *IH*: *is-class P C* ⟹ *is-class P D*

```
    and iPC: is-class P C
  have is-class P D by (rule IH[OF iPC])
  with PDE obtain fsD MsD where classD: class P D = ⌊(E, fsD, MsD)⌋
    by(auto simp add: is-class-def elim!: subcls1.cases)
  thus is-class P E using wf PDE
    by(auto elim!: subcls1.cases dest: class-wf simp: wf-cdecl-def)
qed

lemma is-class-is-subcls:
  wf-prog m P ⟹ is-class P C = P ⊢ C ≼* Object
lemma subcls-antisym:
  ⟦wf-prog m P; P ⊢ C ≼* D; P ⊢ D ≼* C⟧ ⟹ C = D
apply(drule acyclic-subcls1)
apply(drule acyclic-impl-antisym-rtrancl)
apply(drule antisymD)
apply(unfold Transitive-Closure.rtrancl-def)
apply(auto)
done

lemma is-type-pTs:
  ⟦ wf-prog wf-md P; class P C = ⌊(S,fs,ms)⌋; (M,Ts,T,m) ∈ set ms ⟧ ⟹ set Ts ⊆ types P
by(fastforce dest: class-wf simp add: wf-cdecl-def wf-mdecl-def)

lemma widen-asym-1:
  assumes wfP: wf-prog wf-md P
  shows P ⊢ C ≤ D ⟹ C = D ∨ ¬ (P ⊢ D ≤ C)
proof (erule widen.induct)
  fix T
  show T = T ∨ ¬ (P ⊢ T ≤ T) by simp
next
  fix C D
  assume CscD: P ⊢ C ≼* D
  then have CpscD: C = D ∨ (C ≠ D ∧ (subcls1 P)++ C D) by (simp add: rtranclpD)
  { assume P ⊢ D ≼* C
    then have DpscC: D = C ∨ (D ≠ C ∧ (subcls1 P)++ D C) by (simp add: rtranclpD)
    { assume (subcls1 P)++ D C
      with wfP have CnscD: ¬ (subcls1 P)++ C D by (rule subcls-asym)
      with CpscD have C = D by simp
    }
    with DpscC have C = D by blast
  }
  hence Class C = Class D ∨ ¬ (P ⊢ D ≼* C) by blast
  thus Class C = Class D ∨ ¬ P ⊢ Class D ≤ Class C by simp
next
  fix C
  show NT = Class C ∨ ¬ P ⊢ Class C ≤ NT by simp
next
  fix A
  { assume P ⊢ A⌊⌉ ≤ NT
    hence A⌊⌉ = NT by fastforce
    hence False by simp }
  hence ¬ (P ⊢ A⌊⌉ ≤ NT) by blast
  thus NT = A⌊⌉ ∨ ¬ P ⊢ A⌊⌉ ≤ NT by simp
next
```

fix A
show $A\lfloor\rceil = \mathit{Class\ Object} \vee \neg\ P \vdash \mathit{Class\ Object} \leq A\lfloor\rceil$
by(*auto dest*: *Object-widen*)
next
fix $A\ B$
assume *AsU*: $P \vdash A \leq B$ **and** *BnpscA*: $A = B \vee \neg\ P \vdash B \leq A$
{ assume $P \vdash B\lfloor\rceil \leq A\lfloor\rceil$
hence $P \vdash B \leq A$ **by** (*auto dest*: *Array-Array-widen*)
with *BnpscA* **have** $A = B$ **by** *blast*
hence $A\lfloor\rceil = B\lfloor\rceil$ **by** *simp* **}**
thus $A\lfloor\rceil = B\lfloor\rceil \vee \neg\ P \vdash B\lfloor\rceil \leq A\lfloor\rceil$ **by** *blast*
qed

lemma *widen-asym*: $[\![\ \mathit{wf\text{-}prog\ wf\text{-}md}\ P;\ P \vdash C \leq D;\ C \neq D\ ]\!] \Longrightarrow \neg\ (P \vdash D \leq C)$
proof −
assume *wfP*: *wf-prog wf-md P* **and** *CsD*: $P \vdash C \leq D$ **and** *CneqD*: $C \neq D$
from *wfP CsD* **have** $C = D \vee \neg\ (P \vdash D \leq C)$ **by** (*rule widen-asym-1*)
with *CneqD* **show** *?thesis* **by** *simp*
qed

lemma *widen-antisym*:
$[\![\ \mathit{wf\text{-}prog}\ m\ P;\ P \vdash T \leq U;\ P \vdash U \leq T\ ]\!] \Longrightarrow T = U$
by(*auto dest*: *widen-asym*)

lemma *widen-C-Object*: $[\![\ \mathit{wf\text{-}prog\ wf\text{-}md}\ P;\ \mathit{is\text{-}class}\ P\ C\ ]\!] \Longrightarrow P \vdash \mathit{Class}\ C \leq \mathit{Class\ Object}$
by(*simp add*: *subcls-C-Object*)

lemma *is-refType-widen-Object*:
assumes *wfP*: *wf-prog wfmc P*
shows $[\![\ \mathit{is\text{-}type}\ P\ A;\ \mathit{is\text{-}refT}\ A\ ]\!] \Longrightarrow P \vdash A \leq \mathit{Class\ Object}$
by(*induct A*)(*auto elim*: *refTE intro*: *subcls-C-Object*[*OF* - *wfP*] *widen-array-object*)

lemma *is-lub-unique*:
assumes *wf*: *wf-prog wf-md P*
shows $[\![\ P \vdash \mathit{lub}(U,\ V) = T;\ P \vdash \mathit{lub}(U,\ V) = T'\ ]\!] \Longrightarrow T = T'$
by(*auto elim*!: *is-lub.cases intro*: *widen-antisym*[*OF wf*])

3.12.2 Well-formedness and method lookup

lemma *sees-wf-mdecl*:
$[\![\ \mathit{wf\text{-}prog\ wf\text{-}md}\ P;\ P \vdash C\ \mathit{sees}\ M{:}Ts{\rightarrow}T = m\ \mathit{in}\ D\ ]\!] \Longrightarrow \mathit{wf\text{-}mdecl\ wf\text{-}md}\ P\ D\ (M,Ts,T,m)$

lemma *sees-method-mono* [*rule-format* (*no-asm*)]:
$[\![\ P \vdash C' \preceq^* C;\ \mathit{wf\text{-}prog\ wf\text{-}md}\ P\ ]\!] \Longrightarrow$
$\forall D\ Ts\ T\ m.\ P \vdash C\ \mathit{sees}\ M{:}Ts{\rightarrow}T = m\ \mathit{in}\ D \longrightarrow$
$(\exists D'\ Ts'\ T'\ m'.\ P \vdash C'\ \mathit{sees}\ M{:}Ts'{\rightarrow}T' = m'\ \mathit{in}\ D' \wedge P \vdash Ts\ [\leq]\ Ts' \wedge P \vdash T' \leq T)$
apply(*drule rtranclpD*)
apply(*erule disjE*)
apply(*fastforce intro*: *widen-refl widens-refl*)
apply(*erule conjE*)
apply(*erule tranclp-trans-induct*)
prefer *2*
apply(*clarify*)
apply(*drule spec*, *drule spec*, *drule spec*, *drule spec*, *erule* (*1*) *impE*)

apply *clarify*
apply(*fast elim*: *widen-trans widens-trans*)
apply(*clarify*)
apply(*drule subcls1D*)
apply(*clarify*)
apply(*clarsimp simp*:*Method-def*)
apply(*frule* (*2*) *sees-methods-rec*)
apply(*rule refl*)
apply(*case-tac map-of ms M*)
apply(*rule-tac x* = *D* **in** *exI*)
apply(*rule-tac x* = *Ts* **in** *exI*)
apply(*rule-tac x* = *T* **in** *exI*)
apply(*clarsimp simp add*: *widens-refl*)
apply(*rule-tac x* = *m* **in** *exI*)
apply(*fastforce simp add*:*map-add-def split*:*option.split*)
apply *clarsimp*
apply(*rename-tac Ts' T' m'*)
apply(*drule* (*1*) *class-wf*)
apply(*unfold wf-cdecl-def Method-def*)
apply(*frule map-of-SomeD*)
apply(*clarsimp*)
apply(*drule* (*1*) *bspec*)+
apply *clarsimp*
apply(*erule-tac x*=*D* **in** *allE*)
apply(*erule-tac x*=*Ts* **in** *allE*)
apply(*rotate-tac* −*1*)
apply(*erule-tac x*=*T* **in** *allE*)
apply(*fastforce simp*:*map-add-def Method-def split*:*option.split*)
done

lemma *sees-method-mono2*:
⟦ $P \vdash C' \preceq^* C$; *wf-prog wf-md P*;
$P \vdash C$ *sees* M:$Ts \to T = m$ *in* D; $P \vdash C'$ *sees* M:$Ts' \to T' = m'$ *in* D' ⟧
⟹ $P \vdash Ts\ [\leq]\ Ts' \wedge P \vdash T' \leq T$

lemma *mdecls-visible*:
assumes *wf*: *wf-prog wf-md P* **and** *class*: *is-class P C*
shows $\bigwedge D\ fs\ ms.$ *class P C* = *Some*(D,*fs*,*ms*)
⟹ $\exists Mm.$ $P \vdash C$ *sees-methods Mm* $\wedge$ $(\forall (M,Ts,T,m) \in$ *set ms*. *Mm M* = *Some*((*Ts*,*T*,*m*),*C*))
using *wf class*
proof (*induct rule*:*subcls1-induct*)
case *Object*
with *wf* **have** *distinct-fst ms*
by(*auto dest*: *class-wf simp add*: *wf-cdecl-def*)
with *Object* **show** *?case* **by**(*fastforce intro*!: *sees-methods-Object map-of-SomeI*)
next
case *Subcls*
with *wf* **have** *distinct-fst ms*
by(*auto dest*: *class-wf simp add*: *wf-cdecl-def*)
with *Subcls* **show** *?case*
by(*fastforce elim*:*sees-methods-rec dest*:*subcls1D map-of-SomeI*
simp:*is-class-def*)
qed

lemma *mdecl-visible*:
assumes *wf*: *wf-prog wf-md P* **and** *C*: *class P C* = ⌊(*S*,*fs*,*ms*)⌋ **and** *m*: (*M*,*Ts*,*T*,*m*) ∈ *set ms*
shows *P* ⊢ *C sees M*:*Ts*→*T* = *m in C*
proof −
from *C* **have** *is-class P C* **by**(*auto simp*:*is-class-def*)
with *assms* **show** *?thesis*
by(*bestsimp simp*:*Method-def dest*:*mdecls-visible*)
qed

lemma *sees-wf-native*:
⟦ *wf-prog wf-md P*; *P* ⊢ *C sees M*:*Ts*→*T*=*Native in D* ⟧ ⟹ *D*·*M*(*Ts*) :: *T*
apply(*drule* (*1*) *sees-wf-mdecl*)
apply(*simp add*: *wf-mdecl-def*)
done

lemma *Call-lemma*:
⟦ *P* ⊢ *C sees M*:*Ts*→*T* = *m in D*; *P* ⊢ *C*′ ≼* *C*; *wf-prog wf-md P* ⟧
⟹ ∃ *D*′ *Ts*′ *T*′ *m*′.
P ⊢ *C*′ *sees M*:*Ts*′→*T*′ = *m*′ *in D*′ ∧ *P* ⊢ *Ts* [≤] *Ts*′ ∧ *P* ⊢ *T*′ ≤ *T* ∧ *P* ⊢ *C*′ ≼* *D*′
∧ *is-type P T*′ ∧ (∀ *T*∈*set Ts*′. *is-type P T*) ∧ (*m*′ ≠ *Native* ⟶ *wf-md P D*′ (*M*,*Ts*′,*T*′,*the m*′))
apply(*frule* (*2*) *sees-method-mono*)
apply(*fastforce intro*:*sees-method-decl-above dest*:*sees-wf-mdecl*
simp: *wf-mdecl-def*)
done

lemma *sub-Thread-sees-run*:
assumes *wf*: *wf-prog wf-md P*
and *PCThread*: *P* ⊢ *C* ≼* *Thread*
shows ∃ *D mthd*. *P* ⊢ *C sees run*: []→*Void* = ⌊*mthd*⌋ *in D*
proof −
from *class-Thread*[*OF wf*] **obtain** *T*′ *fsT MsT*
where *classT*: *class P Thread* = ⌊(*T*′, *fsT*, *MsT*)⌋ **by** *blast*
hence *wfcThread*: *wf-cdecl wf-md P* (*Thread*, *T*′, *fsT*, *MsT*) **using** *wf* **by**(*rule class-wf*)
then obtain *mrunT* **where** *runThread*: (*run*, [], *Void*, *mrunT*) ∈ *set MsT*
by(*auto simp add*: *wf-cdecl-def*)
moreover have ∃ *MmT*. *P* ⊢ *Thread sees-methods MmT* ∧
(∀ (*M*,*Ts*,*T*,*m*) ∈ *set MsT*. *MmT M* = *Some*((*Ts*,*T*,*m*),*Thread*))
by(*rule mdecls-visible*[*OF wf is-class-Thread*[*OF wf*] *classT*])
then obtain *MmT* **where** *ThreadMmT*: *P* ⊢ *Thread sees-methods MmT*
and *MmT*: ∀ (*M*,*Ts*,*T*,*m*) ∈ *set MsT*. *MmT M* = *Some*((*Ts*,*T*,*m*),*Thread*)
by *blast*
ultimately obtain *mthd*
where *MmT run* = ⌊(([], *Void*, *mthd*), *Thread*)⌋
by(*fastforce*)
with *ThreadMmT* **have** *Tseesrun*: *P* ⊢ *Thread sees run*: []→*Void* = *mthd in Thread*
by(*auto simp add*: *Method-def*)
from *sees-method-mono*[*OF PCThread wf Tseesrun*]
obtain *D*′ *m*′ **where** *P* ⊢ *C sees run*: []→*Void* = *m*′ *in D*′ **by** *auto*
moreover have *m*′ ≠ *None*
proof
assume *m*′ = *None*
with *wf* ‹*P* ⊢ *C sees run*: []→*Void* = *m*′ *in D*′› **have** *D*′·*run*([]) :: *Void*

```
    by(auto intro: sees-wf-native)
   thus False by cases auto
  qed
  ultimately show ?thesis by auto
qed

lemma wf-prog-lift:
  assumes wf: wf-prog (λP C bd. A P C bd) P
  and rule:
  ⋀wf-md C M Ts C T m.
   ⟦ wf-prog wf-md P; P ⊢ C sees M:Ts→T = ⌊m⌋ in C; is-class P C; set Ts ⊆ types P; A P C
(M,Ts,T,m) ⟧
   ⟹ B P C (M,Ts,T,m)
  shows wf-prog (λP C bd. B P C bd) P
proof(cases P)
  case (Program P')
  thus ?thesis using wf
   apply(clarsimp simp add: wf-prog-def wf-cdecl-def)
   apply(drule (1) bspec)
   apply(rename-tac C D fs ms)
   apply(subgoal-tac is-class P C)
    prefer 2
    apply(simp add: is-class-def)
    apply(drule weak-map-of-SomeI)
    apply(simp add: Program)
   apply(clarsimp simp add: Program wf-mdecl-def split del: option.split)
   apply(drule (1) bspec)
   apply clarsimp
   apply(rule conjI)
    apply clarsimp
   apply clarsimp
   apply(frule (1) map-of-SomeI)
   apply(rule rule[OF wf, unfolded Program])
   apply(clarsimp simp add: is-class-def)
   apply(rule mdecl-visible[OF wf[unfolded Program]])
   apply(fastforce intro: is-type-pTs [OF wf, unfolded Program])+
   done
qed
```

3.12.3 Well-formedness and field lookup

```
lemma wf-Fields-Ex:
  ⟦ wf-prog wf-md P; is-class P C ⟧ ⟹ ∃FDTs. P ⊢ C has-fields FDTs

lemma has-fields-types:
  ⟦ P ⊢ C has-fields FDTs; (FD, T, fm) ∈ set FDTs; wf-prog wf-md P ⟧ ⟹ is-type P T

lemma sees-field-is-type:
  ⟦ P ⊢ C sees F:T (fm) in D; wf-prog wf-md P ⟧ ⟹ is-type P T
by(fastforce simp: sees-field-def
           elim: has-fields-types map-of-SomeD[OF map-of-remap-SomeD])

lemma wf-has-field-mono2:
  assumes wf: wf-prog wf-md P
```

```
  and has: P ⊢ C has F:T (fm) in E
  shows ⟦ P ⊢ C ≼* D; P ⊢ D ≼* E ⟧ ⟹ P ⊢ D has F:T (fm) in E
proof(induct rule: rtranclp-induct)
  case base show ?case using has .
next
  case (step D D')
  note DsubD' = ‹P ⊢ D ≺1 D'›
  from DsubD' obtain rest where classD: class P D = ⌊(D', rest)⌋
    and DObj: D ≠ Object by(auto elim!: subcls1.cases)
  from DsubD' ‹P ⊢ D' ≼* E› have DsubE: P ⊢ D ≼* E and DsubE2: (subcls1 P)^++ D E
    by(rule converse-rtranclp-into-rtranclp rtranclp-into-tranclp2)+
  from wf DsubE2 have DnE: D ≠ E by(rule subcls-irrefl)
  from DsubE have hasD: P ⊢ D has F:T (fm) in E by(rule ‹P ⊢ D ≼* E ⟹ P ⊢ D has F:T (fm)
in E›)
  then obtain FDTs where hasf: P ⊢ D has-fields FDTs and FE: map-of FDTs (F, E) = ⌊(T, fm)⌋
    unfolding has-field-def by blast
  from hasf show ?case
  proof cases
    case has-fields-Object with DObj show ?thesis by simp
  next
    case (has-fields-rec DD' fs ms FDTs')
    with classD have [simp]: DD' = D' rest = (fs, ms)
      and hasf': P ⊢ D' has-fields FDTs'
      and FDTs: FDTs = map (λ(F, Tm). ((F, D), Tm)) fs @ FDTs' by auto
    from FDTs FE DnE hasf' show ?thesis by(auto dest: map-of-SomeD simp add: has-field-def)
  qed
qed

lemma wf-has-field-idemp:
  ⟦ wf-prog wf-md P; P ⊢ C has F:T (fm) in D ⟧ ⟹ P ⊢ D has F:T (fm) in D
apply(frule has-field-decl-above)
apply(erule (2) wf-has-field-mono2)
apply(rule rtranclp.rtrancl-refl)
done

lemma map-of-remap-conv:
  ⟦ distinct-fst fs; map-of (map (λ(F, y). ((F, D), y)) fs) (F, D) = ⌊T⌋ ⟧
  ⟹ map-of (map (λ((F, D), T). (F, D, T)) (map (λ(F, y). ((F, D), y)) fs)) F = ⌊(D, T)⌋
apply(induct fs)
apply auto
done

lemma has-field-idemp-sees-field:
  assumes wf: wf-prog wf-md P
  and has: P ⊢ D has F:T (fm) in D
  shows P ⊢ D sees F:T (fm) in D
proof −
  from has obtain FDTs where hasf: P ⊢ D has-fields FDTs
    and FD: map-of FDTs (F, D) = ⌊(T, fm)⌋ unfolding has-field-def by blast
  from hasf have map-of (map (λ((F, D), T). (F, D, T)) FDTs) F = ⌊(D, T, fm)⌋
  proof cases
    case (has-fields-Object D' fs ms)
    from ‹class P Object = ⌊(D', fs, ms)⌋› wf
    have wf-cdecl wf-md P (Object, D', fs, ms) by(rule class-wf)
```

```
    hence distinct-fst fs by(simp add: wf-cdecl-def)
    with FD has-fields-Object show ?thesis by(auto intro: map-of-remap-conv simp del: map-map)
  next
    case (has-fields-rec D' fs ms FDTs')
    hence [simp]: FDTs = map (λ(F, Tm). ((F, D), Tm)) fs @ FDTs'
      and classD: class P D = ⌊(D', fs, ms)⌋ and DnObj: D ≠ Object
      and hasf': P ⊢ D' has-fields FDTs' by auto
    from ‹class P D = ⌊(D', fs, ms)⌋› wf
    have wf-cdecl wf-md P (D, D', fs, ms) by(rule class-wf)
    hence distinct-fst fs by(simp add: wf-cdecl-def)
    moreover have map-of FDTs' (F, D) = None
    proof(rule ccontr)
      assume map-of FDTs' (F, D) ≠ None
      then obtain T' fm' where map-of FDTs' (F, D) = ⌊(T', fm')⌋ by(auto)
      with hasf' have P ⊢ D' ≼* D by(auto dest!: map-of-SomeD intro: has-fields-decl-above)
      with classD DnObj have (subcls1 P)^++ D D
        by(auto intro: subcls1.intros rtranclp-into-tranclp2)
      with wf show False by(auto dest: subcls-irrefl)
    qed
    ultimately show ?thesis using FD hasf'
      by(auto simp add: map-add-Some-iff intro: map-of-remap-conv simp del: map-map)
  qed
  with hasf show ?thesis unfolding sees-field-def by blast
qed

lemma has-fields-distinct:
  assumes wf: wf-prog wf-md P
  and P ⊢ C has-fields FDTs
  shows distinct (map fst FDTs)
using ‹P ⊢ C has-fields FDTs›
proof(induct)
  case (has-fields-Object D fs ms FDTs)
  have eq: map (fst ∘ (λ(F, y). ((F, Object), y))) fs = map ((λF. (F, Object)) ∘ fst) fs by(auto)
  from ‹class P Object = ⌊(D, fs, ms)⌋› wf
  have wf-cdecl wf-md P (Object, D, fs, ms) by(rule class-wf)
  hence distinct (map fst fs) by(simp add: wf-cdecl-def distinct-fst-def)
  hence distinct (map (fst ∘ (λ(F, y). ((F, Object), y))) fs)
    unfolding eq distinct-map by(auto intro: comp-inj-on inj-onI)
  thus ?case using ‹FDTs = map (λ(F, T). ((F, Object), T)) fs› by(simp)
next
  case (has-fields-rec C D fs ms FDTs FDTs')
  have eq: map (fst ∘ (λ(F, y). ((F, C), y))) fs = map ((λF. (F, C)) ∘ fst) fs by(auto)
  from ‹class P C = ⌊(D, fs, ms)⌋› wf
  have wf-cdecl wf-md P (C, D, fs, ms) by(rule class-wf)
  hence distinct (map fst fs) by(simp add: wf-cdecl-def distinct-fst-def)
  hence distinct (map (fst ∘ (λ(F, y). ((F, C), y))) fs)
    unfolding eq distinct-map by(auto intro: comp-inj-on inj-onI)
  moreover from ‹class P C = ⌊(D, fs, ms)⌋› ‹C ≠ Object›
  have P ⊢ C ≺1 D by(rule subcls1.intros)
  with ‹P ⊢ D has-fields FDTs›
  have (fst ∘ (λ(F, y). ((F, C), y))) ‘ set fs ∩ fst ‘ set FDTs = {}
    by(auto dest: subcls-notin-has-fields)
  ultimately show ?case using ‹FDTs' = map (λ(F, T). ((F, C), T)) fs @ FDTs› ‹distinct (map
fst FDTs)› by simp
```

qed

3.12.4 Code generation

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$)
[*inductify*]
wf-overriding
.

Separate subclass acycilicity from class declaration check. Otherwise, cyclic class hierarchies might lead to non-termination as *Methods* recurses over the class hierarchy.

definition *acyclic-class-hierarchy* :: *'m prog* $\Rightarrow$ *bool*
where
acyclic-class-hierarchy P $\longleftrightarrow$
$(\forall (C, D, fs, ml) \in set\ (classes\ P).\ C \neq Object \longrightarrow \neg\ P \vdash D \preceq^* C)$

definition *wf-cdecl'* :: *'m wf-mdecl-test* $\Rightarrow$ *'m prog* $\Rightarrow$ *'m cdecl* $\Rightarrow$ *bool*
where
wf-cdecl' wf-md P = $(\lambda(C,(D,fs,ms)).$
$(\forall f \in set\ fs.\ wf\text{-}fdecl\ P\ f) \wedge distinct\text{-}fst\ fs \wedge$
$(\forall m \in set\ ms.\ wf\text{-}mdecl\ wf\text{-}md\ P\ C\ m) \wedge$
$distinct\text{-}fst\ ms \wedge$
$(C \neq Object \longrightarrow is\text{-}class\ P\ D \wedge (\forall m \in set\ ms.\ wf\text{-}overriding\ P\ D\ m)) \wedge$
$(C = Thread \longrightarrow (\exists m.\ (run, [], Void, m) \in set\ ms)))$

lemma *acyclic-class-hierarchy-code* [*code*]:
acyclic-class-hierarchy P $\longleftrightarrow$ $(\forall (C, D, fs, ml) \in set\ (classes\ P).\ C \neq Object \longrightarrow \neg\ subcls'\ P\ D\ C)$
by(*simp add*: *acyclic-class-hierarchy-def subcls'-def*)

lemma *wf-cdecl'-code* [*code*]:
wf-cdecl' wf-md P = $(\lambda(C,(D,fs,ms)).$
$(\forall f \in set\ fs.\ wf\text{-}fdecl\ P\ f) \wedge\ distinct\text{-}fst\ fs \wedge$
$(\forall m \in set\ ms.\ wf\text{-}mdecl\ wf\text{-}md\ P\ C\ m) \wedge$
$distinct\text{-}fst\ ms \wedge$
$(C \neq Object \longrightarrow is\text{-}class\ P\ D \wedge (\forall m \in set\ ms.\ wf\text{-}overriding\ P\ D\ m)) \wedge$
$(C = Thread \longrightarrow ((run, [], Void) \in set\ (map\ (\lambda(M,\ Ts,\ T,\ b).\ (M,\ Ts,\ T))\ ms))))$
by(*auto simp add*: *wf-cdecl'-def intro*!: *ext intro*: *rev-image-eqI*)

declare *set-append* [*symmetric*, *code-unfold*]

lemma *wf-prog-code* [*code*]:
wf-prog wf-md P $\longleftrightarrow$
acyclic-class-hierarchy P $\wedge$
wf-syscls P $\wedge$ *distinct-fst* (*classes P*) $\wedge$
$(\forall c \in set\ (classes\ P).\ wf\text{-}cdecl'\ wf\text{-}md\ P\ c)$
unfolding *wf-prog-def wf-cdecl-def wf-cdecl'-def acyclic-class-hierarchy-def split-def*
by *blast*

end

3.13 Properties of external calls in well-formed programs

theory *ExternalCallWF*

imports
WellForm
../Framework/FWSemantics
begin

lemma *external-WT-defs-is-type*:
assumes *wf-prog wf-md P* **and** $C{\cdot}M(Ts) :: T$
shows *is-class P C* **and** *is-type P T set Ts* $\subseteq$ *types P*
using *assms* **by**(*auto elim*: *external-WT-defs.cases*)

context *heap-base* **begin**

lemma *WT-red-external-aggr-imp-red-external*:
$[\![$ *wf-prog wf-md P*; $(ta, va, h') \in$ *red-external-aggr P t a M vs h*; $P,h \vdash a{\cdot}M(vs) : U$; $P,h \vdash t\ \surd t$ $]\!]$
$\Longrightarrow P,t \vdash \langle a{\cdot}M(vs), h\rangle -ta\rightarrow ext\ \langle va, h'\rangle$
apply(*drule tconfD*)
apply(*erule external-WT'.cases*)
apply(*clarsimp*)
apply(*drule* (*1*) *sees-wf-native*)
apply(*erule external-WT-defs.cases*)
apply(*case-tac* [!] *hT*)
apply(*auto 4 4 simp add*: *red-external-aggr-def widen-Class intro*: *red-external.intros heap-base.red-external.intros*[**where** *addr2thread-id=addr2thread-id* **and** *thread-id2addr=thread-id2addr* **and** *spurious-wakeups=True* **and** *empty-heap=empty-heap* **and** *allocate=allocate* **and** *typeof-addr=typeof-addr* **and** *heap-read=heap-read* **and** *heap-write=heap-write*] *heap-base.red-external.intros*[**where** *addr2thread-id=addr2thread-id* **and** *thread-id2addr=thread-id2addr* **and** *spurious-wakeups=False* **and** *empty-heap=empty-heap* **and** *allocate=allocate* **and** *typeof-addr=typeof-addr* **and** *heap-read=heap-read* **and** *heap-write=heap-write*] *split*: *if-split-asm dest*: *sees-method-decl-above*)
done

lemma *WT-red-external-list-conv*:
$[\![$ *wf-prog wf-md P*; $P,h \vdash a{\cdot}M(vs) : U$; $P,h \vdash t\ \surd t$ $]\!]$
$\Longrightarrow P,t \vdash \langle a{\cdot}M(vs), h\rangle -ta\rightarrow ext\ \langle va, h'\rangle \longleftrightarrow (ta, va, h') \in$ *red-external-aggr P t a M vs h*
by(*blast intro*: *WT-red-external-aggr-imp-red-external red-external-imp-red-external-aggr*)

lemma *red-external-new-thread-sees*:
$[\![$ *wf-prog wf-md P*; $P,t \vdash \langle a{\cdot}M(vs), h\rangle -ta\rightarrow ext\ \langle va, h'\rangle$; *NewThread t'* (C, M', a') $h'' \in$ *set* $\{\!|ta|\!\}_t$ $]\!]$
$\Longrightarrow$ *typeof-addr h' a'* $= \lfloor$*Class-type C*$\rfloor \wedge (\exists T\ meth\ D.\ P \vdash C$ *sees* M':$[]\rightarrow T = \lfloor meth\rfloor$ *in D*$)$
by(*fastforce elim*!: *red-external.cases simp add*: *widen-Class ta-upd-simps dest*: *sub-Thread-sees-run*)

end

3.13.1 Preservation of heap conformance

context *heap-conf-read* **begin**

lemma *hconf-heap-copy-loc-mono*:
assumes *heap-copy-loc a a' al h obs h'*
and *hconf h*
and $P,h \vdash a@al : T$ $P,h \vdash a'@al : T$
shows *hconf h'*
proof −
from ‹*heap-copy-loc a a' al h obs h'*› **obtain** *v*
where *read*: *heap-read h a al v*

```
    and write: heap-write h a' al v h' by cases auto
  from read ‹P,h ⊢ a@al : T› ‹hconf h› have P,h ⊢ v :≤ T
    by(rule heap-read-conf)
  with write ‹hconf h› ‹P,h ⊢ a'@al : T› show ?thesis
    by(rule hconf-heap-write-mono)
qed

lemma hconf-heap-copies-mono:
  assumes heap-copies a a' als h obs h'
  and hconf h
  and list-all2 (λal T. P,h ⊢ a@al : T) als Ts
  and list-all2 (λal T. P,h ⊢ a'@al : T) als Ts
  shows hconf h'
using assms
proof(induct arbitrary: Ts)
  case Nil thus ?case by simp
next
  case (Cons al h ob h' als obs h'')
  note step = ‹heap-copy-loc a a' al h ob h'›
  from ‹list-all2 (λal T. P,h ⊢ a@al : T) (al # als) Ts›
  obtain T Ts' where [simp]: Ts = T # Ts'
    and P,h ⊢ a@al : T list-all2 (λal T. P,h ⊢ a@al : T) als Ts'
    by(auto simp add: list-all2-Cons1)
  from ‹list-all2 (λal T. P,h ⊢ a'@al : T) (al # als) Ts›
  have P,h ⊢ a'@al : T list-all2 (λal T. P,h ⊢ a'@al : T) als Ts' by simp-all
  from step ‹hconf h› ‹P,h ⊢ a@al : T› ‹P,h ⊢ a'@al : T›
  have hconf h' by(rule hconf-heap-copy-loc-mono)
  moreover from step have h ◁ h' by(rule hext-heap-copy-loc)
  from ‹list-all2 (λal T. P,h ⊢ a@al : T) als Ts'›
  have list-all2 (λal T. P,h' ⊢ a@al : T) als Ts'
    by(rule list-all2-mono)(rule addr-loc-type-hext-mono[OF - ‹h ◁ h'›])
  moreover from ‹list-all2 (λal T. P,h ⊢ a'@al : T) als Ts'›
  have list-all2 (λal T. P,h' ⊢ a'@al : T) als Ts'
    by(rule list-all2-mono)(rule addr-loc-type-hext-mono[OF - ‹h ◁ h'›])
  ultimately show ?case by(rule Cons)
qed

lemma hconf-heap-clone-mono:
  assumes heap-clone P h a h' res
  and hconf h
  shows hconf h'
using ‹heap-clone P h a h' res›
proof cases
  case CloneFail thus ?thesis using ‹hconf h›
    by(fastforce intro: hconf-heap-ops-mono dest: typeof-addr-is-type)
next
  case (ObjClone C h'' a' FDTs obs)
  note FDTs = ‹P ⊢ C has-fields FDTs›
  let ?als = map (λ((F, D), Tfm). CField D F) FDTs
  let ?Ts = map (λ(FD, T). fst (the (map-of FDTs FD))) FDTs
  note ‹heap-copies a a' ?als h'' obs h'›
  moreover from ‹typeof-addr h a = ⌊Class-type C⌋› ‹hconf h› have is-class P C
    by(auto dest: typeof-addr-is-type)
  from ‹(h'', a') ∈ allocate h (Class-type C)› have h ◁ h'' hconf h''
```

```
    by(rule hext-heap-ops hconf-allocate-mono)+(simp-all add: ‹hconf h› ‹is-class P C›)
  note ‹hconf h''›
  moreover
  from ‹typeof-addr h a = ⌊Class-type C⌋› FDTs
  have list-all2 (λal T. P,h ⊢ a@al : T) ?als ?Ts
    unfolding list-all2-map1 list-all2-map2 list-all2-same
    by(fastforce intro: addr-loc-type.intros simp add: has-field-def dest: weak-map-of-SomeI)
  hence list-all2 (λal T. P,h'' ⊢ a@al : T) ?als ?Ts
    by(rule list-all2-mono)(rule addr-loc-type-hext-mono[OF - ‹h ◁ h''›])
  moreover from ‹(h'', a') ∈ allocate h (Class-type C)› ‹is-class P C›
  have typeof-addr h'' a' = ⌊Class-type C⌋ by(auto dest: allocate-SomeD)
  with FDTs have list-all2 (λal T. P,h'' ⊢ a'@al : T) ?als ?Ts
    unfolding list-all2-map1 list-all2-map2 list-all2-same
    by(fastforce intro: addr-loc-type.intros simp add: has-field-def dest: weak-map-of-SomeI)
  ultimately have hconf h' by(rule hconf-heap-copies-mono)
  thus ?thesis using ObjClone by simp
next
  case (ArrClone T n h'' a' FDTs obs)
  let ?als = map (λ((F, D), Tfm). CField D F) FDTs @ map ACell [0..<n]
  let ?Ts = map (λ(FD, T). fst (the (map-of FDTs FD))) FDTs @ replicate n T
  note ‹heap-copies a a' ?als h'' obs h'›
  moreover from ‹typeof-addr h a = ⌊Array-type T n⌋› ‹hconf h› have is-type P (T⌊⌉)
    by(auto dest: typeof-addr-is-type)
  from ‹(h'', a') ∈ allocate h (Array-type T n)› have h ◁ h'' hconf h''
    by(rule hext-heap-ops hconf-allocate-mono)+(simp-all add: ‹hconf h› ‹is-type P (T⌊⌉)›[simplified])
  note ‹hconf h''›
  moreover from ‹h ◁ h''› ‹typeof-addr h a = ⌊Array-type T n⌋›
  have type'a: typeof-addr h'' a = ⌊Array-type T n⌋ by(auto intro: hext-arrD)
  note FDTs = ‹P ⊢ Object has-fields FDTs›
  from type'a FDTs have list-all2 (λal T. P,h'' ⊢ a@al : T) ?als ?Ts
   by(fastforce intro: list-all2-all-nthI addr-loc-type.intros simp add: has-field-def distinct-fst-def list-all2-append
list-all2-map1 list-all2-map2 list-all2-same dest: weak-map-of-SomeI)
  moreover from ‹(h'', a') ∈ allocate h (Array-type T n)› ‹is-type P (T⌊⌉)›
  have typeof-addr h'' a' = ⌊Array-type T n⌋ by(auto dest: allocate-SomeD)
  hence list-all2 (λal T. P,h'' ⊢ a'@al : T) ?als ?Ts using FDTs
   by(fastforce intro: list-all2-all-nthI addr-loc-type.intros simp add: has-field-def distinct-fst-def list-all2-append
list-all2-map1 list-all2-map2 list-all2-same dest: weak-map-of-SomeI)
  ultimately have hconf h' by(rule hconf-heap-copies-mono)
  thus ?thesis using ArrClone by simp
qed

theorem external-call-hconf:
  assumes major: P,t ⊢ ⟨a•M(vs), h⟩ −ta→ext ⟨va, h'⟩
  and minor: P,h ⊢ a•M(vs) : U hconf h
  shows hconf h'
using major minor
by cases(fastforce intro: hconf-heap-clone-mono)+

end

context heap-base begin

primrec conf-extRet :: 'm prog ⇒ 'heap ⇒ 'addr extCallRet ⇒ ty ⇒ bool where
  conf-extRet P h (RetVal v) T = (P,h ⊢ v :≤ T)
```

| *conf-extRet P h (RetExc a) T = (P,h ⊢ Addr a :≤ Class Throwable)*
| *conf-extRet P h RetStaySame T = True*

end

context *heap-conf* **begin**

lemma *red-external-conf-extRet*:
  assumes *wf*: *wf-prog wf-md P*
  shows ⟦*P,t ⊢ ⟨a·M(vs), h⟩ −ta→ext ⟨va, h'⟩*; *P,h ⊢ a·M(vs) : U*; *hconf h*; *preallocated h*; *P,h ⊢ t √t* ⟧
  ⟹ *conf-extRet P h' va U*
using *wf* **apply** −
apply(*frule red-external-hext*)
apply(*drule* (*1*) *preallocated-hext*)
apply(*auto elim*!: *red-external.cases external-WT'.cases external-WT-defs-cases dest*!: *sees-wf-native*[*OF wf*])
apply(*auto simp add*: *conf-def tconf-def intro*: *xcpt-subcls-Throwable dest*!: *hext-heap-write*)
apply(*case-tac hT*)
apply(*auto 4 4 dest*!: *typeof-addr-heap-clone dest*: *typeof-addr-is-type intro*: *widen-array-object subcls-C-Object*)
done

end

3.13.2 Progress theorems for external calls

context *heap-progress* **begin**

lemma *heap-copy-loc-progress*:
  assumes *hconf*: *hconf h*
  and *alconfa*: *P,h ⊢ a@al : T*
  and *alconfa'*: *P,h ⊢ a'@al : T*
  shows ∃*v h'*. *heap-copy-loc a a' al h* ([*ReadMem a al v, WriteMem a' al v*]) *h'* ∧ *P,h ⊢ v :≤ T* ∧ *hconf h'*
proof −
  from *heap-read-total*[*OF hconf alconfa*]
  obtain *v* **where** *heap-read h a al v P,h ⊢ v :≤ T* **by** *blast*
  moreover from *heap-write-total*[*OF hconf alconfa' ‹P,h ⊢ v :≤ T›*] **obtain** *h'* **where** *heap-write h a' al v h'* **..**
  moreover hence *hconf h'* **using** *hconf alconfa' ‹P,h ⊢ v :≤ T›* **by**(*rule hconf-heap-write-mono*)
  ultimately show *?thesis* **by**(*blast intro*: *heap-copy-loc.intros*)
qed

lemma *heap-copies-progress*:
  assumes *hconf h*
  and *list-all2* (λ*al T*. *P,h ⊢ a@al : T*) *als Ts*
  and *list-all2* (λ*al T*. *P,h ⊢ a'@al : T*) *als Ts*
  shows ∃*vs h'*. *heap-copies a a' als h* (*concat* (*map* (λ(*al*, *v*). [*ReadMem a al v, WriteMem a' al v*]) (*zip als vs*))) *h'* ∧ *hconf h'*
using *assms*
proof(*induct als arbitrary*: *h Ts*)
  case *Nil* **thus** *?case* **by**(*auto intro*: *heap-copies.Nil*)
next

case (*Cons al als*)
from ‹*list-all2* (λ*al T*. *P*,*h* ⊢ *a*@*al* : *T*) (*al* # *als*) *Ts*›
obtain *T′ Ts′* **where** [*simp*]: *Ts* = *T′* # *Ts′*
and *P*,*h* ⊢ *a*@*al* : *T′* *list-all2* (λ*al T*. *P*,*h* ⊢ *a*@*al* : *T*) *als Ts′*
by(*auto simp add*: *list-all2-Cons1*)
from ‹*list-all2* (λ*al T*. *P*,*h* ⊢ *a′*@*al* : *T*) (*al* # *als*) *Ts*›
have *P*,*h* ⊢ *a′*@*al* : *T′* **and** *list-all2* (λ*al T*. *P*,*h* ⊢ *a′*@*al* : *T*) *als Ts′* **by** *simp-all*
from ‹*hconf h*› ‹*P*,*h* ⊢ *a*@*al* : *T′*› ‹*P*,*h* ⊢ *a′*@*al* : *T′*›
obtain *v h′* **where** *heap-copy-loc a a′ al h* [*ReadMem a al v*, *WriteMem a′ al v*] *h′*
and *hconf h′* **by**(*fastforce dest*: *heap-copy-loc-progress*)
moreover hence *h* ⊴ *h′* **by**−(*rule hext-heap-copy-loc*)
{
note ‹*hconf h′*›
moreover from ‹*list-all2* (λ*al T*. *P*,*h* ⊢ *a*@*al* : *T*) *als Ts′*›
have *list-all2* (λ*al T*. *P*,*h′* ⊢ *a*@*al* : *T*) *als Ts′*
by(*rule list-all2-mono*)(*rule addr-loc-type-hext-mono*[*OF* - ‹*h* ⊴ *h′*›])
moreover from ‹*list-all2* (λ*al T*. *P*,*h* ⊢ *a′*@*al* : *T*) *als Ts′*›
have *list-all2* (λ*al T*. *P*,*h′* ⊢ *a′*@*al* : *T*) *als Ts′*
by(*rule list-all2-mono*)(*rule addr-loc-type-hext-mono*[*OF* - ‹*h* ⊴ *h′*›])
ultimately have ∃ *vs h″*. *heap-copies a a′ als h′* (*concat* (*map* (λ(*al*, *v*). [*ReadMem a al v*, *WriteMem a′ al v*]) (*zip als vs*))) *h″* ∧ *hconf h″*
by(*rule Cons*) **}**
then obtain *vs h″*
where *heap-copies a a′ als h′* (*concat* (*map* (λ(*al*, *v*). [*ReadMem a al v*, *WriteMem a′ al v*]) (*zip als vs*))) *h″*
and *hconf h″* **by** *blast*
ultimately
have *heap-copies a a′* (*al* # *als*) *h* ([*ReadMem a al v*, *WriteMem a′ al v*] @ (*concat* (*map* (λ(*al*, *v*). [*ReadMem a al v*, *WriteMem a′ al v*]) (*zip als vs*)))) *h″*
by− (*rule heap-copies.Cons*)
also have [*ReadMem a al v*, *WriteMem a′ al v*] @ (*concat* (*map* (λ(*al*, *v*). [*ReadMem a al v*, *WriteMem a′ al v*]) (*zip als vs*))) =
(*concat* (*map* (λ(*al*, *v*). [*ReadMem a al v*, *WriteMem a′ al v*]) (*zip* (*al* # *als*) (*v* # *vs*)))) **by** *simp*
finally show *?case* **using** ‹*hconf h″*› **by** *blast*
qed

lemma *heap-clone-progress*:
assumes *wf*: *wf-prog wf-md P*
and *typea*: *typeof-addr h a* = ⌊*hT*⌋
and *hconf*: *hconf h*
shows ∃ *h′ res*. *heap-clone P h a h′ res*
proof −
from *typea hconf* **have** *is-htype P hT* **by**(*rule typeof-addr-is-type*)
show *?thesis*
proof(*cases allocate h hT* = {})
case *True*
with *typea CloneFail*[*of h a hT P*]
show *?thesis* **by** *auto*
next
case *False*
then obtain *h′ a′* **where** *new*: (*h′*, *a′*) ∈ *allocate h hT* **by**(*rule not-empty-pairE*)
hence *h* ⊴ *h′* **by**(*rule hext-allocate*)
have *hconf h′* **using** *new hconf* ‹*is-htype P hT*› **by**(*rule hconf-allocate-mono*)

```
    show ?thesis
    proof(cases hT)
      case [simp]: (Class-type C)
      from ‹is-htype P hT› have is-class P C by simp
      from wf-Fields-Ex[OF wf this]
      obtain FDTs where FDTs: P ⊢ C has-fields FDTs ..
      let ?als = map (λ((F, D), Tfm). CField D F) FDTs
      let ?Ts = map (λ(FD, T). fst (the (map-of FDTs FD))) FDTs
      from typea FDTs have list-all2 (λal T. P,h ⊢ a@al : T) ?als ?Ts
        unfolding list-all2-map1 list-all2-map2 list-all2-same
        by(fastforce intro: addr-loc-type.intros simp add: has-field-def dest: weak-map-of-SomeI)
      hence list-all2 (λal T. P,h' ⊢ a@al : T) ?als ?Ts
        by(rule list-all2-mono)(simp add: addr-loc-type-hext-mono[OF - ‹h ◁ h'›] split-def)
      moreover from new ‹is-class P C›
      have typeof-addr h' a' = ⌊Class-type C⌋ by(auto dest: allocate-SomeD)
      with FDTs have list-all2 (λal T. P,h' ⊢ a'@al : T) ?als ?Ts
        unfolding list-all2-map1 list-all2-map2 list-all2-same
       by(fastforce intro: addr-loc-type.intros map-of-SomeI simp add: has-field-def dest: weak-map-of-SomeI)
      ultimately obtain obs h'' where heap-copies a a' ?als h' obs h'' hconf h''
        by(blast dest: heap-copies-progress[OF ‹hconf h'›])
      with typea new FDTs ObjClone[of h a C h' a' P FDTs obs h'']
      show ?thesis by auto
    next
      case [simp]: (Array-type T n)
      from wf obtain FDTs where FDTs: P ⊢ Object has-fields FDTs
        by(blast dest: wf-Fields-Ex is-class-Object)
      let ?als = map (λ((F, D), Tfm). CField D F) FDTs @ map ACell [0..<n]
      let ?Ts = map (λ(FD, T). fst (the (map-of FDTs FD))) FDTs @ replicate n T
      from ‹h ◁ h'› typea have type'a: typeof-addr h' a = ⌊Array-type T n⌋
        by(auto intro: hext-arrD)
      from type'a FDTs have list-all2 (λal T. P,h' ⊢ a@al : T) ?als ?Ts
        by(fastforce intro: list-all2-all-nthI addr-loc-type.intros simp add: has-field-def list-all2-append
list-all2-map1 list-all2-map2 list-all2-same dest: weak-map-of-SomeI)
      moreover from new ‹is-htype P hT›
      have typeof-addr h' a' = ⌊Array-type T n⌋
        by(auto dest: allocate-SomeD)
      hence list-all2 (λal T. P,h' ⊢ a'@al : T) ?als ?Ts using FDTs
        by(fastforce intro: list-all2-all-nthI addr-loc-type.intros simp add: has-field-def list-all2-append
list-all2-map1 list-all2-map2 list-all2-same dest: weak-map-of-SomeI)
      ultimately obtain obs h'' where heap-copies a a' ?als h' obs h'' hconf h''
        by(blast dest: heap-copies-progress[OF ‹hconf h'›])
      with typea new FDTs ArrClone[of h a T n h' a' P FDTs obs h'']
      show ?thesis by auto
    qed
  qed
qed

theorem external-call-progress:
  assumes wf: wf-prog wf-md P
  and wt: P,h ⊢ a·M(vs) : U
  and hconf: hconf h
  shows ∃ta va h'. P,t ⊢ ⟨a·M(vs), h⟩ −ta→ext ⟨va, h'⟩
proof −
  note [simp del] = split-paired-Ex
```

from *wt* **obtain** *hT Ts Ts′ D*
where *T*: *typeof-addr h a* = ⌊*hT*⌋ **and** *Ts*: *map typeof*$_h$ *vs* = *map Some Ts*
and *P* ⊢ *class-type-of hT sees M*:*Ts′*→*U* = *Native in D* **and** *subTs*: *P* ⊢ *Ts* [≤] *Ts′*
unfolding *external-WT′-iff* **by** *blast*
from *wf* ‹*P* ⊢ *class-type-of hT sees M*:*Ts′*→*U* = *Native in D*›
have *D*·*M*(*Ts′*) :: *U* **by**(*rule sees-wf-native*)
moreover from ‹*P* ⊢ *class-type-of hT sees M*:*Ts′*→*U* = *Native in D*›
have *P* ⊢ *ty-of-htype hT* ≤ *Class D*
by(*cases hT*)(*auto dest*: *sees-method-decl-above intro*: *widen-trans widen-array-object*)
ultimately show *?thesis* **using** *T Ts subTs*
proof *cases*
assume [*simp*]: *D* = *Object M* = *clone Ts′* = [] *U* = *Class Object*
from *heap-clone-progress*[*OF wf T hconf*] **obtain** *h′ res* **where** *heap-clone P h a h′ res* **by** *blast*
thus *?thesis* **using** *subTs Ts* **by**(*cases res*)(*auto intro*: *red-external.intros*)
qed(*auto simp add*: *widen-Class intro*: *red-external.intros*)
qed

end

3.13.3 Lemmas for preservation of deadlocked threads

context *heap-progress* **begin**

lemma *red-external-wt-hconf-hext*:
assumes *wf*: *wf-prog wf-md P*
and *red*: *P*,*t* ⊢ ⟨*a*·*M*(*vs*),*h*⟩ −*ta*→*ext* ⟨*va*,*h′*⟩
and *hext*: *h″* ⊴ *h*
and *wt*: *P*,*h″* ⊢ *a*·*M*(*vs*) : *U*
and *tconf*: *P*,*h″* ⊢ *t* √*t*
and *hconf*: *hconf h″*
shows ∃ *ta′ va′ h‴*. *P*,*t* ⊢ ⟨*a*·*M*(*vs*),*h″*⟩ −*ta′*→*ext* ⟨*va′*, *h‴*⟩ ∧
collect-locks ⦃*ta*⦄$_l$ = *collect-locks* ⦃*ta′*⦄$_l$ ∧
collect-cond-actions ⦃*ta*⦄$_c$ = *collect-cond-actions* ⦃*ta′*⦄$_c$ ∧
collect-interrupts ⦃*ta*⦄$_i$ = *collect-interrupts* ⦃*ta′*⦄$_i$
using *red wt hext*
proof *cases*
case (*RedClone obs a′*)
from *wt* **obtain** *hT C Ts Ts′ D*
where *T*: *typeof-addr h″ a* = ⌊*hT*⌋
unfolding *external-WT′-iff* **by** *blast*
from *heap-clone-progress*[*OF wf T hconf*]
obtain *h‴ res* **where** *heap-clone P h″ a h‴ res* **by** *blast*
thus *?thesis* **using** *RedClone*
by(*cases res*)(*fastforce intro*: *red-external.intros*)+
next
case *RedCloneFail*
from *wt* **obtain** *hT Ts Ts′*
where *T*: *typeof-addr h″ a* = ⌊*hT*⌋
unfolding *external-WT′-iff* **by** *blast*
from *heap-clone-progress*[*OF wf T hconf*]
obtain *h‴ res* **where** *heap-clone P h″ a h‴ res* **by** *blast*
thus *?thesis* **using** *RedCloneFail*
by(*cases res*)(*fastforce intro*: *red-external.intros*)+
qed(*fastforce simp add*: *ta-upd-simps elim*!: *external-WT′.cases intro*: *red-external.intros*[*simplified*]

dest: *typeof-addr-hext-mono*)+

lemma *red-external-wf-red*:
assumes *wf*: *wf-prog wf-md P*
and *red*: *P*,*t* ⊢ ⟨*a*·*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h*′⟩
and *tconf*: *P*,*h* ⊢ *t* √*t*
and *hconf*: *hconf h*
and *wst*: *wset s t* = *None* ∨ (*M* = *wait* ∧ (∃ *w*. *wset s t* = ⌊*PostWS w*⌋))
obtains *ta*′ *va*′ *h*″
where *P*,*t* ⊢ ⟨*a*·*M*(*vs*), *h*⟩ −*ta*′→*ext* ⟨*va*′, *h*″⟩
and *final-thread.actions-ok final s t ta*′ ∨ *final-thread.actions-ok*′ *s t ta*′ ∧ *final-thread.actions-subset ta*′ *ta*
proof(*atomize-elim*)
let *?a-t* = *thread-id2addr t*
let *?t-a* = *addr2thread-id a*

from *tconf* **obtain** *C* **where** *ht*: *typeof-addr h ?a-t* = ⌊*Class-type C*⌋
and *sub*: *P* ⊢ *C* ≼* *Thread* **by**(*fastforce dest*: *tconfD*)

show ∃ *ta*′ *va*′ *h*′. *P*,*t* ⊢ ⟨*a*·*M*(*vs*), *h*⟩ −*ta*′→*ext* ⟨*va*′, *h*′⟩ ∧ (*final-thread.actions-ok final s t ta*′ ∨ *final-thread.actions-ok*′ *s t ta*′ ∧ *final-thread.actions-subset ta*′ *ta*)
proof(*cases final-thread.actions-ok*′ *s t ta*)
case *True*
have *final-thread.actions-subset ta ta* **by**(*rule final-thread.actions-subset-refl*)
with *True red* **show** *?thesis* **by** *blast*
next
case *False*
note [*simp*] = *final-thread.actions-ok*′*-iff lock-ok-las*′*-def final-thread.cond-action-oks*′*-subset-Join final-thread.actions-subset-iff ta-upd-simps collect-cond-actions-def collect-interrupts-def*
note [*rule del*] = *subsetI*
note [*intro*] = *collect-locks*′*-subset-collect-locks red-external.intros*[*simplified*]

show *?thesis*
proof(*cases wset s t*)
case [*simp*]: (*Some w*)
with *wst* **obtain** *w*′ **where** [*simp*]: *w* = *PostWS w*′ *M* = *wait* **by** *auto*
from *red* **have** [*simp*]: *vs* = [] **by**(*auto elim*: *red-external.cases*)
show *?thesis*
proof(*cases w*′)
case [*simp*]: *WSWokenUp*
let *?ta*′ = {⌈*WokenUp*, *ClearInterrupt t*, *ObsInterrupted t*⌉}
have *final-thread.actions-ok*′ *s t ?ta*′ **by**(*simp add*: *wset-actions-ok-def*)
moreover have *final-thread.actions-subset ?ta*′ *ta*
by(*auto simp add*: *collect-locks*′*-def finfun-upd-apply*)
moreover from *RedWaitInterrupted*
have ∃ *va h*′. *P*,*t* ⊢ ⟨*a*·*M*(*vs*),*h*⟩ −*?ta*′→*ext* ⟨*va*,*h*′⟩ **by** *auto*
ultimately show *?thesis* **by** *blast*
next
case [*simp*]: *WSNotified*
let *?ta*′ = {⌈*Notified*⌉}
have *final-thread.actions-ok*′ *s t ?ta*′ **by**(*simp add*: *wset-actions-ok-def*)
moreover have *final-thread.actions-subset ?ta*′ *ta*
by(*auto simp add*: *collect-locks*′*-def finfun-upd-apply*)
moreover from *RedWaitNotified*

```
      have ∃ va h'. P,t ⊢ ⟨a•M(vs),h⟩ −?ta'→ext ⟨va,h'⟩ by auto
      ultimately show ?thesis by blast
    qed
  next
    case None

    from red False show ?thesis
    proof cases
      case (RedNewThread C)
      note ta = ‹ta = {|NewThread ?t-a (C, run, a) h, ThreadStart ?t-a|}›
      let ?ta' = {|ThreadExists ?t-a True|}
      from ta False None have final-thread.actions-ok' s t ?ta' by(auto)
      moreover from ta have final-thread.actions-subset ?ta' ta by(auto)
      ultimately show ?thesis using RedNewThread by(fastforce)
    next
      case RedNewThreadFail
     then obtain va' h' x where P,t ⊢ ⟨a•M(vs), h⟩ −{|NewThread ?t-a x h', ThreadStart ?t-a|}→ext
⟨va', h'⟩
        by(fastforce)
      moreover from ‹ta = {|ThreadExists ?t-a True|}› False None
      have final-thread.actions-ok' s t {|NewThread ?t-a x h', ThreadStart ?t-a|} by(auto)
      moreover from ‹ta = {|ThreadExists ?t-a True|}›
      have final-thread.actions-subset {|NewThread ?t-a x h', ThreadStart ?t-a|} ta by(auto)
      ultimately show ?thesis by blast
    next
      case RedJoin
      let ?ta = {|IsInterrupted t True, ClearInterrupt t, ObsInterrupted t|}
       from ‹ta = {|Join (addr2thread-id a), IsInterrupted t False, ThreadJoin (addr2thread-id a)|}›
None False
      have t ∈ interrupts s by(auto)
      hence final-thread.actions-ok final s t ?ta
        using None by(auto simp add: final-thread.actions-ok-iff final-thread.cond-action-oks.simps)
       moreover obtain va h' where P,t ⊢ ⟨a•M(vs),h⟩ −?ta→ext ⟨va,h'⟩ using RedJoinInterrupt
RedJoin by auto
      ultimately show ?thesis by blast
    next
      case RedJoinInterrupt
      hence False using False None by(auto)
      thus ?thesis ..
    next
      case RedInterrupt
      let ?ta = {|ThreadExists (addr2thread-id a) False|}
      from RedInterrupt None False
      have free-thread-id (thr s) (addr2thread-id a) by(auto simp add: wset-actions-ok-def)
      hence final-thread.actions-ok final s t ?ta using None
        by(auto simp add: final-thread.actions-ok-iff final-thread.cond-action-oks.simps)
     moreover obtain va h' where P,t ⊢ ⟨a•M(vs),h⟩ −?ta→ext ⟨va,h'⟩ using RedInterruptInexist
RedInterrupt by auto
      ultimately show ?thesis by blast
    next
      case RedInterruptInexist
         let ?ta = {|ThreadExists (addr2thread-id a) True, WakeUp (addr2thread-id a), Interrupt
(addr2thread-id a), ObsInterrupt (addr2thread-id a)|}
      from RedInterruptInexist None False
```

have ¬ *free-thread-id* (*thr s*) (*addr2thread-id a*) **by**(*auto simp add*: *wset-actions-ok-def*)
hence *final-thread.actions-ok final s t ?ta* **using** *None*
by(*auto simp add*: *final-thread.actions-ok-iff final-thread.cond-action-oks.simps wset-actions-ok-def*)
moreover obtain *va h'* **where** *P*,*t* ⊢ ⟨*a*·*M*(*vs*),*h*⟩ −*?ta*→*ext* ⟨*va*,*h'*⟩ **using** *RedInterruptInexist RedInterrupt* **by** *auto*
ultimately show *?thesis* **by** *blast*
next
case (*RedIsInterruptedTrue C*)
let *?ta'* = {⌈*IsInterrupted ?t-a False*⌉}
from *RedIsInterruptedTrue False None* **have** *?t-a* ∉ *interrupts s* **by**(*auto*)
hence *final-thread.actions-ok' s t ?ta'* **using** *None* **by** *auto*
moreover from *RedIsInterruptedTrue* **have** *final-thread.actions-subset ?ta' ta* **by** *auto*
moreover from *RedIsInterruptedTrue RedIsInterruptedFalse* **obtain** *va h'*
where *P*,*t* ⊢ ⟨*a*·*M*(*vs*),*h*⟩ −*?ta'*→*ext* ⟨*va*,*h'*⟩ **by** *auto*
ultimately show *?thesis* **by** *blast*
next
case (*RedIsInterruptedFalse C*)
let *?ta'* = {⌈*IsInterrupted ?t-a True*, *ObsInterrupted ?t-a*⌉}
from *RedIsInterruptedFalse* **have** *?t-a* ∈ *interrupts s*
using *False None* **by**(*auto*)
hence *final-thread.actions-ok final s t ?ta'*
using *None* **by**(*auto simp add*: *final-thread.actions-ok-iff final-thread.cond-action-oks.simps*)
moreover obtain *va h'* **where** *P*,*t* ⊢ ⟨*a*·*M*(*vs*),*h*⟩ −*?ta'*→*ext* ⟨*va*,*h'*⟩
using *RedIsInterruptedFalse RedIsInterruptedTrue* **by** *auto*
ultimately show *?thesis* **by** *blast*
next
case *RedWaitInterrupt*
note *ta* = ‹*ta* = {⌈*Unlock*→*a*, *Lock*→*a*, *IsInterrupted t True*, *ClearInterrupt t*, *ObsInterrupted t*⌉}›
from *ta False None* **have** *hli*: ¬ *has-lock* (*locks s* $ *a*) *t* ∨ *t* ∉ *interrupts s*
by(*fastforce simp add*: *lock-actions-ok'-iff finfun-upd-apply split*: *if-split-asm dest*: *may-lock-t-may-lock-unlock dest*: *has-lock-may-lock*)
show *?thesis*
proof(*cases has-lock* (*locks s* $ *a*) *t*)
case *True*
let *?ta'* = {⌈*Suspend a*, *Unlock*→*a*, *Lock*→*a*, *ReleaseAcquire*→*a*, *IsInterrupted t False*, *SyncUnlock a* ⌉}
from *True hli* **have** *t* ∉ *interrupts s* **by** *simp*
with *True False* **have** *final-thread.actions-ok' s t ?ta'* **using** *None*
by(*auto simp add*: *lock-actions-ok'-iff finfun-upd-apply wset-actions-ok-def Cons-eq-append-conv*)
moreover from *ta* **have** *final-thread.actions-subset ?ta' ta*
by(*auto simp add*: *collect-locks'-def finfun-upd-apply*)
moreover from *RedWait RedWaitInterrupt* **obtain** *va h'* **where** *P*,*t* ⊢ ⟨*a*·*M*(*vs*),*h*⟩ −*?ta'*→*ext* ⟨*va*,*h'*⟩ **by** *auto*
ultimately show *?thesis* **by** *blast*
next
case *False*
let *?ta'* = {⌈*UnlockFail*→*a*⌉}
from *False* **have** *final-thread.actions-ok' s t ?ta'* **using** *None*
by(*auto simp add*: *lock-actions-ok'-iff finfun-upd-apply*)
moreover from *ta* **have** *final-thread.actions-subset ?ta' ta*
by(*auto simp add*: *collect-locks'-def finfun-upd-apply*)
moreover from *RedWaitInterrupt* **obtain** *va h'* **where** *P*,*t* ⊢ ⟨*a*·*M*(*vs*),*h*⟩ −*?ta'*→*ext* ⟨*va*,*h'*⟩ **by**(*fastforce*)

ultimately show *?thesis* **by** *blast*
qed
next
case *RedWait*
note *ta* = ‹*ta* = {⌊*Suspend a*, *Unlock→a*, *Lock→a*, *ReleaseAcquire→a*, *IsInterrupted t False*, *SyncUnlock a*⌋}›

from *ta False None* **have** *hli*: ¬ *has-lock* (*locks s* $ *a*) *t* ∨ *t* ∈ *interrupts s*
by(*auto simp add*: *lock-actions-ok'-iff finfun-upd-apply wset-actions-ok-def Cons-eq-append-conv split*: *if-split-asm dest*: *may-lock-t-may-lock-unlock-lock-t dest*: *has-lock-may-lock*)
show *?thesis*
proof(*cases has-lock* (*locks s* $ *a*) *t*)
case *True*
let *?ta'* = {⌊*Unlock→a*, *Lock→a*, *IsInterrupted t True*, *ClearInterrupt t*, *ObsInterrupted t*⌋}
from *True hli* **have** *t* ∈ *interrupts s* **by** *simp*
with *True False* **have** *final-thread.actions-ok final s t ?ta'* **using** *None*
by(*auto simp add*: *final-thread.actions-ok-iff final-thread.cond-action-oks.simps lock-ok-las-def finfun-upd-apply has-lock-may-lock*)
moreover from *RedWait RedWaitInterrupt* **obtain** *va h'* **where** *P*,*t* ⊢ ⟨*a*·*M*(*vs*),*h*⟩ −*?ta'*→*ext* ⟨*va*,*h'*⟩ **by** *auto*
ultimately show *?thesis* **by** *blast*
next
case *False*
let *?ta'* = {⌊*UnlockFail→a*⌋}
from *False* **have** *final-thread.actions-ok' s t ?ta'* **using** *None*
by(*auto simp add*: *lock-actions-ok'-iff finfun-upd-apply*)
moreover from *ta* **have** *final-thread.actions-subset ?ta' ta*
by(*auto simp add*: *collect-locks'-def finfun-upd-apply*)
moreover from *RedWait RedWaitFail* **obtain** *va h'* **where** *P*,*t* ⊢ ⟨*a*·*M*(*vs*),*h*⟩ −*?ta'*→*ext* ⟨*va*,*h'*⟩ **by**(*fastforce*)
ultimately show *?thesis* **by** *blast*
qed
next
case *RedWaitFail*
note *ta* = ‹*ta* = {⌊*UnlockFail→a*⌋}›
let *?ta'* = *if t* ∈ *interrupts s*
then {⌊*Unlock→a*, *Lock→a*, *IsInterrupted t True*, *ClearInterrupt t*, *ObsInterrupted t*⌋}
else {⌊*Suspend a*, *Unlock→a*, *Lock→a*, *ReleaseAcquire→a*, *IsInterrupted t False*, *SyncUnlock a* ⌋}
from *ta False None* **have** *has-lock* (*locks s* $ *a*) *t*
by(*auto simp add*: *finfun-upd-apply split*: *if-split-asm*)
hence *final-thread.actions-ok final s t ?ta'* **using** *None*
by(*auto simp add*: *final-thread.actions-ok-iff final-thread.cond-action-oks.simps lock-ok-las-def finfun-upd-apply has-lock-may-lock wset-actions-ok-def*)
moreover from *RedWaitFail RedWait RedWaitInterrupt*
obtain *va h'* **where** *P*,*t* ⊢ ⟨*a*·*M*(*vs*),*h*⟩ −*?ta'*→*ext* ⟨*va*,*h'*⟩
by(*cases t* ∈ *interrupts s*) (*auto*)
ultimately show *?thesis* **by** *blast*
next
case *RedWaitNotified*
note *ta* = ‹*ta* = {⌊*Notified*⌋}›
let *?ta'* = *if has-lock* (*locks s* $ *a*) *t*
then (*if t* ∈ *interrupts s*
then {⌊*Unlock→a*, *Lock→a*, *IsInterrupted t True*, *ClearInterrupt t*, *ObsInterrupted*

```
t⦈
                    else ⦇Suspend a, Unlock→a, Lock→a, ReleaseAcquire→a, IsInterrupted t False,
SyncUnlock a ⦈)
               else ⦇UnlockFail→a⦈
      have final-thread.actions-ok final s t ?ta' using None
        by(auto simp add: final-thread.actions-ok-iff final-thread.cond-action-oks.simps lock-ok-las-def
finfun-upd-apply has-lock-may-lock wset-actions-ok-def)
      moreover from RedWaitNotified RedWait RedWaitInterrupt RedWaitFail
      have ∃ va h'. P,t ⊢ ⟨a•M(vs),h⟩ −?ta'→ext ⟨va,h'⟩ by auto
      ultimately show ?thesis by blast
    next
      case RedWaitInterrupted
      note ta = ‹ta = ⦇WokenUp, ClearInterrupt t, ObsInterrupted t⦈›
      let ?ta' = if has-lock (locks s $ a) t
               then (if t ∈ interrupts s
                     then ⦇Unlock→a, Lock→a, IsInterrupted t True, ClearInterrupt t, ObsInterrupted
t⦈
                     else ⦇Suspend a, Unlock→a, Lock→a, ReleaseAcquire→a, IsInterrupted t False,
SyncUnlock a ⦈)
               else ⦇UnlockFail→a⦈
      have final-thread.actions-ok final s t ?ta' using None
        by(auto simp add: final-thread.actions-ok-iff final-thread.cond-action-oks.simps lock-ok-las-def
finfun-upd-apply has-lock-may-lock wset-actions-ok-def)
      moreover from RedWaitInterrupted RedWait RedWaitInterrupt RedWaitFail
      have ∃ va h'. P,t ⊢ ⟨a•M(vs),h⟩ −?ta'→ext ⟨va,h'⟩ by auto
      ultimately show ?thesis by blast
    next
      case RedWaitSpurious
       note ta = ‹ta = ⦇Unlock→a, Lock→a, ReleaseAcquire→a, IsInterrupted t False, SyncUnlock
a⦈›
      from ta False None have hli: ¬ has-lock (locks s $ a) t ∨ t ∈ interrupts s
      by(auto simp add: lock-actions-ok'-iff finfun-upd-apply wset-actions-ok-def Cons-eq-append-conv
split: if-split-asm dest: may-lock-t-may-lock-unlock-lock-t dest: has-lock-may-lock)
      show ?thesis
      proof(cases has-lock (locks s $ a) t)
        case True
        let ?ta' = ⦇Unlock→a, Lock→a, IsInterrupted t True, ClearInterrupt t, ObsInterrupted t⦈
        from True hli have t ∈ interrupts s by simp
        with True False have final-thread.actions-ok final s t ?ta' using None
         by(auto simp add: final-thread.actions-ok-iff final-thread.cond-action-oks.simps lock-ok-las-def
finfun-upd-apply has-lock-may-lock)
        moreover from RedWaitInterrupt RedWaitSpurious(1−5)
        obtain va h' where P,t ⊢ ⟨a•M(vs),h⟩ −?ta'→ext ⟨va,h'⟩ by auto
        ultimately show ?thesis by blast
      next
        case False
        let ?ta' = ⦇UnlockFail→a⦈
        from False have final-thread.actions-ok' s t ?ta' using None
          by(auto simp add: lock-actions-ok'-iff finfun-upd-apply)
        moreover from ta have final-thread.actions-subset ?ta' ta
          by(auto simp add: collect-locks'-def finfun-upd-apply)
        moreover from RedWaitSpurious(1−5) RedWaitFail
        obtain va h' where P,t ⊢ ⟨a•M(vs),h⟩ −?ta'→ext ⟨va,h'⟩ by(fastforce)
        ultimately show ?thesis by blast
```

qed

next
case *RedNotify*
note *ta* = ‹*ta* = {⌊*Notify a*, *Unlock*→*a*, *Lock*→*a*⌋}›
let *?ta′* = {⌊*UnlockFail*→*a*⌋}
from *ta False None* **have** ¬ *has-lock* (*locks s* $ *a*) *t*
by(*fastforce simp add*: *lock-actions-ok′-iff finfun-upd-apply wset-actions-ok-def Cons-eq-append-conv split*: *if-split-asm dest*: *may-lock-t-may-lock-unlock-lock-t has-lock-may-lock*)
hence *final-thread.actions-ok′ s t ?ta′* **using** *None*
by(*auto simp add*: *lock-actions-ok′-iff finfun-upd-apply*)
moreover from *ta* **have** *final-thread.actions-subset ?ta′ ta*
by(*auto simp add*: *collect-locks′-def finfun-upd-apply*)
moreover from *RedNotify* **obtain** *va h′* **where** *P*,*t* ⊢ ⟨*a*·*M*(*vs*),*h*⟩ −*?ta′*→*ext* ⟨*va*,*h′*⟩ **by**(*fastforce*)
ultimately show *?thesis* **by** *blast*
next
case *RedNotifyFail*
note *ta* = ‹*ta* = {⌊*UnlockFail*→*a*⌋}›
let *?ta′* = {⌊*Notify a*, *Unlock*→*a*, *Lock*→*a*⌋}
from *ta False None* **have** *has-lock* (*locks s* $ *a*) *t*
by(*auto simp add*: *finfun-upd-apply split*: *if-split-asm*)
hence *final-thread.actions-ok′ s t ?ta′* **using** *None*
by(*auto simp add*: *finfun-upd-apply simp add*: *wset-actions-ok-def intro*: *has-lock-may-lock*)
moreover from *ta* **have** *final-thread.actions-subset ?ta′ ta*
by(*auto simp add*: *collect-locks′-def finfun-upd-apply*)
moreover from *RedNotifyFail* **obtain** *va h′* **where** *P*,*t* ⊢ ⟨*a*·*M*(*vs*),*h*⟩ −*?ta′*→*ext* ⟨*va*,*h′*⟩ **by**(*fastforce*)
ultimately show *?thesis* **by** *blast*
next
case *RedNotifyAll*
note *ta* = ‹*ta* = {⌊*NotifyAll a*, *Unlock*→*a*, *Lock*→*a*⌋}›
let *?ta′* = {⌊*UnlockFail*→*a*⌋}
from *ta False None* **have** ¬ *has-lock* (*locks s* $ *a*) *t*
by(*auto simp add*: *lock-actions-ok′-iff finfun-upd-apply wset-actions-ok-def Cons-eq-append-conv split*: *if-split-asm dest*: *may-lock-t-may-lock-unlock-lock-t*)
hence *final-thread.actions-ok′ s t ?ta′* **using** *None*
by(*auto simp add*: *lock-actions-ok′-iff finfun-upd-apply*)
moreover from *ta* **have** *final-thread.actions-subset ?ta′ ta*
by(*auto simp add*: *collect-locks′-def finfun-upd-apply*)
moreover from *RedNotifyAll* **obtain** *va h′* **where** *P*,*t* ⊢ ⟨*a*·*M*(*vs*),*h*⟩ −*?ta′*→*ext* ⟨*va*,*h′*⟩ **by**(*fastforce*)
ultimately show *?thesis* **by** *blast*
next
case *RedNotifyAllFail*
note *ta* = ‹*ta* = {⌊*UnlockFail*→*a*⌋}›
let *?ta′* = {⌊*NotifyAll a*, *Unlock*→*a*, *Lock*→*a*⌋}
from *ta False None* **have** *has-lock* (*locks s* $ *a*) *t*
by(*auto simp add*: *finfun-upd-apply split*: *if-split-asm*)
hence *final-thread.actions-ok′ s t ?ta′* **using** *None*
by(*auto simp add*: *finfun-upd-apply wset-actions-ok-def intro*: *has-lock-may-lock*)
moreover from *ta* **have** *final-thread.actions-subset ?ta′ ta*
by(*auto simp add*: *collect-locks′-def finfun-upd-apply*)
moreover from *RedNotifyAllFail* **obtain** *va h′* **where** *P*,*t* ⊢ ⟨*a*·*M*(*vs*),*h*⟩ −*?ta′*→*ext* ⟨*va*,*h′*⟩

by(*fastforce*)
ultimately show *?thesis* **by** *blast*
next
case *RedInterruptedTrue*
let *?ta′* = {⟦*IsInterrupted t False*⟧}
from *RedInterruptedTrue* **have** *final-thread.actions-ok final s t ?ta′*
using *None False* **by**(*auto simp add*: *final-thread.actions-ok-iff final-thread.cond-action-oks.simps*)
moreover obtain *va h′* **where** *P*,*t* ⊢ ⟨*a*·*M*(*vs*),*h*⟩ −*?ta′*→*ext* ⟨*va*,*h′*⟩
using *RedInterruptedFalse RedInterruptedTrue* **by** *auto*
ultimately show *?thesis* **by** *blast*
next
case *RedInterruptedFalse*
let *?ta′* = {⟦*IsInterrupted t True*, *ClearInterrupt t*, *ObsInterrupted t*⟧}
from *RedInterruptedFalse* **have** *final-thread.actions-ok final s t ?ta′*
using *None False*
by(*auto simp add*: *final-thread.actions-ok-iff final-thread.cond-action-oks.simps*)
moreover obtain *va h′* **where** *P*,*t* ⊢ ⟨*a*·*M*(*vs*),*h*⟩ −*?ta′*→*ext* ⟨*va*,*h′*⟩
using *RedInterruptedFalse RedInterruptedTrue* **by** *auto*
ultimately show *?thesis* **by** *blast*
qed(*auto simp add*: *None*)
qed
qed
qed

end

context *heap-base* **begin**

lemma *red-external-ta-satisfiable*:
fixes *final*
assumes *P*,*t* ⊢ ⟨*a*·*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h′*⟩
shows ∃ *s*. *final-thread.actions-ok final s t ta*
proof −
note [*simp*] =
final-thread.actions-ok-iff final-thread.cond-action-oks.simps final-thread.cond-action-ok.simps
lock-ok-las-def finfun-upd-apply wset-actions-ok-def has-lock-may-lock
and [*intro*] =
free-thread-id.intros
and [*cong*] = *conj-cong*

from *assms* **show** *?thesis* **by** *cases*(*fastforce intro*: *exI*[**where** *x*=(*K*$ *None*)(*a* $:= ⌊(*t*, *0*)⌋)] *exI*[**where** *x*=(*K*$ *None*)])+
qed

lemma *red-external-aggr-ta-satisfiable*:
fixes *final*
assumes (*ta*, *va*, *h′*) ∈ *red-external-aggr P t a M vs h*
shows ∃ *s*. *final-thread.actions-ok final s t ta*
proof −
note [*simp*] =
final-thread.actions-ok-iff final-thread.cond-action-oks.simps final-thread.cond-action-ok.simps
lock-ok-las-def finfun-upd-apply wset-actions-ok-def has-lock-may-lock
and [*intro*] =
free-thread-id.intros

and [*cong*] = *conj-cong*

from *assms* **show** *?thesis*
by(*fastforce simp add*: *red-external-aggr-def split-beta ta-upd-simps split*: *if-split-asm intro*: *exI*[**where** *x*=*Map.empty*] *exI*[**where** *x*=(*K*$ *None*)(*a* $:= ⌊(*t*, *0*)⌋)] *exI*[**where** *x*=*K*$ *None*])
qed

end

3.13.4 Determinism

context *heap-base* **begin**

lemma *heap-copy-loc-deterministic*:
assumes *det*: *deterministic-heap-ops*
and *copy*: *heap-copy-loc a a′ al h ops h′ heap-copy-loc a a′ al h ops′ h″*
shows *ops* = *ops′* ∧ *h′* = *h″*
using *copy*
by(*auto elim*!: *heap-copy-loc.cases dest*: *deterministic-heap-ops-readD*[*OF det*] *deterministic-heap-ops-writeD*[*OF det*])

lemma *heap-copies-deterministic*:
assumes *det*: *deterministic-heap-ops*
and *copy*: *heap-copies a a′ als h ops h′ heap-copies a a′ als h ops′ h″*
shows *ops* = *ops′* ∧ *h′* = *h″*
using *copy*
apply(*induct arbitrary*: *ops′ h″*)
apply(*fastforce elim*!: *heap-copies-cases*)
apply(*erule heap-copies-cases*)
apply *clarify*
apply(*drule* (*1*) *heap-copy-loc-deterministic*[*OF det*])
apply *clarify*
apply(*unfold same-append-eq*)
apply *blast*
done

lemma *heap-clone-deterministic*:
assumes *det*: *deterministic-heap-ops*
and *clone*: *heap-clone P h a h′ obs heap-clone P h a h″ obs′*
shows *h′* = *h″* ∧ *obs* = *obs′*
using *clone*
by(*auto 4 4 elim*!: *heap-clone.cases dest*: *heap-copies-deterministic*[*OF det*] *deterministic-heap-ops-allocateD*[*OF det*] *has-fields-fun*)

lemma *red-external-deterministic*:
fixes *final*
assumes *det*: *deterministic-heap-ops*
and *red*: *P*,*t* ⊢ ⟨*a*·*M*(*vs*), (*shr s*)⟩ −*ta*→*ext* ⟨*va*, *h′*⟩ *P*,*t* ⊢ ⟨*a*·*M*(*vs*), (*shr s*)⟩ −*ta′*→*ext* ⟨*va′*, *h″*⟩
and *aok*: *final-thread.actions-ok final s t ta final-thread.actions-ok final s t ta′*
shows *ta* = *ta′* ∧ *va* = *va′* ∧ *h′* = *h″*
using *red aok*
apply(*simp add*: *final-thread.actions-ok-iff lock-ok-las-def*)
apply(*erule red-external.cases*)
apply(*erule-tac* [!] *red-external.cases*)

apply *simp-all*
apply(*auto simp add*: *finfun-upd-apply wset-actions-ok-def dest*: *heap-clone-deterministic*[*OF det*] *split*: *if-split-asm*)
using *deterministic-heap-ops-no-spurious-wakeups*[*OF det*]
apply *simp-all*
done

end

end

3.14 Conformance for threads

theory *ConformThreaded*
imports
../Framework/FWLifting
../Framework/FWWellform
Conform
begin

Every thread must be represented as an object whose address is its thread ID

context *heap-base* **begin**

abbreviation *thread-conf* :: *'m prog* ⇒ (*'addr*,*'thread-id*,*'x*) *thread-info* ⇒ *'heap* ⇒ *bool*
where *thread-conf P* ≡ *ts-ok* (λ*t x m*. *P*,*m* ⊢ *t* √*t*)

lemma *thread-confI*:
(⋀*t xln*. *ts t* = ⌊*xln*⌋ ⟹ *P*,*h* ⊢ *t* √*t*) ⟹ *thread-conf P ts h*
by(*blast intro*: *ts-okI*)

lemma *thread-confD*:
assumes *thread-conf P ts h ts t* = ⌊*xln*⌋
shows *P*,*h* ⊢ *t* √*t*
using *assms* **by**(*cases xln*)(*auto dest*: *ts-okD*)

lemma *thread-conf-ts-upd-eq* [*simp*]:
assumes *tst*: *ts t* = ⌊*xln*⌋
shows *thread-conf P* (*ts*(*t* ↦ *xln'*)) *h* ⟷ *thread-conf P ts h*
proof
assume *tc*: *thread-conf P* (*ts*(*t* ↦ *xln'*)) *h*
show *thread-conf P ts h*
proof(*rule thread-confI*)
fix *T XLN*
assume *ts T* = ⌊*XLN*⌋
with *tc* **show** *P*,*h* ⊢ *T* √*t*
by(*cases T* = *t*)(*auto dest*: *thread-confD*)
qed
next
assume *tc*: *thread-conf P ts h*
show *thread-conf P* (*ts*(*t* ↦ *xln'*)) *h*
proof(*rule thread-confI*)
fix *T XLN*
assume (*ts*(*t* ↦ *xln'*)) *T* = ⌊*XLN*⌋
with *tst* **obtain** *XLN'* **where** *ts T* = ⌊*XLN'*⌋

by(*cases* $T = t$)(*auto*)
with *tc* **show** $P,h \vdash T\ \surd t$
by(*auto dest*: *thread-confD*)
qed
qed

end

context *heap* **begin**

lemma *thread-conf-hext*:
$[\![$ *thread-conf P ts h*; $h \trianglelefteq h'$ $]\!] \Longrightarrow$ *thread-conf P ts* h'
by(*blast intro*: *thread-confI tconf-hext-mono dest*: *thread-confD*)

lemma *thread-conf-start-state*:
$[\![$ *start-heap-ok*; *wf-syscls P* $]\!] \Longrightarrow$ *thread-conf P* (*thr* (*start-state f P C M vs*)) (*shr* (*start-state f P C M vs*))
by(*auto intro*!: *thread-confI simp add*: *start-state-def split-beta split*: *if-split-asm intro*: *tconf-start-heap-start-tid*)

end

context *heap-base* **begin**

lemma *lock-thread-ok-start-state*:
lock-thread-ok (*locks* (*start-state f P C M vs*)) (*thr* (*start-state f P C M vs*))
by(*rule lock-thread-okI*)(*simp add*: *start-state-def split-beta*)

lemma *wset-thread-ok-start-state*:
wset-thread-ok (*wset* (*start-state f P C M vs*)) (*thr* (*start-state f P C M vs*))
by(*auto simp add*: *wset-thread-ok-def start-state-def split-beta*)

lemma *wset-final-ok-start-state*:
final-thread.wset-final-ok final (*wset* (*start-state f P C M vs*)) (*thr* (*start-state f P C M vs*))
by(*rule final-thread.wset-final-okI*)(*simp add*: *start-state-def split-beta*)

end

end

3.15 Binary Operators

theory *BinOp*
imports
WellForm Word-Lib.Bit-Shifts-Infix-Syntax
begin

datatype *bop* = — names of binary operations
Eq
| *NotEq*
| *LessThan*
| *LessOrEqual*
| *GreaterThan*
| *GreaterOrEqual*

| *Add*
| *Subtract*
| *Mult*
| *Div*
| *Mod*
| *BinAnd*
| *BinOr*
| *BinXor*
| *ShiftLeft*
| *ShiftRightZeros*
| *ShiftRightSigned*

3.15.1 The semantics of binary operators

context
includes *bit-operations-syntax*
begin

type-synonym *'addr binop-ret* = *'addr val* + *'addr* — a value or the address of an exception

fun *binop-LessThan* :: *'addr val* ⇒ *'addr val* ⇒ *'addr binop-ret option*
where
binop-LessThan (*Intg i1*) (*Intg i2*) = *Some* (*Inl* (*Bool* (*i1* <*s i2*)))
| *binop-LessThan v1 v2* = *None*

fun *binop-LessOrEqual* :: *'addr val* ⇒ *'addr val* ⇒ *'addr binop-ret option*
where
binop-LessOrEqual (*Intg i1*) (*Intg i2*) = *Some* (*Inl* (*Bool* (*i1* <=*s i2*)))
| *binop-LessOrEqual v1 v2* = *None*

fun *binop-GreaterThan* :: *'addr val* ⇒ *'addr val* ⇒ *'addr binop-ret option*
where
binop-GreaterThan (*Intg i1*) (*Intg i2*) = *Some* (*Inl* (*Bool* (*i2* <*s i1*)))
| *binop-GreaterThan v1 v2* = *None*

fun *binop-GreaterOrEqual* :: *'addr val* ⇒ *'addr val* ⇒ *'addr binop-ret option*
where
binop-GreaterOrEqual (*Intg i1*) (*Intg i2*) = *Some* (*Inl* (*Bool* (*i2* <=*s i1*)))
| *binop-GreaterOrEqual v1 v2* = *None*

fun *binop-Add* :: *'addr val* ⇒ *'addr val* ⇒ *'addr binop-ret option*
where
binop-Add (*Intg i1*) (*Intg i2*) = *Some* (*Inl* (*Intg* (*i1* + *i2*)))
| *binop-Add v1 v2* = *None*

fun *binop-Subtract* :: *'addr val* ⇒ *'addr val* ⇒ *'addr binop-ret option*
where
binop-Subtract (*Intg i1*) (*Intg i2*) = *Some* (*Inl* (*Intg* (*i1* − *i2*)))
| *binop-Subtract v1 v2* = *None*

fun *binop-Mult* :: *'addr val* ⇒ *'addr val* ⇒ *'addr binop-ret option*
where
binop-Mult (*Intg i1*) (*Intg i2*) = *Some* (*Inl* (*Intg* (*i1* ∗ *i2*)))
| *binop-Mult v1 v2* = *None*

fun *binop-BinAnd* :: *'addr val* ⇒ *'addr val* ⇒ *'addr binop-ret option*
where
  binop-BinAnd (*Intg i1*) (*Intg i2*) = *Some* (*Inl* (*Intg* (*i1 AND i2*)))
| *binop-BinAnd* (*Bool b1*) (*Bool b2*) = *Some* (*Inl* (*Bool* (*b1* ∧ *b2*)))
| *binop-BinAnd v1 v2* = *None*

fun *binop-BinOr* :: *'addr val* ⇒ *'addr val* ⇒ *'addr binop-ret option*
where
  binop-BinOr (*Intg i1*) (*Intg i2*) = *Some* (*Inl* (*Intg* (*i1 OR i2*)))
| *binop-BinOr* (*Bool b1*) (*Bool b2*) = *Some* (*Inl* (*Bool* (*b1* ∨ *b2*)))
| *binop-BinOr v1 v2* = *None*

fun *binop-BinXor* :: *'addr val* ⇒ *'addr val* ⇒ *'addr binop-ret option*
where
  binop-BinXor (*Intg i1*) (*Intg i2*) = *Some* (*Inl* (*Intg* (*i1 XOR i2*)))
| *binop-BinXor* (*Bool b1*) (*Bool b2*) = *Some* (*Inl* (*Bool* (*b1* ≠ *b2*)))
| *binop-BinXor v1 v2* = *None*

fun *binop-ShiftLeft* :: *'addr val* ⇒ *'addr val* ⇒ *'addr binop-ret option*
where
  binop-ShiftLeft (*Intg i1*) (*Intg i2*) = *Some* (*Inl* (*Intg* (*i1* << *unat* (*i2 AND 0x1f*))))
| *binop-ShiftLeft v1 v2* = *None*

fun *binop-ShiftRightZeros* :: *'addr val* ⇒ *'addr val* ⇒ *'addr binop-ret option*
where
  binop-ShiftRightZeros (*Intg i1*) (*Intg i2*) = *Some* (*Inl* (*Intg* (*i1* >> *unat* (*i2 AND 0x1f*))))
| *binop-ShiftRightZeros v1 v2* = *None*

fun *binop-ShiftRightSigned* :: *'addr val* ⇒ *'addr val* ⇒ *'addr binop-ret option*
where
  binop-ShiftRightSigned (*Intg i1*) (*Intg i2*) = *Some* (*Inl* (*Intg* (*i1* >>> *unat* (*i2 AND 0x1f*))))
| *binop-ShiftRightSigned v1 v2* = *None*

Division on *'a word* is unsigned, but JLS specifies signed division.

definition *word-sdiv* :: *'a* :: *len word* ⇒ *'a word* ⇒ *'a word* (**infixl** ‹*sdiv*› *70*)
where [*code*]:
  x sdiv y =
   (*let x'* = *sint x*; *y'* = *sint y*;
        negative = (*x'* < *0*) ≠ (*y'* < *0*);
        result = *abs x'* *div* *abs y'*
    in word-of-int (*if negative then* −*result else result*))

definition *word-smod* :: *'a* :: *len word* ⇒ *'a word* ⇒ *'a word* (**infixl** ‹*smod*› *70*)
where [*code*]:
  x smod y =
   (*let x'* = *sint x*; *y'* = *sint y*;
        negative = (*x'* < *0*);
        result = *abs x'* *mod* *abs y'*
    in word-of-int (*if negative then* −*result else result*))

declare *word-sdiv-def* [*simp*] *word-smod-def* [*simp*]

lemma *sdiv-smod-id*: (*a sdiv b*) ∗ *b* + (*a smod b*) = *a*

```
proof −
  have F5: ∀ u::'a word. − (− u) = u
    by simp
  have F7: ∀ v u::'a word. u + v = v + u
    by (simp add: ac-simps)
  have F8: ∀ (w::'a word) (v::int) u::int. word-of-int u + word-of-int v * w = word-of-int (u + v *
sint w)
    by simp
  have ∃ u. u = − sint b ∧ word-of-int (sint a mod u + − (− u * (sint a div u))) = a
    using F5 by simp
  hence word-of-int (sint a mod − sint b + − (sint b * (sint a div − sint b))) = a
    by (metis equation-minus-iff)
  hence word-of-int (sint a mod − sint b) + word-of-int (− (sint a div − sint b)) * b = a
    using F8 by (simp add: ac-simps)
  hence eq: word-of-int (− (sint a div − sint b)) * b + word-of-int (sint a mod − sint b) = a
    using F7 by simp

  show ?thesis
  proof(cases sint a < 0)
    case True note a = this
    show ?thesis
    proof(cases sint b < 0)
      case True
      with a show ?thesis
        by simp (metis F7 F8 eq minus-equation-iff minus-mult-minus mod-div-mult-eq)
    next
      case False
      from eq have word-of-int (− (− sint a div sint b)) * b + word-of-int (− (− sint a mod sint b))
= a
        by (metis div-minus-right mod-minus-right)
      with a False show ?thesis by simp
    qed
  next
    case False note a = this
    show ?thesis
    proof(cases sint b < 0)
      case True
      with a eq show ?thesis by simp
    next
      case False with a show ?thesis
        by (simp add: F7 F8)
    qed
  qed
qed

end

notepad begin
have    5  sdiv ( 3 :: word32) =  1
  and   5  smod ( 3 :: word32) =  2
  and   5  sdiv (−3 :: word32) = −1
  and   5  smod (−3 :: word32) =  2
  and (−5) sdiv ( 3 :: word32) = −1
  and (−5) smod ( 3 :: word32) = −2
```

```
  and (−5) sdiv (−3 :: word32) =  1
  and (−5) smod (−3 :: word32) = −2
  and −2147483648 sdiv 1 = (−2147483648 :: word32)
  by eval+
end

context heap-base
begin

fun binop-Mod :: 'addr val ⇒ 'addr val ⇒ 'addr binop-ret option
where
  binop-Mod (Intg i1) (Intg i2) =
   Some (if i2 = 0 then Inr (addr-of-sys-xcpt ArithmeticException) else Inl (Intg (i1 smod i2)))
| binop-Mod v1 v2 = None

fun binop-Div :: 'addr val ⇒ 'addr val ⇒ 'addr binop-ret option
where
  binop-Div (Intg i1) (Intg i2) =
   Some (if i2 = 0 then Inr (addr-of-sys-xcpt ArithmeticException) else Inl (Intg (i1 sdiv i2)))
| binop-Div v1 v2 = None

primrec binop :: bop ⇒ 'addr val ⇒ 'addr val ⇒ 'addr binop-ret option
where
  binop Eq v1 v2 =  Some (Inl (Bool (v1 = v2)))
| binop NotEq v1 v2 = Some (Inl (Bool (v1 ≠ v2)))
| binop LessThan = binop-LessThan
| binop LessOrEqual = binop-LessOrEqual
| binop GreaterThan = binop-GreaterThan
| binop GreaterOrEqual = binop-GreaterOrEqual
| binop Add = binop-Add
| binop Subtract = binop-Subtract
| binop Mult = binop-Mult
| binop Mod = binop-Mod
| binop Div = binop-Div
| binop BinAnd = binop-BinAnd
| binop BinOr = binop-BinOr
| binop BinXor = binop-BinXor
| binop ShiftLeft = binop-ShiftLeft
| binop ShiftRightZeros = binop-ShiftRightZeros
| binop ShiftRightSigned = binop-ShiftRightSigned

end

context
  includes bit-operations-syntax
begin

lemma [simp]:
  (binop-LessThan v1 v2 = Some va) ⟷
   (∃ i1 i2. v1 = Intg i1 ∧ v2 = Intg i2 ∧ va = Inl (Bool (i1 <s i2)))
by(cases (v1, v2) rule: binop-LessThan.cases) auto

lemma [simp]:
  (binop-LessOrEqual v1 v2 = Some va) ⟷
```

$(\exists\, i1\ i2.\ v1 = Intg\ i1 \wedge v2 = Intg\ i2 \wedge va = Inl\ (Bool\ (i1 <=s\ i2)))$
by(*cases* (*v1*, *v2*) *rule*: *binop-LessOrEqual.cases*) *auto*

lemma [*simp*]:
$(binop\text{-}GreaterThan\ v1\ v2 = Some\ va) \longleftrightarrow$
$(\exists\, i1\ i2.\ v1 = Intg\ i1 \wedge v2 = Intg\ i2 \wedge va = Inl\ (Bool\ (i2 <s\ i1)))$
by(*cases* (*v1*, *v2*) *rule*: *binop-GreaterThan.cases*) *auto*

lemma [*simp*]:
$(binop\text{-}GreaterOrEqual\ v1\ v2 = Some\ va) \longleftrightarrow$
$(\exists\, i1\ i2.\ v1 = Intg\ i1 \wedge v2 = Intg\ i2 \wedge va = Inl\ (Bool\ (i2 <=s\ i1)))$
by(*cases* (*v1*, *v2*) *rule*: *binop-GreaterOrEqual.cases*) *auto*

lemma [*simp*]:
$(binop\text{-}Add\ v_1\ v_2\ = Some\ va) \longleftrightarrow$
$(\exists\, i_1\ i_2.\ v_1 = Intg\ i_1 \wedge v_2 = Intg\ i_2 \wedge va = Inl\ (Intg\ (i_1{+}i_2)))$
by(*cases* (v_1, v_2) *rule*: *binop-Add.cases*) *auto*

lemma [*simp*]:
$(binop\text{-}Subtract\ v1\ v2 = Some\ va) \longleftrightarrow$
$(\exists\, i1\ i2.\ v1 = Intg\ i1 \wedge v2 = Intg\ i2 \wedge va = Inl\ (Intg\ (i1 - i2)))$
by(*cases* (*v1*, *v2*) *rule*: *binop-Subtract.cases*) *auto*

lemma [*simp*]:
$(binop\text{-}Mult\ v1\ v2 = Some\ va) \longleftrightarrow$
$(\exists\, i1\ i2.\ v1 = Intg\ i1 \wedge v2 = Intg\ i2 \wedge va = Inl\ (Intg\ (i1 * i2)))$
by(*cases* (*v1*, *v2*) *rule*: *binop-Mult.cases*) *auto*

lemma [*simp*]:
$(binop\text{-}BinAnd\ v1\ v2 = Some\ va) \longleftrightarrow$
$(\exists\, b1\ b2.\ v1 = Bool\ b1 \wedge v2 = Bool\ b2 \wedge va = Inl\ (Bool\ (b1 \wedge b2))) \vee$
$(\exists\, i1\ i2.\ v1 = Intg\ i1 \wedge v2 = Intg\ i2 \wedge va = Inl\ (Intg\ (i1\ AND\ i2)))$
by(*cases* (*v1*, *v2*) *rule*: *binop-BinAnd.cases*) *auto*

lemma [*simp*]:
$(binop\text{-}BinOr\ v1\ v2 = Some\ va) \longleftrightarrow$
$(\exists\, b1\ b2.\ v1 = Bool\ b1 \wedge v2 = Bool\ b2 \wedge va = Inl\ (Bool\ (b1 \vee b2))) \vee$
$(\exists\, i1\ i2.\ v1 = Intg\ i1 \wedge v2 = Intg\ i2 \wedge va = Inl\ (Intg\ (i1\ OR\ i2)))$
by(*cases* (*v1*, *v2*) *rule*: *binop-BinOr.cases*) *auto*

lemma [*simp*]:
$(binop\text{-}BinXor\ v1\ v2 = Some\ va) \longleftrightarrow$
$(\exists\, b1\ b2.\ v1 = Bool\ b1 \wedge v2 = Bool\ b2 \wedge va = Inl\ (Bool\ (b1 \neq b2))) \vee$
$(\exists\, i1\ i2.\ v1 = Intg\ i1 \wedge v2 = Intg\ i2 \wedge va = Inl\ (Intg\ (i1\ XOR\ i2)))$
by(*cases* (*v1*, *v2*) *rule*: *binop-BinXor.cases*) *auto*

lemma [*simp*]:
$(binop\text{-}ShiftLeft\ v1\ v2 = Some\ va) \longleftrightarrow$
$(\exists\, i1\ i2.\ v1 = Intg\ i1 \wedge v2 = Intg\ i2 \wedge va = Inl\ (Intg\ (i1 << unat\ (i2\ AND\ 0x1f))))$
by(*cases* (*v1*, *v2*) *rule*: *binop-ShiftLeft.cases*) *auto*

lemma [*simp*]:
$(binop\text{-}ShiftRightZeros\ v1\ v2 = Some\ va) \longleftrightarrow$
$(\exists\, i1\ i2.\ v1 = Intg\ i1 \wedge v2 = Intg\ i2 \wedge va = Inl\ (Intg\ (i1 >> unat\ (i2\ AND\ 0x1f))))$

by(*cases* (*v1*, *v2*) *rule*: *binop-ShiftRightZeros.cases*) *auto*

lemma [*simp*]:
(*binop-ShiftRightSigned v1 v2* = *Some va*) ⟷
(∃ *i1 i2*. *v1* = *Intg i1* ∧ *v2* = *Intg i2* ∧ *va* = *Inl* (*Intg* (*i1* >>> *unat* (*i2 AND 0x1f*))))
by(*cases* (*v1*, *v2*) *rule*: *binop-ShiftRightSigned.cases*) *auto*

end

context *heap-base*
begin

lemma [*simp*]:
(*binop-Mod v1 v2* = *Some va*) ⟷
(∃ *i1 i2*. *v1* = *Intg i1* ∧ *v2* = *Intg i2* ∧
va = (*if i2* = *0 then Inr* (*addr-of-sys-xcpt ArithmeticException*) *else Inl* (*Intg*(*i1 smod i2*))))
by(*cases* (*v1*, *v2*) *rule*: *binop-Mod.cases*) *auto*

lemma [*simp*]:
(*binop-Div v1 v2* = *Some va*) ⟷
(∃ *i1 i2*. *v1* = *Intg i1* ∧ *v2* = *Intg i2* ∧
va = (*if i2* = *0 then Inr* (*addr-of-sys-xcpt ArithmeticException*) *else Inl* (*Intg*(*i1 sdiv i2*))))
by(*cases* (*v1*, *v2*) *rule*: *binop-Div.cases*) *auto*

end

3.15.2 Typing for binary operators

inductive *WT-binop* :: *'m prog* ⇒ *ty* ⇒ *bop* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹- ⊢ -«-»- :: -› [*51*,*0*,*0*,*0*,*51*] *50*)
where
WT-binop-Eq:
P ⊢ *T1* ≤ *T2* ∨ *P* ⊢ *T2* ≤ *T1* ⟹ *P* ⊢ *T1*«*Eq*»*T2* :: *Boolean*

| *WT-binop-NotEq*:
P ⊢ *T1* ≤ *T2* ∨ *P* ⊢ *T2* ≤ *T1* ⟹ *P* ⊢ *T1*«*NotEq*»*T2* :: *Boolean*

| *WT-binop-LessThan*:
P ⊢ *Integer*«*LessThan*»*Integer* :: *Boolean*

| *WT-binop-LessOrEqual*:
P ⊢ *Integer*«*LessOrEqual*»*Integer* :: *Boolean*

| *WT-binop-GreaterThan*:
P ⊢ *Integer*«*GreaterThan*»*Integer* :: *Boolean*

| *WT-binop-GreaterOrEqual*:
P ⊢ *Integer*«*GreaterOrEqual*»*Integer* :: *Boolean*

| *WT-binop-Add*:
P ⊢ *Integer*«*Add*»*Integer* :: *Integer*

| *WT-binop-Subtract*:
P ⊢ *Integer*«*Subtract*»*Integer* :: *Integer*

| *WT-binop-Mult*:
  $P \vdash Integer\langle\!\langle Mult\rangle\!\rangle Integer :: Integer$

| *WT-binop-Div*:
  $P \vdash Integer\langle\!\langle Div\rangle\!\rangle Integer :: Integer$

| *WT-binop-Mod*:
  $P \vdash Integer\langle\!\langle Mod\rangle\!\rangle Integer :: Integer$

| *WT-binop-BinAnd-Bool*:
  $P \vdash Boolean\langle\!\langle BinAnd\rangle\!\rangle Boolean :: Boolean$

| *WT-binop-BinAnd-Int*:
  $P \vdash Integer\langle\!\langle BinAnd\rangle\!\rangle Integer :: Integer$

| *WT-binop-BinOr-Bool*:
  $P \vdash Boolean\langle\!\langle BinOr\rangle\!\rangle Boolean :: Boolean$

| *WT-binop-BinOr-Int*:
  $P \vdash Integer\langle\!\langle BinOr\rangle\!\rangle Integer :: Integer$

| *WT-binop-BinXor-Bool*:
  $P \vdash Boolean\langle\!\langle BinXor\rangle\!\rangle Boolean :: Boolean$

| *WT-binop-BinXor-Int*:
  $P \vdash Integer\langle\!\langle BinXor\rangle\!\rangle Integer :: Integer$

| *WT-binop-ShiftLeft*:
  $P \vdash Integer\langle\!\langle ShiftLeft\rangle\!\rangle Integer :: Integer$

| *WT-binop-ShiftRightZeros*:
  $P \vdash Integer\langle\!\langle ShiftRightZeros\rangle\!\rangle Integer :: Integer$

| *WT-binop-ShiftRightSigned*:
  $P \vdash Integer\langle\!\langle ShiftRightSigned\rangle\!\rangle Integer :: Integer$

lemma *WT-binopI* [*intro*]:
  $P \vdash T1 \leq T2 \vee P \vdash T2 \leq T1 \Longrightarrow P \vdash T1\langle\!\langle Eq\rangle\!\rangle T2 :: Boolean$
  $P \vdash T1 \leq T2 \vee P \vdash T2 \leq T1 \Longrightarrow P \vdash T1\langle\!\langle NotEq\rangle\!\rangle T2 :: Boolean$
  $bop = Add \vee bop = Subtract \vee bop = Mult \vee bop = Mod \vee bop = Div \vee bop = BinAnd \vee bop = BinOr \vee bop = BinXor \vee$
  $bop = ShiftLeft \vee bop = ShiftRightZeros \vee bop = ShiftRightSigned$
  $\Longrightarrow P \vdash Integer\langle\!\langle bop\rangle\!\rangle Integer :: Integer$
  $bop = LessThan \vee bop = LessOrEqual \vee bop = GreaterThan \vee bop = GreaterOrEqual \Longrightarrow P \vdash Integer\langle\!\langle bop\rangle\!\rangle Integer :: Boolean$
  $bop = BinAnd \vee bop = BinOr \vee bop = BinXor \Longrightarrow P \vdash Boolean\langle\!\langle bop\rangle\!\rangle Boolean :: Boolean$
by(*auto intro*: *WT-binop.intros*)

inductive-cases [*elim*]:
  $P \vdash T1\langle\!\langle Eq\rangle\!\rangle T2 :: T$
  $P \vdash T1\langle\!\langle NotEq\rangle\!\rangle T2 :: T$
  $P \vdash T1\langle\!\langle LessThan\rangle\!\rangle T2 :: T$
  $P \vdash T1\langle\!\langle LessOrEqual\rangle\!\rangle T2 :: T$
  $P \vdash T1\langle\!\langle GreaterThan\rangle\!\rangle T2 :: T$

$P \vdash T1 \mathbin{«GreaterOrEqual»} T2 :: T$
$P \vdash T1 \mathbin{«Add»} T2 :: T$
$P \vdash T1 \mathbin{«Subtract»} T2 :: T$
$P \vdash T1 \mathbin{«Mult»} T2 :: T$
$P \vdash T1 \mathbin{«Div»} T2 :: T$
$P \vdash T1 \mathbin{«Mod»} T2 :: T$
$P \vdash T1 \mathbin{«BinAnd»} T2 :: T$
$P \vdash T1 \mathbin{«BinOr»} T2 :: T$
$P \vdash T1 \mathbin{«BinXor»} T2 :: T$
$P \vdash T1 \mathbin{«ShiftLeft»} T2 :: T$
$P \vdash T1 \mathbin{«ShiftRightZeros»} T2 :: T$
$P \vdash T1 \mathbin{«ShiftRightSigned»} T2 :: T$

lemma *WT-binop-fun*: $[\![P \vdash T1 \mathbin{«bop»} T2 :: T;\ P \vdash T1 \mathbin{«bop»} T2 :: T']\!] \Longrightarrow T = T'$
by(*cases bop*)(*auto*)

lemma *WT-binop-is-type*:
$[\![P \vdash T1 \mathbin{«bop»} T2 :: T;\ \mathit{is\text{-}type}\ P\ T1;\ \mathit{is\text{-}type}\ P\ T2]\!] \Longrightarrow \mathit{is\text{-}type}\ P\ T$
by(*cases bop*) *auto*

inductive *WTrt-binop* :: *'m prog* ⇒ *ty* ⇒ *bop* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹- ⊢ -«-»- : -› [*51*,*0*,*0*,*0*,*51*] *50*)
where
WTrt-binop-Eq:
$P \vdash T1 \mathbin{«Eq»} T2 : \mathit{Boolean}$

| *WTrt-binop-NotEq*:
$P \vdash T1 \mathbin{«NotEq»} T2 : \mathit{Boolean}$

| *WTrt-binop-LessThan*:
$P \vdash \mathit{Integer} \mathbin{«LessThan»} \mathit{Integer} : \mathit{Boolean}$

| *WTrt-binop-LessOrEqual*:
$P \vdash \mathit{Integer} \mathbin{«LessOrEqual»} \mathit{Integer} : \mathit{Boolean}$

| *WTrt-binop-GreaterThan*:
$P \vdash \mathit{Integer} \mathbin{«GreaterThan»} \mathit{Integer} : \mathit{Boolean}$

| *WTrt-binop-GreaterOrEqual*:
$P \vdash \mathit{Integer} \mathbin{«GreaterOrEqual»} \mathit{Integer} : \mathit{Boolean}$

| *WTrt-binop-Add*:
$P \vdash \mathit{Integer} \mathbin{«Add»} \mathit{Integer} : \mathit{Integer}$

| *WTrt-binop-Subtract*:
$P \vdash \mathit{Integer} \mathbin{«Subtract»} \mathit{Integer} : \mathit{Integer}$

| *WTrt-binop-Mult*:
$P \vdash \mathit{Integer} \mathbin{«Mult»} \mathit{Integer} : \mathit{Integer}$

| *WTrt-binop-Div*:
$P \vdash \mathit{Integer} \mathbin{«Div»} \mathit{Integer} : \mathit{Integer}$

| *WTrt-binop-Mod*:
$P \vdash \mathit{Integer} \mathbin{«Mod»} \mathit{Integer} : \mathit{Integer}$

| *WTrt-binop-BinAnd-Bool*:
  $P \vdash Boolean\langle\!\langle BinAnd\rangle\!\rangle Boolean : Boolean$

| *WTrt-binop-BinAnd-Int*:
  $P \vdash Integer\langle\!\langle BinAnd\rangle\!\rangle Integer : Integer$

| *WTrt-binop-BinOr-Bool*:
  $P \vdash Boolean\langle\!\langle BinOr\rangle\!\rangle Boolean : Boolean$

| *WTrt-binop-BinOr-Int*:
  $P \vdash Integer\langle\!\langle BinOr\rangle\!\rangle Integer : Integer$

| *WTrt-binop-BinXor-Bool*:
  $P \vdash Boolean\langle\!\langle BinXor\rangle\!\rangle Boolean : Boolean$

| *WTrt-binop-BinXor-Int*:
  $P \vdash Integer\langle\!\langle BinXor\rangle\!\rangle Integer : Integer$

| *WTrt-binop-ShiftLeft*:
  $P \vdash Integer\langle\!\langle ShiftLeft\rangle\!\rangle Integer : Integer$

| *WTrt-binop-ShiftRightZeros*:
  $P \vdash Integer\langle\!\langle ShiftRightZeros\rangle\!\rangle Integer : Integer$

| *WTrt-binop-ShiftRightSigned*:
  $P \vdash Integer\langle\!\langle ShiftRightSigned\rangle\!\rangle Integer : Integer$

lemma *WTrt-binopI* [*intro*]:
  $P \vdash T1\langle\!\langle Eq\rangle\!\rangle T2 : Boolean$
  $P \vdash T1\langle\!\langle NotEq\rangle\!\rangle T2 : Boolean$
  $bop = Add \vee bop = Subtract \vee bop = Mult \vee bop = Div \vee bop = Mod \vee bop = BinAnd \vee bop = BinOr \vee bop = BinXor \vee$
   $bop = ShiftLeft \vee bop = ShiftRightZeros \vee bop = ShiftRightSigned$
   $\Longrightarrow P \vdash Integer\langle\!\langle bop\rangle\!\rangle Integer : Integer$
  $bop = LessThan \vee bop = LessOrEqual \vee bop = GreaterThan \vee bop = GreaterOrEqual \Longrightarrow P \vdash Integer\langle\!\langle bop\rangle\!\rangle Integer : Boolean$
  $bop = BinAnd \vee bop = BinOr \vee bop = BinXor \Longrightarrow P \vdash Boolean\langle\!\langle bop\rangle\!\rangle Boolean : Boolean$
by(*auto intro*: *WTrt-binop.intros*)

inductive-cases *WTrt-binop-cases* [*elim*]:
  $P \vdash T1\langle\!\langle Eq\rangle\!\rangle T2 : T$
  $P \vdash T1\langle\!\langle NotEq\rangle\!\rangle T2 : T$
  $P \vdash T1\langle\!\langle LessThan\rangle\!\rangle T2 : T$
  $P \vdash T1\langle\!\langle LessOrEqual\rangle\!\rangle T2 : T$
  $P \vdash T1\langle\!\langle GreaterThan\rangle\!\rangle T2 : T$
  $P \vdash T1\langle\!\langle GreaterOrEqual\rangle\!\rangle T2 : T$
  $P \vdash T1\langle\!\langle Add\rangle\!\rangle T2 : T$
  $P \vdash T1\langle\!\langle Subtract\rangle\!\rangle T2 : T$
  $P \vdash T1\langle\!\langle Mult\rangle\!\rangle T2 : T$
  $P \vdash T1\langle\!\langle Div\rangle\!\rangle T2 : T$
  $P \vdash T1\langle\!\langle Mod\rangle\!\rangle T2 : T$
  $P \vdash T1\langle\!\langle BinAnd\rangle\!\rangle T2 : T$
  $P \vdash T1\langle\!\langle BinOr\rangle\!\rangle T2 : T$

$P \vdash T1 \mathbin{«BinXor»} T2 : T$
$P \vdash T1 \mathbin{«ShiftLeft»} T2 : T$
$P \vdash T1 \mathbin{«ShiftRightZeros»} T2 : T$
$P \vdash T1 \mathbin{«ShiftRightSigned»} T2 : T$

inductive-simps *WTrt-binop-simps* [*simp*]:
$P \vdash T1 \mathbin{«Eq»} T2 : T$
$P \vdash T1 \mathbin{«NotEq»} T2 : T$
$P \vdash T1 \mathbin{«LessThan»} T2 : T$
$P \vdash T1 \mathbin{«LessOrEqual»} T2 : T$
$P \vdash T1 \mathbin{«GreaterThan»} T2 : T$
$P \vdash T1 \mathbin{«GreaterOrEqual»} T2 : T$
$P \vdash T1 \mathbin{«Add»} T2 : T$
$P \vdash T1 \mathbin{«Subtract»} T2 : T$
$P \vdash T1 \mathbin{«Mult»} T2 : T$
$P \vdash T1 \mathbin{«Div»} T2 : T$
$P \vdash T1 \mathbin{«Mod»} T2 : T$
$P \vdash T1 \mathbin{«BinAnd»} T2 : T$
$P \vdash T1 \mathbin{«BinOr»} T2 : T$
$P \vdash T1 \mathbin{«BinXor»} T2 : T$
$P \vdash T1 \mathbin{«ShiftLeft»} T2 : T$
$P \vdash T1 \mathbin{«ShiftRightZeros»} T2 : T$
$P \vdash T1 \mathbin{«ShiftRightSigned»} T2 : T$

fun *binop-relevant-class* :: *bop* ⇒ *'m prog* ⇒ *cname* ⇒ *bool*
where
binop-relevant-class Div = (λ*P C*. *P* ⊢ *ArithmeticException* ⪯* *C*)
| *binop-relevant-class Mod* = (λ*P C*. *P* ⊢ *ArithmeticException* ⪯* *C*)
| *binop-relevant-class* - = (λ*P C*. *False*)

lemma *WT-binop-WTrt-binop*:
$P \vdash T1 \mathbin{«bop»} T2 :: T \Longrightarrow P \vdash T1 \mathbin{«bop»} T2 : T$
by(*auto elim*: *WT-binop.cases*)

context *heap* **begin**

lemma *binop-progress*:
⟦ *typeof*$_h$ *v1* = ⌊*T1*⌋; *typeof*$_h$ *v2* = ⌊*T2*⌋; *P* ⊢ *T1* «*bop*» *T2* : *T* ⟧
⟹ ∃*va*. *binop bop v1 v2* = ⌊*va*⌋
by(*cases bop*)(*auto del*: *disjCI split del*: *if-split*)

lemma *binop-type*:
assumes *wf*: *wf-prog wf-md P*
and *pre*: *preallocated h*
and *type*: *typeof*$_h$ *v1* = ⌊*T1*⌋ *typeof*$_h$ *v2* = ⌊*T2*⌋ *P* ⊢ *T1* «*bop*» *T2* : *T*
shows *binop bop v1 v2* = ⌊*Inl v*⌋ ⟹ *P*,*h* ⊢ *v* :≤ *T*
and *binop bop v1 v2* = ⌊*Inr a*⌋ ⟹ *P*,*h* ⊢ *Addr a* :≤ *Class Throwable*
using *type*
apply(*case-tac* [!] *bop*)
apply(*auto split*: *if-split-asm simp add*: *conf-def wf-preallocatedD*[*OF wf pre*])
done

lemma *binop-relevant-class*:
assumes *wf*: *wf-prog wf-md P*

and *pre*: *preallocated h*
and *bop*: *binop bop v1 v2* = ⌊*Inr a*⌋
and *sup*: *P* ⊢ *cname-of h a* ⪯* *C*
shows *binop-relevant-class bop P C*
using *assms*
by(*cases bop*)(*auto split*: *if-split-asm*)

end

lemma *WTrt-binop-fun*: ⟦ *P* ⊢ *T1* «*bop*» *T2* : *T*; *P* ⊢ *T1* «*bop*» *T2* : *T'* ⟧ ⟹ *T* = *T'*
by(*cases bop*)(*auto*)

lemma *WTrt-binop-THE* [*simp*]: *P* ⊢ *T1* «*bop*» *T2* : *T* ⟹ *The* (*WTrt-binop P T1 bop T2*) = *T*
by(*auto dest*: *WTrt-binop-fun*)

lemma *WTrt-binop-widen-mono*:
⟦ *P* ⊢ *T1* «*bop*» *T2* : *T*; *P* ⊢ *T1'* ≤ *T1*; *P* ⊢ *T2'* ≤ *T2* ⟧ ⟹ ∃ *T'*. *P* ⊢ *T1'* «*bop*» *T2'* : *T'* ∧ *P* ⊢ *T'* ≤ *T*
by(*cases bop*)(*auto elim*!: *WTrt-binop-cases*)

lemma *WTrt-binop-is-type*:
⟦ *P* ⊢ *T1* «*bop*» *T2* : *T*; *is-type P T1*; *is-type P T2* ⟧ ⟹ *is-type P T*
by(*cases bop*) *auto*

3.15.3 Code generator setup

lemmas [*code*] =
heap-base.binop-Div.simps
heap-base.binop-Mod.simps
heap-base.binop.simps

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*)
WT-binop
.

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*)
WTrt-binop
.

lemma *eval-WTrt-binop-i-i-i-i-o*:
Predicate.eval (*WTrt-binop-i-i-i-i-o P T1 bop T2*) *T* ⟷ *P* ⊢ *T1* «*bop*» *T2* : *T*
by(*auto elim*: *WTrt-binop-i-i-i-i-oE intro*: *WTrt-binop-i-i-i-i-oI*)

lemma *the-WTrt-binop-code*:
(*THE T*. *P* ⊢ *T1* «*bop*» *T2* : *T*) = *Predicate.the* (*WTrt-binop-i-i-i-i-o P T1 bop T2*)
by(*simp add*: *Predicate.the-def eval-WTrt-binop-i-i-i-i-o*)

end

3.16 The Jinja Type System as a Semilattice

theory *SemiType*

imports
WellForm
../DFA/Semilattices
begin

inductive-set
widen1 :: *'a prog* ⇒ (*ty* × *ty*) *set*
and *widen1-syntax* :: *'a prog* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹- ⊢ - <^{1} -› [*71*,*71*,*71*] *70*)
for *P* :: *'a prog*
where
P ⊢ *C* <^{1} *D* ≡ (*C*, *D*) ∈ *widen1 P*

| *widen1-Array-Object*:
P ⊢ *Array* (*Class Object*) <^{1} *Class Object*

| *widen1-Array-Integer*:
P ⊢ *Array Integer* <^{1} *Class Object*

| *widen1-Array-Boolean*:
P ⊢ *Array Boolean* <^{1} *Class Object*

| *widen1-Array-Void*:
P ⊢ *Array Void* <^{1} *Class Object*

| *widen1-Class*:
P ⊢ *C* ≺^{1} *D* ⟹ *P* ⊢ *Class C* <^{1} *Class D*

| *widen1-Array-Array*:
⟦ *P* ⊢ *T* <^{1} *U*; ¬ *is-NT-Array T* ⟧ ⟹ *P* ⊢ *Array T* <^{1} *Array U*

abbreviation *widen1-trancl* :: *'a prog* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹- ⊢ - <$^+$ -› [*71*,*71*,*71*] *70*) **where**
P ⊢ *T* <$^+$ *U* ≡ (*T*, *U*) ∈ *trancl* (*widen1 P*)

abbreviation *widen1-rtrancl* :: *'a prog* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹- ⊢ - <* -› [*71*,*71*,*71*] *70*) **where**
P ⊢ *T* <* *U* ≡ (*T*, *U*) ∈ *rtrancl* (*widen1 P*)

inductive-simps *widen1-simps1* [*simp*]:
P ⊢ *Integer* <^{1} *T*
P ⊢ *Boolean* <^{1} *T*
P ⊢ *Void* <^{1} *T*
P ⊢ *Class Object* <^{1} *T*
P ⊢ *NT* <^{1} *U*

inductive-simps *widen1-simps* [*simp*]:
P ⊢ *Array* (*Class Object*) <^{1} *T*
P ⊢ *Array Integer* <^{1} *T*
P ⊢ *Array Boolean* <^{1} *T*
P ⊢ *Array Void* <^{1} *T*
P ⊢ *Class C* <^{1} *T*
P ⊢ *T* <^{1} *Array U*

lemma *is-type-widen1*:
assumes *icO*: *is-class P Object*
shows *P* ⊢ *T* <^{1} *U* ⟹ *is-type P T*

by(*induct rule*: *widen1.induct*)(*auto intro*: *subcls-is-class icO split*: *ty.split dest*: *is-type-ground-type*)

lemma *widen1-NT-Array*:
assumes *is-NT-Array T*
shows ¬ *P* ⊢ *T*⌊⌉ <¹ *U*
proof
assume *P* ⊢ *T*⌊⌉ <¹ *U* **thus** *False* **using** *assms*
by(*induct T*⌊⌉ *U arbitrary*: *T*) *auto*
qed

lemma *widen1-is-type*:
assumes *wfP*: *wf-prog wfmd P*
shows (*A*, *B*) ∈ *widen1 P* ⟹ *is-type P B*
proof(*induct rule*: *widen1.induct*)
case (*widen1-Class C D*)
hence *is-class P C is-class P D*
by(*auto intro*: *subcls-is-class converse-subcls-is-class*[*OF wfP*])
thus *?case* **by** *simp*
next
case (*widen1-Array-Array T U*)
thus *?case* **by**(*cases U*)(*auto elim*: *widen1.cases*)
qed(*insert wfP*, *auto*)

lemma *widen1-trancl-is-type*:
assumes *wfP*: *wf-prog wfmd P*
shows (*A*, *B*) ∈ (*widen1 P*)^+ ⟹ *is-type P B*
apply(*induct rule*: *trancl-induct*)
apply(*auto intro*: *widen1-is-type*[*OF wfP*])
done

lemma *single-valued-widen1*:
assumes *wf*: *wf-prog wf-md P*
shows *single-valued* (*widen1 P*)
proof(*rule single-valuedI*)
fix *x y z*
assume *P* ⊢ *x* <¹ *y P* ⊢ *x* <¹ *z*
thus *y* = *z*
proof(*induct arbitrary*: *z rule*: *widen1.induct*)
case *widen1-Class*
with *single-valued-subcls1*[*OF wf*] **show** *?case*
by(*auto dest*: *single-valuedpD*)
next
case (*widen1-Array-Array T U z*)
from ‹*P* ⊢ *T*⌊⌉ <¹ *z*› ‹*P* ⊢ *T* <¹ *U*› ‹¬ *is-NT-Array T*›
obtain *z'* **where** *z'*: *z* = *z'*⌊⌉ **and** *Tz'*: *P* ⊢ *T* <¹ *z'*
by(*auto elim*: *widen1.cases*)
with ‹*P* ⊢ *T* <¹ *z'* ⟹ *U* = *z'*› **have** *U* = *z'* **by** *blast*
with *z'* **show** *?case* **by** *simp*
qed *simp-all*
qed

function *inheritance-level* :: *'a prog* ⇒ *cname* ⇒ *nat* **where**
inheritance-level P C =
(*if acyclicP* (*subcls1 P*) ∧ *is-class P C* ∧ *C* ≠ *Object*

```
    then Suc (inheritance-level P (fst (the (class P C))))
    else 0)
by(pat-completeness, auto)
termination
proof(relation same-fst (λP. acyclicP (subcls1 P)) (λP. {(C, C'). (subcls1 P)^-1-1 C C'}))
  show wf (same-fst (λP. acyclicP (subcls1 P)) (λP. {(C, C'). (subcls1 P)^-1-1 C C'}))
    by(rule wf-same-fst)(rule acyclicP-wf-subcls1[unfolded wfp-def])
qed(auto simp add: is-class-def intro: subcls1I)

fun subtype-measure :: 'a prog ⇒ ty ⇒ nat where
  subtype-measure P (Class C) = inheritance-level P C
| subtype-measure P (Array T) = 1 + subtype-measure P T
| subtype-measure P T = 0

lemma subtype-measure-measure:
  assumes acyclic: acyclicP (subcls1 P)
  and widen1: P ⊢ x <1 y
  shows subtype-measure P y < subtype-measure P x
using widen1
proof(induct rule: widen1.induct)
  case (widen1-Class C D)
  then obtain rest where is-class P C C ≠ Object class P C = ⌊(D, rest)⌋
    by(auto elim!: subcls1.cases simp: is-class-def)
  thus ?case using acyclic by(simp)
qed(simp-all)

lemma wf-converse-widen1:
  assumes wfP: wf-prog wfmc P
  shows wf ((widen1 P)^-1)
proof(rule wf-subset)
  from wfP have acyclicP (subcls1 P) by(rule acyclic-subcls1)
  thus (widen1 P)^-1 ⊆ measure (subtype-measure P)
    by(auto dest: subtype-measure-measure)
qed simp

fun super :: 'a prog ⇒ ty ⇒ ty
where
  super P (Array Integer) = Class Object
| super P (Array Boolean) = Class Object
| super P (Array Void) = Class Object
| super P (Array (Class C)) = (if C = Object then Class Object else Array (super P (Class C)))
| super P (Array (Array T)) = Array (super P (Array T))
| super P (Class C) = Class (fst (the (class P C)))

lemma superI:
  P ⊢ T <1 U ⟹ super P T = U
proof(induct rule: widen1.induct)
  case (widen1-Array-Array T U)
  thus ?case by(cases T) auto
qed(auto dest: subcls1D)

lemma Class-widen1-super:
  P ⊢ Class C' <1 U' ⟷ is-class P C' ∧ C' ≠ Object ∧ U' = super P (Class C')
  (is ?lhs ⟷ ?rhs)
```

```
proof(rule iffI)
  assume ?lhs thus ?rhs
    by(auto intro: subcls-is-class simp add: superI simp del: super.simps)
next
  assume ?rhs thus ?lhs
    by(auto simp add: is-class-def intro: subcls1.intros)
qed

lemma super-widen1:
  assumes icO: is-class P Object
  shows P ⊢ T <¹ U ⟷ is-type P T ∧ (case T of Class C ⇒ (C ≠ Object ∧ U = super P T)
                                                | Array T' ⇒ U = super P T
                                                | -        ⇒ False)
proof(induct T arbitrary: U)
  case Class thus ?case using Class-widen1-super by(simp)
next
  case (Array T' U')
  note IH = this
  have P ⊢ T'⌊⌉ <¹ U' = (is-type P (T'⌊⌉) ∧ U' = super P (T'⌊⌉))
  proof(rule iffI)
    assume wd: P ⊢ T'⌊⌉ <¹ U'
    with icO have is-type P (T'⌊⌉) by(rule is-type-widen1)
    moreover from wd have super P (T'⌊⌉) = U' by(rule superI)
    ultimately show is-type P (T'⌊⌉) ∧ U' = super P (T'⌊⌉) by simp
  next
    assume is-type P (T'⌊⌉) ∧ U' = super P (T'⌊⌉)
    then obtain is-type P (T'⌊⌉) and U': U' = super P (T'⌊⌉) ..
    thus P ⊢ T'⌊⌉ <¹ U'
    proof(cases T')
      case (Class D)
      thus ?thesis using U' icO ‹is-type P (T'⌊⌉)›
        by(cases D = Object)(auto simp add: is-class-def intro: subcls1.intros)
    next
      case Array thus ?thesis
        using IH ‹is-type P (T'⌊⌉)› U' by(auto simp add: ty.split-asm)
    qed simp-all
  qed
  thus ?case by(simp)
qed(simp-all)

definition sup :: 'c prog ⇒ ty ⇒ ty ⇒ ty err where
  sup P T U ≡
   if is-refT T ∧ is-refT U
   then OK (if U = NT then T
            else if T = NT then U
            else exec-lub (widen1 P) (super P) T U)
   else if (T = U) then OK T else Err

lemma sup-def':
  sup P = (λT U.
   if is-refT T ∧ is-refT U
   then OK (if U = NT then T
            else if T = NT then U
            else exec-lub (widen1 P) (super P) T U)
```

```
   else if (T = U) then OK T else Err)
  by (simp add: fun-eq-iff sup-def)

definition esl :: 'm prog ⇒ ty esl
where
  esl P = (types P, widen P, sup P)

lemma order-widen [intro,simp]:
  wf-prog m P ⟹ order (widen P)
unfolding Semilat.order-def lesub-def
by (auto intro: widen-trans widen-antisym)

lemma subcls1-trancl-widen1-trancl:
  (subcls1 P)^++ C D ⟹ P ⊢ Class C <^+ Class D
by(induct rule: tranclp-induct[consumes 1, case-names base step])
  (auto intro: trancl-into-trancl)

lemma subcls-into-widen1-rtrancl:
  P ⊢ C ⪯* D ⟹ P ⊢ Class C <* Class D
by(induct rule: rtranclp-induct)(auto intro: rtrancl-into-rtrancl)

lemma not-widen1-NT-Array:
  P ⊢ U <¹ T ⟹ ¬ is-NT-Array T
by(induct rule: widen1.induct)(auto)

lemma widen1-trancl-into-Array-widen1-trancl:
  ⟦ P ⊢ A <^+ B; ¬ is-NT-Array A ⟧ ⟹ P ⊢ A⌊⌉ <^+ B⌊⌉
by(induct rule: converse-trancl-induct)
  (auto intro: trancl-into-trancl2 widen1-Array-Array dest: not-widen1-NT-Array)

lemma widen1-rtrancl-into-Array-widen1-rtrancl:
  ⟦ P ⊢ A <* B; ¬ is-NT-Array A ⟧ ⟹ P ⊢ A⌊⌉ <* B⌊⌉
by(blast elim: rtranclE intro: trancl-into-rtrancl widen1-trancl-into-Array-widen1-trancl rtrancl-into-trancl1)

lemma Array-Object-widen1-trancl:
  assumes wf: wf-prog wmdc P
  and itA: is-type P (A⌊⌉)
  shows P ⊢ A⌊⌉ <^+ Class Object
using itA
proof(induction A)
  case (Class C)
  hence is-class P C by simp
  hence P ⊢ C ⪯* Object by(rule subcls-C-Object[OF - wf])
  hence P ⊢ Class C <* Class Object by(rule subcls-into-widen1-rtrancl)
  hence P ⊢ Class C⌊⌉ <* Class Object⌊⌉
    by(rule widen1-rtrancl-into-Array-widen1-rtrancl) simp
  thus ?case by(rule rtrancl-into-trancl1) simp
next
  case (Array A)
  from ‹is-type P (A⌊⌉⌊⌉)› have is-type P (A⌊⌉) by(rule is-type-ArrayD)
  hence P ⊢ A⌊⌉ <^+ Class Object by(rule Array.IH)
  moreover from ‹is-type P (A⌊⌉⌊⌉)› have ¬ is-NT-Array (A⌊⌉) by auto
  ultimately have P ⊢ A⌊⌉⌊⌉ <^+ Class Object⌊⌉
    by(rule widen1-trancl-into-Array-widen1-trancl)
```

thus *?case* **by**(*rule trancl-into-trancl*) *simp*
qed *auto*

lemma *widen-into-widen1-trancl*:
assumes *wf*: *wf-prog wfmd P*
shows ⟦ $P \vdash A \leq B$; $A \neq B$; $A \neq NT$; *is-type P A* ⟧ ⟹ $P \vdash A <^+ B$
proof(*induct rule*: *widen.induct*)
case (*widen-subcls C D*)
from ‹*Class C* ≠ *Class D*› ‹$P \vdash C \preceq^* D$› **have** $(subcls1\ P)^{++}\ C\ D$
by(*auto elim*: *rtranclp.cases intro*: *rtranclp-into-tranclp1*)
thus *?case* **by**(*rule subcls1-trancl-widen1-trancl*)
next
case *widen-array-object* **thus** *?case* **by**(*auto intro*: *Array-Object-widen1-trancl*[*OF wf*])
next
case (*widen-array-array A B*)
hence $P \vdash A <^+ B$ **by**(*cases A*) *auto*
with ‹*is-type P* (*A*⌊⌉)› **show** *?case* **by**(*auto intro*: *widen1-trancl-into-Array-widen1-trancl*)
qed(*auto*)

lemma *wf-prog-impl-acc-widen*:
assumes *wfP*: *wf-prog wfmd P*
shows *acc* (*types P*) (*widen P*)
proof −
from *wf-converse-widen1*[*OF wfP*]
have *wf* (((*widen1 P*)⁻¹ − 1)⁺) **by**(*rule wf-trancl*)

hence *wfw1t*: ⋀*M T*. $T \in M \Longrightarrow (\exists z{\in}M.\ \forall y.\ (y, z) \in ((widen1\ P)^{-1})^+ \longrightarrow y \notin M)$
by(*auto simp only*: *wf-eq-minimal*)
have *wf* {(*y*, *x*). *is-type P x* ∧ *is-type P y* ∧ *widen P x y* ∧ $x \neq y$}
unfolding *wf-eq-minimal*
proof(*intro strip*)
fix *M* **and** *T* :: *ty*
assume *TM*: $T \in M$
show $\exists z{\in}M.\ \forall y.\ (y, z) \in \{(y, T).\ is\text{-}type\ P\ T \wedge is\text{-}type\ P\ y \wedge widen\ P\ T\ y \wedge T \neq y\} \longrightarrow y \notin M$
proof(*cases* (∃ *C*. *Class C* ∈ *M* ∧ *is-class P C*) ∨ (∃ *U*. *U*⌊⌉ ∈ *M* ∧ *is-type P* (*U*⌊⌉)))
case *True*
have *BNTthesis*: ⋀*B*. ⟦ *B* ∈ (*M* ∩ *types P*) − {*NT*} ⟧ ⟹ *?thesis*
proof −
fix *B*
assume *BM*: *B* ∈ *M* ∩ *types P* − {*NT*}
from *wfw1t*[*OF BM*] **obtain** *z*
where *zM*: *z* ∈ *M*
and *znnt*: *z* ≠ *NT*
and *itz*: *is-type P z*
and *y*: ⋀*y*. (*y*, *z*) ∈ ((*widen1 P*)⁻¹)⁺ ⟹ *y* ∉ *M* ∩ *types P* − {*NT*} **by** *blast*
show *?thesis B*
proof(*rule bexI*[*OF* - *zM*], *rule allI*, *rule impI*)
fix *y*
assume (*y*, *z*) ∈ {(*y*, *T*). *is-type P T* ∧ *is-type P y* ∧ *widen P T y* ∧ *T* ≠ *y*}
hence *Pzy*: $P \vdash z \leq y$ **and** *zy*: $z \neq y$ **and** *is-type P y* **by** *auto*
hence $P \vdash z <^+ y$ **using** *znnt itz*
by −(*rule widen-into-widen1-trancl*[*OF wfP*])
hence *ynM*: *y* ∉ *M* ∩ *types P* − {*NT*}
by −(*rule y*, *simp add*: *trancl-converse*)

```
        thus y ∉ M using Pzy znnt ‹is-type P y› by auto
      qed
    qed
    from True show ?thesis by(fastforce intro: BNTthesis)
  next
    case False

    hence not-is-class: ⋀C. Class C ∈ M ⟹ ¬ is-class P C
      and not-is-array: ⋀U. U⌊⌋ ∈ M ⟹ ¬ is-type P (U⌊⌋) by simp-all

    show ?thesis
    proof(cases ∃ C. Class C ∈ M)
      case True
      then obtain C where Class C ∈ M ..
      with not-is-class[of C] show ?thesis
        by(blast dest: rtranclpD subcls-is-class Class-widen)
    next
      case False
      show ?thesis
      proof(cases ∃ T. Array T ∈ M)
        case True
        then obtain U where U: Array U ∈ M ..
        hence ¬ is-type P (U⌊⌋) by(rule not-is-array)
        thus ?thesis using U by(auto simp del: is-type.simps)
      next
        case False
        with ‹¬ (∃ C. Class C ∈ M)› TM
        have ∀ y. P ⊢ T ≤ y ∧ T ≠ y ⟶ y ∉ M
          by(cases T)(fastforce simp add: NT-widen)+
        thus ?thesis using TM by blast
      qed
    qed
  qed
  thus ?thesis by(simp add: Semilat.acc-def lesssub-def lesub-def)
qed

lemmas wf-widen-acc = wf-prog-impl-acc-widen
declare wf-widen-acc [intro, simp]

lemma acyclic-widen1:
  wf-prog wfmc P ⟹ acyclic (widen1 P)
by(auto dest: wf-converse-widen1 wf-acyclic simp add: acyclic-converse)

lemma widen1-into-widen:
  (A, B) ∈ widen1 P ⟹ P ⊢ A ≤ B
by(induct rule: widen1.induct)(auto intro: widen.intros)

lemma widen1-rtrancl-into-widen:
  P ⊢ A <* B ⟹ P ⊢ A ≤ B
by(induct rule: rtrancl-induct)(auto dest!: widen1-into-widen elim: widen-trans)

lemma widen-eq-widen1-trancl:
  ⟦ wf-prog wf-md P; T ≠ NT; T ≠ U; is-type P T ⟧ ⟹ P ⊢ T ≤ U ⟷ P ⊢ T <+ U
```

by(*blast intro*: *widen-into-widen1-trancl widen1-rtrancl-into-widen trancl-into-rtrancl*)

lemma *sup-is-type*:
assumes *wf*: *wf-prog wf-md P*
and *itA*: *is-type P A*
and *itB*: *is-type P B*
and *sup*: *sup P A B* = *OK T*
shows *is-type P T*
proof −
{ **assume** *ANT*: $A \neq NT$
and *BNT*: $B \neq NT$
and *AnB*: $A \neq B$
and *RTA*: *is-refT A*
and *RTB*: *is-refT B*
with *itA itB* **have** *AObject*: $P \vdash A \leq$ *Class Object*
and *BObject*: $P \vdash B \leq$ *Class Object*
by(*auto intro*: *is-refType-widen-Object*[*OF wf*])
have *is-type P* (*exec-lub* (*widen1 P*) (*super P*) *A B*)
proof(*cases A* = *Class Object* ∨ *B* = *Class Object*)
case *True*
hence *exec-lub* (*widen1 P*) (*super P*) *A B* = *Class Object*
proof(*rule disjE*)
assume *A*: *A* = *Class Object*
moreover
from *BObject BNT itB* **have** $P \vdash B <^*$ *Class Object*
by(*cases B* = *Class Object*)(*auto intro*: *trancl-into-rtrancl widen-into-widen1-trancl*[*OF wf*])
hence *is-ub* ((*widen1 P*)*) (*Class Object*) *B* (*Class Object*)
by(*auto intro*: *is-ubI*)
hence *is-lub* ((*widen1 P*)*) (*Class Object*) *B* (*Class Object*)
by(*auto simp add*: *is-lub-def dest*: *is-ubD*)
with *acyclic-widen1*[*OF wf*]
have *exec-lub* (*widen1 P*) (*super P*) (*Class Object*) *B* = *Class Object*
by(*auto intro*: *exec-lub-conv superI*)
ultimately show *exec-lub* (*widen1 P*) (*super P*) *A B* = *Class Object* **by** *simp*
next
assume *B*: *B* = *Class Object*
moreover
from *AObject ANT itA*
have (*A*, *Class Object*) ∈ (*widen1 P*)*
by(*cases A* = *Class Object*, *auto intro*: *trancl-into-rtrancl widen-into-widen1-trancl*[*OF wf*])
hence *is-ub* ((*widen1 P*)*) (*Class Object*) *A* (*Class Object*)
by(*auto intro*: *is-ubI*)
hence *is-lub* ((*widen1 P*)*) (*Class Object*) *A* (*Class Object*)
by(*auto simp add*: *is-lub-def dest*: *is-ubD*)
with *acyclic-widen1*[*OF wf*]
have *exec-lub* (*widen1 P*) (*super P*) *A* (*Class Object*) = *Class Object*
by(*auto intro*: *exec-lub-conv superI*)
ultimately show *exec-lub* (*widen1 P*) (*super P*) *A B* = *Class Object* **by** *simp*
qed
with *wf* **show** *?thesis* **by**(*simp*)
next
case *False*
hence *AnObject*: *A* ≠ *Class Object*
and *BnObject*: *B* ≠ *Class Object* **by** *auto*

```
    from widen-into-widen1-trancl[OF wf AObject AnObject ANT itA]
    have P ⊢ A <* Class Object by(rule trancl-into-rtrancl)
    moreover from widen-into-widen1-trancl[OF wf BObject BnObject BNT itB]
    have P ⊢ B <* Class Object by(rule trancl-into-rtrancl)
    ultimately have is-lub ((widen1 P)*) A B (exec-lub (widen1 P) (super P) A B)
      by(rule is-lub-exec-lub[OF single-valued-widen1[OF wf] acyclic-widen1[OF wf]])(auto intro:
superI)
    hence Aew1: P ⊢ A <* exec-lub (widen1 P) (super P) A B
     by(auto simp add: is-lub-def dest!: is-ubD)
    thus ?thesis
    proof(rule rtranclE)
     assume A = exec-lub (widen1 P) (super P) A B
     with itA show ?thesis by simp
    next
     fix A'
     assume P ⊢ A' <1 exec-lub (widen1 P) (super P) A B
     thus ?thesis by(rule widen1-is-type[OF wf])
    qed
   qed }
  with is-class-Object[OF wf] sup itA itB show ?thesis unfolding sup-def
   by(cases A = B)(auto split: if-split-asm simp add: exec-lub-refl)
qed

lemma closed-err-types:
  assumes wfP: wf-prog wf-mb P
  shows closed (err (types P)) (lift2 (sup P))
proof −
  { fix A B
    assume it: is-type P A is-type P B
     and A ≠ NT B ≠ NT A ≠ B
     and is-refT A is-refT B
    hence is-type P (exec-lub (widen1 P) (super P) A B)
     using sup-is-type[OF wfP it] by(simp add: sup-def) }
  with is-class-Object[OF wfP] show ?thesis
   unfolding closed-def plussub-def lift2-def sup-def'
   by(auto split: err.split ty.splits)(auto simp add: exec-lub-refl)
qed

lemma widen-into-widen1-rtrancl:
  ⟦wf-prog wfmd P; widen P A B; A ≠ NT; is-type P A ⟧ ⟹ (A, B) ∈ (widen1 P)*
by(cases A = B)(auto intro: trancl-into-rtrancl widen-into-widen1-trancl)


lemma sup-widen-greater:
  assumes wfP: wf-prog wf-mb P
  and it1: is-type P t1
  and it2: is-type P t2
  and sup: sup P t1 t2 = OK s
  shows widen P t1 s ∧ widen P t2 s
proof −
  { assume t1: is-refT t1
     and t2: is-refT t2
     and t1NT: t1 ≠ NT
     and t2NT: t2 ≠ NT
```

```
    with it1 it2 wfP have P ⊢ t1 ≤ Class Object P ⊢ t2 ≤ Class Object
      by(auto intro: is-refType-widen-Object)
    with t1NT t2NT it1 it2
    have P ⊢ t1 <* Class Object P ⊢ t2 <* Class Object
      by(auto intro: widen-into-widen1-rtrancl[OF wfP])
    with single-valued-widen1[OF wfP]
    obtain u where is-lub ((widen1 P)^*) t1 t2 u
      by (blast dest: single-valued-has-lubs)
    hence P ⊢ t1 ≤ exec-lub (widen1 P) (super P) t1 t2 ∧
          P ⊢ t2 ≤ exec-lub (widen1 P) (super P) t1 t2
      using acyclic-widen1[OF wfP] superI[of - - P]
      by(simp add: exec-lub-conv)(blast dest: is-lubD is-ubD intro: widen1-rtrancl-into-widen) }
  with it1 it2 sup show ?thesis
    by (cases s) (auto simp add: sup-def split: if-split-asm elim: refTE)
qed

lemma sup-widen-smallest:
  assumes wfP: wf-prog wf-mb P
  and itT: is-type P T
  and itU: is-type P U
  and TwV: P ⊢ T ≤ V
  and UwV: P ⊢ U ≤ V
  and sup: sup P T U = OK W
  shows widen P W V
proof −
  { assume rT: is-refT T
      and rU: is-refT U
      and UNT: U ≠ NT
      and TNT: T ≠ NT
      and W: exec-lub (widen1 P) (super P) T U = W
    from itU itT rT rU UNT TNT have P ⊢ T ≤ Class Object P ⊢ U ≤ Class Object
      by(auto intro:is-refType-widen-Object[OF wfP])
    with UNT TNT itT itU
    have P ⊢ T <* Class Object P ⊢ U <* Class Object
      by(auto intro: widen-into-widen1-rtrancl[OF wfP])
    with single-valued-widen1[OF wfP]
    obtain X where lub: is-lub ((widen1 P)^* ) T U X
      by (blast dest: single-valued-has-lubs)
    with acyclic-widen1[OF wfP]
    have exec-lub (widen1 P) (super P) T U = X
      by (blast intro: superI exec-lub-conv)
    also from TwV TNT UwV UNT itT itU have P ⊢ T <* V P ⊢ U <* V
      by(auto intro: widen-into-widen1-rtrancl[OF wfP])
    with lub have P ⊢ X <* V
      by (clarsimp simp add: is-lub-def is-ub-def)
    finally have P ⊢ exec-lub (widen1 P) (super P) T U ≤ V
      by(rule widen1-rtrancl-into-widen)
    with W have P ⊢ W ≤ V by simp }
  with sup itT itU TwV UwV show ?thesis
    by(simp add: sup-def split: if-split-asm)
qed

lemma sup-exists:
  ⟦ widen P a c; widen P b c ⟧ ⟹ ∃ T. sup P a b = OK T
```

by(*cases b a rule*: *ty.exhaust*[*case-product ty.exhaust*])(*auto simp add*: *sup-def*)

lemma *err-semilat-JType-esl*:
assumes *wf-prog*: *wf-prog wf-mb P*
shows *err-semilat* (*esl P*)
proof −
from *wf-prog* **have** *order* (*widen P*) **..**
moreover from *wf-prog*
have *closed* (*err* (*types P*)) (*lift2* (*sup P*))
by (*rule closed-err-types*)
moreover
from *wf-prog* **have**
(∀ *x*∈*err* (*types P*). ∀ *y*∈*err* (*types P*). *x* ⊑$_{Err.le\ (widen\ P)}$ *x* ⊔$_{lift2\ (sup\ P)}$ *y*) ∧
(∀ *x*∈*err* (*types P*). ∀ *y*∈*err* (*types P*). *y* ⊑$_{Err.le\ (widen\ P)}$ *x* ⊔$_{lift2\ (sup\ P)}$ *y*)
by(*auto simp add*: *lesub-def plussub-def Err.le-def lift2-def sup-widen-greater split*: *err.split*)
moreover from *wf-prog* **have**
∀ *x*∈*err* (*types P*). ∀ *y*∈*err* (*types P*). ∀ *z*∈*err* (*types P*).
x ⊑$_{Err.le\ (widen\ P)}$ *z* ∧ *y* ⊑$_{Err.le\ (widen\ P)}$ *z* ⟶ *x* ⊔$_{lift2\ (sup\ P)}$ *y* ⊑$_{Err.le\ (widen\ P)}$ *z*
unfolding *lift2-def plussub-def lesub-def Err.le-def*
by(*auto intro*: *sup-widen-smallest dest*:*sup-exists simp add*: *split*: *err.split*)
ultimately show *?thesis* **by** (*simp add*: *esl-def semilat-def sl-def Err.sl-def*)
qed

3.16.1 Relation between *SemiType.sup P T U = OK V* and *P* ⊢ *lub*(*T*, *U*) = *V*

lemma *sup-is-lubI*:
assumes *wf*: *wf-prog wf-md P*
and *it*: *is-type P T is-type P U*
and *sup*: *sup P T U = OK V*
shows *P* ⊢ *lub*(*T*, *U*) = *V*
proof
from *sup-widen-greater*[*OF wf it sup*]
show *P* ⊢ *T* ≤ *V P* ⊢ *U* ≤ *V* **by** *blast*+
next
fix *T*′
assume *P* ⊢ *T* ≤ *T*′ *P* ⊢ *U* ≤ *T*′
thus *P* ⊢ *V* ≤ *T*′ **using** *sup* **by**(*rule sup-widen-smallest*[*OF wf it*])
qed

lemma *is-lub-subD*:
assumes *wf*: *wf-prog wf-md P*
and *it*: *is-type P T is-type P U*
and *lub*: *P* ⊢ *lub*(*T*, *U*) = *V*
shows *sup P T U = OK V*
proof −
from *lub* **have** *P* ⊢ *T* ≤ *V P* ⊢ *U* ≤ *V* **by**(*blast dest*: *is-lub-upper*)+
from *sup-exists*[*OF this*] **obtain** *W* **where** *sup P T U = OK W* **by** *blast*
moreover
with *wf it* **have** *P* ⊢ *lub*(*T*, *U*) = *W* **by**(*rule sup-is-lubI*)
with *lub* **have** *V* = *W* **by**(*auto dest*: *is-lub-unique*[*OF wf*])
ultimately show *?thesis* **by** *simp*
qed

lemma *is-lub-is-type*:

⟦ *wf-prog wf-md P*; *is-type P T*; *is-type P U*; *P* ⊢ *lub*(*T*, *U*) = *V* ⟧ ⟹ *is-type P V*
by(*frule* (*3*) *is-lub-subD*)(*erule* (*3*) *sup-is-type*)

3.16.2 Code generator setup

code-pred *widen1p* **.**
lemmas [*code*] = *widen1-def*

lemma *eval-widen1p-i-i-o-conv*:
Predicate.eval (*widen1p-i-i-o P T*) = (λ*U*. *P* ⊢ *T* <¹ *U*)
by(*auto elim*: *widen1p-i-i-oE intro*: *widen1p-i-i-oI simp add*: *widen1-def fun-eq-iff*)

lemma *rtrancl-widen1-code* [*code-unfold*]:
(*widen1 P*)^* = {(*a*, *b*). *Predicate.holds* (*rtrancl-tab-FioB-i-i-i* (*widen1p-i-i-o P*) [] *a b*)}
by(*auto simp add*: *fun-eq-iff Predicate.holds-eq widen1-def rtrancl-def rtranclp-eq-rtrancl-tab-nil eval-widen1p-i-i-o-c*
intro!: *rtrancl-tab-FioB-i-i-iI elim*!: *rtrancl-tab-FioB-i-i-iE*)

declare *exec-lub-def* [*code-unfold*]

end
theory *Common-Main*
imports
../Basic/Auxiliary
../Framework/FWProgress
../Framework/FWBisimDeadlock
../Framework/FWBisimLift
../DFA/Abstract-BV
ExternalCallWF
ConformThreaded
BinOp
SemiType
begin

end

Chapter 4

JinjaThreads source language

4.1 Program State

theory *State*
imports
../Common/Heap
begin

type-synonym
'addr locals = *vname* ⇀ *'addr val* — local vars, incl. params and "this"
type-synonym
(*'addr*, *'heap*) *Jstate* = *'heap* × *'addr locals* — the heap and the local vars

definition *hp* :: *'heap* × *'x* ⇒ *'heap* **where** *hp* ≡ *fst*

definition *lcl* :: *'heap* × *'x* ⇒ *'x* **where** *lcl* ≡ *snd*

lemma *hp-conv* [*simp*]: *hp* (*h*, *l*) = *h*
by(*simp add*: *hp-def*)

lemma *lcl-conv* [*simp*]: *lcl* (*h*, *l*) = *l*
by(*simp add*: *lcl-def*)

end

4.2 Expressions

theory *Expr*
imports
../Common/BinOp
begin

datatype (*dead 'a*, *dead 'b*, *dead 'addr*) *exp*
= *new cname* — class instance creation
| *newArray ty* ('*a*,'*b*,'*addr*) *exp* (‹*newA* -⌊-⌉› [*99*,*0*] *90*) — array instance creation: type, size in outermost dimension
| *Cast ty* ('*a*,'*b*,'*addr*) *exp* — type cast
| *InstanceOf* ('*a*,'*b*,'*addr*) *exp ty* (‹- *instanceof* -› [*99*, *99*] *90*) — instance of

| *Val 'addr val* — value
| *BinOp ('a,'b,'addr) exp bop ('a,'b,'addr) exp* (‹- «-» -› [80,0,81] 80) — binary operation
| *Var 'a* — local variable (incl. parameter)
| *LAss 'a ('a,'b,'addr) exp* (‹-:=-› [90,90]90) — local assignment
| *AAcc ('a,'b,'addr) exp ('a,'b,'addr) exp* (‹-⌊-⌋› [99,0] 90) — array cell read
| *AAss ('a,'b,'addr) exp ('a,'b,'addr) exp ('a,'b,'addr) exp* (‹-⌊-⌋ := -› [10,99,90] 90) — array cell assignment
| *ALen ('a,'b,'addr) exp* (‹-•length› [10] 90) — array length
| *FAcc ('a,'b,'addr) exp vname cname* (‹-•-{-}› [10,90,99]90) — field access
| *FAss ('a,'b,'addr) exp vname cname ('a,'b,'addr) exp* (‹-•-{-} := -› [10,90,99,90]90) — field assignment
| *CompareAndSwap ('a,'b,'addr) exp cname vname ('a,'b,'addr) exp ('a,'b,'addr) exp* (‹-•compareAndSwap('(-•-, -, -'))› [10,90,90,90,90] 90) — compare and swap
| *Call ('a,'b,'addr) exp mname ('a,'b,'addr) exp list* (‹-•-'(-')› [90,99,0] 90) — method call
| *Block 'a ty 'addr val option ('a,'b,'addr) exp* (‹'{-:-=-; -}›)
| *Synchronized 'b ('a,'b,'addr) exp ('a,'b,'addr) exp* (‹*sync*_ '(-') -› [99,99,90] 90)
| *InSynchronized 'b 'addr ('a,'b,'addr) exp* (‹*insync*_ '(-') -› [99,99,90] 90)
| *Seq ('a,'b,'addr) exp ('a,'b,'addr) exp* (‹-;;/ -› [61,60]60)
| *Cond ('a,'b,'addr) exp ('a,'b,'addr) exp ('a,'b,'addr) exp* (‹*if* '(-') -/ *else* -› [80,79,79]70)
| *While ('a,'b,'addr) exp ('a,'b,'addr) exp* (‹*while* '(-') -› [80,79]70)
| *throw ('a,'b,'addr) exp*
| *TryCatch ('a,'b,'addr) exp cname 'a ('a,'b,'addr) exp* (‹*try* -/ *catch*'(- -') -› [0,99,80,79] 70)

type-synonym
'addr expr = (*vname*, *unit*, *'addr*) *exp* — Jinja expression
type-synonym
'addr J-mb = *vname list* × *'addr expr* — Jinja method body: parameter names and expression
type-synonym
'addr J-prog = *'addr J-mb prog* — Jinja program

translations
(*type*) *'addr expr* <= (*type*) (*String.literal*, *unit*, *'addr*) *exp*
(*type*) *'addr J-prog* <= (*type*) (*String.literal list* × *'addr expr*) *prog*

4.2.1 Syntactic sugar

abbreviation *unit* :: ('a,'b,'addr) *exp*
where *unit* ≡ *Val Unit*

abbreviation *null* :: ('a,'b,'addr) *exp*
where *null* ≡ *Val Null*

abbreviation *addr* :: *'addr* ⇒ ('a,'b,'addr) *exp*
where *addr a* == *Val* (*Addr a*)

abbreviation *true* :: ('a,'b,'addr) *exp*
where *true* == *Val* (*Bool True*)

abbreviation *false* :: ('a,'b,'addr) *exp*
where *false* == *Val* (*Bool False*)

abbreviation *Throw* :: *'addr* ⇒ ('a,'b,'addr) *exp*
where *Throw a* == *throw* (*Val* (*Addr a*))

abbreviation (**in** *heap-base*) *THROW* :: *cname* ⇒ ('*a*,'*b*,'*addr*) *exp*
where *THROW xc* == *Throw* (*addr-of-sys-xcpt xc*)

abbreviation *sync-unit-syntax* :: ('*a*,*unit*,'*addr*) *exp* ⇒ ('*a*,*unit*,'*addr*) *exp* ⇒ ('*a*,*unit*,'*addr*) *exp* (‹*sync*'(-') -› [*99*,*90*] *90*)
where *sync*(*e1*) *e2* ≡ *sync*$_{()}$ (*e1*) *e2*

abbreviation *insync-unit-syntax* :: '*addr* ⇒ ('*a*,*unit*,'*addr*) *exp* ⇒ ('*a*,*unit*,'*addr*) *exp* (‹*insync*'(-') -› [*99*,*90*] *90*)
where *insync*(*a*) *e2* ≡ *insync*$_{()}$ (*a*) *e2*

Java syntax for binary operators

abbreviation *BinOp-Eq* :: ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp*
(‹- «==» -› [*80*,*81*] *80*)
where *e* «==» *e'* ≡ *e* «*Eq*» *e'*

abbreviation *BinOp-NotEq* :: ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp*
(‹- «!=» -› [*80*,*81*] *80*)
where *e* «!=» *e'* ≡ *e* «*NotEq*» *e'*

abbreviation *BinOp-LessThan* :: ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp*
(‹- «<» -› [*80*,*81*] *80*)
where *e* «<» *e'* ≡ *e* «*LessThan*» *e'*

abbreviation *BinOp-LessOrEqual* :: ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp*
(‹- «<=» -› [*80*,*81*] *80*)
where *e* «<=» *e'* ≡ *e* «*LessOrEqual*» *e'*

abbreviation *BinOp-GreaterThan* :: ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp*
(‹- «>» -› [*80*,*81*] *80*)
where *e* «>» *e'* ≡ *e* «*GreaterThan*» *e'*

abbreviation *BinOp-GreaterOrEqual* :: ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp*
(‹- «>=» -› [*80*,*81*] *80*)
where *e* «>=» *e'* ≡ *e* «*GreaterOrEqual*» *e'*

abbreviation *BinOp-Add* :: ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp*
(‹- «+» -› [*80*,*81*] *80*)
where *e* «+» *e'* ≡ *e* «*Add*» *e'*

abbreviation *BinOp-Subtract* :: ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp*
(‹- «−» -› [*80*,*81*] *80*)
where *e* «−» *e'* ≡ *e* «*Subtract*» *e'*

abbreviation *BinOp-Mult* :: ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp*
(‹- «*» -› [*80*,*81*] *80*)
where *e* «*» *e'* ≡ *e* «*Mult*» *e'*

abbreviation *BinOp-Div* :: ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp*
(‹- «'/» -› [*80*,*81*] *80*)
where *e* «/» *e'* ≡ *e* «*Div*» *e'*

abbreviation *BinOp-Mod* :: ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp* ⇒ ('*a*, '*b*, '*c*) *exp*

(‹- «%» -› [80,81] 80)
where *e* «%» *e′* ≡ *e* «*Mod*» *e′*

abbreviation *BinOp-BinAnd* :: ('a, 'b, 'c) *exp* ⇒ ('a, 'b, 'c) *exp* ⇒ ('a, 'b, 'c) *exp*
(‹- «&» -› [80,81] 80)
where *e* «&» *e′* ≡ *e* «*BinAnd*» *e′*

abbreviation *BinOp-BinOr* :: ('a, 'b, 'c) *exp* ⇒ ('a, 'b, 'c) *exp* ⇒ ('a, 'b, 'c) *exp*
(‹- «|» -› [80,81] 80)
where *e* «|» *e′* ≡ *e* «*BinOr*» *e′*

abbreviation *BinOp-BinXor* :: ('a, 'b, 'c) *exp* ⇒ ('a, 'b, 'c) *exp* ⇒ ('a, 'b, 'c) *exp*
(‹- «^» -› [80,81] 80)
where *e* «^» *e′* ≡ *e* «*BinXor*» *e′*

abbreviation *BinOp-ShiftLeft* :: ('a, 'b, 'c) *exp* ⇒ ('a, 'b, 'c) *exp* ⇒ ('a, 'b, 'c) *exp*
(‹- «<<» -› [80,81] 80)
where *e* «<<» *e′* ≡ *e* «*ShiftLeft*» *e′*

abbreviation *BinOp-ShiftRightZeros* :: ('a, 'b, 'c) *exp* ⇒ ('a, 'b, 'c) *exp* ⇒ ('a, 'b, 'c) *exp*
(‹- «>>>» -› [80,81] 80)
where *e* «>>>» *e′* ≡ *e* «*ShiftRightZeros*» *e′*

abbreviation *BinOp-ShiftRightSigned* :: ('a, 'b, 'c) *exp* ⇒ ('a, 'b, 'c) *exp* ⇒ ('a, 'b, 'c) *exp*
(‹- «>>» -› [80,81] 80)
where *e* «>>» *e′* ≡ *e* «*ShiftRightSigned*» *e′*

abbreviation *BinOp-CondAnd* :: ('a, 'b, 'c) *exp* ⇒ ('a, 'b, 'c) *exp* ⇒ ('a, 'b, 'c) *exp*
(‹- «&&» -› [80,81] 80)
where *e* «&&» *e′* ≡ *if* (*e*) *e′ else false*

abbreviation *BinOp-CondOr* :: ('a, 'b, 'c) *exp* ⇒ ('a, 'b, 'c) *exp* ⇒ ('a, 'b, 'c) *exp*
(‹- «||» -› [80,81] 80)
where *e* «||» *e′* ≡ *if* (*e*) *true else e′*

lemma *inj-Val* [*simp*]: *inj Val*
by(*rule inj-onI*)(*simp*)

lemma *expr-ineqs* [*simp*]: *Val v* ;; *e* ≠ *e if* (*e1*) *e else e2* ≠ *e if* (*e1*) *e2 else e* ≠ *e*
by(*induct e*) *auto*

4.2.2 Free Variables

primrec *fv* :: ('a,'b,'addr) *exp* ⇒ 'a *set*
and *fvs* :: ('a,'b,'addr) *exp list* ⇒ 'a *set*
where
fv(*new C*) = {}
| *fv*(*newA T*⌊*e*⌉) = *fv e*
| *fv*(*Cast C e*) = *fv e*
| *fv*(*e instanceof T*) = *fv e*
| *fv*(*Val v*) = {}
| *fv*(*e*₁ «*bop*» *e*₂) = *fv e*₁ ∪ *fv e*₂
| *fv*(*Var V*) = {*V*}
| *fv*(*a*⌊*i*⌉) = *fv a* ∪ *fv i*

| $fv(AAss\ a\ i\ e) = fv\ a \cup fv\ i \cup fv\ e$
| $fv(a{\cdot}length) = fv\ a$
| $fv(LAss\ V\ e) = \{V\} \cup fv\ e$
| $fv(e{\cdot}F\{D\}) = fv\ e$
| $fv(FAss\ e_1\ F\ D\ e_2) = fv\ e_1 \cup fv\ e_2$
| $fv(e_1{\cdot}compareAndSwap(D{\cdot}F,\ e_2,\ e_3)) = fv\ e_1 \cup fv\ e_2 \cup fv\ e_3$
| $fv(e{\cdot}M(es)) = fv\ e \cup fvs\ es$
| $fv(\{V{:}T{=}vo;\ e\}) = fv\ e - \{V\}$
| $fv(sync_V\ (h)\ e) = fv\ h \cup fv\ e$
| $fv(insync_V\ (a)\ e) = fv\ e$
| $fv(e_1;;e_2) = fv\ e_1 \cup fv\ e_2$
| $fv(if\ (b)\ e_1\ else\ e_2) = fv\ b \cup fv\ e_1 \cup fv\ e_2$
| $fv(while\ (b)\ e) = fv\ b \cup fv\ e$
| $fv(throw\ e) = fv\ e$
| $fv(try\ e_1\ catch(C\ V)\ e_2) = fv\ e_1 \cup (fv\ e_2 - \{V\})$

| $fvs([]) = \{\}$
| $fvs(e\#es) = fv\ e \cup fvs\ es$

lemma [*simp*]: $fvs(es\ @\ es') = fvs\ es \cup fvs\ es'$
by(*induct es*) *auto*

lemma [*simp*]: $fvs(map\ Val\ vs) = \{\}$
by (*induct vs*) *auto*

4.2.3 Locks and addresses

primrec *expr-locks* :: $('a,'b,'addr)\ exp \Rightarrow 'addr \Rightarrow nat$
  and *expr-lockss* :: $('a,'b,'addr)\ exp\ list \Rightarrow 'addr \Rightarrow nat$
where
  $expr\text{-}locks\ (new\ C) = (\lambda ad.\ 0)$
| $expr\text{-}locks\ (newA\ T\lfloor e\rceil) = expr\text{-}locks\ e$
| $expr\text{-}locks\ (Cast\ T\ e) = expr\text{-}locks\ e$
| $expr\text{-}locks\ (e\ instanceof\ T) = expr\text{-}locks\ e$
| $expr\text{-}locks\ (Val\ v) = (\lambda ad.\ 0)$
| $expr\text{-}locks\ (Var\ v) = (\lambda ad.\ 0)$
| $expr\text{-}locks\ (e\ «bop»\ e') = (\lambda ad.\ expr\text{-}locks\ e\ ad + expr\text{-}locks\ e'\ ad)$
| $expr\text{-}locks\ (V := e) = expr\text{-}locks\ e$
| $expr\text{-}locks\ (a\lfloor i\rceil) = (\lambda ad.\ expr\text{-}locks\ a\ ad + expr\text{-}locks\ i\ ad)$
| $expr\text{-}locks\ (AAss\ a\ i\ e) = (\lambda ad.\ expr\text{-}locks\ a\ ad + expr\text{-}locks\ i\ ad + expr\text{-}locks\ e\ ad)$
| $expr\text{-}locks\ (a{\cdot}length) = expr\text{-}locks\ a$
| $expr\text{-}locks\ (e{\cdot}F\{D\}) = expr\text{-}locks\ e$
| $expr\text{-}locks\ (FAss\ e\ F\ D\ e') = (\lambda ad.\ expr\text{-}locks\ e\ ad + expr\text{-}locks\ e'\ ad)$
| $expr\text{-}locks\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e'')) = (\lambda ad.\ expr\text{-}locks\ e\ ad + expr\text{-}locks\ e'\ ad + expr\text{-}locks\ e''\ ad)$
| $expr\text{-}locks\ (e{\cdot}m(ps)) = (\lambda ad.\ expr\text{-}locks\ e\ ad + expr\text{-}lockss\ ps\ ad)$
| $expr\text{-}locks\ (\{V : T{=}vo;\ e\}) = expr\text{-}locks\ e$
| $expr\text{-}locks\ (sync_V\ (o')\ e) = (\lambda ad.\ expr\text{-}locks\ o'\ ad + expr\text{-}locks\ e\ ad)$
| $expr\text{-}locks\ (insync_V\ (a)\ e) = (\lambda ad.\ if\ (a = ad)\ then\ Suc\ (expr\text{-}locks\ e\ ad)\ else\ expr\text{-}locks\ e\ ad)$
| $expr\text{-}locks\ (e;;e') = (\lambda ad.\ expr\text{-}locks\ e\ ad + expr\text{-}locks\ e'\ ad)$
| $expr\text{-}locks\ (if\ (b)\ e\ else\ e') = (\lambda ad.\ expr\text{-}locks\ b\ ad + expr\text{-}locks\ e\ ad + expr\text{-}locks\ e'\ ad)$
| $expr\text{-}locks\ (while\ (b)\ e) = (\lambda ad.\ expr\text{-}locks\ b\ ad + expr\text{-}locks\ e\ ad)$
| $expr\text{-}locks\ (throw\ e) = expr\text{-}locks\ e$
| $expr\text{-}locks\ (try\ e\ catch(C\ v)\ e') = (\lambda ad.\ expr\text{-}locks\ e\ ad + expr\text{-}locks\ e'\ ad)$

| *expr-lockss* [] = (λ*a*. *0*)
| *expr-lockss* (*x*#*xs*) = (λ*ad*. *expr-locks x ad* + *expr-lockss xs ad*)

lemma *expr-lockss-append* [*simp*]:
expr-lockss (*es* @ *es'*) = (λ*ad*. *expr-lockss es ad* + *expr-lockss es' ad*)
by(*induct es*) *auto*

lemma *expr-lockss-map-Val* [*simp*]: *expr-lockss* (*map Val vs*) = (λ*ad*. *0*)
by(*induct vs*) *auto*

primrec *contains-insync* :: ('*a*,'*b*,'*addr*) *exp* ⇒ *bool*
and *contains-insyncs* :: ('*a*,'*b*,'*addr*) *exp list* ⇒ *bool*
where
contains-insync (*new C*) = *False*
| *contains-insync* (*newA T*⌊*i*⌉) = *contains-insync i*
| *contains-insync* (*Cast T e*) = *contains-insync e*
| *contains-insync* (*e instanceof T*) = *contains-insync e*
| *contains-insync* (*Val v*) = *False*
| *contains-insync* (*Var v*) = *False*
| *contains-insync* (*e* «*bop*» *e'*) = (*contains-insync e* ∨ *contains-insync e'*)
| *contains-insync* (*V* := *e*) = *contains-insync e*
| *contains-insync* (*a*⌊*i*⌉) = (*contains-insync a* ∨ *contains-insync i*)
| *contains-insync* (*AAss a i e*) = (*contains-insync a* ∨ *contains-insync i* ∨ *contains-insync e*)
| *contains-insync* (*a*·*length*) = *contains-insync a*
| *contains-insync* (*e*·*F*{*D*}) = *contains-insync e*
| *contains-insync* (*FAss e F D e'*) = (*contains-insync e* ∨ *contains-insync e'*)
| *contains-insync* (*e*·*compareAndSwap*(*D*·*F*, *e'*, *e''*)) = (*contains-insync e* ∨ *contains-insync e'* ∨ *contains-insync e''*)
| *contains-insync* (*e*·*m*(*pns*)) = (*contains-insync e* ∨ *contains-insyncs pns*)
| *contains-insync* ({*V* : *T*=*vo*; *e*}) = *contains-insync e*
| *contains-insync* (*sync*$_V$ (*o'*) *e*) = (*contains-insync o'* ∨ *contains-insync e*)
| *contains-insync* (*insync*$_V$ (*a*) *e*) = *True*
| *contains-insync* (*e*;;*e'*) = (*contains-insync e* ∨ *contains-insync e'*)
| *contains-insync* (*if* (*b*) *e else e'*) = (*contains-insync b* ∨ *contains-insync e* ∨ *contains-insync e'*)
| *contains-insync* (*while* (*b*) *e*) = (*contains-insync b* ∨ *contains-insync e*)
| *contains-insync* (*throw e*) = *contains-insync e*
| *contains-insync* (*try e catch*(*C v*) *e'*) = (*contains-insync e* ∨ *contains-insync e'*)

| *contains-insyncs* [] = *False*
| *contains-insyncs* (*x* # *xs*) = (*contains-insync x* ∨ *contains-insyncs xs*)

lemma *contains-insyncs-append* [*simp*]:
contains-insyncs (*es* @ *es'*) ⟷ *contains-insyncs es* ∨ *contains-insyncs es'*
by(*induct es*, *auto*)

lemma fixes *e* :: ('*a*, '*b*, '*addr*) *exp*
and *es* :: ('*a*, '*b*, '*addr*) *exp list*
shows *contains-insync-conv*: (*contains-insync e* ⟷ (∃ *ad*. *expr-locks e ad* > *0*))
and *contains-insyncs-conv*: (*contains-insyncs es* ⟷ (∃ *ad*. *expr-lockss es ad* > *0*))
by(*induct e* **and** *es rule*: *expr-locks.induct expr-lockss.induct*)(*auto*)

lemma *contains-insyncs-map-Val* [*simp*]: ¬ *contains-insyncs* (*map Val vs*)
by(*induct vs*) *auto*

4.2.4 Value expressions

inductive *is-val* :: (*'a*,*'b*,*'addr*) *exp* ⇒ *bool* **where**
  is-val (*Val v*)

declare *is-val.intros* [*simp*]
declare *is-val.cases* [*elim*!]

lemma *is-val-iff*: *is-val e* ⟷ (∃ *v*. *e* = *Val v*)
by(*auto*)

code-pred *is-val* **.**

fun *is-vals* :: (*'a*,*'b*,*'addr*) *exp list* ⇒ *bool* **where**
  is-vals [] = *True*
| *is-vals* (*e*#*es*) = (*is-val e* ∧ *is-vals es*)

lemma *is-vals-append* [*simp*]: *is-vals* (*es* @ *es'*) ⟷ *is-vals es* ∧ *is-vals es'*
by(*induct es*) *auto*

lemma *is-vals-conv*: *is-vals es* = (∃ *vs*. *es* = *map Val vs*)
by(*induct es*)(*auto simp add*: *Cons-eq-map-conv*)

lemma *is-vals-map-Vals* [*simp*]: *is-vals* (*map Val vs*) = *True*
unfolding *is-vals-conv* **by** *auto*

inductive *is-addr* :: (*'a*,*'b*,*'addr*) *exp* ⇒ *bool*
where *is-addr* (*addr a*)

declare *is-addr.intros*[*intro*!]
declare *is-addr.cases*[*elim*!]

lemma [*simp*]: (*is-addr e*) ⟷ (∃ *a*. *e* = *addr a*)
by *auto*

primrec *the-Val* :: (*'a*, *'b*, *'addr*) *exp* ⇒ *'addr val*
where
  the-Val (*Val v*) = *v*

inductive *is-Throws* :: (*'a*, *'b*, *'addr*) *exp list* ⇒ *bool*
where
  is-Throws (*Throw a* # *es*)
| *is-Throws es* ⟹ *is-Throws* (*Val v* # *es*)

inductive-simps *is-Throws-simps*:
  is-Throws []
  is-Throws (*e* # *es*)

code-pred *is-Throws* **.**

lemma *is-Throws-conv*: *is-Throws es* ⟷ (∃ *vs a es'*. *es* = *map Val vs* @ *Throw a* # *es'*)
  (**is** *?lhs* ⟷ *?rhs*)
proof
  assume *?lhs* **thus** *?rhs*

by(*induct*)(*fastforce simp add*: *Cons-eq-append-conv Cons-eq-map-conv*)+
next
assume *?rhs* **thus** *?lhs*
by(*induct es*)(*auto simp add*: *is-Throws-simps Cons-eq-map-conv Cons-eq-append-conv*)
qed

4.2.5 *blocks*

fun *blocks* :: *'a list* ⇒ *ty list* ⇒ *'addr val list* ⇒ (*'a*,*'b*,*'addr*) *exp* ⇒ (*'a*,*'b*,*'addr*) *exp*
where
blocks (*V* # *Vs*) (*T* # *Ts*) (*v* # *vs*) *e* = {*V*:*T*=⌊*v*⌋; *blocks Vs Ts vs e*}
| *blocks* [] [] [] *e* = *e*

lemma [*simp*]:
⟦ *size vs* = *size Vs*; *size Ts* = *size Vs* ⟧ ⟹ *fv* (*blocks Vs Ts vs e*) = *fv e* − *set Vs*
by(*induct rule*:*blocks.induct*)(*simp-all*, *blast*)

lemma *expr-locks-blocks*:
⟦ *length vs* = *length pns*; *length Ts* = *length pns* ⟧
⟹ *expr-locks* (*blocks pns Ts vs e*) = *expr-locks e*
by(*induct pns Ts vs e rule*: *blocks.induct*)(*auto*)

4.2.6 Final expressions

inductive *final* :: (*'a*,*'b*,*'addr*) *exp* ⇒ *bool* **where**
final (*Val v*)
| *final* (*Throw a*)

declare *final.cases* [*elim*]
declare *final.intros*[*simp*]

lemmas *finalE*[*consumes 1*, *case-names Val Throw*] = *final.cases*

lemma *final-iff*: *final e* ⟷ (∃ *v*. *e* = *Val v*) ∨ (∃ *a*. *e* = *Throw a*)
by(*auto*)

lemma *final-locks*: *final e* ⟹ *expr-locks e l* = *0*
by(*auto elim*: *finalE*)

inductive *finals* :: (*'a*,*'b*,*'addr*) *exp list* ⇒ *bool*
where
finals []
| *finals* (*Throw a* # *es*)
| *finals es* ⟹ *finals* (*Val v* # *es*)

inductive-simps *finals-simps*:
finals (*e* # *es*)

lemma [*iff*]: *finals* []
by(*rule finals.intros*)

lemma [*iff*]: *finals* (*Val v* # *es*) = *finals es*
by(*simp add*: *finals-simps*)

lemma *finals-app-map* [*iff*]: *finals* (*map Val vs* @ *es*) = *finals es*
by(*induct vs*) *simp-all*

lemma [*iff*]: *finals* (*throw e* # *es*) = (∃ *a*. *e* = *addr a*)
by(*simp add*: *finals-simps*)

lemma *not-finals-ConsI*: ¬ *final e* ⟹ ¬ *finals* (*e* # *es*)
by(*simp add*: *finals-simps final-iff*)

lemma *finals-iff*: *finals es* ⟷ (∃ *vs*. *es* = *map Val vs*) ∨ (∃ *vs a es'*. *es* = *map Val vs* @ *Throw a* # *es'*)
(**is** *?lhs* ⟷ *?rhs*)
proof
assume *?lhs* **thus** *?rhs*
by *induct*(*auto simp add*: *Cons-eq-append-conv Cons-eq-map-conv*, *metis*)
next
assume *?rhs* **thus** *?lhs* **by**(*induct es*) *auto*
qed

code-pred *final* **.**

4.2.7 converting results from external calls

primrec *extRet2J* :: ('*a*, '*b*, '*addr*) *exp* ⇒ '*addr extCallRet* ⇒ ('*a*, '*b*, '*addr*) *exp*
where
extRet2J e (*RetVal v*) = *Val v*
| *extRet2J e* (*RetExc a*) = *Throw a*
| *extRet2J e RetStaySame* = *e*

lemma *fv-extRet2J* [*simp*]: *fv* (*extRet2J e va*) ⊆ *fv e*
by(*cases va*) *simp-all*

4.2.8 expressions at a call

primrec *call* :: ('*a*,'*b*,'*addr*) *exp* ⇒ ('*addr* × *mname* × '*addr val list*) *option*
and *calls* :: ('*a*,'*b*,'*addr*) *exp list* ⇒ ('*addr* × *mname* × '*addr val list*) *option*
where
call (*new C*) = *None*
| *call* (*newA T*⌊*e*⌉) = *call e*
| *call* (*Cast C e*) = *call e*
| *call* (*e instanceof T*) = *call e*
| *call* (*Val v*) = *None*
| *call* (*Var V*) = *None*
| *call* (*V*:=*e*) = *call e*
| *call* (*e* «*bop*» *e'*) = (*if is-val e then call e' else call e*)
| *call* (*a*⌊*i*⌉) = (*if is-val a then call i else call a*)
| *call* (*AAss a i e*) = (*if is-val a then* (*if is-val i then call e else call i*) *else call a*)
| *call* (*a*·*length*) = *call a*
| *call* (*e*·*F*{*D*}) = *call e*
| *call* (*FAss e F D e'*) = (*if is-val e then call e' else call e*)
| *call* (*e*·*compareAndSwap*(*D*·*F*, *e'*, *e''*)) = (*if is-val e then if is-val e' then call e'' else call e' else call e*)
| *call* (*e*·*M*(*es*)) = (*if is-val e then*
(*if is-vals es* ∧ *is-addr e then* ⌊(*THE a*. *e* = *addr a*, *M*, *THE vs*. *es* = *map Val vs*)⌋

else calls es)
else call e)
| *call* ({ *V*:*T*=*vo*; *e*}) = *call e*
| *call* (*sync*$_V$ (*o'*) *e*) = *call o'*
| *call* (*insync*$_V$ (*a*) *e*) = *call e*
| *call* (*e*;;*e'*) = *call e*
| *call* (*if* (*e*) *e1 else e2*) = *call e*
| *call* (*while*(*b*) *e*) = *None*
| *call* (*throw e*) = *call e*
| *call* (*try e1 catch*(*C V*) *e2*) = *call e1*

| *calls* [] = *None*
| *calls* (*e*#*es*) = (*if is-val e then calls es else call e*)

lemma *calls-append* [*simp*]:
calls (*es* @ *es'*) = (*if calls es* = *None* ∧ *is-vals es then calls es' else calls es*)
by(*induct es*) *auto*

lemma *call-callE* [*consumes 1*, *case-names CallObj CallParams Call*]:
⟦ *call* (*obj*·*M*(*pns*)) = ⌊(*a*, *M'*, *vs*)⌋;
call obj = ⌊(*a*, *M'*, *vs*)⌋ ⟹ *thesis*;
⋀*v*. ⟦ *obj* = *Val v*; *calls pns* = ⌊(*a*, *M'*, *vs*)⌋ ⟧ ⟹ *thesis*;
⟦ *obj* = *addr a*; *pns* = *map Val vs*; *M* = *M'* ⟧ ⟹ *thesis* ⟧ ⟹ *thesis*
by(*auto split*: *if-split-asm simp add*: *is-vals-conv*)

lemma *calls-map-Val* [*simp*]:
calls (*map Val vs*) = *None*
by(*induct vs*) *auto*

lemma *call-not-is-val* [*dest*]: *call e* = ⌊*aMvs*⌋ ⟹ ¬ *is-val e*
by(*cases e*) *auto*

lemma *is-calls-not-is-vals* [*dest*]: *calls es* = ⌊*aMvs*⌋ ⟹ ¬ *is-vals es*
by(*induct es*) *auto*

end

4.3 Abstract heap locales for source code programs

theory *JHeap*
imports
../Common/Conform
Expr
begin

locale *J-heap-base* = *heap-base* +
constrains *addr2thread-id* :: ('*addr* :: *addr*) ⇒ '*thread-id*
and *thread-id2addr* :: '*thread-id* ⇒ '*addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: '*heap*
and *allocate* :: '*heap* ⇒ *htype* ⇒ ('*heap* × '*addr*) *set*
and *typeof-addr* :: '*heap* ⇒ '*addr* ⇀ *htype*
and *heap-read* :: '*heap* ⇒ '*addr* ⇒ *addr-loc* ⇒ '*addr val* ⇒ *bool*

and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*

locale *J-heap* = *heap* +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *P* :: *'addr J-prog*

sublocale *J-heap* < *J-heap-base* **.**

locale *J-heap-conf-base* = *heap-conf-base* +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr J-prog*

sublocale *J-heap-conf-base* < *J-heap-base* **.**

locale *J-heap-conf* =
J-heap-conf-base +
heap-conf +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr J-prog*

sublocale *J-heap-conf* < *J-heap*
by(*unfold-locales*)

locale *J-progress* =
heap-progress +
J-heap-conf-base +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*

and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr J-prog*

sublocale *J-progress* < *J-heap* **by**(*unfold-locales*)

locale *J-conf-read* =
heap-conf-read +
J-heap-conf +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr J-prog*

sublocale *J-conf-read* < *J-heap* **by**(*unfold-locales*)

locale *J-typesafe* =
heap-typesafe +
J-conf-read +
J-progress +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr J-prog*

end

4.4 Small Step Semantics

theory *SmallStep*
imports
Expr
State
JHeap
begin

type-synonym
(*'addr*, *'thread-id*, *'heap*) *J-thread-action* =
(*'addr*, *'thread-id*, *'addr expr* × *'addr locals*,*'heap*) *Jinja-thread-action*

type-synonym
(*'addr*, *'thread-id*, *'heap*) *J-state* =
(*'addr*,*'thread-id*,*'addr expr* × *'addr locals*,*'heap*,*'addr*) *state*

print-translation ‹
let
fun tr'
[*a1*, *t*
, *Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax exp*}, -) $
Const (@{*type-syntax String.literal*}, -) $ *Const* (@{*type-syntax unit*}, -) $ *a2*) $
(*Const* (@{*type-syntax fun*}, -) $
Const (@{*type-syntax String.literal*}, -) $
(*Const* (@{*type-syntax option*}, -) $
(*Const* (@{*type-syntax val*}, -) $ *a3*)))
, *h*] =
if a1 = *a2 andalso a2* = *a3 then Syntax.const* @{*type-syntax J-thread-action*} $ *a1* $ *t* $ *h*
else raise Match;
in [(@{*type-syntax Jinja-thread-action*}, *K tr'*)]
end
›
typ (*'addr*,*'thread-id*,*'heap*) *J-thread-action*

print-translation ‹
let
fun tr'
[*a1*, *t*
, *Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax exp*}, -) $
Const (@{*type-syntax String.literal*}, -) $ *Const* (@{*type-syntax unit*}, -) $ *a2*) $
(*Const* (@{*type-syntax fun*}, -) $
Const (@{*type-syntax String.literal*}, -) $
(*Const* (@{*type-syntax option*}, -) $
(*Const* (@{*type-syntax val*}, -) $ *a3*)))
, *h*, *a4*] =
if a1 = *a2 andalso a2* = *a3 andalso a3* = *a4 then Syntax.const* @{*type-syntax J-state*} $ *a1* $ *t*
$ *h*
else raise Match;
in [(@{*type-syntax state*}, *K tr'*)]
end
›
typ (*'addr*, *'thread-id*, *'heap*) *J-state*

definition *extNTA2J* :: *'addr J-prog* ⇒ (*cname* × *mname* × *'addr*) ⇒ *'addr expr* × *'addr locals*
where *extNTA2J P* = (λ(*C*, *M*, *a*). *let* (*D*,*Ts*,*T*,*meth*) = *method P C M*; (*pns*,*body*) = *the meth*
in ({*this*:*Class D*=⌊*Addr a*⌋; *body*}, *Map.empty*))

abbreviation *J-local-start* ::
cname ⇒ *mname* ⇒ *ty list* ⇒ *ty* ⇒ *'addr J-mb* ⇒ *'addr val list*
⇒ *'addr expr* × *'addr locals*
where

J-local-start ≡
λ*C M Ts T* (*pns*, *body*) *vs*.
(*blocks* (*this* # *pns*) (*Class C* # *Ts*) (*Null* # *vs*) *body*, *Map.empty*)

abbreviation (**in** *J-heap-base*)
J-start-state :: *'addr J-prog* ⇒ *cname* ⇒ *mname* ⇒ *'addr val list* ⇒ (*'addr*, *'thread-id*, *'heap*) *J-state*
where
J-start-state ≡ *start-state J-local-start*

lemma *extNTA2J-iff* [*simp*]:
extNTA2J P (*C*, *M*, *a*) = ({*this*:*Class* (*fst* (*method P C M*))=⌊*Addr a*⌋; *snd* (*the* (*snd* (*snd* (*snd* (*method P C M*)))))}, *Map.empty*)
by(*simp add*: *extNTA2J-def split-beta*)

abbreviation *extTA2J* ::
'addr J-prog ⇒ (*'addr*, *'thread-id*, *'heap*) *external-thread-action* ⇒ (*'addr*, *'thread-id*, *'heap*) *J-thread-action*
where *extTA2J P* ≡ *convert-extTA* (*extNTA2J P*)

lemma *extTA2J-ε*: *extTA2J P ε* = *ε*
by(*simp*)

Locking mechanism: The expression on which the thread is synchronized is evaluated first to a value. If this expression evaluates to null, a null pointer expression is thrown. If this expression evaluates to an address, a lock must be obtained on this address, the sync expression is rewritten to insync. For insync expressions, the body expression may be evaluated. If the body expression is only a value or a thrown exception, the lock is released and the synchronized expression reduces to the body's expression. This is the normal Java semantics, not the one as presented in LNCS 1523, Cenciarelli/Knapp/Reus/Wirsing. There the expression on which the thread synchronized is evaluated except for the last step. If the thread can obtain the lock on the object immediately after the last evaluation step, the evaluation is done and the lock acquired. If the lock cannot be obtained, the evaluation step is discarded. If another thread changes the evaluation result of this last step, the thread then will try to synchronize on the new object.

context *J-heap-base* **begin**

inductive *red* ::
(('*addr*, '*thread-id*, '*heap*) *external-thread-action* ⇒ ('*addr*, '*thread-id*, '*x*,'*heap*) *Jinja-thread-action*)
⇒ *'addr J-prog* ⇒ *'thread-id*
⇒ *'addr expr* ⇒ ('*addr*, '*heap*) *Jstate*
⇒ ('*addr*, '*thread-id*, '*x*,'*heap*) *Jinja-thread-action*
⇒ *'addr expr* ⇒ ('*addr*, '*heap*) *Jstate* ⇒ *bool*
(‹-,-,- ⊢ ((*1*⟨-,/-⟩) −-→/ (*1*⟨-,/-⟩))› [*51*,*51*,*0*,*0*,*0*,*0*,*0*,*0*] *81*)
and *reds* ::
(('*addr*, '*thread-id*, '*heap*) *external-thread-action* ⇒ ('*addr*, '*thread-id*, '*x*,'*heap*) *Jinja-thread-action*)
⇒ *'addr J-prog* ⇒ *'thread-id*
⇒ *'addr expr list* ⇒ ('*addr*, '*heap*) *Jstate*
⇒ ('*addr*, '*thread-id*, '*x*,'*heap*) *Jinja-thread-action*
⇒ *'addr expr list* ⇒ ('*addr*, '*heap*) *Jstate* ⇒ *bool*
(‹-,-,- ⊢ ((*1*⟨-,/-⟩) [−-→]/ (*1*⟨-,/-⟩))› [*51*,*51*,*0*,*0*,*0*,*0*,*0*,*0*] *81*)
for *extTA* :: ('*addr*, '*thread-id*, '*heap*) *external-thread-action* ⇒ ('*addr*, '*thread-id*, '*x*, '*heap*) *Jinja-thread-action*
and *P* :: *'addr J-prog* **and** *t* :: *'thread-id*
where

RedNew:
$(h', a) \in allocate\ h\ (Class\text{-}type\ C)$
$\Longrightarrow extTA,P,t \vdash \langle new\ C, (h, l)\rangle -\{\!|NewHeapElem\ a\ (Class\text{-}type\ C)|\!\}\rightarrow \langle addr\ a, (h', l)\rangle$

| *RedNewFail*:
$allocate\ h\ (Class\text{-}type\ C) = \{\}$
$\Longrightarrow extTA,P,t \vdash \langle new\ C, (h, l)\rangle -\varepsilon\rightarrow \langle THROW\ OutOfMemory, (h, l)\rangle$

| *NewArrayRed*:
$extTA,P,t \vdash \langle e, s\rangle -ta\rightarrow \langle e', s'\rangle \Longrightarrow extTA,P,t \vdash \langle newA\ T\lfloor e\rceil, s\rangle -ta\rightarrow \langle newA\ T\lfloor e'\rceil, s'\rangle$

| *RedNewArray*:
$[\![\ 0 <=s\ i;\ (h', a) \in allocate\ h\ (Array\text{-}type\ T\ (nat\ (sint\ i)))\ ]\!]$
$\Longrightarrow extTA,P,t \vdash \langle newA\ T\lfloor Val\ (Intg\ i)\rceil, (h, l)\rangle -\{\!|NewHeapElem\ a\ (Array\text{-}type\ T\ (nat\ (sint\ i)))|\!\}\rightarrow$
$\langle addr\ a, (h', l)\rangle$

| *RedNewArrayNegative*:
$i <s\ 0 \Longrightarrow extTA,P,t \vdash \langle newA\ T\lfloor Val\ (Intg\ i)\rceil, s\rangle -\varepsilon\rightarrow \langle THROW\ NegativeArraySize, s\rangle$

| *RedNewArrayFail*:
$[\![\ 0 <=s\ i;\ allocate\ h\ (Array\text{-}type\ T\ (nat\ (sint\ i))) = \{\}\ ]\!]$
$\Longrightarrow extTA,P,t \vdash \langle newA\ T\lfloor Val\ (Intg\ i)\rceil, (h, l)\rangle -\varepsilon\rightarrow \langle THROW\ OutOfMemory, (h, l)\rangle$

| *CastRed*:
$extTA,P,t \vdash \langle e, s\rangle -ta\rightarrow \langle e', s'\rangle \Longrightarrow extTA,P,t \vdash \langle Cast\ C\ e, s\rangle -ta\rightarrow \langle Cast\ C\ e', s'\rangle$

| *RedCast*:
$[\![\ typeof_{hp\ s}\ v = \lfloor U\rfloor;\ P \vdash U \leq T\ ]\!]$
$\Longrightarrow extTA,P,t \vdash \langle Cast\ T\ (Val\ v), s\rangle -\varepsilon\rightarrow \langle Val\ v, s\rangle$

| *RedCastFail*:
$[\![\ typeof_{hp\ s}\ v = \lfloor U\rfloor;\ \neg\ P \vdash U \leq T\ ]\!]$
$\Longrightarrow extTA,P,t \vdash \langle Cast\ T\ (Val\ v), s\rangle -\varepsilon\rightarrow \langle THROW\ ClassCast, s\rangle$

| *InstanceOfRed*:
$extTA,P,t \vdash \langle e, s\rangle -ta\rightarrow \langle e', s'\rangle \Longrightarrow extTA,P,t \vdash \langle e\ instanceof\ T, s\rangle -ta\rightarrow \langle e'\ instanceof\ T, s'\rangle$

| *RedInstanceOf*:
$[\![\ typeof_{hp\ s}\ v = \lfloor U\rfloor;\ b \longleftrightarrow v \neq Null \wedge P \vdash U \leq T\ ]\!]$
$\Longrightarrow extTA,P,t \vdash \langle (Val\ v)\ instanceof\ T, s\rangle -\varepsilon\rightarrow \langle Val\ (Bool\ b), s\rangle$

| *BinOpRed1*:
$extTA,P,t \vdash \langle e, s\rangle -ta\rightarrow \langle e', s'\rangle \Longrightarrow extTA,P,t \vdash \langle e\ «bop»\ e2, s\rangle -ta\rightarrow \langle e'\ «bop»\ e2, s'\rangle$

| *BinOpRed2*:
$extTA,P,t \vdash \langle e, s\rangle -ta\rightarrow \langle e', s'\rangle \Longrightarrow extTA,P,t \vdash \langle (Val\ v)\ «bop»\ e, s\rangle -ta\rightarrow \langle (Val\ v)\ «bop»\ e', s'\rangle$

| *RedBinOp*:
$binop\ bop\ v1\ v2 = Some\ (Inl\ v) \Longrightarrow$
$extTA,P,t \vdash \langle (Val\ v1)\ «bop»\ (Val\ v2), s\rangle -\varepsilon\rightarrow \langle Val\ v, s\rangle$

| *RedBinOpFail*:
$binop\ bop\ v1\ v2 = Some\ (Inr\ a) \Longrightarrow$
$extTA,P,t \vdash \langle (Val\ v1)\ «bop»\ (Val\ v2), s\rangle -\varepsilon\rightarrow \langle Throw\ a, s\rangle$

| *RedVar*:
lcl s V = Some v $\Longrightarrow$
extTA,P,t $\vdash \langle Var\ V, s\rangle -\varepsilon\rightarrow \langle Val\ v, s\rangle$

| *LAssRed*:
extTA,P,t $\vdash \langle e, s\rangle -ta\rightarrow \langle e', s'\rangle \Longrightarrow$ *extTA,P,t* $\vdash \langle V{:=}e, s\rangle -ta\rightarrow \langle V{:=}e', s'\rangle$

| *RedLAss*:
extTA,P,t $\vdash \langle V{:=}(Val\ v), (h, l)\rangle -\varepsilon\rightarrow \langle unit, (h, l(V \mapsto v))\rangle$

| *AAccRed1*:
extTA,P,t $\vdash \langle a, s\rangle -ta\rightarrow \langle a', s'\rangle \Longrightarrow$ *extTA,P,t* $\vdash \langle a\lfloor i\rceil, s\rangle -ta\rightarrow \langle a'\lfloor i\rceil, s'\rangle$

| *AAccRed2*:
extTA,P,t $\vdash \langle i, s\rangle -ta\rightarrow \langle i', s'\rangle \Longrightarrow$ *extTA,P,t* $\vdash \langle (Val\ a)\lfloor i\rceil, s\rangle -ta\rightarrow \langle (Val\ a)\lfloor i'\rceil, s'\rangle$

| *RedAAccNull*:
extTA,P,t $\vdash \langle null\lfloor Val\ i\rceil, s\rangle -\varepsilon\rightarrow \langle THROW\ NullPointer, s\rangle$

| *RedAAccBounds*:
$[\![$ *typeof-addr (hp s) a* $= \lfloor$*Array-type T n*$\rfloor$; *i* $<_s$ *0* $\vee$ *sint i* $\geq$ *int n* $]\!]$
$\Longrightarrow$ *extTA,P,t* $\vdash \langle (addr\ a)\lfloor Val\ (Intg\ i)\rceil, s\rangle -\varepsilon\rightarrow \langle THROW\ ArrayIndexOutOfBounds, s\rangle$

| *RedAAcc*:
$[\![$ *typeof-addr h a* $= \lfloor$*Array-type T n*$\rfloor$; *0* $<=_s$ *i*; *sint i* $<$ *int n*;
heap-read h a (ACell (nat (sint i))) v $]\!]$
$\Longrightarrow$ *extTA,P,t* $\vdash \langle (addr\ a)\lfloor Val\ (Intg\ i)\rceil, (h, l)\rangle -\{\!|ReadMem\ a\ (ACell\ (nat\ (sint\ i)))\ v|\!\}\rightarrow \langle Val\ v, (h, l)\rangle$

| *AAssRed1*:
extTA,P,t $\vdash \langle a, s\rangle -ta\rightarrow \langle a', s'\rangle \Longrightarrow$ *extTA,P,t* $\vdash \langle a\lfloor i\rceil := e, s\rangle -ta\rightarrow \langle a'\lfloor i\rceil := e, s'\rangle$

| *AAssRed2*:
extTA,P,t $\vdash \langle i, s\rangle -ta\rightarrow \langle i', s'\rangle \Longrightarrow$ *extTA,P,t* $\vdash \langle (Val\ a)\lfloor i\rceil := e, s\rangle -ta\rightarrow \langle (Val\ a)\lfloor i'\rceil := e, s'\rangle$

| *AAssRed3*:
extTA,P,t $\vdash \langle (e::'addr\ expr), s\rangle -ta\rightarrow \langle e', s'\rangle \Longrightarrow$ *extTA,P,t* $\vdash \langle (Val\ a)\lfloor Val\ i\rceil := e, s\rangle -ta\rightarrow \langle (Val\ a)\lfloor Val\ i\rceil := e', s'\rangle$

| *RedAAssNull*:
extTA,P,t $\vdash \langle null\lfloor Val\ i\rceil := (Val\ e::'addr\ expr), s\rangle -\varepsilon\rightarrow \langle THROW\ NullPointer, s\rangle$

| *RedAAssBounds*:
$[\![$ *typeof-addr (hp s) a* $= \lfloor$*Array-type T n*$\rfloor$; *i* $<_s$ *0* $\vee$ *sint i* $\geq$ *int n* $]\!]$
$\Longrightarrow$ *extTA,P,t* $\vdash \langle (addr\ a)\lfloor Val\ (Intg\ i)\rceil := (Val\ e::'addr\ expr), s\rangle -\varepsilon\rightarrow \langle THROW\ ArrayIndexOutOfBounds, s\rangle$

| *RedAAssStore*:
$[\![$ *typeof-addr (hp s) a* $= \lfloor$*Array-type T n*$\rfloor$; *0* $<=_s$ *i*; *sint i* $<$ *int n*;
$typeof_{hp\ s}\ w = \lfloor U\rfloor$; $\neg\ (P \vdash U \leq T)$ $]\!]$
$\Longrightarrow$ *extTA,P,t* $\vdash \langle (addr\ a)\lfloor Val\ (Intg\ i)\rceil := (Val\ w::'addr\ expr), s\rangle -\varepsilon\rightarrow \langle THROW\ ArrayStore, s\rangle$

| *RedAAss*:

⟦ *typeof-addr h a* = ⌊*Array-type T n*⌋; *0* <=*s i*; *sint i* < *int n*; *typeof*$_h$ *w* = *Some U*; *P* ⊢ *U* ≤ *T*; *heap-write h a* (*ACell* (*nat* (*sint i*))) *w h′* ⟧
⟹ *extTA,P,t* ⊢ ⟨(*addr a*)⌊*Val* (*Intg i*)⌉ := *Val w*::′*addr expr*, (*h*, *l*)⟩ −{| *WriteMem a* (*ACell* (*nat* (*sint i*))) *w*|}→ ⟨*unit*, (*h′*, *l*)⟩

| *ALengthRed*:
extTA,P,t ⊢ ⟨*a*, *s*⟩ −*ta*→ ⟨*a′*, *s′*⟩ ⟹ *extTA,P,t* ⊢ ⟨*a*·*length*, *s*⟩ −*ta*→ ⟨*a′*·*length*, *s′*⟩

| *RedALength*:
typeof-addr h a = ⌊*Array-type T n*⌋
⟹ *extTA,P,t* ⊢ ⟨*addr a*·*length*, (*h*, *l*)⟩ −*ε*→ ⟨*Val* (*Intg* (*word-of-nat n*)), (*h*, *l*)⟩

| *RedALengthNull*:
extTA,P,t ⊢ ⟨*null*·*length*, *s*⟩ −*ε*→ ⟨*THROW NullPointer*, *s*⟩

| *FAccRed*:
extTA,P,t ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩ ⟹ *extTA,P,t* ⊢ ⟨*e*·*F*{*D*}, *s*⟩ −*ta*→ ⟨*e′*·*F*{*D*}, *s′*⟩

| *RedFAcc*:
heap-read h a (*CField D F*) *v*
⟹ *extTA,P,t* ⊢ ⟨(*addr a*)·*F*{*D*}, (*h*, *l*)⟩ −{| *ReadMem a* (*CField D F*) *v*|}→ ⟨*Val v*, (*h*, *l*)⟩

| *RedFAccNull*:
extTA,P,t ⊢ ⟨*null*·*F*{*D*}, *s*⟩ −*ε*→ ⟨*THROW NullPointer*, *s*⟩

| *FAssRed1*:
extTA,P,t ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩ ⟹ *extTA,P,t* ⊢ ⟨*e*·*F*{*D*}:=*e2*, *s*⟩ −*ta*→ ⟨*e′*·*F*{*D*}:=*e2*, *s′*⟩

| *FAssRed2*:
extTA,P,t ⊢ ⟨(*e*::′*addr expr*), *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩ ⟹ *extTA,P,t* ⊢ ⟨*Val v*·*F*{*D*}:=*e*, *s*⟩ −*ta*→ ⟨*Val v*·*F*{*D*}:=*e′*, *s′*⟩

| *RedFAss*:
heap-write h a (*CField D F*) *v h′* ⟹
extTA,P,t ⊢ ⟨(*addr a*)·*F*{*D*}:= *Val v*, (*h*, *l*)⟩ −{| *WriteMem a* (*CField D F*) *v*|}→ ⟨*unit*, (*h′*, *l*)⟩

| *RedFAssNull*:
extTA,P,t ⊢ ⟨*null*·*F*{*D*}:=*Val v*::′*addr expr*, *s*⟩ −*ε*→ ⟨*THROW NullPointer*, *s*⟩

| *CASRed1*:
extTA,P,t ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩ ⟹
extTA,P,t ⊢ ⟨*e*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *s*⟩ −*ta*→ ⟨*e′*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *s′*⟩

| *CASRed2*:
extTA,P,t ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩ ⟹
extTA,P,t ⊢ ⟨*Val v*·*compareAndSwap*(*D*·*F*, *e*, *e3*), *s*⟩ −*ta*→ ⟨*Val v*·*compareAndSwap*(*D*·*F*, *e′*, *e3*), *s′*⟩

| *CASRed3*:
extTA,P,t ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩ ⟹
extTA,P,t ⊢ ⟨*Val v*·*compareAndSwap*(*D*·*F*, *Val v′*, *e*), *s*⟩ −*ta*→ ⟨*Val v*·*compareAndSwap*(*D*·*F*, *Val v′*, *e′*), *s′*⟩

| *CASNull*:

$extTA,P,t \vdash \langle null{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v,\ Val\ v'),\ s\rangle\ -\varepsilon\rightarrow\ \langle THROW\ NullPointer,\ s\rangle$

| *RedCASSucceed*:
$[\![\ heap\text{-}read\ h\ a\ (CField\ D\ F)\ v;\ heap\text{-}write\ h\ a\ (CField\ D\ F)\ v'\ h'\ ]\!] \Longrightarrow$
$extTA,P,t \vdash \langle addr\ a{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v,\ Val\ v'),\ (h,\ l)\rangle$
$-\{\!|ReadMem\ a\ (CField\ D\ F)\ v,\ WriteMem\ a\ (CField\ D\ F)\ v'|\!\}\rightarrow$
$\langle true,\ (h',\ l)\rangle$

| *RedCASFail*:
$[\![\ heap\text{-}read\ h\ a\ (CField\ D\ F)\ v'';\ v \neq v''\ ]\!] \Longrightarrow$
$extTA,P,t \vdash \langle addr\ a{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v,\ Val\ v'),\ (h,\ l)\rangle$
$-\{\!|ReadMem\ a\ (CField\ D\ F)\ v''|\!\}\rightarrow$
$\langle false,\ (h,\ l)\rangle$

| *CallObj*:
$extTA,P,t \vdash \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle \Longrightarrow extTA,P,t \vdash \langle e{\cdot}M(es),\ s\rangle\ -ta\rightarrow\ \langle e'{\cdot}M(es),\ s'\rangle$

| *CallParams*:
$extTA,P,t \vdash \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',s'\rangle \Longrightarrow$
$extTA,P,t \vdash \langle (Val\ v){\cdot}M(es),s\rangle\ -ta\rightarrow\ \langle (Val\ v){\cdot}M(es'),s'\rangle$

| *RedCall*:
$[\![\ typeof\text{-}addr\ (hp\ s)\ a = \lfloor hU\rfloor;\ P \vdash class\text{-}type\text{-}of\ hU\ sees\ M{:}Ts\rightarrow T = \lfloor(pns,body)\rfloor\ in\ D;$
$size\ vs = size\ pns;\ size\ Ts = size\ pns\ ]\!]$
$\Longrightarrow extTA,P,t \vdash \langle (addr\ a){\cdot}M(map\ Val\ vs),\ s\rangle\ -\varepsilon\rightarrow\ \langle blocks\ (this\ \#\ pns)\ (Class\ D\ \#\ Ts)\ (Addr\ a\ \#\ vs)\ body,\ s\rangle$

| *RedCallExternal*:
$[\![\ typeof\text{-}addr\ (hp\ s)\ a = \lfloor hU\rfloor;\ P \vdash class\text{-}type\text{-}of\ hU\ sees\ M{:}Ts\rightarrow T = Native\ in\ D;$
$P,t \vdash \langle a{\cdot}M(vs),\ hp\ s\rangle\ -ta\rightarrow ext\ \langle va,\ h'\rangle;$
$ta' = extTA\ ta;\ e' = extRet2J\ ((addr\ a){\cdot}M(map\ Val\ vs))\ va;\ s' = (h',\ lcl\ s)\ ]\!]$
$\Longrightarrow extTA,P,t \vdash \langle (addr\ a){\cdot}M(map\ Val\ vs),\ s\rangle\ -ta'\rightarrow\ \langle e',\ s'\rangle$

| *RedCallNull*:
$extTA,P,t \vdash \langle null{\cdot}M(map\ Val\ vs),\ s\rangle\ -\varepsilon\rightarrow\ \langle THROW\ NullPointer,\ s\rangle$

| *BlockRed*:
$extTA,P,t \vdash \langle e,\ (h,\ l(V{:=}vo))\rangle\ -ta\rightarrow\ \langle e',\ (h',\ l')\rangle$
$\Longrightarrow extTA,P,t \vdash \langle\{V{:}T{=}vo;\ e\},\ (h,\ l)\rangle\ -ta\rightarrow\ \langle\{V{:}T{=}l'\ V;\ e'\},\ (h',\ l'(V\ :=\ l\ V))\rangle$

| *RedBlock*:
$extTA,P,t \vdash \langle\{V{:}T{=}vo;\ Val\ u\},\ s\rangle\ -\varepsilon\rightarrow\ \langle Val\ u,\ s\rangle$

| *SynchronizedRed1*:
$extTA,P,t \vdash \langle o',\ s\rangle\ -ta\rightarrow\ \langle o'',\ s'\rangle \Longrightarrow extTA,P,t \vdash \langle sync(o')\ e,\ s\rangle\ -ta\rightarrow\ \langle sync(o'')\ e,\ s'\rangle$

| *SynchronizedNull*:
$extTA,P,t \vdash \langle sync(null)\ e,\ s\rangle\ -\varepsilon\rightarrow\ \langle THROW\ NullPointer,\ s\rangle$

| *LockSynchronized*:
$extTA,P,t \vdash \langle sync(addr\ a)\ e,\ s\rangle\ -\{\!|Lock\rightarrow a,\ SyncLock\ a|\!\}\rightarrow\ \langle insync(a)\ e,\ s\rangle$

| *SynchronizedRed2*:
$extTA,P,t \vdash \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle \Longrightarrow extTA,P,t \vdash \langle insync(a)\ e,\ s\rangle\ -ta\rightarrow\ \langle insync(a)\ e',\ s'\rangle$

| *UnlockSynchronized*:
$extTA,P,t \vdash \langle insync(a)\ (Val\ v),\ s\rangle\ -\{\!|\ Unlock{\rightarrow}a,\ SyncUnlock\ a\ |\!\}{\rightarrow}\ \langle Val\ v,\ s\rangle$

| *SeqRed*:
$extTA,P,t \vdash \langle e,\ s\rangle\ -ta{\rightarrow}\ \langle e',\ s'\rangle \Longrightarrow extTA,P,t \vdash \langle e;;e2,\ s\rangle\ -ta{\rightarrow}\ \langle e';;e2,\ s'\rangle$

| *RedSeq*:
$extTA,P,t \vdash \langle (Val\ v);;e,\ s\rangle\ -\varepsilon{\rightarrow}\ \langle e,\ s\rangle$

| *CondRed*:
$extTA,P,t \vdash \langle b,\ s\rangle\ -ta{\rightarrow}\ \langle b',\ s'\rangle \Longrightarrow extTA,P,t \vdash \langle if\ (b)\ e1\ else\ e2,\ s\rangle\ -ta{\rightarrow}\ \langle if\ (b')\ e1\ else\ e2,\ s'\rangle$

| *RedCondT*:
$extTA,P,t \vdash \langle if\ (true)\ e1\ else\ e2,\ s\rangle\ -\varepsilon{\rightarrow}\ \langle e1,\ s\rangle$

| *RedCondF*:
$extTA,P,t \vdash \langle if\ (false)\ e1\ else\ e2,\ s\rangle\ -\varepsilon{\rightarrow}\ \langle e2,\ s\rangle$

| *RedWhile*:
$extTA,P,t \vdash \langle while(b)\ c,\ s\rangle\ -\varepsilon{\rightarrow}\ \langle if\ (b)\ (c;;while(b)\ c)\ else\ unit,\ s\rangle$

| *ThrowRed*:
$extTA,P,t \vdash \langle e,\ s\rangle\ -ta{\rightarrow}\ \langle e',\ s'\rangle \Longrightarrow extTA,P,t \vdash \langle throw\ e,\ s\rangle\ -ta{\rightarrow}\ \langle throw\ e',\ s'\rangle$

| *RedThrowNull*:
$extTA,P,t \vdash \langle throw\ null,\ s\rangle\ -\varepsilon{\rightarrow}\ \langle THROW\ NullPointer,\ s\rangle$

| *TryRed*:
$extTA,P,t \vdash \langle e,\ s\rangle\ -ta{\rightarrow}\ \langle e',\ s'\rangle \Longrightarrow extTA,P,t \vdash \langle try\ e\ catch(C\ V)\ e2,\ s\rangle\ -ta{\rightarrow}\ \langle try\ e'\ catch(C\ V)\ e2,\ s'\rangle$

| *RedTry*:
$extTA,P,t \vdash \langle try\ (Val\ v)\ catch(C\ V)\ e2,\ s\rangle\ -\varepsilon{\rightarrow}\ \langle Val\ v,\ s\rangle$

| *RedTryCatch*:
$[\![\ typeof\text{-}addr\ (hp\ s)\ a = \lfloor Class\text{-}type\ D\rfloor;\ P \vdash D \preceq^*\ C\ ]\!]$
$\Longrightarrow extTA,P,t \vdash \langle try\ (Throw\ a)\ catch(C\ V)\ e2,\ s\rangle\ -\varepsilon{\rightarrow}\ \langle\{V{:}Class\ C{=}\lfloor Addr\ a\rfloor;\ e2\},\ s\rangle$

| *RedTryFail*:
$[\![\ typeof\text{-}addr\ (hp\ s)\ a = \lfloor Class\text{-}type\ D\rfloor;\ \neg\ P \vdash D \preceq^*\ C\ ]\!]$
$\Longrightarrow extTA,P,t \vdash \langle try\ (Throw\ a)\ catch(C\ V)\ e2,\ s\rangle\ -\varepsilon{\rightarrow}\ \langle Throw\ a,\ s\rangle$

| *ListRed1*:
$extTA,P,t \vdash \langle e,s\rangle\ -ta{\rightarrow}\ \langle e',s'\rangle \Longrightarrow$
$extTA,P,t \vdash \langle e\#es,s\rangle\ [-ta{\rightarrow}]\ \langle e'\#es,s'\rangle$

| *ListRed2*:
$extTA,P,t \vdash \langle es,s\rangle\ [-ta{\rightarrow}]\ \langle es',s'\rangle \Longrightarrow$
$extTA,P,t \vdash \langle Val\ v\ \#\ es,s\rangle\ [-ta{\rightarrow}]\ \langle Val\ v\ \#\ es',s'\rangle$

— Exception propagation

| *NewArrayThrow*: $extTA,P,t \vdash \langle newA\ T\lfloor Throw\ a\rceil,\ s\rangle\ -\varepsilon{\rightarrow}\ \langle Throw\ a,\ s\rangle$

| *CastThrow*: $extTA,P,t \vdash \langle Cast\ C\ (Throw\ a),\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *InstanceOfThrow*: $extTA,P,t \vdash \langle (Throw\ a)\ instanceof\ T,\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *BinOpThrow1*: $extTA,P,t \vdash \langle (Throw\ a)\ «bop»\ e_2,\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *BinOpThrow2*: $extTA,P,t \vdash \langle (Val\ v_1)\ «bop»\ (Throw\ a),\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *LAssThrow*: $extTA,P,t \vdash \langle V{:=}(Throw\ a),\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *AAccThrow1*: $extTA,P,t \vdash \langle (Throw\ a)\lfloor i\rceil,\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *AAccThrow2*: $extTA,P,t \vdash \langle (Val\ v)\lfloor Throw\ a\rceil,\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *AAssThrow1*: $extTA,P,t \vdash \langle (Throw\ a)\lfloor i\rceil := e,\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *AAssThrow2*: $extTA,P,t \vdash \langle (Val\ v)\lfloor Throw\ a\rceil := e,\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *AAssThrow3*: $extTA,P,t \vdash \langle (Val\ v)\lfloor Val\ i\rceil := Throw\ a :: 'addr\ expr,\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *ALengthThrow*: $extTA,P,t \vdash \langle (Throw\ a){\cdot}length,\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *FAccThrow*: $extTA,P,t \vdash \langle (Throw\ a){\cdot}F\{D\},\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *FAssThrow1*: $extTA,P,t \vdash \langle (Throw\ a){\cdot}F\{D\}{:=}e_2,\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *FAssThrow2*: $extTA,P,t \vdash \langle Val\ v{\cdot}F\{D\}{:=}(Throw\ a{::}'addr\ expr),\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *CASThrow*: $extTA,P,t \vdash \langle Throw\ a{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3),\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *CASThrow2*: $extTA,P,t \vdash \langle Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ Throw\ a,\ e3),\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *CASThrow3*: $extTA,P,t \vdash \langle Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v',\ Throw\ a),\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *CallThrowObj*: $extTA,P,t \vdash \langle (Throw\ a){\cdot}M(es),\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *CallThrowParams*: $[\![\ es = map\ Val\ vs\ @\ Throw\ a\ \#\ es'\ ]\!] \Longrightarrow extTA,P,t \vdash \langle (Val\ v){\cdot}M(es),\ s\rangle\ -\varepsilon\rightarrow \langle Throw\ a,\ s\rangle$
| *BlockThrow*: $extTA,P,t \vdash \langle \{V{:}T{=}vo;\ Throw\ a\},\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *SynchronizedThrow1*: $extTA,P,t \vdash \langle sync(Throw\ a)\ e,\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *SynchronizedThrow2*: $extTA,P,t \vdash \langle insync(a)\ Throw\ ad,\ s\rangle\ -\{\!|Unlock{\rightarrow}a,\ SyncUnlock\ a|\!\}\rightarrow\ \langle Throw\ ad,\ s\rangle$
| *SeqThrow*: $extTA,P,t \vdash \langle (Throw\ a);;e_2,\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *CondThrow*: $extTA,P,t \vdash \langle if\ (Throw\ a)\ e_1\ else\ e_2,\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$
| *ThrowThrow*: $extTA,P,t \vdash \langle throw(Throw\ a),\ s\rangle\ -\varepsilon\rightarrow\ \langle Throw\ a,\ s\rangle$

inductive-cases *red-cases*:
$extTA,P,t \vdash \langle new\ C,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
$extTA,P,t \vdash \langle newA\ T\lfloor e\rceil,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
$extTA,P,t \vdash \langle Cast\ T\ e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
$extTA,P,t \vdash \langle e\ instanceof\ T,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
$extTA,P,t \vdash \langle e\ «bop»\ e',\ s\rangle\ -ta\rightarrow\ \langle e'',\ s'\rangle$
$extTA,P,t \vdash \langle Var\ V,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
$extTA,P,t \vdash \langle V{:=}e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
$extTA,P,t \vdash \langle a\lfloor i\rceil,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
$extTA,P,t \vdash \langle a\lfloor i\rceil := e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
$extTA,P,t \vdash \langle a{\cdot}length,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
$extTA,P,t \vdash \langle e{\cdot}F\{D\},\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
$extTA,P,t \vdash \langle e{\cdot}F\{D\} := e',\ s\rangle\ -ta\rightarrow\ \langle e'',\ s'\rangle$
$extTA,P,t \vdash \langle e{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e''),\ s\rangle\ -ta\rightarrow\ \langle e''',\ s'\rangle$
$extTA,P,t \vdash \langle e{\cdot}M(es),\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
$extTA,P,t \vdash \langle \{V{:}T{=}vo;\ e\},\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
$extTA,P,t \vdash \langle sync(o')\ e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
$extTA,P,t \vdash \langle insync(a)\ e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
$extTA,P,t \vdash \langle e;;e',\ s\rangle\ -ta\rightarrow\ \langle e'',\ s'\rangle$
$extTA,P,t \vdash \langle if\ (b)\ e1\ else\ e2,\ s\ \rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
$extTA,P,t \vdash \langle while\ (b)\ e,\ s\ \rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
$extTA,P,t \vdash \langle throw\ e,\ s\ \rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
$extTA,P,t \vdash \langle try\ e\ catch(C\ V)\ e',\ s\rangle\ -ta\rightarrow\ \langle e'',\ s'\rangle$

inductive-cases *reds-cases*:
$extTA,P,t \vdash \langle e\ \#\ es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle$

abbreviation *red'* ::
'addr J-prog ⇒ *'thread-id* ⇒ *'addr expr* ⇒ (*'heap* × *'addr locals*)
⇒ (*'addr*, *'thread-id*, *'heap*) *J-thread-action* ⇒ *'addr expr* ⇒ (*'heap* × *'addr locals*) ⇒ *bool*
(‹-,- ⊢ ((*1*⟨-,/-⟩) -▪→/ (*1*⟨-,/-⟩))› [*51*,*0*,*0*,*0*,*0*,*0*,*0*] *81*)
where *red' P* ≡ *red* (*extTA2J P*) *P*

abbreviation *reds'* ::
'addr J-prog ⇒ *'thread-id* ⇒ *'addr expr list* ⇒ (*'heap* × *'addr locals*)
⇒ (*'addr*, *'thread-id*, *'heap*) *J-thread-action* ⇒ *'addr expr list* ⇒ (*'heap* × *'addr locals*) ⇒ *bool*
(‹-,- ⊢ ((*1*⟨-,/-⟩) [-▪→]/ (*1*⟨-,/-⟩))› [*51*,*0*,*0*,*0*,*0*,*0*,*0*] *81*)
where *reds' P* ≡ *reds* (*extTA2J P*) *P*

4.4.1 Some easy lemmas

lemma [*iff*]:
¬ *extTA*,*P*,*t* ⊢ ⟨*Val v*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩
by(*fastforce elim*:*red.cases*)

lemma *red-no-val* [*dest*]:
⟦ *extTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*tas*→ ⟨*e'*, *s'*⟩; *is-val e* ⟧ ⟹ *False*
by(*auto*)

lemma [*iff*]: ¬ *extTA*,*P*,*t* ⊢ ⟨*Throw a*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩
by(*fastforce elim*: *red-cases*)

lemma *reds-map-Val-Throw*:
extTA,*P*,*t* ⊢ ⟨*map Val vs* @ *Throw a* # *es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩ ⟷ *False*
by(*induct vs arbitrary*: *es'*)(*auto elim*!: *reds-cases*)

lemma *reds-preserves-len*:
extTA,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩ ⟹ *length es'* = *length es*
by(*induct es arbitrary*: *es'*)(*auto elim*: *reds.cases*)

lemma *red-lcl-incr*: *extTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩ ⟹ *dom* (*lcl s*) ⊆ *dom* (*lcl s'*)
and *reds-lcl-incr*: *extTA*,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩ ⟹ *dom* (*lcl s*) ⊆ *dom* (*lcl s'*)
apply(*induct rule*:*red-reds.inducts*)
apply(*auto simp del*: *fun-upd-apply split*: *if-split-asm*)
done

lemma *red-lcl-add-aux*:
extTA,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩ ⟹ *extTA*,*P*,*t* ⊢ ⟨*e*, (*hp s*, *l0* ++ *lcl s*)⟩ −*ta*→ ⟨*e'*, (*hp s'*, *l0* ++ *lcl s'*)⟩
and *reds-lcl-add-aux*:
extTA,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩ ⟹ *extTA*,*P*,*t* ⊢ ⟨*es*, (*hp s*, *l0* ++ *lcl s*)⟩ [−*ta*→] ⟨*es'*, (*hp s'*, *l0* ++ *lcl s'*)⟩
proof (*induct arbitrary*: *l0* **and** *l0 rule*:*red-reds.inducts*)
case (*BlockRed e h x V vo ta e' h' x' T*)
note *IH* = ‹⋀*l0*. *extTA*,*P*,*t* ⊢ ⟨*e*,(*hp* (*h*, *x*(*V* := *vo*)), *l0* ++ *lcl* (*h*, *x*(*V* := *vo*)))⟩ −*ta*→ ⟨*e'*,(*hp* (*h'*, *x'*), *l0* ++ *lcl* (*h'*, *x'*))⟩›[*simplified*]
have *lrew*: ⋀*x x'*. *x*(*V* := *vo*) ++ *x'*(*V* := *vo*) = (*x* ++ *x'*)(*V* := *vo*)
by(*simp add*:*fun-eq-iff map-add-def*)
have *lrew1*: ⋀*X X' X''* *vo*. (*X*(*V* := *vo*) ++ *X'*)(*V* := (*X* ++ *X''*) *V*) = *X* ++ *X'*(*V* := *X''* *V*)
by(*simp add*: *fun-eq-iff map-add-def*)

```
  have lrew2: ⋀X X'. (X(V := None) ++ X') V = X' V
    by(simp add: map-add-def)
  show ?case
  proof(cases vo)
    case None
    from IH[of l0(V := vo)]
    show ?thesis
      apply(simp del: fun-upd-apply add: lrew)
      apply(drule red-reds.BlockRed)
      by(simp only: lrew1 None lrew2)
  next
    case (Some v)
    with ‹extTA,P,t ⊢ ⟨e,(h, x(V := vo))⟩ −ta→ ⟨e',(h', x')⟩›
    have x' V ≠ None
      by −(drule red-lcl-incr, auto split: if-split-asm)
    with IH[of l0(V := vo)]
    show ?thesis
      apply(clarsimp simp del: fun-upd-apply simp add: lrew)
      apply(drule red-reds.BlockRed)
      by(simp add: lrew1 Some del: fun-upd-apply)
  qed
next
  case RedTryFail thus ?case
    by(auto intro: red-reds.RedTryFail)
qed(fastforce intro:red-reds.intros simp del: fun-upd-apply)+

lemma red-lcl-add: extTA,P,t ⊢ ⟨e, (h, l)⟩ −ta→ ⟨e', (h', l')⟩ ⟹ extTA,P,t ⊢ ⟨e, (h, l0 ++ l)⟩ −ta→
⟨e', (h', l0 ++ l')⟩
  and reds-lcl-add: extTA,P,t ⊢ ⟨es, (h, l)⟩ [−ta→] ⟨es', (h', l')⟩ ⟹ extTA,P,t ⊢ ⟨es, (h, l0 ++ l)⟩
[−ta→] ⟨es', (h', l0 ++ l')⟩
by(auto dest:red-lcl-add-aux reds-lcl-add-aux)

lemma reds-no-val [dest]:
  ⟦ extTA,P,t ⊢ ⟨es, s⟩ [−ta→] ⟨es', s'⟩; is-vals es ⟧ ⟹ False
apply(induct es arbitrary: s ta es' s')
 apply(blast elim: reds.cases)
apply(erule reds.cases)
apply(auto, blast)
done

lemma red-no-Throw [dest!]:
  extTA,P,t ⊢ ⟨Throw a, s⟩ −ta→ ⟨e', s'⟩ ⟹ False
by(auto elim!: red-cases)

lemma red-lcl-sub:
  ⟦ extTA,P,t ⊢ ⟨e, s⟩ −ta→ ⟨e', s'⟩; fv e ⊆ W ⟧
  ⟹ extTA,P,t ⊢ ⟨e, (hp s, (lcl s)|`W)⟩ −ta→ ⟨e', (hp s', (lcl s')|`W)⟩

  and reds-lcl-sub:
  ⟦ extTA,P,t ⊢ ⟨es, s⟩ [−ta→] ⟨es', s'⟩; fvs es ⊆ W ⟧
  ⟹ extTA,P,t ⊢ ⟨es, (hp s, (lcl s)|`W)⟩ [−ta→] ⟨es', (hp s', (lcl s')|`W)⟩
proof(induct arbitrary: W and W rule: red-reds.inducts)
  case (RedLAss V v h l W)
  have extTA,P,t ⊢ ⟨V:=Val v,(h, l |` W)⟩ −ε→ ⟨unit,(h, (l |`W)(V ↦ v))⟩
```

```
    by(rule red-reds.RedLAss)
  with RedLAss show ?case by(simp del: fun-upd-apply)
next
  case (BlockRed e h x V vo ta e' h' x' T)
  have IH: ⋀W. fv e ⊆ W ⟹ extTA,P,t ⊢ ⟨e,(hp (h, x(V := vo)), lcl (h, x(V := vo)) |‘ W)⟩ −ta→
⟨e',(hp (h', x'), lcl (h', x') |‘ W)⟩ by fact
  from ‹fv {V:T=vo; e} ⊆ W› have fve: fv e ⊆ insert V W by auto
  show ?case
  proof(cases V ∈ W)
    case True
    with fve have fv e ⊆ W by auto
     from True IH[OF this] have extTA,P,t ⊢ ⟨e,(h, (x |‘ W )(V := vo))⟩ −ta→ ⟨e',(h', x' |‘ W)⟩
by(simp)
    with True have extTA,P,t ⊢ ⟨{V:T=vo; e},(h, x |‘ W)⟩ −ta→ ⟨{V:T=x' V; e'},(h', (x' |‘ W)(V
:= x V))⟩
      by −(drule red-reds.BlockRed[where T=T], simp)
    with True show ?thesis by(simp del: fun-upd-apply)
  next
    case False
    with IH[OF fve] have extTA,P,t ⊢ ⟨e,(h, (x |‘ W)(V := vo))⟩ −ta→ ⟨e',(h', x' |‘ insert V W)⟩
by(simp)
    with False have extTA,P,t ⊢ ⟨{V:T=vo; e},(h, x |‘ W)⟩ −ta→ ⟨{V:T=x' V; e'},(h', (x' |‘ W))⟩
      by −(drule red-reds.BlockRed[where T=T],simp)
    with False show ?thesis by(simp del: fun-upd-apply)
  qed
next
  case RedTryFail thus ?case by(auto intro: red-reds.RedTryFail)
qed(fastforce intro: red-reds.intros)+

lemma red-notfree-unchanged: ⟦ extTA,P,t ⊢ ⟨e, s⟩ −ta→ ⟨e', s'⟩; V ∉ fv e ⟧ ⟹ lcl s' V = lcl s V
  and reds-notfree-unchanged: ⟦ extTA,P,t ⊢ ⟨es, s⟩ [−ta→] ⟨es', s'⟩; V ∉ fvs es ⟧ ⟹ lcl s' V = lcl s
V
apply(induct rule: red-reds.inducts)
apply(fastforce)+
done

lemma red-dom-lcl: extTA,P,t ⊢ ⟨e, s⟩ −ta→ ⟨e', s'⟩ ⟹ dom (lcl s') ⊆ dom (lcl s) ∪ fv e
  and reds-dom-lcl: extTA,P,t ⊢ ⟨es, s⟩ [−ta→] ⟨es', s'⟩ ⟹ dom (lcl s') ⊆ dom (lcl s) ∪ fvs es
proof (induct rule:red-reds.inducts)
  case (BlockRed e h x V vo ta e' h' x' T)
  thus ?case by(clarsimp)(fastforce split:if-split-asm)
qed auto

lemma red-Suspend-is-call:
  ⟦ convert-extTA extNTA,P,t ⊢ ⟨e, s⟩ −ta→ ⟨e', s'⟩; Suspend w ∈ set {|ta|}w ⟧
  ⟹ ∃a vs hT Ts Tr D. call e' = ⌊(a, wait, vs)⌋ ∧ typeof-addr (hp s) a = ⌊hT⌋ ∧ P ⊢ class-type-of
hT sees wait:Ts→Tr = Native in D
  and reds-Suspend-is-calls:
  ⟦ convert-extTA extNTA,P,t ⊢ ⟨es, s⟩ [−ta→] ⟨es', s'⟩; Suspend w ∈ set {|ta|}w ⟧
  ⟹ ∃a vs hT Ts Tr D. calls es' = ⌊(a, wait, vs)⌋ ∧ typeof-addr (hp s) a = ⌊hT⌋ ∧ P ⊢ class-type-of
hT sees wait:Ts→Tr = Native in D
proof(induct rule: red-reds.inducts)
  case RedCallExternal
  thus ?case
```

apply *clarsimp*
apply(*frule red-external-Suspend-StaySame*, *simp*)
apply(*drule red-external-Suspend-waitD*, *fastforce+*)
done
qed *auto*

end

context *J-heap* **begin**

lemma *red-hext-incr*: $extTA,P,t \vdash \langle e, s\rangle -ta\rightarrow \langle e', s'\rangle \Longrightarrow hp\ s \trianglelefteq hp\ s'$
and *reds-hext-incr*: $extTA,P,t \vdash \langle es, s\rangle\ [-ta\rightarrow]\ \langle es', s'\rangle \Longrightarrow hp\ s \trianglelefteq hp\ s'$
by(*induct rule:red-reds.inducts*)(*auto intro*: *hext-heap-ops red-external-hext*)

lemma *red-preserves-tconf*: $[\![extTA,P,t \vdash \langle e, s\rangle -ta\rightarrow \langle e', s'\rangle; P,hp\ s \vdash t\ \surd t]\!] \Longrightarrow P,hp\ s' \vdash t\ \surd t$
by(*drule red-hext-incr*)(*rule tconf-hext-mono*)

lemma *reds-preserves-tconf*: $[\![extTA,P,t \vdash \langle es, s\rangle\ [-ta\rightarrow]\ \langle es', s'\rangle; P,hp\ s \vdash t\ \surd t]\!] \Longrightarrow P,hp\ s' \vdash t\ \surd t$
by(*drule reds-hext-incr*)(*rule tconf-hext-mono*)

end

4.4.2 Code generation

context *J-heap-base* **begin**

lemma *RedCall-code*:
$[\![$ *is-vals es*; *typeof-addr* (*hp s*) *a* = $\lfloor hU \rfloor$; $P \vdash$ *class-type-of hU sees M*:*Ts*$\rightarrow$*T* = $\lfloor$(*pns*,*body*)$\rfloor$ *in D*;
size es = *size pns*; *size Ts* = *size pns* $]\!]$
$\Longrightarrow$ $extTA,P,t \vdash \langle$(*addr a*)$\cdot$*M*(*es*), *s*$\rangle -\varepsilon\rightarrow \langle$*blocks* (*this* # *pns*) (*Class D* # *Ts*) (*Addr a* # *map the-Val es*) *body*, *s*$\rangle$

and *RedCallExternal-code*:
$[\![$ *is-vals es*; *typeof-addr* (*hp s*) *a* = $\lfloor hU \rfloor$; $P \vdash$ *class-type-of hU sees M*:*Ts*$\rightarrow$*T* = *Native in D*;
$P,t \vdash \langle a\cdot M($*map the-Val es*$)$, *hp s*$\rangle -ta\rightarrow ext\ \langle va, h'\rangle$ $]\!]$
$\Longrightarrow$ $extTA,P,t \vdash \langle$(*addr a*)$\cdot$*M*(*es*), *s*$\rangle -extTA\ ta\rightarrow \langle extRet2J\ ($(*addr a*)$\cdot$*M*(*es*)$)\ va, (h', lcl\ s)\rangle$

and *RedCallNull-code*:
is-vals es $\Longrightarrow$ $extTA,P,t \vdash \langle$*null*$\cdot$*M*(*es*), *s*$\rangle -\varepsilon\rightarrow \langle$*THROW NullPointer*, *s*$\rangle$

and *CallThrowParams-code*:
is-Throws es $\Longrightarrow$ $extTA,P,t \vdash \langle$(*Val v*)$\cdot$*M*(*es*), *s*$\rangle -\varepsilon\rightarrow \langle$*hd* (*dropWhile is-val es*), *s*$\rangle$

apply(*auto simp add*: *is-vals-conv is-Throws-conv o-def intro*: *RedCall RedCallExternal RedCallNull simp del*: *blocks.simps*)
apply(*subst dropWhile-append2*)
apply(*auto intro*: *CallThrowParams*)
done

end

lemmas [*code-pred-intro*] =
J-heap-base.RedNew[*folded Predicate-Compile.contains-def*] *J-heap-base.RedNewFail J-heap-base.NewArrayRed*

J-heap-base.RedNewArray[*folded Predicate-Compile.contains-def*]
J-heap-base.RedNewArrayNegative J-heap-base.RedNewArrayFail
J-heap-base.CastRed J-heap-base.RedCast J-heap-base.RedCastFail J-heap-base.InstanceOfRed
J-heap-base.RedInstanceOf J-heap-base.BinOpRed1 J-heap-base.BinOpRed2 J-heap-base.RedBinOp
J-heap-base.RedBinOpFail
J-heap-base.RedVar J-heap-base.LAssRed J-heap-base.RedLAss
J-heap-base.AAccRed1 J-heap-base.AAccRed2 J-heap-base.RedAAccNull
J-heap-base.RedAAccBounds J-heap-base.RedAAcc J-heap-base.AAssRed1 J-heap-base.AAssRed2 J-heap-base.AAssRed3
J-heap-base.RedAAssNull J-heap-base.RedAAssBounds J-heap-base.RedAAssStore J-heap-base.RedAAss
J-heap-base.ALengthRed
J-heap-base.RedALength J-heap-base.RedALengthNull J-heap-base.FAccRed J-heap-base.RedFAcc J-heap-base.RedFAccNull
J-heap-base.FAssRed1 J-heap-base.FAssRed2 J-heap-base.RedFAss J-heap-base.RedFAssNull
J-heap-base.CASRed1 J-heap-base.CASRed2 J-heap-base.CASRed3 J-heap-base.CASNull J-heap-base.RedCASSucceed
J-heap-base.RedCASFail
J-heap-base.CallObj J-heap-base.CallParams

declare
J-heap-base.RedCall-code[*code-pred-intro RedCall-code*]
J-heap-base.RedCallExternal-code[*code-pred-intro RedCallExternal-code*]
J-heap-base.RedCallNull-code[*code-pred-intro RedCallNull-code*]

lemmas [*code-pred-intro*] =
J-heap-base.BlockRed J-heap-base.RedBlock J-heap-base.SynchronizedRed1 J-heap-base.SynchronizedNull
J-heap-base.LockSynchronized J-heap-base.SynchronizedRed2 J-heap-base.UnlockSynchronized
J-heap-base.SeqRed J-heap-base.RedSeq J-heap-base.CondRed J-heap-base.RedCondT J-heap-base.RedCondF
J-heap-base.RedWhile
J-heap-base.ThrowRed

declare
J-heap-base.RedThrowNull[*code-pred-intro RedThrowNull'*]

lemmas [*code-pred-intro*] =
J-heap-base.TryRed J-heap-base.RedTry J-heap-base.RedTryCatch
J-heap-base.RedTryFail J-heap-base.ListRed1 J-heap-base.ListRed2
J-heap-base.NewArrayThrow J-heap-base.CastThrow J-heap-base.InstanceOfThrow J-heap-base.BinOpThrow1
J-heap-base.BinOpThrow2
J-heap-base.LAssThrow J-heap-base.AAccThrow1 J-heap-base.AAccThrow2 J-heap-base.AAssThrow1
J-heap-base.AAssThrow2
J-heap-base.AAssThrow3 J-heap-base.ALengthThrow J-heap-base.FAccThrow J-heap-base.FAssThrow1
J-heap-base.FAssThrow2
J-heap-base.CASThrow J-heap-base.CASThrow2 J-heap-base.CASThrow3
J-heap-base.CallThrowObj

declare
J-heap-base.CallThrowParams-code[*code-pred-intro CallThrowParams-code*]

lemmas [*code-pred-intro*] =
J-heap-base.BlockThrow J-heap-base.SynchronizedThrow1 J-heap-base.SynchronizedThrow2 J-heap-base.SeqThrow
J-heap-base.CondThrow

declare
J-heap-base.ThrowThrow[*code-pred-intro ThrowThrow'*]

code-pred
(*modes*:
J-heap-base.red: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow (i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool) \Rightarrow (i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool) \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$
and
J-heap-base.reds: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow (i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool) \Rightarrow (i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool) \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$)
[*detect-switches*, *skip-proof*] — proofs are possible, but take veeerry long
J-heap-base.red
proof −
case *red*
from *red.prems* **show** *thesis*
proof(*cases rule*: *J-heap-base.red.cases*[*consumes 1*, *case-names*
RedNew RedNewFail NewArrayRed RedNewArray RedNewArrayNegative RedNewArrayFail CastRed RedCast RedCastFail InstanceOfRed
RedInstanceOf BinOpRed1 BinOpRed2 RedBinOp RedBinOpFail RedVar LAssRed RedLAss
AAccRed1 AAccRed2 RedAAccNull RedAAccBounds RedAAcc
AAssRed1 AAssRed2 AAssRed3 RedAAssNull RedAAssBounds RedAAssStore RedAAss ALengthRed RedALength RedALengthNull FAccRed
RedFAcc RedFAccNull FAssRed1 FAssRed2 RedFAss RedFAssNull CASRed1 CASRed2 CASRed3 RedCASNull RedCASSucceed RedCASFail
CallObj CallParams RedCall RedCallExternal RedCallNull
BlockRed RedBlock SynchronizedRed1 SynchronizedNull LockSynchronized SynchronizedRed2 UnlockSynchronized SeqRed
RedSeq CondRed RedCondT RedCondF RedWhile ThrowRed RedThrowNull TryRed RedTry RedTryCatch RedTryFail
NewArrayThrow CastThrow InstanceOfThrow BinOpThrow1 BinOpThrow2 LAssThrow AAccThrow1 AAccThrow2 AAssThrow1 AAssThrow2
AAssThrow3 ALengthThrow FAccThrow FAssThrow1 FAssThrow2 CASThrow CASThrow2 CASThrow3
CallThrowObj CallThrowParams BlockThrow SynchronizedThrow1
SynchronizedThrow2 SeqThrow CondThrow ThrowThrow])

case (*RedCall s a U M Ts T pns body D vs*)
with *red.RedCall-code*[*OF refl refl refl refl refl refl refl refl refl refl refl*, *of a M map Val vs s pns D Ts body U T*]
show *?thesis* **by**(*simp add*: *o-def*)
next
case (*RedCallExternal s a U M Ts T D vs ta va h' ta' e' s'*)
with *red.RedCallExternal-code*[*OF refl refl refl refl refl refl refl refl refl refl refl*, *of a M map Val vs s ta va h' U Ts T D*]
show *?thesis* **by**(*simp add*: *o-def*)
next
case (*RedCallNull M vs s*)
with *red.RedCallNull-code*[*OF refl refl refl refl refl refl refl refl refl refl refl*, *of M map Val vs s*]
show *?thesis* **by**(*simp add*: *o-def*)
next
case (*CallThrowParams es vs a es' v M s*)
with *red.CallThrowParams-code*[*OF refl refl refl refl refl refl refl refl refl refl refl*, *of v M map Val vs @ Throw a # es' s*]
show *?thesis*
apply(*auto simp add*: *is-Throws-conv*)
apply(*erule meta-impE*)
apply(*subst dropWhile-append2*)

apply *auto*
done
next
case *RedThrowNull* **thus** *?thesis*
by−(*erule* (*4*) *red.RedThrowNull'*[*OF refl refl refl refl refl refl refl refl refl refl refl*])
next
case *ThrowThrow* **thus** *?thesis*
by−(*erule* (*4*) *red.ThrowThrow'*[*OF refl refl refl refl refl refl refl refl refl refl refl*])
qed(*assumption*|*erule* (*4*) *red.that*[*unfolded Predicate-Compile.contains-def*, *OF refl refl refl refl refl refl refl refl refl refl refl*])+
next
case *reds*
from *reds.prems* **show** *thesis*
by(*rule J-heap-base.reds.cases*)(*assumption*|*erule* (*4*) *reds.that*[*OF refl refl refl refl refl refl refl refl refl refl refl*])+
qed

end

4.5 Weak well-formedness of Jinja programs

theory *WWellForm*
imports
../Common/WellForm
Expr
begin

definition
wwf-J-mdecl :: *'addr J-prog* ⇒ *cname* ⇒ *'addr J-mb mdecl* ⇒ *bool*
where
wwf-J-mdecl P C ≡ λ(*M*,*Ts*,*T*,(*pns*,*body*)).
length Ts = *length pns* ∧ *distinct pns* ∧ *this* ∉ *set pns* ∧ *fv body* ⊆ {*this*} ∪ *set pns*

lemma *wwf-J-mdecl*[*simp*]:
wwf-J-mdecl P C (*M*,*Ts*,*T*,*pns*,*body*) =
(*length Ts* = *length pns* ∧ *distinct pns* ∧ *this* ∉ *set pns* ∧ *fv body* ⊆ {*this*} ∪ *set pns*)
abbreviation *wwf-J-prog* :: *'addr J-prog* ⇒ *bool*
where *wwf-J-prog* == *wf-prog wwf-J-mdecl*

end

4.6 Well-typedness of Jinja expressions

theory *WellType*
imports
Expr
State
../Common/ExternalCallWF
../Common/WellForm
../Common/SemiType
begin

declare *Listn.lesub-list-impl-same-size*[*simp del*]

declare *listE-length* [*simp del*]

type-synonym
env = *vname* ⇀ *ty*

inductive
WT :: (*ty* ⇒ *ty* ⇒ *ty* ⇒ *bool*) ⇒ 'addr *J-prog* ⇒ *env* ⇒ 'addr *expr* ⇒ *ty* ⇒ *bool* (‹-,-,- ⊢ - :: -› [51,51,51,51]50)
and *WTs* :: (*ty* ⇒ *ty* ⇒ *ty* ⇒ *bool*) ⇒ 'addr *J-prog* ⇒ *env* ⇒ 'addr *expr list* ⇒ *ty list* ⇒ *bool*
(‹-,-,- ⊢ - [::] -› [51,51,51,51]50)
for *is-lub* :: *ty* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹⊢ *lub*'((-,/ -)') = -› [51,51,51] 50)
and *P* :: 'addr *J-prog*
where

WTNew:
is-class P C ⟹
is-lub,*P*,*E* ⊢ *new C* :: *Class C*

| *WTNewArray*:
⟦ *is-lub*,*P*,*E* ⊢ *e* :: *Integer*; *is-type P* (*T*⌊⌉) ⟧ ⟹
is-lub,*P*,*E* ⊢ *newA T*⌊*e*⌉ :: *T*⌊⌉

| *WTCast*:
⟦ *is-lub*,*P*,*E* ⊢ *e* :: *T*; *P* ⊢ *U* ≤ *T* ∨ *P* ⊢ *T* ≤ *U*; *is-type P U* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *Cast U e* :: *U*

| *WTInstanceOf*:
⟦ *is-lub*,*P*,*E* ⊢ *e* :: *T*; *P* ⊢ *U* ≤ *T* ∨ *P* ⊢ *T* ≤ *U*; *is-type P U*; *is-refT U* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *e instanceof U* :: *Boolean*

| *WTVal*:
typeof v = *Some T* ⟹
is-lub,*P*,*E* ⊢ *Val v* :: *T*

| *WTVar*:
E V = *Some T* ⟹
is-lub,*P*,*E* ⊢ *Var V* :: *T*

| *WTBinOp*:
⟦ *is-lub*,*P*,*E* ⊢ *e1* :: *T1*; *is-lub*,*P*,*E* ⊢ *e2* :: *T2*; *P* ⊢ *T1*«*bop*»*T2* :: *T* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *e1*«*bop*»*e2* :: *T*

| *WTLAss*:
⟦ *E V* = *Some T*; *is-lub*,*P*,*E* ⊢ *e* :: *T'*; *P* ⊢ *T'* ≤ *T*; *V* ≠ *this* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *V*:=*e* :: *Void*

| *WTAAcc*:
⟦ *is-lub*,*P*,*E* ⊢ *a* :: *T*⌊⌉; *is-lub*,*P*,*E* ⊢ *i* :: *Integer* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *a*⌊*i*⌉ :: *T*

| *WTAAss*:
⟦ *is-lub*,*P*,*E* ⊢ *a* :: *T*⌊⌉; *is-lub*,*P*,*E* ⊢ *i* :: *Integer*; *is-lub*,*P*,*E* ⊢ *e* :: *T'*; *P* ⊢ *T'* ≤ *T* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *a*⌊*i*⌉ := *e* :: *Void*

| *WTALength*:
is-lub,*P*,*E* ⊢ *a* :: *T*⌊⌉ ⟹ *is-lub*,*P*,*E* ⊢ *a*·*length* :: *Integer*

| *WTFAcc*:
⟦ *is-lub*,*P*,*E* ⊢ *e* :: *U*; *class-type-of*′ *U* = ⌊*C*⌋; *P* ⊢ *C sees F*:*T* (*fm*) *in D* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *e*·*F*{*D*} :: *T*

| *WTFAss*:
⟦ *is-lub*,*P*,*E* ⊢ e_1 :: *U*; *class-type-of*′ *U* = ⌊*C*⌋; *P* ⊢ *C sees F*:*T* (*fm*) *in D*; *is-lub*,*P*,*E* ⊢ e_2 :: *T*′; *P* ⊢ *T*′ ≤ *T* ⟧
⟹ *is-lub*,*P*,*E* ⊢ e_1·*F*{*D*}:=e_2 :: *Void*

| *WTCAS*:
⟦ *is-lub*,*P*,*E* ⊢ *e1* :: *U*; *class-type-of*′ *U* = ⌊*C*⌋; *P* ⊢ *C sees F*:*T* (*fm*) *in D*; *volatile fm*;
is-lub,*P*,*E* ⊢ *e2* :: *T*′; *P* ⊢ *T*′ ≤ *T*; *is-lub*,*P*,*E* ⊢ *e3* :: *T*″; *P* ⊢ *T*″ ≤ *T* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*) :: *Boolean*

| *WTCall*:
⟦ *is-lub*,*P*,*E* ⊢ *e* :: *U*; *class-type-of*′ *U* = ⌊*C*⌋; *P* ⊢ *C sees M*:*Ts* → *T* = *meth in D*;
is-lub,*P*,*E* ⊢ *es* [::] *Ts*′; *P* ⊢ *Ts*′ [≤] *Ts* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *e*·*M*(*es*) :: *T*

| *WTBlock*:
⟦ *is-type P T*; *is-lub*,*P*,*E*(*V* ↦ *T*) ⊢ *e* :: *T*′; *case vo of None* ⇒ *True* | ⌊*v*⌋ ⇒ ∃ *T*′. *typeof v* = ⌊*T*′⌋ ∧ *P* ⊢ *T*′ ≤ *T* ⟧
⟹ *is-lub*,*P*,*E* ⊢ {*V*:*T*=*vo*; *e*} :: *T*′

| *WTSynchronized*:
⟦ *is-lub*,*P*,*E* ⊢ *o*′ :: *T*; *is-refT T*; *T* ≠ *NT*; *is-lub*,*P*,*E* ⊢ *e* :: *T*′ ⟧
⟹ *is-lub*,*P*,*E* ⊢ *sync*(*o*′) *e* :: *T*′

— Note that insync is not statically typable.

| *WTSeq*:
⟦ *is-lub*,*P*,*E* ⊢ e_1::T_1; *is-lub*,*P*,*E* ⊢ e_2::T_2 ⟧
⟹ *is-lub*,*P*,*E* ⊢ e_1;;e_2 :: T_2

| *WTCond*:
⟦ *is-lub*,*P*,*E* ⊢ *e* :: *Boolean*; *is-lub*,*P*,*E* ⊢ e_1::T_1; *is-lub*,*P*,*E* ⊢ e_2::T_2; ⊢ *lub*(T_1, T_2) = *T* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *if* (*e*) e_1 *else* e_2 :: *T*

| *WTWhile*:
⟦ *is-lub*,*P*,*E* ⊢ *e* :: *Boolean*; *is-lub*,*P*,*E* ⊢ *c*::*T* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *while* (*e*) *c* :: *Void*

| *WTThrow*:
⟦ *is-lub*,*P*,*E* ⊢ *e* :: *Class C*; *P* ⊢ *C* ≼* *Throwable* ⟧ ⟹
is-lub,*P*,*E* ⊢ *throw e* :: *Void*

| *WTTry*:
⟦ *is-lub*,*P*,*E* ⊢ e_1 :: *T*; *is-lub*,*P*,*E*(*V* ↦ *Class C*) ⊢ e_2 :: *T*; *P* ⊢ *C* ≼* *Throwable* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *try* e_1 *catch*(*C V*) e_2 :: *T*

| *WTNil*: *is-lub*,*P*,*E* ⊢ [] [::] []

| *WTCons*: ⟦ *is-lub*,*P*,*E* ⊢ *e* :: *T*; *is-lub*,*P*,*E* ⊢ *es* [::] *Ts* ⟧ ⟹ *is-lub*,*P*,*E* ⊢ *e*#*es* [::] *T*#*Ts*

abbreviation *WT'* :: *'addr J-prog* ⇒ *env* ⇒ *'addr expr* ⇒ *ty* ⇒ *bool* (‹-,- ⊢ - :: -› [*51*,*51*,*51*] *50*)
where *WT' P* ≡ *WT* (*TypeRel.is-lub P*) *P*

abbreviation *WTs'* :: *'addr J-prog* ⇒ *env* ⇒ *'addr expr list* ⇒ *ty list* ⇒ *bool* (‹-,- ⊢ - [::] -› [*51*,*51*,*51*] *50*)
where *WTs' P* ≡ *WTs* (*TypeRel.is-lub P*) *P*

declare *WT-WTs.intros*[*intro*!]

inductive-simps *WTs-iffs* [*iff*]:
is-lub',*P*,*E* ⊢ [] [::] *Ts*
is-lub',*P*,*E* ⊢ *e*#*es* [::] *T*#*Ts*
is-lub',*P*,*E* ⊢ *e*#*es* [::] *Ts*

lemma *WTs-conv-list-all2*:
fixes *is-lub*
shows *is-lub*,*P*,*E* ⊢ *es* [::] *Ts* = *list-all2* (*WT is-lub P E*) *es Ts*
by(*induct es arbitrary*: *Ts*)(*auto simp add*: *list-all2-Cons1 elim*: *WTs.cases*)

lemma *WTs-append* [*iff*]: ⋀*is-lub Ts*. (*is-lub*,*P*,*E* ⊢ *es*$_1$ @ *es*$_2$ [::] *Ts*) =
(∃ *Ts*$_1$ *Ts*$_2$. *Ts* = *Ts*$_1$ @ *Ts*$_2$ ∧ *is-lub*,*P*,*E* ⊢ *es*$_1$ [::] *Ts*$_1$ ∧ *is-lub*,*P*,*E* ⊢ *es*$_2$[::]*Ts*$_2$)
by(*auto simp add*: *WTs-conv-list-all2 list-all2-append1 dest*: *list-all2-lengthD*[*symmetric*])

inductive-simps *WT-iffs* [*iff*]:
is-lub',*P*,*E* ⊢ *Val v* :: *T*
is-lub',*P*,*E* ⊢ *Var V* :: *T*
is-lub',*P*,*E* ⊢ *e*$_1$;;*e*$_2$ :: *T*$_2$
is-lub',*P*,*E* ⊢ {*V*:*T*=*vo*; *e*} :: *T'*

inductive-cases *WT-elim-cases*[*elim*!]:
is-lub',*P*,*E* ⊢ *V* :=*e* :: *T*
is-lub',*P*,*E* ⊢ *sync*(*o'*) *e* :: *T*
is-lub',*P*,*E* ⊢ *if* (*e*) *e*$_1$ *else e*$_2$ :: *T*
is-lub',*P*,*E* ⊢ *while* (*e*) *c* :: *T*
is-lub',*P*,*E* ⊢ *throw e* :: *T*
is-lub',*P*,*E* ⊢ *try e*$_1$ *catch*(*C V*) *e*$_2$ :: *T*
is-lub',*P*,*E* ⊢ *Cast D e* :: *T*
is-lub',*P*,*E* ⊢ *e instanceof U* :: *T*
is-lub',*P*,*E* ⊢ *a*·*F*{*D*} :: *T*
is-lub',*P*,*E* ⊢ *a*·*F*{*D*} := *v* :: *T*
is-lub',*P*,*E* ⊢ *e*·*compareAndSwap*(*D*·*F*, *e'*, *e''*) :: *T*
is-lub',*P*,*E* ⊢ *e*$_1$ «*bop*» *e*$_2$ :: *T*
is-lub',*P*,*E* ⊢ *new C* :: *T*
is-lub',*P*,*E* ⊢ *newA T*⌊*e*⌉ :: *T'*
is-lub',*P*,*E* ⊢ *a*⌊*i*⌉ := *e* :: *T*
is-lub',*P*,*E* ⊢ *a*⌊*i*⌉ :: *T*
is-lub',*P*,*E* ⊢ *a*·*length* :: *T*
is-lub',*P*,*E* ⊢ *e*·*M*(*ps*) :: *T*
is-lub',*P*,*E* ⊢ *sync*(*o'*) *e* :: *T*
is-lub',*P*,*E* ⊢ *insync*(*a*) *e* :: *T*

lemma fixes *is-lub* :: *ty* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹⊢ *lub*′((-,/ -)′) = -› [*51*,*51*,*51*] *50*)
assumes *is-lub-unique*: ⋀*T1 T2 T3 T4*. ⟦ ⊢ *lub*(*T1*, *T2*) = *T3*; ⊢ *lub*(*T1*, *T2*) = *T4* ⟧ ⟹ *T3* = *T4*
shows *WT-unique*: ⟦ *is-lub*,*P*,*E* ⊢ *e* :: *T*; *is-lub*,*P*,*E* ⊢ *e* :: *T*′ ⟧ ⟹ *T* = *T*′
and *WTs-unique*: ⟦ *is-lub*,*P*,*E* ⊢ *es* [::] *Ts*; *is-lub*,*P*,*E* ⊢ *es* [::] *Ts*′ ⟧ ⟹ *Ts* = *Ts*′
apply(*induct arbitrary*: *T*′ **and** *Ts*′ *rule*: *WT-WTs.inducts*)
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *fastforce*
apply *fastforce*
apply(*fastforce dest*: *WT-binop-fun*)
apply *fastforce*
apply *fastforce*
apply *fastforce*
apply *fastforce*
apply(*fastforce dest*: *sees-field-fun*)
apply(*fastforce dest*: *sees-field-fun*)
apply *blast*
apply(*fastforce dest*: *sees-method-fun*)
apply *fastforce*
apply *fastforce*
apply *fastforce*
apply(*blast dest*: *is-lub-unique*)
apply *fastforce*
apply *fastforce*
apply *blast*
apply *fastforce*
apply *fastforce*
done

lemma fixes *is-lub*
shows *wt-env-mono*: *is-lub*,*P*,*E* ⊢ *e* :: *T* ⟹ (⋀*E*′. *E* ⊆$_m$ *E*′ ⟹ *is-lub*,*P*,*E*′ ⊢ *e* :: *T*)
and *wts-env-mono*: *is-lub*,*P*,*E* ⊢ *es* [::] *Ts* ⟹ (⋀*E*′. *E* ⊆$_m$ *E*′ ⟹ *is-lub*,*P*,*E*′ ⊢ *es* [::] *Ts*)
apply(*induct rule*: *WT-WTs.inducts*)
apply(*simp add*: *WTNew*)
apply(*simp add*: *WTNewArray*)
apply(*fastforce simp*: *WTCast*)
apply(*fastforce simp*: *WTInstanceOf*)
apply(*fastforce simp*: *WTVal*)
apply(*simp add*: *WTVar map-le-def dom-def*)
apply(*fastforce simp*: *WTBinOp*)
apply(*force simp*:*map-le-def*)
apply(*simp add*: *WTAAcc*)
apply(*simp add*: *WTAAss*, *fastforce*)
apply(*simp add*: *WTALength*, *fastforce*)
apply(*fastforce simp*: *WTFAcc*)
apply(*fastforce simp*: *WTFAss del*:*WT-WTs.intros WT-elim-cases*)
apply *blast*
apply(*fastforce*)
apply(*fastforce simp*: *map-le-def WTBlock*)
apply(*fastforce simp*: *WTSynchronized*)
apply(*fastforce simp*: *WTSeq*)

apply(*fastforce simp*: *WTCond*)
apply(*fastforce simp*: *WTWhile*)
apply(*fastforce simp*: *WTThrow*)
apply(*fastforce simp*: *WTTry map-le-def dom-def*)
apply(*fastforce*)+
done

lemma fixes *is-lub*
  shows *WT-fv*: *is-lub*,*P*,*E* ⊢ *e* :: *T* ⟹ *fv e* ⊆ *dom E*
  and *WT-fvs*: *is-lub*,*P*,*E* ⊢ *es* [::] *Ts* ⟹ *fvs es* ⊆ *dom E*
apply(*induct rule*:*WT-WTs.inducts*)
apply(*simp-all del*: *fun-upd-apply*)
apply *fast*+
done

lemma fixes *is-lub*
  shows *WT-expr-locks*: *is-lub*,*P*,*E* ⊢ *e* :: *T* ⟹ *expr-locks e* = (λ*ad*. *0*)
  and *WTs-expr-lockss*: *is-lub*,*P*,*E* ⊢ *es* [::] *Ts* ⟹ *expr-lockss es* = (λ*ad*. *0*)
by(*induct rule*: *WT-WTs.inducts*)(*auto*)

lemma
  fixes *is-lub* :: *ty* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹⊢ *lub*′((-,/ -)′) = -› [*51*,*51*,*51*] *50*)
  assumes *is-lub-is-type*: ⋀*T1 T2 T3*. ⟦ ⊢ *lub*(*T1*, *T2*) = *T3*; *is-type P T1*; *is-type P T2* ⟧ ⟹ *is-type P T3*
  and *wf*: *wf-prog wf-md P*
  shows *WT-is-type*: ⟦ *is-lub*,*P*,*E* ⊢ *e* :: *T*; *ran E* ⊆ *types P* ⟧ ⟹ *is-type P T*
  and *WTs-is-type*: ⟦ *is-lub*,*P*,*E* ⊢ *es* [::] *Ts*; *ran E* ⊆ *types P* ⟧ ⟹ *set Ts* ⊆ *types P*
apply(*induct rule*: *WT-WTs.inducts*)
apply *simp*
apply *simp*
apply *simp*
apply *simp*
apply (*simp add*:*typeof-lit-is-type*)
apply (*fastforce intro*:*nth-mem simp add*: *ran-def*)
apply(*simp add*: *WT-binop-is-type*)
apply(*simp*)
apply(*simp del*: *is-type-array add*: *is-type-ArrayD*)
apply(*simp*)
apply(*simp*)
apply(*simp add*:*sees-field-is-type*[*OF* - *wf*])
apply(*simp*)
apply *simp*
apply(*fastforce dest*: *sees-wf-mdecl*[*OF wf*] *simp*:*wf-mdecl-def*)
apply(*fastforce simp add*: *ran-def split*: *if-split-asm*)
apply(*simp add*: *is-class-Object*[*OF wf*])
apply(*simp*)
apply(*simp*)
apply(*fastforce intro*: *is-lub-is-type*)
apply(*simp*)
apply(*simp*)
apply *simp*
apply *simp*
apply *simp*
done

lemma
fixes *is-lub1* :: *ty* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹⊢*1* *lub*′((-,/ -)′) = -› [*51*,*51*,*51*] *50*)
and *is-lub2* :: *ty* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹⊢*2* *lub*′((-,/ -)′) = -› [*51*,*51*,*51*] *50*)
assumes *wf*: *wf-prog wf-md P*
and *is-lub1-into-is-lub2*: ⋀*T1 T2 T3*. ⟦ ⊢*1* *lub*(*T1*, *T2*) = *T3*; *is-type P T1*; *is-type P T2* ⟧ ⟹ ⊢*2* *lub*(*T1*, *T2*) = *T3*
and *is-lub2-is-type*: ⋀*T1 T2 T3*. ⟦ ⊢*2* *lub*(*T1*, *T2*) = *T3*; *is-type P T1*; *is-type P T2* ⟧ ⟹ *is-type P T3*
shows *WT-change-is-lub*: ⟦ *is-lub1*,*P*,*E* ⊢ *e* :: *T*; *ran E* ⊆ *types P* ⟧ ⟹ *is-lub2*,*P*,*E* ⊢ *e* :: *T*
and *WTs-change-is-lub*: ⟦ *is-lub1*,*P*,*E* ⊢ *es* [::] *Ts*; *ran E* ⊆ *types P* ⟧ ⟹ *is-lub2*,*P*,*E* ⊢ *es* [::] *Ts*
proof(*induct rule*: *WT-WTs.inducts*)
case (*WTBlock U E V e*′ *T vo*)
from ‹*is-type P U*› ‹*ran E* ⊆ *types P*›
have *ran* (*E*(*V* ↦ *U*)) ⊆ *types P* **by**(*auto simp add*: *ran-def*)
hence *is-lub2*,*P*,*E*(*V* ↦ *U*) ⊢ *e*′ :: *T* **by**(*rule WTBlock*)
with ‹*is-type P U*› **show** *?case*
using ‹*case vo of None* ⇒ *True* | ⌊*v*⌋ ⇒ ∃ *T*′. *typeof v* = ⌊*T*′⌋ ∧ *P* ⊢ *T*′ ≤ *U*› **by** *auto*
next
case (*WTCond E e e1 T1 e2 T2 T*)
from ‹*ran E* ⊆ *types P*› **have** *is-lub2*,*P*,*E* ⊢ *e* :: *Boolean* *is-lub2*,*P*,*E* ⊢ *e1* :: *T1* *is-lub2*,*P*,*E* ⊢ *e2* :: *T2*
by(*rule WTCond*)+
moreover from *is-lub2-is-type wf* ‹*is-lub2*,*P*,*E* ⊢ *e1* :: *T1*› ‹*ran E* ⊆ *types P*›
have *is-type P T1* **by**(*rule WT-is-type*)
from *is-lub2-is-type wf* ‹*is-lub2*,*P*,*E* ⊢ *e2* :: *T2*› ‹*ran E* ⊆ *types P*›
have *is-type P T2* **by**(*rule WT-is-type*)
with ‹⊢*1* *lub*(*T1*, *T2*) = *T*› ‹*is-type P T1*›
have ⊢*2* *lub*(*T1*, *T2*) = *T* **by**(*rule is-lub1-into-is-lub2*)
ultimately show *?case* **..**
next
case (*WTTry E e1 T V C e2*)
from ‹*ran E* ⊆ *types P*› **have** *is-lub2*,*P*,*E* ⊢ *e1* :: *T* **by**(*rule WTTry*)
moreover from ‹*P* ⊢ *C* ≼* *Throwable*› **have** *is-class P C*
by(*rule is-class-sub-Throwable*[*OF wf*])
with ‹*ran E* ⊆ *types P*› **have** *ran* (*E*(*V* ↦ *Class C*)) ⊆ *types P*
by(*auto simp add*: *ran-def*)
hence *is-lub2*,*P*,*E*(*V* ↦ *Class C*) ⊢ *e2* :: *T* **by**(*rule WTTry*)
ultimately show *?case* **using** ‹*P* ⊢ *C* ≼* *Throwable*› **..**
qed *auto*

4.6.1 Code generator setup

lemma *WTBlock-code*:
⋀*is-lub*. ⟦ *is-type P T*; *is-lub*,*P*,*E*(*V* ↦ *T*) ⊢ *e* :: *T*′;
case vo of None ⇒ *True* | ⌊*v*⌋ ⇒ *case typeof v of None* ⇒ *False* | *Some T*′ ⇒ *P* ⊢ *T*′ ≤ *T* ⟧
⟹ *is-lub*,*P*,*E* ⊢ {*V*:*T*=*vo*; *e*} :: *T*′
by(*auto*)

lemmas [*code-pred-intro*] =
WTNew WTNewArray WTCast WTInstanceOf WTVal WTVar WTBinOp WTLAss WTAAcc WTAAss WTALength WTFAcc WTFAss WTCAS WTCall
declare
WTBlock-code [*code-pred-intro WTBlock*′]

```
lemmas [code-pred-intro] =
  WTSynchronized WTSeq WTCond WTWhile WTThrow WTTry
  WTNil WTCons

code-pred
  (modes:
    (i ⇒ i ⇒ o ⇒ bool) ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool,
    (i ⇒ i ⇒ o ⇒ bool) ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool)
  [detect-switches, skip-proof]
  WT
proof −
  case WT
  from WT.prems show thesis
  proof cases
    case (WTBlock T V e vo)
    thus thesis using WT.WTBlock'[OF refl refl refl, of V T vo e] by(auto)
  qed(assumption|erule WT.that[OF refl refl refl]|rule refl)+
next
  case WTs
  from WTs.prems WTs.that show thesis by cases blast+
qed

inductive is-lub-sup :: 'm prog ⇒ ty ⇒ ty ⇒ ty ⇒ bool
for P T1 T2 T3
where
  sup P T1 T2 = OK T3 ⟹ is-lub-sup P T1 T2 T3

code-pred
  (modes: i ⇒ i ⇒ i ⇒ o ⇒ bool, i ⇒ i ⇒ i ⇒ i ⇒ bool)
  is-lub-sup
.

definition WT-code :: 'addr J-prog ⇒ env ⇒ 'addr expr ⇒ ty ⇒ bool (‹-,- ⊢ - ::'' -› [51,51,51] 50)
where WT-code P ≡ WT (is-lub-sup P) P

definition WTs-code :: 'addr J-prog ⇒ env ⇒ 'addr expr list ⇒ ty list ⇒ bool (‹-,- ⊢ - [::'] -›
[51,51,51] 50)
where WTs-code P ≡ WTs (is-lub-sup P) P

lemma assumes wf: wf-prog wf-md P
  shows WT-code-into-WT:
  ⟦ P,E ⊢ e ::' T; ran E ⊆ types P ⟧ ⟹ P,E ⊢ e :: T

  and WTs-code-into-WTs:
  ⟦ P,E ⊢ es [::'] Ts; ran E ⊆ types P ⟧ ⟹ P,E ⊢ es [::] Ts
proof −
  assume ran: ran E ⊆ types P
  { assume wt: P,E ⊢ e ::' T
    show P,E ⊢ e :: T
      by(rule WT-change-is-lub[OF wf - - wt[unfolded WT-code-def] ran])(blast elim!: is-lub-sup.cases
intro: sup-is-lubI[OF wf] is-lub-is-type[OF wf])+ }
  { assume wts: P,E ⊢ es [::'] Ts
    show P,E ⊢ es [::] Ts
     by(rule WTs-change-is-lub[OF wf - - wts[unfolded WTs-code-def] ran])(blast elim!: is-lub-sup.cases
```

intro: *sup-is-lubI*[*OF wf*] *is-lub-is-type*[*OF wf*])+ **}**
qed

lemma assumes *wf*: *wf-prog wf-md P*
shows *WT-into-WT-code*:
⟦ *P*,*E* ⊢ *e* :: *T*; *ran E* ⊆ *types P* ⟧ ⟹ *P*,*E* ⊢ *e* ::′ *T*

and *WT-into-WTs-code-OK*:
⟦ *P*,*E* ⊢ *es* [::] *Ts*; *ran E* ⊆ *types P* ⟧ ⟹ *P*,*E* ⊢ *es* [::′] *Ts*
proof −
assume *ran*: *ran E* ⊆ *types P*
{ assume *wt*: *P*,*E* ⊢ *e* :: *T*
show *P*,*E* ⊢ *e* ::′ *T* **unfolding** *WT-code-def*
by(*rule WT-change-is-lub*[*OF wf - - wt ran*])(*blast intro*!: *is-lub-sup.intros intro*: *is-lub-subD*[*OF wf*] *sup-is-type*[*OF wf*] *elim*!: *is-lub-sup.cases*)+ **}**
{ assume *wts*: *P*,*E* ⊢ *es* [::] *Ts*
show *P*,*E* ⊢ *es* [::′] *Ts* **unfolding** *WTs-code-def*
by(*rule WTs-change-is-lub*[*OF wf - - wts ran*])(*blast intro*!: *is-lub-sup.intros intro*: *is-lub-subD*[*OF wf*] *sup-is-type*[*OF wf*] *elim*!: *is-lub-sup.cases*)+ **}**
qed

theorem *WT-eq-WT-code*:
assumes *wf-prog wf-md P*
and *ran E* ⊆ *types P*
shows *P*,*E* ⊢ *e* :: *T* ⟷ *P*,*E* ⊢ *e* ::′ *T*
using *assms* **by**(*blast intro*: *WT-code-into-WT WT-into-WT-code*)

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
[*inductify*]
WT-code
.

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
[*inductify*]
WTs-code
.

end

4.7 Definite assignment

theory *DefAss*
imports
Expr
begin

4.7.1 Hypersets

type-synonym *′a hyperset* = *′a set option*

definition *hyperUn* :: *′a hyperset* ⇒ *′a hyperset* ⇒ *′a hyperset* (**infixl** ‹⊔› *65*)
where

$A \sqcup B \equiv$ *case A of None* $\Rightarrow$ *None*
$\quad \mid \lfloor A \rfloor \Rightarrow$ (*case B of None* $\Rightarrow$ *None* $\mid \lfloor B \rfloor \Rightarrow \lfloor A \cup B \rfloor$)

definition *hyperInt* :: *'a hyperset* $\Rightarrow$ *'a hyperset* $\Rightarrow$ *'a hyperset* (**infixl** ‹$\sqcap$› *70*)
where
$A \sqcap B \equiv$ *case A of None* $\Rightarrow B$
$\quad \mid \lfloor A \rfloor \Rightarrow$ (*case B of None* $\Rightarrow \lfloor A \rfloor \mid \lfloor B \rfloor \Rightarrow \lfloor A \cap B \rfloor$)

definition *hyperDiff1* :: *'a hyperset* $\Rightarrow$ *'a* $\Rightarrow$ *'a hyperset* (**infixl** ‹$\ominus$› *65*)
where
$A \ominus a \equiv$ *case A of None* $\Rightarrow$ *None* $\mid \lfloor A \rfloor \Rightarrow \lfloor A - \{a\} \rfloor$

definition *hyper-isin* :: *'a* $\Rightarrow$ *'a hyperset* $\Rightarrow$ *bool* (**infix** ‹$\in\in$› *50*)
where
$a \in\in A \equiv$ *case A of None* $\Rightarrow$ *True* $\mid \lfloor A \rfloor \Rightarrow a \in A$

definition *hyper-subset* :: *'a hyperset* $\Rightarrow$ *'a hyperset* $\Rightarrow$ *bool* (**infix** ‹$\sqsubseteq$› *50*)
where
$A \sqsubseteq B \equiv$ *case B of None* $\Rightarrow$ *True*
$\quad \mid \lfloor B \rfloor \Rightarrow$ (*case A of None* $\Rightarrow$ *False* $\mid \lfloor A \rfloor \Rightarrow A \subseteq B$)

lemmas *hyperset-defs* =
hyperUn-def hyperInt-def hyperDiff1-def hyper-isin-def hyper-subset-def

lemma [*simp*]: $\lfloor \{\} \rfloor \sqcup A = A \ \wedge \ A \sqcup \lfloor \{\} \rfloor = A$
lemma [*simp*]: $\lfloor A \rfloor \sqcup \lfloor B \rfloor = \lfloor A \cup B \rfloor \wedge \lfloor A \rfloor \ominus a = \lfloor A - \{a\} \rfloor$
lemma [*simp*]: *None* $\sqcup A =$ *None* $\wedge A \sqcup$ *None* $=$ *None*
lemma [*simp*]: $a \in\in$ *None* $\wedge$ *None* $\ominus a =$ *None*
lemma *hyperUn-assoc*: $(A \sqcup B) \sqcup C = A \sqcup (B \sqcup C)$
lemma *hyper-insert-comm*: $A \sqcup \lfloor \{a\} \rfloor = \lfloor \{a\} \rfloor \sqcup A \wedge A \sqcup (\lfloor \{a\} \rfloor \sqcup B) = \lfloor \{a\} \rfloor \sqcup (A \sqcup B)$
lemma *sqSub-mem-lem* [*elim*]: $[\![A \sqsubseteq A'; a \in\in A]\!] \Longrightarrow a \in\in A'$
by(*auto simp add*: *hyperset-defs*)

lemma [*iff*]: $A \sqsubseteq$ *None*
by(*auto simp add*: *hyperset-defs*)

lemma [*simp*]: $A \sqsubseteq A$
by(*auto simp add*: *hyperset-defs*)

lemma [*iff*]: $\lfloor A \rfloor \sqsubseteq \lfloor B \rfloor \longleftrightarrow A \subseteq B$
by(*auto simp add*: *hyperset-defs*)

lemma *sqUn-lem2*: $A \sqsubseteq A' \Longrightarrow B \sqcup A \sqsubseteq B \sqcup A'$
by(*simp add*:*hyperset-defs*) *blast*

lemma *sqSub-trans* [*trans*, *intro*]: $[\![A \sqsubseteq B; B \sqsubseteq C]\!] \Longrightarrow A \sqsubseteq C$
by(*auto simp add*: *hyperset-defs*)

lemma *hyperUn-comm*: $A \sqcup B = B \sqcup A$
by(*auto simp add*: *hyperset-defs*)

lemma *hyperUn-leftComm*: $A \sqcup (B \sqcup C) = B \sqcup (A \sqcup C)$
by(*auto simp add*: *hyperset-defs*)

lemmas *hyperUn-ac* = *hyperUn-comm hyperUn-leftComm hyperUn-assoc*

lemma [*simp*]: $\lfloor\{\}\rfloor \sqcup B = B$
by(*auto*)

lemma [*simp*]: $\lfloor\{\}\rfloor \sqsubseteq A$
by(*auto simp add*: *hyperset-defs*)

lemma *sqInt-lem*: $A \sqsubseteq A' \Longrightarrow A \sqcap B \sqsubseteq A' \sqcap B$
by(*auto simp add*: *hyperset-defs*)

4.7.2 Definite assignment

primrec $\mathcal{A}$ :: ('*a*,'*b*,'*addr*) *exp* ⇒ '*a hyperset*
  and $\mathcal{A}s$ :: ('*a*,'*b*,'*addr*) *exp list* ⇒ '*a hyperset*
where
  $\mathcal{A}$ (*new C*) = $\lfloor\{\}\rfloor$
| $\mathcal{A}$ (*newA T*$\lfloor e \rceil$) = $\mathcal{A}$ *e*
| $\mathcal{A}$ (*Cast C e*) = $\mathcal{A}$ *e*
| $\mathcal{A}$ (*e instanceof T*) = $\mathcal{A}$ *e*
| $\mathcal{A}$ (*Val v*) = $\lfloor\{\}\rfloor$
| $\mathcal{A}$ (*e*$_1$ «*bop*» *e*$_2$) = $\mathcal{A}$ *e*$_1$ ⊔ $\mathcal{A}$ *e*$_2$
| $\mathcal{A}$ (*Var V*) = $\lfloor\{\}\rfloor$
| $\mathcal{A}$ (*LAss V e*) = $\lfloor\{V\}\rfloor$ ⊔ $\mathcal{A}$ *e*
| $\mathcal{A}$ (*a*$\lfloor i \rceil$) = $\mathcal{A}$ *a* ⊔ $\mathcal{A}$ *i*
| $\mathcal{A}$ (*a*$\lfloor i \rceil$:= *e*) = $\mathcal{A}$ *a* ⊔ $\mathcal{A}$ *i* ⊔ $\mathcal{A}$ *e*
| $\mathcal{A}$ (*a*·*length*) = $\mathcal{A}$ *a*
| $\mathcal{A}$ (*e*·*F*{*D*}) = $\mathcal{A}$ *e*
| $\mathcal{A}$ (*e*$_1$·*F*{*D*}:=*e*$_2$) = $\mathcal{A}$ *e*$_1$ ⊔ $\mathcal{A}$ *e*$_2$
| $\mathcal{A}$ (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) = $\mathcal{A}$ *e1* ⊔ $\mathcal{A}$ *e2* ⊔ $\mathcal{A}$ *e3*
| $\mathcal{A}$ (*e*·*M*(*es*)) = $\mathcal{A}$ *e* ⊔ $\mathcal{A}s$ *es*
| $\mathcal{A}$ ({*V*:*T*=*vo*; *e*}) = $\mathcal{A}$ *e* ⊖ *V*
| $\mathcal{A}$ (*sync*$_V$ (*o'*) *e*) = $\mathcal{A}$ *o'* ⊔ $\mathcal{A}$ *e*
| $\mathcal{A}$ (*insync*$_V$ (*a*) *e*) = $\mathcal{A}$ *e*
| $\mathcal{A}$ (*e*$_1$;;*e*$_2$) = $\mathcal{A}$ *e*$_1$ ⊔ $\mathcal{A}$ *e*$_2$
| $\mathcal{A}$ (*if* (*e*) *e*$_1$ *else e*$_2$) = $\mathcal{A}$ *e* ⊔ ($\mathcal{A}$ *e*$_1$ ⊓ $\mathcal{A}$ *e*$_2$)
| $\mathcal{A}$ (*while* (*b*) *e*) = $\mathcal{A}$ *b*
| $\mathcal{A}$ (*throw e*) = *None*
| $\mathcal{A}$ (*try e*$_1$ *catch*(*C V*) *e*$_2$) = $\mathcal{A}$ *e*$_1$ ⊓ ($\mathcal{A}$ *e*$_2$ ⊖ *V*)

| $\mathcal{A}s$ ([]) = $\lfloor\{\}\rfloor$
| $\mathcal{A}s$ (*e*#*es*) = $\mathcal{A}$ *e* ⊔ $\mathcal{A}s$ *es*

primrec $\mathcal{D}$ :: ('*a*,'*b*,'*addr*) *exp* ⇒ '*a hyperset* ⇒ *bool*
  and $\mathcal{D}s$ :: ('*a*,'*b*,'*addr*) *exp list* ⇒ '*a hyperset* ⇒ *bool*
where
  $\mathcal{D}$ (*new C*) *A* = *True*
| $\mathcal{D}$ (*newA T*$\lfloor e \rceil$) *A* = $\mathcal{D}$ *e A*
| $\mathcal{D}$ (*Cast C e*) *A* = $\mathcal{D}$ *e A*
| $\mathcal{D}$ (*e instanceof T*) = $\mathcal{D}$ *e*
| $\mathcal{D}$ (*Val v*) *A* = *True*
| $\mathcal{D}$ (*e*$_1$ «*bop*» *e*$_2$) *A* = ($\mathcal{D}$ *e*$_1$ *A* ∧ $\mathcal{D}$ *e*$_2$ (*A* ⊔ $\mathcal{A}$ *e*$_1$))
| $\mathcal{D}$ (*Var V*) *A* = (*V* ∈∈ *A*)
| $\mathcal{D}$ (*LAss V e*) *A* = $\mathcal{D}$ *e A*

| $\mathcal{D}\ (a\lfloor i\rceil)\ A = (\mathcal{D}\ a\ A \wedge \mathcal{D}\ i\ (A \sqcup \mathcal{A}\ a))$
| $\mathcal{D}\ (a\lfloor i\rceil := e)\ A = (\mathcal{D}\ a\ A \wedge \mathcal{D}\ i\ (A \sqcup \mathcal{A}\ a) \wedge \mathcal{D}\ e\ (A \sqcup \mathcal{A}\ a \sqcup \mathcal{A}\ i))$
| $\mathcal{D}\ (a{\cdot}length)\ A = \mathcal{D}\ a\ A$
| $\mathcal{D}\ (e{\cdot}F\{D\})\ A = \mathcal{D}\ e\ A$
| $\mathcal{D}\ (e_1{\cdot}F\{D\}{:=}e_2)\ A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e_1))$
| $\mathcal{D}\ (e1{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3))\ A = (\mathcal{D}\ e1\ A \wedge \mathcal{D}\ e2\ (A \sqcup \mathcal{A}\ e1) \wedge \mathcal{D}\ e3\ (A \sqcup \mathcal{A}\ e1 \sqcup \mathcal{A}\ e2))$
| $\mathcal{D}\ (e{\cdot}M(es))\ A = (\mathcal{D}\ e\ A \wedge \mathcal{D}s\ es\ (A \sqcup \mathcal{A}\ e))$
| $\mathcal{D}\ (\{V{:}T{=}vo;\ e\})\ A = (if\ vo = None\ then\ \mathcal{D}\ e\ (A \ominus V)\ else\ \mathcal{D}\ e\ (A \sqcup \lfloor\{V\}\rfloor))$
| $\mathcal{D}\ (sync_V\ (o')\ e)\ A = (\mathcal{D}\ o'\ A \wedge \mathcal{D}\ e\ (A \sqcup \mathcal{A}\ o'))$
| $\mathcal{D}\ (insync_V\ (a)\ e)\ A = \mathcal{D}\ e\ A$
| $\mathcal{D}\ (e_1;;e_2)\ A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e_1))$
| $\mathcal{D}\ (if\ (e)\ e_1\ else\ e_2)\ A = (\mathcal{D}\ e\ A \wedge \mathcal{D}\ e_1\ (A \sqcup \mathcal{A}\ e) \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e))$
| $\mathcal{D}\ (while\ (e)\ c)\ A = (\mathcal{D}\ e\ A \wedge \mathcal{D}\ c\ (A \sqcup \mathcal{A}\ e))$
| $\mathcal{D}\ (throw\ e)\ A = \mathcal{D}\ e\ A$
| $\mathcal{D}\ (try\ e_1\ catch(C\ V)\ e_2)\ A = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \lfloor\{V\}\rfloor))$

| $\mathcal{D}s\ ([])\ A = True$
| $\mathcal{D}s\ (e\#es)\ A = (\mathcal{D}\ e\ A \wedge \mathcal{D}s\ es\ (A \sqcup \mathcal{A}\ e))$

lemma *As-map-Val*[*simp*]: $\mathcal{A}s\ (map\ Val\ vs) = \lfloor\{\}\rfloor$
lemma *As-append* [*simp*]: $\mathcal{A}s\ (xs\ @\ ys) = (\mathcal{A}s\ xs) \sqcup (\mathcal{A}s\ ys)$
by(*induct xs*, *auto simp add*: *hyperset-defs*)

lemma *Ds-map-Val*[*simp*]: $\mathcal{D}s\ (map\ Val\ vs)\ A$
lemma *D-append*[*iff*]: $\bigwedge A.\ \mathcal{D}s\ (es\ @\ es')\ A = (\mathcal{D}s\ es\ A \wedge \mathcal{D}s\ es'\ (A \sqcup \mathcal{A}s\ es))$

lemma fixes *e* :: ('*a*,'*b*,'*addr*) *exp* **and** *es* :: ('*a*,'*b*,'*addr*) *exp list*
  shows *A-fv*: $\bigwedge A.\ \mathcal{A}\ e = \lfloor A\rfloor \Longrightarrow A \subseteq fv\ e$
  and $\bigwedge A.\ \mathcal{A}s\ es = \lfloor A\rfloor \Longrightarrow A \subseteq fvs\ es$
apply(*induct e* **and** *es rule*: $\mathcal{A}$.*induct* $\mathcal{A}s$.*induct*)
apply (*simp-all add*:*hyperset-defs*)
apply *fast*+
done

lemma *sqUn-lem*: $A \sqsubseteq A' \Longrightarrow A \sqcup B \sqsubseteq A' \sqcup B$
lemma *diff-lem*: $A \sqsubseteq A' \Longrightarrow A \ominus b \sqsubseteq A' \ominus b$

lemma fixes *e* :: ('*a*, '*b*, '*addr*) *exp* **and** *es* :: ('*a*, '*b*, '*addr*) *exp list*
  shows *D-mono*: $\bigwedge A\ A'.\ A \sqsubseteq A' \Longrightarrow \mathcal{D}\ e\ A \Longrightarrow \mathcal{D}\ e\ A'$
  and *Ds-mono*: $\bigwedge A\ A'.\ A \sqsubseteq A' \Longrightarrow \mathcal{D}s\ es\ A \Longrightarrow \mathcal{D}s\ es\ A'$

lemma *D-mono'*: $\mathcal{D}\ e\ A \Longrightarrow A \sqsubseteq A' \Longrightarrow \mathcal{D}\ e\ A'$
and *Ds-mono'*: $\mathcal{D}s\ es\ A \Longrightarrow A \sqsubseteq A' \Longrightarrow \mathcal{D}s\ es\ A'$
declare *hyperUn-comm* [*simp*]
declare *hyperUn-leftComm* [*simp*]

end

4.8 Well-formedness Constraints

theory *JWellForm*

imports
WWellForm
WellType
DefAss
begin

definition *wf-J-mdecl* :: *'addr J-prog* ⇒ *cname* ⇒ *'addr J-mb mdecl* ⇒ *bool*
where
wf-J-mdecl P C ≡ λ(*M*,*Ts*,*T*,(*pns*,*body*)).
length Ts = *length pns* ∧
distinct pns ∧
this ∉ *set pns* ∧
(∃ *T'*. *P*,[*this*↦*Class C*,*pns*[↦]*Ts*] ⊢ *body* :: *T'* ∧ *P* ⊢ *T'* ≤ *T*) ∧
$\mathcal{D}$ *body* ⌊{*this*} ∪ *set pns*⌋

lemma *wf-J-mdecl*[*simp*]:
wf-J-mdecl P C (*M*,*Ts*,*T*,*pns*,*body*) ≡
(*length Ts* = *length pns* ∧
distinct pns ∧
this ∉ *set pns* ∧
(∃ *T'*. *P*,[*this*↦*Class C*,*pns*[↦]*Ts*] ⊢ *body* :: *T'* ∧ *P* ⊢ *T'* ≤ *T*) ∧
$\mathcal{D}$ *body* ⌊{*this*} ∪ *set pns*⌋)

abbreviation *wf-J-prog* :: *'addr J-prog* ⇒ *bool*
where *wf-J-prog* == *wf-prog wf-J-mdecl*

lemma *wf-mdecl-wwf-mdecl*: *wf-J-mdecl P C Md* ⟹ *wwf-J-mdecl P C Md*

4.9 The source language as an instance of the framework

theory *Threaded*
imports
SmallStep
JWellForm
../Common/ConformThreaded
../Common/ExternalCallWF
../Framework/FWLiftingSem
../Framework/FWProgressAux
begin

context *heap-base* **begin** — Move to ?? - also used in BV

lemma *wset-Suspend-ok-start-state*:
fixes *final r convert-RA*
assumes *start-state f P C M vs* ∈ *I*
shows *start-state f P C M vs* ∈ *multithreaded-base.wset-Suspend-ok final r convert-RA I*
using *assms*
by(*rule multithreaded-base.wset-Suspend-okI*)(*simp add*: *start-state-def split-beta*)

end

abbreviation *final-expr* :: *'addr expr* × *'addr locals* ⇒ *bool***where**
final-expr ≡ λ(*e*, *x*). *final e*

lemma *final-locks*: *final e* $\Longrightarrow$ *expr-locks e l* = *0*
by(*auto elim*: *finalE*)

context *J-heap-base* **begin**

abbreviation *mred*
:: *'addr J-prog* $\Rightarrow$ (*'addr*, *'thread-id*, *'addr expr* $\times$ *'addr locals*, *'heap*, *'addr*, (*'addr*, *'thread-id*) *obs-event*) *semantics*
where
mred P t $\equiv$ (λ((*e*, *l*), *h*) *ta* ((*e'*, *l'*), *h'*). *P*,*t* $\vdash$ $\langle$*e*, (*h*, *l*)$\rangle$ $-$*ta*$\rightarrow$ $\langle$*e'*, (*h'*, *l'*)$\rangle$)

lemma *red-new-thread-heap*:
$[\![$ *convert-extTA extNTA*,*P*,*t* $\vdash$ $\langle$*e*, *s*$\rangle$ $-$*ta*$\rightarrow$ $\langle$*e'*, *s'*$\rangle$; *NewThread t'' ex'' h''* $\in$ *set* $\{\!|$*ta*$|\!\}_t$ $]\!]$ $\Longrightarrow$ *h''* = *hp s'*
and *reds-new-thread-heap*:
$[\![$ *convert-extTA extNTA*,*P*,*t* $\vdash$ $\langle$*es*, *s*$\rangle$ [$-$*ta*$\rightarrow$] $\langle$*es'*, *s'*$\rangle$; *NewThread t'' ex'' h''* $\in$ *set* $\{\!|$*ta*$|\!\}_t$ $]\!]$ $\Longrightarrow$ *h''* = *hp s'*
apply(*induct rule*: *red-reds.inducts*)
apply(*fastforce dest*: *red-ext-new-thread-heap simp add*: *ta-upd-simps*)+
done

lemma *red-ta-Wakeup-no-Join-no-Lock-no-Interrupt*:
$[\![$ *convert-extTA extNTA*,*P*,*t* $\vdash$ $\langle$*e*, *s*$\rangle$ $-$*ta*$\rightarrow$ $\langle$*e'*, *s'*$\rangle$; *Notified* $\in$ *set* $\{\!|$*ta*$|\!\}_w$ $\vee$ *WokenUp* $\in$ *set* $\{\!|$*ta*$|\!\}_w$ $]\!]$
$\Longrightarrow$ *collect-locks* $\{\!|$*ta*$|\!\}_l$ = {} $\wedge$ *collect-cond-actions* $\{\!|$*ta*$|\!\}_c$ = {} $\wedge$ *collect-interrupts* $\{\!|$*ta*$|\!\}_i$ = {}
and *reds-ta-Wakeup-no-Join-no-Lock-no-Interrupt*:
$[\![$ *convert-extTA extNTA*,*P*,*t* $\vdash$ $\langle$*es*, *s*$\rangle$ [$-$*ta*$\rightarrow$] $\langle$*es'*, *s'*$\rangle$; *Notified* $\in$ *set* $\{\!|$*ta*$|\!\}_w$ $\vee$ *WokenUp* $\in$ *set* $\{\!|$*ta*$|\!\}_w$ $]\!]$
$\Longrightarrow$ *collect-locks* $\{\!|$*ta*$|\!\}_l$ = {} $\wedge$ *collect-cond-actions* $\{\!|$*ta*$|\!\}_c$ = {} $\wedge$ *collect-interrupts* $\{\!|$*ta*$|\!\}_i$ = {}
apply(*induct rule*: *red-reds.inducts*)
apply(*auto simp add*: *ta-upd-simps dest*: *red-external-Wakeup-no-Join-no-Lock-no-Interrupt del*: *conjI*)
done

lemma *final-no-red*:
final e $\Longrightarrow$ $\neg$ *P*,*t* $\vdash$ $\langle$*e*, (*h*, *l*)$\rangle$ $-$*ta*$\rightarrow$ $\langle$*e'*, (*h'*, *l'*)$\rangle$
by(*auto elim*: *red.cases finalE*)

lemma *red-mthr*: *multithreaded final-expr* (*mred P*)
by(*unfold-locales*)(*auto dest*: *red-new-thread-heap*)

end

sublocale *J-heap-base* < *red-mthr*: *multithreaded*
final-expr
mred P
convert-RA
for *P*
by(*rule red-mthr*)

context *J-heap-base* **begin**

abbreviation
mredT ::
'addr J-prog $\Rightarrow$ (*'addr*,*'thread-id*,*'addr expr* $\times$ *'addr locals*,*'heap*,*'addr*) *state*

⇒ ('thread-id × ('addr, 'thread-id, 'addr expr × 'addr locals,'heap) Jinja-thread-action)
⇒ ('addr,'thread-id,'addr expr × 'addr locals,'heap,'addr) state ⇒ bool
where
mredT P ≡ *red-mthr.redT P*

abbreviation
mredT-syntax1 :: 'addr J-prog ⇒ ('addr,'thread-id,'addr expr × 'addr locals,'heap,'addr) state
⇒ 'thread-id ⇒ ('addr, 'thread-id, 'addr expr × 'addr locals,'heap) Jinja-thread-action
⇒ ('addr,'thread-id,'addr expr × 'addr locals,'heap,'addr) state ⇒ bool
(‹- ⊢ - −-▷-→ -› [50,0,0,0,50] 80)
where
mredT-syntax1 P s t ta s' ≡ *mredT P s* (*t*, *ta*) *s'*

abbreviation
mRedT-syntax1 ::
'addr J-prog
⇒ ('addr,'thread-id,'addr expr × 'addr locals,'heap,'addr) state
⇒ ('thread-id × ('addr, 'thread-id, 'addr expr × 'addr locals,'heap) Jinja-thread-action) list
⇒ ('addr,'thread-id,'addr expr × 'addr locals,'heap,'addr) state ⇒ bool
(‹- ⊢ - −▷-→* -› [50,0,0,50] 80)
where
P ⊢ *s* −▷*ttas*→* *s'* ≡ *red-mthr.RedT P s ttas s'*

end

context *J-heap* **begin**

lemma *redT-hext-incr*:
P ⊢ *s* −*t*▷*ta*→ *s'* ⟹ *shr s* ≤ *shr s'*
by(*erule red-mthr.redT.cases*)(*auto dest*!: *red-hext-incr intro*: *hext-trans*)

lemma *RedT-hext-incr*:
assumes *P* ⊢ *s* −▷*tta*→* *s'*
shows *shr s* ≤ *shr s'*
using *assms* **unfolding** *red-mthr.RedT-def*
by(*induct*)(*auto dest*: *redT-hext-incr intro*: *hext-trans*)

end

4.9.1 Lifting *tconf* to multithreaded states

context *J-heap* **begin**

lemma *red-NewThread-Thread-Object*:
⟦ *convert-extTA extNTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; *NewThread t' x m* ∈ *set* {|*ta*|}*t* ⟧
⟹ ∃ *C*. *typeof-addr* (*hp s'*) (*thread-id2addr t'*) = ⌊*Class-type C*⌋ ∧ *P* ⊢ *C* ≼* *Thread*
and *reds-NewThread-Thread-Object*:
⟦ *convert-extTA extNTA*,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *NewThread t' x m* ∈ *set* {|*ta*|}*t* ⟧
⟹ ∃ *C*. *typeof-addr* (*hp s'*) (*thread-id2addr t'*) = ⌊*Class-type C*⌋ ∧ *P* ⊢ *C* ≼* *Thread*
apply(*induct rule*: *red-reds.inducts*)
apply(*fastforce dest*: *red-external-new-thread-exists-thread-object simp add*: *ta-upd-simps*)+
done

lemma *lifting-wf-tconf*:

lifting-wf final-expr (*mred P*) (λ*t ex h*. *P*,*h* ⊢ *t* √*t*)
by(*unfold-locales*)(*fastforce dest*: *red-hext-incr red-NewThread-Thread-Object elim*!: *tconf-hext-mono intro*: *tconfI*)+

end

sublocale *J-heap* < *red-tconf*: *lifting-wf final-expr mred P convert-RA* λ*t ex h*. *P*,*h* ⊢ *t* √*t*
by(*rule lifting-wf-tconf*)

4.9.2 Towards agreement between the framework semantics' lock state and the locks stored in the expressions

primrec *sync-ok* :: ('*a*,'*b*,'*addr*) *exp* ⇒ *bool*
and *sync-oks* :: ('*a*,'*b*,'*addr*) *exp list* ⇒ *bool*
where
sync-ok (*new C*) = *True*
| *sync-ok* (*newA T*⌊*i*⌉) = *sync-ok i*
| *sync-ok* (*Cast T e*) = *sync-ok e*
| *sync-ok* (*e instanceof T*) = *sync-ok e*
| *sync-ok* (*Val v*) = *True*
| *sync-ok* (*Var v*) = *True*
| *sync-ok* (*e* «*bop*» *e*′) = (*sync-ok e* ∧ *sync-ok e*′ ∧ (*contains-insync e*′ ⟶ *is-val e*))
| *sync-ok* (*V* := *e*) = *sync-ok e*
| *sync-ok* (*a*⌊*i*⌉) = (*sync-ok a* ∧ *sync-ok i* ∧ (*contains-insync i* ⟶ *is-val a*))
| *sync-ok* (*AAss a i e*) = (*sync-ok a* ∧ *sync-ok i* ∧ *sync-ok e* ∧ (*contains-insync i* ⟶ *is-val a*) ∧ (*contains-insync e* ⟶ *is-val a* ∧ *is-val i*))
| *sync-ok* (*a*•*length*) = *sync-ok a*
| *sync-ok* (*e*•*F*{*D*}) = *sync-ok e*
| *sync-ok* (*FAss e F D e*′) = (*sync-ok e* ∧ *sync-ok e*′ ∧ (*contains-insync e*′ ⟶ *is-val e*))
| *sync-ok* (*e*•*compareAndSwap*(*D*•*F*, *e*′, *e*′′)) = (*sync-ok e* ∧ *sync-ok e*′ ∧ *sync-ok e*′′ ∧ (*contains-insync e*′ ⟶ *is-val e*) ∧ (*contains-insync e*′′ ⟶ *is-val e* ∧ *is-val e*′))
| *sync-ok* (*e*•*m*(*pns*)) = (*sync-ok e* ∧ *sync-oks pns* ∧ (*contains-insyncs pns* ⟶ *is-val e*))
| *sync-ok* ({*V* : *T*=*vo*; *e*}) = *sync-ok e*
| *sync-ok* (*sync*$_V$ (*o*′) *e*) = (*sync-ok o*′ ∧ ¬ *contains-insync e*)
| *sync-ok* (*insync*$_V$ (*a*) *e*) = *sync-ok e*
| *sync-ok* (*e*;;*e*′) = (*sync-ok e* ∧ ¬ *contains-insync e*′)
| *sync-ok* (*if* (*b*) *e else e*′) = (*sync-ok b* ∧ ¬ *contains-insync e* ∧ ¬ *contains-insync e*′)
| *sync-ok* (*while* (*b*) *e*) = (¬ *contains-insync b* ∧ ¬ *contains-insync e*)
| *sync-ok* (*throw e*) = *sync-ok e*
| *sync-ok* (*try e catch*(*C v*) *e*′) = (*sync-ok e* ∧ ¬ *contains-insync e*′)
| *sync-oks* [] = *True*
| *sync-oks* (*x* # *xs*) = (*sync-ok x* ∧ *sync-oks xs* ∧ (*contains-insyncs xs* ⟶ *is-val x*))

lemma *sync-oks-append* [*simp*]:
sync-oks (*xs* @ *ys*) ⟷ *sync-oks xs* ∧ *sync-oks ys* ∧ (*contains-insyncs ys* ⟶ (∃ *vs*. *xs* = *map Val vs*))
by(*induct xs*)(*auto simp add*: *Cons-eq-map-conv*)

lemma fixes *e* :: ('*a*,'*b*,'*addr*) *exp* **and** *es* :: ('*a*,'*b*,'*addr*) *exp list*
shows *not-contains-insync-sync-ok*: ¬ *contains-insync e* ⟹ *sync-ok e*
and *not-contains-insyncs-sync-oks*: ¬ *contains-insyncs es* ⟹ *sync-oks es*
by(*induct e* **and** *es rule*: *sync-ok.induct sync-oks.induct*)(*auto*)

lemma *expr-locks-sync-ok*: (⋀*ad*. *expr-locks e ad* = *0*) ⟹ *sync-ok e*

and *expr-lockss-sync-oks*: $(\bigwedge ad.\ expr\text{-}lockss\ es\ ad = 0) \Longrightarrow sync\text{-}oks\ es$
by(*auto intro*!: *not-contains-insync-sync-ok not-contains-insyncs-sync-oks*
simp add: *contains-insync-conv contains-insyncs-conv*)

lemma *sync-ok-extRet2J* [*simp*, *intro*!]: *sync-ok e* $\Longrightarrow$ *sync-ok* (*extRet2J e va*)
by(*cases va*) *auto*

abbreviation
sync-es-ok :: ('*addr*,'*thread-id*,('*a*,'*b*,'*addr*) *exp*×'*c*) *thread-info* ⇒ '*heap* ⇒ *bool*
where
sync-es-ok ≡ *ts-ok* (λ*t* (*e*, *x*) *m*. *sync-ok e*)

lemma *sync-es-ok-blocks* [*simp*]:
⟦ *length pns* = *length Ts*; *length Ts* = *length vs* ⟧ $\Longrightarrow$ *sync-ok* (*blocks pns Ts vs e*) = *sync-ok e*
by(*induct pns Ts vs e rule*: *blocks.induct*) *auto*

context *J-heap-base* **begin**

lemma assumes *wf*: *wf-J-prog P*
shows *red-preserve-sync-ok*: ⟦ *extTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩; *sync-ok e* ⟧ $\Longrightarrow$ *sync-ok e*′
and *reds-preserve-sync-oks*: ⟦ *extTA*,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩; *sync-oks es* ⟧ $\Longrightarrow$ *sync-oks es*′
proof(*induct rule*: *red-reds.inducts*)
case (*RedCall s a U M Ts T pns body D vs*)
from *wf* ‹*P* ⊢ *class-type-of U sees M*: *Ts*→*T* = ⌊(*pns*, *body*)⌋ *in D*›
have *wf-mdecl wf-J-mdecl P D* (*M*,*Ts*,*T*,⌊(*pns*,*body*)⌋)
by(*rule sees-wf-mdecl*)
then obtain *T* **where** *P*,[*this*↦*Class D*,*pns*[↦]*Ts*] ⊢ *body* :: *T*
by(*auto simp add*: *wf-mdecl-def*)
hence *expr-locks body* = (λ*ad*. *0*) **by**(*rule WT-expr-locks*)
with ‹*length vs* = *length pns*› ‹*length Ts* = *length pns*›
have *expr-locks* (*blocks pns Ts vs body*) = (λ*ad*. *0*)
by(*simp add*: *expr-locks-blocks*)
thus *?case* **by**(*auto intro*: *expr-locks-sync-ok*)
qed(*fastforce intro*: *not-contains-insync-sync-ok*)+

lemma assumes *wf*: *wf-J-prog P*
shows *expr-locks-new-thread*:
⟦ *P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩; *NewThread t*′′ (*e*′′, *x*′′) *h* ∈ *set* {|*ta*|}*t* ⟧ $\Longrightarrow$ *expr-locks e*′′ = (λ*ad*. *0*)

and *expr-locks-new-thread*′:
⟦ *P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩; *NewThread t*′′ (*e*′′, *x*′′) *h* ∈ *set* {|*ta*|}*t* ⟧ $\Longrightarrow$ *expr-locks e*′′ = (λ*ad*. *0*)
proof(*induct rule*: *red-reds.inducts*)
case (*RedCallExternal s a U M Ts T D vs ta va h*′ *ta*′ *e*′ *s*′)
then obtain *C fs a* **where** *subThread*: *P* ⊢ *C* ≼* *Thread* **and** *ext*: *extNTA2J P* (*C*, *run*, *a*) = (*e*′′, *x*′′)
by(*fastforce dest*: *red-external-new-thread-sub-thread*)
from *sub-Thread-sees-run*[*OF wf subThread*] **obtain** *D pns body*
where *sees*: *P* ⊢ *C sees run*: []→*Void* = ⌊(*pns*, *body*)⌋ *in D* **by** *auto*
from *sees-wf-mdecl*[*OF wf this*] **obtain** *T* **where** *P*,[*this* ↦ *Class D*] ⊢ *body* :: *T*
by(*auto simp add*: *wf-mdecl-def*)
hence *expr-locks body* = (λ*ad*. *0*) **by**(*rule WT-expr-locks*)
with *sees ext* **show** *?case* **by**(*auto simp add*: *extNTA2J-def*)
qed(*auto simp add*: *ta-upd-simps*)

lemma assumes *wf*: *wf-J-prog P*
shows *red-new-thread-sync-ok*: ⟦ *P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩; *NewThread t*′′ (*e*′′, *x*′′) *h*′′ ∈ *set* {|*ta*|}*t* ⟧ ⟹ *sync-ok e*′′
and *reds-new-thread-sync-ok*: ⟦ *P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩; *NewThread t*′′ (*e*′′, *x*′′) *h*′′ ∈ *set* {|*ta*|}*t* ⟧ ⟹ *sync-ok e*′′
by(*auto dest*!: *expr-locks-new-thread*[*OF wf*] *expr-locks-new-thread*′[*OF wf*] *intro*: *expr-locks-sync-ok expr-lockss-sync-oks*)

lemma *lifting-wf-sync-ok*: *wf-J-prog P* ⟹ *lifting-wf final-expr* (*mred P*) (λ*t* (*e*, *x*) *m*. *sync-ok e*)
by(*unfold-locales*)(*auto intro*: *red-preserve-sync-ok red-new-thread-sync-ok*)

lemma *redT-preserve-sync-ok*:
assumes *red*: *P* ⊢ *s* −*t*▷*ta*→ *s*′
shows ⟦ *wf-J-prog P*; *sync-es-ok* (*thr s*) (*shr s*) ⟧ ⟹ *sync-es-ok* (*thr s*′) (*shr s*′)
by(*rule lifting-wf.redT-preserves*[*OF lifting-wf-sync-ok red*])

lemma *RedT-preserves-sync-ok*:
⟦*wf-J-prog P*; *P* ⊢ *s* −▷*ttas*→∗ *s*′; *sync-es-ok* (*thr s*) (*shr s*)⟧
⟹ *sync-es-ok* (*thr s*′) (*shr s*′)
by(*rule lifting-wf.RedT-preserves*[*OF lifting-wf-sync-ok*])

lemma *sync-es-ok-J-start-state*:
⟦ *wf-J-prog P*; *P* ⊢ *C sees M*:*Ts*→*T*=⌊(*pns*, *body*)⌋ *in D*; *length Ts* = *length vs* ⟧
⟹ *sync-es-ok* (*thr* (*J-start-state P C M vs*)) *m*
apply(*rule ts-okI*)
apply(*clarsimp simp add*: *start-state-def split-beta split*: *if-split-asm*)
apply(*drule* (*1*) *sees-wf-mdecl*)
apply(*clarsimp simp add*: *wf-mdecl-def*)
apply(*drule WT-expr-locks*)
apply(*rule expr-locks-sync-ok*)
apply *simp*
done

end

Framework lock state agrees with locks stored in the expression

definition *lock-ok* :: ('*addr*,'*thread-id*) *locks* ⇒ ('*addr*,'*thread-id*,('*a*, '*b*,'*addr*) *exp* × '*x*) *thread-info* ⇒ *bool* **where**
⋀*ln*. *lock-ok ls ts* ≡ ∀ *t*. (*case* (*ts t*) *of None* ⇒ (∀ *l*. *has-locks* (*ls* $ *l*) *t* = *0*)
| ⌊((*e*, *x*), *ln*)⌋ ⇒ (∀ *l*. *has-locks* (*ls* $ *l*) *t* + *ln* $ *l* = *expr-locks e l*))

lemma *lock-okI*:
⟦ ⋀*t l*. *ts t* = *None* ⟹ *has-locks* (*ls* $ *l*) *t* = *0*; ⋀*t e x ln l*. *ts t* = ⌊((*e*, *x*), *ln*)⌋ ⟹ *has-locks* (*ls* $ *l*) *t* + *ln* $ *l*= *expr-locks e l* ⟧ ⟹ *lock-ok ls ts*
apply(*fastforce simp add*: *lock-ok-def*)
done

lemma *lock-okE*:
⟦ *lock-ok ls ts*;
∀ *t*. *ts t* = *None* ⟶ (∀ *l*. *has-locks* (*ls* $ *l*) *t* = *0*) ⟹ *Q*;
∀ *t e x ln*. *ts t* = ⌊((*e*, *x*), *ln*)⌋ ⟶ (∀ *l*. *has-locks* (*ls* $ *l*) *t* + *ln* $ *l* = *expr-locks e l*) ⟹ *Q* ⟧
⟹ *Q*
by(*fastforce simp add*: *lock-ok-def*)

lemma *lock-okD1*:
⟦ *lock-ok ls ts*; *ts t* = *None* ⟧ ⟹ ∀ *l*. *has-locks* (*ls* $ *l*) *t* = *0*
apply(*simp add*: *lock-ok-def*)
apply(*erule-tac x*=*t* **in** *allE*)
apply(*auto*)
done

lemma *lock-okD2*:
⋀*ln*. ⟦ *lock-ok ls ts*; *ts t* = ⌊((*e*, *x*), *ln*)⌋ ⟧ ⟹ ∀ *l*. *has-locks* (*ls* $ *l*) *t* + *ln* $ *l* = *expr-locks e l*
apply(*fastforce simp add*: *lock-ok-def*)
done

lemma *lock-ok-lock-thread-ok*:
assumes *lock*: *lock-ok ls ts*
shows *lock-thread-ok ls ts*
proof(*rule lock-thread-okI*)
fix *l t*
assume *lsl*: *has-lock* (*ls* $ *l*) *t*
show ∃ *xw*. *ts t* = ⌊*xw*⌋
proof(*cases ts t*)
case *None*
with *lock* **have** *has-locks* (*ls* $ *l*) *t* = *0*
by(*auto dest*: *lock-okD1*)
with *lsl* **show** *?thesis* **by** *simp*
next
case (*Some a*) **thus** *?thesis* **by** *blast*
qed
qed

lemma (**in** *J-heap-base*) *lock-ok-J-start-state*:
⟦ *wf-J-prog P*; *P* ⊢ *C sees M*:*Ts*→*T*=⌊(*pns*, *body*)⌋ *in D*; *length Ts* = *length vs* ⟧
⟹ *lock-ok* (*locks* (*J-start-state P C M vs*)) (*thr* (*J-start-state P C M vs*))
apply(*rule lock-okI*)
apply(*auto simp add*: *start-state-def split*: *if-split-asm*)
apply(*drule* (*1*) *sees-wf-mdecl*)
apply(*clarsimp simp add*: *wf-mdecl-def*)
apply(*drule WT-expr-locks*)
apply(*simp add*: *expr-locks-blocks*)
done

4.9.3 Preservation of lock state agreement

fun *upd-expr-lock-action* :: *int* ⇒ *lock-action* ⇒ *int*
where
upd-expr-lock-action i Lock = *i* + *1*
| *upd-expr-lock-action i Unlock* = *i* − *1*
| *upd-expr-lock-action i UnlockFail* = *i*
| *upd-expr-lock-action i ReleaseAcquire* = *i*

fun *upd-expr-lock-actions* :: *int* ⇒ *lock-action list* ⇒ *int* **where**
upd-expr-lock-actions n [] = *n*
| *upd-expr-lock-actions n* (*L* # *Ls*) = *upd-expr-lock-actions* (*upd-expr-lock-action n L*) *Ls*

lemma *upd-expr-lock-actions-append* [*simp*]:

upd-expr-lock-actions n (Ls @ Ls′) = upd-expr-lock-actions (upd-expr-lock-actions n Ls) Ls′
by(*induct Ls arbitrary*: *n*, *auto*)

definition *upd-expr-locks* :: (*′l* ⇒ *int*) ⇒ *′l lock-actions* ⇒ *′l* ⇒ *int*
where *upd-expr-locks els las* ≡ λ*l*. *upd-expr-lock-actions* (*els l*) (*las* $ *l*)

lemma *upd-expr-locks-iff* [*simp*]:
upd-expr-locks els las l = upd-expr-lock-actions (*els l*) (*las* $ *l*)
by(*simp add*: *upd-expr-locks-def*)

lemma *upd-expr-lock-action-add* [*simp*]:
upd-expr-lock-action (*l* + *l′*) *L* = *upd-expr-lock-action l L* + *l′*
by(*cases L*, *auto*)

lemma *upd-expr-lock-actions-add* [*simp*]:
upd-expr-lock-actions (*l* + *l′*) *Ls* = *upd-expr-lock-actions l Ls* + *l′*
by(*induct Ls arbitrary*: *l*, *auto*)

lemma *upd-expr-locks-add* [*simp*]:
upd-expr-locks (λ*a*. *x a* + *y a*) *las* = (λ*a*. *upd-expr-locks x las a* + *y a*)
by(*auto intro*: *ext*)

lemma *expr-locks-extRet2J* [*simp*, *intro*!]: *expr-locks e* = (λ*ad*. *0*) ⟹ *expr-locks* (*extRet2J e va*) = (λ*ad*. *0*)
by(*cases va*) *auto*

lemma (**in** *J-heap-base*)
assumes *wf*: *wf-J-prog P*
shows *red-update-expr-locks*:
⟦ *convert-extTA extNTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩; *sync-ok e* ⟧
⟹ *upd-expr-locks* (*int o expr-locks e*) {|*ta*|}*l* = *int o expr-locks e′*
and *reds-update-expr-lockss*:
⟦ *convert-extTA extNTA*,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es′*, *s′*⟩; *sync-oks es* ⟧
⟹ *upd-expr-locks* (*int o expr-lockss es*) {|*ta*|}*l* = *int o expr-lockss es′*
proof −
have ⟦ *convert-extTA extNTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩; *sync-ok e* ⟧
⟹ *upd-expr-locks* (λ*ad*. *0*) {|*ta*|}*l* = (λ*ad*. (*int o expr-locks e′*) *ad* − (*int o expr-locks e*) *ad*)
and ⟦ *convert-extTA extNTA*,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es′*, *s′*⟩; *sync-oks es* ⟧
⟹ *upd-expr-locks* (λ*ad*. *0*) {|*ta*|}*l* = (λ*ad*. (*int o expr-lockss es′*) *ad* − (*int o expr-lockss es*) *ad*)
proof(*induct rule*: *red-reds.inducts*)
case (*RedCall s a U M Ts T pns body D vs*)
from *wf* ‹*P* ⊢ *class-type-of U sees M*: *Ts*→*T* = ⌊(*pns*, *body*)⌋ *in D*›
have *wf-mdecl wf-J-mdecl P D* (*M*,*Ts*,*T*,⌊(*pns*,*body*)⌋)
by(*rule sees-wf-mdecl*)
then obtain *T* **where** *P*,[*this*↦*Class D*,*pns*[↦] *Ts*] ⊢ *body* :: *T*
by(*auto simp add*: *wf-mdecl-def*)
hence *expr-locks body* = (λ*ad*. *0*) **by**(*rule WT-expr-locks*)
with ‹*length vs* = *length pns*› ‹*length Ts* = *length pns*›
have *expr-locks* (*blocks pns Ts vs body*) = (λ*ad*. *0*)
by(*simp add*: *expr-locks-blocks*)
thus *?case* **by**(*auto intro*: *expr-locks-sync-ok*)
next
case *RedCallExternal* **thus** *?case*
by(*auto simp add*: *fun-eq-iff contains-insync-conv contains-insyncs-conv finfun-upd-apply ta-upd-simps*

elim!: *red-external.cases*)
qed(*fastforce simp add*: *fun-eq-iff contains-insync-conv contains-insyncs-conv finfun-upd-apply ta-upd-simps*)+
hence ⟦ *convert-extTA extNTA,P,t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; *sync-ok e* ⟧
⟹ *upd-expr-locks* (λ*ad*. *0* + (*int* ∘ *expr-locks e*) *ad*) {|*ta*|}$_l$ = *int* ∘ *expr-locks e'*
and ⟦ *convert-extTA extNTA,P,t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *sync-oks es* ⟧
⟹ *upd-expr-locks* (λ*ad*. *0* + (*int* ∘ *expr-lockss es*) *ad*) {|*ta*|}$_l$ = *int* ∘ *expr-lockss es'*
by(*auto intro*: *ext simp only*: *upd-expr-locks-add*)
thus ⟦ *convert-extTA extNTA,P,t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; *sync-ok e* ⟧
⟹ *upd-expr-locks* (*int o expr-locks e*) {|*ta*|}$_l$ = *int o expr-locks e'*
and ⟦ *convert-extTA extNTA,P,t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *sync-oks es* ⟧
⟹ *upd-expr-locks* (*int o expr-lockss es*) {|*ta*|}$_l$ = *int o expr-lockss es'*
by(*auto simp add*: *o-def*)
qed

definition *lock-expr-locks-ok* :: *'t FWState.lock* ⇒ *'t* ⇒ *nat* ⇒ *int* ⇒ *bool* **where**
lock-expr-locks-ok l t n i ≡ (*i* = *int* (*has-locks l t*) + *int n*) ∧ *i* ≥ *0*

lemma *upd-lock-upd-expr-lock-action-preserve-lock-expr-locks-ok*:
assumes *lao*: *lock-action-ok l t L*
and *lelo*: *lock-expr-locks-ok l t n i*
shows *lock-expr-locks-ok* (*upd-lock l t L*) *t* (*upd-threadR n l t L*) (*upd-expr-lock-action i L*)
proof −
from *lelo* **have** *i*: *i* ≥ *0*
and *hl*: *i* = *int* (*has-locks l t*) + *int n*
by(*auto simp add*: *lock-expr-locks-ok-def*)
from *lelo*
show *?thesis*
proof(*cases L*)
case *Lock*
with *lao* **have** *may-lock l t* **by**(*simp*)
with *hl* **have** *has-locks* (*lock-lock l t*) *t* = (*Suc* (*has-locks l t*)) **by**(*auto*)
with *Lock i hl* **show** *?thesis*
by(*simp add*: *lock-expr-locks-ok-def*)
next
case *Unlock*
with *lao* **have** *has-lock l t* **by** *simp*
then obtain *n'*
where *hl'*: *has-locks l t* = *Suc n'*
by(*auto dest*: *has-lock-has-locks-Suc*)
hence *has-locks* (*unlock-lock l*) *t* = *n'* **by** *simp*
with *Unlock i hl hl'* **show** *?thesis*
by(*simp add*: *lock-expr-locks-ok-def*)
qed(*auto simp add*: *lock-expr-locks-ok-def*)
qed

lemma *upd-locks-upd-expr-lock-preserve-lock-expr-locks-ok*:
⟦ *lock-actions-ok l t Ls*; *lock-expr-locks-ok l t n i* ⟧
⟹ *lock-expr-locks-ok* (*upd-locks l t Ls*) *t* (*upd-threadRs n l t Ls*) (*upd-expr-lock-actions i Ls*)
by(*induct Ls arbitrary*: *l i n*)(*auto intro*: *upd-lock-upd-expr-lock-action-preserve-lock-expr-locks-ok*)

definition *ls-els-ok* :: (*'addr*,*'thread-id*) *locks* ⇒ *'thread-id* ⇒ (*'addr* ⇒*f nat*) ⇒ (*'addr* ⇒ *int*) ⇒ *bool*
where
⋀*ln*. *ls-els-ok ls t ln els* ≡ ∀ *l*. *lock-expr-locks-ok* (*ls* $ *l*) *t* (*ln* $ *l*) (*els l*)

lemma *ls-els-okI*:
⋀*ln*. (⋀*l*. *lock-expr-locks-ok* (*ls* $ *l*) *t* (*ln* $ *l*) (*els l*)) ⟹ *ls-els-ok ls t ln els*
by(*auto simp add*: *ls-els-ok-def*)

lemma *ls-els-okE*:
⋀*ln*. ⟦ *ls-els-ok ls t ln els*; ∀ *l*. *lock-expr-locks-ok* (*ls* $ *l*) *t* (*ln* $ *l*) (*els l*) ⟹ *P* ⟧ ⟹ *P*
by(*auto simp add*: *ls-els-ok-def*)

lemma *ls-els-okD*:
⋀*ln*. *ls-els-ok ls t ln els* ⟹ *lock-expr-locks-ok* (*ls* $ *l*) *t* (*ln* $ *l*) (*els l*)
by(*auto simp add*: *ls-els-ok-def*)

lemma *redT-updLs-upd-expr-locks-preserves-ls-els-ok*:
⋀*ln*. ⟦ *ls-els-ok ls t ln els*; *lock-ok-las ls t las* ⟧
⟹ *ls-els-ok* (*redT-updLs ls t las*) *t* (*redT-updLns ls t ln las*) (*upd-expr-locks els las*)
by(*auto intro*!: *ls-els-okI upd-locks-upd-expr-lock-preserve-lock-expr-locks-ok elim*!: *ls-els-okE simp add*: *redT-updLs-def lock-ok-las-def*)

lemma *sync-ok-redT-updT*:
assumes *sync-es-ok ts h*
and *nt*: ⋀*t e x h''*. *ta* = *NewThread t* (*e*, *x*) *h''* ⟹ *sync-ok e*
shows *sync-es-ok* (*redT-updT ts ta*) *h'*
using *assms*
proof(*cases ta*)
case (*NewThread T x m*)
obtain *E X* **where** [*simp*]: *x* = (*E*, *X*) **by** (*cases x*, *auto*)
with *NewThread* **have** *sync-ok E* **by**(*simp*)(*rule nt*)
with *NewThread* ‹*sync-es-ok ts h*› **show** *?thesis*
apply −
apply(*rule ts-okI*)
apply(*case-tac t*=*T*)
by(*auto dest*: *ts-okD*)
qed(*auto intro*: *ts-okI dest*: *ts-okD*)

lemma *sync-ok-redT-updTs*:
⟦ *sync-es-ok ts h*; ⋀*t e x h*. *NewThread t* (*e*, *x*) *h* ∈ *set tas* ⟹ *sync-ok e* ⟧
⟹ *sync-es-ok* (*redT-updTs ts tas*) *h'*
proof(*induct tas arbitrary*: *ts*)
case *Nil* **thus** *?case* **by**(*auto intro*: *ts-okI dest*: *ts-okD*)
next
case (*Cons TA TAS TS*)
note *IH* = ‹⋀*ts*. ⟦*sync-es-ok ts h*; ⋀*t e x h''*. *NewThread t* (*e*, *x*) *h''* ∈ *set TAS* ⟹ *sync-ok e*⟧
⟹ *sync-es-ok* (*redT-updTs ts TAS*) *h'*›
note *nt* = ‹⋀*t e x h*. *NewThread t* (*e*, *x*) *h* ∈ *set* (*TA* # *TAS*) ⟹ *sync-ok e*›
from ‹*sync-es-ok TS h*› *nt*
have *sync-es-ok* (*redT-updT TS TA*) *h*
by(*auto elim*!: *sync-ok-redT-updT*)
hence *sync-es-ok* (*redT-updTs* (*redT-updT TS TA*) *TAS*) *h'*
by(*rule IH*)(*auto intro*: *nt*)
thus *?case* **by** *simp*
qed

lemma *lock-ok-thr-updI*:
⋀*ln*. ⟦ *lock-ok ls ts*; *ts t* = ⌊((*e*, *xs*), *ln*)⌋; *expr-locks e* = *expr-locks e′* ⟧
⟹ *lock-ok ls* (*ts*(*t* ↦ ((*e′*, *xs′*), *ln*)))
by(*rule lock-okI*)(*auto split*: *if-split-asm dest*: *lock-okD2 lock-okD1*)

context *J-heap-base* **begin**

lemma *redT-preserves-lock-ok*:
assumes *wf*: *wf-J-prog P*
and *P* ⊢ *s* −*t*▷*ta*→ *s′*
and *lock-ok* (*locks s*) (*thr s*)
and *sync-es-ok* (*thr s*) (*shr s*)
shows *lock-ok* (*locks s′*) (*thr s′*)
proof −
obtain *ls ts h ws is* **where** *s* [*simp*]: *s* = (*ls*, (*ts*, *h*), *ws*, *is*) **by**(*cases s*) *fastforce*
obtain *ls′ ts′ h′ ws′ is′* **where** *s′* [*simp*]: *s′* = (*ls′*, (*ts′*, *h′*), *ws′*, *is′*) **by**(*cases s′*) *fastforce*
from *assms* **have** *redT*: *P* ⊢ (*ls*, (*ts*, *h*), *ws*, *is*) −*t*▷*ta*→ (*ls′*, (*ts′*, *h′*), *ws′*, *is′*)
and *loes*: *lock-ok ls ts*
and *aoes*: *sync-es-ok ts h* **by** *auto*
from *redT* **have** *lock-ok ls′ ts′*
proof(*cases rule*: *red-mthr.redT-elims*)
case (*normal a a′ m′*)
moreover obtain *e x* **where** *a* = (*e*, *x*) **by** (*cases a*, *auto*)
moreover obtain *e′ x′* **where** *a′* = (*e′*, *x′*) **by** (*cases a′*, *auto*)
ultimately have *P*: *P*,*t* ⊢ ⟨*e*,(*h*, *x*)⟩ −*ta*→ ⟨*e′*,(*m′*, *x′*)⟩
and *est*: *ts t* = ⌊((*e*, *x*), *no-wait-locks*)⌋
and *lota*: *lock-ok-las ls t* {|*ta*|}*l*
and *cctta*: *thread-oks ts* {|*ta*|}*t*
and *ls′*: *ls′* = *redT-updLs ls t* {|*ta*|}*l*
and *s′*: *ts′* = (*redT-updTs ts* {|*ta*|}*t*)(*t* ↦ ((*e′*, *x′*), *redT-updLns ls t no-wait-locks* {|*ta*|}*l*))
by *auto*
let *?ts′* = (*redT-updTs ts* {|*ta*|}*t*)(*t* ↦ ((*e′*, *x′*), *redT-updLns ls t no-wait-locks* {|*ta*|}*l*))
from *est aoes* **have** *aoe*: *sync-ok e* **by**(*auto dest*: *ts-okD*)
from *aoe P* **have** *aoe′*: *sync-ok e′* **by**(*auto dest*: *red-preserve-sync-ok*[*OF wf*])
from *aoes red-new-thread-sync-ok*[*OF wf P*]
have *sync-es-ok* (*redT-updTs ts* {|*ta*|}*t*) *h′*
by(*rule sync-ok-redT-updTs*)
with *aoe′* **have** *aoes′*: *sync-es-ok ?ts′ m′*
by(*auto intro*!: *ts-okI dest*: *ts-okD split*: *if-split-asm*)
have *lock-ok ls′ ?ts′*
proof(*rule lock-okI*)
fix *t″ l*
assume *?ts′ t″* = *None*
hence *ts t″* = *None*
by(*auto split*: *if-split-asm intro*: *redT-updTs-None*)
with *loes* **have** *has-locks* (*ls* $ *l*) *t″* = *0*
by(*auto dest*: *lock-okD1*)
moreover from ‹*?ts′ t″* = *None*›
have *t* ≠ *t″* **by**(*simp split*: *if-split-asm*)
ultimately show *has-locks* (*ls′* $ *l*) *t″* = *0*
by(*simp add*: *red-mthr.redT-has-locks-inv*[*OF redT*])
next
fix *t″ e″ x″ l ln″*
assume *ts′t″*: *?ts′ t″* = ⌊((*e″*, *x″*), *ln″*)⌋

with *aoes'* **have** *aoe''*: *sync-ok e''* **by**(*auto dest*: *ts-okD*)
show *has-locks* (*ls'* $ *l*) *t''* + *ln''* $ *l* = *expr-locks e''* *l*
proof(*cases t* = *t''*)
case *True*
note *tt''* = ‹*t* = *t''*›
with *ts't''* **have** *e''*: *e''* = *e'* **and** *x''*: *x''* = *x'*
and *ln''*: *ln''* = *redT-updLns ls t no-wait-locks* {|*ta*|}*l* **by** *auto*
have *ls-els-ok ls t no-wait-locks* (*int o expr-locks e*)
proof(*rule ls-els-okI*)
fix *l*
note *lock-okD2*[*OF loes*, *OF est*]
thus *lock-expr-locks-ok* (*ls* $ *l*) *t* (*no-wait-locks* $ *l*) ((*int* ∘ *expr-locks e*) *l*)
by(*simp add*: *lock-expr-locks-ok-def*)
qed
hence *ls-els-ok* (*redT-updLs ls t* {|*ta*|}*l*) *t* (*redT-updLns ls t no-wait-locks* {|*ta*|}*l*) (*upd-expr-locks* (*int o expr-locks e*) {|*ta*|}*l*)
by(*rule redT-updLs-upd-expr-locks-preserves-ls-els-ok*[*OF - lota*])
hence *ls-els-ok* (*redT-updLs ls t* {|*ta*|}*l*) *t* (*redT-updLns ls t no-wait-locks* {|*ta*|}*l*) (*int o expr-locks e'*)
by(*simp only*: *red-update-expr-locks*[*OF wf P aoe*])
thus *?thesis* **using** *ls' e'' tt'' ln''*
by(*auto dest*: *ls-els-okD*[**where** *l* = *l*] *simp*: *lock-expr-locks-ok-def*)
next
case *False*
note *tt''* = ‹*t* ≠ *t''*›
from *lota* **have** *lao*: *lock-actions-ok* (*ls* $ *l*) *t* ({|*ta*|}*l* $ *l*)
by(*simp add*: *lock-ok-las-def*)
show *?thesis*
proof(*cases ts t''*)
case *None*
with *est ts't'' tt'' cctta*
obtain *m* **where** *NewThread t''* (*e''*, *x''*) *m* ∈ *set* {|*ta*|}*t* **and** *ln''*: *ln''* = *no-wait-locks*
by(*auto dest*: *redT-updTs-new-thread*)
moreover with *P* **have** *m'* = *m* **by**(*auto dest*: *red-new-thread-heap*)
ultimately have *NewThread t''* (*e''*, *x''*) *m'* ∈ *set* {|*ta*|}*t* **by** *simp*
with *wf P ln''* **have** *expr-locks e''* = (λ*ad*. *0*)
by −(*rule expr-locks-new-thread*)
hence *elel*: *expr-locks e'' l* = *0* **by** *simp*
from *loes None* **have** *has-locks* (*ls* $ *l*) *t''* = *0*
by(*auto dest*: *lock-okD1*)
moreover note *lock-actions-ok-has-locks-upd-locks-eq-has-locks*[*OF lao tt''*[*symmetric*]]
ultimately have *has-locks* (*redT-updLs ls t* {|*ta*|}*l* $ *l*) *t''* = *0*
by(*auto simp add*: *fun-eq-iff*)
with *elel ls' ln''* **show** *?thesis* **by**(*auto*)
next
case (*Some a*)
then obtain *E X LN* **where** *est''*: *ts t''* = ⌊((*E*, *X*), *LN*)⌋ **by**(*cases a*, *auto*)
with *loes* **have** *IH*: *has-locks* (*ls* $ *l*) *t''* + *LN* $ *l* = *expr-locks E l*
by(*auto dest*: *lock-okD2*)
from *est est'' tt'' cctta* **have** *?ts' t''* = ⌊((*E*, *X*), *LN*)⌋
by(*simp*)(*rule redT-updTs-Some*, *simp-all*)
with *ts't''* **have** *e''*: *E* = *e''* **and** *x''*: *X* = *x''*
and *ln''*: *ln''* = *LN* **by**(*simp-all*)
with *lock-actions-ok-has-locks-upd-locks-eq-has-locks*[*OF lao tt''*[*symmetric*]] *IH ls'*

```
      show ?thesis by(clarsimp simp add: redT-updLs-def fun-eq-iff)
     qed
    qed
  qed
  with s' show ?thesis by simp
next
  case (acquire a ln n)
  hence [simp]: ta = (K$ [], [], [], [], [], convert-RA ln) ws' = ws h' = h
    and ls': ls' = acquire-all ls t ln
    and ts': ts' = ts(t ↦ (a, no-wait-locks))
    and ts t = ⌊(a, ln)⌋
    and may-acquire-all ls t ln
    by auto
  obtain e x where [simp]: a = (e, x) by (cases a, auto)
  from ts' have ts': ts' = ts(t ↦ ((e, x), no-wait-locks)) by simp
  from ‹ts t = ⌊(a, ln)⌋› have tst: ts t = ⌊((e, x), ln)⌋ by simp
  show ?thesis
  proof(rule lock-okI)
    fix t'' l
    assume rtutes: ts' t'' = None
    with ts' have tst'': ts t'' = None
      by(simp split: if-split-asm)
    with tst have tt'': t ≠ t'' by auto
    from tst'' loes have has-locks (ls $ l) t'' = 0
      by(auto dest: lock-okD1)
    thus has-locks (ls' $ l) t'' = 0
      by(simp add: red-mthr.redT-has-locks-inv[OF redT tt''])
  next
    fix t'' e'' x'' ln'' l
    assume ts't'': ts' t'' = ⌊((e'', x''), ln'')⌋
    show has-locks (ls' $ l) t'' + ln'' $ l = expr-locks e'' l
    proof(cases t = t'')
      case True
      note [simp] = this
      with ts't'' ts' tst
      have [simp]: ln'' = no-wait-locks e = e'' by auto
      from tst loes have has-locks (ls $ l) t + ln $ l = expr-locks e l
        by(auto dest: lock-okD2)
      show ?thesis
      proof(cases ln $ l > 0)
        case True
        with ‹may-acquire-all ls t ln› ls' have may-lock (ls $ l) t
          by(auto elim: may-acquire-allE)
        with ls'
        have has-locks (ls' $ l) t = has-locks (ls $ l) t + ln $ l
          by(simp add: has-locks-acquire-locks-conv)
        with ‹has-locks (ls $ l) t + ln $ l = expr-locks e l›
        show ?thesis by(simp)
      next
        case False
        hence ln $ l = 0 by simp
        with ls' have has-locks (ls' $ l) t = has-locks (ls $ l) t
          by(simp)
        with ‹has-locks (ls $ l) t + ln $ l = expr-locks e l› ‹ln $ l = 0›
```

```
        show ?thesis by(simp)
       qed
      next
       case False
       with ts' ts't'' have tst'': ts t'' = ⌊((e'', x''), ln'')⌋ by(simp)
       with loes have has-locks (ls $ l) t'' + ln'' $ l = expr-locks e'' l
        by(auto dest: lock-okD2)
       show ?thesis
       proof(cases ln $ l > 0)
        case False
        with ‹t ≠ t''› ls'
        have has-locks (ls' $ l) t'' = has-locks (ls $ l) t'' by(simp)
        with ‹has-locks (ls $ l) t'' + ln'' $ l = expr-locks e'' l›
        show ?thesis by(simp)
       next
        case True
        with ‹may-acquire-all ls t ln› have may-lock (ls $ l) t
         by(auto elim: may-acquire-allE)
        with ls' ‹t ≠ t''› have has-locks (ls' $ l) t'' = has-locks (ls $ l) t''
         by(simp add: has-locks-acquire-locks-conv')
        with ls' ‹has-locks (ls $ l) t'' + ln'' $ l = expr-locks e'' l›
        show ?thesis by(simp)
       qed
      qed
     qed
    qed
    thus ?thesis by simp
qed

lemma invariant3p-sync-es-ok-lock-ok:
  assumes wf: wf-J-prog P
  shows invariant3p (mredT P) {s. sync-es-ok (thr s) (shr s) ∧ lock-ok (locks s) (thr s)}
apply(rule invariant3pI)
apply clarify
apply(rule conjI)
 apply(rule lifting-wf.redT-preserves[OF lifting-wf-sync-ok[OF wf]], blast)
 apply(assumption)
apply(erule (2) redT-preserves-lock-ok[OF wf])
done

lemma RedT-preserves-lock-ok:
  assumes wf: wf-J-prog P
  and Red: P ⊢ s −▷ttas→* s'
  and ae: sync-es-ok (thr s) (shr s)
  and loes: lock-ok (locks s) (thr s)
  shows lock-ok (locks s') (thr s')
using invariant3p-rtrancl3p[OF invariant3p-sync-es-ok-lock-ok[OF wf] Red[unfolded red-mthr.RedT-def]]
ae loes
by simp

end
```

4.9.4 Determinism

context *J-heap-base* **begin**

lemma
fixes *final*
assumes *det*: *deterministic-heap-ops*
shows *red-deterministic*:
⟦ *convert-extTA extTA*,*P*,*t* ⊢ ⟨*e*, (*shr s*, *xs*)⟩ −*ta*→ ⟨*e'*, *s'*⟩;
convert-extTA extTA,*P*,*t* ⊢ ⟨*e*, (*shr s*, *xs*)⟩ −*ta'*→ ⟨*e''*, *s''*⟩;
final-thread.actions-ok final s t ta; *final-thread.actions-ok final s t ta'* ⟧
⟹ *ta* = *ta'* ∧ *e'* = *e''* ∧ *s'* = *s''*
and *reds-deterministic*:
⟦ *convert-extTA extTA*,*P*,*t* ⊢ ⟨*es*, (*shr s*, *xs*)⟩ [−*ta*→] ⟨*es'*, *s'*⟩;
convert-extTA extTA,*P*,*t* ⊢ ⟨*es*, (*shr s*, *xs*)⟩ [−*ta'*→] ⟨*es''*, *s''*⟩;
final-thread.actions-ok final s t ta; *final-thread.actions-ok final s t ta'* ⟧
⟹ *ta* = *ta'* ∧ *es'* = *es''* ∧ *s'* = *s''*
proof(*induct e* (*shr s*, *xs*) *ta e' s'* **and** *es* (*shr s*, *xs*) *ta es' s' arbitrary*: *e'' s'' xs* **and** *es'' s'' xs rule*: *red-reds.inducts*)
case *RedNew*
thus *?case* **by**(*auto elim*!: *red-cases dest*: *deterministic-heap-ops-allocateD*[*OF det*])
next
case *RedNewArray*
thus *?case* **by**(*auto elim*!: *red-cases dest*: *deterministic-heap-ops-allocateD*[*OF det*])
next
case *RedCall* **thus** *?case*
by(*auto elim*!: *red-cases dest*: *sees-method-fun simp add*: *map-eq-append-conv*)
next
case *RedCallExternal* **thus** *?case*
by(*auto elim*!: *red-cases dest*: *red-external-deterministic*[*OF det*] *simp add*: *final-thread.actions-ok-iff map-eq-append-conv dest*: *sees-method-fun*)
next
case *RedCallNull* **thus** *?case* **by**(*auto elim*!: *red-cases dest*: *sees-method-fun simp add*: *map-eq-append-conv*)
next
case *CallThrowParams* **thus** *?case*
by(*auto elim*!: *red-cases dest*: *sees-method-fun simp add*: *map-eq-append-conv append-eq-map-conv append-eq-append-conv2 reds-map-Val-Throw Cons-eq-append-conv append-eq-Cons-conv*)
qed(*fastforce elim*!: *red-cases reds-cases dest*: *deterministic-heap-ops-readD*[*OF det*] *deterministic-heap-ops-writeD*[*OF det*] *iff*: *reds-map-Val-Throw*)+

lemma *red-mthr-deterministic*:
assumes *det*: *deterministic-heap-ops*
shows *red-mthr.deterministic P UNIV*
proof(*rule red-mthr.determisticI*)
fix *s t x ta' x' m' ta'' x'' m''*
assume *thr s t* = ⌊(*x*, *no-wait-locks*)⌋
and *red*: *mred P t* (*x*, *shr s*) *ta'* (*x'*, *m'*) *mred P t* (*x*, *shr s*) *ta''* (*x''*, *m''*)
and *aok*: *red-mthr.actions-ok s t ta' red-mthr.actions-ok s t ta''*
moreover obtain *e xs* **where** [*simp*]: *x* = (*e*, *xs*) **by**(*cases x*)
moreover obtain *e' xs'* **where** [*simp*]: *x'* = (*e'*, *xs'*) **by**(*cases x'*)
moreover obtain *e'' xs''* **where** [*simp*]: *x''* = (*e''*, *xs''*) **by**(*cases x''*)
ultimately have *extTA2J P*,*P*,*t* ⊢ ⟨*e*,(*shr s*, *xs*)⟩ −*ta'*→ ⟨*e'*,(*m'*, *xs'*)⟩
and *extTA2J P*,*P*,*t* ⊢ ⟨*e*,(*shr s*, *xs*)⟩ −*ta''*→ ⟨*e''*,(*m''*, *xs''*)⟩
by *simp-all*

from *red-deterministic*[*OF det this aok*]
show $ta' = ta'' \wedge x' = x'' \wedge m' = m''$ **by** *simp*
qed *simp*

end

end

4.10 Runtime Well-typedness

theory *WellTypeRT*
imports
WellType
JHeap
begin

context *J-heap-base* **begin**

inductive *WTrt* :: *'addr J-prog* ⇒ *'heap* ⇒ *env* ⇒ *'addr expr* ⇒ *ty* ⇒ *bool*
and *WTrts* :: *'addr J-prog* ⇒ *'heap* ⇒ *env* ⇒ *'addr expr list* ⇒ *ty list* ⇒ *bool*
for *P* :: *'addr J-prog* **and** *h* :: *'heap*
where

WTrtNew:
is-class P C ⟹ *WTrt P h E* (*new C*) (*Class C*)

| *WTrtNewArray*:
⟦ *WTrt P h E e Integer*; *is-type P* (*T*⌊⌋) ⟧
⟹ *WTrt P h E* (*newA T*⌊*e*⌋) (*T*⌊⌋)

| *WTrtCast*:
⟦ *WTrt P h E e T*; *is-type P U* ⟧ ⟹ *WTrt P h E* (*Cast U e*) *U*

| *WTrtInstanceOf*:
⟦ *WTrt P h E e T*; *is-type P U* ⟧ ⟹ *WTrt P h E* (*e instanceof U*) *Boolean*

| *WTrtVal*:
typeof$_h$ *v* = *Some T* ⟹ *WTrt P h E* (*Val v*) *T*

| *WTrtVar*:
E V = *Some T* ⟹ *WTrt P h E* (*Var V*) *T*

| *WTrtBinOp*:
⟦ *WTrt P h E e1 T1*; *WTrt P h E e2 T2*; *P* ⊢ *T1* «*bop*» *T2* : *T* ⟧
⟹ *WTrt P h E* (*e1* «*bop*» *e2*) *T*

| *WTrtLAss*:
⟦ *E V* = *Some T*; *WTrt P h E e T'*; *P* ⊢ *T'* ≤ *T* ⟧
⟹ *WTrt P h E* (*V*:=*e*) *Void*

| *WTrtAAcc*:
⟦ *WTrt P h E a* (*T*⌊⌋); *WTrt P h E i Integer* ⟧
⟹ *WTrt P h E* (*a*⌊*i*⌋) *T*

| *WTrtAAccNT*:
  ⟦ *WTrt P h E a NT*; *WTrt P h E i Integer* ⟧
  ⟹ *WTrt P h E* (*a*⌊*i*⌉) *T*

| *WTrtAAss*:
  ⟦ *WTrt P h E a* (*T*⌊⌉); *WTrt P h E i Integer*; *WTrt P h E e T′* ⟧
  ⟹ *WTrt P h E* (*a*⌊*i*⌉ := *e*) *Void*

| *WTrtAAssNT*:
  ⟦ *WTrt P h E a NT*; *WTrt P h E i Integer*; *WTrt P h E e T′* ⟧
  ⟹ *WTrt P h E* (*a*⌊*i*⌉ := *e*) *Void*

| *WTrtALength*:
  WTrt P h E a (*T*⌊⌉) ⟹ *WTrt P h E* (*a*·*length*) *Integer*

| *WTrtALengthNT*:
  WTrt P h E a NT ⟹ *WTrt P h E* (*a*·*length*) *T*

| *WTrtFAcc*:
  ⟦ *WTrt P h E e U*; *class-type-of′ U* = ⌊*C*⌋; *P* ⊢ *C has F*:*T* (*fm*) *in D* ⟧ ⟹
  WTrt P h E (*e*·*F*{*D*}) *T*

| *WTrtFAccNT*:
  WTrt P h E e NT ⟹ *WTrt P h E* (*e*·*F*{*D*}) *T*

| *WTrtFAss*:
  ⟦ *WTrt P h E e1 U*; *class-type-of′ U* = ⌊*C*⌋; *P* ⊢ *C has F*:*T* (*fm*) *in D*; *WTrt P h E e2 T2*; *P* ⊢ *T2* ≤ *T* ⟧
  ⟹ *WTrt P h E* (*e1*·*F*{*D*}:=*e2*) *Void*

| *WTrtFAssNT*:
  ⟦ *WTrt P h E e1 NT*; *WTrt P h E e2 T2* ⟧
  ⟹ *WTrt P h E* (*e1*·*F*{*D*}:=*e2*) *Void*

| *WTrtCAS*:
  ⟦ *WTrt P h E e1 U*; *class-type-of′ U* = ⌊*C*⌋; *P* ⊢ *C has F*:*T* (*fm*) *in D*; *volatile fm*;
    WTrt P h E e2 T2; *P* ⊢ *T2* ≤ *T*; *WTrt P h E e3 T3*; *P* ⊢ *T3* ≤ *T* ⟧
  ⟹ *WTrt P h E* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *Boolean*

| *WTrtCASNT*:
  ⟦ *WTrt P h E e1 NT*; *WTrt P h E e2 T2*; *WTrt P h E e3 T3* ⟧
  ⟹ *WTrt P h E* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *Boolean*

| *WTrtCall*:
  ⟦ *WTrt P h E e U*; *class-type-of′ U* = ⌊*C*⌋; *P* ⊢ *C sees M*:*Ts* → *T* = *meth in D*;
    WTrts P h E es Ts′; *P* ⊢ *Ts′* [≤] *Ts* ⟧
  ⟹ *WTrt P h E* (*e*·*M*(*es*)) *T*

| *WTrtCallNT*:
  ⟦ *WTrt P h E e NT*; *WTrts P h E es Ts* ⟧
  ⟹ *WTrt P h E* (*e*·*M*(*es*)) *T*

| *WTrtBlock*:

⟦ *WTrt P h (E(V↦T)) e T'*; *case vo of None ⇒ True | ⌊v⌋ ⇒ ∃ T'. typeof*$_h$ *v = ⌊T'⌋ ∧ P ⊢ T' ≤ T* ⟧
⟹ *WTrt P h E {V:T=vo; e} T'*

| *WTrtSynchronized*:
⟦ *WTrt P h E o' T*; *is-refT T*; *WTrt P h E e T'* ⟧
⟹ *WTrt P h E (sync(o') e) T'*

| *WTrtInSynchronized*:
⟦ *WTrt P h E (addr a) T*; *WTrt P h E e T'* ⟧
⟹ *WTrt P h E (insync(a) e) T'*

| *WTrtSeq*:
⟦ *WTrt P h E e1 T1*; *WTrt P h E e2 T2* ⟧
⟹ *WTrt P h E (e1;;e2) T2*

| *WTrtCond*:
⟦ *WTrt P h E e Boolean*; *WTrt P h E e1 T1*; *WTrt P h E e2 T2*; *P ⊢ lub(T1, T2) = T* ⟧
⟹ *WTrt P h E (if (e) e1 else e2) T*

| *WTrtWhile*:
⟦ *WTrt P h E e Boolean*; *WTrt P h E c T* ⟧
⟹ *WTrt P h E (while(e) c) Void*

| *WTrtThrow*:
⟦ *WTrt P h E e T*; *P ⊢ T ≤ Class Throwable* ⟧
⟹ *WTrt P h E (throw e) T'*

| *WTrtTry*:
⟦ *WTrt P h E e1 T1*; *WTrt P h (E(V ↦ Class C)) e2 T2*; *P ⊢ T1 ≤ T2* ⟧
⟹ *WTrt P h E (try e1 catch(C V) e2) T2*

| *WTrtNil*: *WTrts P h E* [] []

| *WTrtCons*: ⟦ *WTrt P h E e T*; *WTrts P h E es Ts* ⟧ ⟹ *WTrts P h E (e # es) (T # Ts)*

abbreviation
WTrt-syntax :: *'addr J-prog ⇒ env ⇒ 'heap ⇒ 'addr expr ⇒ ty ⇒ bool* (‹-,-,- ⊢ - : -› [51,51,51]50)
where
P,E,h ⊢ e : T ≡ WTrt P h E e T

abbreviation
WTrts-syntax :: *'addr J-prog ⇒ env ⇒ 'heap ⇒ 'addr expr list ⇒ ty list ⇒ bool* (‹-,-,- ⊢ - [:] -› [51,51,51]50)
where
P,E,h ⊢ es [:] Ts ≡ WTrts P h E es Ts

lemmas [*intro*!] =
WTrtNew WTrtNewArray WTrtCast WTrtInstanceOf WTrtVal WTrtVar WTrtBinOp WTrtLAss
WTrtBlock WTrtSynchronized WTrtInSynchronized WTrtSeq WTrtCond WTrtWhile
WTrtThrow WTrtTry WTrtNil WTrtCons

lemmas [*intro*] =
WTrtFAcc WTrtFAccNT WTrtFAss WTrtFAssNT WTrtCall WTrtCallNT

WTrtAAcc WTrtAAccNT WTrtAAss WTrtAAssNT WTrtALength WTrtALengthNT

4.10.1 Easy consequences

inductive-simps *WTrts-iffs* [*iff*]:
$P,E,h \vdash [] [:] Ts$
$P,E,h \vdash e\#es [:] T\#Ts$
$P,E,h \vdash (e\#es) [:] Ts$

lemma *WTrts-conv-list-all2*: $P,E,h \vdash es [:] Ts = list\text{-}all2\ (WTrt\ P\ h\ E)\ es\ Ts$
by(*induct es arbitrary*: *Ts*)(*auto simp add*: *list-all2-Cons1 elim*: *WTrts.cases*)

lemma [*simp*]: $(P,E,h \vdash es_1 @ es_2 [:] Ts) =$
$(\exists Ts_1\ Ts_2.\ Ts = Ts_1 @ Ts_2 \wedge P,E,h \vdash es_1 [:] Ts_1 \ \&\ P,E,h \vdash es_2[:]Ts_2)$
by(*auto simp add*: *WTrts-conv-list-all2 list-all2-append1 dest*: *list-all2-lengthD*[*symmetric*])

inductive-simps *WTrt-iffs* [*iff*]:
$P,E,h \vdash Val\ v : T$
$P,E,h \vdash Var\ v : T$
$P,E,h \vdash e_1;;e_2 : T_2$
$P,E,h \vdash \{V:T=vo;\ e\} : T'$

inductive-cases *WTrt-elim-cases*[*elim!*]:
$P,E,h \vdash newA\ T\lfloor i \rceil : U$
$P,E,h \vdash v := e : T$
$P,E,h \vdash if\ (e)\ e_1\ else\ e_2 : T$
$P,E,h \vdash while(e)\ c : T$
$P,E,h \vdash throw\ e : T$
$P,E,h \vdash try\ e_1\ catch(C\ V)\ e_2 : T$
$P,E,h \vdash Cast\ D\ e : T$
$P,E,h \vdash e\ instanceof\ U : T$
$P,E,h \vdash a\lfloor i \rceil : T$
$P,E,h \vdash a\lfloor i \rceil := e : T$
$P,E,h \vdash a{\cdot}length : T$
$P,E,h \vdash e{\cdot}F\{D\} : T$
$P,E,h \vdash e{\cdot}F\{D\} := v : T$
$P,E,h \vdash e{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3) : T$
$P,E,h \vdash e_1\ «bop»\ e_2 : T$
$P,E,h \vdash new\ C : T$
$P,E,h \vdash e{\cdot}M(es) : T$
$P,E,h \vdash sync(o')\ e : T$
$P,E,h \vdash insync(a)\ e : T$

4.10.2 Some interesting lemmas

lemma *WTrts-Val*[*simp*]:
$P,E,h \vdash map\ Val\ vs [:] Ts \longleftrightarrow map\ (typeof_h)\ vs = map\ Some\ Ts$
by(*induct vs arbitrary*: *Ts*) *auto*

lemma *WTrt-env-mono*: $P,E,h \vdash e : T \Longrightarrow (\bigwedge E'.\ E \subseteq_m E' \Longrightarrow P,E',h \vdash e : T)$
and *WTrts-env-mono*: $P,E,h \vdash es [:] Ts \Longrightarrow (\bigwedge E'.\ E \subseteq_m E' \Longrightarrow P,E',h \vdash es [:] Ts)$
apply(*induct rule*: *WTrt-WTrts.inducts*)
apply(*simp add*: *WTrtNew*)
apply(*fastforce simp*: *WTrtNewArray*)

apply(*fastforce simp*: *WTrtCast*)
apply(*fastforce simp*: *WTrtInstanceOf*)
apply(*fastforce simp*: *WTrtVal*)
apply(*simp add*: *WTrtVar map-le-def dom-def*)
apply(*fastforce simp add*: *WTrtBinOp*)
apply(*force simp*: *map-le-def*)
apply(*force simp*: *WTrtAAcc*)
apply(*force simp*: *WTrtAAccNT*)
apply(*rule WTrtAAss*, *fastforce*, *blast*, *blast*)
apply(*fastforce*)
apply(*rule WTrtALength*, *blast*)
apply(*blast*)
apply(*fastforce simp*: *WTrtFAcc*)
apply(*simp add*: *WTrtFAccNT*)
apply(*fastforce simp*: *WTrtFAss*)
apply(*fastforce simp*: *WTrtFAssNT*)
apply(*fastforce simp*: *WTrtCAS*)
apply(*fastforce simp*: *WTrtCASNT*)
apply(*fastforce simp*: *WTrtCall*)
apply(*fastforce simp*: *WTrtCallNT*)
apply(*fastforce simp*: *map-le-def*)
apply(*fastforce*)
apply(*fastforce*)
apply(*fastforce*)
apply(*fastforce*)
apply(*fastforce simp*: *WTrtSeq*)
apply(*fastforce simp*: *WTrtCond*)
apply(*fastforce simp*: *WTrtWhile*)
apply(*fastforce simp*: *WTrtThrow*)
apply(*auto simp*: *WTrtTry map-le-def dom-def*)
done

lemma *WT-implies-WTrt*: $P,E \vdash e :: T \Longrightarrow P,E,h \vdash e : T$
and *WTs-implies-WTrts*: $P,E \vdash es\ [::]\ Ts \Longrightarrow P,E,h \vdash es\ [:]\ Ts$
apply(*induct rule*: *WT-WTs.inducts*)
apply *fast*
apply *fast*
apply *fast*
apply *fast*
apply(*fastforce dest*:*typeof-lit-typeof*)
apply(*simp*)
apply(*fastforce intro*: *WT-binop-WTrt-binop*)
apply(*fastforce*)
apply(*erule WTrtAAcc*)
apply(*assumption*)
apply(*erule WTrtAAss*)
apply(*assumption*)+
apply(*erule WTrtALength*)
apply(*fastforce intro*: *has-visible-field*)
apply(*fastforce simp*: *WTrtFAss dest*: *has-visible-field*)
apply(*fastforce simp*: *WTrtCAS dest*: *has-visible-field*)
apply(*fastforce simp*: *WTrtCall*)
apply(*clarsimp simp del*: *fun-upd-apply*, *blast intro*: *typeof-lit-typeof*)
apply(*fastforce*)+

done

lemma *wt-blocks*:
$\bigwedge E.\ [\![\ length\ Vs = length\ Ts;\ length\ vs = length\ Ts\ ]\!] \Longrightarrow$
$(P,E,h \vdash blocks\ Vs\ Ts\ vs\ e : T) =$
$(P,E(Vs[\mapsto]Ts),h \vdash e{:}T \wedge (\exists Ts'.\ map\ (typeof_h)\ vs = map\ Some\ Ts' \wedge P \vdash Ts'\ [\leq]\ Ts))$
apply(*induct Vs Ts vs e rule*:*blocks.induct*)
apply (*force*)
apply *simp-all*
done

end

context *J-heap* **begin**

lemma *WTrt-hext-mono*: $P,E,h \vdash e : T \Longrightarrow h \trianglelefteq h' \Longrightarrow P,E,h' \vdash e : T$
and *WTrts-hext-mono*: $P,E,h \vdash es\ [:]\ Ts \Longrightarrow h \trianglelefteq h' \Longrightarrow P,E,h' \vdash es\ [:]\ Ts$
apply(*induct rule*: *WTrt-WTrts.inducts*)
apply(*simp add*: *WTrtNew*)
apply(*fastforce simp*: *WTrtNewArray*)
apply(*fastforce simp*: *WTrtCast*)
apply(*fastforce simp*: *WTrtInstanceOf*)
apply(*fastforce simp*: *WTrtVal dest*:*hext-typeof-mono*)
apply(*simp add*: *WTrtVar*)
apply(*fastforce simp add*: *WTrtBinOp*)
apply(*fastforce simp add*: *WTrtLAss*)
apply *fastforce*
apply *fastforce*
apply *fastforce*
apply *fastforce*
apply *fastforce*
apply *fastforce*
apply(*fast*)
apply(*simp add*: *WTrtFAccNT*)
apply(*fastforce simp*: *WTrtFAss del*:*WTrt-WTrts.intros WTrt-elim-cases*)
apply(*fastforce simp*: *WTrtFAssNT*)
apply(*fastforce simp*: *WTrtCAS*)
apply(*fastforce simp*: *WTrtCASNT*)
apply(*fastforce simp*: *WTrtCall*)
apply(*fastforce simp*: *WTrtCallNT*)
apply(*fastforce intro*: *hext-typeof-mono*)
apply *fastforce+*
done

end

end

4.11 Progress of Small Step Semantics

theory *Progress*
imports
WellTypeRT

DefAss
SmallStep
../Common/ExternalCallWF
WWellForm
begin

context *J-heap* **begin**

lemma *final-addrE* [*consumes 3*, *case-names addr Throw*]:
⟦ *P*,*E*,*h* ⊢ *e* : *T*; *class-type-of′ T* = ⌊*U*⌋; *final e*;
⋀*a*. *e* = *addr a* ⟹ *R*;
⋀*a*. *e* = *Throw a* ⟹ *R* ⟧ ⟹ *R*
apply(*auto elim*!: *final.cases*)
apply(*case-tac v*)
apply *auto*
done

lemma *finalRefE* [*consumes 3*, *case-names null Class Array Throw*]:
⟦ *P*,*E*,*h* ⊢ *e* : *T*; *is-refT T*; *final e*;
e = *null* ⟹ *R*;
⋀*a C*. ⟦ *e* = *addr a*; *T* = *Class C* ⟧ ⟹ *R*;
⋀*a U*. ⟦ *e* = *addr a*; *T* = *U*⌊⌉ ⟧ ⟹ *R*;
⋀*a*. *e* = *Throw a* ⟹ *R* ⟧ ⟹ *R*
apply(*auto simp*:*final-iff*)
apply(*case-tac v*)
apply(*auto elim*!: *is-refT.cases*)
done

end

theorem (**in** *J-progress*) *red-progress*:
assumes *wf*: *wwf-J-prog P* **and** *hconf*: *hconf h*
shows *progress*: ⟦ *P*,*E*,*h* ⊢ *e* : *T*; *D e* ⌊*dom l*⌋; ¬ *final e* ⟧ ⟹ ∃ *e′ s′ ta*. *extTA*,*P*,*t* ⊢ ⟨*e*,(*h*,*l*)⟩ −*ta*→ ⟨*e′*,*s′*⟩
and *progresss*: ⟦ *P*,*E*,*h* ⊢ *es* [:] *Ts*; *Ds es* ⌊*dom l*⌋; ¬ *finals es* ⟧ ⟹ ∃ *es′ s′ ta*. *extTA*,*P*,*t* ⊢ ⟨*es*,(*h*,*l*)⟩ [−*ta*→] ⟨*es′*,*s′*⟩
proof (*induct arbitrary*: *l* **and** *l rule*:*WTrt-WTrts.inducts*)
case (*WTrtNew C*)
thus *?case* **using** *WTrtNew*
by(*cases allocate h* (*Class-type C*) = {})(*fastforce intro*: *RedNewFail RedNew*)+
next
case (*WTrtNewArray E e T l*)
have *IH*: ⋀*l*. ⟦*D e* ⌊*dom l*⌋; ¬ *final e*⟧ ⟹ ∃ *e′ s′ tas*. *extTA*,*P*,*t* ⊢ ⟨*e*,(*h*,*l*)⟩ −*tas*→ ⟨*e′*, *s′*⟩
and *D*: *D* (*newA T*⌊*e*⌉) ⌊*dom l*⌋
and *ei*: *P*,*E*,*h* ⊢ *e* : *Integer* **by** *fact*+
from *D* **have** *De*: *D e* ⌊*dom l*⌋ **by** *auto*
show *?case*
proof *cases*
assume *final e*
thus *?thesis*
proof (*rule finalE*)
fix *v*
assume *e* [*simp*]: *e* = *Val v*
with *ei* **have** *typeof*$_h$ *v* = *Some Integer* **by** *fastforce*

```
      hence exei: ∃ i. v = Intg i by fastforce
      then obtain i where v: v = Intg i by blast
      thus ?thesis
      proof (cases 0 <=s i)
        case True
        thus ?thesis using True ‹v = Intg i› WTrtNewArray.prems
          by(cases allocate h (Array-type T (nat (sint i))) = {})(auto simp del: split-paired-Ex intro:
RedNewArrayFail RedNewArray)
      next
        assume ¬ 0 <=s i
        hence i <s 0 by simp
       then have extTA,P,t ⊢ ⟨newA T⌊Val(Intg i)⌉,(h, l)⟩ −ε→ ⟨THROW NegativeArraySize,(h, l)⟩
          by − (rule RedNewArrayNegative, auto)
        with e v show ?thesis by blast
      qed
    next
      fix exa
      assume e: e = Throw exa
      then have extTA,P,t ⊢ ⟨newA T⌊Throw exa⌉,(h, l)⟩ −ε→ ⟨Throw exa,(h, l)⟩
        by − (rule NewArrayThrow)
      with e show ?thesis by blast
    qed
  next
    assume ¬ final e
    with IH De have exes: ∃ e' s' ta. extTA,P,t ⊢ ⟨e,(h, l)⟩ −ta→ ⟨e',s'⟩ by simp
    then obtain e' s' ta where extTA,P,t ⊢ ⟨e,(h, l)⟩ −ta→ ⟨e',s'⟩ by blast
    hence extTA,P,t ⊢ ⟨newA T⌊e⌉,(h, l)⟩ −ta→ ⟨newA T⌊e'⌉,s'⟩ by − (rule NewArrayRed)
    thus ?thesis by blast
  qed
next
  case (WTrtCast E e T U l)
  have wte: P,E,h ⊢ e : T
   and IH: ⋀l. ⟦𝒟 e ⌊dom l⌋; ¬ final e⟧
              ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨e,(h,l)⟩ −tas→ ⟨e',s'⟩
   and D: 𝒟 (Cast U e) ⌊dom l⌋ by fact+
  from D have De: 𝒟 e ⌊dom l⌋ by auto
  show ?case
  proof (cases final e)
    assume final e
    thus ?thesis
    proof (rule finalE)
      fix v
      assume ev: e = Val v
      with WTrtCast obtain V where thvU: typeof_h v = ⌊V⌋ by fastforce
      thus ?thesis
      proof (cases P ⊢ V ≤ U)
        assume P ⊢ V ≤ U
        with thvU have extTA,P,t ⊢ ⟨Cast U (Val v),(h, l)⟩ −ε→ ⟨Val v,(h,l)⟩
          by − (rule RedCast, auto)
        with ev show ?thesis by blast
      next
        assume ¬ P ⊢ V ≤ U
        with thvU have extTA,P,t ⊢ ⟨Cast U (Val v),(h, l)⟩ −ε→ ⟨THROW ClassCast,(h,l)⟩
          by − (rule RedCastFail, auto)
```

```
      with ev show ?thesis by blast
    qed
  next
    fix a
    assume e = Throw a
    thus ?thesis by(blast intro!:CastThrow)
  qed
 next
  assume nf: ¬ final e
  from IH[OF De nf] show ?thesis by (blast intro:CastRed)
 qed
next
 case (WTrtInstanceOf E e T U l)
 have wte: P,E,h ⊢ e : T
  and IH: ⋀l. ⟦𝒟 e ⌊dom l⌋; ¬ final e⟧
            ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨e,(h,l)⟩ −tas→ ⟨e',s'⟩
  and D: 𝒟 (e instanceof U) ⌊dom l⌋ by fact+
 from D have De: 𝒟 e ⌊dom l⌋ by auto
 show ?case
 proof (cases final e)
  assume final e
  thus ?thesis
  proof (rule finalE)
   fix v
   assume ev: e = Val v
   with WTrtInstanceOf obtain V where thvU: typeof_h v = ⌊V⌋ by fastforce
   hence extTA,P,t ⊢ ⟨(Val v) instanceof U,(h, l)⟩ −ε→ ⟨Val (Bool (v ≠ Null ∧ P ⊢ V ≤ U)),(h,l)⟩
    by −(rule RedInstanceOf, auto)
   with ev show ?thesis by blast
  next
   fix a
   assume e = Throw a
   thus ?thesis by(blast intro!:InstanceOfThrow)
  qed
 next
  assume nf: ¬ final e
  from IH[OF De nf] show ?thesis by (blast intro:InstanceOfRed)
 qed
next
 case WTrtVal thus ?case by(simp add:final-iff)
next
 case WTrtVar thus ?case by(fastforce intro:RedVar simp:hyper-isin-def)
next
 case (WTrtBinOp E e1 T1 e2 T2 bop T)
 show ?case
 proof cases
  assume final e1
  thus ?thesis
  proof (rule finalE)
   fix v1 assume [simp]: e1 = Val v1
   show ?thesis
   proof cases
    assume final e2
    thus ?thesis
```

```
      proof (rule finalE)
        fix v2 assume [simp]: e2 = Val v2
        with WTrtBinOp have type: typeof_h v1 = ⌊T1⌋ typeof_h v2 = ⌊T2⌋ by auto
        from binop-progress[OF this ‹P ⊢ T1 «bop» T2 : T›] obtain va
          where binop bop v1 v2 = ⌊va⌋ by blast
        thus ?thesis by(cases va)(fastforce intro: RedBinOp RedBinOpFail)+
      next
        fix a assume e2 = Throw a
        thus ?thesis by(fastforce intro:BinOpThrow2)
      qed
    next
      assume ¬ final e2 with WTrtBinOp show ?thesis
        by simp (fast intro!:BinOpRed2)
      qed
  next
    fix a assume e1 = Throw a
    thus ?thesis by simp (fast intro:BinOpThrow1)
  qed
next
  assume ¬ final e1 with WTrtBinOp show ?thesis
    by simp (fast intro:BinOpRed1)
  qed
next
  case (WTrtLAss E V T e T')
  show ?case
  proof cases
    assume final e with WTrtLAss show ?thesis
      by(fastforce simp:final-iff intro!:RedLAss LAssThrow)
  next
    assume ¬ final e with WTrtLAss show ?thesis
      by simp (fast intro:LAssRed)
  qed
next
  case (WTrtAAcc E a T i l)
  have wte: P,E,h ⊢ a : T⌊⌉
   and wtei: P,E,h ⊢ i : Integer
   and IHa: ⋀l. ⟦𝒟 a ⌊dom l⌋; ¬ final a⟧
                ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨a,(h,l)⟩ −tas→ ⟨e',s'⟩
   and IHi: ⋀l. ⟦𝒟 i ⌊dom l⌋; ¬ final i⟧
                ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨i,(h,l)⟩ −tas→ ⟨e',s'⟩
   and D: 𝒟 (a⌊i⌉) ⌊dom l⌋ by fact+
  have ref: is-refT (T⌊⌉) by simp
  from D have Da: 𝒟 a ⌊dom l⌋ by simp
  show ?case
  proof (cases final a)
    assume final a
    with wte ref show ?case
    proof (cases rule: finalRefE)
      case null
      thus ?thesis
      proof (cases final i)
        assume final i
        thus ?thesis
        proof (rule finalE)
```

```
      fix v
      assume i: i = Val v
      have extTA,P,t ⊢ ⟨null⌊Val v⌉, (h, l)⟩ −ε→ ⟨THROW NullPointer, (h,l)⟩
        by(rule RedAAccNull)
      with i null show ?thesis by blast
    next
      fix ex
      assume i: i = Throw ex
      have extTA,P,t ⊢ ⟨null⌊Throw ex⌉, (h, l)⟩ −ε→ ⟨Throw ex, (h,l)⟩
        by(rule AAccThrow2)
      with i null show ?thesis by blast
    qed
  next
    assume ¬ final i
    from WTrtAAcc null show ?thesis
      by simp
  qed
next
  case (Array ad U)
  with wte obtain n where ty: typeof-addr h ad = ⌊Array-type U n⌋ by auto
  thus ?thesis
  proof (cases final i)
    assume final i
    thus ?thesis
    proof(rule finalE)
      fix v
      assume [simp]: i = Val v
      with wtei have typeof_h v = Some Integer by fastforce
      hence ∃ i. v = Intg i by fastforce
      then obtain i where [simp]: v = Intg i by blast
      thus ?thesis
      proof (cases i <s 0 ∨ sint i ≥ int n)
        case True
        with WTrtAAcc Array ty show ?thesis by (fastforce intro: RedAAccBounds)
      next
        case False
        then have nat (sint i) < n
          by (simp add: not-le word-sless-alt nat-less-iff)
        with ty have P,h ⊢ ad@ACell (nat (sint i)) : U by(auto intro!: addr-loc-type.intros)
        from heap-read-total[OF hconf this]
        obtain v where heap-read h ad (ACell (nat (sint i))) v by blast
        with False Array ty show ?thesis by(fastforce intro: RedAAcc)
      qed
    next
      fix ex
      assume i = Throw ex
      with WTrtAAcc Array show ?thesis by (fastforce intro: AAccThrow2)
    qed
  next
    assume ¬ final i
    with WTrtAAcc Array show ?thesis by (fastforce intro: AAccRed2)
  qed
next
  fix ex
```

```
      assume a = Throw ex
      with WTrtAAcc show ?thesis by (fastforce intro: AAccThrow1)
    qed simp
  next
    assume ¬ final a
    with WTrtAAcc show ?thesis by (fastforce intro: AAccRed1)
  qed
next
  case (WTrtAAccNT E a i T l)
  have wte: P,E,h ⊢ a : NT
   and wtei: P,E,h ⊢ i : Integer
   and IHa: ⋀l. ⟦𝒟 a ⌊dom l⌋; ¬ final a⟧
              ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨a,(h,l)⟩ −tas→ ⟨e',s'⟩
   and IHi: ⋀l. ⟦𝒟 i ⌊dom l⌋; ¬ final i⟧
              ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨i,(h,l)⟩ −tas→ ⟨e',s'⟩ by fact+
  have ref: is-refT NT by simp
  with WTrtAAccNT have Da: 𝒟 a ⌊dom l⌋ by simp
  thus ?case
  proof (cases final a)
    case True
    with wte ref show ?thesis
    proof (cases rule: finalRefE)
      case null
      thus ?thesis
      proof (cases final i)
        assume final i
        thus ?thesis
        proof (rule finalE)
          fix v
          assume i: i = Val v
          have extTA,P,t ⊢ ⟨null⌊Val v⌉, (h, l)⟩ −ε→ ⟨THROW NullPointer, (h,l)⟩
            by (rule RedAAccNull)
          with WTrtAAccNT ‹final a› null ‹final i› i show ?thesis by blast
        next
          fix ex
          assume i: i = Throw ex
          have extTA,P,t ⊢ ⟨null⌊Throw ex⌉, (h, l)⟩ −ε→ ⟨Throw ex, (h,l)⟩
            by(rule AAccThrow2)
          with WTrtAAccNT ‹final a› null ‹final i› i show ?thesis by blast
        qed
      next
        assume ¬ final i
        with WTrtAAccNT null show ?thesis
          by(fastforce intro: AAccRed2)
      qed
    next
      case Throw thus ?thesis by (fastforce intro: AAccThrow1)
    qed simp-all
  next
    case False
    with WTrtAAccNT Da show ?thesis by (fastforce intro:AAccRed1)
  qed
next
  case (WTrtAAss E a T i e T' l)
```

```
  have wta: P,E,h ⊢ a : T⌊⌋
    and wti: P,E,h ⊢ i : Integer
    and wte: P,E,h ⊢ e : T'
    and D: 𝒟 (a⌊i⌋ := e) ⌊dom l⌋
    and IH1: ⋀l. ⟦𝒟 a ⌊dom l⌋; ¬ final a⟧ ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨a,(h, l)⟩ −tas→ ⟨e',s'⟩
    and IH2: ⋀l. ⟦𝒟 i ⌊dom l⌋; ¬ final i⟧ ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨i,(h, l)⟩ −tas→ ⟨e',s'⟩
    and IH3: ⋀l. ⟦𝒟 e ⌊dom l⌋; ¬ final e⟧ ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨e,(h, l)⟩ −tas→ ⟨e',s'⟩ by
fact+
  have ref: is-refT (T⌊⌋) by simp
  show ?case
  proof (cases final a)
    assume fa: final a
    with wta ref show ?thesis
    proof(cases rule: finalRefE)
      case null
      show ?thesis
      proof(cases final i)
        assume final i
        thus ?thesis
        proof (rule finalE)
          fix v
          assume i: i = Val v
          with wti have typeof_h v = Some Integer by fastforce
          then obtain idx where v = Intg idx by fastforce
          thus ?thesis
          proof (cases final e)
            assume final e
            thus ?thesis
            proof (rule finalE)
              fix w
              assume e = Val w
              with WTrtAAss null show ?thesis by (fastforce intro: RedAAssNull)
            next
              fix ex
              assume e = Throw ex
              with WTrtAAss null show ?thesis by (fastforce intro: AAssThrow3)
            qed
          next
            assume ¬ final e
            with WTrtAAss null show ?thesis by (fastforce intro: AAssRed3)
          qed
        next
          fix ex
          assume i = Throw ex
          with WTrtAAss null show ?thesis by (fastforce intro: AAssThrow2)
        qed
      next
        assume ¬ final i
        with WTrtAAss null show ?thesis by (fastforce intro: AAssRed2)
      qed
    next
      case (Array ad U)
      with wta obtain n where ty: typeof-addr h ad = ⌊Array-type U n⌋ by auto
      thus ?thesis
```

```
proof (cases final i)
  assume fi: final i
  thus ?thesis
  proof (rule finalE)
    fix v
    assume ivalv: i = Val v
    with wti have typeof_h v = Some Integer by fastforce
    then obtain idx where vidx: v = Intg idx by fastforce
    thus ?thesis
    proof (cases final e)
      assume fe: final e
      thus ?thesis
      proof(rule finalE)
        fix w
        assume evalw: e = Val w
        show ?thesis
        proof(cases idx <s 0 ∨ sint idx ≥ int n)
          case True
          with ty evalw Array ivalv vidx show ?thesis by(fastforce intro: RedAAssBounds)
        next
          case False
          then have nat (sint idx) < n
            by (simp add: not-le word-sless-alt nat-less-iff)
          with ty have adal: P,h ⊢ ad@ACell (nat (sint idx)) : U
            by(auto intro!: addr-loc-type.intros)
          show ?thesis
          proof(cases P ⊢ T' ≤ U)
            case True
            with wte evalw have P,h ⊢ w :≤ U
              by(auto simp add: conf-def)
            from heap-write-total[OF hconf adal this]
            obtain h' where h': heap-write h ad (ACell (nat (sint idx))) w h' ..
            with ty False vidx ivalv evalw Array wte True
            show ?thesis by(fastforce intro: RedAAss)
          next
            case False
            with ty vidx ivalv evalw Array wte ‹¬ (idx <s 0 ∨ sint idx ≥ int n)›
            show ?thesis by(fastforce intro: RedAAssStore)
          qed
        qed
      next
        fix ex
        assume e = Throw ex
        with Array ivalv show ?thesis by (fastforce intro: AAssThrow3)
      qed
    next
      assume ¬ final e
      with WTrtAAss Array fi ivalv vidx show ?thesis by (fastforce intro: AAssRed3)
    qed
  next
    fix ex
    assume i = Throw ex
    with WTrtAAss Array show ?thesis by (fastforce intro: AAssThrow2)
  qed
```

```
      next
        assume ¬ final i
        with WTrtAAss Array show ?thesis by (fastforce intro: AAssRed2)
      qed
    next
      fix ex
      assume a = Throw ex
      with WTrtAAss show ?thesis by (fastforce intro:AAssThrow1)
    qed simp-all
  next
    assume ¬ final a
    with WTrtAAss show ?thesis by (fastforce intro: AAssRed1)
  qed
next
  case (WTrtAAssNT E a i e T' l)
  have wta: P,E,h ⊢ a : NT
    and wti: P,E,h ⊢ i : Integer
    and wte: P,E,h ⊢ e : T'
    and D: 𝒟 (a⌊i⌉ := e) ⌊dom l⌋
    and IH1: ⋀l. ⟦𝒟 a ⌊dom l⌋; ¬ final a⟧ ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨a,(h, l)⟩ −tas→ ⟨e',s'⟩
    and IH2: ⋀l. ⟦𝒟 i ⌊dom l⌋; ¬ final i⟧ ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨i,(h, l)⟩ −tas→ ⟨e',s'⟩
    and IH3: ⋀l. ⟦𝒟 e ⌊dom l⌋; ¬ final e⟧ ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨e,(h, l)⟩ −tas→ ⟨e',s'⟩ by
fact+
  have ref: is-refT NT by simp
  show ?case
  proof (cases final a)
    assume fa: final a
    show ?case
    proof (cases final i)
      assume fi: final i
      show ?case
      proof (cases final e)
        assume fe: final e
        with WTrtAAssNT fa fi show ?thesis
          by (fastforce simp:final-iff intro: RedAAssNull AAssThrow1 AAssThrow2 AAssThrow3)
      next
        assume ¬ final e
        with WTrtAAssNT fa fi show ?thesis
          by (fastforce simp: final-iff intro!:AAssRed3 AAssThrow1 AAssThrow2)
      qed
    next
      assume ¬ final i
      with WTrtAAssNT fa show ?thesis
        by (fastforce simp: final-iff intro!:AAssRed2 AAssThrow1)
    qed
  next
    assume ¬ final a
    with WTrtAAssNT show ?thesis by (fastforce simp: final-iff intro!:AAssRed1)
  qed
next
  case (WTrtALength E a T l)
  show ?case
  proof(cases final a)
    case True
```

```
    note wta = ‹P,E,h ⊢ a : T⌊⌉›
    thus ?thesis
    proof(rule finalRefE[OF - - True])
      show is-refT (T⌊⌉) by simp
    next
      assume a = null
      thus ?thesis by(fastforce intro: RedALengthNull)
    next
      fix ad U
      assume a = addr ad and T⌊⌉ = U⌊⌉
      with wta show ?thesis by(fastforce intro: RedALength)
    next
      fix ad
      assume a = Throw ad
      thus ?thesis by (fastforce intro: ALengthThrow)
    qed simp
  next
    case False
    from ‹𝒟 (a·length) ⌊dom l⌋› have 𝒟 a ⌊dom l⌋ by simp
    with False ‹⟦𝒟 a ⌊dom l⌋; ¬ final a⟧ ⟹ ∃ e' s' ta. extTA,P,t ⊢ ⟨a,(h, l)⟩ −ta→ ⟨e',s'⟩›
    obtain e' s' ta where extTA,P,t ⊢ ⟨a,(h, l)⟩ −ta→ ⟨e',s'⟩ by blast
    thus ?thesis by(blast intro: ALengthRed)
  qed
next
  case (WTrtALengthNT E a T l)
  show ?case
  proof(cases final a)
    case True
    note wta = ‹P,E,h ⊢ a : NT›
    thus ?thesis
    proof(rule finalRefE[OF - - True])
      show is-refT NT by simp
    next
      assume a = null
      thus ?thesis by(blast intro: RedALengthNull)
    next
      fix ad
      assume a = Throw ad
      thus ?thesis by(blast intro: ALengthThrow)
    qed simp-all
  next
    case False
    from ‹𝒟 (a·length) ⌊dom l⌋› have 𝒟 a ⌊dom l⌋ by simp
    with False ‹⟦𝒟 a ⌊dom l⌋; ¬ final a⟧ ⟹ ∃ e' s' ta. extTA,P,t ⊢ ⟨a,(h, l)⟩ −ta→ ⟨e',s'⟩›
    obtain e' s' ta where extTA,P,t ⊢ ⟨a,(h, l)⟩ −ta→ ⟨e',s'⟩ by blast
    thus ?thesis by(blast intro: ALengthRed)
  qed
next
  case (WTrtFAcc E e U C F T fm D l)
  have wte: P,E,h ⊢ e : U
    and icto: class-type-of' U = ⌊C⌋
   and field: P ⊢ C has F:T (fm) in D by fact+
  show ?case
  proof cases
```

```
    assume final e
    with wte icto show ?thesis
    proof (cases rule: final-addrE)
      case (addr a)
      with wte obtain hU where ty: typeof-addr h a = ⌊hU⌋ U = ty-of-htype hU by auto
      with icto field have P,h ⊢ a@CField D F : T by(auto intro: addr-loc-type.intros)
      from heap-read-total[OF hconf this]
      obtain v where heap-read h a (CField D F) v by blast
      with addr ty show ?thesis by(fastforce intro: RedFAcc)
    next
      fix a assume e = Throw a
      thus ?thesis by(fastforce intro:FAccThrow)
    qed
  next
    assume ¬ final e with WTrtFAcc show ?thesis
      by(fastforce intro!:FAccRed)
  qed
next
  case (WTrtFAccNT E e F D T l)
  show ?case
  proof cases
    assume final e  — e is null or throw
    with WTrtFAccNT show ?thesis
      by(fastforce simp:final-iff intro: RedFAccNull FAccThrow)
  next
    assume ¬ final e — e reduces by IH
    with WTrtFAccNT show ?thesis by simp (fast intro:FAccRed)
  qed
next
  case (WTrtFAss E e1 U C F T fm D e2 T2 l)
  have wte1: P,E,h ⊢ e1 : U
    and icto: class-type-of' U = ⌊C⌋
    and field: P ⊢ C has F:T (fm) in D by fact+
  show ?case
  proof cases
    assume final e1
    with wte1 icto show ?thesis
    proof (rule final-addrE)
      fix a assume e1: e1 = addr a
      show ?thesis
      proof cases
        assume final e2
        thus ?thesis
        proof (rule finalE)
          fix v assume e2: e2 = Val v
          from wte1 field icto e1 have adal: P,h ⊢ a@CField D F : T
            by(auto intro: addr-loc-type.intros)
          from e2 ‹P ⊢ T2 ≤ T› ‹P,E,h ⊢ e2 : T2›
          have P,h ⊢ v :≤ T by(auto simp add: conf-def)
          from heap-write-total[OF hconf adal this] obtain h'
            where heap-write h a (CField D F) v h' ..
          with wte1 field e1 e2 show ?thesis
            by(fastforce intro: RedFAss)
        next
```

```
        fix a assume e2 = Throw a
        thus ?thesis using e1 by(fastforce intro:FAssThrow2)
      qed
    next
      assume ¬ final e2 with WTrtFAss ‹final e1› e1 show ?thesis
        by simp (fast intro!:FAssRed2)
      qed
    next
      fix a assume e1 = Throw a
      thus ?thesis by(fastforce intro:FAssThrow1)
    qed
  next
    assume ¬ final e1 with WTrtFAss show ?thesis
      by(simp del: split-paired-Ex)(blast intro!:FAssRed1)
  qed
next
  case (WTrtFAssNT E e1 e2 T2 F D l)
  show ?case
  proof cases
    assume final e1 — e1 is null or throw
    show ?thesis
    proof cases
      assume final e2 — e2 is Val or throw
      with WTrtFAssNT ‹final e1› show ?thesis
        by(fastforce simp:final-iff intro: RedFAssNull FAssThrow1 FAssThrow2)
    next
      assume ¬ final e2 — e2 reduces by IH
      with WTrtFAssNT ‹final e1› show ?thesis
        by (fastforce  simp:final-iff intro!:FAssRed2 FAssThrow1)
    qed
  next
    assume ¬ final e1 — e1 reduces by IH
    with WTrtFAssNT show ?thesis by (fastforce intro:FAssRed1)
  qed
next
  case (WTrtCAS E e1 U C F T fm D e2 T2 e3 T3)
  show ?case
  proof(cases final e1)
    case e1: True
    with WTrtCAS.hyps(1,3) show ?thesis
    proof(rule final-addrE)
      fix a
      assume e1: e1 = addr a
      with WTrtCAS.hyps(1) obtain hU
        where ty: typeof-addr h a = ⌊hU⌋ U = ty-of-htype hU by auto
     with WTrtCAS.hyps(3,4) have adal: P,h ⊢ a@CField D F : T by(auto intro: addr-loc-type.intros)
      from heap-read-total[OF hconf this]
      obtain v where v: heap-read h a (CField D F) v by blast
      show ?thesis
      proof(cases final e2)
        case e2: True
        show ?thesis
        proof(cases final e3)
          case e3: True
```

```
        consider (Val2) v2 where e2 = Val v2 | (Throw2) a2 where e2 = Throw a2
          using e2 by(auto simp add: final-iff)
        then show ?thesis
        proof(cases)
          case Val2
          consider (Succeed) v3 where e3 = Val v3 v2 = v
            | (Fail) v3 where e3 = Val v3 v2 ≠ v
            | (Throw3) a3 where e3 = Throw a3
            using e3 by(auto simp add: final-iff)
          then show ?thesis
          proof cases
            case Succeed
            with WTrtCAS.hyps(9,11) adal have P,h ⊢ v3 :≤ T by(auto simp add: conf-def)
            from heap-write-total[OF hconf adal this] obtain h'
              where heap-write h a (CField D F) v3 h' ..
            with Val2 e1 v Succeed show ?thesis
              by(auto intro: RedCASSucceed simp del: split-paired-Ex)
          next
            case Fail
            with Val2 e1 v show ?thesis
              by(auto intro: RedCASFail simp del: split-paired-Ex)
          next
            case Throw3
            then show ?thesis using e1 Val2 by(auto intro: CASThrow3 simp del: split-paired-Ex)
          qed
        next
          case Throw2
          then show ?thesis using e1 by(auto intro: CASThrow2 simp del: split-paired-Ex)
        qed
      next
        case False
        with WTrtCAS e1 e2 show ?thesis
          by(fastforce simp del: split-paired-Ex simp add: final-iff intro: CASRed3 CASThrow2)
      qed
    next
      case False
      with WTrtCAS e1 show ?thesis
        by(fastforce intro: CASRed2 CASThrow2 simp del: split-paired-Ex)
    qed
  qed(fastforce intro: CASThrow)
next
  case False
  then show ?thesis using WTrtCAS by(fastforce intro: CASRed1)
qed
next
  case (WTrtCASNT E e1 e2 T2 e3 T3 D F)
  note [simp del] = split-paired-Ex
  show ?case
  proof(cases final e1)
    case e1: True
    show ?thesis
    proof(cases final e2)
      case e2: True
      show ?thesis
```

```
    proof(cases final e3)
      case True
      with e1 e2 WTrtCASNT show ?thesis
        by(fastforce simp add: final-iff intro: CASNull CASThrow CASThrow2 CASThrow3)
    next
      case False
      with e1 e2 WTrtCASNT show ?thesis
        by(fastforce simp add: final-iff intro: CASRed3 CASThrow CASThrow2)
    qed
  next
    case False
    with e1 WTrtCASNT show ?thesis
      by(fastforce simp add: final-iff intro: CASRed2 CASThrow)
  qed
next
  case False
  with WTrtCASNT show ?thesis
    by(fastforce simp add: final-iff intro: CASRed1)
qed
next
  case (WTrtCall E e U C M Ts T meth D es Ts' l)
  have wte: P,E,h ⊢ e : U
    and icto: class-type-of' U = ⌊C⌋ by fact+
  have IHe: ⋀l. ⟦ 𝒟 e ⌊dom l⌋; ¬ final e ⟧
          ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨e,(h, l)⟩ −tas→ ⟨e',s'⟩ by fact
  have sees: P ⊢ C sees M: Ts→T = meth in D by fact
  have wtes: P,E,h ⊢ es [:] Ts' by fact
  have IHes: ⋀l. ⟦𝒟s es ⌊dom l⌋; ¬ finals es⟧ ⟹ ∃ es' s' ta. extTA,P,t ⊢ ⟨es,(h, l)⟩ [−ta→] ⟨es',s'⟩
by fact
  have subtype: P ⊢ Ts' [≤] Ts by fact
  have dae: 𝒟 (e•M(es)) ⌊dom l⌋ by fact
  show ?case
  proof(cases final e)
    assume fine: final e
    with wte icto show ?thesis
    proof (rule final-addrE)
      fix a assume e-addr: e = addr a
      show ?thesis
      proof(cases ∃ vs. es = map Val vs)
        assume es: ∃ vs. es = map Val vs
        from wte e-addr obtain hU where ha: typeof-addr h a = ⌊hU⌋ U = ty-of-htype hU by(auto)
      have length es = length Ts' using wtes by(auto simp add: WTrts-conv-list-all2 dest: list-all2-lengthD)
        moreover from subtype have length Ts' = length Ts by(auto dest: list-all2-lengthD)
        ultimately have esTs: length es = length Ts by(auto)
        show ?thesis
        proof(cases meth)
          case (Some pnsbody)
          with esTs e-addr ha sees subtype es sees-wf-mdecl[OF wf sees] icto
          show ?thesis by(cases pnsbody) (fastforce intro!: RedCall simp:list-all2-iff wf-mdecl-def)
        next
          case None
          with sees wf have D•M(Ts) :: T by(auto intro: sees-wf-native)
          moreover from es obtain vs where vs: es = map Val vs ..
          with wtes have tyes: map typeof_h vs = map Some Ts' by simp
```

```
        with ha ‹D·M(Ts) :: T› icto sees None
        have P,h ⊢ a·M(vs) : T using subtype by(auto simp add: external-WT'-iff)
        from external-call-progress[OF wf this hconf, of t] obtain ta va h'
          where P,t ⊢ ⟨a·M(vs), h⟩ −ta→ext ⟨va, h'⟩ by blast
        thus ?thesis using ha icto None sees vs e-addr
          by(fastforce intro: RedCallExternal simp del: split-paired-Ex)
      qed
    next
      assume ¬(∃ vs. es = map Val vs)
      hence not-all-Val: ¬(∀ e ∈ set es. ∃ v. e = Val v)
        by(simp add:ex-map-conv)
      let ?ves = takeWhile (λe. ∃ v. e = Val v) es
      let ?rest = dropWhile (λe. ∃ v. e = Val v) es
      let ?ex = hd ?rest let ?rst = tl ?rest
      from not-all-Val have nonempty: ?rest ≠ [] by auto
      hence es: es = ?ves @ ?ex # ?rst by simp
      have ∀ e ∈ set ?ves. ∃ v. e = Val v by(fastforce dest:set-takeWhileD)
      then obtain vs where ves: ?ves = map Val vs
        using ex-map-conv by blast
      show ?thesis
      proof cases
        assume final ?ex
        moreover from nonempty have ¬(∃ v. ?ex = Val v)
          by(auto simp:neq-Nil-conv simp del:dropWhile-eq-Nil-conv)
            (simp add:dropWhile-eq-Cons-conv)
        ultimately obtain b where ex-Throw: ?ex = Throw b
          by(fast elim!:finalE)
        show ?thesis using e-addr es ex-Throw ves
          by(fastforce intro:CallThrowParams)
      next
        assume not-fin: ¬ final ?ex
        have finals es = finals(?ves @ ?ex # ?rst) using es
          by(rule arg-cong)
        also have ... = finals(?ex # ?rst) using ves by simp
        finally have finals es = finals(?ex # ?rst) .
        hence ¬ finals es using not-finals-ConsI[OF not-fin] by blast
        thus ?thesis using e-addr dae IHes  by(fastforce intro!:CallParams)
      qed
    qed
  next
    fix a assume e = Throw a
    thus ?thesis by(fast intro!:CallThrowObj)
  qed
next
  assume ¬ final e
  with WTrtCall show ?thesis by(simp del: split-paired-Ex)(blast intro!:CallObj)
qed
next
  case (WTrtCallNT E e es Ts M T l)
  have wte: P,E,h ⊢ e : NT by fact
  have IHe: ⋀l. ⟦ 𝒟 e ⌊dom l⌋; ¬ final e ⟧
        ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨e,(h, l)⟩ −tas→ ⟨e',s'⟩  by fact
  have IHes: ⋀l. ⟦𝒟s es ⌊dom l⌋; ¬ finals es⟧ ⟹ ∃ es' s' ta. extTA,P,t ⊢ ⟨es,(h, l)⟩ [−ta→] ⟨es',s'⟩
by fact
```

```
  have wtes: P,E,h ⊢ es [:] Ts by fact
  have dae: 𝒟 (e•M(es)) ⌊dom l⌋ by fact
  show ?case
  proof(cases final e)
    assume final e
    moreover
    { fix v assume e = Val v
      hence e = null using WTrtCallNT by simp
      have ?case
      proof cases
        assume finals es
        moreover
        { fix vs assume es = map Val vs
          with WTrtCallNT ‹e = null› ‹finals es› have ?thesis by(fastforce intro: RedCallNull) }
        moreover
        { fix vs a es' assume es = map Val vs @ Throw a # es'
          with WTrtCallNT ‹e = null› ‹finals es› have ?thesis by(fastforce intro: CallThrowParams)
}
        ultimately show ?thesis by(fastforce simp:finals-iff)
      next
        assume ¬ finals es — es reduces by IH
        with WTrtCallNT ‹e = null› show ?thesis by(fastforce intro: CallParams)
      qed
    }
    moreover
    { fix a assume e = Throw a
      with WTrtCallNT have ?case by(fastforce intro: CallThrowObj) }
    ultimately show ?thesis by(fastforce simp:final-iff)
  next
    assume ¬ final e — e reduces by IH
    with WTrtCallNT show ?thesis by (fastforce intro:CallObj)
  qed
next
  case (WTrtBlock E V T e T' vo l)
  have IH: ⋀l. ⟦𝒟 e ⌊dom l⌋; ¬ final e⟧
              ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨e,(h,l)⟩ −tas→ ⟨e',s'⟩
   and D: 𝒟 {V:T=vo; e} ⌊dom l⌋ by fact+
  show ?case
  proof cases
    assume final e
    thus ?thesis
    proof (rule finalE)
      fix v assume e = Val v thus ?thesis by(fast intro:RedBlock)
    next
      fix a assume e = Throw a
      thus ?thesis by(fast intro:BlockThrow)
    qed
  next
    assume not-fin: ¬ final e
    from D have De: 𝒟 e ⌊dom(l(V:=vo))⌋ by(auto simp add:hyperset-defs)
    from IH[OF De not-fin]
    obtain h' l' e' tas where red: extTA,P,t ⊢ ⟨e,(h,l(V:=vo))⟩ −tas→ ⟨e',(h',l')⟩
      by auto
    thus ?thesis by(blast intro: BlockRed)
```

```
  qed
next
  case (WTrtSynchronized E o' T e T' l)
  note wto = ‹P,E,h ⊢ o' : T›
  note IHe = ‹⋀l. ⟦𝒟 e ⌊dom l⌋; ¬ final e ⟧ ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨e,(h, l)⟩ −tas→ ⟨e',s'⟩›
  note wte = ‹P,E,h ⊢ e : T'›
  note IHo = ‹⋀l. ⟦𝒟 o' ⌊dom l⌋; ¬ final o' ⟧ ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨o',(h, l)⟩ −tas→ ⟨e',s'⟩›
  note refT = ‹is-refT T›
  note dae = ‹𝒟 (sync(o') e) ⌊dom l⌋›
  show ?case
  proof(cases final o')
    assume fino: final o'
    thus ?thesis
    proof (rule finalE)
      fix v
      assume oval: o' = Val v
      with wto refT show ?thesis
      proof(cases v)
        assume vnull: v = Null
        with oval vnull show ?thesis
          by(fastforce intro: SynchronizedNull)
      next
        fix ad
        assume vaddr: v = Addr ad
        thus ?thesis using oval
          by(fastforce intro: LockSynchronized)
      qed(auto elim: refTE)
    next
      fix a
      assume othrow: o' = Throw a
      thus ?thesis by(fastforce intro: SynchronizedThrow1)
    qed
  next
    assume nfino: ¬ final o'
    with dae IHo show ?case by(fastforce intro: SynchronizedRed1)
  qed
next
  case (WTrtInSynchronized E a T e T' l)
  show ?case
  proof(cases final e)
    case True thus ?thesis
      by(fastforce elim!: finalE intro: UnlockSynchronized SynchronizedThrow2)
  next
    case False
    moreover from ‹𝒟 (insync(a) e) ⌊dom l⌋› have 𝒟 e ⌊dom l⌋ by simp
    moreover note IHe = ‹⋀l. ⟦𝒟 e ⌊dom l⌋; ¬ final e⟧ ⟹ ∃ e' s' tas. extTA,P,t ⊢ ⟨e,(h, l)⟩ −tas→
⟨e',s'⟩›
    ultimately show ?thesis by(fastforce intro: SynchronizedRed2)
  qed
next
  case (WTrtSeq E e1 T1 e2 T2 l)
  show ?case
  proof cases
    assume final e1
```

```
    thus ?thesis
      by(fast elim:finalE intro:intro:RedSeq SeqThrow)
  next
    assume ¬ final e1 with WTrtSeq show ?thesis
      by(simp del: split-paired-Ex)(blast intro!:SeqRed)
  qed
next
  case (WTrtCond E e e1 T1 e2 T2 T l)
  have wt: P,E,h ⊢ e : Boolean by fact
  show ?case
  proof cases
    assume final e
    thus ?thesis
    proof (rule finalE)
      fix v assume val: e = Val v
      then obtain b where v: v = Bool b using wt by auto
      show ?thesis
      proof (cases b)
        case True with val v show ?thesis by(fastforce intro:RedCondT)
      next
        case False with val v show ?thesis by(fastforce intro:RedCondF)
      qed
    next
      fix a assume e = Throw a
      thus ?thesis by(fast intro:CondThrow)
    qed
  next
    assume ¬ final e with WTrtCond show ?thesis
      by simp (fast intro:CondRed)
  qed
next
  case WTrtWhile show ?case by(fast intro:RedWhile)
next
  case (WTrtThrow E e T T' l)
  show ?case
  proof cases
    assume final e — Then e must be throw or null
    thus ?thesis
    proof(induct rule: finalE)
      case (Val v)
      with ‹P ⊢ T ≤ Class Throwable› ‹¬ final (throw e)› ‹P,E,h ⊢ e : T›
      have v = Null by(cases v)(auto simp add: final-iff widen-Class)
      thus ?thesis using Val by(fastforce intro: RedThrowNull)
    next
      case (Throw a)
      thus ?thesis by(fastforce intro: ThrowThrow)
    qed
  next
    assume ¬ final e — Then e must reduce
    with WTrtThrow show ?thesis by simp (blast intro:ThrowRed)
  qed
next
  case (WTrtTry E e1 T1 V C e2 T2 l)
  have wt1: P,E,h ⊢ e1 : T1 by fact
```

```
  show ?case
  proof cases
    assume final e1
    thus ?thesis
    proof (rule finalE)
      fix v assume e1 = Val v
      thus ?thesis by(fast intro:RedTry)
    next
      fix a
      assume e1-Throw: e1 = Throw a
      with wt1 obtain D where ha: typeof-addr h a = ⌊Class-type D⌋
        by(auto simp add: widen-Class)
      thus ?thesis using e1-Throw
        by(cases P ⊢ D ≼* C)(fastforce intro:RedTryCatch RedTryFail)+
    qed
  next
    assume ¬ final e1
    with WTrtTry show ?thesis by simp (fast intro:TryRed)
  qed
next
  case WTrtNil thus ?case by simp
next
  case (WTrtCons E e T es Ts)
  have IHe: ⋀l. ⟦𝒟 e ⌊dom l⌋; ¬ final e⟧
              ⟹ ∃ e' s' ta. extTA,P,t ⊢ ⟨e,(h,l)⟩ −ta→ ⟨e',s'⟩
   and IHes: ⋀l. ⟦𝒟s es ⌊dom l⌋; ¬ finals es⟧
            ⟹ ∃ es' s' ta. extTA,P,t ⊢ ⟨es,(h,l)⟩ [−ta→] ⟨es',s'⟩
   and D: 𝒟s (e#es) ⌊dom l⌋ and not-fins: ¬ finals(e # es) by fact+
  have De: 𝒟 e ⌊dom l⌋ and Des: 𝒟s es (⌊dom l⌋ ⊔ 𝒜 e)
    using D by auto
  show ?case
  proof cases
    assume final e
    thus ?thesis
    proof (rule finalE)
      fix v assume e: e = Val v
      hence Des': 𝒟s es ⌊dom l⌋ using De Des by auto
      have not-fins-tl: ¬ finals es using not-fins e by simp
      show ?thesis using e IHes[OF Des' not-fins-tl]
        by (blast intro!:ListRed2)
    next
      fix a assume e = Throw a
      hence False using not-fins by simp
      thus ?thesis ..
    qed
  next
    assume ¬ final e
    with IHe[OF De] show ?thesis by(fast intro!:ListRed1)
  qed
qed

end
```

4.12 Preservation of definite assignment

theory *DefAssPreservation*
imports
DefAss
JWellForm
SmallStep
begin

Preservation of definite assignment more complex and requires a few lemmas first.

lemma *D-extRetJ* [*intro*!]: $\mathcal{D}\ e\ A \Longrightarrow \mathcal{D}\ (extRet2J\ e\ va)\ A$
by(*cases va*) *simp-all*

lemma *blocks-defass* [*iff*]: $\bigwedge A.\ [\![\ length\ Vs = length\ Ts;\ length\ vs = length\ Ts]\!] \Longrightarrow$
$\mathcal{D}\ (blocks\ Vs\ Ts\ vs\ e)\ A = \mathcal{D}\ e\ (A \sqcup \lfloor set\ Vs \rfloor)$
context *J-heap-base* **begin**

lemma *red-lA-incr*: $extTA,P,t \vdash \langle e,s\rangle\ -ta\rightarrow\ \langle e',s'\rangle \Longrightarrow \lfloor dom\ (lcl\ s)\rfloor \sqcup \mathcal{A}\ e \sqsubseteq \lfloor dom\ (lcl\ s')\rfloor \sqcup \mathcal{A}\ e'$
and *reds-lA-incr*: $extTA,P,t \vdash \langle es,s\rangle\ [-ta\rightarrow]\ \langle es',s'\rangle \Longrightarrow \lfloor dom\ (lcl\ s)\rfloor \sqcup \mathcal{A}s\ es \sqsubseteq \lfloor dom\ (lcl\ s')\rfloor \sqcup \mathcal{A}s\ es'$
apply(*induct rule*:*red-reds.inducts*)
apply(*simp-all del*:*fun-upd-apply add*:*hyperset-defs*)
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply(*force split*: *if-split-asm*)
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply(*blast dest*: *red-lcl-incr*)
apply(*blast dest*: *red-lcl-incr*)
by *blast*+

end

Now preservation of definite assignment.

declare *hyperUn-comm* [*simp del*]
declare *hyperUn-leftComm* [*simp del*]

context *J-heap-base* **begin**

lemma assumes *wf*: *wf-J-prog P*
shows *red-preserves-defass*: *extTA,P,t* ⊢ ⟨*e,s*⟩ −*ta*→ ⟨*e′,s′*⟩ ⟹ $\mathcal{D}$ *e* ⌊*dom* (*lcl s*)⌋ ⟹ $\mathcal{D}$ *e′* ⌊*dom* (*lcl s′*)⌋
and *reds-preserves-defass*: *extTA,P,t* ⊢ ⟨*es,s*⟩ [−*ta*→] ⟨*es′,s′*⟩ ⟹ $\mathcal{D}s$ *es* ⌊*dom* (*lcl s*)⌋ ⟹ $\mathcal{D}s$ *es′* ⌊*dom* (*lcl s′*)⌋
proof (*induction rule*:*red-reds.inducts*)
case *BinOpRed1* **thus** *?case* **by** (*auto elim*!: *D-mono*[*OF red-lA-incr*])
next
case *AAccRed1* **thus** *?case* **by** (*auto elim*!: *D-mono*[*OF red-lA-incr*])
next
case *AAssRed1* **thus** *?case* **by**(*auto intro*: *red-lA-incr sqUn-lem D-mono*)
next
case *AAssRed2* **thus** *?case* **by** (*auto elim*!: *D-mono*[*OF red-lA-incr*])
next
case *FAssRed1* **thus** *?case* **by** (*auto elim*!: *D-mono*[*OF red-lA-incr*])
next
case *CASRed1* **thus** *?case* **by**(*auto intro*: *red-lA-incr sqUn-lem D-mono*)
next
case *CASRed2* **thus** *?case* **by** (*auto elim*!: *D-mono*[*OF red-lA-incr*])
next
case *CallObj* **thus** *?case* **by** (*auto elim*!: *Ds-mono*[*OF red-lA-incr*])
next
case *CallParams* **thus** *?case* **by**(*auto elim*!: *Ds-mono*[*OF red-lA-incr*])
next
case *RedCall* **thus** *?case* **by**(*auto dest*!:*sees-wf-mdecl*[*OF wf*] *simp*:*wf-mdecl-def elim*!:*D-mono′*)
next
case *BlockRed* **thus** *?case*
by(*auto simp*:*hyperset-defs elim*!:*D-mono′ simp del*:*fun-upd-apply split*: *if-split-asm*)
next
case *SynchronizedRed1* **thus** *?case* **by**(*auto elim*!: *D-mono*[*OF red-lA-incr*])
next
case *SeqRed* **thus** *?case* **by** (*auto elim*!: *D-mono*[*OF red-lA-incr*])
next
case *CondRed* **thus** *?case* **by** (*auto elim*!: *D-mono*[*OF red-lA-incr*])
next
case *TryRed* **thus** *?case*
by (*fastforce dest*:*red-lcl-incr intro*:*D-mono′ simp*:*hyperset-defs*)
next
case *RedWhile* **thus** *?case* **by**(*auto simp*:*hyperset-defs elim*!:*D-mono′*)
next
case *ListRed1* **thus** *?case* **by** (*auto elim*!: *Ds-mono*[*OF red-lA-incr*])
qed (*auto simp*:*hyperset-defs*)

end

end

4.13 Type Safety Proof

theory *TypeSafe*
imports
Progress

DefAssPreservation
begin

4.13.1 Basic preservation lemmas

First two easy preservation lemmas.

theorem (**in** *J-conf-read*)
shows *red-preserves-hconf*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e*′,*s*′⟩; *P*,*E*,*hp s* ⊢ *e* : *T*; *hconf* (*hp s*) ⟧ ⟹ *hconf* (*hp s*′)
and *reds-preserves-hconf*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*es*,*s*⟩ [−*ta*→] ⟨*es*′,*s*′⟩; *P*,*E*,*hp s* ⊢ *es* [:] *Ts*; *hconf* (*hp s*) ⟧ ⟹ *hconf* (*hp s*′)
proof (*induct arbitrary*: *T E* **and** *Ts E rule*: *red-reds.inducts*)
case *RedNew* **thus** *?case*
by(*auto intro*: *hconf-heap-ops-mono*)
next
case *RedNewFail* **thus** *?case*
by(*auto intro*: *hconf-heap-ops-mono*)
next
case *RedNewArray* **thus** *?case*
by(*auto intro*: *hconf-heap-ops-mono*)
next
case *RedNewArrayFail* **thus** *?case*
by(*auto intro*: *hconf-heap-ops-mono*)
next
case (*RedAAss h a U n i v U*′ *h*′ *l*)
from ‹*sint i* < *int n*› ‹*0* <=*s i*›
have *nat* (*sint i*) < *n*
by (*simp add*: *word-sle-eq nat-less-iff*)
thus *?case* **using** *RedAAss*
by(*fastforce elim*: *hconf-heap-write-mono intro*: *addr-loc-type.intros simp add*: *conf-def*)
next
case *RedFAss* **thus** *?case*
by(*fastforce elim*: *hconf-heap-write-mono intro*: *addr-loc-type.intros simp add*: *conf-def*)
next
case *RedCASSucceed* **thus** *?case*
by(*fastforce elim*: *hconf-heap-write-mono intro*: *addr-loc-type.intros simp add*: *conf-def*)
next
case (*RedCallExternal s a U M Ts T*′ *D vs ta va h*′ *ta*′ *e*′ *s*′)
hence *P*,*hp s* ⊢ *a*·*M*(*vs*) : *T*
by(*fastforce simp add*: *external-WT*′*-iff dest*: *sees-method-fun*)
with *RedCallExternal* **show** *?case* **by**(*auto dest*: *external-call-hconf*)
qed *auto*

theorem (**in** *J-heap*) *red-preserves-lconf*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e*′,*s*′⟩; *P*,*E*,*hp s* ⊢ *e*:*T*; *P*,*hp s* ⊢ *lcl s* (:≤) *E* ⟧ ⟹ *P*,*hp s*′ ⊢ *lcl s*′ (:≤) *E*
and *reds-preserves-lconf*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*es*,*s*⟩ [−*ta*→] ⟨*es*′,*s*′⟩; *P*,*E*,*hp s* ⊢ *es*[:]*Ts*; *P*,*hp s* ⊢ *lcl s* (:≤) *E* ⟧ ⟹ *P*,*hp s*′ ⊢ *lcl s*′
(:≤) *E*
proof(*induct arbitrary*: *T E* **and** *Ts E rule*:*red-reds.inducts*)
case *RedNew* **thus** *?case*
by(*fastforce intro*:*lconf-hext hext-heap-ops simp del*: *fun-upd-apply*)
next
case *RedNewFail* **thus** *?case*
by(*auto intro*:*lconf-hext hext-heap-ops simp del*: *fun-upd-apply*)

next
case *RedNewArray* **thus** *?case*
by(*fastforce intro*:*lconf-hext hext-heap-ops simp del*: *fun-upd-apply*)
next
case *RedNewArrayFail* **thus** *?case*
by(*fastforce intro*:*lconf-hext hext-heap-ops simp del*: *fun-upd-apply*)
next
case *RedLAss* **thus** *?case*
by(*fastforce elim*: *lconf-upd simp add*: *conf-def simp del*: *fun-upd-apply*)
next
case *RedAAss* **thus** *?case*
by(*fastforce intro*:*lconf-hext hext-heap-ops simp del*: *fun-upd-apply*)
next
case *RedFAss* **thus** *?case*
by(*fastforce intro*:*lconf-hext hext-heap-ops simp del*: *fun-upd-apply*)
next
case *RedCASSucceed* **thus** *?case*
by(*fastforce intro*:*lconf-hext hext-heap-ops simp del*: *fun-upd-apply*)
next
case (*BlockRed e h x V vo ta e' h' x' T T' E*)
note *red* = ‹*extTA*,*P*,*t* ⊢ ⟨*e*,(*h*, *x*(*V* := *vo*))⟩ −*ta*→ ⟨*e'*,(*h'*, *x'*)⟩›
note *IH* = ‹⋀*T E*. ⟦*P*,*E*,*hp* (*h*, *x*(*V* := *vo*)) ⊢ *e* : *T*;
P,*hp* (*h*, *x*(*V* := *vo*)) ⊢ *lcl* (*h*, *x*(*V* := *vo*)) (:≤) *E*⟧
⟹ *P*,*hp* (*h'*, *x'*) ⊢ *lcl* (*h'*, *x'*) (:≤) *E*›
note *wt* = ‹*P*,*E*,*hp* (*h*, *x*) ⊢ {*V*:*T*=*vo*; *e*} : *T'*›
note *lconf* = ‹*P*,*hp* (*h*, *x*) ⊢ *lcl* (*h*, *x*) (:≤) *E*›
from *lconf-hext*[*OF lconf*[*simplified*] *red-hext-incr*[*OF red*, *simplified*]]
have *P*,*h'* ⊢ *x* (:≤) *E* **.**
moreover from *wt* **have** *P*,*E*(*V*↦*T*),*h* ⊢ *e* : *T'* **by**(*cases vo*, *auto*)
moreover from *lconf wt* **have** *P*,*h* ⊢ *x*(*V* := *vo*) (:≤) *E*(*V* ↦ *T*)
by(*cases vo*)(*simp add*: *lconf-def*,*auto intro*: *lconf-upd2 simp add*: *conf-def*)
ultimately have *P*,*h'* ⊢ *x'* (:≤) *E*(*V*↦*T*)
by(*auto intro*: *IH*[*simplified*])
with ‹*P*,*h'* ⊢ *x* (:≤) *E*› **show** *?case*
by(*auto simp add*: *lconf-def split*: *if-split-asm*)
next
case (*RedCallExternal s a U M Ts T' D vs ta va h' ta' e' s'*)
from ‹*P*,*t* ⊢ ⟨*a*·*M*(*vs*),*hp s*⟩ −*ta*→*ext* ⟨*va*,*h'*⟩› **have** *hp s* ◁ *h'* **by**(*rule red-external-hext*)
with ‹*s'* = (*h'*, *lcl s*)› ‹*P*,*hp s* ⊢ *lcl s* (:≤) *E*› **show** *?case* **by**(*auto intro*: *lconf-hext*)
qed *auto*

Combining conformance of heap and local variables:

definition (**in** *J-heap-conf-base*) *sconf* :: *env* ⇒ ('*addr*, '*heap*) *Jstate* ⇒ *bool* (‹- ⊢ - √› [*51*,*51*]*50*)
where *E* ⊢ *s* √ ≡ *let* (*h*,*l*) = *s in hconf h* ∧ *P*,*h* ⊢ *l* (:≤) *E* ∧ *preallocated h*

context *J-conf-read* **begin**

lemma *red-preserves-sconf*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*tas*→ ⟨*e'*,*s'*⟩; *P*,*E*,*hp s* ⊢ *e* : *T*; *E* ⊢ *s* √ ⟧ ⟹ *E* ⊢ *s'* √
apply(*auto dest*: *red-preserves-hconf red-preserves-lconf simp add*:*sconf-def*)
apply(*fastforce dest*: *red-hext-incr intro*: *preallocated-hext*)
done

lemma *reds-preserves-sconf*:

⟦ *extTA*,*P*,*t* ⊢ ⟨*es*,*s*⟩ [−*ta*→] ⟨*es*′,*s*′⟩; *P*,*E*,*hp s* ⊢ *es* [:] *Ts*; *E* ⊢ *s* √ ⟧ ⟹ *E* ⊢ *s*′ √
apply(*auto dest*: *reds-preserves-hconf reds-preserves-lconf simp add*: *sconf-def*)
apply(*fastforce dest*: *reds-hext-incr intro*: *preallocated-hext*)
done

end

lemma (**in** *J-heap-base*) *wt-external-call*:
⟦ *conf-extRet P h va T*; *P*,*E*,*h* ⊢ *e* : *T* ⟧ ⟹ ∃ *T*′. *P*,*E*,*h* ⊢ *extRet2J e va* : *T*′ ∧ *P* ⊢ *T*′ ≤ *T*
by(*cases va*)(*auto simp add*: *conf-def*)

4.13.2 Subject reduction

theorem (**in** *J-conf-read*) **assumes** *wf*: *wf-J-prog P*
shows *subject-reduction*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e*′,*s*′⟩; *E* ⊢ *s* √; *P*,*E*,*hp s* ⊢ *e*:*T*; *P*,*hp s* ⊢ *t* √*t* ⟧
⟹ ∃ *T*′. *P*,*E*,*hp s*′ ⊢ *e*′:*T*′ ∧ *P* ⊢ *T*′ ≤ *T*
and *subjects-reduction*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*es*,*s*⟩ [−*ta*→] ⟨*es*′,*s*′⟩; *E* ⊢ *s* √; *P*,*E*,*hp s* ⊢ *es*[:]*Ts*; *P*,*hp s* ⊢ *t* √*t* ⟧
⟹ ∃ *Ts*′. *P*,*E*,*hp s*′ ⊢ *es*′[:]*Ts*′ ∧ *P* ⊢ *Ts*′ [≤] *Ts*
proof (*induct arbitrary*: *T E* **and** *Ts E rule*:*red-reds.inducts*)
case *RedNew*
thus *?case* **by**(*auto dest*: *allocate-SomeD*)
next
case *RedNewFail* **thus** *?case* **unfolding** *sconf-def*
by(*fastforce intro*:*typeof-OutOfMemory preallocated-heap-ops simp add*: *xcpt-subcls-Throwable*[*OF - wf*])
next
case *NewArrayRed*
thus *?case* **by** *fastforce*
next
case *RedNewArray*
thus *?case* **by**(*auto dest*: *allocate-SomeD*)
next
case *RedNewArrayNegative* **thus** *?case* **unfolding** *sconf-def*
by(*fastforce intro*: *preallocated-heap-ops simp add*: *xcpt-subcls-Throwable*[*OF - wf*])
next
case *RedNewArrayFail* **thus** *?case* **unfolding** *sconf-def*
by(*fastforce intro*:*typeof-OutOfMemory preallocated-heap-ops simp add*: *xcpt-subcls-Throwable*[*OF - wf*])
next
case (*CastRed e s ta e*′ *s*′ *C T E*)
have *esse*: *extTA*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e*′,*s*′⟩
and *IH*: ⋀*T E*. ⟦*E* ⊢ *s* √; *P*,*E*,*hp s* ⊢ *e* : *T*; *P*,*hp s* ⊢ *t* √*t*⟧ ⟹ ∃ *T*′. *P*,*E*,*hp s*′ ⊢ *e*′ : *T*′ ∧ *P* ⊢ *T*′ ≤ *T*
and *hconf*: *E* ⊢ *s* √
and *wtc*: *P*,*E*,*hp s* ⊢ *Cast C e* : *T* **by** *fact*+
thus *?case*
proof(*clarsimp*)
fix *T*′
assume *wte*: *P*,*E*,*hp s* ⊢ *e* : *T*′ *is-type P C*
from *wte* **and** *hconf* **and** *IH* **and** ‹*P*,*hp s* ⊢ *t* √*t*› **have** ∃ *U*. *P*,*E*,*hp s*′ ⊢ *e*′ : *U* ∧ *P* ⊢ *U* ≤ *T*′ **by** *simp*
then obtain *U* **where** *wtee*: *P*,*E*,*hp s*′ ⊢ *e*′ : *U* **and** *UsTT*: *P* ⊢ *U* ≤ *T*′ **by** *blast*

from *wtee* ‹*is-type P C*› **have** $P,E,hp\ s' \vdash Cast\ C\ e' : C$ **by**(*rule WTrtCast*)
thus $\exists\, T'.\ P,E,hp\ s' \vdash Cast\ C\ e' : T' \wedge P \vdash T' \leq C$ **by** *blast*
qed
next
case *RedCast* **thus** *?case*
by(*clarsimp simp add*: *is-refT-def*)
next
case *RedCastFail* **thus** *?case* **unfolding** *sconf-def*
by(*fastforce simp add*: *xcpt-subcls-Throwable*[*OF - wf*])
next
case (*InstanceOfRed e s ta e' s' U T E*)
have *IH*: $\bigwedge T\ E.\ [\![E \vdash s\ \surd;\ P,E,hp\ s \vdash e : T;\ P,hp\ s \vdash t\ \surd t]\!] \Longrightarrow \exists\, T'.\ P,E,hp\ s' \vdash e' : T' \wedge P \vdash T' \leq T$
and *hconf*: $E \vdash s\ \surd$
and *wtc*: $P,E,hp\ s \vdash e\ instanceof\ U : T$
and *tconf*: $P,hp\ s \vdash t\ \surd t$ **by** *fact+*
from *wtc* **obtain** T' **where** $P,E,hp\ s \vdash e : T'$ **by** *auto*
from *IH*[*OF hconf this tconf*] **obtain** T'' **where** $P,E,hp\ s' \vdash e' : T''$ **by** *auto*
with *wtc* **show** *?case* **by** *auto*
next
case *RedInstanceOf* **thus** *?case*
by(*clarsimp*)
next
case (*BinOpRed1 e₁ s ta e₁' s' bop e₂ T E*)
have *red*: $extTA,P,t \vdash \langle e_1,\ s\rangle\ -ta\rightarrow\ \langle e_1',\ s'\rangle$
and *IH*: $\bigwedge T\ E.\ [\![E \vdash s\ \surd;\ P,E,hp\ s \vdash e_1 : T;\ P,hp\ s \vdash t\ \surd t]\!]$
$\Longrightarrow \exists\, U.\ P,E,hp\ s' \vdash e_1' : U \wedge P \vdash U \leq T$
and *conf*: $E \vdash s\ \surd$ **and** *wt*: $P,E,hp\ s \vdash e_1\ «bop»\ e_2 : T$
and *tconf*: $P,hp\ s \vdash t\ \surd t$ **by** *fact+*
from *wt* **obtain** *T1 T2* **where** *wt1*: $P,E,hp\ s \vdash e_1 : T1$
and *wt2*: $P,E,hp\ s \vdash e_2 : T2$ **and** *wtbop*: $P \vdash T1 «bop» T2 : T$ **by** *auto*
from *IH*[*OF conf wt1 tconf*] **obtain** $T1'$ **where** *wt1'*: $P,E,hp\ s' \vdash e_1' : T1'$
and *sub*: $P \vdash T1' \leq T1$ **by** *blast*
from *WTrt-binop-widen-mono*[*OF wtbop sub widen-refl*]
obtain T' **where** *wtbop'*: $P \vdash T1' «bop» T2 : T'$ **and** *sub'*: $P \vdash T' \leq T$ **by** *blast*
from *wt1' WTrt-hext-mono*[*OF wt2 red-hext-incr*[*OF red*]] *wtbop'*
have $P,E,hp\ s' \vdash e_1' «bop» e_2 : T'$ **by**(*rule WTrtBinOp*)
with *sub'* **show** *?case* **by** *blast*
next
case (*BinOpRed2 e₂ s ta e₂' s' v₁ bop T E*)
have *red*: $extTA,P,t \vdash \langle e_2,s\rangle\ -ta\rightarrow\ \langle e_2',s'\rangle$ **by** *fact*
have *IH*: $\bigwedge E\ T.\ [\![E \vdash s\ \surd;\ P,E,hp\ s \vdash e_2 : T;\ P,hp\ s \vdash t\ \surd t]\!]$
$\Longrightarrow \exists\, U.\ P,E,hp\ s' \vdash e_2' : U \wedge P \vdash U \leq T$
and *tconf*: $P,hp\ s \vdash t\ \surd t$ **by** *fact+*
have *conf*: $E \vdash s\ \surd$ **and** *wt*: $P,E,hp\ s \vdash (Val\ v_1)\ «bop»\ e_2 : T$ **by** *fact+*
from *wt* **obtain** *T1 T2* **where** *wt1*: $P,E,hp\ s \vdash Val\ v_1 : T1$
and *wt2*: $P,E,hp\ s \vdash e_2 : T2$ **and** *wtbop*: $P \vdash T1 «bop» T2 : T$ **by** *auto*
from *IH*[*OF conf wt2 tconf*] **obtain** $T2'$ **where** *wt2'*: $P,E,hp\ s' \vdash e_2' : T2'$
and *sub*: $P \vdash T2' \leq T2$ **by** *blast*
from *WTrt-binop-widen-mono*[*OF wtbop widen-refl sub*]
obtain T' **where** *wtbop'*: $P \vdash T1 «bop» T2' : T'$ **and** *sub'*: $P \vdash T' \leq T$ **by** *blast*
from *WTrt-hext-mono*[*OF wt1 red-hext-incr*[*OF red*]] *wt2' wtbop'*
have $P,E,hp\ s' \vdash Val\ v_1\ «bop»\ e_2' : T'$ **by**(*rule WTrtBinOp*)
with *sub'* **show** *?case* **by** *blast*

next
case (*RedBinOp bop v1 v2 v s*)
from ‹$E \vdash s\ \surd$› **have** *preh*: *preallocated* (*hp s*) **by**(*cases s*)(*simp add*: *sconf-def*)
from ‹$P,E,hp\ s \vdash Val\ v1\ «bop»\ Val\ v2 : T$› **obtain** *T1 T2*
where $typeof_{hp\ s}\ v1 = \lfloor T1 \rfloor\ typeof_{hp\ s}\ v2 = \lfloor T2 \rfloor\ P \vdash T1 «bop» T2 : T$ **by** *auto*
with *wf preh* **have** $P,hp\ s \vdash v :\leq T$ **using** ‹$binop\ bop\ v1\ v2 = \lfloor Inl\ v \rfloor$›
by(*rule binop-type*)
thus *?case* **by**(*auto simp add*: *conf-def*)
next
case (*RedBinOpFail bop v1 v2 a s*)
from ‹$E \vdash s\ \surd$› **have** *preh*: *preallocated* (*hp s*) **by**(*cases s*)(*simp add*: *sconf-def*)
from ‹$P,E,hp\ s \vdash Val\ v1\ «bop»\ Val\ v2 : T$› **obtain** *T1 T2*
where $typeof_{hp\ s}\ v1 = \lfloor T1 \rfloor\ typeof_{hp\ s}\ v2 = \lfloor T2 \rfloor\ P \vdash T1 «bop» T2 : T$ **by** *auto*
with *wf preh* **have** $P,hp\ s \vdash Addr\ a :\leq Class\ Throwable$ **using** ‹$binop\ bop\ v1\ v2 = \lfloor Inr\ a \rfloor$›
by(*rule binop-type*)
thus *?case* **by**(*auto simp add*: *conf-def*)
next
case *RedVar* **thus** *?case* **by** (*fastforce simp*:*sconf-def lconf-def conf-def*)
next
case *LAssRed* **thus** *?case* **by**(*blast intro*:*widen-trans*)
next
case *RedLAss* **thus** *?case* **by** *fastforce*
next
case (*AAccRed1 a s ta a' s' i T E*)
have *IH*: $\bigwedge E\ T.\ [\![E \vdash s\ \surd;\ P,E,hp\ s \vdash a : T;\ P,hp\ s \vdash t\ \surd t]\!] \Longrightarrow \exists T'.\ P,E,hp\ s' \vdash a' : T' \wedge P \vdash T' \leq T$
and *assa*: $extTA,P,t \vdash \langle a,s \rangle\ -ta\rightarrow\ \langle a',s' \rangle$
and *wt*: $P,E,hp\ s \vdash a\lfloor i \rceil : T$
and *hconf*: $E \vdash s\ \surd$
and *tconf*: $P,hp\ s \vdash t\ \surd t$ **by** *fact+*
from *wt* **have** *wti*: $P,E,hp\ s \vdash i : Integer$ **by** *auto*
from *wti red-hext-incr*[*OF assa*] **have** *wti'*: $P,E,hp\ s' \vdash i : Integer$ **by** $-$ (*rule WTrt-hext-mono*)
{ assume *wta*: $P,E,hp\ s \vdash a : T\lfloor \rceil$
from *IH*[*OF hconf wta tconf*]
obtain *U* **where** *wta'*: $P,E,hp\ s' \vdash a' : U$ **and** *UsubT*: $P \vdash U \leq T\lfloor \rceil$ **by** *fastforce*
with *wta' wti'* **have** *?case* **by**(*cases U*, *auto simp add*: *widen-Array*) **}**
moreover
{ assume *wta*: $P,E,hp\ s \vdash a : NT$
from *IH*[*OF hconf wta tconf*] **have** $P,E,hp\ s' \vdash a' : NT$ **by** *fastforce*
from *this wti'* **have** *?case*
by(*fastforce intro*:*WTrtAAccNT*) **}**
ultimately show *?case* **using** *wt* **by** *auto*
next
case (*AAccRed2 i s ta i' s' a T E*)
have *IH*: $\bigwedge E\ T.\ [\![E \vdash s\ \surd;\ P,E,hp\ s \vdash i : T;\ P,hp\ s \vdash t\ \surd t]\!] \Longrightarrow \exists T'.\ P,E,hp\ s' \vdash i' : T' \wedge P \vdash T' \leq T$
and *issi*: $extTA,P,t \vdash \langle i,s \rangle\ -ta\rightarrow\ \langle i',s' \rangle$
and *wt*: $P,E,hp\ s \vdash Val\ a\lfloor i \rceil : T$
and *sconf*: $E \vdash s\ \surd$
and *tconf*: $P,hp\ s \vdash t\ \surd t$ **by** *fact+*
from *wt* **have** *wti*: $P,E,hp\ s \vdash i : Integer$ **by** *auto*
from *wti IH sconf tconf* **have** *wti'*: $P,E,hp\ s' \vdash i' : Integer$ **by** *blast*
from *wt* **show** *?case*
proof (*rule WTrt-elim-cases*)

```
    assume wta: P,E,hp s ⊢ Val a : T⌊⌋
    from wta red-hext-incr[OF issi] have wta': P,E,hp s' ⊢ Val a : T⌊⌋ by (rule WTrt-hext-mono)
    from wta' wti' show ?case by(fastforce)
  next
    assume wta: P,E,hp s ⊢ Val a : NT
    from wta red-hext-incr[OF issi] have wta': P,E,hp s' ⊢ Val a : NT by (rule WTrt-hext-mono)
    from wta' wti' show ?case
      by(fastforce elim:WTrtAAccNT)
  qed
next
  case RedAAccNull thus ?case unfolding sconf-def
    by(fastforce simp add: xcpt-subcls-Throwable[OF - wf])
next
  case RedAAccBounds thus ?case unfolding sconf-def
    by(fastforce simp add: xcpt-subcls-Throwable[OF - wf])
next
  case (RedAAcc h a T n i v l T' E)
  from ‹E ⊢ (h, l) √› have hconf h by(clarsimp simp add: sconf-def)
  from ‹0 <=s i› ‹sint i < int n›
  have nat (sint i) < n
    by (simp add: word-sle-eq nat-less-iff)
  with ‹typeof-addr h a = ⌊Array-type T n⌋› have P,h ⊢ a@ACell (nat (sint i)) : T
    by(auto intro: addr-loc-type.intros)
  from heap-read-conf[OF ‹heap-read h a (ACell (nat (sint i))) v› this] ‹hconf h›
  have P,h ⊢ v :≤ T by simp
  thus ?case using RedAAcc by(auto simp add: conf-def)
next
  case (AAssRed1 a s ta a' s' i e T E)
  have IH: ⋀E T. ⟦E ⊢ s √; P,E,hp s ⊢ a : T; P,hp s ⊢ t √t⟧ ⟹ ∃T'. P,E,hp s' ⊢ a' : T' ∧ P ⊢
T' ≤ T
    and assa: extTA,P,t ⊢ ⟨a,s⟩ −ta→ ⟨a',s'⟩
    and wt: P,E,hp s ⊢ a⌊i⌋ := e : T
    and sconf: E ⊢ s √
    and tconf: P,hp s ⊢ t √t by fact+
  from wt have void: T = Void by blast
  from wt have wti: P,E,hp s ⊢ i : Integer by auto
  from wti red-hext-incr[OF assa] have wti': P,E,hp s' ⊢ i : Integer by − (rule WTrt-hext-mono)
  { assume wta: P,E,hp s ⊢ a : NT
    from IH[OF sconf wta tconf] have wta': P,E,hp s' ⊢ a' : NT by fastforce
    from wt wta obtain V where wte: P,E,hp s ⊢ e : V by(auto)
    from wte red-hext-incr[OF assa] have wte': P,E,hp s' ⊢ e : V by − (rule WTrt-hext-mono)
    from wta' wti' wte' void have ?case
      by(fastforce elim: WTrtAAssNT) }
  moreover
  { fix U
    assume wta: P,E,hp s ⊢ a : U⌊⌋
    from IH[OF sconf wta tconf]
    obtain U' where wta': P,E,hp s' ⊢ a' : U' and UsubT: P ⊢ U' ≤ U⌊⌋ by fastforce
    with wta' have ?case
    proof(cases U')
      case NT
      assume UNT: U' = NT
      from UNT wt wta obtain V where wte: P,E,hp s ⊢ e : V by(auto)
      from wte red-hext-incr[OF assa] have wte': P,E,hp s' ⊢ e : V by − (rule WTrt-hext-mono)
```

```
    from wta' UNT wti' wte' void show ?thesis
      by(fastforce elim: WTrtAAssNT)
  next
    case (Array A)
    have UA: U' = A⌊⌉ by fact
    with UA UsubT wt wta obtain V where wte: P,E,hp s ⊢ e : V by auto
    from wte red-hext-incr[OF assa] have wte': P,E,hp s' ⊢ e : V by − (rule WTrt-hext-mono)
    with wta' wte' UA wti' void show ?thesis by (fast elim:WTrtAAss)
  qed(simp-all add: widen-Array) }
 ultimately show ?case using wt by blast
next
 case (AAssRed2 i s ta i' s' a e T E)
 have IH: ⋀E T. ⟦E ⊢ s √; P,E,hp s ⊢ i : T; P,hp s ⊢ t √t ⟧ ⟹ ∃ T'. P,E,hp s' ⊢ i' : T' ∧ P ⊢
T' ≤ T
   and issi: extTA,P,t ⊢ ⟨i,s⟩ −ta→ ⟨i',s'⟩
   and wt: P,E,hp s ⊢ Val a⌊i⌉ := e : T
   and sconf: E ⊢ s √ and tconf: P,hp s ⊢ t √t by fact+
 from wt have void: T = Void by blast
 from wt have wti: P,E,hp s ⊢ i : Integer by auto
 from IH[OF sconf wti tconf] have wti': P,E,hp s' ⊢ i' : Integer by fastforce
 from wt show ?case
 proof(rule WTrt-elim-cases)
   fix U T'
   assume wta: P,E,hp s ⊢ Val a : U⌊⌉
     and wte: P,E,hp s ⊢ e : T'
   from wte red-hext-incr[OF issi] have wte': P,E,hp s' ⊢ e : T' by − (rule WTrt-hext-mono)
   from wta red-hext-incr[OF issi] have wta': P,E,hp s' ⊢ Val a : U⌊⌉ by − (rule WTrt-hext-mono)
   from wta' wti' wte' void show ?case by (fastforce elim:WTrtAAss)
 next
   fix T'
   assume wta: P,E,hp s ⊢ Val a : NT
     and wte: P,E,hp s ⊢ e : T'
   from wte red-hext-incr[OF issi] have wte': P,E,hp s' ⊢ e : T' by − (rule WTrt-hext-mono)
   from wta red-hext-incr[OF issi] have wta': P,E,hp s' ⊢ Val a : NT by − (rule WTrt-hext-mono)
   from wta' wti' wte' void show ?case by (fastforce elim:WTrtAAss)
 qed
next
 case (AAssRed3 e s ta e' s' a i T E)
 have IH: ⋀E T. ⟦E ⊢ s √; P,E,hp s ⊢ e : T; P,hp s ⊢ t √t⟧ ⟹ ∃ T'. P,E,hp s' ⊢ e' : T' ∧ P ⊢
T' ≤ T
   and issi: extTA,P,t ⊢ ⟨e,s⟩ −ta→ ⟨e',s'⟩
   and wt: P,E,hp s ⊢ Val a⌊Val i⌉ := e : T
   and sconf: E ⊢ s √ and tconf: P,hp s ⊢ t √t by fact+
 from wt have void: T = Void by blast
 from wt have wti: P,E,hp s ⊢ Val i : Integer by auto
 from wti red-hext-incr[OF issi] have wti': P,E,hp s' ⊢ Val i : Integer by − (rule WTrt-hext-mono)
 from wt show ?case
 proof(rule WTrt-elim-cases)
   fix U T'
   assume wta: P,E,hp s ⊢ Val a : U⌊⌉
     and wte: P,E,hp s ⊢ e : T'
   from wta red-hext-incr[OF issi] have wta': P,E,hp s' ⊢ Val a : U⌊⌉ by − (rule WTrt-hext-mono)
   from IH[OF sconf wte tconf]
   obtain V where wte': P,E,hp s' ⊢ e' : V by fastforce
```

```
    from wta' wti' wte' void show ?case by (fastforce elim:WTrtAAss)
  next
    fix T'
    assume wta: P,E,hp s ⊢ Val a : NT
      and wte: P,E,hp s ⊢ e : T'
    from wta red-hext-incr[OF issi] have wta': P,E,hp s' ⊢ Val a : NT by − (rule WTrt-hext-mono)
    from IH[OF sconf wte tconf]
    obtain V where wte': P,E,hp s' ⊢ e' : V by fastforce
    from wta' wti' wte' void show ?case by (fastforce elim:WTrtAAss)
  qed
next
  case RedAAssNull thus ?case unfolding sconf-def
    by(fastforce simp add: xcpt-subcls-Throwable[OF - wf])
next
  case RedAAssBounds thus ?case unfolding sconf-def
    by(fastforce simp add: xcpt-subcls-Throwable[OF - wf])
next
  case RedAAssStore thus ?case unfolding sconf-def
    by(fastforce simp add: xcpt-subcls-Throwable[OF - wf])
next
  case RedAAss thus ?case
    by(auto simp del:fun-upd-apply)
next
  case (ALengthRed a s ta a' s' T E)
  note IH = ‹⋀T'. ⟦E ⊢ s √; P,E,hp s ⊢ a : T'; P,hp s ⊢ t √t⟧
     ⟹ ∃T''. P,E,hp s' ⊢ a' : T'' ∧ P ⊢ T'' ≤ T'›
  from ‹P,E,hp s ⊢ a•length : T›
  show ?case
  proof(rule WTrt-elim-cases)
    fix T'
    assume [simp]: T = Integer
      and wta: P,E,hp s ⊢ a : T'⌊⌉
    from wta ‹E ⊢ s √› IH ‹P,hp s ⊢ t √t›
    obtain T'' where wta': P,E,hp s' ⊢ a' : T''
      and sub: P ⊢ T'' ≤ T'⌊⌉ by blast
    from sub have P,E,hp s' ⊢ a'•length : Integer
      unfolding widen-Array
    proof(rule disjE)
      assume T'' = NT
      with wta' show ?thesis by(auto)
    next
      assume ∃V. T'' = V⌊⌉ ∧ P ⊢ V ≤ T'
      then obtain V where T'' = V⌊⌉ P ⊢ V ≤ T' by blast
      with wta' show ?thesis by −(rule WTrtALength, simp)
    qed
    thus ?thesis by(simp)
  next
    assume P,E,hp s ⊢ a : NT
    with ‹E ⊢ s √› IH ‹P,hp s ⊢ t √t›
    obtain T'' where wta': P,E,hp s' ⊢ a' : T''
      and sub: P ⊢ T'' ≤ NT by blast
    from sub have T'' = NT by auto
    with wta' show ?thesis by(auto)
  qed
```

next
case (*RedALength h a T n l T′ E*)
from ‹*P*,*E*,*hp* (*h*, *l*) ⊢ *addr a*·*length* : *T′*› ‹*typeof-addr h a* = ⌊*Array-type T n*⌋›
have [*simp*]: *T′* = *Integer* **by**(*auto*)
thus *?case* **by**(*auto*)
next
case *RedALengthNull* **thus** *?case* **unfolding** *sconf-def*
by(*fastforce simp add*: *xcpt-subcls-Throwable*[*OF - wf*])
next
case (*FAccRed e s ta e′ s′ F D T E*)
have *IH*: ⋀*E T*. ⟦*E* ⊢ *s* √; *P*,*E*,*hp s* ⊢ *e* : *T*; *P*,*hp s* ⊢ *t* √*t*⟧
⟹ ∃ *U*. *P*,*E*,*hp s′* ⊢ *e′* : *U* ∧ *P* ⊢ *U* ≤ *T*
and *conf*: *E* ⊢ *s* √ **and** *wt*: *P*,*E*,*hp s* ⊢ *e*·*F*{*D*} : *T*
and *tconf*: *P*,*hp s* ⊢ *t* √*t* **by** *fact+*
— Now distinguish the two cases how wt can have arisen.
{ fix *T′ C fm*
assume *wte*: *P*,*E*,*hp s* ⊢ *e* : *T′*
and *icto*: *class-type-of′ T′* = ⌊*C*⌋
and *has*: *P* ⊢ *C has F*:*T* (*fm*) *in D*
from *IH*[*OF conf wte tconf*]
obtain *U* **where** *wte′*: *P*,*E*,*hp s′* ⊢ *e′* : *U* **and** *UsubC*: *P* ⊢ *U* ≤ *T′* **by** *auto*
— Now distinguish what *U* can be.
with *UsubC* **have** *?case*
proof(*cases U* = *NT*)
case *True*
thus *?thesis* **using** *wte′* **by**(*blast intro*:*WTrtFAccNT widen-refl*)
next
case *False*
with *icto UsubC* **obtain** *C′* **where** *icto′*: *class-type-of′ U* = ⌊*C′*⌋
and *C′subC*: *P* ⊢ *C′* ≼* *C*
by(*rule widen-is-class-type-of*)
from *has-field-mono*[*OF has C′subC*] *wte′ icto′*
show *?thesis* **by**(*auto intro*!:*WTrtFAcc*)
qed }
moreover
{ assume *P*,*E*,*hp s* ⊢ *e* : *NT*
hence *P*,*E*,*hp s′* ⊢ *e′* : *NT* **using** *IH*[*OF conf - tconf*] **by** *fastforce*
hence *?case* **by**(*fastforce intro*:*WTrtFAccNT widen-refl*) **}**
ultimately show *?case* **using** *wt* **by** *blast*
next
case *RedFAcc* **thus** *?case* **unfolding** *sconf-def*
by(*fastforce dest*: *heap-read-conf intro*: *addr-loc-type.intros simp add*: *conf-def*)
next
case *RedFAccNull* **thus** *?case* **unfolding** *sconf-def*
by(*fastforce simp add*: *xcpt-subcls-Throwable*[*OF - wf*])
next
case (*FAssRed1 e s ta e′ s′ F D e*$_2$)
have *red*: *extTA*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩
and *IH*: ⋀*E T*. ⟦*E* ⊢ *s* √; *P*,*E*,*hp s* ⊢ *e* : *T*; *P*,*hp s* ⊢ *t* √*t*⟧
⟹ ∃ *U*. *P*,*E*,*hp s′* ⊢ *e′* : *U* ∧ *P* ⊢ *U* ≤ *T*
and *conf*: *E* ⊢ *s* √ **and** *wt*: *P*,*E*,*hp s* ⊢ *e*·*F*{*D*}:=*e*$_2$: *T*
and *tconf*: *P*,*hp s* ⊢ *t* √*t* **by** *fact+*
from *wt* **have** *void*: *T* = *Void* **by** *blast*
— We distinguish if *e* has type *NT* or a Class type

{ assume $P,E,hp\ s \vdash e : NT$
hence $P,E,hp\ s' \vdash e' : NT$ **using** $IH[OF\ conf\ -\ tconf]$ **by** *fastforce*
moreover obtain T_2 **where** $P,E,hp\ s \vdash e_2 : T_2$ **using** wt **by** *auto*
from *this red-hext-incr*$[OF\ red]$ **have** $P,E,hp\ s' \vdash e_2 : T_2$
by(*rule WTrt-hext-mono*)
ultimately have *?case* **using** *void* **by**(*blast intro*!:*WTrtFAssNT*)
}
moreover
{ fix $T'\ C\ TF\ T_2\ fm$
assume wt_1: $P,E,hp\ s \vdash e : T'$ **and** *icto*: *class-type-of*$'\ T' = \lfloor C \rfloor$ **and** wt_2: $P,E,hp\ s \vdash e_2 : T_2$
and *has*: $P \vdash C\ has\ F{:}TF\ (fm)\ in\ D$ **and** *sub*: $P \vdash T_2 \leq TF$
obtain U **where** wt_1': $P,E,hp\ s' \vdash e' : U$ **and** *UsubC*: $P \vdash U \leq T'$
using $IH[OF\ conf\ wt_1\ tconf]$ **by** *blast*
have wt_2': $P,E,hp\ s' \vdash e_2 : T_2$
by(*rule WTrt-hext-mono*$[OF\ wt_2\ red\text{-}hext\text{-}incr[OF\ red]]$)
— Is U the null type or a class type?
have *?case*
proof(*cases* $U = NT$)
case *True*
with $wt_1'\ wt_2'$ *void* **show** *?thesis* **by**(*blast intro*!:*WTrtFAssNT*)
next
case *False*
with *icto UsubC* **obtain** C' **where** *icto'*: *class-type-of*$'\ U = \lfloor C' \rfloor$
and *subclass*: $P \vdash C' \preceq^* C$ **by**(*rule widen-is-class-type-of*)
have $P \vdash C'\ has\ F{:}TF\ (fm)\ in\ D$ **by**(*rule has-field-mono*$[OF\ has\ subclass]$)
with wt_1' **show** *?thesis* **using** wt_2' *sub void icto'* **by**(*blast intro*:*WTrtFAss*)
qed }
ultimately show *?case* **using** wt **by** *blast*
next
case (*FAssRed2* $e_2\ s\ ta\ e_2'\ s'\ v\ F\ D\ T\ E$)
have *red*: $extTA,P,t \vdash \langle e_2,s\rangle\ -ta\rightarrow\ \langle e_2',s'\rangle$
and *IH*: $\bigwedge E\ T.\ [\![E \vdash s\ \surd;\ P,E,hp\ s \vdash e_2 : T;\ P,hp\ s \vdash t\ \surd t]\!] \Longrightarrow \exists U.\ P,E,hp\ s' \vdash e_2' : U \wedge P \vdash U \leq T$
and *conf*: $E \vdash s\ \surd$ **and** *wt*: $P,E,hp\ s \vdash Val\ v{\cdot}F\{D\}{:=}e_2 : T$
and *tconf*: $P,hp\ s \vdash t\ \surd t$ **by** *fact+*
from *wt* **have** [*simp*]: $T = Void$ **by** *auto*
from *wt* **show** *?case*
proof (*rule WTrt-elim-cases*)
fix $U\ C\ TF\ T_2\ fm$
assume wt_1: $P,E,hp\ s \vdash Val\ v : U$
and *icto*: *class-type-of*$'\ U = \lfloor C \rfloor$
and *has*: $P \vdash C\ has\ F{:}TF\ (fm)\ in\ D$
and wt_2: $P,E,hp\ s \vdash e_2 : T_2$ **and** *TsubTF*: $P \vdash T_2 \leq TF$
have wt_1': $P,E,hp\ s' \vdash Val\ v : U$
by(*rule WTrt-hext-mono*$[OF\ wt_1\ red\text{-}hext\text{-}incr[OF\ red]]$)
obtain T_2' **where** wt_2': $P,E,hp\ s' \vdash e_2' : T_2'$ **and** $T'subT$: $P \vdash T_2' \leq T_2$
using $IH[OF\ conf\ wt_2\ tconf]$ **by** *blast*
have $P,E,hp\ s' \vdash Val\ v{\cdot}F\{D\}{:=}e_2' : Void$
by(*rule WTrtFAss*$[OF\ wt_1'\ icto\ has\ wt_2'\ widen\text{-}trans[OF\ T'subT\ TsubTF]]$)
thus *?case* **by** *auto*
next
fix T_2 **assume** *null*: $P,E,hp\ s \vdash Val\ v : NT$ **and** wt_2: $P,E,hp\ s \vdash e_2 : T_2$
from *null* **have** $v = Null$ **by** *simp*
moreover

```
    obtain T2' where P,E,hp s' ⊢ e2' : T2' ∧ P ⊢ T2' ≤ T2
      using IH[OF conf wt2 tconf] by blast
    ultimately show ?thesis by(fastforce intro:WTrtFAssNT)
  qed
next
  case RedFAss thus ?case by(auto simp del:fun-upd-apply)
next
  case RedFAssNull thus ?case unfolding sconf-def
    by(fastforce simp add: xcpt-subcls-Throwable[OF - wf])
next
  case (CASRed1 e s ta e' s' D F e2 e3)
  from CASRed1.prems(2) consider (NT) T2 T3 where
      P,E,hp s ⊢ e : NT T = Boolean P,E,hp s ⊢ e2 : T2 P,E,hp s ⊢ e3 : T3
    | (RefT) U T' C fm T2 T3 where
      P,E,hp s ⊢ e : U T = Boolean class-type-of' U = ⌊C⌋ P ⊢ C has F:T' (fm) in D
      P,E,hp s ⊢ e2 : T2 P ⊢ T2 ≤ T' P,E,hp s ⊢ e3 : T3 P ⊢ T3 ≤ T' volatile fm by fastforce
  thus ?case
  proof cases
    case NT
   have P,E,hp s'⊢ e' : NT using CASRed1.hyps(2)[OF CASRed1.prems(1) NT(1) CASRed1.prems(3)]
by auto
    moreover from NT CASRed1.hyps(1)[THEN red-hext-incr]
    have P,E,hp s' ⊢ e2 : T2 P,E,hp s' ⊢ e3 : T3 by(auto intro: WTrt-hext-mono)
    ultimately show ?thesis using NT by(auto intro: WTrtCASNT)
  next
    case RefT
    from CASRed1.hyps(2)[OF CASRed1.prems(1) RefT(1) CASRed1.prems(3)]
    obtain U' where wt1: P,E,hp s' ⊢ e' : U' P ⊢ U' ≤ U by blast
    from RefT CASRed1.hyps(1)[THEN red-hext-incr]
    have wt2: P,E,hp s' ⊢ e2 : T2 and wt3: P,E,hp s' ⊢ e3 : T3 by(auto intro: WTrt-hext-mono)
    show ?thesis
    proof(cases U' = NT)
      case True
      with RefT wt1 wt2 wt3 show ?thesis by(auto intro: WTrtCASNT)
    next
      case False
      with RefT(3) wt1 obtain C' where icto': class-type-of' U' = ⌊C'⌋
        and subclass: P ⊢ C' ≼* C by(blast intro: widen-is-class-type-of)
      have P ⊢ C' has F:T' (fm) in D by(rule has-field-mono[OF RefT(4) subclass])
      with RefT wt1 wt2 wt3 icto' show ?thesis by(auto intro!: WTrtCAS)
    qed
  qed
next
  case (CASRed2 e s ta e' s' v D F e3)
  consider (Null) v = Null | (Val) U C T' fm T2 T3 where
    class-type-of' U = ⌊C⌋ P ⊢ C has F:T' (fm) in D volatile fm
    P,E,hp s ⊢ e : T2 P ⊢ T2 ≤ T' P,E,hp s ⊢ e3 : T3 P ⊢ T3 ≤ T' T = Boolean
    typeof_hp s v = ⌊U⌋ using CASRed2.prems(2) by auto
  then show ?case
  proof cases
    case Null
    then show ?thesis using CASRed2
      by(force dest: red-hext-incr intro: WTrt-hext-mono WTrtCASNT)
  next
```

case *Val*
from *CASRed2.hyps(1)* **have** *hext*: $hp\ s \trianglelefteq hp\ s'$ **by**(*auto dest*: *red-hext-incr*)
with *Val(9)* **have** $typeof_{hp\ s'}\ v = \lfloor U \rfloor$ **by**(*rule type-of-hext-type-of*)
moreover from *CASRed2.hyps(2)*[*OF CASRed2.prems(1) Val(4) CASRed2.prems(3)*] *Val(5)*
obtain *T2'* **where** $P,E,hp\ s' \vdash e' : T2'\ P \vdash T2' \leq T'$ **by**(*auto intro*: *widen-trans*)
moreover from *Val(6) hext* **have** $P,E,hp\ s' \vdash e3 : T3$ **by**(*rule WTrt-hext-mono*)
ultimately show *?thesis* **using** *Val* **by**(*auto intro*: *WTrtCAS*)
qed
next
case (*CASRed3 e s ta e' s' v D F v'*)
consider (*Null*) $v = Null$ | (*Val*) *U C T' fm T2 T3* **where**
$T = Boolean$ *class-type-of'* $U = \lfloor C \rfloor$ $P \vdash C$ *has* $F{:}T'\ (fm)$ *in D volatile fm*
$P \vdash T2 \leq T'$ $P,E,hp\ s \vdash e : T3$ $P \vdash T3 \leq T'$
$typeof_{hp\ s}\ v = \lfloor U \rfloor$ $typeof_{hp\ s}\ v' = \lfloor T2 \rfloor$
using *CASRed3.prems(2)* **by** *auto*
then show *?case*
proof *cases*
case *Null*
then show *?thesis* **using** *CASRed3*
by(*force dest*: *red-hext-incr intro*: *type-of-hext-type-of WTrtCASNT*)
next
case *Val*
from *CASRed3.hyps(1)* **have** *hext*: $hp\ s \trianglelefteq hp\ s'$ **by**(*auto dest*: *red-hext-incr*)
with *Val(8,9)* **have** $typeof_{hp\ s'}\ v = \lfloor U \rfloor$ $typeof_{hp\ s'}\ v' = \lfloor T2 \rfloor$
by(*blast intro*: *type-of-hext-type-of*)+
moreover from *CASRed3.hyps(2)*[*OF CASRed3.prems(1) Val(6) CASRed3.prems(3)*] *Val(7)*
obtain *T3'* **where** $P,E,hp\ s' \vdash e' : T3'\ P \vdash T3' \leq T'$ **by**(*auto intro*: *widen-trans*)
ultimately show *?thesis* **using** *Val* **by**(*auto intro*: *WTrtCAS*)
qed
next
case *CASNull* **thus** *?case* **unfolding** *sconf-def*
by(*fastforce simp add*: *xcpt-subcls-Throwable*[*OF - wf*])
next
case (*CallObj e s ta e' s' M es T E*)
have *red*: $extTA,P,t \vdash \langle e,s \rangle\ -ta\rightarrow\ \langle e',s' \rangle$
and *IH*: $\bigwedge E\ T.\ [\![E \vdash s\ \surd;\ P,E,hp\ s \vdash e : T;\ P,hp\ s \vdash t\ \surd t]\!] \Longrightarrow \exists U.\ P,E,hp\ s' \vdash e' : U \wedge P \vdash U \leq T$
and *conf*: $E \vdash s\ \surd$ **and** *wt*: $P,E,hp\ s \vdash e{\cdot}M(es) : T$
and *tconf*: $P,hp\ s \vdash t\ \surd t$ **by** *fact*+
— We distinguish if e has type NT or a Class type
from *wt* **show** *?case*
proof(*rule WTrt-elim-cases*)
fix *T' C Ts meth D Us*
assume *wte*: $P,E,hp\ s \vdash e : T'$ **and** *icto*: *class-type-of'* $T' = \lfloor C \rfloor$
and *method*: $P \vdash C$ *sees* $M{:}Ts\rightarrow T = meth$ *in D*
and *wtes*: $P,E,hp\ s \vdash es\ [:]\ Us$ **and** *subs*: $P \vdash Us\ [\leq]\ Ts$
obtain *U* **where** *wte'*: $P,E,hp\ s' \vdash e' : U$ **and** *UsubC*: $P \vdash U \leq T'$
using *IH*[*OF conf wte tconf*] **by** *blast*
show *?thesis*
proof(*cases* $U = NT$)
case *True*
moreover have $P,E,hp\ s' \vdash es\ [:]\ Us$
by(*rule WTrts-hext-mono*[*OF wtes red-hext-incr*[*OF red*]])
ultimately show *?thesis* **using** *wte'* **by**(*blast intro*!: *WTrtCallNT*)

```
    next
      case False
      with icto UsubC obtain C'
        where icto': class-type-of' U = ⌊C'⌋ and subclass: P ⊢ C' ≼* C
        by(rule widen-is-class-type-of)

      obtain Ts' T' meth' D'
        where method': P ⊢ C' sees M:Ts'→T' = meth' in D'
        and subs': P ⊢ Ts [≤] Ts' and sub': P ⊢ T' ≤ T
        using Call-lemma[OF method subclass wf] by fast
      have wtes': P,E,hp s' ⊢ es [:] Us
        by(rule WTrts-hext-mono[OF wtes red-hext-incr[OF red]])
      show ?thesis using wtes' wte' icto' subs method' subs' sub' by(blast intro:widens-trans)
    qed
  next
    fix Ts
    assume P,E,hp s ⊢ e:NT
    hence P,E,hp s' ⊢ e' : NT using IH[OF conf - tconf] by fastforce
    moreover
    fix Ts assume wtes: P,E,hp s ⊢ es [:] Ts
    have P,E,hp s' ⊢ es [:] Ts
      by(rule WTrts-hext-mono[OF wtes red-hext-incr[OF red]])
    ultimately show ?thesis by(blast intro!:WTrtCallNT)
  qed
next
  case (CallParams es s ta es' s' v M T E)
  have reds: extTA,P,t ⊢ ⟨es,s⟩ [−ta→] ⟨es',s'⟩
   and IH: ⋀Ts E. ⟦E ⊢ s √; P,E,hp s ⊢ es [:] Ts; P,hp s ⊢ t √t⟧
         ⟹ ∃Ts'. P,E,hp s' ⊢ es' [:] Ts' ∧ P ⊢ Ts' [≤] Ts
   and conf: E ⊢ s √ and wt: P,E,hp s ⊢ Val v·M(es) : T
   and tconf: P,hp s ⊢ t √t by fact+
  from wt show ?case
  proof (rule WTrt-elim-cases)
    fix U C Ts meth D Us
    assume wte: P,E,hp s ⊢ Val v : U and icto: class-type-of' U = ⌊C⌋
      and P ⊢ C sees M:Ts→T = meth in D
      and wtes: P,E,hp s ⊢ es [:] Us and P ⊢ Us [≤] Ts
    moreover have P,E,hp s' ⊢ Val v : U
      by(rule WTrt-hext-mono[OF wte reds-hext-incr[OF reds]])
    moreover obtain Us' where P,E,hp s' ⊢ es' [:] Us' P ⊢ Us' [≤] Us
      using IH[OF conf wtes tconf] by blast
    ultimately show ?thesis by(fastforce intro:WTrtCall widens-trans)
  next
    fix Us
    assume null: P,E,hp s ⊢ Val v : NT and wtes: P,E,hp s ⊢ es [:] Us
    from null have v = Null by simp
    moreover
    obtain Us' where P,E,hp s' ⊢ es' [:] Us' ∧ P ⊢ Us' [≤] Us
      using IH[OF conf wtes tconf] by blast
    ultimately show ?thesis by(fastforce intro:WTrtCallNT)
  qed
next
  case (RedCall s a U M Ts T pns body D vs T' E)
  have hp: typeof-addr (hp s) a = ⌊U⌋
```

and *method*: $P \vdash$ *class-type-of* U *sees* M: $Ts \to T = \lfloor(pns,body)\rfloor$ *in* D
and *wt*: $P,E,hp\ s \vdash addr\ a{\cdot}M(map\ Val\ vs) : T'$ **by** *fact+*
obtain Ts' **where** *wtes*: $P,E,hp\ s \vdash map\ Val\ vs\ [:]\ Ts'$
and *subs*: $P \vdash Ts'\ [\leq]\ Ts$ **and** $T'isT$: $T' = T$
using *wt method hp wf* **by**(*auto 4 3 dest*: *sees-method-fun*)
from *wtes subs* **have** *length-vs*: *length vs* = *length Ts*
by(*auto simp add*: *WTrts-conv-list-all2 dest*!: *list-all2-lengthD*)
have *UsubD*: $P \vdash$ *ty-of-htype* $U \leq$ *Class* (*class-type-of* U)
by(*cases* U)(*simp-all add*: *widen-array-object*)
from *sees-wf-mdecl*[*OF wf method*] **obtain** T''
where *wtabody*: $P,[this\#pns\ [\mapsto]\ Class\ D\#Ts] \vdash body :: T''$
and $T''subT$: $P \vdash T'' \leq T$ **and** *length-pns*: *length pns* = *length Ts*
by(*fastforce simp*:*wf-mdecl-def simp del*:*map-upds-twist*)
from *wtabody* **have** $P,Map.empty(this\#pns\ [\mapsto]\ Class\ D\#Ts),hp\ s \vdash body : T''$
by(*rule WT-implies-WTrt*)
hence $P,E(this\#pns\ [\mapsto]\ Class\ D\#Ts),hp\ s \vdash body : T''$
by(*rule WTrt-env-mono*) *simp*
hence $P,E,hp\ s \vdash blocks\ (this\#pns)\ (Class\ D\#Ts)\ (Addr\ a\#vs)\ body : T''$
using *wtes subs hp sees-method-decl-above*[*OF method*] *length-vs length-pns UsubD*
by(*auto simp add*:*wt-blocks rel-list-all2-Cons2 intro*: *widen-trans*)
with $T''subT\ T'isT$ **show** *?case* **by** *blast*
next
case (*RedCallExternal s a U M Ts T' D vs ta va h' ta' e' s'*)
from $\langle P,t \vdash \langle a{\cdot}M(vs),hp\ s\rangle\ -ta\to ext\ \langle va,h'\rangle\rangle$ **have** *hp s* $\trianglelefteq$ *h'* **by**(*rule red-external-hext*)
with $\langle P,E,hp\ s \vdash addr\ a{\cdot}M(map\ Val\ vs) : T\rangle$
have $P,E,h' \vdash addr\ a{\cdot}M(map\ Val\ vs) : T$ **by**(*rule WTrt-hext-mono*)
moreover from $\langle typeof\text{-}addr\ (hp\ s)\ a = \lfloor U\rfloor\rangle$ $\langle P \vdash$ *class-type-of* U *sees* M: $Ts \to T' =$ *Native in* $D\rangle$
$\langle P,E,hp\ s \vdash addr\ a{\cdot}M(map\ Val\ vs) : T\rangle$
have $P,hp\ s \vdash a{\cdot}M(vs) : T'$
by(*fastforce simp add*: *external-WT'-iff dest*: *sees-method-fun*)
ultimately show *?case* **using** *RedCallExternal*
by(*auto 4 3 intro*: *red-external-conf-extRet*[*OF wf*] *intro*!: *wt-external-call simp add*: *sconf-def dest*: *sees-method-fun*[**where** C=*class-type-of* U])
next
case *RedCallNull* **thus** *?case* **unfolding** *sconf-def*
by(*fastforce simp add*: *xcpt-subcls-Throwable*[*OF - wf*])
next
case (*BlockRed e h x V vo ta e' h' x' T T' E*)
note $IH = \langle \bigwedge T\ E.\ [\![E \vdash (h,\ x(V := vo))\ \surd;\ P,E,hp\ (h,\ x(V := vo)) \vdash e : T;\ P,hp\ (h,\ x(V := vo)) \vdash t\ \surd t]\!]$
$\Longrightarrow \exists T'.\ P,E,hp\ (h',\ x') \vdash e' : T' \wedge P \vdash T' \leq T\rangle$[*simplified*]
from $\langle P,E,hp\ (h,\ x) \vdash \{V{:}T{=}vo;\ e\} : T'\rangle$ **have** $P,E(V{\mapsto}T),h \vdash e : T'$ **by**(*cases vo*, *auto*)
moreover from $\langle E \vdash (h,\ x)\ \surd\rangle$ $\langle P,E,hp\ (h,\ x) \vdash \{V{:}T{=}vo;\ e\} : T'\rangle$
have $(E(V \mapsto T)) \vdash (h,\ x(V := vo))\ \surd$
by(*cases vo*)(*simp add*: *lconf-def sconf-def*,*auto simp add*: *sconf-def conf-def intro*: *lconf-upd2*)
ultimately obtain T'' **where** wt': $P,E(V{\mapsto}T),h' \vdash e' : T''\ P \vdash T'' \leq T'$ **using** $\langle P,hp\ (h,\ x) \vdash t\ \surd t\rangle$
by(*auto dest*: *IH*)
{ fix *v*
assume *vo*: $x'\ V = \lfloor v\rfloor$
from $\langle (E(V \mapsto T)) \vdash (h,\ x(V := vo))\ \surd\rangle$ $\langle extTA,P,t \vdash \langle e,(h,\ x(V := vo))\rangle\ -ta\to\ \langle e',(h',\ x')\rangle\rangle$
$\langle P,E(V{\mapsto}T),h \vdash e : T'\rangle$
have $P,h' \vdash x'\ (:\leq)\ (E(V \mapsto T))$ **by**(*auto simp add*: *sconf-def dest*: *red-preserves-lconf*)
with *vo* **have** $\exists T'.\ typeof_{h'}\ v = \lfloor T'\rfloor \wedge P \vdash T' \leq T$ **by**(*fastforce simp add*: *sconf-def lconf-def*

conf-def)
then obtain T' **where** $typeof_{h'}\ v = \lfloor T' \rfloor\ P \vdash T' \leq T$ **by** *blast*
hence *?case* **using** *wt' vo* **by**(*auto*) **}**
moreover
{ assume $x'\ V = None$ **with** *wt'* **have** *?case* **by**(*auto*) **}**
ultimately show *?case* **by** *blast*
next
case *RedBlock* **thus** *?case* **by** *auto*
next
case (*SynchronizedRed1 o' s ta o'' s' e T E*)
have *red*: *extTA*,*P*,*t* ⊢ ⟨*o'*,*s*⟩ −*ta*→ ⟨*o''*,*s'*⟩ **by** *fact*
have *IH*: $\bigwedge T\ E.\ [\![E \vdash s\ \surd;\ P,E,hp\ s \vdash o' : T;\ P,hp\ s \vdash t\ \surd t]\!] \Longrightarrow \exists T'.\ P,E,hp\ s' \vdash o'' : T' \wedge P \vdash T' \leq T$ **by** *fact*
have *conf*: $E \vdash s\ \surd$ **by** *fact*
have *wt*: $P,E,hp\ s \vdash sync(o')\ e : T$ **by** *fact+*
thus *?case*
proof(*rule WTrt-elim-cases*)
fix *To*
assume *wto*: $P,E,hp\ s \vdash o' : To$
and *refT*: *is-refT To*
and *wte*: $P,E,hp\ s \vdash e : T$
from *IH*[*OF conf wto* ‹$P,hp\ s \vdash t\ \surd t$›] **obtain** To' **where** $P,E,hp\ s' \vdash o'' : To'$ **and** *sub*: $P \vdash To' \leq To$ **by** *auto*
moreover have $P,E,hp\ s' \vdash e : T$
by(*rule WTrt-hext-mono*[*OF wte red-hext-incr*[*OF red*]])
moreover have *is-refT To'* **using** *refT sub* **by**(*auto intro*: *widen-refT*)
ultimately show *?thesis* **by**(*auto*)
qed
next
case *SynchronizedNull* **thus** *?case* **unfolding** *sconf-def*
by(*fastforce simp add*: *xcpt-subcls-Throwable*[*OF - wf*])
next
case *LockSynchronized* **thus** *?case* **by**(*auto*)
next
case (*SynchronizedRed2 e s ta e' s' a T E*)
have *red*: *extTA*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e'*,*s'*⟩ **by** *fact*
have *IH*: $\bigwedge T\ E.\ [\![E \vdash s\ \surd;\ P,E,hp\ s \vdash e : T;\ P,hp\ s \vdash t\ \surd t]\!] \Longrightarrow \exists T'.\ P,E,hp\ s' \vdash e' : T' \wedge P \vdash T' \leq T$ **by** *fact*
have *conf*: $E \vdash s\ \surd$ **by** *fact*
have *wt*: $P,E,hp\ s \vdash insync(a)\ e : T$ **by** *fact*
thus *?case*
proof(*rule WTrt-elim-cases*)
fix *Ta*
assume $P,E,hp\ s \vdash e : T$
and *hpa*: *typeof-addr* (*hp s*) *a* = ⌊*Ta*⌋
from ‹$P,E,hp\ s \vdash e : T$› *conf* ‹$P,hp\ s \vdash t\ \surd t$› **obtain** T'
where $P,E,hp\ s' \vdash e' : T'\ P \vdash T' \leq T$ **by**(*blast dest*: *IH*)
moreover from *red* **have** *hext*: $hp\ s \trianglelefteq hp\ s'$ **by**(*auto dest*: *red-hext-incr*)
with *hpa* **have** $P,E,hp\ s' \vdash addr\ a : ty\text{-}of\text{-}htype\ Ta$
by(*auto intro*: *typeof-addr-hext-mono*)
ultimately show *?thesis* **by** *auto*
qed
next
case *UnlockSynchronized* **thus** *?case* **by**(*auto*)

next
case *SeqRed* **thus** *?case*
apply(*auto*)
apply(*drule WTrt-hext-mono*[*OF - red-hext-incr*], *assumption*)
by *auto*
next
case (*CondRed b s ta b′ s′ e1 e2 T E*)
have *red*: *extTA*,*P*,*t* ⊢ ⟨*b*,*s*⟩ −*ta*→ ⟨*b′*,*s′*⟩ **by** *fact*
have *IH*: ⋀*T E*. ⟦*E* ⊢ *s* √; *P*,*E*,*hp s* ⊢ *b* : *T*; *P*,*hp s* ⊢ *t* √*t*⟧ ⟹ ∃ *T′*. *P*,*E*,*hp s′* ⊢ *b′* : *T′* ∧ *P* ⊢ *T′* ≤ *T* **by** *fact*
have *conf*: *E* ⊢ *s* √ **by** *fact*
have *wt*: *P*,*E*,*hp s* ⊢ *if* (*b*) *e1 else e2* : *T* **by** *fact*
thus *?case*
proof(*rule WTrt-elim-cases*)
fix *T1 T2*
assume *wtb*: *P*,*E*,*hp s* ⊢ *b* : *Boolean*
and *wte1*: *P*,*E*,*hp s* ⊢ *e1* : *T1*
and *wte2*: *P*,*E*,*hp s* ⊢ *e2* : *T2*
and *lub*: *P* ⊢ *lub*(*T1*, *T2*) = *T*
from *IH*[*OF conf wtb* ‹*P*,*hp s* ⊢ *t* √*t*›] **have** *P*,*E*,*hp s′* ⊢ *b′* : *Boolean* **by**(*auto*)
moreover have *P*,*E*,*hp s′* ⊢ *e1* : *T1*
by(*rule WTrt-hext-mono*[*OF wte1 red-hext-incr*[*OF red*]])
moreover have *P*,*E*,*hp s′* ⊢ *e2* : *T2*
by(*rule WTrt-hext-mono*[*OF wte2 red-hext-incr*[*OF red*]])
ultimately show *?thesis* **using** *lub* **by** *auto*
qed
next
case (*ThrowRed e s ta e′ s′ T E*)
have *IH*: ⋀*T E*. ⟦*E* ⊢ *s* √; *P*,*E*,*hp s* ⊢ *e* : *T*; *P*,*hp s* ⊢ *t* √*t*⟧ ⟹ ∃ *T′*. *P*,*E*,*hp s′* ⊢ *e′* : *T′* ∧ *P* ⊢ *T′* ≤ *T* **by** *fact*
have *conf*: *E* ⊢ *s* √ **by** *fact*
have *wt*: *P*,*E*,*hp s* ⊢ *throw e* : *T* **by** *fact*
then obtain *T′*
where *wte*: *P*,*E*,*hp s* ⊢ *e* : *T′*
and *nobject*: *P* ⊢ *T′* ≤ *Class Throwable* **by** *auto*
from *IH*[*OF conf wte* ‹*P*,*hp s* ⊢ *t* √*t*›] **obtain** *T″*
where *wte′*: *P*,*E*,*hp s′* ⊢ *e′* : *T″*
and *PT′T″*: *P* ⊢ *T″* ≤ *T′* **by** *blast*
from *nobject PT′T″* **have** *P* ⊢ *T″* ≤ *Class Throwable*
by(*auto simp add*: *widen-Class*)(*erule notE*, *rule rtranclp-trans*)
hence *P*,*E*,*hp s′* ⊢ *throw e′* : *T* **using** *wte′ PT′T″*
by −(*erule WTrtThrow*)
thus *?case* **by**(*auto*)
next
case *RedThrowNull* **thus** *?case* **unfolding** *sconf-def*
by(*fastforce simp add*: *xcpt-subcls-Throwable*[*OF - wf*])
next
case (*TryRed e s ta e′ s′ C V e2 T E*)
have *red*: *extTA*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩ **by** *fact*
have *IH*: ⋀*T E*. ⟦*E* ⊢ *s* √; *P*,*E*,*hp s* ⊢ *e* : *T*; *P*,*hp s* ⊢ *t* √*t*⟧ ⟹ ∃ *T′*. *P*,*E*,*hp s′* ⊢ *e′* : *T′* ∧ *P* ⊢ *T′* ≤ *T* **by** *fact*
have *conf*: *E* ⊢ *s* √ **by** *fact*
have *wt*: *P*,*E*,*hp s* ⊢ *try e catch*(*C V*) *e2* : *T* **by** *fact*
thus *?case*

proof(*rule WTrt-elim-cases*)
fix *T1*
assume *wte*: *P*,*E*,*hp s* ⊢ *e* : *T1*
and *wte2*: *P*,*E*(*V* ↦ *Class C*),*hp s* ⊢ *e2* : *T*
and *sub*: *P* ⊢ *T1* ≤ *T*
from *IH*[*OF conf wte* ‹*P*,*hp s* ⊢ *t* √*t*›] **obtain** *T1′* **where** *P*,*E*,*hp s′* ⊢ *e′* : *T1′* **and** *P* ⊢ *T1′* ≤ *T1* **by**(*auto*)
moreover have *P*,*E*(*V* ↦ *Class C*),*hp s′* ⊢ *e2* : *T*
by(*rule WTrt-hext-mono*[*OF wte2 red-hext-incr*[*OF red*]])
ultimately show *?thesis* **using** *sub* **by**(*auto elim*: *widen-trans*)
qed
next
case *RedTryFail* **thus** *?case* **unfolding** *sconf-def*
by(*fastforce simp add*: *xcpt-subcls-Throwable*[*OF - wf*])
next
case *RedSeq* **thus** *?case* **by** *auto*
next
case *RedCondT* **thus** *?case* **by**(*auto dest*: *is-lub-upper*)
next
case *RedCondF* **thus** *?case* **by**(*auto dest*: *is-lub-upper*)
next
case *RedWhile* **thus** *?case* **by**(*fastforce*)
next
case *RedTry* **thus** *?case* **by** *auto*
next
case *RedTryCatch* **thus** *?case* **by**(*fastforce*)
next
case (*ListRed1 e s ta e′ s′ es Ts E*)
note *IH* = ‹⋀*T E*. ⟦*E* ⊢ *s* √; *P*,*E*,*hp s* ⊢ *e* : *T*; *P*,*hp s* ⊢ *t* √*t*⟧ ⟹ ∃*T′*. *P*,*E*,*hp s′* ⊢ *e′* : *T′* ∧ *P* ⊢ *T′* ≤ *T*›
from ‹*P*,*E*,*hp s* ⊢ *e* # *es* [:] *Ts*› **obtain** *T Ts′* **where** *Ts* = *T* # *Ts′* *P*,*E*,*hp s* ⊢ *e* : *T* *P*,*E*,*hp s* ⊢ *es* [:] *Ts′* **by** *auto*
with *IH*[*of E T*] ‹*E* ⊢ *s* √› *WTrts-hext-mono*[*OF* ‹*P*,*E*,*hp s* ⊢ *es* [:] *Ts′*› *red-hext-incr*[*OF* ‹*extTA*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩›]]
show *?case* **using** ‹*P*,*hp s* ⊢ *t* √*t*› **by**(*auto simp add*: *list-all2-Cons2 intro*: *widens-refl*)
next
case *ListRed2* **thus** *?case*
by(*fastforce dest*: *hext-typeof-mono*[*OF reds-hext-incr*])
qed(*fastforce*)+

end

4.14 Progress and type safety theorem for the multithreaded system

theory *ProgressThreaded*
imports
Threaded
TypeSafe
../Framework/FWProgress
begin

lemma *lock-ok-ls-Some-ex-ts-not-final*:

assumes *lock*: *lock-ok ls ts*
and *hl*: *has-lock* (*ls* $ *l*) *t*
shows ∃ *e x ln*. *ts t* = ⌊((*e*, *x*), *ln*)⌋ ∧ ¬ *final e*
proof −
from *lock* **have** *lock-thread-ok ls ts*
by(*rule lock-ok-lock-thread-ok*)
with *hl* **obtain** *e x ln*
where *tst*: *ts t* = ⌊((*e*, *x*), *ln*)⌋
by(*auto dest*!: *lock-thread-okD*)
{ assume *final e*
hence *expr-locks e l* = *0* **by**(*rule final-locks*)
with *lock tst* **have** *has-locks* (*ls* $ *l*) *t* = *0*
by(*auto dest*: *lock-okD2*[*rule-format*, **where** *l*=*l*])
with *hl* **have** *False* **by** *simp* **}**
with *tst* **show** *?thesis* **by** *auto*
qed

4.14.1 Preservation lemmata

4.14.2 Definite assignment

abbreviation
def-ass-ts-ok :: (*'addr*,*'thread-id*,*'addr expr* × *'addr locals*) *thread-info* ⇒ *'heap* ⇒ *bool*
where
def-ass-ts-ok ≡ *ts-ok* (λ*t* (*e*, *x*) *h*. *D e* ⌊*dom x*⌋)

context *J-heap-base* **begin**

lemma assumes *wf*: *wf-J-prog P*
shows *red-def-ass-new-thread*:
⟦ *P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; *NewThread t''* (*e''*, *x''*) *c''* ∈ *set* {|*ta*|}*t* ⟧ ⟹ *D e''* ⌊*dom x''*⌋

and *reds-def-ass-new-thread*:
⟦ *P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *NewThread t''* (*e''*, *x''*) *c''* ∈ *set* {|*ta*|}*t* ⟧ ⟹ *D e''* ⌊*dom x''*⌋
proof(*induct rule*: *red-reds.inducts*)
case (*RedCallExternal s a T M vs ta va h' ta' e' s'*)
then obtain *C fs a* **where** *subThread*: *P* ⊢ *C* ⪯* *Thread* **and** *ext*: *extNTA2J P* (*C*, *run*, *a*) = (*e''*, *x''*)
by(*fastforce dest*: *red-external-new-thread-sub-thread*)
from *sub-Thread-sees-run*[*OF wf subThread*] **obtain** *D pns body*
where *sees*: *P* ⊢ *C sees run*: []→*Void* = ⌊(*pns*, *body*)⌋ *in D* **by** *auto*
from *sees-wf-mdecl*[*OF wf this*] **have** *D body* ⌊{*this*}⌋
by(*auto simp add*: *wf-mdecl-def*)
with *sees ext* **show** *?case* **by**(*clarsimp simp del*: *fun-upd-apply*)
qed(*auto simp add*: *ta-upd-simps*)

lemma *lifting-wf-def-ass*: *wf-J-prog P* ⟹ *lifting-wf final-expr* (*mred P*) (λ*t* (*e*, *x*) *m*. *D e* ⌊*dom x*⌋)
apply(*unfold-locales*)
apply(*auto dest*: *red-preserves-defass red-def-ass-new-thread*)
done

lemma *def-ass-ts-ok-J-start-state*:
⟦ *wf-J-prog P*; *P* ⊢ *C sees M*:*Ts*→*T* = ⌊(*pns*, *body*)⌋ *in D*; *length vs* = *length Ts* ⟧ ⟹
def-ass-ts-ok (*thr* (*J-start-state P C M vs*)) *h*

apply(*rule ts-okI*)
apply(*drule (1) sees-wf-mdecl*)
apply(*clarsimp simp add*: *wf-mdecl-def start-state-def split*: *if-split-asm*)
done

end

4.14.3 typeability

context *J-heap-base* **begin**

definition *type-ok* :: *'addr J-prog* ⇒ *env* × *ty* ⇒ *'addr expr* ⇒ *'heap* ⇒ *bool*
where *type-ok P* ≡ (λ(*E*, *T*) *e c*. (∃ *T'*. (*P*,*E*,*c* ⊢ *e* : *T'* ∧ *P* ⊢ *T'* ≤ *T*)))

definition *J-sconf-type-ET-start* :: *'m prog* ⇒ *cname* ⇒ *mname* ⇒ (*'thread-id* ⇀ (*env* × *ty*))
where
J-sconf-type-ET-start P C M ≡
let (-, -, *T*, -) = *method P C M*
in ([*start-tid* ↦ (*Map.empty*, *T*)])

lemma fixes *E* :: *env*
assumes *wf*: *wf-J-prog P*
shows *red-type-newthread*:
⟦ *P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; *P*,*E*,*hp s* ⊢ *e* : *T*; *NewThread t''* (*e''*, *x''*) (*hp s'*) ∈ *set* {|*ta*|}*t* ⟧
⟹ ∃ *E T*. *P*,*E*,*hp s'* ⊢ *e''* : *T* ∧ *P*,*hp s'* ⊢ *x''* (:≤) *E*
and *reds-type-newthread*:
⟦ *P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *NewThread t''* (*e''*, *x''*) (*hp s'*) ∈ *set* {|*ta*|}*t*; *P*,*E*,*hp s* ⊢ *es* [:] *Ts* ⟧
⟹ ∃ *E T*. *P*,*E*,*hp s'* ⊢ *e''* : *T* ∧ *P*,*hp s'* ⊢ *x''* (:≤) *E*
proof(*induct arbitrary*: *E T* **and** *E Ts rule*: *red-reds.inducts*)
case (*RedCallExternal s a U M Ts T' D vs ta va h' ta' e' s'*)
from ‹*NewThread t''* (*e''*, *x''*) (*hp s'*) ∈ *set* {|*ta'*|}*t*› ‹*ta'* = *extTA2J P ta*›
obtain *C' M' a'* **where** *nt*: *NewThread t''* (*C'*, *M'*, *a'*) (*hp s'*) ∈ *set* {|*ta*|}*t*
and *extNTA2J P* (*C'*, *M'*, *a'*) = (*e''*, *x''*) **by** *fastforce*
from *red-external-new-thread-sees*[*OF wf* ‹*P*,*t* ⊢ ⟨*a*•*M*(*vs*),*hp s*⟩ −*ta*→*ext* ⟨*va*,*h'*⟩› *nt*] ‹*typeof-addr* (*hp s*) *a* = ⌊*U*⌋›
obtain *T pns body D* **where** *h'a'*: *typeof-addr h' a'* = ⌊*Class-type C'*⌋
and *sees*: *P* ⊢ *C' sees M'*: []→*T* = ⌊(*pns*, *body*)⌋ *in D* **by** *auto*
from *sees-wf-mdecl*[*OF wf sees*] **obtain** *T* **where** *P*,[*this* ↦ *Class D*] ⊢ *body* :: *T*
by(*auto simp add*: *wf-mdecl-def*)
hence *WTrt P* (*hp s'*) [*this* ↦ *Class D*] *body T* **by**(*rule WT-implies-WTrt*)
moreover from *sees* **have** *P* ⊢ *C'* ≼* *D* **by**(*rule sees-method-decl-above*)
with *h'a'* **have** *P*,*h'* ⊢ [*this* ↦ *Addr a'*] (:≤) [*this* ↦ *Class D*] **by**(*auto simp add*: *lconf-def conf-def*)
ultimately show *?case* **using** *h'a' sees* ‹*s'* = (*h'*, *lcl s*)›
‹*extNTA2J P* (*C'*, *M'*, *a'*) = (*e''*, *x''*)› **by**(*fastforce intro*: *sees-method-decl-above*)
qed(*fastforce simp add*: *ta-upd-simps*)+

end

context *J-heap-conf-base* **begin**

definition *sconf-type-ok* :: (*env* × *ty*) ⇒ *'thread-id* ⇒ *'addr expr* × *'addr locals* ⇒ *'heap* ⇒ *bool*
where
sconf-type-ok ET t ex h ≡ *fst ET* ⊢ (*h*, *snd ex*) √ ∧ *type-ok P ET* (*fst ex*) *h* ∧ *P*,*h* ⊢ *t* √*t*

abbreviation *sconf-type-ts-ok* ::
(*'thread-id* ⇀ (*env* × *ty*)) ⇒ (*'addr*,*'thread-id*,*'addr expr* × *'addr locals*) *thread-info* ⇒ *'heap* ⇒ *bool*
where
sconf-type-ts-ok ≡ *ts-inv sconf-type-ok*

lemma *ts-inv-ok-J-sconf-type-ET-start*:
ts-inv-ok (*thr* (*J-start-state P C M vs*)) (*J-sconf-type-ET-start P C M*)
by(*rule ts-inv-okI*)(*simp add*: *start-state-def J-sconf-type-ET-start-def split-beta*)

end

lemma (**in** *J-heap*) *red-preserve-welltype*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*e*, (*h*, *x*)⟩ −*ta*→ ⟨*e'*, (*h'*, *x'*)⟩; *P*,*E*,*h* ⊢ *e''* : *T* ⟧ ⟹ *P*,*E*,*h'* ⊢ *e''* : *T*
by(*auto elim*: *WTrt-hext-mono dest*!: *red-hext-incr*)

context *J-heap-conf* **begin**

lemma *sconf-type-ts-ok-J-start-state*:
⟦ *wf-J-prog P*; *wf-start-state P C M vs* ⟧
⟹ *sconf-type-ts-ok* (*J-sconf-type-ET-start P C M*) (*thr* (*J-start-state P C M vs*)) (*shr* (*J-start-state P C M vs*))
apply(*erule wf-start-state.cases*)
apply(*rule ts-invI*)
apply(*simp add*: *start-state-def split*: *if-split-asm*)
apply(*frule* (*1*) *sees-wf-mdecl*)
apply(*auto simp add*: *wf-mdecl-def J-sconf-type-ET-start-def sconf-type-ok-def sconf-def type-ok-def*)
apply(*erule hconf-start-heap*)
apply(*erule preallocated-start-heap*)
apply(*erule wf-prog-wf-syscls*)
apply(*frule list-all2-lengthD*)
apply(*auto simp add*: *wt-blocks confs-conv-map intro*: *WT-implies-WTrt*)[*1*]
apply(*erule tconf-start-heap-start-tid*)
apply(*erule wf-prog-wf-syscls*)
done

lemma *J-start-state-sconf-type-ok*:
assumes *wf*: *wf-J-prog P*
and *ok*: *wf-start-state P C M vs*
shows *ts-ok* (λ*t x h*. ∃ *ET*. *sconf-type-ok ET t x h*) (*thr* (*J-start-state P C M vs*)) *start-heap*
using *sconf-type-ts-ok-J-start-state*[*OF assms*]
unfolding *shr-start-state* **by**(*rule ts-inv-into-ts-ok-Ex*)

end

context *J-conf-read* **begin**

lemma *red-preserves-type-ok*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e'*,*s'*⟩; *wf-J-prog P*; *E* ⊢ *s* √; *type-ok P* (*E*, *T*) *e* (*hp s*); *P*,*hp s* ⊢ *t* √*t* ⟧
⟹ *type-ok P* (*E*, *T*) *e'* (*hp s'*)
apply(*clarsimp simp add*: *type-ok-def*)
apply(*subgoal-tac* ∃ *T''*. *P*,*E*,*hp s'* ⊢ *e'* : *T''* ∧ *P* ⊢ *T''* ≤ *T'*)
apply(*fast elim*: *widen-trans*)
by(*rule subject-reduction*)

lemma *lifting-inv-sconf-subject-ok*:
assumes *wf*: *wf-J-prog P*
shows *lifting-inv final-expr* (*mred P*) *sconf-type-ok*
proof(*unfold-locales*)
fix *t x m ta x' m' i*
assume *mred*: *mred P t* (*x*, *m*) *ta* (*x'*, *m'*)
and *sconf-type-ok i t x m*
moreover obtain *e l* **where** *x* [*simp*]: $x = (e, l)$ **by**(*cases x*, *auto*)
moreover obtain *e' l'* **where** *x'* [*simp*]: $x' = (e', l')$ **by**(*cases x'*, *auto*)
moreover obtain *E T* **where** *i* [*simp*]: $i = (E, T)$ **by**(*cases i*, *auto*)
ultimately have *sconf-type*: *sconf-type-ok* (E, T) *t* (e, l) *m*
and *red*: $P,t \vdash \langle e, (m, l)\rangle -ta\rightarrow \langle e', (m', l')\rangle$ **by** *auto*
from *sconf-type* **have** *sconf*: $E \vdash (m, l) \surd$ **and** *type-ok P* (E, T) *e m* **and** *tconf*: $P,m \vdash t \surd t$
by(*auto simp add*: *sconf-type-ok-def*)
then obtain T' **where** $P,E,m \vdash e : T'$ $P \vdash T' \leq T$ **by**(*auto simp add*: *type-ok-def*)
from $\langle E \vdash (m, l) \surd\rangle$ $\langle P,E,m \vdash e : T'\rangle$ *red tconf*
have $E \vdash (m', l') \surd$ **by**(*auto elim*: *red-preserves-sconf*)
moreover
from *red* $\langle P,E,m \vdash e : T'\rangle$ *wf* $\langle E \vdash (m, l) \surd\rangle$ *tconf*
obtain T'' **where** $P,E,m' \vdash e' : T''$ $P \vdash T'' \leq T'$
by(*auto dest*: *subject-reduction*)
note $\langle P,E,m' \vdash e' : T''\rangle$
moreover
from $\langle P \vdash T'' \leq T'\rangle$ $\langle P \vdash T' \leq T\rangle$
have $P \vdash T'' \leq T$ **by**(*rule widen-trans*)
moreover from *mred tconf* **have** $P,m' \vdash t \surd t$ **by**(*rule red-tconf.preserves-red*)
ultimately have *sconf-type-ok* (E, T) *t* (e', l') *m'*
by(*auto simp add*: *sconf-type-ok-def type-ok-def*)
thus *sconf-type-ok i t x' m'* **by** *simp*
next
fix *t x m ta x' m' i t'' x''*
assume *mred*: *mred P t* (*x*, *m*) *ta* (*x'*, *m'*)
and *sconf-type-ok i t x m*
and *NewThread t'' x'' m'* $\in$ *set* $\{\!|ta|\!\}_t$
moreover obtain *e l* **where** *x* [*simp*]: $x = (e, l)$ **by**(*cases x*, *auto*)
moreover obtain *e' l'* **where** *x'* [*simp*]: $x' = (e', l')$ **by**(*cases x'*, *auto*)
moreover obtain *E T* **where** *i* [*simp*]: $i = (E, T)$ **by**(*cases i*, *auto*)
moreover obtain *e'' l''* **where** *x''* [*simp*]: $x'' = (e'', l'')$ **by**(*cases x''*, *auto*)
ultimately have *sconf-type*: *sconf-type-ok* (E, T) *t* (e, l) *m*
and *red*: $P,t \vdash \langle e, (m, l)\rangle -ta\rightarrow \langle e', (m', l')\rangle$
and *nt*: *NewThread t''* (e'', l'') *m'* $\in$ *set* $\{\!|ta|\!\}_t$ **by** *auto*
from *sconf-type* **have** *sconf*: $E \vdash (m, l) \surd$ **and** *type-ok P* (E, T) *e m* **and** *tconf*: $P,m \vdash t \surd t$
by(*auto simp add*: *sconf-type-ok-def*)
then obtain T' **where** $P,E,m \vdash e : T'$ $P \vdash T' \leq T$ **by**(*auto simp add*: *type-ok-def*)
from *nt* $\langle P,E,m \vdash e : T'\rangle$ *red* **have** $\exists E\ T.\ P,E,m' \vdash e'' : T \wedge P,m' \vdash l'' (:\leq) E$
by(*fastforce dest*: *red-type-newthread*[*OF wf*])
then obtain E'' T'' **where** $P,E'',m' \vdash e'' : T''$ $P,m' \vdash l'' (:\leq) E''$ **by** *blast*
moreover
from *sconf red* $\langle P,E,m \vdash e : T'\rangle$ *tconf* **have** $E \vdash (m', l') \surd$
by(*auto intro*: *red-preserves-sconf*)
moreover from *mred tconf* $\langle$*NewThread t'' x'' m'* $\in$ *set* $\{\!|ta|\!\}_t\rangle$ **have** $P,m' \vdash t'' \surd t$
by(*rule red-tconf.preserves-NewThread*)
ultimately show $\exists i''$. *sconf-type-ok i'' t'' x'' m'*
by(*auto simp add*: *sconf-type-ok-def type-ok-def sconf-def*)

next
fix $t\ x\ m\ ta\ x'\ m'\ i\ i''\ t''\ x''$
assume *mred*: $mred\ P\ t\ (x,\ m)\ ta\ (x',\ m')$
and *sconf-type-ok* $i\ t\ x\ m$
and *sconf-type-ok* $i''\ t''\ x''\ m$
moreover obtain $e\ l$ **where** x [*simp*]: $x = (e,\ l)$ **by**(*cases* x, *auto*)
moreover obtain $e'\ l'$ **where** x' [*simp*]: $x' = (e',\ l')$ **by**(*cases* x', *auto*)
moreover obtain $E\ T$ **where** i [*simp*]: $i = (E,\ T)$ **by**(*cases* i, *auto*)
moreover obtain $e''\ l''$ **where** x'' [*simp*]: $x'' = (e'',\ l'')$ **by**(*cases* x'', *auto*)
moreover obtain $E''\ T''$ **where** i'' [*simp*]: $i'' = (E'',\ T'')$ **by**(*cases* i'', *auto*)
ultimately have *sconf-type*: *sconf-type-ok* $(E,\ T)\ t\ (e,\ l)\ m$
and *red*: $P,t \vdash \langle e,\ (m,\ l)\rangle\ -ta\rightarrow\ \langle e',\ (m',\ l')\rangle$
and *sc*: *sconf-type-ok* $(E'',\ T'')\ t''\ (e'',\ l'')\ m$ **by** *auto*
from *sconf-type* **obtain** T' **where** $P,E,m \vdash e : T'$ **and** $P,m \vdash t\ \surd t$
by(*auto simp add*: *sconf-type-ok-def type-ok-def*)
from *sc* **have** *sconf*: $E'' \vdash (m,\ l'')\ \surd$ **and** *type-ok* $P\ (E'',\ T'')\ e''\ m$ **and** $P,m \vdash t''\ \surd t$
by(*auto simp add*: *sconf-type-ok-def*)
then obtain T''' **where** $P,E'',m \vdash e'' : T'''\ P \vdash T''' \leq T''$ **by**(*auto simp add*: *type-ok-def*)
moreover from *red* $\langle P,E'',m \vdash e'' : T'''\rangle$ **have** $P,E'',m' \vdash e'' : T'''$
by(*rule red-preserve-welltype*)
moreover from *sconf red* $\langle P,E,m \vdash e : T'\rangle$ **have** *hconf* m'
unfolding *sconf-def* **by**(*auto dest*: *red-preserves-hconf*)
moreover {
from *red* **have** *hext* $m\ m'$ **by**(*auto dest*: *red-hext-incr*)
moreover from *sconf* **have** $P,m \vdash l''\ (:\leq)\ E''$ *preallocated* m
by(*simp-all add*: *sconf-def*)
ultimately have $P,m' \vdash l''\ (:\leq)\ E''$ *preallocated* m'
by(*blast intro*: *lconf-hext preallocated-hext*)+ }
moreover from *mred* $\langle P,m \vdash t\ \surd t\rangle\ \langle P,m \vdash t''\ \surd t\rangle$
have $P,m' \vdash t''\ \surd t$ **by**(*rule red-tconf.preserves-other*)
ultimately have *sconf-type-ok* $(E'',\ T'')\ t''\ (e'',\ l'')\ m'$
by(*auto simp add*: *sconf-type-ok-def sconf-def type-ok-def*)
thus *sconf-type-ok* $i''\ t''\ x''\ m'$ **by** *simp*
qed

end

4.14.4 *wf-red*

context *J-progress* **begin**

context begin

declare *red-mthr.actions-ok-iff* [*simp del*]
declare *red-mthr.actions-ok.cases* [*rule del*]
declare *red-mthr.actions-ok.intros* [*rule del*]

lemma assumes *wf*: *wf-prog wf-md P*
shows *red-wf-red-aux*:
$[\![\ P,t \vdash \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',s'\rangle;\ \neg$ *red-mthr.actions-ok'* $(ls,\ (ts,\ m),\ ws,\ is)\ t\ ta;$
sync-ok e; *hconf* $(hp\ s)$; $P,hp\ s \vdash t\ \surd t;$
$\forall l.$ *has-locks* $(ls\ \$\ l)\ t \geq$ *expr-locks* $e\ l;$
$ws\ t = None \vee$
$(\exists a\ vs\ w\ T\ Ts\ Tr\ D.$ *call* $e = \lfloor(a,\ wait,\ vs)\rfloor \wedge$ *typeof-addr* $(hp\ s)\ a = \lfloor T\rfloor \wedge P \vdash$ *class-type-of* T

sees wait: *Ts*→*Tr* = *Native in D* ∧ *ws t* = ⌊*PostWS w*⌋) ⟧
⟹ ∃ *e''* *s''* *ta'*. *P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta'*→ ⟨*e''*,*s''*⟩ ∧
(*red-mthr.actions-ok* (*ls*, (*ts*, *m*), *ws*, *is*) *t ta'* ∨
red-mthr.actions-ok' (*ls*, (*ts*, *m*), *ws*, *is*) *t ta'* ∧ *red-mthr.actions-subset ta' ta*)
(**is** ⟦ -; -; -; -; -; -; *?wakeup e s* ⟧ ⟹ *?concl e s ta*)
and *reds-wf-red-aux*:
⟦ *P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*,*s'*⟩; ¬ *red-mthr.actions-ok'* (*ls*, (*ts*, *m*), *ws*, *is*) *t ta*;
sync-oks es; *hconf* (*hp s*); *P*,*hp s* ⊢ *t* √*t*;
∀ *l*. *has-locks* (*ls* $ *l*) *t* ≥ *expr-lockss es l*;
ws t = *None* ∨
(∃ *a vs w T Ts T Tr D*. *calls es* = ⌊(*a*, *wait*, *vs*)⌋ ∧ *typeof-addr* (*hp s*) *a* = ⌊*T*⌋ ∧ *P* ⊢ *class-type-of* *T sees wait*: *Ts*→*Tr* = *Native in D* ∧ *ws t* = ⌊*PostWS w*⌋) ⟧
⟹ ∃ *es''* *s''* *ta'*. *P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta'*→] ⟨*es''*,*s''*⟩ ∧
(*red-mthr.actions-ok* (*ls*, (*ts*, *m*), *ws*, *is*) *t ta'* ∨
red-mthr.actions-ok' (*ls*, (*ts*, *m*), *ws*, *is*) *t ta'* ∧ *red-mthr.actions-subset ta' ta*)
proof(*induct rule*: *red-reds.inducts*)
case (*SynchronizedRed2 e s ta e' s' a*)
note *IH* = ‹⟦¬ *red-mthr.actions-ok'* (*ls*, (*ts*, *m*), *ws*, *is*) *t ta*; *sync-ok e*; *hconf* (*hp s*); *P*,*hp s* ⊢ *t* √*t*;
∀ *l*. *expr-locks e l* ≤ *has-locks* (*ls* $ *l*) *t*; *?wakeup e s*⟧
⟹ *?concl e s ta*›
note ‹¬ *red-mthr.actions-ok'* (*ls*, (*ts*, *m*), *ws*, *is*) *t ta*›
moreover from ‹*sync-ok* (*insync*(*a*) *e*)› **have** *sync-ok e* **by** *simp*
moreover note ‹*hconf* (*hp s*)› ‹*P*,*hp s* ⊢ *t* √*t*›
moreover from ‹∀ *l*. *expr-locks* (*insync*(*a*) *e*) *l* ≤ *has-locks* (*ls* $ *l*) *t*›
have ∀ *l*. *expr-locks e l* ≤ *has-locks* (*ls* $ *l*) *t* **by**(*force split*: *if-split-asm*)
moreover from ‹*?wakeup* (*insync*(*a*) *e*) *s*› **have** *?wakeup e s* **by** *auto*
ultimately have *?concl e s ta* **by**(*rule IH*)
thus *?case* **by**(*fastforce intro*: *red-reds.SynchronizedRed2*)
next
case *RedCall* **thus** *?case*
by(*auto simp add*: *is-val-iff contains-insync-conv contains-insyncs-conv red-mthr.actions-ok'-empty* *red-mthr.actions-ok'-ta-upd-obs dest*: *sees-method-fun*)
next
case (*RedCallExternal s a U M Ts T D vs ta va h' ta' e' s'*)
from ‹*?wakeup* (*addr a*•*M*(*map Val vs*)) *s*›
have *wset* (*ls*, (*ts*, *m*), *ws*, *is*) *t* = *None* ∨ (*M* = *wait* ∧ (∃ *w*. *wset* (*ls*, (*ts*, *m*), *ws*, *is*) *t* = ⌊*PostWS* *w*⌋)) **by** *auto*
with *wf* ‹*P*,*t* ⊢ ⟨*a*•*M*(*vs*),*hp s*⟩ −*ta*→*ext* ⟨*va*, *h'*⟩› ‹*P*,*hp s* ⊢ *t* √*t*› ‹*hconf* (*hp s*)›
obtain *ta''* *va'* *h''* **where** *red'*: *P*,*t* ⊢ ⟨*a*•*M*(*vs*),*hp s*⟩ −*ta''*→*ext* ⟨*va'*,*h''*⟩
and *aok*: *red-mthr.actions-ok* (*ls*, (*ts*, *m*), *ws*, *is*) *t ta''* ∨
red-mthr.actions-ok' (*ls*, (*ts*, *m*), *ws*, *is*) *t ta''* ∧ *final-thread.actions-subset ta'' ta*
by(*rule red-external-wf-red*)
from *aok* ‹*ta'* = *extTA2J P ta*›
have *red-mthr.actions-ok* (*ls*, (*ts*, *m*), *ws*, *is*) *t* (*extTA2J P ta''*) ∨
red-mthr.actions-ok' (*ls*, (*ts*, *m*), *ws*, *is*) *t* (*extTA2J P ta''*) ∧ *red-mthr.actions-subset* (*extTA2J* *P ta''*) *ta'*
by(*auto simp add*: *red-mthr.actions-ok'-convert-extTA red-mthr.actions-ok-iff elim*: *final-thread.actions-subset.cases* *del*: *subsetI*)
moreover from *red'* ‹*typeof-addr* (*hp s*) *a* = ⌊*U*⌋› ‹*P* ⊢ *class-type-of U sees M*: *Ts*→*T* = *Native in* *D*›
obtain *s''* *e''* **where** *P*,*t* ⊢ ⟨*addr a*•*M*(*map Val vs*),*s*⟩ −*extTA2J P ta''*→ ⟨*e''*,*s''*⟩
by(*fastforce intro*: *red-reds.RedCallExternal*)
ultimately show *?case* **by** *blast*
next

```
  case LockSynchronized
  hence False by(auto simp add: lock-ok-las'-def finfun-upd-apply ta-upd-simps)
  thus ?case ..
next
  case (UnlockSynchronized a v s)
  from ‹∀ l. expr-locks (insync(a) Val v) l ≤ has-locks (ls $ l) t›
  have has-lock (ls $ a) t by(force split: if-split-asm)
 with UnlockSynchronized have False by(auto simp add: lock-ok-las'-def finfun-upd-apply ta-upd-simps)
  thus ?case ..
next
  case (SynchronizedThrow2 a ad s)
  from ‹∀ l. expr-locks (insync(a) Throw ad) l ≤ has-locks (ls $ l) t›
  have has-lock (ls $ a) t by(force split: if-split-asm)
  with SynchronizedThrow2 have False
    by(auto simp add: lock-ok-las'-def finfun-upd-apply ta-upd-simps)
  thus ?case ..
next
  case BlockRed thus ?case by(simp)(blast intro: red-reds.intros)
qed
 (simp-all add: is-val-iff contains-insync-conv contains-insyncs-conv red-mthr.actions-ok'-empty
   red-mthr.actions-ok'-ta-upd-obs thread-action'-to-thread-action.simps red-mthr.actions-ok-iff
  split: if-split-asm del: split-paired-Ex,
  (blast intro: red-reds.intros elim: add-leE)+)

end

end

context J-heap-base begin

lemma shows red-ta-satisfiable:
  P,t ⊢ ⟨e, s⟩ −ta→ ⟨e', s'⟩ ⟹ ∃ s. red-mthr.actions-ok s t ta
  and reds-ta-satisfiable:
  P,t ⊢ ⟨es, s⟩ [−ta→] ⟨es', s'⟩ ⟹ ∃ s. red-mthr.actions-ok s t ta
apply(induct rule: red-reds.inducts)
apply(fastforce simp add: lock-ok-las-def finfun-upd-apply intro: exI[where x=K$ None] exI[where
x=K$ ⌊(t, 0)⌋] may-lock.intros dest: red-external-ta-satisfiable[where final=final-expr :: ('addr expr
× 'addr locals) ⇒ bool])+
done

end

context J-typesafe begin

lemma wf-progress:
  assumes wf: wf-J-prog P
  shows progress final-expr (mred P)
          (red-mthr.wset-Suspend-ok P ({s. sync-es-ok (thr s) (shr s) ∧ lock-ok (locks s) (thr s)} ∩ {s.
∃ Es. sconf-type-ts-ok Es (thr s) (shr s)} ∩ {s. def-ass-ts-ok (thr s) (shr s)}))
  (is progress - - ?wf-state)
proof
  {
    fix s t x ta x' m' w
    assume mred P t (x, shr s) ta (x', m')
```

```
    and Suspend: Suspend w ∈ set {|ta|}w
  moreover obtain e xs where x: x = (e, xs) by(cases x)
  moreover obtain e' xs' where x': x' = (e', xs') by(cases x')
  ultimately have red: P,t ⊢ ⟨e, (shr s, xs)⟩ −ta→ ⟨e', (m', xs')⟩ by simp
  from red-Suspend-is-call[OF red Suspend]
  show ¬ final-expr x' by(auto simp add: x')
 }
 note Suspend-final = this
 {
  fix s
  assume s: s ∈ ?wf-state
  hence lock-thread-ok (locks s) (thr s)
   by(auto dest: red-mthr.wset-Suspend-okD1 intro: lock-ok-lock-thread-ok)
  moreover
  have red-mthr.wset-final-ok (wset s) (thr s)
  proof(rule red-mthr.wset-final-okI)
   fix t w
   assume wset s t = ⌊w⌋
   from red-mthr.wset-Suspend-okD2[OF s this]
   obtain x0 ta x m1 w' ln'' and s0 :: ('addr, 'thread-id, 'heap) J-state
    where mred: mred P t (x0, shr s0) ta (x, m1)
    and Suspend: Suspend w' ∈ set {|ta|}w
    and tst: thr s t = ⌊(x, ln'')⌋ by blast
   from Suspend-final[OF mred Suspend] tst
   show ∃x ln. thr s t = ⌊(x, ln)⌋ ∧ ¬ final-expr x by blast
  qed
  ultimately show lock-thread-ok (locks s) (thr s) ∧ red-mthr.wset-final-ok (wset s) (thr s) ..
 }
next
 fix s t ex ta e'x' m'
 assume wfs: s ∈ ?wf-state
   and thr s t = ⌊(ex, no-wait-locks)⌋
   and mred P t (ex, shr s) ta (e'x', m')
   and wait: ¬ waiting (wset s t)
 moreover obtain ls ts m ws is where s: s = (ls, (ts, m), ws, is) by(cases s) fastforce
 moreover obtain e x where ex: ex = (e, x) by(cases ex)
 moreover obtain e' x' where e'x': e'x' = (e', x') by(cases e'x')
 ultimately have tst: ts t = ⌊(ex, no-wait-locks)⌋
   and red: P,t ⊢ ⟨e, (m, x)⟩ −ta→ ⟨e', (m', x')⟩ by auto
 from wf have wwf: wwf-J-prog P by(rule wf-prog-wwf-prog)
 from wfs s obtain Es where aeos: sync-es-ok ts m
   and lockok: lock-ok ls ts
   and sconf-type-ts-ok Es ts m
   by(auto dest: red-mthr.wset-Suspend-okD1)
 with tst ex obtain E T where sconf: sconf-type-ok (E, T) t (e, x) m
   and aoe: sync-ok e by(fastforce dest: ts-okD ts-invD)
 then obtain T' where hconf m P,E,m ⊢ e : T' preallocated m
   by(auto simp add: sconf-type-ok-def sconf-def type-ok-def)
 from ‹sconf-type-ts-ok Es ts m› s have thread-conf P (thr s) (shr s)
   by(auto dest: ts-invD intro!: ts-okI simp add: sconf-type-ok-def)
 with ‹thr s t = ⌊(ex, no-wait-locks)⌋› have P,shr s ⊢ t √t by(auto dest: ts-okD)

 show ∃ta' x' m'. mred P t (ex, shr s) ta' (x', m') ∧
       (red-mthr.actions-ok s t ta' ∨ red-mthr.actions-ok' s t ta' ∧ red-mthr.actions-subset ta' ta)
```

proof(*cases red-mthr.actions-ok′ s t ta*)
case *True*
have *red-mthr.actions-subset ta ta* **..**
with *True* ‹*mred P t* (*ex*, *shr s*) *ta* (*e′x′*, *m′*)› **show** *?thesis* **by** *blast*
next
case *False*
from *lock-okD2*[*OF lockok*, *OF tst*[*unfolded ex*]]
have *locks*: ∀ *l*. *has-locks* (*ls* $ *l*) *t* ≥ *expr-locks e l* **by** *simp*
have *ws t* = *None* ∨ (∃ *a vs w T Ts Tr D*. *call e* = ⌊(*a*, *wait*, *vs*)⌋ ∧ *typeof-addr* (*hp* (*m*, *x*)) *a* = ⌊*T*⌋ ∧ *P* ⊢ *class-type-of T sees wait*: *Ts*→*Tr* = *Native in D* ∧ *ws t* = ⌊*PostWS w*⌋)
proof(*cases ws t*)
case *None* **thus** *?thesis* **..**
next
case (*Some w*)
with *red-mthr.wset-Suspend-okD2*[*OF wfs*, *of t w*] *tst ex s*
obtain *e0 x0 m0 ta0 w′ s1 tta1*
where *red0*: *P*,*t* ⊢ ⟨*e0*, (*m0*, *x0*)⟩ −*ta0*→ ⟨*e*, (*shr s1*, *x*)⟩
and *Suspend*: *Suspend w′* ∈ *set* {|*ta0*|}*w*
and *s1*: *P* ⊢ *s1* −▷*tta1*→∗ *s* **by** *auto*
from *red-Suspend-is-call*[*OF red0 Suspend*] **obtain** *a vs T Ts Tr D*
where *call*: *call e* = ⌊(*a*, *wait*, *vs*)⌋
and *type*: *typeof-addr m0 a* = ⌊*T*⌋
and *iec*: *P* ⊢ *class-type-of T sees wait*: *Ts*→*Tr* = *Native in D* **by** *fastforce*
from *red0* **have** *m0* ≤ *shr s1* **by**(*auto dest*: *red-hext-incr*)
also from *s1* **have** *shr s1* ≤ *shr s* **by**(*rule RedT-hext-incr*)
finally have *typeof-addr* (*shr s*) *a* = ⌊*T*⌋ **using** *type*
by(*rule typeof-addr-hext-mono*)
moreover from *Some wait s* **obtain** *w′* **where** *ws t* = ⌊*PostWS w′*⌋
by(*auto simp add*: *not-waiting-iff*)
ultimately show *?thesis* **using** *call iec s* **by** *auto*
qed
from *red-wf-red-aux*[*OF wf red False*[*unfolded s*] *aoe - - locks*, *OF - - this*] ‹*hconf m*› ‹*P*,*shr s* ⊢ *t* √*t*› *ex s*
show *?thesis* **by** *fastforce*
qed
next
fix *s t x*
assume *wfs*: *s* ∈ *?wf-state*
and *tst*: *thr s t* = ⌊(*x*, *no-wait-locks*)⌋
and *nfin*: ¬ *final-expr x*
obtain *e xs* **where** *x*: *x* = (*e*, *xs*) **by**(*cases x*)
from *wfs* **have** *def-ass-ts-ok* (*thr s*) (*shr s*) **by**(*auto dest*: *red-mthr.wset-Suspend-okD1*)
with *tst x* **have** *DA*: $\mathcal{D}$ *e* ⌊*dom xs*⌋ **by**(*auto dest*: *ts-okD*)
from *wfs* **obtain** *Es* **where** *sconf-type-ts-ok Es* (*thr s*) (*shr s*)
by(*auto dest*: *red-mthr.wset-Suspend-okD1*)
with *tst x* **obtain** *E T* **where** *sconf-type-ok* (*E*, *T*) *t* (*e*, *xs*) (*shr s*) **by**(*auto dest*: *ts-invD*)
then obtain *T′* **where** *hconf* (*shr s*) *P*,*E*,*shr s* ⊢ *e* : *T′*
by(*auto simp add*: *sconf-type-ok-def sconf-def type-ok-def*)
from *red-progress*(*1*)[*OF wf-prog-wwf-prog*[*OF wf*] *this DA*, **where** *extTA*=*extTA2J P* **and** *t*=*t*] *nfin x*
show ∃ *ta x′ m′*. *mred P t* (*x*, *shr s*) *ta* (*x′*, *m′*) **by** *fastforce*
next
fix *s t x xm ta xm′*
assume *s* ∈ *?wf-state*

```
    and thr s t = ⌊(x, no-wait-locks)⌋
    and mred P t xm ta xm'
    and Notified ∈ set ⦃ta⦄w ∨ WokenUp ∈ set ⦃ta⦄w
  thus collect-waits ta = {}
    by(auto dest: red-ta-Wakeup-no-Join-no-Lock-no-Interrupt simp: split-beta)
next
  fix s t x ta x' m'
  assume s ∈ ?wf-state
    and thr s t = ⌊(x, no-wait-locks)⌋
    and mred P t (x, shr s) ta (x', m')
  thus ∃ s'. red-mthr.actions-ok s' t ta
    by(fastforce simp add: split-beta dest!: red-ta-satisfiable)
qed

lemma redT-progress-deadlock:
  assumes wf: wf-J-prog P
  and wf-start: wf-start-state P C M vs
  and Red: P ⊢ J-start-state P C M vs −▷ttas→* s
  and ndead: ¬ red-mthr.deadlock P s
  shows ∃ t' ta' s'. P ⊢ s −t'▷ta'→ s'
proof −
  let ?wf-state = red-mthr.wset-Suspend-ok P ({s. sync-es-ok (thr s) (shr s) ∧ lock-ok (locks s) (thr
s)} ∩ {s. ∃ Es. sconf-type-ts-ok Es (thr s) (shr s)} ∩ {s. def-ass-ts-ok (thr s) (shr s)})
  interpret red-mthr: progress
    final-expr mred P convert-RA ?wf-state
    using wf by(rule wf-progress)
  from wf-start obtain Ts T pns body D
    where start: start-heap-ok P ⊢ C sees M:Ts→T = ⌊(pns, body)⌋ in D P,start-heap ⊢ vs [:≤] Ts
    by(cases) auto
  from start have len: length Ts = length vs by(auto dest: list-all2-lengthD)

  have invariant3p (mredT P) ?wf-state
   by(rule red-mthr.invariant3p-wset-Suspend-ok) (intro invariant3p-IntI invariant3p-sync-es-ok-lock-ok[OF
wf] lifting-inv.invariant3p-ts-inv[OF lifting-inv-sconf-subject-ok[OF wf]] lifting-wf.invariant3p-ts-ok[OF
lifting-wf-def-ass[OF wf]])
  moreover note Red moreover
  have start': J-start-state P C M vs ∈ ?wf-state
    apply(rule red-mthr.wset-Suspend-okI)
   apply(blast intro: sconf-type-ts-ok-J-start-state sync-es-ok-J-start-state lock-ok-J-start-state def-ass-ts-ok-J-start-state
start wf len len[symmetric] wf-start)
    apply(simp add: start-state-def split-beta)
    done
  ultimately have s ∈ ?wf-state unfolding red-mthr.RedT-def
    by(rule invariant3p-rtrancl3p)
  thus ?thesis using ndead by(rule red-mthr.redT-progress)
qed

lemma redT-progress-deadlocked:
  assumes wf: wf-J-prog P
  and wf-start: wf-start-state P C M vs
  and Red: P ⊢ J-start-state P C M vs −▷ttas→* s
  and ndead: red-mthr.not-final-thread s t ¬ t ∈ red-mthr.deadlocked P s
  shows ∃ t' ta' s'. P ⊢ s −t'▷ta'→ s'
using wf wf-start Red
```

proof(*rule redT-progress-deadlock*)
from *ndead* **show** ¬ *red-mthr.deadlock P s*
unfolding *red-mthr.deadlock-eq-deadlocked′*
by(*auto simp add*: *red-mthr.deadlocked′-def*)
qed

4.14.5 Type safety proof

theorem *TypeSafetyT*:
fixes *C* **and** *M* **and** *ttas* **and** *Es*
defines *Es* == *J-sconf-type-ET-start P C M*
and *Es′* == *upd-invs Es sconf-type-ok* (*concat* (*map* (*thr-a* ∘ *snd*) *ttas*))
assumes *wf*: *wf-J-prog P*
and *start-wf*: *wf-start-state P C M vs*
and *RedT*: *P* ⊢ *J-start-state P C M vs* −▷*ttas*→∗ *s′*
and *nored*: ¬ (∃ *t ta s″*. *P* ⊢ *s′* −*t*▷*ta*→ *s″*)
shows *thread-conf P* (*thr s′*) (*shr s′*)
and *thr s′ t* = ⌊((*e′*, *x′*), *ln′*)⌋ ⟹
(∃ *v*. *e′* = *Val v* ∧ (∃ *E T*. *Es′ t* = ⌊(*E*, *T*)⌋ ∧ *P*,*shr s′* ⊢ *v* :≤ *T*) ∧ *ln′* = *no-wait-locks*)
∨ (∃ *a C*. *e′* = *Throw a* ∧ *typeof-addr* (*shr s′*) *a* = ⌊*Class-type C*⌋ ∧ *P* ⊢ *C* ⪯∗ *Throwable* ∧ *ln′* = *no-wait-locks*)
∨ (*t* ∈ *red-mthr.deadlocked P s′* ∧ (∃ *E T*. *Es′ t* = ⌊(*E*, *T*)⌋ ∧ (∃ *T′*. *P*,*E*,*shr s′* ⊢ *e′* : *T′* ∧ *P* ⊢ *T′* ≤ *T*)))
(**is** - ⟹ *?thesis2*)
and *Es* ⊆*m* *Es′*
proof −
from *start-wf* **obtain** *Ts T pns body D*
where *start-heap*: *start-heap-ok*
and *sees*: *P* ⊢ *C sees M*:*Ts*→*T* = ⌊(*pns*, *body*)⌋ *in D*
and *conf*: *P*,*start-heap* ⊢ *vs* [:≤] *Ts*
by *cases auto*

from *RedT* **show** *thread-conf P* (*thr s′*) (*shr s′*)
by(*rule red-tconf.RedT-preserves*)(*rule thread-conf-start-state*[*OF start-heap wf-prog-wf-syscls*[*OF wf*]])

show *Es* ⊆*m* *Es′* **using** *RedT ts-inv-ok-J-sconf-type-ET-start*
unfolding *Es′-def Es-def* **by**(*rule red-mthr.RedT-upd-inv-ext*)

assume *thr s′ t* = ⌊((*e′*, *x′*), *ln′*)⌋
moreover obtain *ls′ ts′ m′ ws′ is′* **where** *s′* [*simp*]: *s′* = (*ls′*, (*ts′*, *m′*), *ws′*, *is′*) **by**(*cases s′*) *fastforce*
ultimately have *es′t*: *ts′ t* = ⌊((*e′*, *x′*), *ln′*)⌋ **by** *simp*
from *wf* **have** *wwf*: *wwf-J-prog P* **by**(*rule wf-prog-wwf-prog*)
from *conf* **have** *len*: *length vs* = *length Ts* **by**(*rule list-all2-lengthD*)
from *RedT def-ass-ts-ok-J-start-state*[*OF wf sees len*] **have** *defass′*: *def-ass-ts-ok ts′ m′*
by(*fastforce dest*: *lifting-wf.RedT-preserves*[*OF lifting-wf-def-ass*, *OF wf*])
from *RedT sync-es-ok-J-start-state*[*OF wf sees len*[*symmetric*]] *lock-ok-J-start-state*[*OF wf sees len*[*symmetric*]]
have *lock′*: *lock-ok ls′ ts′* **by** (*fastforce dest*: *RedT-preserves-lock-ok*[*OF wf*])
from *RedT sync-es-ok-J-start-state*[*OF wf sees len*[*symmetric*]] **have** *addr′*: *sync-es-ok ts′ m′*
by(*fastforce dest*: *RedT-preserves-sync-ok*[*OF wf*])
from *RedT sconf-type-ts-ok-J-start-state*[*OF wf start-wf*]
have *sconf-subject′*: *sconf-type-ts-ok Es′ ts′ m′* **unfolding** *Es′-def Es-def*
by(*fastforce dest*: *lifting-inv.RedT-invariant*[*OF lifting-inv-sconf-subject-ok*, *OF wf*] *intro*: *thread-conf-start-state*[- *wf-prog-wf-syscls*[*OF wf*]])

```
  with es't obtain E T where ET: Es' t = ⌊(E, T)⌋
    and sconf-type-ok (E, T) t (e', x') m' by(auto dest!: ts-invD)
  { assume final e'
    have ln' = no-wait-locks
    proof(rule ccontr)
      assume ln' ≠ no-wait-locks
      then obtain l where ln' $ l > 0
        by(auto simp add: neq-no-wait-locks-conv)
      from lock' es't have has-locks (ls' $ l) t + ln' $ l = expr-locks e' l
        by(auto dest: lock-okD2)
      with ‹ln' $ l > 0› have expr-locks e' l > 0 by simp
      moreover from ‹final e'› have expr-locks e' l = 0 by(rule final-locks)
      ultimately show False by simp
    qed }
  note ln' = this
  { assume ∃ v. e' = Val v
    then obtain v where v: e' = Val v by blast
    with sconf-subject' ET es't have P,m' ⊢ v :≤ T
      by(auto dest: ts-invD simp add: type-ok-def sconf-type-ok-def conf-def)
    moreover from v ln' have ln' = no-wait-locks by(auto)
   ultimately have ∃ v. e' = Val v ∧ (∃ E T. Es' t = ⌊(E, T)⌋ ∧ P,m' ⊢ v :≤ T ∧ ln' = no-wait-locks)
      using ET v by blast }
  moreover
  { assume ∃ a. e' = Throw a
    then obtain a where a: e' = Throw a by blast
    with sconf-subject' ET es't have ∃ T'. P,E,m' ⊢ e' : T' ∧ P ⊢ T' ≤ T
      apply −
      apply(drule ts-invD, assumption)
      by(clarsimp simp add: type-ok-def sconf-type-ok-def)
    then obtain T' where P,E,m' ⊢ e' : T' and P ⊢ T' ≤ T by blast
    with a have ∃ C. typeof-addr m' a = ⌊Class-type C⌋ ∧ P ⊢ C ⪯* Throwable
      by(auto simp add: widen-Class)
    moreover from a ln' have ln' = no-wait-locks by(auto)
   ultimately have ∃ a C. e' = Throw a ∧ typeof-addr m' a = ⌊Class-type C⌋ ∧ P ⊢ C ⪯* Throwable
∧ ln' = no-wait-locks
      using a by blast }
  moreover
  { assume nfine': ¬ final e'
    with es't have red-mthr.not-final-thread s' t
      by(auto intro: red-mthr.not-final-thread.intros)
    with nored have t ∈ red-mthr.deadlocked P s'
      by −(erule contrapos-np, rule redT-progress-deadlocked[OF wf start-wf RedT])
    moreover
    from ‹sconf-type-ok (E, T) t (e', x') m'›
    obtain T'' where P,E,m' ⊢ e' : T'' P ⊢ T'' ≤ T
      by(auto simp add: sconf-type-ok-def type-ok-def)
    with ET have ∃ E T. Es' t = ⌊(E, T)⌋ ∧ (∃ T'. P,E,m' ⊢ e' : T' ∧ P ⊢ T' ≤ T)
      by blast
    ultimately have t ∈ red-mthr.deadlocked P s' ∧ (∃ E T. Es' t = ⌊(E, T)⌋ ∧ (∃ T'. P,E,m' ⊢ e' :
T' ∧ P ⊢ T' ≤ T)) .. }
  ultimately show ?thesis2 by simp(blast)
qed

end
```

end

4.15 Preservation of Deadlock

theory *Deadlocked*
imports
ProgressThreaded
begin

context *J-progress* **begin**

lemma *red-wt-hconf-hext*:
assumes *wf*: *wf-J-prog P*
and *hconf*: *hconf H*
and *tconf*: $P,H \vdash t\ \surd t$
shows ⟦ *convert-extTA extNTA,P,t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; *P,E,H* ⊢ *e* : *T*; *hext H* (*hp s*) ⟧
⟹ ∃ *ta' e' s'*. *convert-extTA extNTA,P,t* ⊢ ⟨*e*, (*H*, *lcl s*)⟩ −*ta'*→ ⟨*e'*, *s'*⟩ ∧
collect-locks ⦃*ta*⦄$_l$ = *collect-locks* ⦃*ta'*⦄$_l$ ∧ *collect-cond-actions* ⦃*ta*⦄$_c$ = *collect-cond-actions* ⦃*ta'*⦄$_c$ ∧
collect-interrupts ⦃*ta*⦄$_i$ = *collect-interrupts* ⦃*ta'*⦄$_i$
and ⟦ *convert-extTA extNTA,P,t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *P,E,H* ⊢ *es* [:] *Ts*; *hext H* (*hp s*) ⟧
⟹ ∃ *ta' es' s'*. *convert-extTA extNTA,P,t* ⊢ ⟨*es*, (*H*, *lcl s*)⟩ [−*ta'*→] ⟨*es'*, *s'*⟩ ∧
collect-locks ⦃*ta*⦄$_l$ = *collect-locks* ⦃*ta'*⦄$_l$ ∧ *collect-cond-actions* ⦃*ta*⦄$_c$ = *collect-cond-actions* ⦃*ta'*⦄$_c$ ∧
collect-interrupts ⦃*ta*⦄$_i$ = *collect-interrupts* ⦃*ta'*⦄$_i$
proof(*induct arbitrary*: *E T* **and** *E Ts rule*: *red-reds.inducts*)
case (*RedNew h' a h C l*)
thus *?case*
by(*cases allocate H* (*Class-type C*) = {})(*fastforce simp add*: *ta-upd-simps intro*: *RedNewFail red-reds.RedNew*)+
next
case (*RedNewFail h C l*)
thus *?case*
by(*cases allocate H* (*Class-type C*) = {})(*fastforce simp add*: *ta-upd-simps intro*: *red-reds.RedNewFail RedNew*)+
next
case *NewArrayRed* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case (*RedNewArray i h' a h T l E T'*)
thus *?case*
by(*cases allocate H* (*Array-type T* (*nat* (*sint i*))) = {})(*fastforce simp add*: *ta-upd-simps intro*: *red-reds.RedNewArray RedNewArrayFail*)+
next
case *RedNewArrayNegative* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case (*RedNewArrayFail i h T l E T'*)
thus *?case*
by(*cases allocate H* (*Array-type T* (*nat* (*sint i*))) = {})(*fastforce simp add*: *ta-upd-simps intro*: *RedNewArray red-reds.RedNewArrayFail*)+
next
case *CastRed* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next

```
  case (RedCast s v U T E T')
  from ‹P,E,H ⊢ Cast T (Val v) : T'› show ?case
  proof(rule WTrt-elim-cases)
    fix T''
    assume wt: P,E,H ⊢ Val v : T'' T' = T
    thus ?thesis
      by(cases P ⊢ T'' ≤ T)(fastforce intro: red-reds.RedCast red-reds.RedCastFail)+
  qed
next
  case (RedCastFail s v U T E T')
  from ‹P,E,H ⊢ Cast T (Val v) : T'›
  obtain T'' where P,E,H ⊢ Val v : T'' T = T' by auto
  thus ?case
    by(cases P ⊢ T'' ≤ T)(fastforce intro: red-reds.RedCast red-reds.RedCastFail)+
next
  case InstanceOfRed thus ?case by(fastforce intro: red-reds.intros)
next
  case RedInstanceOf thus ?case
    using [[hypsubst-thin = true]]
    by auto((rule exI conjI red-reds.RedInstanceOf)+, auto)
next
  case BinOpRed1 thus ?case by(fastforce intro: red-reds.intros)
next
  case BinOpRed2 thus ?case by(fastforce intro: red-reds.intros)
next
  case RedBinOp thus ?case by(fastforce intro: red-reds.intros)
next
  case RedBinOpFail thus ?case by(fastforce intro: red-reds.intros)
next
  case RedVar thus ?case by(fastforce intro: red-reds.intros)
next
  case LAssRed thus ?case by(fastforce intro: red-reds.intros)
next
  case RedLAss thus ?case by(fastforce intro: red-reds.intros)
next
  case AAccRed1 thus ?case by(fastforce intro: red-reds.intros)
next
  case AAccRed2 thus ?case by(fastforce intro: red-reds.intros)
next
  case RedAAccNull thus ?case by(fastforce intro: red-reds.intros)
next
  case RedAAccBounds thus ?case
    by(fastforce intro: red-reds.RedAAccBounds dest: hext-arrD)
next
  case (RedAAcc h a T n i v l E T')
  from ‹P,E,H ⊢ addr a⌊Val (Intg i)⌉ : T'›
  have wt: P,E,H ⊢ addr a : T'⌊⌉ by(auto)
  with ‹H ◁ hp (h, l)› ‹typeof-addr h a = ⌊Array-type T n⌋›
  have Ha: typeof-addr H a = ⌊Array-type T n⌋ by(auto dest: hext-arrD)
  with ‹0 <=s i› ‹sint i < int n›
  have nat (sint i) < n
    by (simp add: word-sle-eq nat-less-iff)
  with Ha have P,H ⊢ a@ACell (nat (sint i)) : T
    by(auto intro: addr-loc-type.intros)
```

from *heap-read-total*[*OF hconf this*]
obtain *v* **where** *heap-read H a* (*ACell* (*nat* (*sint i*))) *v* **by** *blast*
with *Ha* ‹*0* <=*s i*› ‹*sint i* < *int n*› **show** *?case*
by(*fastforce intro*: *red-reds.RedAAcc simp add*: *ta-upd-simps*)
next
case *AAssRed1* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *AAssRed2* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *AAssRed3* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *RedAAssNull* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *RedAAssBounds* **thus** *?case* **by**(*fastforce intro*: *red-reds.RedAAssBounds dest*: *hext-arrD*)
next
case (*RedAAssStore s a T n i w U E T'*)
from ‹*P*,*E*,*H* ⊢ *addr a*⌊*Val* (*Intg i*)⌉ := *Val w* : *T'*›
obtain *T''* *T'''* **where** *wt*: *P*,*E*,*H* ⊢ *addr a* : *T''*⌊⌉
and *wtw*: *P*,*E*,*H* ⊢ *Val w* : *T'''* **by** *auto*
with ‹*H* ◁ *hp s*› ‹*typeof-addr* (*hp s*) *a* = ⌊*Array-type T n*⌋›
have *Ha*: *typeof-addr H a* = ⌊*Array-type T n*⌋ **by**(*auto dest*: *hext-arrD*)
from ‹*typeof*$_{hp\ s}$ *w* = ⌊*U*⌋› *wtw* ‹*H* ◁ *hp s*› **have** *typeof*$_H$ *w* = ⌊*U*⌋
by(*auto dest*: *type-of-hext-type-of*)
with *Ha* ‹*0* <=*s i*› ‹*sint i* < *int n*› ‹¬ *P* ⊢ *U* ≤ *T*› **show** *?case*
by(*fastforce intro*: *red-reds.RedAAssStore*)
next
case (*RedAAss h a T n i w U h' l E T'*)
from ‹*P*,*E*,*H* ⊢ *addr a*⌊*Val* (*Intg i*)⌉ := *Val w* : *T'*›
obtain *T''* *T'''* **where** *wt*: *P*,*E*,*H* ⊢ *addr a* : *T''*⌊⌉
and *wtw*: *P*,*E*,*H* ⊢ *Val w* : *T'''* **by** *auto*
with ‹*H* ◁ *hp* (*h*, *l*)› ‹*typeof-addr h a* = ⌊*Array-type T n*⌋›
have *Ha*: *typeof-addr H a* = ⌊*Array-type T n*⌋ **by**(*auto dest*: *hext-arrD*)
from ‹*typeof*$_h$ *w* = ⌊*U*⌋› *wtw* ‹*H* ◁ *hp* (*h*, *l*)› **have** *typeof*$_H$ *w* = ⌊*U*⌋
by(*auto dest*: *type-of-hext-type-of*)
moreover
with ‹*P* ⊢ *U* ≤ *T*› **have** *conf*: *P*,*H* ⊢ *w* :≤ *T*
by(*auto simp add*: *conf-def*)
from ‹*0* <=*s i*› ‹*sint i* < *int n*›
have *nat* (*sint i*) < *n*
by (*simp add*: *word-sle-eq nat-less-iff*)
with *Ha* **have** *P*,*H* ⊢ *a*@*ACell* (*nat* (*sint i*)) : *T*
by(*auto intro*: *addr-loc-type.intros*)
from *heap-write-total*[*OF hconf this conf*]
obtain *H'* **where** *heap-write H a* (*ACell* (*nat* (*sint i*))) *w H'* **..**
ultimately show *?case* **using** ‹*0* <=*s i*› ‹*sint i* < *int n*› *Ha* ‹*P* ⊢ *U* ≤ *T*›
by(*fastforce simp del*: *split-paired-Ex intro*: *red-reds.RedAAss*)
next
case *ALengthRed* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case (*RedALength h a T n l E T'*)
from ‹*P*,*E*,*H* ⊢ *addr a*•*length* : *T'*›
obtain *T''* **where** [*simp*]: *T'* = *Integer*
and *wta*: *P*,*E*,*H* ⊢ *addr a* : *T''*⌊⌉ **by**(*auto*)
then obtain *n''* **where** *typeof-addr H a* = ⌊*Array-type T'' n''*⌋ **by**(*auto*)

```
  thus ?case by(fastforce intro: red-reds.RedALength)
next
  case RedALengthNull show ?case by(fastforce intro: red-reds.RedALengthNull)
next
  case FAccRed thus ?case by(fastforce intro: red-reds.intros)
next
  case (RedFAcc h a D F v l E T)
  from ‹P,E,H ⊢ addr a•F{D} : T› obtain U C' fm
    where wt: P,E,H ⊢ addr a : U
    and icto: class-type-of' U = ⌊C'⌋
    and has: P ⊢ C' has F:T (fm) in D
    by(auto)
  then obtain hU where Ha: typeof-addr H a = ⌊hU⌋ U = ty-of-htype hU by(auto)
  with icto ‹P ⊢ C' has F:T (fm) in D› have P,H ⊢ a@CField D F : T
    by(auto intro: addr-loc-type.intros)
  from heap-read-total[OF hconf this]
  obtain v where heap-read H a (CField D F) v by blast
  thus ?case by(fastforce intro: red-reds.RedFAcc simp add: ta-upd-simps)
next
  case RedFAccNull thus ?case by(fastforce intro: red-reds.intros)
next
  case FAssRed1 thus ?case by(fastforce intro: red-reds.intros)
next
  case FAssRed2 thus ?case by(fastforce intro: red-reds.intros)
next
  case RedFAssNull thus ?case by(fastforce intro: red-reds.intros)
next
  case (RedFAss h a D F v h' l E T)
  from ‹P,E,H ⊢ addr a•F{D} := Val v : T› obtain U C' T' T2 fm
    where wt: P,E,H ⊢ addr a : U
    and icto: class-type-of' U = ⌊C'⌋
    and has: P ⊢ C' has F:T' (fm) in D
    and wtv: P,E,H ⊢ Val v : T2
    and T2T: P ⊢ T2 ≤ T' by(auto)
  moreover from wt obtain hU where Ha: typeof-addr H a = ⌊hU⌋ U = ty-of-htype hU by(auto)
  with icto has have adal: P,H ⊢ a@CField D F : T' by(auto intro: addr-loc-type.intros)
  from wtv T2T have P,H ⊢ v :≤ T' by(auto simp add: conf-def)
  from heap-write-total[OF hconf adal this]
  obtain h' where heap-write H a (CField D F) v h' ..
  thus ?case by(fastforce intro: red-reds.RedFAss)
next
  case CASRed1 thus ?case by(fastforce intro: red-reds.intros)
next
  case CASRed2 thus ?case by(fastforce intro: red-reds.intros)
next
  case CASRed3 thus ?case by(fastforce intro: red-reds.intros)
next
  case CASNull thus ?case by(fastforce intro: red-reds.intros)
next
  case (RedCASSucceed h a D F v v' h' l)
  note split-paired-Ex[simp del]
  from RedCASSucceed.prems(1) obtain T' fm T2 T3 U C where *:
    T = Boolean class-type-of' U = ⌊C⌋ P ⊢ C has F:T' (fm) in D
    volatile fm P ⊢ T2 ≤ T' P ⊢ T3 ≤ T'
```

```
    P,E,H ⊢ Val v : T2 P,E,H ⊢ Val v' : T3 P,E,H ⊢ addr a : U by auto
  then have adal: P,H ⊢ a@CField D F : T' by(auto intro: addr-loc-type.intros)
  from heap-read-total[OF hconf this] obtain v'' where v': heap-read H a (CField D F) v'' by blast
  show ?case
  proof(cases v'' = v)
    case True
    from ∗ have P,H ⊢ v' :≤ T' by(auto simp add: conf-def)
    from heap-write-total[OF hconf adal this] True ∗ v'
    show ?thesis by(fastforce intro: red-reds.RedCASSucceed)
  next
    case False
    then show ?thesis using ∗ v' by(fastforce intro: RedCASFail)
  qed
next
  case (RedCASFail h a D F v'' v v' l)
  note split-paired-Ex[simp del]
  from RedCASFail.prems(1) obtain T' fm T2 T3 U C where ∗:
    T = Boolean class-type-of' U = ⌊C⌋ P ⊢ C has F:T' (fm) in D
    volatile fm P ⊢ T2 ≤ T' P ⊢ T3 ≤ T'
    P,E,H ⊢ Val v : T2 P,E,H ⊢ Val v' : T3 P,E,H ⊢ addr a : U by auto
  then have adal: P,H ⊢ a@CField D F : T' by(auto intro: addr-loc-type.intros)
  from heap-read-total[OF hconf this] obtain v''' where v'': heap-read H a (CField D F) v''' by blast
  show ?case
  proof(cases v''' = v)
    case True
    from ∗ have P,H ⊢ v' :≤ T' by(auto simp add: conf-def)
    from heap-write-total[OF hconf adal this] True ∗ v''
    show ?thesis by(fastforce intro: red-reds.RedCASSucceed)
  next
    case False
    then show ?thesis using ∗ v'' by(fastforce intro: red-reds.RedCASFail)
  qed
next
  case CallObj thus ?case by(fastforce intro: red-reds.intros)
next
  case CallParams thus ?case by(fastforce intro: red-reds.intros)
next
  case (RedCall s a U M Ts T pns body D vs E T')
  from ‹P,E,H ⊢ addr a•M(map Val vs) : T'›
  obtain U' C' Ts' meth D' Ts''
    where wta: P,E,H ⊢ addr a : U'
    and icto: class-type-of' U' = ⌊C'⌋
    and sees: P ⊢ C' sees M: Ts'→T' = meth in D'
    and wtes: P,E,H ⊢ map Val vs [:] Ts''
    and widens: P ⊢ Ts'' [≤] Ts' by auto
  from wta obtain hU' where Ha: typeof-addr H a = ⌊hU'⌋ U' = ty-of-htype hU' by(auto)
  moreover from ‹typeof-addr (hp s) a = ⌊U⌋› ‹H ◁≤ hp s› Ha
  have [simp]: U = hU' by(auto dest: typeof-addr-hext-mono)
  from wtes have length vs = length Ts''
    by(auto intro: map-eq-imp-length-eq)
  moreover from widens have length Ts'' = length Ts'
    by(auto dest: widens-lengthD)
  moreover from sees icto sees ‹P ⊢ class-type-of U sees M: Ts→T = ⌊(pns, body)⌋ in D› Ha
  have [simp]: meth = ⌊(pns, body)⌋ by(auto dest: sees-method-fun)
```

with *sees wf* **have** *wf-mdecl wf-J-mdecl P D′* (*M*, *Ts′*, *T′*, ⌊(*pns*, *body*)⌋)
by(*auto intro*: *sees-wf-mdecl*)
hence *length pns* = *length Ts′* **by**(*simp add*: *wf-mdecl-def*)
ultimately show *?case* **using** *sees icto*
by(*fastforce intro*: *red-reds.RedCall*)
next
case (*RedCallExternal s a U M Ts T′ D vs ta va h′ ta′ e′ s′*)
from ‹*P*,*E*,*H* ⊢ *addr a*·*M*(*map Val vs*) : *T*›
obtain *U′ C′ Ts′ meth D′ Ts″*
where *wta*: *P*,*E*,*H* ⊢ *addr a* : *U′* **and** *icto*: *class-type-of′ U′* = ⌊*C′*⌋
and *sees*: *P* ⊢ *C′ sees M*: *Ts′*→*T* = *meth in D′*
and *wtvs*: *P*,*E*,*H* ⊢ *map Val vs* [:] *Ts″*
and *sub*: *P* ⊢ *Ts″* [≤] *Ts′* **by** *auto*
from *wta* ‹*typeof-addr* (*hp s*) *a* = ⌊*U*⌋› ‹*hext H* (*hp s*)› **have** [*simp*]: *U′* = *ty-of-htype U*
by(*auto dest*: *typeof-addr-hext-mono*)
with *icto* **have** [*simp*]: *C′* = *class-type-of U* **by**(*auto*)
from *sees* ‹*P* ⊢ *class-type-of U sees M*: *Ts*→*T′* = *Native in D*›
have [*simp*]: *meth* = *Native* **by**(*auto dest*: *sees-method-fun*)
with *wta sees icto wtvs sub* **have** *P*,*H* ⊢ *a*·*M*(*vs*) : *T*
by(*cases U*)(*auto 4 4 simp add*: *external-WT′-iff*)
from *red-external-wt-hconf-hext*[*OF wf* ‹*P*,*t* ⊢ ⟨*a*·*M*(*vs*),*hp s*⟩ −*ta*→*ext* ⟨*va*,*h′*⟩› ‹*H* ◁ *hp s*› *this tconf hconf*]
wta icto sees ‹*ta′* = *convert-extTA extNTA ta*› ‹*e′* = *extRet2J* (*addr a*·*M*(*map Val vs*)) *va*› ‹*s′* = (*h′*, *lcl s*)›
show *?case* **by**(*cases U*)(*auto 4 5 intro*: *red-reds.RedCallExternal simp del*: *split-paired-Ex*)
next
case *RedCallNull* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case (*BlockRed e h l V vo ta e′ h′ l′ T E T′*)
note *IH* = *BlockRed.hyps*(*2*)
from *IH*[*of E*(*V* ↦ *T*) *T′*] ‹*P*,*E*,*H* ⊢ {*V*:*T*=*vo*; *e*} : *T′*› ‹*hext H* (*hp* (*h*, *l*))›
show *?case* **by**(*fastforce dest*: *red-reds.BlockRed*)
next
case *RedBlock* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *SynchronizedRed1* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *SynchronizedNull* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *LockSynchronized* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *SynchronizedRed2* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *UnlockSynchronized* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *SeqRed* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *RedSeq* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *CondRed* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *RedCondT* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *RedCondF* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)

next
case *RedWhile* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *ThrowRed* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *RedThrowNull* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *TryRed* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *RedTry* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case (*RedTryCatch s a D C V e2 E T*)
from ‹*P*,*E*,*H* ⊢ *try Throw a catch*(*C V*) *e2* : *T*›
obtain *T'* **where** *P*,*E*,*H* ⊢ *addr a* : *T'* **by** *auto*
with ‹*typeof-addr* (*hp s*) *a* = ⌊*Class-type D*⌋› ‹*hext H* (*hp s*)›
have *Ha*: *typeof-addr H a* = ⌊*Class-type D*⌋
by(*auto dest*: *typeof-addr-hext-mono*)
with ‹*P* ⊢ *D* ≼* *C*› **show** *?case*
by(*fastforce intro*: *red-reds.RedTryCatch*)
next
case (*RedTryFail s a D C V e2 E T*)
from ‹*P*,*E*,*H* ⊢ *try Throw a catch*(*C V*) *e2* : *T*›
obtain *T'* **where** *P*,*E*,*H* ⊢ *addr a* : *T'* **by** *auto*
with ‹*typeof-addr* (*hp s*) *a* = ⌊*Class-type D*⌋› ‹*hext H* (*hp s*)›
have *Ha*: *typeof-addr H a* = ⌊*Class-type D*⌋
by(*auto dest*: *typeof-addr-hext-mono*)
with ‹¬ *P* ⊢ *D* ≼* *C*› **show** *?case*
by(*fastforce intro*: *red-reds.RedTryFail*)
next
case *ListRed1* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *ListRed2* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *NewArrayThrow* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *CastThrow* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *InstanceOfThrow* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *BinOpThrow1* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *BinOpThrow2* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *LAssThrow* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *AAccThrow1* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *AAccThrow2* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *AAssThrow1* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *AAssThrow2* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *AAssThrow3* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)

next
case *ALengthThrow* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *FAccThrow* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *FAssThrow1* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *FAssThrow2* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *CASThrow* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *CASThrow2* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *CASThrow3* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *CallThrowObj* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *CallThrowParams* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *BlockThrow* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *SynchronizedThrow1* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *SynchronizedThrow2* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *SeqThrow* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *CondThrow* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
next
case *ThrowThrow* **thus** *?case* **by**(*fastforce intro*: *red-reds.intros*)
qed

lemma *can-lock-devreserp*:
⟦ *wf-J-prog P*; *red-mthr.can-sync P t* (*e*, *l*) *h′ L*; *P*,*E*,*h* ⊢ *e* : *T*; *P*,*h* ⊢ *t* √*t*; *hconf h*; *h* ◁ *h′* ⟧
⟹ *red-mthr.can-sync P t* (*e*, *l*) *h L*
apply(*erule red-mthr.can-syncE*)
apply(*clarsimp*)
apply(*drule red-wt-hconf-hext*, *assumption*+)
apply(*simp*)
apply(*fastforce intro*!: *red-mthr.can-syncI*)
done

end

context *J-typesafe* **begin**

lemma *preserve-deadlocked*:
assumes *wf*: *wf-J-prog P*
shows *preserve-deadlocked final-expr* (*mred P*) *convert-RA* ({*s*. *sync-es-ok* (*thr s*) (*shr s*) ∧ *lock-ok* (*locks s*) (*thr s*)} ∩ {*s*. ∃ *Es*. *sconf-type-ts-ok Es* (*thr s*) (*shr s*)} ∩ {*s*. *def-ass-ts-ok* (*thr s*) (*shr s*)})
(**is** *preserve-deadlocked - - - ?wf-state*)
proof(*unfold-locales*)
show *inv*: *invariant3p* (*mredT P*) *?wf-state*
by(*intro invariant3p-IntI invariant3p-sync-es-ok-lock-ok*[*OF wf*] *lifting-inv.invariant3p-ts-inv*[*OF*

lifting-inv-sconf-subject-ok[*OF wf*]] *lifting-wf.invariant3p-ts-ok*[*OF lifting-wf-def-ass*[*OF wf*]])

fix *s t′ ta′ s′ t x ln*
assume *wfs*: *s* ∈ *?wf-state*
and *redT*: *P* ⊢ *s* −*t′*▷*ta′*→ *s′*
and *tst*: *thr s t* = ⌊(*x*, *ln*)⌋
from *redT* **have** *hext*: *shr s* ≤| *shr s′* **by**(*rule redT-hext-incr*)

from *inv redT wfs* **have** *wfs′*: *s′* ∈ *?wf-state* **by**(*rule invariant3pD*)
from *redT tst* **obtain** *x′ ln′* **where** *ts′t*: *thr s′ t*= ⌊(*x′*, *ln′*)⌋
by(*cases thr s′ t*)(*cases s*, *cases s′*, *auto dest*: *red-mthr.redT-thread-not-disappear*)

from *wfs tst* **obtain** *E T* **where** *wt*: *P*,*E*,*shr s* ⊢ *fst x* : *T*
and *hconf*: *hconf* (*shr s*)
and *da*: *D* (*fst x*) ⌊*dom* (*snd x*)⌋
and *tconf*: *P*,*shr s* ⊢ *t* √*t*
by(*force dest*: *ts-invD ts-okD simp add*: *type-ok-def sconf-def sconf-type-ok-def*)
from *wt hext* **have** *wt′*: *P*,*E*,*shr s′* ⊢ *fst x* : *T* **by**(*rule WTrt-hext-mono*)
from *wfs′ ts′t* **have** *hconf′*: *hconf* (*shr s′*)
by(*auto dest*: *ts-invD simp add*: *type-ok-def sconf-def sconf-type-ok-def*)

{
assume *cs*: *red-mthr.must-sync P t x* (*shr s*)
from *cs* **have** ¬ *final* (*fst x*) **by**(*auto elim*!: *red-mthr.must-syncE simp add*: *split-beta*)

from *progress*[*OF wf-prog-wwf-prog*[*OF wf*] *hconf′ wt′ da this*, *of extTA2J P t*]
obtain *e′ h x′ ta* **where** *P*,*t* ⊢ ⟨*fst x*,(*shr s′*, *snd x*)⟩ −*ta*→ ⟨*e′*, (*h*, *x′*)⟩ **by** *auto*
with *red-ta-satisfiable*[*OF this*]
show *red-mthr.must-sync P t x* (*shr s′*)
by−(*rule red-mthr.must-syncI*, *fastforce simp add*: *split-beta*)
next
fix *LT*
assume *red-mthr.can-sync P t x* (*shr s′*) *LT*
with *can-lock-devreserp*[*OF wf - wt tconf hconf hext*, *of snd x LT*]
show ∃ *LT′*⊆*LT*. *red-mthr.can-sync P t x* (*shr s*) *LT′* **by** *auto*
}
qed

end

end

4.16 Program annotation

theory *Annotate*
imports
WellType
begin

abbreviation (**output**)
unanFAcc :: *′addr expr* ⇒ *vname* ⇒ *′addr expr* (‹(-•-)› [*10*,*10*] *90*)
where
unanFAcc e F ≡ *FAcc e F* (*STR ′′•′′*)

abbreviation (**output**)
unanFAss :: *'addr expr ⇒ vname ⇒ 'addr expr ⇒ 'addr expr* (‹(-•- := -)› [*10*,*0*,*90*] *90*)
where
unanFAss e F e' ≡ FAss e F (*STR ''''*) *e'*

definition *array-length-field-name* :: *vname*
where *array-length-field-name = STR ''length''*

notation (**output**) *array-length-field-name* (‹*length*›)

definition *super* :: *vname*
where *super = STR ''super''*

lemma *super-neq-this* [*simp*]: *super ≠ this this ≠ super*
by(*simp-all add*: *this-def super-def*)

inductive *Anno* :: (*ty ⇒ ty ⇒ ty ⇒ bool*) *⇒ 'addr J-prog ⇒ env ⇒ 'addr expr ⇒ 'addr expr ⇒ bool*

(‹-,-,- ⊢ - ⇝ -› [*51*,*51*,*0*,*0*,*51*]*50*)
and *Annos* :: (*ty ⇒ ty ⇒ ty ⇒ bool*) *⇒ 'addr J-prog ⇒ env ⇒ 'addr expr list ⇒ 'addr expr list ⇒ bool*
(‹-,-,- ⊢ - [⇝] -› [*51*,*51*,*0*,*0*,*51*]*50*)
for *is-lub* :: *ty ⇒ ty ⇒ ty ⇒ bool* **and** *P* :: *'addr J-prog*
where
AnnoNew: *is-lub*,*P*,*E* ⊢ *new C* ⇝ *new C*
| *AnnoNewArray*: *is-lub*,*P*,*E* ⊢ *i* ⇝ *i'* ⟹ *is-lub*,*P*,*E* ⊢ *newA T*⌊*i*⌉ ⇝ *newA T*⌊*i'*⌉
| *AnnoCast*: *is-lub*,*P*,*E* ⊢ *e* ⇝ *e'* ⟹ *is-lub*,*P*,*E* ⊢ *Cast C e* ⇝ *Cast C e'*
| *AnnoInstanceOf*: *is-lub*,*P*,*E* ⊢ *e* ⇝ *e'* ⟹ *is-lub*,*P*,*E* ⊢ *e instanceof T* ⇝ *e' instanceof T*
| *AnnoVal*: *is-lub*,*P*,*E* ⊢ *Val v* ⇝ *Val v*
| *AnnoVarVar*: ⟦ *E V* = ⌊*T*⌋; *V ≠ super* ⟧ ⟹ *is-lub*,*P*,*E* ⊢ *Var V* ⇝ *Var V*
| *AnnoVarField*:
— There is no need to handle access of array fields explicitly, because arrays do not implement methods, i.e. *this* is always of a *Class* type.
⟦ *E V* = *None*; *V ≠ super*; *E this* = ⌊*Class C*⌋; *P* ⊢ *C sees V*:*T* (*fm*) *in D* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *Var V* ⇝ *Var this*•*V*{*D*}
| *AnnoBinOp*:
⟦ *is-lub*,*P*,*E* ⊢ *e1* ⇝ *e1'*; *is-lub*,*P*,*E* ⊢ *e2* ⇝ *e2'* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *e1* «*bop*» *e2* ⇝ *e1'* «*bop*» *e2'*
| *AnnoLAssVar*:
⟦ *E V* = ⌊*T*⌋; *V ≠ super*; *is-lub*,*P*,*E* ⊢ *e* ⇝ *e'* ⟧ ⟹ *is-lub*,*P*,*E* ⊢ *V*:=*e* ⇝ *V*:=*e'*
| *AnnoLAssField*:
⟦ *E V* = *None*; *V ≠ super*; *E this* = ⌊*Class C*⌋; *P* ⊢ *C sees V*:*T* (*fm*) *in D*; *is-lub*,*P*,*E* ⊢ *e* ⇝ *e'* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *V*:=*e* ⇝ *Var this*•*V*{*D*} := *e'*
| *AnnoAAcc*:
⟦ *is-lub*,*P*,*E* ⊢ *a* ⇝ *a'*; *is-lub*,*P*,*E* ⊢ *i* ⇝ *i'* ⟧ ⟹ *is-lub*,*P*,*E* ⊢ *a*⌊*i*⌉ ⇝ *a'*⌊*i'*⌉
| *AnnoAAss*:
⟦ *is-lub*,*P*,*E* ⊢ *a* ⇝ *a'*; *is-lub*,*P*,*E* ⊢ *i* ⇝ *i'*; *is-lub*,*P*,*E* ⊢ *e* ⇝ *e'* ⟧ ⟹ *is-lub*,*P*,*E* ⊢ *a*⌊*i*⌉ := *e* ⇝ *a'*⌊*i'*⌉ := *e'*
| *AnnoALength*:
is-lub,*P*,*E* ⊢ *a* ⇝ *a'* ⟹ *is-lub*,*P*,*E* ⊢ *a*•*length* ⇝ *a'*•*length*
| — All arrays implicitly declare a final field called *length* to store the array length, which hides a potential field of the same name in *Object* (cf. JLS 6.4.5). The last premise implements the hiding because field lookup does does not model the implicit declaration.

AnnoFAcc:
⟦ *is-lub*,*P*,*E* ⊢ *e* ⇝ *e′*; *is-lub*,*P*,*E* ⊢ *e′* :: *U*; *class-type-of′ U* = ⌊*C*⌋; *P* ⊢ *C sees F*:*T* (*fm*) *in D*;
is-Array U ⟶ *F* ≠ *array-length-field-name* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *e*·*F*{*STR ″″*} ⇝ *e′*·*F*{*D*}
| *AnnoFAccALength*:
⟦ *is-lub*,*P*,*E* ⊢ *e* ⇝ *e′*; *is-lub*,*P*,*E* ⊢ *e′* :: *T*⌊⌋ ⟧
⟹ *is-lub*,*P*,*E* ⊢ *e*·*array-length-field-name*{*STR ″″*} ⇝ *e′*·*length*
| *AnnoFAccSuper*:
— In class C with super class D, "super" is syntactic sugar for "((D) this)" (cf. JLS, 15.11.2)
⟦ *E this* = ⌊*Class C*⌋; *C* ≠ *Object*; *class P C* = ⌊(*D*, *fs*, *ms*)⌋;
P ⊢ *D sees F*:*T* (*fm*) *in D′* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *Var super*·*F*{*STR ″″*} ⇝ (*Cast* (*Class D*) (*Var this*))·*F*{*D′*}
| *AnnoFAss*:
⟦ *is-lub*,*P*,*E* ⊢ *e1* ⇝ *e1′*; *is-lub*,*P*,*E* ⊢ *e2* ⇝ *e2′*;
is-lub,*P*,*E* ⊢ *e1′* :: *U*; *class-type-of′ U* = ⌊*C*⌋; *P* ⊢ *C sees F*:*T* (*fm*) *in D*;
is-Array U ⟶ *F* ≠ *array-length-field-name* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *e1*·*F*{*STR ″″*} := *e2* ⇝ *e1′*·*F*{*D*} := *e2′*
| *AnnoFAssSuper*:
⟦ *E this* = ⌊*Class C*⌋; *C* ≠ *Object*; *class P C* = ⌊(*D*, *fs*, *ms*)⌋;
P ⊢ *D sees F*:*T* (*fm*) *in D′*; *is-lub*,*P*,*E* ⊢ *e* ⇝ *e′* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *Var super*·*F*{*STR ″″*} := *e* ⇝ (*Cast* (*Class D*) (*Var this*))·*F*{*D′*} := *e′*
| *AnnoCAS*:
⟦ *is-lub*,*P*,*E* ⊢ *e1* ⇝ *e1′*; *is-lub*,*P*,*E* ⊢ *e2* ⇝ *e2′*; *is-lub*,*P*,*E* ⊢ *e3* ⇝ *e3′* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*) ⇝ *e1′*·*compareAndSwap*(*D*·*F*, *e2′*, *e3′*)
| *AnnoCall*:
⟦ *is-lub*,*P*,*E* ⊢ *e* ⇝ *e′*; *is-lub*,*P*,*E* ⊢ *es* [⇝] *es′* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *Call e M es* ⇝ *Call e′ M es′*
| *AnnoBlock*:
is-lub,*P*,*E*(*V* ↦ *T*) ⊢ *e* ⇝ *e′* ⟹ *is-lub*,*P*,*E* ⊢ {*V*:*T*=*vo*; *e*} ⇝ {*V*:*T*=*vo*; *e′*}
| *AnnoSync*:
⟦ *is-lub*,*P*,*E* ⊢ *e1* ⇝ *e1′*; *is-lub*,*P*,*E* ⊢ *e2* ⇝ *e2′* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *sync*(*e1*) *e2* ⇝ *sync*(*e1′*) *e2′*
| *AnnoComp*:
⟦ *is-lub*,*P*,*E* ⊢ *e1* ⇝ *e1′*; *is-lub*,*P*,*E* ⊢ *e2* ⇝ *e2′* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *e1*;;*e2* ⇝ *e1′*;;*e2′*
| *AnnoCond*:
⟦ *is-lub*,*P*,*E* ⊢ *e* ⇝ *e′*; *is-lub*,*P*,*E* ⊢ *e1* ⇝ *e1′*; *is-lub*,*P*,*E* ⊢ *e2* ⇝ *e2′* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *if* (*e*) *e1 else e2* ⇝ *if* (*e′*) *e1′ else e2′*
| *AnnoLoop*:
⟦ *is-lub*,*P*,*E* ⊢ *e* ⇝ *e′*; *is-lub*,*P*,*E* ⊢ *c* ⇝ *c′* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *while* (*e*) *c* ⇝ *while* (*e′*) *c′*
| *AnnoThrow*:
is-lub,*P*,*E* ⊢ *e* ⇝ *e′* ⟹ *is-lub*,*P*,*E* ⊢ *throw e* ⇝ *throw e′*
| *AnnoTry*:
⟦ *is-lub*,*P*,*E* ⊢ *e1* ⇝ *e1′*; *is-lub*,*P*,*E*(*V* ↦ *Class C*) ⊢ *e2* ⇝ *e2′* ⟧
⟹ *is-lub*,*P*,*E* ⊢ *try e1 catch*(*C V*) *e2* ⇝ *try e1′ catch*(*C V*) *e2′*

| *AnnoNil*:
is-lub,*P*,*E* ⊢ [] [⇝] []
| *AnnoCons*:
⟦ *is-lub*,*P*,*E* ⊢ *e* ⇝ *e′*; *is-lub*,*P*,*E* ⊢ *es* [⇝] *es′* ⟧ ⟹ *is-lub*,*P*,*E* ⊢ *e*#*es* [⇝] *e′*#*es′*

inductive-cases *Anno-cases* [*elim*!]:
is-lub′,*P*,*E* ⊢ *new C* ⇝ *e*

is-lub',*P*,*E* ⊢ *newA T*⌊*e*⌋ ↝ *e'*
is-lub',*P*,*E* ⊢ *Cast T e* ↝ *e'*
is-lub',*P*,*E* ⊢ *e instanceof T* ↝ *e'*
is-lub',*P*,*E* ⊢ *Val v* ↝ *e'*
is-lub',*P*,*E* ⊢ *Var V* ↝ *e'*
is-lub',*P*,*E* ⊢ *e1* «*bop*» *e2* ↝ *e'*
is-lub',*P*,*E* ⊢ *V* := *e* ↝ *e'*
is-lub',*P*,*E* ⊢ *e1*⌊*e2*⌉ ↝ *e'*
is-lub',*P*,*E* ⊢ *e1*⌊*e2*⌉ := *e3* ↝ *e'*
is-lub',*P*,*E* ⊢ *e*·*length* ↝ *e'*
is-lub',*P*,*E* ⊢ *e*·*F*{*D*} ↝ *e'*
is-lub',*P*,*E* ⊢ *e1*·*F*{*D*} := *e2* ↝ *e'*
is-lub',*P*,*E* ⊢ *e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*) ↝ *e'*
is-lub',*P*,*E* ⊢ *e*·*M*(*es*) ↝ *e'*
is-lub',*P*,*E* ⊢ {*V*:*T*=*vo*; *e*} ↝ *e'*
is-lub',*P*,*E* ⊢ *sync*(*e1*) *e2* ↝ *e'*
is-lub',*P*,*E* ⊢ *insync*(*a*) *e2* ↝ *e'*
is-lub',*P*,*E* ⊢ *e1*;; *e2* ↝ *e'*
is-lub',*P*,*E* ⊢ *if* (*e*) *e1 else e2* ↝ *e'*
is-lub',*P*,*E* ⊢ *while*(*e1*) *e2* ↝ *e'*
is-lub',*P*,*E* ⊢ *throw e* ↝ *e'*
is-lub',*P*,*E* ⊢ *try e1 catch*(*C V*) *e2* ↝ *e'*

inductive-cases *Annos-cases* [*elim*!]:
is-lub',*P*,*E* ⊢ [] [↝] *es'*
is-lub',*P*,*E* ⊢ *e* # *es* [↝] *es'*

abbreviation *Anno'* :: *'addr J-prog* ⇒ *env* ⇒ *'addr expr* ⇒ *'addr expr* ⇒ *bool* (‹-,- ⊢ - ↝ -› [*51*,*0*,*0*,*51*]*50*)
where *Anno' P* ≡ *Anno* (*TypeRel.is-lub P*) *P*

abbreviation *Annos'* :: *'addr J-prog* ⇒ *env* ⇒ *'addr expr list* ⇒ *'addr expr list* ⇒ *bool* (‹-,- ⊢ - [↝] -› [*51*,*0*,*0*,*51*]*50*)
where *Annos' P* ≡ *Annos* (*TypeRel.is-lub P*) *P*

definition *annotate* :: *'addr J-prog* ⇒ *env* ⇒ *'addr expr* ⇒ *'addr expr*
where *annotate P E e* = *THE-default e* (λ*e'*. *P*,*E* ⊢ *e* ↝ *e'*)

lemma fixes *is-lub* :: *ty* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹⊢ *lub'*((-,/ -)') = -› [*51*,*51*,*51*] *50*)
assumes *is-lub-unique*: ⋀*T1 T2 T3 T4*. ⟦ ⊢ *lub*(*T1*, *T2*) = *T3*; ⊢ *lub*(*T1*, *T2*) = *T4* ⟧ ⟹ *T3* = *T4*
shows *Anno-fun*: ⟦ *is-lub*,*P*,*E* ⊢ *e* ↝ *e'*; *is-lub*,*P*,*E* ⊢ *e* ↝ *e''* ⟧ ⟹ *e'* = *e''*
and *Annos-fun*: ⟦ *is-lub*,*P*,*E* ⊢ *es* [↝] *es'*; *is-lub*,*P*,*E* ⊢ *es* [↝] *es''* ⟧ ⟹ *es'* = *es''*
proof(*induct arbitrary*: *e''* **and** *es''* *rule*: *Anno-Annos.inducts*)
case (*AnnoFAcc E e e' U C F T fm D*)
from ‹*is-lub*,*P*,*E* ⊢ *e*·*F*{*STR ''''*} ↝ *e''*› **show** *?case*
proof(*rule Anno-cases*)
fix *e''' U' C' T' fm' D'*
assume *is-lub*,*P*,*E* ⊢ *e* ↝ *e'''* *is-lub*,*P*,*E* ⊢ *e'''* :: *U'*
and *class-type-of' U'* = ⌊*C'*⌋
and *P* ⊢ *C' sees F*:*T'* (*fm'*) *in D' e''* = *e'''*·*F*{*D'*}
from ‹*is-lub*,*P*,*E* ⊢ *e* ↝ *e'''*› **have** *e'* = *e'''* **by**(*rule AnnoFAcc*)
with ‹*is-lub*,*P*,*E* ⊢ *e'* :: *U*› ‹*is-lub*,*P*,*E* ⊢ *e'''* :: *U'*›
have *U* = *U'* **by**(*auto intro*: *WT-unique is-lub-unique*)

with ‹*class-type-of′ U* = ⌊*C*⌋› ‹*class-type-of′ U′* = ⌊*C′*⌋›
have *C* = *C′* **by**(*auto*)
with ‹*P* ⊢ *C′ sees F*:*T′* (*fm′*) *in D′*› ‹*P* ⊢ *C sees F*:*T* (*fm*) *in D*›
have *D′* = *D* **by**(*auto dest*: *sees-field-fun*)
with ‹*e″* = *e‴*·*F*{*D′*}› ‹*e′* = *e‴*› **show** *?thesis* **by** *simp*
next
fix *e‴ T*
assume *e″* = *e‴*·*length*
and *is-lub*,*P*,*E* ⊢ *e‴* :: *T*⌊⌋
and *is-lub*,*P*,*E* ⊢ *e* ⇝ *e‴*
and *F* = *array-length-field-name*
from ‹*is-lub*,*P*,*E* ⊢ *e* ⇝ *e‴*› **have** *e′* = *e‴* **by**(*rule AnnoFAcc*)
with ‹*is-lub*,*P*,*E* ⊢ *e′* :: *U*› ‹*is-lub*,*P*,*E* ⊢ *e‴* :: *T*⌊⌋› **have** *U* = *T*⌊⌋ **by**(*auto intro*: *WT-unique is-lub-unique*)
with ‹*class-type-of′ U* = ⌊*C*⌋› ‹*is-Array U* ⟶ *F* ≠ *array-length-field-name*›
show *?thesis* **using** ‹*F* = *array-length-field-name*› **by** *simp*
next
fix *C′ D′ fs ms T D″*
assume *E this* = ⌊*Class C′*⌋
and *class P C′* = ⌊(*D′*, *fs*, *ms*)⌋
and *e* = *Var super*
and *e″* = *Cast* (*Class D′*) (*Var this*)·*F*{*D″*}
with ‹*is-lub*,*P*,*E* ⊢ *e* ⇝ *e′*› **have** *False* **by**(*auto*)
thus *?thesis* **..**
qed
next
case *AnnoFAccALength* **thus** *?case* **by**(*fastforce intro*: *WT-unique*[*OF is-lub-unique*])
next
case (*AnnoFAss E e1 e1′ e2 e2′ U C F T fm D*)
from ‹*is-lub*,*P*,*E* ⊢ *e1*·*F*{*STR ‴*}› := *e2* ⇝ *e″*›
show *?case*
proof(*rule Anno-cases*)
fix *e1″ e2″ U′ C′ T′ fm′ D′*
assume *is-lub*,*P*,*E* ⊢ *e1* ⇝ *e1″ is-lub*,*P*,*E* ⊢ *e2* ⇝ *e2″*
and *is-lub*,*P*,*E* ⊢ *e1″* :: *U′* **and** *class-type-of′ U′* = ⌊*C′*⌋
and *P* ⊢ *C′ sees F*:*T′* (*fm′*) *in D′*
and *e″* = *e1″*·*F*{*D′*} := *e2″*
from ‹*is-lub*,*P*,*E* ⊢ *e1* ⇝ *e1″*› **have** *e1′* = *e1″* **by**(*rule AnnoFAss*)
moreover with ‹*is-lub*,*P*,*E* ⊢ *e1′* :: *U*› ‹*is-lub*,*P*,*E* ⊢ *e1″* :: *U′*›
have *U* = *U′* **by**(*auto intro*: *WT-unique is-lub-unique*)
with ‹*class-type-of′ U* = ⌊*C*⌋› ‹*class-type-of′ U′* = ⌊*C′*⌋›
have *C* = *C′* **by**(*auto*)
with ‹*P* ⊢ *C′ sees F*:*T′* (*fm′*) *in D′*› ‹*P* ⊢ *C sees F*:*T* (*fm*) *in D*›
have *D′* = *D* **by**(*auto dest*: *sees-field-fun*)
moreover from ‹*is-lub*,*P*,*E* ⊢ *e2* ⇝ *e2″*› **have** *e2′* = *e2″* **by**(*rule AnnoFAss*)
ultimately show *?thesis* **using** ‹*e″* = *e1″*·*F*{*D′*} := *e2″*› **by** *simp*
next
fix *C′ D′ fs ms T′ fm′ D″ e‴*
assume *e″* = *Cast* (*Class D′*) (*Var this*)·*F*{*D″*} := *e‴*
and *E this* = ⌊*Class C′*⌋
and *class P C′* = ⌊(*D′*, *fs*, *ms*)⌋
and *P* ⊢ *D′ sees F*:*T′* (*fm′*) *in D″*
and *is-lub*,*P*,*E* ⊢ *e2* ⇝ *e‴*
and *e1* = *Var super*

with ‹*is-lub*,*P*,*E* ⊢ *e1* ⇝ *e1'*› **have** *False* **by**(*auto elim*: *Anno-cases*)
thus *?thesis* **..**
qed
qed(*fastforce dest*: *sees-field-fun*)+

4.16.1 Code generation

definition *Anno-code* :: *'addr J-prog* ⇒ *env* ⇒ *'addr expr* ⇒ *'addr expr* ⇒ *bool* (‹-,- ⊢ - ⇝'' -› [*51*,*0*,*0*,*51*]*50*)
where *Anno-code P* = *Anno* (*is-lub-sup P*) *P*

definition *Annos-code* :: *'addr J-prog* ⇒ *env* ⇒ *'addr expr list* ⇒ *'addr expr list* ⇒ *bool* (‹-,- ⊢ - [⇝'] -› [*51*,*0*,*0*,*51*]*50*)
where *Annos-code P* = *Annos* (*is-lub-sup P*) *P*

primrec *block-types* :: (*'a*, *'b*, *'addr*) *exp* ⇒ *ty list*
and *blocks-types* :: (*'a*, *'b*, *'addr*) *exp list* ⇒ *ty list*
where
block-types (*new C*) = []
| *block-types* (*newA T*⌊*e*⌉) = *block-types e*
| *block-types* (*Cast U e*) = *block-types e*
| *block-types* (*e instanceof U*) = *block-types e*
| *block-types* (*e1*«*bop*»*e2*) = *block-types e1* @ *block-types e2*
| *block-types* (*Val v*) = []
| *block-types* (*Var V*) = []
| *block-types* (*V* := *e*) = *block-types e*
| *block-types* (*a*⌊*i*⌉) = *block-types a* @ *block-types i*
| *block-types* (*a*⌊*i*⌉ := *e*) = *block-types a* @ *block-types i* @ *block-types e*
| *block-types* (*a*·*length*) = *block-types a*
| *block-types* (*e*·*F*{*D*}) = *block-types e*
| *block-types* (*e*·*F*{*D*} := *e'*) = *block-types e* @ *block-types e'*
| *block-types* (*e*·*compareAndSwap*(*D*·*F*, *e'*, *e''*)) = *block-types e* @ *block-types e'* @ *block-types e''*
| *block-types* (*e*·*M*(*es*)) = *block-types e* @ *blocks-types es*
| *block-types* {*V*:*T*=*vo*; *e*} = *T* # *block-types e*
| *block-types* (*sync*$_V$(*e*) *e'*) = *block-types e* @ *block-types e'*
| *block-types* (*insync*$_V$(*a*) *e*) = *block-types e*
| *block-types* (*e*;;*e'*) = *block-types e* @ *block-types e'*
| *block-types* (*if* (*e*) *e1 else e2*) = *block-types e* @ *block-types e1* @ *block-types e2*
| *block-types* (*while* (*b*) *c*) = *block-types b* @ *block-types c*
| *block-types* (*throw e*) = *block-types e*
| *block-types* (*try e catch*(*C V*) *e'*) = *block-types e* @ *Class C* # *block-types e'*

| *blocks-types* [] = []
| *blocks-types* (*e*#*es*) = *block-types e* @ *blocks-types es*

lemma fixes *is-lub1* :: *ty* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹⊢*1* *lub'*((-,/ -)') = -› [*51*,*51*,*51*] *50*)
and *is-lub2* :: *ty* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹⊢*2* *lub'*((-,/ -)') = -› [*51*,*51*,*51*] *50*)
assumes *wf*: *wf-prog wf-md P*
and *is-lub1-into-is-lub2*: ⋀*T1 T2 T3*. ⟦ ⊢*1* *lub*(*T1*, *T2*) = *T3*; *is-type P T1*; *is-type P T2* ⟧ ⟹ ⊢*2* *lub*(*T1*, *T2*) = *T3*
and *is-lub2-is-type*: ⋀*T1 T2 T3*. ⟦ ⊢*2* *lub*(*T1*, *T2*) = *T3*; *is-type P T1*; *is-type P T2* ⟧ ⟹ *is-type P T3*
shows *Anno-change-is-lub*:
⟦ *is-lub1*,*P*,*E* ⊢ *e* ⇝ *e'*; *ran E* ∪ *set* (*block-types e*) ⊆ *types P* ⟧ ⟹ *is-lub2*,*P*,*E* ⊢ *e* ⇝ *e'*

and *Annos-change-is-lub*:
⟦ *is-lub1*,*P*,*E* ⊢ *es* [⇝] *es'*; *ran E* ∪ *set* (*blocks-types es*) ⊆ *types P* ⟧ ⟹ *is-lub2*,*P*,*E* ⊢ *es* [⇝] *es'*
proof(*induct rule*: *Anno-Annos.inducts*)
case (*AnnoBlock E V T e e' vo*)
from ‹*ran E* ∪ *set* (*block-types* {*V*:*T*=*vo*; *e*}) ⊆ *types P*›
have *ran* (*E*(*V* ↦ *T*)) ∪ *set* (*block-types e*) ⊆ *types P*
by(*auto simp add*: *ran-def*)
thus *?case* **using** *AnnoBlock* **by**(*blast intro*: *Anno-Annos.intros*)
next
case (*AnnoTry E e1 e1' V C e2 e2'*)
from ‹*ran E* ∪ *set* (*block-types* (*try e1 catch*(*C V*) *e2*)) ⊆ *types P*›
have *ran* (*E*(*V* ↦ *Class C*)) ∪ *set* (*block-types e2*) ⊆ *types P*
by(*auto simp add*: *ran-def*)
thus *?case* **using** *AnnoTry* **by**(*simp del*: *fun-upd-apply*)(*blast intro*: *Anno-Annos.intros*)
qed(*simp-all del*: *is-Array.simps is-Array-conv*, (*blast intro*: *Anno-Annos.intros WT-change-is-lub*[*OF wf*, **where** *?is-lub1.0*=*is-lub1* **and** *?is-lub2.0*=*is-lub2*] *is-lub1-into-is-lub2 is-lub2-is-type*)+)

lemma assumes *wf*: *wf-prog wf-md P*
shows *Anno-into-Anno-code*: ⟦ *P*,*E* ⊢ *e* ⇝ *e'*; *ran E* ∪ *set* (*block-types e*) ⊆ *types P* ⟧ ⟹ *P*,*E* ⊢ *e* ⇝′ *e'*
and *Annos-into-Annos-code*: ⟦ *P*,*E* ⊢ *es* [⇝] *es'*; *ran E* ∪ *set* (*blocks-types es*) ⊆ *types P* ⟧ ⟹ *P*,*E* ⊢ *es* [⇝′] *es'*
proof −
assume *anno*: *P*,*E* ⊢ *e* ⇝ *e'*
and *ran*: *ran E* ∪ *set* (*block-types e*) ⊆ *types P*
show *P*,*E* ⊢ *e* ⇝′ *e'* **unfolding** *Anno-code-def*
by(*rule Anno-change-is-lub*[*OF wf - - anno ran*])(*blast intro*!: *is-lub-sup.intros intro*: *is-lub-subD*[*OF wf*] *sup-is-type*[*OF wf*] *elim*!: *is-lub-sup.cases*)+
next
assume *annos*: *P*,*E* ⊢ *es* [⇝] *es'*
and *ran*: *ran E* ∪ *set* (*blocks-types es*) ⊆ *types P*
show *P*,*E* ⊢ *es* [⇝′] *es'* **unfolding** *Annos-code-def*
by(*rule Annos-change-is-lub*[*OF wf - - annos ran*])(*blast intro*!: *is-lub-sup.intros intro*: *is-lub-subD*[*OF wf*] *sup-is-type*[*OF wf*] *elim*!: *is-lub-sup.cases*)+
qed

lemma assumes *wf*: *wf-prog wf-md P*
shows *Anno-code-into-Anno*: ⟦ *P*,*E* ⊢ *e* ⇝′ *e'*; *ran E* ∪ *set* (*block-types e*) ⊆ *types P* ⟧ ⟹ *P*,*E* ⊢ *e* ⇝ *e'*
and *Annos-code-into-Annos*: ⟦ *P*,*E* ⊢ *es* [⇝′] *es'*; *ran E* ∪ *set* (*blocks-types es*) ⊆ *types P* ⟧ ⟹ *P*,*E* ⊢ *es* [⇝] *es'*
proof −
assume *anno*: *P*,*E* ⊢ *e* ⇝′ *e'*
and *ran*: *ran E* ∪ *set* (*block-types e*) ⊆ *types P*
show *P*,*E* ⊢ *e* ⇝ *e'*
by(*rule Anno-change-is-lub*[*OF wf - - anno*[*unfolded Anno-code-def*] *ran*])(*blast elim*!: *is-lub-sup.cases intro*: *sup-is-lubI*[*OF wf*] *is-lub-is-type*[*OF wf*])+
next
assume *annos*: *P*,*E* ⊢ *es* [⇝′] *es'*
and *ran*: *ran E* ∪ *set* (*blocks-types es*) ⊆ *types P*
show *P*,*E* ⊢ *es* [⇝] *es'*
by(*rule Annos-change-is-lub*[*OF wf - - annos*[*unfolded Annos-code-def*] *ran*])(*blast elim*!: *is-lub-sup.cases intro*: *sup-is-lubI*[*OF wf*] *is-lub-is-type*[*OF wf*])+
qed

lemma fixes *is-lub*
  assumes *wf*: *wf-prog wf-md P*
  shows *WT-block-types-is-type*: *is-lub*,*P*,*E* ⊢ *e* :: *T* ⟹ *set* (*block-types e*) ⊆ *types P*
  and *WTs-blocks-types-is-type*: *is-lub*,*P*,*E* ⊢ *es* [::] *Ts* ⟹ *set* (*blocks-types es*) ⊆ *types P*
apply(*induct rule*: *WT-WTs.inducts*)
apply(*auto intro*: *is-class-sub-Throwable*[*OF wf*])
done

lemma fixes *is-lub*
  shows *Anno-block-types*: *is-lub*,*P*,*E* ⊢ *e* ⇝ *e'* ⟹ *block-types e* = *block-types e'*
  and *Annos-blocks-types*: *is-lub*,*P*,*E* ⊢ *es* [⇝] *es'* ⟹ *blocks-types es* = *blocks-types es'*
by(*induct rule*: *Anno-Annos.inducts*) *auto*

code-pred
  (*modes*: (*i* ⇒ *i* ⇒ *o* ⇒ *bool*) ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
  [*detect-switches*, *skip-proof*]
  Anno
.

definition *annotate-code* :: *'addr J-prog* ⇒ *env* ⇒ *'addr expr* ⇒ *'addr expr*
where *annotate-code P E e* = *THE-default e* (λ*e'*. *P*,*E* ⊢ *e* ⇝' *e'*)

code-pred
  (*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
  [*inductify*]
  Anno-code
.

lemma *eval-Anno-i-i-i-o-conv*:
  Predicate.eval (*Anno-code-i-i-i-o P E e*) = (λ*e'*. *P*,*E* ⊢ *e* ⇝' *e'*)
by(*auto intro*!: *ext intro*: *Anno-code-i-i-i-oI elim*: *Anno-code-i-i-i-oE*)

lemma *annotate-code* [*code*]:
  annotate-code P E e = *Predicate.singleton* (λ-. *Code.abort* (*STR ''annotate''*) (λ-. *e*)) (*Anno-code-i-i-i-o P E e*)
by(*simp add*: *THE-default-def Predicate.singleton-def annotate-code-def eval-Anno-i-i-i-o-conv*)

end
theory *J-Main*
imports
  State
  Deadlocked
  Annotate
begin

end

Chapter 5

Jinja Virtual Machine

5.1 State of the JVM

theory *JVMState*
imports
../Common/Observable-Events
begin

5.1.1 Frame Stack

type-synonym
pc = *nat*

type-synonym
'addr frame = *'addr val list* × *'addr val list* × *cname* × *mname* × *pc*
— operand stack
— registers (including this pointer, method parameters, and local variables)
— name of class where current method is defined
— parameter types
— program counter within frame

print-translation ‹
let
fun tr'
[*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax val*}, -) $ *a1*),
Const (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax val*}, -) $ *a2*)) $
(*Const* (@{*type-syntax prod*}, -) $
Const (@{*type-syntax String.literal*}, -) $
(*Const* (@{*type-syntax prod*}, -) $
Const (@{*type-syntax String.literal*}, -) $
Const (@{*type-syntax nat*}, -)))] =
if a1 = *a2 then Syntax.const* @{*type-syntax frame*} $ *a1*
else raise Match;
in [(@{*type-syntax prod*}, *K tr'*)]
end
›
typ *'addr frame*

5.1.2 Runtime State

type-synonym
(*'addr*, *'heap*) *jvm-state* = *'addr option* × *'heap* × *'addr frame list*
— exception flag, heap, frames

type-synonym
'addr jvm-thread-state = *'addr option* × *'addr frame list*
— exception flag, frames, thread lock state

type-synonym
(*'addr*, *'thread-id*, *'heap*) *jvm-thread-action* = (*'addr*, *'thread-id*, *'addr jvm-thread-state*,*'heap*) *Jinja-thread-action*

type-synonym
(*'addr*, *'thread-id*, *'heap*) *jvm-ta-state* = (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action* × (*'addr*, *'heap*) *jvm-state*

print-translation ‹
let
fun tr'
[*a1*, *t*
, *Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax option*}, -) $ *a2*) $
(*Const* (@{*type-syntax list*}, -) $
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax val*}, -) $ *a3*)) $
(∗ *Next bit*: *same syntax translation as for frame* ∗)
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax val*}, -) $ *a4*)) $
(*Const* (@{*type-syntax prod*}, -) $
Const (@{*type-syntax String.literal*}, -) $
(*Const* (@{*type-syntax prod*}, -) $
Const (@{*type-syntax String.literal*}, -) $
Const (@{*type-syntax nat*}, -))))))
, *h*] =
if a1 = *a2 andalso a2* = *a3 andalso a3* = *a4 then Syntax.const* @{*type-syntax jvm-thread-action*}
$ *a1* $ *t* $ *h*
else raise Match;
in [(@{*type-syntax Jinja-thread-action*}, *K tr'*)]
end
›
typ (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action*

end

5.2 Instructions of the JVM

theory *JVMInstructions*
imports
JVMState
../Common/BinOp
begin

datatype *'addr instr*
  = *Load nat* — load from local variable
  | *Store nat* — store into local variable
  | *Push 'addr val* — push a value (constant)
  | *New cname* — create object
  | *NewArray ty* — create array for elements of given type
  | *ALoad* — Load array element from heap to stack
  | *AStore* — Set element in array
  | *ALength* — Return the length of the array
  | *Getfield vname cname* — Fetch field from object
  | *Putfield vname cname* — Set field in object
  | *CAS vname cname* — Compare-and-swap instruction
  | *Checkcast ty* — Check whether object is of given type
  | *Instanceof ty* — instanceof test
  | *Invoke mname nat* — inv. instance meth of an object
  | *Return* — return from method
  | *Pop* — pop top element from opstack
  | *Dup* — duplicate top stack element
  | *Swap* — swap top stack elements
  | *BinOpInstr bop* — binary operator instruction
  | *Goto int* — goto relative address
  | *IfFalse int* — branch if top of stack false
  | *ThrowExc* — throw top of stack as exception
  | *MEnter* — enter the monitor of object on top of the stack
  | *MExit* — exit the monitor of object on top of the stack

abbreviation *CmpEq* :: *'addr instr*
where *CmpEq* ≡ *BinOpInstr Eq*

abbreviation *CmpLeq* :: *'addr instr*
where *CmpLeq* ≡ *BinOpInstr LessOrEqual*

abbreviation *CmpGeq* :: *'addr instr*
where *CmpGeq* ≡ *BinOpInstr GreaterOrEqual*

abbreviation *CmpLt* :: *'addr instr*
where *CmpLt* ≡ *BinOpInstr LessThan*

abbreviation *CmpGt* :: *'addr instr*
where *CmpGt* ≡ *BinOpInstr GreaterThan*

abbreviation *IAdd* :: *'addr instr*
where *IAdd* ≡ *BinOpInstr Add*

abbreviation *ISub* :: *'addr instr*
where *ISub* ≡ *BinOpInstr Subtract*

abbreviation *IMult* :: *'addr instr*
where *IMult* ≡ *BinOpInstr Mult*

abbreviation *IDiv* :: *'addr instr*
where *IDiv* ≡ *BinOpInstr Div*

abbreviation *IMod* :: *'addr instr*

where *IMod* ≡ *BinOpInstr Mod*

abbreviation *IShl* :: *'addr instr*
where *IShl* ≡ *BinOpInstr ShiftLeft*

abbreviation *IShr* :: *'addr instr*
where *IShr* ≡ *BinOpInstr ShiftRightSigned*

abbreviation *IUShr* :: *'addr instr*
where *IUShr* ≡ *BinOpInstr ShiftRightZeros*

abbreviation *IAnd* :: *'addr instr*
where *IAnd* ≡ *BinOpInstr BinAnd*

abbreviation *IOr* :: *'addr instr*
where *IOr* ≡ *BinOpInstr BinOr*

abbreviation *IXor* :: *'addr instr*
where *IXor* ≡ *BinOpInstr BinXor*

type-synonym
'addr bytecode = *'addr instr list*

type-synonym
ex-entry = *pc* × *pc* × *cname option* × *pc* × *nat*
— start-pc, end-pc, exception type (None = Any), handler-pc, remaining stack depth

type-synonym
ex-table = *ex-entry list*

type-synonym
'addr jvm-method = *nat* × *nat* × *'addr bytecode* × *ex-table*
— max stacksize
— number of local variables. Add 1 + no. of parameters to get no. of registers
— instruction sequence
— exception handler table

type-synonym
'addr jvm-prog = *'addr jvm-method prog*

end

5.3 Abstract heap locales for byte code programs

theory *JVMHeap*
imports
../Common/Conform
JVMInstructions
begin

locale *JVM-heap-base* =
heap-base +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*

and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*

locale *JVM-heap* =
JVM-heap-base +
heap +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *P* :: *'addr jvm-prog*

locale *JVM-heap-conf-base* =
heap-conf-base +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr jvm-prog*

sublocale *JVM-heap-conf-base* < *JVM-heap-base* **.**

locale *JVM-heap-conf-base'* =
JVM-heap-conf-base +
heap +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr jvm-prog*

sublocale *JVM-heap-conf-base'* < *JVM-heap* **by**(*unfold-locales*)

locale *JVM-heap-conf* =
JVM-heap-conf-base' +

heap-conf +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr jvm-prog*

locale *JVM-progress* =
heap-progress +
JVM-heap-conf-base' +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr jvm-prog*

locale *JVM-conf-read* =
heap-conf-read +
JVM-heap-conf +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr jvm-prog*

locale *JVM-typesafe* =
heap-typesafe +
JVM-conf-read +
JVM-progress +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr jvm-prog*

end

5.4 JVM Instruction Semantics

theory *JVMExecInstr*
imports
JVMInstructions
JVMHeap
../Common/ExternalCall
begin

primrec *extRet2JVM* ::
nat ⇒ *'heap* ⇒ *'addr val list* ⇒ *'addr val list* ⇒ *cname* ⇒ *mname* ⇒ *pc* ⇒ *'addr frame list*
⇒ *'addr extCallRet* ⇒ (*'addr*, *'heap*) *jvm-state*
where
extRet2JVM n h stk loc C M pc frs (*RetVal v*) = (*None*, *h*, (*v* # *drop* (*Suc n*) *stk*, *loc*, *C*, *M*, *pc* + *1*) # *frs*)
| *extRet2JVM n h stk loc C M pc frs* (*RetExc a*) = (⌊*a*⌋, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs*)
| *extRet2JVM n h stk loc C M pc frs RetStaySame* = (*None*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs*)

lemma *eq-extRet2JVM-conv* [*simp*]:
(*xcp*, *h'*, *frs'*) = *extRet2JVM n h stk loc C M pc frs va* ⟷
h' = *h* ∧ (*case va of RetVal v* ⇒ *xcp* = *None* ∧ *frs'* = (*v* # *drop* (*Suc n*) *stk*, *loc*, *C*, *M*, *pc* + *1*) # *frs*
| *RetExc a* ⇒ *xcp* = ⌊*a*⌋ ∧ *frs'* = (*stk*, *loc*, *C*, *M*, *pc*) # *frs*
| *RetStaySame* ⇒ *xcp* = *None* ∧ *frs'* = (*stk*, *loc*, *C*, *M*, *pc*) # *frs*)
by(*cases va*) *auto*

definition *extNTA2JVM* :: *'addr jvm-prog* ⇒ (*cname* × *mname* × *'addr*) ⇒ *'addr jvm-thread-state*
where *extNTA2JVM P* ≡ (λ(*C*, *M*, *a*). *let* (*D*,*M'*,*Ts*,*meth*) = *method P C M*; (*mxs*,*mxl0*,*ins*,*xt*) = *the meth*
in (*None*, [([],*Addr a* # *replicate mxl0 undefined-value*, *D*, *M*, *0*)]))

abbreviation *extTA2JVM* ::
'addr jvm-prog ⇒ (*'addr*, *'thread-id*, *'heap*) *external-thread-action* ⇒ (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action*
where *extTA2JVM P* ≡ *convert-extTA* (*extNTA2JVM P*)

context *JVM-heap-base* **begin**

primrec *exec-instr* ::
'addr instr ⇒ *'addr jvm-prog* ⇒ *'thread-id* ⇒ *'heap* ⇒ *'addr val list* ⇒ *'addr val list*
⇒ *cname* ⇒ *mname* ⇒ *pc* ⇒ *'addr frame list* ⇒
(((*'addr*, *'thread-id*, *'heap*) *jvm-thread-action* × (*'addr*, *'heap*) *jvm-state*) *set*
where
exec-instr-Load:
exec-instr (*Load n*) *P t h stk loc* C_0 M_0 *pc frs* =
{(*ε*, (*None*, *h*, ((*loc* ! *n*) # *stk*, *loc*, C_0, M_0, *pc+1*)#*frs*))}

| *exec-instr* (*Store n*) *P t h stk loc* C_0 M_0 *pc frs* =
{(*ε*, (*None*, *h*, (*tl stk*, *loc*[*n*:=*hd stk*], C_0, M_0, *pc+1*)#*frs*))}

| *exec-instr-Push*:

```
 exec-instr (Push v) P t h stk loc C0 M0 pc frs =
     {(ε, (None, h, (v # stk, loc, C0, M0, pc+1)#frs))}

| exec-instr-New:
 exec-instr (New C) P t h stk loc C0 M0 pc frs =
  (let HA = allocate h (Class-type C)
   in if HA = {} then {(ε, ⌊addr-of-sys-xcpt OutOfMemory⌋, h, (stk, loc, C0, M0, pc) # frs)}
      else (λ(h', a). ({|NewHeapElem a (Class-type C)|}, None, h', (Addr a # stk, loc, C0, M0, pc +
1)#frs)) ‘ HA)

| exec-instr-NewArray:
  exec-instr (NewArray T) P t h stk loc C0 M0 pc frs =
  (let si = the-Intg (hd stk);
       i = nat (sint si)
   in (if si <s 0
       then {(ε, ⌊addr-of-sys-xcpt NegativeArraySize⌋, h, (stk, loc, C0, M0, pc) # frs)}
        else let HA = allocate h (Array-type T i)
             in if HA = {} then {(ε, ⌊addr-of-sys-xcpt OutOfMemory⌋, h, (stk, loc, C0, M0, pc) # frs)}
                else (λ(h', a). ({|NewHeapElem a (Array-type T i)|}, None, h', (Addr a # tl stk, loc, C0,
M0, pc + 1) # frs)) ‘ HA))

| exec-instr-ALoad:
  exec-instr ALoad P t h stk loc C0 M0 pc frs =
   (let i = the-Intg (hd stk);
        va = hd (tl stk);
        a = the-Addr va;
        len = alen-of-htype (the (typeof-addr h a))
    in (if va = Null then {(ε, ⌊addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C0, M0, pc) # frs)}
        else if i <s 0 ∨ int len ≤ sint i then
             {(ε, ⌊addr-of-sys-xcpt ArrayIndexOutOfBounds⌋, h, (stk, loc, C0, M0, pc) # frs)}
        else {({|ReadMem a (ACell (nat (sint i))) v|}, None, h, (v # tl (tl stk), loc, C0, M0, pc + 1)
# frs) | v.
              heap-read h a (ACell (nat (sint i))) v }))

| exec-instr-AStore:
  exec-instr AStore P t h stk loc C0 M0 pc frs =
  (let ve = hd stk;
       vi = hd (tl stk);
       va = hd (tl (tl stk))
   in (if va = Null then {(ε, ⌊addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C0, M0, pc) # frs)}
       else (let i = the-Intg vi;
                 idx = nat (sint i);
                 a = the-Addr va;
                 hT = the (typeof-addr h a);
                 T = ty-of-htype hT;
                 len = alen-of-htype hT;
                 U = the (typeof_h ve)
             in (if i <s 0 ∨ int len ≤ sint i then
                      {(ε, ⌊addr-of-sys-xcpt ArrayIndexOutOfBounds⌋, h, (stk, loc, C0, M0, pc) # frs)}
                 else if P ⊢ U ≤ the-Array T then
                   {({|WriteMem a (ACell idx) ve|}, None, h', (tl (tl (tl stk)), loc, C0, M0, pc+1) # frs)
                      | h'. heap-write h a (ACell idx) ve h'}
                 else {(ε, (⌊addr-of-sys-xcpt ArrayStore⌋, h, (stk, loc, C0, M0, pc) # frs))})))))
```

```
| exec-instr-ALength:
  exec-instr ALength P t h stk loc C0 M0 pc frs =
   {(ε, (let va = hd stk
         in if va = Null
            then (⌊addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C0, M0, pc) # frs)
           else (None, h, (Intg (word-of-int (int (alen-of-htype (the (typeof-addr h (the-Addr va)))))) #
tl stk, loc, C0, M0, pc+1) # frs)))}

| exec-instr (Getfield F C) P t h stk loc C0 M0 pc frs =
   (let v = hd stk
    in if v = Null then {(ε, ⌊addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C0, M0, pc) # frs)}
       else let a = the-Addr v
            in {(⦃ReadMem a (CField C F) v'⦄, None, h, (v' # (tl stk), loc, C0, M0, pc + 1) # frs) |
v'.
                heap-read h a (CField C F) v'})

| exec-instr (Putfield F C) P t h stk loc C0 M0 pc frs =
  (let v = hd stk;
       r = hd (tl stk)
   in if r = Null then {(ε, ⌊addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C0, M0, pc) # frs)}
      else let a = the-Addr r
           in {(⦃WriteMem a (CField C F) v⦄, None, h', (tl (tl stk), loc, C0, M0, pc + 1) # frs) | h'.
               heap-write h a (CField C F) v h'})

| exec-instr (CAS F C) P t h stk loc C0 M0 pc frs =
  (let v'' = hd stk; v' = hd (tl stk); v = hd (tl (tl stk))
   in if v = Null then {(ε, ⌊addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C0, M0, pc) # frs)}
      else let a = the-Addr v
           in {(⦃ReadMem a (CField C F) v', WriteMem a (CField C F) v''⦄, None, h', (Bool True #
tl (tl (tl stk)), loc, C0, M0, pc + 1) # frs) | h' .
                heap-read h a (CField C F) v' ∧ heap-write h a (CField C F) v'' h'} ∪
               {(⦃ReadMem a (CField C F) v''⦄, None, h, (Bool False # tl (tl (tl stk)), loc, C0, M0, pc
+ 1) # frs) | v''.
                heap-read h a (CField C F) v'' ∧ v'' ≠ v'})

| exec-instr (Checkcast T) P t h stk loc C0 M0 pc frs =
  {(ε, let U = the (typeof_h (hd stk))
       in if P ⊢ U ≤ T then (None, h, (stk, loc, C0, M0, pc + 1) # frs)
          else (⌊addr-of-sys-xcpt ClassCast⌋, h, (stk, loc, C0, M0, pc) # frs))}

| exec-instr (Instanceof T) P t h stk loc C0 M0 pc frs =
  {(ε, None, h, (Bool (hd stk ≠ Null ∧ P ⊢ the (typeof_h (hd stk)) ≤ T) # tl stk, loc, C0, M0, pc +
1) # frs)}

| exec-instr-Invoke:
 exec-instr (Invoke M n) P t h stk loc C0 M0 pc frs =
  (let ps = rev (take n stk);
       r = stk ! n;
       a = the-Addr r;
       T = the (typeof-addr h a)
   in (if r = Null then {(ε, ⌊addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C0, M0, pc) # frs)}
       else
         let C = class-type-of T;
             (D,M',Ts,meth)= method P C M
```

```
        in case meth of
             Native ⇒
             {(extTA2JVM P ta, extRet2JVM n h' stk loc C0 M0 pc frs va) | ta va h'.
              (ta, va, h') ∈ red-external-aggr P t a M ps h}
           | ⌊(mxs,mxl₀,ins,xt)⌋ ⇒
             let f' = ([],[r]@ps@(replicate mxl₀ undefined-value),D,M,0)
             in {(ε, None, h, f' # (stk, loc, C0, M0, pc) # frs)}))

| exec-instr Return P t h stk₀ loc₀ C₀ M₀ pc frs =
  {(ε, (if frs=[] then (None, h, []) else
       let v = hd stk₀;
           (stk,loc,C,m,pc) = hd frs;
           n = length (fst (snd (method P C₀ M₀)))
       in (None, h, (v#(drop (n+1) stk),loc,C,m,pc+1)#tl frs)) )}

| exec-instr Pop P t h stk loc C₀ M₀ pc frs =
     {(ε, (None, h, (tl stk, loc, C₀, M₀, pc+1)#frs) )}

| exec-instr Dup P t h stk loc C₀ M₀ pc frs =
     {(ε, (None, h, (hd stk # stk, loc, C₀, M₀, pc+1)#frs) )}

| exec-instr Swap P t h stk loc C₀ M₀ pc frs =
     {(ε, (None, h, (hd (tl stk) # hd stk # tl (tl stk), loc, C₀, M₀, pc+1)#frs) )}

| exec-instr (BinOpInstr bop) P t h stk loc C0 M0 pc frs =
  {(ε,
   case the (binop bop (hd (tl stk)) (hd stk)) of
     Inl v ⇒ (None, h, (v # tl (tl stk), loc, C0, M0, pc+1) # frs)
   | Inr a ⇒ (Some a, h, (stk, loc, C0, M0, pc) # frs))}

| exec-instr (IfFalse i) P t h stk loc C₀ M₀ pc frs =
  {(ε, (let pc' = if hd stk = Bool False then nat(int pc+i) else pc+1
        in (None, h, (tl stk, loc, C₀, M₀, pc')#frs)) )}

| exec-instr-Goto:
 exec-instr (Goto i) P t h stk loc C₀ M₀ pc frs =
     {(ε, (None, h, (stk, loc, C₀, M₀, nat(int pc+i))#frs) )}

| exec-instr ThrowExc P t h stk loc C₀ M₀ pc frs =
  {(ε, (let xp' = if hd stk = Null then ⌊addr-of-sys-xcpt NullPointer⌋ else ⌊the-Addr(hd stk)⌋
        in (xp', h, (stk, loc, C₀, M₀, pc)#frs)) )}

| exec-instr-MEnter:
 exec-instr MEnter P t h stk loc C₀ M₀ pc frs =
  {let v = hd stk
   in if v = Null
      then (ε, ⌊addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C₀, M₀, pc) # frs)
      else ({|Lock→the-Addr v, SyncLock (the-Addr v)|}, None, h, (tl stk, loc, C₀, M₀, pc + 1) # frs)}

| exec-instr-MExit:
 exec-instr MExit P t h stk loc C₀ M₀ pc frs =
  (let v = hd stk
   in if v = Null
      then {(ε, ⌊addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C₀, M₀, pc)#frs)}
```

else {(⦃*Unlock*→*the-Addr v*, *SyncUnlock* (*the-Addr v*)⦄, *None*, *h*, (*tl stk*, *loc*, C_0, M_0, *pc* + *1*) # *frs*),
(⦃*UnlockFail*→*the-Addr v*⦄, ⌊*addr-of-sys-xcpt IllegalMonitorState*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*) # *frs*)})

end

end

5.5 Exception handling in the JVM

theory *JVMExceptions*
imports
JVMInstructions
begin

abbreviation *Any* :: *cname option*
where *Any* ≡ *None*

definition *matches-ex-entry* :: *'m prog* ⇒ *cname* ⇒ *pc* ⇒ *ex-entry* ⇒ *bool*
where
matches-ex-entry P C pc xcp ≡
let (*s*, *e*, *C'*, *h*, *d*) = *xcp in*
s ≤ *pc* ∧ *pc* < *e* ∧ (*case C' of None* ⇒ *True* | ⌊*C''*⌋ ⇒ *P* ⊢ *C* ≼* *C''*)

primrec
match-ex-table :: *'m prog* ⇒ *cname* ⇒ *pc* ⇒ *ex-table* ⇒ (*pc* × *nat*) *option*
where
match-ex-table P C pc [] = *None*
| *match-ex-table P C pc* (*e*#*es*) = (*if matches-ex-entry P C pc e*
then Some (*snd*(*snd*(*snd e*)))
else match-ex-table P C pc es)

abbreviation *ex-table-of* :: *'addr jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *ex-table*
where *ex-table-of P C M* == *snd* (*snd* (*snd* (*the* (*snd* (*snd* (*snd*(*method P C M*)))))))

lemma *match-ex-table-SomeD*:
match-ex-table P C pc xt = *Some* (*pc'*,*d'*) ⟹
∃(*f*,*t*,*D*,*h*,*d*) ∈ *set xt*. *matches-ex-entry P C pc* (*f*,*t*,*D*,*h*,*d*) ∧ *h* = *pc'* ∧ *d*=*d'*
by (*induct xt*) (*auto split*: *if-split-asm*)

end

5.6 Program Execution in the JVM

theory *JVMExec*
imports
JVMExecInstr
JVMExceptions
../Common/StartConfig
begin

abbreviation *instrs-of* :: *'addr jvm-prog ⇒ cname ⇒ mname ⇒ 'addr instr list*
where *instrs-of P C M == fst(snd(snd(the(snd(snd(snd(method P C M)))))))*

5.6.1 single step execution

context *JVM-heap-base* **begin**

fun *exception-step* :: *'addr jvm-prog ⇒ 'addr ⇒ 'heap ⇒ 'addr frame ⇒ 'addr frame list ⇒ ('addr, 'heap) jvm-state*
where
exception-step P a h (stk, loc, C, M, pc) frs =
(case match-ex-table P (cname-of h a) pc (ex-table-of P C M) of
None ⇒ (⌊a⌋, h, frs)
| Some (pc', d) ⇒ (None, h, (Addr a # drop (size stk − d) stk, loc, C, M, pc') # frs))

lemma *exception-step-def-raw*:
exception-step =
(λP a h (stk, loc, C, M, pc) frs.
case match-ex-table P (cname-of h a) pc (ex-table-of P C M) of
None ⇒ (⌊a⌋, h, frs)
| Some (pc', d) ⇒ (None, h, (Addr a # drop (size stk − d) stk, loc, C, M, pc') # frs))
by(*intro ext*) *auto*

fun *exec* :: *'addr jvm-prog ⇒ 'thread-id ⇒ ('addr, 'heap) jvm-state ⇒ ('addr, 'thread-id, 'heap) jvm-ta-state set* **where**
exec P t (xcp, h, []) = {}
| exec P t (None, h, (stk, loc, C, M, pc) # frs) = exec-instr (instrs-of P C M ! pc) P t h stk loc C M pc frs
| exec P t (⌊a⌋, h, fr # frs) = {(ε, exception-step P a h fr frs)}

5.6.2 relational view

inductive *exec-1* ::
'addr jvm-prog ⇒ 'thread-id ⇒ ('addr, 'heap) jvm-state
⇒ ('addr, 'thread-id, 'heap) jvm-thread-action ⇒ ('addr, 'heap) jvm-state ⇒ bool
(‹-,- ⊢/ - −-−jvm→/ -› [61,0,61,0,61] 60)
for *P* :: *'addr jvm-prog* **and** *t* :: *'thread-id*
where
exec-1I:
(ta, σ') ∈ exec P t σ ⟹ P,t ⊢ σ −ta−jvm→ σ'

lemma *exec-1-iff*:
P,t ⊢ σ −ta−jvm→ σ' ⟷ (ta, σ') ∈ exec P t σ
by(*auto intro*: *exec-1I elim*: *exec-1.cases*)

end

The start configuration of the JVM: in the start heap, we call a method m of class C in program P with parameters *vs*. The *this* pointer of the frame is set to *Null* to simulate a static method invokation.

abbreviation *JVM-local-start* ::
cname ⇒ mname ⇒ ty list ⇒ ty ⇒ 'addr jvm-method ⇒ 'addr val list
⇒ 'addr jvm-thread-state
where

JVM-local-start ≡
λ*C M Ts T* (*mxs*, *mxl0*, *b*) *vs*.
(*None*, [([], *Null* # *vs* @ *replicate mxl0 undefined-value*, *C*, *M*, *0*)])

context *JVM-heap-base* **begin**

abbreviation *JVM-start-state* ::
'addr jvm-prog ⇒ *cname* ⇒ *mname* ⇒ *'addr val list* ⇒ (*'addr*,*'thread-id*,*'addr jvm-thread-state*,*'heap*,*'addr*) *state*
where
JVM-start-state ≡ *start-state JVM-local-start*

definition *JVM-start-state'* :: *'addr jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *'addr val list* ⇒ (*'addr*, *'heap*) *jvm-state*
where
JVM-start-state' P C M vs ≡
let (*D*, *Ts*, *T*, *meth*) = *method P C M*;
(*mxs*, *mxl0*, *ins*, *xt*) = *the meth*
in (*None*, *start-heap*, [([], *Null* # *vs* @ *replicate mxl0 undefined-value*, *D*, *M*, *0*)])

end

end

5.7 A Defensive JVM

theory *JVMDefensive*
imports *JVMExec ../Common/ExternalCallWF*
begin

Extend the state space by one element indicating a type error (or other abnormal termination)

datatype *'a type-error* = *TypeError* | *Normal 'a*

context *JVM-heap-base* **begin**

definition *is-Array-ref* :: *'addr val* ⇒ *'heap* ⇒ *bool* **where**
is-Array-ref v h ≡
is-Ref v ∧
(*v* ≠ *Null* ⟶ *typeof-addr h* (*the-Addr v*) ≠ *None* ∧ *is-Array* (*ty-of-htype* (*the* (*typeof-addr h* (*the-Addr v*)))))

declare *is-Array-ref-def*[*simp*]

primrec *check-instr* :: [*'addr instr*, *'addr jvm-prog*, *'heap*, *'addr val list*, *'addr val list*, *cname*, *mname*, *pc*, *'addr frame list*] ⇒ *bool*
where
check-instr-Load:
check-instr (*Load n*) *P h stk loc C M*$_0$ *pc frs* =
(*n* < *length loc*)

| *check-instr-Store*:
check-instr (*Store n*) *P h stk loc C*$_0$ *M*$_0$ *pc frs* =

$(0 < \mathit{length}\ \mathit{stk} \wedge n < \mathit{length}\ \mathit{loc})$

| *check-instr-Push*:
$\mathit{check\text{-}instr}\ (\mathit{Push}\ v)\ P\ h\ \mathit{stk}\ \mathit{loc}\ C_0\ M_0\ \mathit{pc}\ \mathit{frs} =$
$(\neg \mathit{is\text{-}Addr}\ v)$

| *check-instr-New*:
$\mathit{check\text{-}instr}\ (\mathit{New}\ C)\ P\ h\ \mathit{stk}\ \mathit{loc}\ C_0\ M_0\ \mathit{pc}\ \mathit{frs} =$
$\mathit{is\text{-}class}\ P\ C$

| *check-instr-NewArray*:
$\mathit{check\text{-}instr}\ (\mathit{NewArray}\ T)\ P\ h\ \mathit{stk}\ \mathit{loc}\ C0\ M0\ \mathit{pc}\ \mathit{frs} =$
$(\mathit{is\text{-}type}\ P\ (T\lfloor\rceil) \wedge 0 < \mathit{length}\ \mathit{stk} \wedge \mathit{is\text{-}Intg}\ (\mathit{hd}\ \mathit{stk}))$

| *check-instr-ALoad*:
$\mathit{check\text{-}instr}\ \mathit{ALoad}\ P\ h\ \mathit{stk}\ \mathit{loc}\ C0\ M0\ \mathit{pc}\ \mathit{frs} =$
$(1 < \mathit{length}\ \mathit{stk} \wedge \mathit{is\text{-}Intg}\ (\mathit{hd}\ \mathit{stk}) \wedge \mathit{is\text{-}Array\text{-}ref}\ (\mathit{hd}\ (\mathit{tl}\ \mathit{stk}))\ h)$

| *check-instr-AStore*:
$\mathit{check\text{-}instr}\ \mathit{AStore}\ P\ h\ \mathit{stk}\ \mathit{loc}\ C0\ M0\ \mathit{pc}\ \mathit{frs} =$
$(2 < \mathit{length}\ \mathit{stk} \wedge \mathit{is\text{-}Intg}\ (\mathit{hd}\ (\mathit{tl}\ \mathit{stk})) \wedge \mathit{is\text{-}Array\text{-}ref}\ (\mathit{hd}\ (\mathit{tl}\ (\mathit{tl}\ \mathit{stk})))\ h \wedge \mathit{typeof}_h\ (\mathit{hd}\ \mathit{stk}) \neq \mathit{None})$

| *check-instr-ALength*:
$\mathit{check\text{-}instr}\ \mathit{ALength}\ P\ h\ \mathit{stk}\ \mathit{loc}\ C0\ M0\ \mathit{pc}\ \mathit{frs} =$
$(0 < \mathit{length}\ \mathit{stk} \wedge \mathit{is\text{-}Array\text{-}ref}\ (\mathit{hd}\ \mathit{stk})\ h)$

| *check-instr-Getfield*:
$\mathit{check\text{-}instr}\ (\mathit{Getfield}\ F\ C)\ P\ h\ \mathit{stk}\ \mathit{loc}\ C_0\ M_0\ \mathit{pc}\ \mathit{frs} =$
$(0 < \mathit{length}\ \mathit{stk} \wedge (\exists\, C'\ T\ \mathit{fm}.\ P \vdash C\ \mathit{sees}\ F{:}T\ (\mathit{fm})\ \mathit{in}\ C') \wedge$
$(\mathit{let}\ (C',\ T,\ \mathit{fm}) = \mathit{field}\ P\ C\ F;\ \mathit{ref} = \mathit{hd}\ \mathit{stk}\ \mathit{in}$
$C' = C \wedge \mathit{is\text{-}Ref}\ \mathit{ref} \wedge (\mathit{ref} \neq \mathit{Null} \longrightarrow$
$(\exists\, T.\ \mathit{typeof\text{-}addr}\ h\ (\mathit{the\text{-}Addr}\ \mathit{ref}) = \lfloor T \rfloor \wedge P \vdash \mathit{class\text{-}type\text{-}of}\ T \preceq^* C))))$

| *check-instr-Putfield*:
$\mathit{check\text{-}instr}\ (\mathit{Putfield}\ F\ C)\ P\ h\ \mathit{stk}\ \mathit{loc}\ C_0\ M_0\ \mathit{pc}\ \mathit{frs} =$
$(1 < \mathit{length}\ \mathit{stk} \wedge (\exists\, C'\ T\ \mathit{fm}.\ P \vdash C\ \mathit{sees}\ F{:}T\ (\mathit{fm})\ \mathit{in}\ C') \wedge$
$(\mathit{let}\ (C',\ T,\ \mathit{fm}) = \mathit{field}\ P\ C\ F;\ v = \mathit{hd}\ \mathit{stk};\ \mathit{ref} = \mathit{hd}\ (\mathit{tl}\ \mathit{stk})\ \mathit{in}$
$C' = C \wedge \mathit{is\text{-}Ref}\ \mathit{ref} \wedge (\mathit{ref} \neq \mathit{Null} \longrightarrow$
$(\exists\, T'.\ \mathit{typeof\text{-}addr}\ h\ (\mathit{the\text{-}Addr}\ \mathit{ref}) = \lfloor T' \rfloor \wedge P \vdash \mathit{class\text{-}type\text{-}of}\ T' \preceq^* C \wedge P,h \vdash v :\leq T))))$

| *check-instr-CAS*:
$\mathit{check\text{-}instr}\ (\mathit{CAS}\ F\ C)\ P\ h\ \mathit{stk}\ \mathit{loc}\ C0\ M0\ \mathit{pc}\ \mathit{frs} =$
$(2 < \mathit{length}\ \mathit{stk} \wedge (\exists\, C'\ T\ \mathit{fm}.\ P \vdash C\ \mathit{sees}\ F{:}T\ (\mathit{fm})\ \mathit{in}\ C') \wedge$
$(\mathit{let}\ (C',\ T,\ \mathit{fm}) = \mathit{field}\ P\ C\ F;\ v'' = \mathit{hd}\ \mathit{stk};\ v' = \mathit{hd}\ (\mathit{tl}\ \mathit{stk});\ v = \mathit{hd}\ (\mathit{tl}\ (\mathit{tl}\ \mathit{stk}))\ \mathit{in}$
$C' = C \wedge \mathit{is\text{-}Ref}\ v \wedge \mathit{volatile}\ \mathit{fm} \wedge (v \neq \mathit{Null} \longrightarrow$
$(\exists\, T'.\ \mathit{typeof\text{-}addr}\ h\ (\mathit{the\text{-}Addr}\ v) = \lfloor T' \rfloor \wedge P \vdash \mathit{class\text{-}type\text{-}of}\ T' \preceq^* C \wedge P,h \vdash v' :\leq T \wedge P,h \vdash v'' :\leq T))))$

| *check-instr-Checkcast*:
$\mathit{check\text{-}instr}\ (\mathit{Checkcast}\ T)\ P\ h\ \mathit{stk}\ \mathit{loc}\ C_0\ M_0\ \mathit{pc}\ \mathit{frs} =$
$(0 < \mathit{length}\ \mathit{stk} \wedge \mathit{is\text{-}type}\ P\ T)$

| *check-instr-Instanceof*:
$\mathit{check\text{-}instr}\ (\mathit{Instanceof}\ T)\ P\ h\ \mathit{stk}\ \mathit{loc}\ C_0\ M_0\ \mathit{pc}\ \mathit{frs} =$

$(0 < \mathit{length}\ \mathit{stk} \wedge \mathit{is\text{-}type}\ P\ T \wedge \mathit{is\text{-}Ref}\ (\mathit{hd}\ \mathit{stk}))$

| *check-instr-Invoke*:
$\mathit{check\text{-}instr}\ (\mathit{Invoke}\ M\ n)\ P\ h\ \mathit{stk}\ \mathit{loc}\ C_0\ M_0\ \mathit{pc}\ \mathit{frs} =$
$(n < \mathit{length}\ \mathit{stk} \wedge \mathit{is\text{-}Ref}\ (\mathit{stk}!n) \wedge$
$(\mathit{stk}!n \neq \mathit{Null} \longrightarrow$
$(\mathit{let}\ a = \mathit{the\text{-}Addr}\ (\mathit{stk}!n);$
$T = \mathit{the}\ (\mathit{typeof\text{-}addr}\ h\ a);$
$C = \mathit{class\text{-}type\text{-}of}\ T;$
$(D,\ Ts,\ Tr,\ \mathit{meth}) = \mathit{method}\ P\ C\ M$
$\mathit{in}\ \mathit{typeof\text{-}addr}\ h\ a \neq \mathit{None} \wedge P \vdash C\ \mathit{has}\ M \wedge$
$P,h \vdash \mathit{rev}\ (\mathit{take}\ n\ \mathit{stk})\ [:\leq]\ Ts \wedge$
$(\mathit{meth} = \mathit{None} \longrightarrow D\cdot M(Ts) :: Tr))))$

| *check-instr-Return*:
$\mathit{check\text{-}instr}\ \mathit{Return}\ P\ h\ \mathit{stk}\ \mathit{loc}\ C_0\ M_0\ \mathit{pc}\ \mathit{frs} =$
$(0 < \mathit{length}\ \mathit{stk} \wedge ((0 < \mathit{length}\ \mathit{frs}) \longrightarrow$
$(P \vdash C_0\ \mathit{has}\ M_0) \wedge$
$(\mathit{let}\ v = \mathit{hd}\ \mathit{stk};$
$T = \mathit{fst}\ (\mathit{snd}\ (\mathit{snd}\ (\mathit{method}\ P\ C_0\ M_0)))$
$\mathit{in}\ P,h \vdash v :\leq T)))$

| *check-instr-Pop*:
$\mathit{check\text{-}instr}\ \mathit{Pop}\ P\ h\ \mathit{stk}\ \mathit{loc}\ C_0\ M_0\ \mathit{pc}\ \mathit{frs} =$
$(0 < \mathit{length}\ \mathit{stk})$

| *check-instr-Dup*:
$\mathit{check\text{-}instr}\ \mathit{Dup}\ P\ h\ \mathit{stk}\ \mathit{loc}\ C_0\ M_0\ \mathit{pc}\ \mathit{frs} =$
$(0 < \mathit{length}\ \mathit{stk})$

| *check-instr-Swap*:
$\mathit{check\text{-}instr}\ \mathit{Swap}\ P\ h\ \mathit{stk}\ \mathit{loc}\ C_0\ M_0\ \mathit{pc}\ \mathit{frs} =$
$(1 < \mathit{length}\ \mathit{stk})$

| *check-instr-BinOpInstr*:
$\mathit{check\text{-}instr}\ (\mathit{BinOpInstr}\ \mathit{bop})\ P\ h\ \mathit{stk}\ \mathit{loc}\ \mathit{C0}\ \mathit{M0}\ \mathit{pc}\ \mathit{frs} =$
$(1 < \mathit{length}\ \mathit{stk} \wedge (\exists\, \mathit{T1}\ \mathit{T2}\ T.\ \mathit{typeof}_h\ (\mathit{hd}\ \mathit{stk}) = \lfloor \mathit{T2} \rfloor \wedge \mathit{typeof}_h\ (\mathit{hd}\ (\mathit{tl}\ \mathit{stk})) = \lfloor \mathit{T1} \rfloor \wedge P \vdash \mathit{T1}\,\mathit{«bop»}\,\mathit{T2} : T))$

| *check-instr-IfFalse*:
$\mathit{check\text{-}instr}\ (\mathit{IfFalse}\ b)\ P\ h\ \mathit{stk}\ \mathit{loc}\ C_0\ M_0\ \mathit{pc}\ \mathit{frs} =$
$(0 < \mathit{length}\ \mathit{stk} \wedge \mathit{is\text{-}Bool}\ (\mathit{hd}\ \mathit{stk}) \wedge 0 \leq \mathit{int}\ \mathit{pc}{+}b)$

| *check-instr-Goto*:
$\mathit{check\text{-}instr}\ (\mathit{Goto}\ b)\ P\ h\ \mathit{stk}\ \mathit{loc}\ C_0\ M_0\ \mathit{pc}\ \mathit{frs} =$
$(0 \leq \mathit{int}\ \mathit{pc}{+}b)$

| *check-instr-Throw*:
$\mathit{check\text{-}instr}\ \mathit{ThrowExc}\ P\ h\ \mathit{stk}\ \mathit{loc}\ C_0\ M_0\ \mathit{pc}\ \mathit{frs} =$
$(0 < \mathit{length}\ \mathit{stk} \wedge \mathit{is\text{-}Ref}\ (\mathit{hd}\ \mathit{stk}) \wedge P \vdash \mathit{the}\ (\mathit{typeof}_h\ (\mathit{hd}\ \mathit{stk})) \leq \mathit{Class}\ \mathit{Throwable})$

| *check-instr-MEnter*:
$\mathit{check\text{-}instr}\ \mathit{MEnter}\ P\ h\ \mathit{stk}\ \mathit{loc}\ C_0\ M_0\ \mathit{pc}\ \mathit{frs} =$
$(0 < \mathit{length}\ \mathit{stk} \wedge \mathit{is\text{-}Ref}\ (\mathit{hd}\ \mathit{stk}))$

| *check-instr-MExit*:
check-instr MExit P h stk loc C_0 M_0 *pc frs* =
(*0* < *length stk* ∧ *is-Ref* (*hd stk*))

definition *check-xcpt* :: *'addr jvm-prog* ⇒ *'heap* ⇒ *nat* ⇒ *pc* ⇒ *ex-table* ⇒ *'addr* ⇒ *bool*
where
check-xcpt P h n pc xt a ⟷
(∃ *C*. *typeof-addr h a* = ⌊*Class-type C*⌋ ∧
(*case match-ex-table P C pc xt of None* ⇒ *True* | *Some* (*pc'*, *d'*) ⇒ *d'* ≤ *n*))

definition *check* :: *'addr jvm-prog* ⇒ (*'addr*, *'heap*) *jvm-state* ⇒ *bool*
where
check P σ ≡ *let* (*xcpt*, *h*, *frs*) = *σ in*
(*case frs of* [] ⇒ *True* | (*stk*,*loc*,*C*,*M*,*pc*)#*frs'* ⇒
P ⊢ *C has M* ∧
(*let* (*C'*,*Ts*,*T*,*meth*) = *method P C M*; (*mxs*,*mxl*$_0$,*ins*,*xt*) = *the meth*; *i* = *ins*!*pc in*
meth ≠ *None* ∧ *pc* < *size ins* ∧ *size stk* ≤ *mxs* ∧
(*case xcpt of None* ⇒ *check-instr i P h stk loc C M pc frs'*
| *Some a* ⇒ *check-xcpt P h* (*length stk*) *pc xt a*)))

definition *exec-d* ::
'addr jvm-prog ⇒ *'thread-id* ⇒ (*'addr*, *'heap*) *jvm-state* ⇒ (*'addr*, *'thread-id*, *'heap*) *jvm-ta-state set type-error*
where
exec-d P t σ ≡ *if check P σ then Normal* (*exec P t σ*) *else TypeError*

inductive
exec-1-d ::
'addr jvm-prog ⇒ *'thread-id* ⇒ (*'addr*, *'heap*) *jvm-state type-error*
⇒ (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action* ⇒ (*'addr*, *'heap*) *jvm-state type-error* ⇒ *bool*
(‹-,- ⊢ - −−−*jvmd*→ -› [*61*,*0*,*61*,*0*,*61*] *60*)
for *P* :: *'addr jvm-prog* **and** *t* :: *'thread-id*
where
exec-1-d-ErrorI: *exec-d P t σ* = *TypeError* ⟹ *P*,*t* ⊢ *Normal σ* −*ε*−*jvmd*→ *TypeError*
| *exec-1-d-NormalI*: ⟦ *exec-d P t σ* = *Normal* Σ; (*tas*, *σ'*) ∈ Σ ⟧ ⟹ *P*,*t* ⊢ *Normal σ* −*tas*−*jvmd*→ *Normal σ'*

lemma *jvmd-NormalD*:
P,*t* ⊢ *Normal σ* −*ta*−*jvmd*→ *Normal σ'* ⟹ *check P σ* ∧ (*ta*, *σ'*) ∈ *exec P t σ* ∧ (∃ *xcp h f frs*. *σ* = (*xcp*, *h*, *f* # *frs*))
apply(*erule exec-1-d.cases*, *auto simp add*: *exec-d-def split*: *if-split-asm*)
apply(*case-tac b*, *auto*)
done

lemma *jvmd-NormalE*:
assumes *P*,*t* ⊢ *Normal σ* −*ta*−*jvmd*→ *Normal σ'*
obtains *xcp h f frs* **where** *check P σ* (*ta*, *σ'*) ∈ *exec P t σ σ* = (*xcp*, *h*, *f* # *frs*)
using *assms*
by(*auto dest*: *jvmd-NormalD*)

lemma *exec-d-eq-TypeError*: *exec-d P t σ* = *TypeError* ⟷ ¬ *check P σ*
by(*simp add*: *exec-d-def*)

lemma *exec-d-eq-Normal*: *exec-d P t σ = Normal (exec P t σ) ⟷ check P σ*
by(*auto simp add*: *exec-d-def*)

end

declare *split-paired-All* [*simp del*]
declare *split-paired-Ex* [*simp del*]

lemma *if-neq* [*dest*!]:
  (*if P then A else B*) ≠ *B* ⟹ *P*
  by (*cases P*, *auto*)

context *JVM-heap-base* **begin**

lemma *exec-d-no-errorI* [*intro*]:
  check P σ ⟹ *exec-d P t σ* ≠ *TypeError*
  by (*unfold exec-d-def*) *simp*

theorem *no-type-error-commutes*:
  exec-d P t σ ≠ *TypeError* ⟹ *exec-d P t σ = Normal* (*exec P t σ*)
  by (*unfold exec-d-def*, *auto*)

lemma *defensive-imp-aggressive-1*:
  P,*t* ⊢ (*Normal σ*) −*tas*−*jvmd*→ (*Normal σ′*) ⟹ *P*,*t* ⊢ *σ* −*tas*−*jvm*→ *σ′*
by(*auto elim*!: *exec-1-d.cases intro*!: *exec-1.intros simp add*: *exec-d-def split*: *if-split-asm*)

end

context *JVM-heap* **begin**

lemma *check-exec-hext*:
  assumes *exec*: (*ta*, *xcp′*, *h′*, *frs′*) ∈ *exec P t* (*xcp*, *h*, *frs*)
  and *check*: *check P* (*xcp*, *h*, *frs*)
  shows *h* ⊴ *h′*
proof −
  from *exec* **have** *frs* ≠ [] **by**(*auto*)
  then obtain *f Frs* **where** *frs* [*simp*]: *frs = f # Frs*
    by(*fastforce simp add*: *neq-Nil-conv*)
  obtain *stk loc C0 M0 pc* **where** *f* [*simp*]: *f* = (*stk*, *loc*, *C0*, *M0*, *pc*)
    by(*cases f*, *blast*)
  show *?thesis*
  proof(*cases xcp*)
    case *None*
    with *check* **obtain** *C′ Ts T mxs mxl0 ins xt*
      where *mthd*: *P* ⊢ *C0 sees M0* : *Ts* → *T* = ⌊(*mxs*, *mxl0*, *ins*, *xt*)⌋ *in C′*
                method P C0 M0 = (*C′*, *Ts*, *T*, ⌊(*mxs*, *mxl0*, *ins*, *xt*)⌋)
      and *check-ins*: *check-instr* (*ins* ! *pc*) *P h stk loc C0 M0 pc Frs*
      and *pc* < *length ins*
      and *length stk* ≤ *mxs*
      by(*auto simp add*: *check-def has-method-def*)
    from *None exec mthd*
    have *xexec*: (*ta*, *xcp′*, *h′*, *frs′*) ∈ *exec-instr* (*ins* ! *pc*) *P t h stk loc C0 M0 pc Frs* **by**(*clarsimp*)
    thus *?thesis*

```
    proof(cases ins ! pc)
      case (New C)
      with xexec show ?thesis
        by(auto intro: hext-allocate split: if-split-asm)
    next
      case (NewArray T)
      with xexec show ?thesis
        by(auto intro: hext-allocate split: if-split-asm)
    next
      case AStore
      with xexec check-ins show ?thesis
        by(auto simp add: split-beta split: if-split-asm intro: hext-heap-write)
    next
      case Putfield
      with xexec check-ins show ?thesis
        by(auto intro: hext-heap-write simp add: split-beta split: if-split-asm)
    next
      case CAS
      with xexec check-ins show ?thesis
        by(auto intro: hext-heap-write simp add: split-beta split: if-split-asm)
    next
      case (Invoke M n)
      with xexec check-ins show ?thesis
        apply(auto simp add: min-def split-beta is-Ref-def extRet2JVM-def has-method-def
                split: if-split-asm intro: red-external-aggr-hext)
        apply(case-tac va)
        apply(auto 4 3 intro: red-external-aggr-hext is-native.intros)
        done
    next
      case (BinOpInstr bop)
      with xexec check-ins show ?thesis by(auto split: sum.split-asm)
    qed(auto simp add: split-beta split: if-split-asm)
  next
    case (Some a)
    with exec have h' = h by auto
    thus ?thesis by auto
  qed
qed

lemma exec-1-d-hext:
  ⟦ P,t ⊢ Normal (xcp, h, frs) −ta−jvmd→ Normal (xcp', h', frs') ⟧ ⟹ h ◁ h'
by(auto elim!: exec-1-d.cases simp add: exec-d-def split: if-split-asm intro: check-exec-hext)

end

end
```

5.8 Instantiating the framework semantics with the JVM

```
theory JVMThreaded
imports
  JVMDefensive
  ../Common/ConformThreaded
```

../Framework/FWLiftingSem
../Framework/FWProgressAux
begin

primrec *JVM-final* :: *'addr jvm-thread-state* ⇒ *bool*
where
JVM-final (*xcp*, *frs*) = (*frs* = [])

The aggressive JVM

context *JVM-heap-base* **begin**

abbreviation *mexec* ::
'addr jvm-prog ⇒ *'thread-id* ⇒ (*'addr jvm-thread-state* × *'heap*)
⇒ (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action* ⇒ (*'addr jvm-thread-state* × *'heap*) ⇒ *bool*
where
mexec P t ≡ (λ((*xcp*, *frstls*), *h*) *ta* ((*xcp'*, *frstls'*), *h'*). *P*,*t* ⊢ (*xcp*, *h*, *frstls*) −*ta*−*jvm*→ (*xcp'*, *h'*, *frstls'*))

lemma *NewThread-memory-exec-instr*:
⟦ (*ta*, *s*) ∈ *exec-instr I P t h stk loc C M pc frs*; *NewThread t' x m* ∈ *set* {|*ta*|}*t* ⟧ ⟹ *m* = *fst* (*snd s*)
apply(*cases I*)
apply(*auto split*: *if-split-asm simp add*: *split-beta ta-upd-simps*)
apply(*auto dest*!: *red-ext-aggr-new-thread-heap simp add*: *extRet2JVM-def split*: *extCallRet.split*)
done

lemma *NewThread-memory-exec*:
⟦ *P*,*t* ⊢ σ −*ta*−*jvm*→ σ'; *NewThread t' x m* ∈ *set* {|*ta*|}*t* ⟧ ⟹ *m* = (*fst* (*snd* σ'))
apply(*erule exec-1.cases*)
apply(*clarsimp*)
apply(*case-tac bb*, *simp*)
apply(*case-tac ag*, *auto simp add*: *exception-step-def-raw split*: *list.split-asm*)
apply(*drule NewThread-memory-exec-instr*, *simp*+)
done

lemma *exec-instr-Wakeup-no-Lock-no-Join-no-Interrupt*:
⟦ (*ta*, *s*) ∈ *exec-instr I P t h stk loc C M pc frs*; *Notified* ∈ *set* {|*ta*|}*w* ∨ *WokenUp* ∈ *set* {|*ta*|}*w* ⟧
⟹ *collect-locks* {|*ta*|}*l* = {} ∧ *collect-cond-actions* {|*ta*|}*c* = {} ∧ *collect-interrupts* {|*ta*|}*i* = {}
apply(*cases I*)
apply(*auto split*: *if-split-asm simp add*: *split-beta ta-upd-simps dest*: *red-external-aggr-Wakeup-no-Join*)
done

lemma *mexec-instr-Wakeup-no-Join*:
⟦ *P*,*t* ⊢ σ −*ta*−*jvm*→ σ'; *Notified* ∈ *set* {|*ta*|}*w* ∨ *WokenUp* ∈ *set* {|*ta*|}*w* ⟧
⟹ *collect-locks* {|*ta*|}*l* = {} ∧ *collect-cond-actions* {|*ta*|}*c* = {} ∧ *collect-interrupts* {|*ta*|}*i* = {}
apply(*erule exec-1.cases*)
apply(*clarsimp*)
apply(*case-tac bb*, *simp*)
apply(*case-tac ag*, *clarsimp simp add*: *exception-step-def-raw split*: *list.split-asm del*: *disjE*)
apply(*drule exec-instr-Wakeup-no-Lock-no-Join-no-Interrupt*)
apply *auto*
done

lemma *mexec-final*:
⟦ *mexec P t* (*x*, *m*) *ta* (*x'*, *m'*); *JVM-final x* ⟧ ⟹ *False*

by(*cases x*)(*auto simp add*: *exec-1-iff*)

lemma *exec-mthr*: *multithreaded JVM-final* (*mexec P*)
apply(*unfold-locales*)
apply(*clarsimp*, *drule NewThread-memory-exec*, *fastforce*, *simp*)
apply(*erule* (*1*) *mexec-final*)
done

end

sublocale *JVM-heap-base* < *exec-mthr*:
multithreaded
JVM-final
mexec P
convert-RA
for *P*
by(*rule exec-mthr*)

context *JVM-heap-base* **begin**

abbreviation *mexecT* ::
'addr jvm-prog
⇒ (*'addr*,*'thread-id*,*'addr jvm-thread-state*,*'heap*,*'addr*) *state*
⇒ *'thread-id* × (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action*
⇒ (*'addr*,*'thread-id*,*'addr jvm-thread-state*,*'heap*,*'addr*) *state* ⇒ *bool*
where
mexecT P ≡ *exec-mthr.redT P*

abbreviation *mexecT-syntax1* ::
'addr jvm-prog ⇒ (*'addr*,*'thread-id*,*'addr jvm-thread-state*,*'heap*,*'addr*) *state*
⇒ *'thread-id* ⇒ (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action*
⇒ (*'addr*,*'thread-id*,*'addr jvm-thread-state*,*'heap*,*'addr*) *state* ⇒ *bool*
(‹- ⊢ - −▷-→$_{jvm}$ -› [*50*,*0*,*0*,*0*,*50*] *80*)
where
mexecT-syntax1 P s t ta s' ≡ *mexecT P s* (*t*, *ta*) *s'*

abbreviation *mExecT-syntax1* ::
'addr jvm-prog ⇒ (*'addr*,*'thread-id*,*'addr jvm-thread-state*,*'heap*,*'addr*) *state*
⇒ (*'thread-id* × (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action*) *list*
⇒ (*'addr*,*'thread-id*,*'addr jvm-thread-state*,*'heap*,*'addr*) *state* ⇒ *bool*
(‹- ⊢ - −▷-→$_{jvm}$* -› [*50*,*0*,*0*,*50*] *80*)
where
P ⊢ *s* −▷*ttas*→$_{jvm}$* *s'* ≡ *exec-mthr.RedT P s ttas s'*

The defensive JVM

abbreviation *mexecd* ::
'addr jvm-prog ⇒ *'thread-id* ⇒ *'addr jvm-thread-state* × *'heap*
⇒ (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action* ⇒ *'addr jvm-thread-state* × *'heap* ⇒ *bool*
where
mexecd P t ≡ (λ((*xcp*, *frstls*), *h*) *ta* ((*xcp'*, *frstls'*), *h'*). *P*,*t* ⊢ *Normal* (*xcp*, *h*, *frstls*) −*ta*−*jvmd*→ *Normal* (*xcp'*, *h'*, *frstls'*))

lemma *execd-mthr*: *multithreaded JVM-final* (*mexecd P*)

apply(*unfold-locales*)
apply(*fastforce dest*: *defensive-imp-aggressive-1 NewThread-memory-exec*)
apply(*auto elim*: *jvmd-NormalE*)
done

end

sublocale *JVM-heap-base* < *execd-mthr*:
multithreaded
JVM-final
mexecd P
convert-RA
for *P*
by(*rule execd-mthr*)

context *JVM-heap-base* **begin**

abbreviation *mexecdT* ::
'addr jvm-prog ⇒ (*'addr*,*'thread-id*,*'addr jvm-thread-state*,*'heap*,*'addr*) *state*
⇒ *'thread-id* × (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action*
⇒ (*'addr*,*'thread-id*,*'addr jvm-thread-state*,*'heap*,*'addr*) *state* ⇒ *bool*
where
mexecdT P ≡ *execd-mthr.redT P*

abbreviation *mexecdT-syntax1* ::
'addr jvm-prog ⇒ (*'addr*,*'thread-id*,*'addr jvm-thread-state*,*'heap*,*'addr*) *state*
⇒ *'thread-id* ⇒ (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action*
⇒ (*'addr*,*'thread-id*,*'addr jvm-thread-state*,*'heap*,*'addr*) *state* ⇒ *bool*
(‹- ⊢ - −-▷-→$_{jvmd}$ -› [*50*,*0*,*0*,*0*,*50*] *80*)
where
mexecdT-syntax1 P s t ta s' ≡ *mexecdT P s* (*t*, *ta*) *s'*

abbreviation *mExecdT-syntax1* ::
'addr jvm-prog ⇒ (*'addr*,*'thread-id*,*'addr jvm-thread-state*,*'heap*,*'addr*) *state*
⇒ (*'thread-id* × (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action*) *list*
⇒ (*'addr*,*'thread-id*,*'addr jvm-thread-state*,*'heap*,*'addr*) *state* ⇒ *bool*
(‹- ⊢ - −-▷-→$_{jvmd}$* -› [*50*,*0*,*0*,*50*] *80*)
where
P ⊢ *s* −▷*ttas*→$_{jvmd}$* *s'* ≡ *execd-mthr.RedT P s ttas s'*

lemma *mexecd-Suspend-Invoke*:
⟦ *mexecd P t* (*x*, *m*) *ta* (*x'*, *m'*); *Suspend w* ∈ *set* {|*ta*|}$_w$ ⟧
⟹ ∃ *stk loc C M pc frs' n a T Ts Tr D*. *x'* = (*None*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs'*) ∧ *instrs-of P C M* ! *pc* = *Invoke wait n* ∧ *stk* ! *n* = *Addr a* ∧ *typeof-addr m a* = ⌊*T*⌋ ∧ *P* ⊢ *class-type-of T sees wait*:*Ts*→*Tr* = *Native in D* ∧ *D*•*wait*(*Ts*) :: *Tr*
apply(*cases x'*)
apply(*cases x*)
apply(*cases fst x*)
apply(*auto elim*!: *jvmd-NormalE simp add*: *split-beta*)
apply(*rename-tac* [!] *stk loc C M pc frs*)
apply(*case-tac* [!] *instrs-of P C M* ! *pc*)
apply(*auto split*: *if-split-asm simp add*: *split-beta check-def is-Ref-def has-method-def*)
apply(*frule red-external-aggr-Suspend-StaySame*, *simp*, *drule red-external-aggr-Suspend-waitD*, *simp*,

fastforce)+
done

end

context *JVM-heap* **begin**

lemma *exec-instr-New-Thread-exists-thread-object*:
⟦ (*ta*, *xcp'*, *h'*, *frs'*) ∈ *exec-instr ins P t h stk loc C M pc frs*;
check-instr ins P h stk loc C M pc frs;
NewThread t' x h'' ∈ *set* {|*ta*|}$_t$ ⟧
⟹ ∃ *C*. *typeof-addr h'* (*thread-id2addr t'*) = ⌊*Class-type C*⌋ ∧ *P* ⊢ *C* ⪯* *Thread*
apply(*cases ins*)
apply(*fastforce simp add*: *split-beta ta-upd-simps split*: *if-split-asm intro*: *red-external-aggr-new-thread-exists-thread*
done

lemma *exec-New-Thread-exists-thread-object*:
⟦ *P*,*t* ⊢ *Normal* (*xcp*, *h*, *frs*) −*ta*−*jvmd*→ *Normal* (*xcp'*, *h'*, *frs'*); *NewThread t' x h''* ∈ *set* {|*ta*|}$_t$ ⟧
⟹ ∃ *C*. *typeof-addr h'* (*thread-id2addr t'*) = ⌊*Class-type C*⌋ ∧ *P* ⊢ *C* ⪯* *Thread*
apply(*cases xcp*)
apply(*case-tac* [!] *frs*)
apply(*auto simp add*: *check-def elim*!: *jvmd-NormalE dest*!: *exec-instr-New-Thread-exists-thread-object*)
done

lemma *exec-instr-preserve-tconf*:
⟦ (*ta*, *xcp'*, *h'*, *frs'*) ∈ *exec-instr ins P t h stk loc C M pc frs*;
check-instr ins P h stk loc C M pc frs;
P,*h* ⊢ *t'* √*t* ⟧
⟹ *P*,*h'* ⊢ *t'* √*t*
apply(*cases ins*)
apply(*auto intro*: *tconf-hext-mono hext-allocate hext-heap-write red-external-aggr-preserves-tconf split*: *if-split-asm sum.split-asm simp add*: *split-beta has-method-def intro*!: *is-native.intros cong del*: *image-cong-simp*)
done

lemma *exec-preserve-tconf*:
⟦ *P*,*t* ⊢ *Normal* (*xcp*, *h*, *frs*) −*ta*−*jvmd*→ *Normal* (*xcp'*, *h'*, *frs'*); *P*,*h* ⊢ *t'* √*t* ⟧ ⟹ *P*,*h'* ⊢ *t'* √*t*
apply(*cases xcp*)
apply(*case-tac* [!] *frs*)
apply(*auto simp add*: *check-def elim*!: *jvmd-NormalE elim*!: *exec-instr-preserve-tconf*)
done

lemma *lifting-wf-thread-conf*: *lifting-wf JVM-final* (*mexecd P*) (λ*t x m*. *P*,*m* ⊢ *t* √*t*)
by(*unfold-locales*)(*auto intro*: *exec-preserve-tconf dest*: *exec-New-Thread-exists-thread-object intro*: *tconfI*)

end

sublocale *JVM-heap* < *execd-tconf*: *lifting-wf JVM-final mexecd P convert-RA* λ*t x m*. *P*,*m* ⊢ *t* √*t*
by(*rule lifting-wf-thread-conf*)

context *JVM-heap* **begin**

lemma *execd-hext*:
P ⊢ *s* −*t*▷*ta*→$_{jvmd}$ *s'* ⟹ *shr s* ≤| *shr s'*

by(*auto elim*!: *execd-mthr.redT.cases dest*!: *exec-1-d-hext intro*: *hext-trans*)

lemma *Execd-hext*:
  assumes $P \vdash s\ -\triangleright tta\rightarrow_{jvmd}*\ s'$
  shows *shr s* $\trianglelefteq$ *shr s'*
using *assms* **unfolding** *execd-mthr.RedT-def*
by(*induct*)(*auto dest*!: *execd-hext intro*: *hext-trans simp add*: *execd-mthr.RedT-def*)

end

end
theory *JVM-Main*
imports
  JVMState
  JVMThreaded
begin

end

Chapter 6

Bytecode verifier

6.1 The JVM Type System as Semilattice

theory *JVM-SemiType*
imports
../Common/SemiType
begin

type-synonym $ty_l = ty\ err\ list$
type-synonym $ty_s = ty\ list$
type-synonym $ty_i = ty_s \times ty_l$
type-synonym $ty_i' = ty_i\ option$
type-synonym $ty_m = ty_i'\ list$
type-synonym $ty_P = mname \Rightarrow cname \Rightarrow ty_m$

definition *stk-esl* :: $'c\ prog \Rightarrow nat \Rightarrow ty_s\ esl$
where
stk-esl P mxs $\equiv$ *upto-esl mxs* (*SemiType.esl P*)

definition *loc-sl* :: $'c\ prog \Rightarrow nat \Rightarrow ty_l\ sl$
where
loc-sl P mxl $\equiv$ *Listn.sl mxl* (*Err.sl* (*SemiType.esl P*))

definition *sl* :: $'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ sl$
where
sl P mxs mxl $\equiv$
Err.sl(*Opt.esl*(*Product.esl* (*stk-esl P mxs*) (*Err.esl*(*loc-sl P mxl*))))

definition *states* :: $'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ set$
where
states P mxs mxl $\equiv$ *fst*(*sl P mxs mxl*)

definition *le* :: $'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ ord$
where
le P mxs mxl $\equiv$ *fst*(*snd*(*sl P mxs mxl*))

definition *sup* :: $'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ binop$
where
sup P mxs mxl $\equiv$ *snd*(*snd*(*sl P mxs mxl*))

definition *sup-ty-opt* :: [*'c prog*,*ty err*,*ty err*] ⇒ *bool*
(‹- ⊢ - ≤⊤ -› [*71*,*71*,*71*] *70*)
where
sup-ty-opt P ≡ *Err.le* (*widen P*)

definition *sup-state* :: [*'c prog*,*ty*$_i$,*ty*$_i$] ⇒ *bool*
(‹- ⊢ - ≤$_i$ -› [*71*,*71*,*71*] *70*)
where
sup-state P ≡ *Product.le* (*Listn.le* (*widen P*)) (*Listn.le* (*sup-ty-opt P*))

definition *sup-state-opt* :: [*'c prog*,*ty*$_i$*'*,*ty*$_i$*'*] ⇒ *bool*
(‹- ⊢ - ≤'' -› [*71*,*71*,*71*] *70*)
where
sup-state-opt P ≡ *Opt.le* (*sup-state P*)

abbreviation *sup-loc* :: [*'c prog*,*ty*$_l$,*ty*$_l$] ⇒ *bool* (‹- ⊢ - [≤⊤] -› [*71*,*71*,*71*] *70*)
where *P* ⊢ *LT* [≤⊤] *LT'* ≡ *list-all2* (*sup-ty-opt P*) *LT LT'*

notation (*ASCII*)
sup-ty-opt (‹- |- - <=*T* -› [*71*,*71*,*71*] *70*) **and**
sup-state (‹- |- - <=*i* -› [*71*,*71*,*71*] *70*) **and**
sup-state-opt (‹- |- - <=' -› [*71*,*71*,*71*] *70*) **and**
sup-loc (‹- |- - [<=*T*] -› [*71*,*71*,*71*] *70*)

6.1.1 Unfolding

lemma *JVM-states-unfold*:
states P mxs mxl ≡ *err*(*opt*((*Union* {*list n* (*types P*) |*n*. *n* <= *mxs*}) ×
list mxl (*err*(*types P*))))
apply (*unfold states-def sl-def Opt.esl-def Err.sl-def*
stk-esl-def loc-sl-def Product.esl-def
Listn.sl-def upto-esl-def SemiType.esl-def Err.esl-def)
apply *simp*
done

lemma *JVM-le-unfold*:
le P m n ≡
Err.le(*Opt.le*(*Product.le*(*Listn.le*(*widen P*))(*Listn.le*(*Err.le*(*widen P*)))))
apply (*unfold le-def sl-def Opt.esl-def Err.sl-def*
stk-esl-def loc-sl-def Product.esl-def
Listn.sl-def upto-esl-def SemiType.esl-def Err.esl-def)
apply *simp*
done

lemma *sl-def2*:
JVM-SemiType.sl P mxs mxl ≡
(*states P mxs mxl*, *JVM-SemiType.le P mxs mxl*, *JVM-SemiType.sup P mxs mxl*)
by (*unfold JVM-SemiType.sup-def states-def JVM-SemiType.le-def*) *simp*

lemma *JVM-le-conv*:
le P m n (*OK t1*) (*OK t2*) = *P* ⊢ *t1* ≤' *t2*
by (*simp add*: *JVM-le-unfold Err.le-def lesub-def sup-state-opt-def*
sup-state-def sup-ty-opt-def)

lemma *JVM-le-Err-conv*:
le P m n = *Err.le* (*sup-state-opt P*)
by (*unfold sup-state-opt-def sup-state-def*
sup-ty-opt-def JVM-le-unfold) *simp*

lemma *err-le-unfold* [*iff*]:
Err.le r (*OK a*) (*OK b*) = *r a b*
by (*simp add*: *Err.le-def lesub-def*)

6.1.2 Semilattice

lemma *order-sup-state-opt* [*intro*, *simp*]:
wf-prog wf-mb P ⟹ *order* (*sup-state-opt P*)
by (*unfold sup-state-opt-def sup-state-def sup-ty-opt-def*) *blast*

lemma *semilat-JVM* [*intro?*]:
wf-prog wf-mb P ⟹ *semilat* (*JVM-SemiType.sl P mxs mxl*)
apply (*unfold JVM-SemiType.sl-def stk-esl-def loc-sl-def*)
apply (*blast intro*: *err-semilat-Product-esl err-semilat-upto-esl*
Listn-sl err-semilat-JType-esl)
done

lemma *acc-JVM* [*intro*]:
wf-prog wf-mb P ⟹ *acc* (*JVM-SemiType.states P mxs mxl*) (*JVM-SemiType.le P mxs mxl*)
by(*unfold JVM-le-unfold JVM-states-unfold*) *blast*

6.1.3 Widening with ⊤

lemma *widen-refl*[*iff*]: *widen P t t* **by** (*simp add*: *fun-of-def*)

lemma *sup-ty-opt-refl* [*iff*]: *P* ⊢ *T* ≤⊤ *T*
apply (*unfold sup-ty-opt-def*)
apply (*fold lesub-def*)
apply (*rule le-err-refl*)
apply (*simp add*: *lesub-def*)
done

lemma *Err-any-conv* [*iff*]: *P* ⊢ *Err* ≤⊤ *T* = (*T* = *Err*)
by (*unfold sup-ty-opt-def*) (*rule Err-le-conv* [*simplified lesub-def*])

lemma *any-Err* [*iff*]: *P* ⊢ *T* ≤⊤ *Err*
by (*unfold sup-ty-opt-def*) (*rule le-Err* [*simplified lesub-def*])

lemma *OK-OK-conv* [*iff*]:
P ⊢ *OK T* ≤⊤ *OK T'* = *P* ⊢ *T* ≤ *T'*
by (*simp add*: *sup-ty-opt-def fun-of-def*)

lemma *any-OK-conv* [*iff*]:
P ⊢ *X* ≤⊤ *OK T'* = (∃ *T*. *X* = *OK T* ∧ *P* ⊢ *T* ≤ *T'*)
apply (*unfold sup-ty-opt-def*)
apply (*rule le-OK-conv* [*simplified lesub-def*])
done

lemma *OK-any-conv*:
$P \vdash OK\ T \leq_\top X = (X = Err \vee (\exists\, T'.\ X = OK\ T' \wedge P \vdash T \leq T'))$
apply (*unfold sup-ty-opt-def*)
apply (*rule OK-le-conv* [*simplified lesub-def*])
done

lemma *sup-ty-opt-trans* [*intro?*, *trans*]:
$[\![P \vdash a \leq_\top b;\ P \vdash b \leq_\top c]\!] \Longrightarrow P \vdash a \leq_\top c$
by (*auto intro*: *widen-trans*
simp add: *sup-ty-opt-def Err.le-def lesub-def fun-of-def*
split: *err.splits*)

6.1.4 Stack and Registers

lemma *stk-convert*:
$P \vdash ST\ [\leq]\ ST' = Listn.le\ (widen\ P)\ ST\ ST'$
by (*simp add*: *Listn.le-def lesub-def*)

lemma *sup-loc-refl* [*iff*]: $P \vdash LT\ [\leq_\top]\ LT$
by (*rule list-all2-refl*) *simp*

lemmas *sup-loc-Cons1* [*iff*] = *list-all2-Cons1* [*of sup-ty-opt P*] **for** *P*

lemma *sup-loc-def*:
$P \vdash LT\ [\leq_\top]\ LT' \equiv Listn.le\ (sup\text{-}ty\text{-}opt\ P)\ LT\ LT'$
by (*simp add*: *Listn.le-def lesub-def*)

lemma *sup-loc-widens-conv* [*iff*]:
$P \vdash map\ OK\ Ts\ [\leq_\top]\ map\ OK\ Ts' = P \vdash Ts\ [\leq]\ Ts'$
by (*simp add*: *list-all2-map1 list-all2-map2*)

lemma *sup-loc-trans* [*intro?*, *trans*]:
$[\![P \vdash a\ [\leq_\top]\ b;\ P \vdash b\ [\leq_\top]\ c]\!] \Longrightarrow P \vdash a\ [\leq_\top]\ c$
by (*rule list-all2-trans*, *rule sup-ty-opt-trans*)

6.1.5 State Type

lemma *sup-state-conv* [*iff*]:
$P \vdash (ST,LT) \leq_i (ST',LT') = (P \vdash ST\ [\leq]\ ST' \wedge P \vdash LT\ [\leq_\top]\ LT')$
by (*auto simp add*: *sup-state-def stk-convert lesub-def Product.le-def sup-loc-def*)

lemma *sup-state-conv2*:
$P \vdash s1 \leq_i s2 = (P \vdash fst\ s1\ [\leq]\ fst\ s2 \wedge P \vdash snd\ s1\ [\leq_\top]\ snd\ s2)$
by (*cases s1*, *cases s2*) *simp*

lemma *sup-state-refl* [*iff*]: $P \vdash s \leq_i s$
by (*auto simp add*: *sup-state-conv2 intro*: *list-all2-refl*)

lemma *sup-state-trans* [*intro?*, *trans*]:
$[\![P \vdash a \leq_i b;\ P \vdash b \leq_i c]\!] \Longrightarrow P \vdash a \leq_i c$
by (*auto intro*: *sup-loc-trans widens-trans simp add*: *sup-state-conv2*)

lemma *sup-state-opt-None-any* [*iff*]:
$P \vdash None \leq' s$

by (*simp add*: *sup-state-opt-def Opt.le-def*)

lemma *sup-state-opt-any-None* [*iff*]:
$P \vdash s \leq' None = (s = None)$
by (*simp add*: *sup-state-opt-def Opt.le-def*)

lemma *sup-state-opt-Some-Some* [*iff*]:
$P \vdash Some\ a \leq' Some\ b = P \vdash a \leq_i b$
by (*simp add*: *sup-state-opt-def Opt.le-def lesub-def*)

lemma *sup-state-opt-any-Some*:
$P \vdash (Some\ s) \leq' X = (\exists s'.\ X = Some\ s' \wedge P \vdash s \leq_i s')$
by (*simp add*: *sup-state-opt-def Opt.le-def lesub-def*)

lemma *sup-state-opt-refl* [*iff*]: $P \vdash s \leq' s$
by (*simp add*: *sup-state-opt-def Opt.le-def lesub-def*)

lemma *sup-state-opt-trans* [*intro?*, *trans*]:
$[\![P \vdash a \leq' b;\ P \vdash b \leq' c]\!] \Longrightarrow P \vdash a \leq' c$
apply (*unfold sup-state-opt-def Opt.le-def lesub-def*)
apply (*simp del*: *split-paired-All*)
apply (*rule sup-state-trans*, *assumption+*)
done

end

6.2 Effect of Instructions on the State Type

theory *Effect*
imports
JVM-SemiType
../JVM/JVMExceptions
begin

locale *jvm-method* = *prog* +
fixes *mxs* :: *nat*
fixes mxl_0 :: *nat*
fixes *Ts* :: *ty list*
fixes T_r :: *ty*
fixes *is* :: *'addr instr list*
fixes *xt* :: *ex-table*

fixes *mxl* :: *nat*
defines *mxl-def*: $mxl \equiv 1+size\ Ts+mxl_0$

Program counter of successor instructions:

primrec *succs* :: *'addr instr* $\Rightarrow$ ty_i $\Rightarrow$ *pc* $\Rightarrow$ *pc list*
where
succs (*Load idx*) τ *pc* = [*pc+1*]
| *succs* (*Store idx*) τ *pc* = [*pc+1*]
| *succs* (*Push v*) τ *pc* = [*pc+1*]
| *succs* (*Getfield F C*) τ *pc* = [*pc+1*]
| *succs* (*Putfield F C*) τ *pc* = [*pc+1*]
| *succs* (*CAS F C*) τ *pc* = [*pc+1*]

| *succs* (*New C*) τ *pc* = [*pc+1*]
| *succs* (*NewArray T*) τ *pc* = [*pc+1*]
| *succs ALoad* τ *pc* = (*if* (*fst* τ)!*1* = *NT then* [] *else* [*pc+1*])
| *succs AStore* τ *pc* = (*if* (*fst* τ)!*2* = *NT then* [] *else* [*pc+1*])
| *succs ALength* τ *pc* = (*if* (*fst* τ)!*0* = *NT then* [] *else* [*pc+1*])
| *succs* (*Checkcast C*) τ *pc* = [*pc+1*]
| *succs* (*Instanceof T*) τ *pc* = [*pc+1*]
| *succs Pop* τ *pc* = [*pc+1*]
| *succs Dup* τ *pc* = [*pc+1*]
| *succs Swap* τ *pc* = [*pc+1*]
| *succs* (*BinOpInstr b*) τ *pc* = [*pc+1*]
| *succs-IfFalse*:
succs (*IfFalse b*) τ *pc* = [*pc+1*, *nat* (*int pc* + *b*)]
| *succs-Goto*:
succs (*Goto b*) τ *pc* = [*nat* (*int pc* + *b*)]
| *succs-Return*:
succs Return τ *pc* = []
| *succs-Invoke*:
succs (*Invoke M n*) τ *pc* = (*if* (*fst* τ)!*n* = *NT then* [] *else* [*pc+1*])
| *succs-Throw*:
succs ThrowExc τ *pc* = []
| *succs MEnter* τ *pc* = (*if* (*fst* τ)!*0* = *NT then* [] *else* [*pc+1*])
| *succs MExit* τ *pc* = (*if* (*fst* τ)!*0* = *NT then* [] *else* [*pc+1*])

Effect of instruction on the state type:

fun eff_i :: *'addr instr* × *'m prog* × ty_i ⇒ ty_i
where
eff_i-*Load*:
eff_i (*Load n*, *P*, (*ST*, *LT*)) = (*ok-val* (*LT* ! *n*) # *ST*, *LT*)

| eff_i-*Store*:
eff_i (*Store n*, *P*, (*T*#*ST*, *LT*)) = (*ST*, *LT*[*n*:= *OK T*])

| eff_i-*Push*:
eff_i (*Push v*, *P*, (*ST*, *LT*)) = (*the* (*typeof v*) # *ST*, *LT*)

| eff_i-*Getfield*:
eff_i (*Getfield F C*, *P*, (*T*#*ST*, *LT*)) = (*fst* (*snd* (*field P C F*)) # *ST*, *LT*)

| eff_i-*Putfield*:
eff_i (*Putfield F C*, *P*, (T_1#T_2#*ST*, *LT*)) = (*ST*,*LT*)

| eff_i-*CAS*:
eff_i (*CAS F C*, *P*, (T_1#T_2#T_3#*ST*, *LT*)) = (*Boolean* # *ST*, *LT*)

| eff_i-*New*:
eff_i (*New C*, *P*, (*ST*,*LT*)) = (*Class C* # *ST*, *LT*)

| eff_i-*NewArray*:
eff_i (*NewArray Ty*, *P*, (*T*#*ST*,*LT*)) = (*Ty*⌊⌋ # *ST*,*LT*)

| eff_i-*ALoad*:
eff_i (*ALoad*, *P*, (*T1*#*T2*#*ST*,*LT*)) = (*the-Array T2*# *ST*,*LT*)

| *eff$_i$-AStore*:
eff$_i$ (*AStore*, *P*, (*T1*#*T2*#*T3*#*ST*,*LT*)) = (*ST*,*LT*)

| *eff$_i$-ALength*:
eff$_i$ (*ALength*, *P*, (*T1*#*ST*,*LT*)) = (*Integer*#*ST*,*LT*)

| *eff$_i$-Checkcast*:
eff$_i$ (*Checkcast Ty*, *P*, (*T*#*ST*,*LT*)) = (*Ty* # *ST*,*LT*)

| *eff$_i$-Instanceof*:
eff$_i$ (*Instanceof Ty*, *P*, (*T*#*ST*,*LT*)) = (*Boolean* # *ST*,*LT*)

| *eff$_i$-Pop*:
eff$_i$ (*Pop*, *P*, (*T*#*ST*,*LT*)) = (*ST*,*LT*)

| *eff$_i$-Dup*:
eff$_i$ (*Dup*, *P*, (*T*#*ST*,*LT*)) = (*T*#*T*#*ST*,*LT*)

| *eff$_i$-Swap*:
eff$_i$ (*Swap*, *P*, (*T1*#*T2*#*ST*,*LT*)) = (*T2*#*T1*#*ST*,*LT*)

| *eff$_i$-BinOpInstr*:
eff$_i$ (*BinOpInstr bop*, *P*, (*T2*#*T1*#*ST*,*LT*)) = ((*THE T*. *P* ⊢ *T1*«*bop*»*T2* : *T*)#*ST*, *LT*)

| *eff$_i$-IfFalse*:
eff$_i$ (*IfFalse b*, *P*, (T_1#*ST*,*LT*)) = (*ST*,*LT*)

| *eff$_i$-Invoke*:
eff$_i$ (*Invoke M n*, *P*, (*ST*,*LT*)) =
(*let U* = *fst* (*snd* (*snd* (*method P* (*the* (*class-type-of'* (*ST* ! *n*))) *M*)))
in (*U* # *drop* (*n*+*1*) *ST*, *LT*))

| *eff$_i$-Goto*:
eff$_i$ (*Goto n*, *P*, *s*) = *s*

| *eff$_i$-MEnter*:
eff$_i$ (*MEnter*, *P*, (*T1*#*ST*,*LT*)) = (*ST*,*LT*)

| *eff$_i$-MExit*:
eff$_i$ (*MExit*, *P*, (*T1*#*ST*,*LT*)) = (*ST*,*LT*)

fun *is-relevant-class* :: *'addr instr* ⇒ *'m prog* ⇒ *cname* ⇒ *bool*
where
rel-Getfield:
is-relevant-class (*Getfield F D*) = (λ*P C*. *P* ⊢ *NullPointer* ≼* *C*)
| *rel-Putfield*:
is-relevant-class (*Putfield F D*) = (λ*P C*. *P* ⊢ *NullPointer* ≼* *C*)
| *rel-CAS*:
is-relevant-class (*CAS F D*) = (λ*P C*. *P* ⊢ *NullPointer* ≼* *C*)
| *rel-Checcast*:
is-relevant-class (*Checkcast T*) = (λ*P C*. *P* ⊢ *ClassCast* ≼* *C*)
| *rel-New*:
is-relevant-class (*New D*) = (λ*P C*. *P* ⊢ *OutOfMemory* ≼* *C*)

| *rel-Throw*:
is-relevant-class ThrowExc = (λ*P C*. *True*)
| *rel-Invoke*:
is-relevant-class (*Invoke M n*) = (λ*P C*. *True*)
| *rel-NewArray*:
is-relevant-class (*NewArray T*) = (λ*P C*. (*P* ⊢ *OutOfMemory* ⪯* *C*) ∨ (*P* ⊢ *NegativeArraySize* ⪯* *C*))
| *rel-ALoad*:
is-relevant-class ALoad = (λ*P C*. *P* ⊢ *ArrayIndexOutOfBounds* ⪯* *C* ∨ *P* ⊢ *NullPointer* ⪯* *C*)
| *rel-AStore*:
is-relevant-class AStore = (λ*P C*. *P* ⊢ *ArrayIndexOutOfBounds* ⪯* *C* ∨ *P* ⊢ *ArrayStore* ⪯* *C* ∨ *P* ⊢ *NullPointer* ⪯* *C*)
| *rel-ALength*:
is-relevant-class ALength = (λ*P C*. *P* ⊢ *NullPointer* ⪯* *C*)
| *rel-MEnter*:
is-relevant-class MEnter = (λ*P C*. *P* ⊢ *IllegalMonitorState* ⪯* *C* ∨ *P* ⊢ *NullPointer* ⪯* *C*)
| *rel-MExit*:
is-relevant-class MExit = (λ*P C*. *P* ⊢ *IllegalMonitorState* ⪯* *C* ∨ *P* ⊢ *NullPointer* ⪯* *C*)
| *rel-BinOp*:
is-relevant-class (*BinOpInstr bop*) = *binop-relevant-class bop*
| *rel-default*:
is-relevant-class i = (λ*P C*. *False*)

definition *is-relevant-entry* :: *'m prog* ⇒ *'addr instr* ⇒ *pc* ⇒ *ex-entry* ⇒ *bool*
where
is-relevant-entry P i pc e ≡
let (*f*,*t*,*C*,*h*,*d*) = *e*
in (*case C of None* ⇒ *True* | ⌊*C'*⌋ ⇒ *is-relevant-class i P C'*) ∧ *pc* ∈ {*f*..<*t*}

definition *relevant-entries* :: *'m prog* ⇒ *'addr instr* ⇒ *pc* ⇒ *ex-table* ⇒ *ex-table*
where
relevant-entries P i pc ≡ *filter* (*is-relevant-entry P i pc*)

definition *xcpt-eff* :: *'addr instr* ⇒ *'m prog* ⇒ *pc* ⇒ *ty*$_i$ ⇒ *ex-table* ⇒ (*pc* × *ty*$_i$*'*) *list*
where
xcpt-eff i P pc τ *et* ≡ *let* (*ST*,*LT*) = τ *in*
map (λ(*f*,*t*,*C*,*h*,*d*). (*h*, *Some* ((*case C of None* ⇒ *Class Throwable* | *Some C'* ⇒ *Class C'*)#*drop* (*size ST* − *d*) *ST*, *LT*))) (*relevant-entries P i pc et*)

definition *norm-eff* :: *'addr instr* ⇒ *'m prog* ⇒ *nat* ⇒ *ty*$_i$ ⇒ (*pc* × *ty*$_i$*'*) *list*
where *norm-eff i P pc* τ ≡ *map* (λ*pc'*. (*pc'*,*Some* (*eff*$_i$ (*i*,*P*,τ)))) (*succs i* τ *pc*)

definition *eff* :: *'addr instr* ⇒ *'m prog* ⇒ *pc* ⇒ *ex-table* ⇒ *ty*$_i$*'* ⇒ (*pc* × *ty*$_i$*'*) *list*
where
eff i P pc et t ≡
case t of
None ⇒ []
| *Some* τ ⇒ (*norm-eff i P pc* τ) @ (*xcpt-eff i P pc* τ *et*)

lemma *eff-None*:
eff i P pc xt None = []
by (*simp add*: *eff-def*)

lemma *eff-Some*:
eff i P pc xt (*Some* τ) = *norm-eff i P pc* τ @ *xcpt-eff i P pc* τ *xt*
by (*simp add*: *eff-def*)

Conditions under which eff is applicable:

fun app_i :: *'addr instr* × *'m prog* × *pc* × *nat* × *ty* × ty_i ⇒ *bool*
where
app$_i$*-Load*:
app$_i$ (*Load n*, *P*, *pc*, *mxs*, T_r, (*ST*,*LT*)) =
(*n* < *length LT* ∧ *LT* ! *n* ≠ *Err* ∧ *length ST* < *mxs*)
| *app*$_i$*-Store*:
app$_i$ (*Store n*, *P*, *pc*, *mxs*, T_r, (*T*#*ST*, *LT*)) =
(*n* < *length LT*)
| *app*$_i$*-Push*:
app$_i$ (*Push v*, *P*, *pc*, *mxs*, T_r, (*ST*,*LT*)) =
(*length ST* < *mxs* ∧ *typeof v* ≠ *None*)
| *app*$_i$*-Getfield*:
app$_i$ (*Getfield F C*, *P*, *pc*, *mxs*, T_r, (*T*#*ST*, *LT*)) =
(∃ T_f *fm*. *P* ⊢ *C sees F*:T_f (*fm*) *in C* ∧ *P* ⊢ *T* ≤ *Class C*)
| *app*$_i$*-Putfield*:
app$_i$ (*Putfield F C*, *P*, *pc*, *mxs*, T_r, (T_1#T_2#*ST*, *LT*)) =
(∃ T_f *fm*. *P* ⊢ *C sees F*:T_f (*fm*) *in C* ∧ *P* ⊢ T_2 ≤ (*Class C*) ∧ *P* ⊢ T_1 ≤ T_f)
| *app*$_i$*-CAS*:
app$_i$ (*CAS F C*, *P*, *pc*, *mxs*, T_r, (T_3#T_2#T_1#*ST*, *LT*)) =
(∃ T_f *fm*. *P* ⊢ *C sees F*:T_f (*fm*) *in C* ∧ *volatile fm* ∧ *P* ⊢ T_1 ≤ *Class C* ∧ *P* ⊢ T_2 ≤ T_f ∧ *P* ⊢ T_3 ≤ T_f)
| *app*$_i$*-New*:
app$_i$ (*New C*, *P*, *pc*, *mxs*, T_r, (*ST*,*LT*)) =
(*is-class P C* ∧ *length ST* < *mxs*)
| *app*$_i$*-NewArray*:
app$_i$ (*NewArray Ty*, *P*, *pc*, *mxs*, T_r, (*Integer*#*ST*,*LT*)) =
is-type P (*Ty*⌊⌉)
| *app*$_i$*-ALoad*:
app$_i$ (*ALoad*, *P*, *pc*, *mxs*, T_r, (*T1*#*T2*#*ST*,*LT*)) =
(*T1* = *Integer* ∧ (*T2* ≠ *NT* ⟶ (∃ *Ty*. *T2* = *Ty*⌊⌉)))
| *app*$_i$*-AStore*:
app$_i$ (*AStore*, *P*, *pc*, *mxs*, T_r, (*T1*#*T2*#*T3*#*ST*,*LT*)) =
(*T2* = *Integer* ∧ (*T3* ≠ *NT* ⟶ (∃ *Ty*. *T3* = *Ty*⌊⌉)))
| *app*$_i$*-ALength*:
app$_i$ (*ALength*, *P*, *pc*, *mxs*, T_r, (*T1*#*ST*,*LT*)) =
(*T1* = *NT* ∨ (∃ *Ty*. *T1* = *Ty*⌊⌉))
| *app*$_i$*-Checkcast*:
app$_i$ (*Checkcast Ty*, *P*, *pc*, *mxs*, T_r, (*T*#*ST*,*LT*)) =
(*is-type P Ty*)
| *app*$_i$*-Instanceof*:
app$_i$ (*Instanceof Ty*, *P*, *pc*, *mxs*, T_r, (*T*#*ST*,*LT*)) =
(*is-type P Ty* ∧ *is-refT T*)
| *app*$_i$*-Pop*:
app$_i$ (*Pop*, *P*, *pc*, *mxs*, T_r, (*T*#*ST*,*LT*)) =
True
| *app*$_i$*-Dup*:
app$_i$ (*Dup*, *P*, *pc*, *mxs*, T_r, (*T*#*ST*,*LT*)) =
(*Suc* (*length ST*) < *mxs*)

| app_i-*Swap*:
  app_i (*Swap*, *P*, *pc*, *mxs*, T_r, (*T1*#*T2*#*ST*,*LT*)) = *True*
| app_i-*BinOpInstr*:
  app_i (*BinOpInstr bop*, *P*, *pc*, *mxs*, T_r, (*T2*#*T1*#*ST*,*LT*)) = (∃ *T*. *P* ⊢ *T1* «*bop*» *T2* : *T*)
| app_i-*IfFalse*:
  app_i (*IfFalse b*, *P*, *pc*, *mxs*, T_r, (*Boolean*#*ST*,*LT*)) =
    (0 ≤ *int pc* + *b*)
| app_i-*Goto*:
  app_i (*Goto b*, *P*, *pc*, *mxs*, T_r, *s*) = (0 ≤ *int pc* + *b*)
| app_i-*Return*:
  app_i (*Return*, *P*, *pc*, *mxs*, T_r, (*T*#*ST*,*LT*)) = (*P* ⊢ *T* ≤ T_r)
| app_i-*Throw*:
  app_i (*ThrowExc*, *P*, *pc*, *mxs*, T_r, (*T*#*ST*,*LT*)) =
    (*T* = *NT* ∨ (∃ *C*. *T* = *Class C* ∧ *P* ⊢ *C* ≼* *Throwable*))
| app_i-*Invoke*:
  app_i (*Invoke M n*, *P*, *pc*, *mxs*, T_r, (*ST*,*LT*)) =
    (*n* < *length ST* ∧
    (*ST*!*n* ≠ *NT* ⟶
      (∃ *C D Ts T m*. *class-type-of'* (*ST* ! *n*) = ⌊*C*⌋ ∧ *P* ⊢ *C sees M*:*Ts* → *T* = *m in D* ∧ *P* ⊢ *rev* (*take n ST*) [≤] *Ts*)))
| app_i-*MEnter*:
  app_i (*MEnter*,*P*, *pc*,*mxs*,T_r,(*T*#*ST*,*LT*)) = (*is-refT T*)
| app_i-*MExit*:
  app_i (*MExit*,*P*, *pc*,*mxs*,T_r,(*T*#*ST*,*LT*)) = (*is-refT T*)
| app_i-*default*:
  app_i (*i*,*P*, *pc*,*mxs*,T_r,*s*) = *False*

definition *xcpt-app* :: *'addr instr* ⇒ *'m prog* ⇒ *pc* ⇒ *nat* ⇒ *ex-table* ⇒ ty_i ⇒ *bool*
where
  xcpt-app i P pc mxs xt τ ≡ ∀ (*f*,*t*,*C*,*h*,*d*) ∈ *set* (*relevant-entries P i pc xt*). (*case C of None* ⇒ *True*
| *Some C'* ⇒ *is-class P C'*) ∧ *d* ≤ *size* (*fst* τ) ∧ *d* < *mxs*

definition *app* :: *'addr instr* ⇒ *'m prog* ⇒ *nat* ⇒ *ty* ⇒ *nat* ⇒ *nat* ⇒ *ex-table* ⇒ ty_i' ⇒ *bool*
where
  app i P mxs T_r *pc mpc xt t* ≡ *case t of None* ⇒ *True* | *Some* τ ⇒
  app_i (*i*,*P*,*pc*,*mxs*,T_r,τ) ∧ *xcpt-app i P pc mxs xt* τ ∧
  (∀ (*pc'*,τ') ∈ *set* (*eff i P pc xt t*). *pc'* < *mpc*)

lemma *app-Some*:
  app i P mxs T_r *pc mpc xt* (*Some* τ) =
  (app_i (*i*,*P*,*pc*,*mxs*,T_r,τ) ∧ *xcpt-app i P pc mxs xt* τ ∧
  (∀ (*pc'*,*s'*) ∈ *set* (*eff i P pc xt* (*Some* τ)). *pc'* < *mpc*))
by (*simp add*: *app-def*)

locale *eff* = *jvm-method* +
  fixes eff_i **and** app_i **and** *eff* **and** *app*
  fixes *norm-eff* **and** *xcpt-app* **and** *xcpt-eff*

  fixes *mpc*
  defines *mpc* ≡ *size is*

  defines eff_i *i* τ ≡ *Effect*.eff_i (*i*,*P*,τ)

notes *eff$_i$-simps* [*simp*] = *Effect.eff$_i$.simps* [**where** *P* = *P*, *folded eff$_i$-def*]

defines *app$_i$ i pc τ* ≡ *Effect.app$_i$* (*i*, *P*, *pc*, *mxs*, *T$_r$*, *τ*)
notes *app$_i$-simps* [*simp*] = *Effect.app$_i$.simps* [**where** *P*=*P* **and** *mxs*=*mxs* **and** *T$_r$*=*T$_r$*, *folded app$_i$-def*]

defines *xcpt-eff i pc τ* ≡ *Effect.xcpt-eff i P pc τ xt*
notes *xcpt-eff* = *Effect.xcpt-eff-def* [*of* - *P* - - *xt*, *folded xcpt-eff-def*]

defines *norm-eff i pc τ* ≡ *Effect.norm-eff i P pc τ*
notes *norm-eff* = *Effect.norm-eff-def* [*of* - *P*, *folded norm-eff-def eff$_i$-def*]

defines *eff i pc* ≡ *Effect.eff i P pc xt*
notes *eff* = *Effect.eff-def* [*of* - *P* - *xt*, *folded eff-def norm-eff-def xcpt-eff-def*]

defines *xcpt-app i pc τ* ≡ *Effect.xcpt-app i P pc mxs xt τ*
notes *xcpt-app* = *Effect.xcpt-app-def* [*of* - *P* - *mxs xt*, *folded xcpt-app-def*]

defines *app i pc* ≡ *Effect.app i P mxs T$_r$ pc mpc xt*
notes *app* = *Effect.app-def* [*of* - *P mxs T$_r$* - *mpc xt*, *folded app-def xcpt-app-def app$_i$-def eff-def*]

lemma *length-cases2*:
assumes ⋀*LT*. *P* ([],*LT*)
assumes ⋀*l ST LT*. *P* (*l*#*ST*,*LT*)
shows *P s*
by (*cases s*, *cases fst s*) (*auto intro*!: *assms*)

lemma *length-cases3*:
assumes ⋀*LT*. *P* ([],*LT*)
assumes ⋀*l LT*. *P* ([*l*],*LT*)
assumes ⋀*l l′ ST LT*. *P* (*l*#*l′*#*ST*,*LT*)
shows *P s*
apply(*rule length-cases2*; (*rule assms*)*?*)
subgoal for *l ST LT* **by**(*cases ST*; *clarsimp simp*: *assms*)
done

lemma *length-cases4*:
assumes ⋀*LT*. *P* ([],*LT*)
assumes ⋀*l LT*. *P* ([*l*],*LT*)
assumes ⋀*l l′ LT*. *P* ([*l*,*l′*],*LT*)
assumes ⋀*l l′ l″ ST LT*. *P* (*l*#*l′*#*l″*#*ST*,*LT*)
shows *P s*
apply(*rule length-cases3*; (*rule assms*)*?*)
subgoal for *l l′ ST LT* **by**(*cases ST*; *clarsimp simp*: *assms*)
done

lemma *length-cases5*:
assumes ⋀*LT*. *P* ([],*LT*)
assumes ⋀*l LT*. *P* ([*l*],*LT*)
assumes ⋀*l l′ LT*. *P* ([*l*,*l′*],*LT*)
assumes ⋀*l l′ l″ LT*. *P* ([*l*,*l′*,*l″*],*LT*)
assumes ⋀*l l′ l″ l‴ ST LT*. *P* (*l*#*l′*#*l″*#*l‴*#*ST*,*LT*)

shows *P s*
apply(*rule length-cases4*; (*rule assms*)*?*)
subgoal for *l l' l'' ST LT* **by**(*cases ST*; *clarsimp simp*: *assms*)
done

simp rules for *app*

lemma *appNone*[*simp*]: *app i P mxs* T_r *pc mpc et None* = *True*
by (*simp add*: *app-def*)

lemma *appLoad*[*simp*]:
app$_i$ (*Load idx*, *P*, T_r, *mxs*, *pc*, *s*) = (∃ *ST LT*. *s* = (*ST*,*LT*) ∧ *idx* < *length LT* ∧ *LT*!*idx* ≠ *Err* ∧ *length ST* < *mxs*)
by (*cases s*, *simp*)

lemma *appStore*[*simp*]:
app$_i$ (*Store idx*,*P*,*pc*,*mxs*,T_r,*s*) = (∃ *ts ST LT*. *s* = (*ts*#*ST*,*LT*) ∧ *idx* < *length LT*)
by (*rule length-cases2*, *auto*)

lemma *appPush*[*simp*]:
app$_i$ (*Push v*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *ST LT*. *s* = (*ST*,*LT*) ∧ *length ST* < *mxs* ∧ *typeof v* ≠ *None*)
by (*cases s*, *simp*)

lemma *appGetField*[*simp*]:
app$_i$ (*Getfield F C*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *oT vT ST LT fm*. *s* = (*oT*#*ST*, *LT*) ∧
P ⊢ *C sees F*:*vT* (*fm*) *in C* ∧ *P* ⊢ *oT* ≤ (*Class C*))
by (*rule length-cases2* [*of* - *s*]) *auto*

lemma *appPutField*[*simp*]:
app$_i$ (*Putfield F C*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *vT vT' oT ST LT fm*. *s* = (*vT*#*oT*#*ST*, *LT*) ∧
P ⊢ *C sees F*:*vT'* (*fm*) *in C* ∧ *P* ⊢ *oT* ≤ (*Class C*) ∧ *P* ⊢ *vT* ≤ *vT'*)
by (*rule length-cases4* [*of* - *s*], *auto*)

lemma *appCAS*[*simp*]:
app$_i$ (*CAS F C*, *P*, *pc*, *mxs*, T_r, *s*) =
(∃ *T1 T2 T3 T' ST LT fm*. *s* = (*T3* # *T2* # *T1* # *ST*, *LT*) ∧
P ⊢ *C sees F*:*T'* (*fm*) *in C* ∧ *volatile fm* ∧ *P* ⊢ *T1* ≤ *Class C* ∧ *P* ⊢ *T2* ≤ *T'* ∧ *P* ⊢ *T3* ≤ *T'*)
by(*rule length-cases4* [*of* - *s*]) *auto*

lemma *appNew*[*simp*]:
app$_i$ (*New C*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *ST LT*. *s*=(*ST*,*LT*) ∧ *is-class P C* ∧ *length ST* < *mxs*)
by (*cases s*, *simp*)

lemma *appNewArray*[*simp*]:
app$_i$ (*NewArray Ty*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *ST LT*. *s*=(*Integer*#*ST*,*LT*) ∧ *is-type P* (*Ty*⌊⌋))
by (*cases s*, *simp*, *cases fst s*, *simp*)(*cases hd* (*fst s*), *auto*)

lemma *appALoad*[*simp*]:
app$_i$ (*ALoad*,*P*,*pc*,*mxs*,T_r,*s*) =

$(\exists\, T\ ST\ LT.\ s{=}(\mathit{Integer}\#T\#ST,LT) \wedge (T \neq NT \longrightarrow (\exists\, T'.\ \ T = T'\lfloor\rceil)))$
proof −
obtain *ST LT* **where** [*simp*]: $s = (ST,\ LT)$ **by** (*cases s*)
have $ST = [] \vee (\exists\, T.\ ST = [T]) \vee (\exists\, T_1\ T_2\ ST'.\ ST = T_1\#T_2\#ST')$
by (*cases ST*, *auto*, *case-tac list*, *auto*)
moreover
{ assume $ST = []$ **hence** *?thesis* **by** *simp* **}**
moreover
{ fix *T* **assume** $ST = [T]$ **hence** *?thesis* **by** (*cases T*, *auto*) **}**
moreover
{ fix $T_1\ T_2\ ST'$ **assume** $ST = T_1\#T_2\#ST'$
hence *?thesis* **by** (*cases* T_1, *auto*)
}
ultimately show *?thesis* **by** *blast*
qed

lemma *appAStore*[*simp*]:
$\mathit{app}_i\ (\mathit{AStore},P,pc,mxs,T_r,s) =$
$(\exists\, T\ U\ ST\ LT.\ s{=}(T\#\mathit{Integer}\#U\#ST,LT) \wedge (U \neq NT \longrightarrow (\exists\, T'.\ U = T'\lfloor\rceil)))$
proof −
obtain *ST LT* **where** [*simp*]: $s = (ST,\ LT)$ **by** (*cases s*)
have $ST = [] \vee (\exists\, T.\ ST = [T]) \vee (\exists\, T_1\ T_2.\ ST = [T_1,\ T_2]) \vee (\exists\, T1\ T2\ T3\ ST'.\ ST = T1\ \#\ T2\ \#\ T3\ \#\ ST')$
by (*cases ST*, *auto*, *case-tac list*, *auto*, *case-tac lista*, *auto*)
moreover
{ assume $ST = []$ **hence** *?thesis* **by** *simp* **}**
moreover
{ fix *T* **assume** $ST = [T]$ **hence** *?thesis* **by**(*simp*) **}**
moreover
{ fix *T1 T2* **assume** $ST = [T1,\ T2]$ **hence** *?thesis* **by** *simp* **}**
moreover
{ fix *T1 T2 T3 ST'* **assume** $ST = T1\ \#\ T2\ \#\ T3\ \#\ ST'$ **hence** *?thesis* **by**(*cases T2*, *auto*) **}**
ultimately show *?thesis* **by** *blast*
qed

lemma *appALength*[*simp*]:
$\mathit{app}_i\ (\mathit{ALength},P,pc,mxs,T_r,s) =$
$(\exists\, T\ ST\ LT.\ s{=}(T\#ST,LT) \wedge (T \neq NT \longrightarrow (\exists\, T'.\ \ T = T'\lfloor\rceil)))$
by (*cases s*, *cases fst s*, *simp add*: *app-def*) (*cases hd* (*fst s*), *auto*)

lemma *appCheckcast*[*simp*]:
$\mathit{app}_i\ (\mathit{Checkcast}\ Ty,P,pc,mxs,T_r,s) =$
$(\exists\, T\ ST\ LT.\ s = (T\#ST,LT) \wedge \mathit{is\text{-}type}\ P\ Ty)$
by (*cases s*, *cases fst s*, *simp add*: *app-def*) (*cases hd* (*fst s*), *auto*)

lemma *appInstanceof*[*simp*]:
$\mathit{app}_i\ (\mathit{Instanceof}\ Ty,P,pc,mxs,T_r,s) =$
$(\exists\, T\ ST\ LT.\ s = (T\#ST,LT) \wedge \mathit{is\text{-}type}\ P\ Ty \wedge \mathit{is\text{-}refT}\ T)$
by (*cases s*, *cases fst s*, *simp add*: *app-def*) (*cases hd* (*fst s*), *auto*)

lemma $\mathit{app}_i\mathit{Pop}$[*simp*]:
$\mathit{app}_i\ (\mathit{Pop},P,pc,mxs,T_r,s) = (\exists\, ts\ ST\ LT.\ s = (ts\#ST,LT))$
by (*rule length-cases2*, *auto*)

lemma *appDup*[*simp*]:
app_i (*Dup*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *T ST LT*. *s* = (*T*#*ST*,*LT*) ∧ *Suc* (*length ST*) < *mxs*)
by (*cases s*, *cases fst s*, *simp-all*)

lemma *app*$_i$*Swap*[*simp*]:
app_i (*Swap*,*P*,*pc*,*mxs*,T_r,*s*) = (∃ *T1 T2 ST LT*. *s* = (*T1*#*T2*#*ST*,*LT*))
by(*rule length-cases4*) *auto*

lemma *appBinOp*[*simp*]:
app_i (*BinOpInstr bop*,*P*,*pc*,*mxs*,T_r,*s*) = (∃ *T1 T2 ST LT T*. *s* = (*T2* # *T1* # *ST*, *LT*) ∧ *P* ⊢ *T1* «*bop*» *T2* : *T*)
proof −
obtain *ST LT* **where** [*simp*]: *s* = (*ST*,*LT*) **by** (*cases s*)
have *ST* = [] ∨ (∃ *T*. *ST* = [*T*]) ∨ (∃ T_1 T_2 *ST'*. *ST* = T_1#T_2#*ST'*)
by (*cases ST*, *auto*, *case-tac list*, *auto*)
moreover
{ assume *ST* = [] **hence** *?thesis* **by** *simp* **}**
moreover
{ fix *T* **assume** *ST* = [*T*] **hence** *?thesis* **by** (*cases T*, *auto*) **}**
moreover
{ fix T_1 T_2 *ST'* **assume** *ST* = T_1#T_2#*ST'*
hence *?thesis* **by** *simp*
}
ultimately show *?thesis* **by** *blast*
qed

lemma *appIfFalse* [*simp*]:
app_i (*IfFalse b*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *ST LT*. *s* = (*Boolean*#*ST*,*LT*) ∧ *0* ≤ *int pc* + *b*)
apply (*rule length-cases2*)
apply *simp*
apply (*case-tac l*)
apply *auto*
done

lemma *appReturn*[*simp*]:
app_i (*Return*,*P*,*pc*,*mxs*,T_r,*s*) = (∃ *T ST LT*. *s* = (*T*#*ST*,*LT*) ∧ *P* ⊢ *T* ≤ T_r)
by (*rule length-cases2*, *auto*)

lemma *appThrow*[*simp*]:
app_i (*ThrowExc*,*P*,*pc*,*mxs*,T_r,*s*) = (∃ *T ST LT*. *s*=(*T*#*ST*,*LT*) ∧ (*T* = *NT* ∨ (∃ *C*. *T* = *Class C* ∧ *P* ⊢ *C* ≼* *Throwable*)))
by (*rule length-cases2*, *auto*)

lemma *appMEnter*[*simp*]:
app_i (*MEnter*,*P*,*pc*,*mxs*,T_r,*s*) = (∃ *T ST LT*. *s*=(*T*#*ST*,*LT*) ∧ *is-refT T*)
by (*rule length-cases2*, *auto*)

lemma *appMExit*[*simp*]:
app_i (*MExit*,*P*,*pc*,*mxs*,T_r,*s*) = (∃ *T ST LT*. *s*=(*T*#*ST*,*LT*) ∧ *is-refT T*)
by (*rule length-cases2*, *auto*)

lemma *effNone*:

$(pc', s') \in set\ (eff\ i\ P\ pc\ et\ None) \Longrightarrow s' = None$
by (*auto simp add*: *eff-def xcpt-eff-def norm-eff-def*)

lemma *relevant-entries-append* [*simp*]:
relevant-entries P i pc (*xt* @ *xt'*) = *relevant-entries P i pc xt* @ *relevant-entries P i pc xt'*
by (*unfold relevant-entries-def*) *simp*

lemma *xcpt-app-append* [*iff*]:
xcpt-app i P pc mxs (*xt*@*xt'*) τ = (*xcpt-app i P pc mxs xt* τ $\wedge$ *xcpt-app i P pc mxs xt'* τ)
unfolding *xcpt-app-def* **by** *force*

lemma *xcpt-eff-append* [*simp*]:
xcpt-eff i P pc τ (*xt*@*xt'*) = *xcpt-eff i P pc* τ *xt* @ *xcpt-eff i P pc* τ *xt'*
by (*unfold xcpt-eff-def*, *cases* τ) *simp*

lemma *app-append* [*simp*]:
app i P pc T mxs mpc (*xt*@*xt'*) τ = (*app i P pc T mxs mpc xt* τ $\wedge$ *app i P pc T mxs mpc xt'* τ)
by (*unfold app-def eff-def*) *auto*

6.2.1 Code generator setup

declare *list-all2-Nil* [*code*]
declare *list-all2-Cons* [*code*]

lemma *eff$_i$-BinOpInstr-code*:
eff$_i$ (*BinOpInstr bop*, *P*, (*T2*#*T1*#*ST*,*LT*)) = (*Predicate.the* (*WTrt-binop-i-i-i-i-o P T1 bop T2*) # *ST*, *LT*)
by(*simp add*: *the-WTrt-binop-code*)

lemmas *eff$_i$-code*[*code*] =
eff$_i$-Load eff$_i$-Store eff$_i$-Push eff$_i$-Getfield eff$_i$-Putfield eff$_i$-New eff$_i$-NewArray eff$_i$-ALoad
eff$_i$-AStore eff$_i$-ALength eff$_i$-Checkcast eff$_i$-Instanceof eff$_i$-Pop eff$_i$-Dup eff$_i$-Swap eff$_i$-BinOpInstr-code
eff$_i$-IfFalse eff$_i$-Invoke eff$_i$-Goto eff$_i$-MEnter eff$_i$-MExit

lemma *app$_i$-Getfield-code*:
app$_i$ (*Getfield F C*, *P*, *pc*, *mxs*, T_r, (*T*#*ST*, *LT*)) $\longleftrightarrow$
Predicate.holds (*Predicate.bind* (*sees-field-i-i-i-o-o-i P C F C*) (λT. *Predicate.single* ())) $\wedge$ *P* $\vdash$ *T* $\leq$ *Class C*
apply(*clarsimp simp add*: *Predicate.bind-def Predicate.single-def holds-eq eval-sees-field-i-i-i-o-i-conv*)
done

lemma *app$_i$-Putfield-code*:
app$_i$ (*Putfield F C*, *P*, *pc*, *mxs*, T_r, (T_1#T_2#*ST*, *LT*)) $\longleftrightarrow$
P $\vdash$ T_2 $\leq$ (*Class C*) $\wedge$
Predicate.holds (*Predicate.bind* (*sees-field-i-i-i-o-o-i P C F C*) (λ(*T*, *fm*). *if P* $\vdash$ T_1 $\leq$ *T then Predicate.single* () *else bot*))
by (*auto simp add*: *holds-eq eval-sees-field-i-i-i-o-i-conv split*: *if-splits*)

lemma *app$_i$-CAS-code*:
app$_i$ (*CAS F C*, *P*, *pc*, *mxs*, T_r, (T_3#T_2#T_1#*ST*, *LT*)) $\longleftrightarrow$
P $\vdash$ T_1 $\leq$ *Class C* $\wedge$
Predicate.holds (*Predicate.bind* (*sees-field-i-i-i-o-o-i P C F C*) (λ(*T*, *fm*). *if P* $\vdash$ T_2 $\leq$ *T* $\wedge$ *P* $\vdash$ T_3 $\leq$ *T* $\wedge$ *volatile fm then Predicate.single* () *else bot*))

by(*auto simp add*: *holds-eq eval-sees-field-i-i-i-o-i-conv*)

lemma *app$_i$-ALoad-code*:
app$_i$ (*ALoad*, *P*, *pc*, *mxs*, T_r, (*T1*#*T2*#*ST*,*LT*)) =
(*T1* = *Integer* ∧ (*case T2 of Ty*⌊⌉ ⇒ *True* | *NT* ⇒ *True* | - ⇒ *False*))
by(*simp add*: *split*: *ty.split*)

lemma *app$_i$-AStore-code*:
app$_i$ (*AStore*, *P*, *pc*, *mxs*, T_r, (*T1*#*T2*#*T3*#*ST*,*LT*)) =
(*T2* = *Integer* ∧ (*case T3 of Ty*⌊⌉ ⇒ *True* | *NT* ⇒ *True* | - ⇒ *False*))
by(*simp add*: *split*: *ty.split*)

lemma *app$_i$-ALength-code*:
app$_i$ (*ALength*, *P*, *pc*, *mxs*, T_r, (*T1*#*ST*,*LT*)) =
(*case T1 of Ty*⌊⌉ ⇒ *True* | *NT* ⇒ *True* | - ⇒ *False*)
by(*simp add*: *split*: *ty.split*)

lemma *app$_i$-BinOpInstr-code*:
app$_i$ (*BinOpInstr bop*, *P*, *pc*, *mxs*, T_r, (*T2*#*T1*#*ST*,*LT*)) =
Predicate.holds (*Predicate.bind* (*WTrt-binop-i-i-i-i-o P T1 bop T2*) (λ*T*. *Predicate.single* ()))
by (*auto simp add*: *holds-eq eval-WTrt-binop-i-i-i-i-o*)

lemma *app$_i$-Invoke-code*:
app$_i$ (*Invoke M n*, *P*, *pc*, *mxs*, T_r, (*ST*,*LT*)) =
(*n* < *length ST* ∧
(*ST*!*n* ≠ *NT* ⟶
(*case class-type-of'* (*ST* ! *n*) *of Some C* ⇒
Predicate.holds (*Predicate.bind* (*Method-i-i-i-o-o-o-o P C M*)
(λ(*Ts*, -). *if P* ⊢ *rev* (*take n ST*) [≤] *Ts then Predicate.single* () *else bot*))
| - ⇒ *False*)))
proof −
have *bind-Ex*: ⋀*P f*. *Predicate.bind P f* = *Predicate.Pred* (λ*x*. (∃ *y*. *Predicate.eval P y* ∧ *Predicate.eval* (*f y*) *x*))
by (*rule pred-eqI*) *auto*
thus *?thesis*
by (*auto simp add*: *bind-Ex Predicate.single-def holds-eq eval-Method-i-i-i-o-o-o-o-conv split*: *ty.split*)
qed

lemma *app$_i$-Throw-code*:
app$_i$ (*ThrowExc*, *P*, *pc*, *mxs*, T_r, (*T*#*ST*,*LT*)) =
(*case T of NT* ⇒ *True* | *Class C* ⇒ *P* ⊢ *C* ≼* *Throwable* | - ⇒ *False*)
by(*simp split*: *ty.split*)

lemmas *app$_i$-code* [*code*] =
app$_i$-Load app$_i$-Store app$_i$-Push
app$_i$-Getfield-code app$_i$-Putfield-code app$_i$-CAS-code
app$_i$-New app$_i$-NewArray
app$_i$-ALoad-code app$_i$-AStore-code app$_i$-ALength-code
app$_i$-Checkcast app$_i$-Instanceof
app$_i$-Pop app$_i$-Dup app$_i$-Swap app$_i$-BinOpInstr-code app$_i$-IfFalse app$_i$-Goto
app$_i$-Return app$_i$-Throw-code app$_i$-Invoke-code app$_i$-MEnter app$_i$-MExit
app$_i$-default

end

6.3 The Bytecode Verifier

theory *BVSpec*
imports
Effect
begin

This theory contains a specification of the BV. The specification describes correct typings of method bodies; it corresponds to type *checking*.

definition *check-types* :: $'m\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ list \Rightarrow bool$
where
check-types P mxs mxl $\tau s \equiv$ *set* $\tau s \subseteq$ *states P mxs mxl*

— An instruction is welltyped if it is applicable and its effect
— is compatible with the type at all successor instructions:
definition *wt-instr* :: $['m\ prog,ty,nat,pc,ex\text{-}table,'addr\ instr,pc,ty_m] \Rightarrow bool$
($\langle$-,-,-,-,- $\vdash$ -,- :: -$\rangle$ [60,0,0,0,0,0,0,61] 60)
where
$P,T,mxs,mpc,xt \vdash i,pc :: \tau s \equiv$
app i P mxs T pc mpc xt ($\tau s!pc$) $\wedge$
$(\forall (pc',\tau') \in$ *set* (*eff i P pc xt* ($\tau s!pc$)). $P \vdash \tau' \leq' \tau s!pc')$

— The type at *pc=0* conforms to the method calling convention:
definition *wt-start* :: $['m\ prog,cname,ty\ list,nat,ty_m] \Rightarrow bool$
where
wt-start P C Ts mxl_0 $\tau s \equiv$
$P \vdash$ *Some* ([],*OK* (*Class C*)#*map OK Ts*@*replicate* mxl_0 *Err*) $\leq' \tau s!0$

— A method is welltyped if the body is not empty,
— if the method type covers all instructions and mentions
— declared classes only, if the method calling convention is respected, and
— if all instructions are welltyped.
definition *wt-method* :: $['m\ prog,cname,ty\ list,ty,nat,nat,'addr\ instr\ list,\ ex\text{-}table,ty_m] \Rightarrow bool$
where
wt-method P C Ts T_r *mxs* mxl_0 *is xt* $\tau s \equiv$
0 $<$ *size is* $\wedge$ *size* τs = *size is* $\wedge$
check-types P mxs (*1*+*size Ts*+mxl_0) (*map OK* τs) $\wedge$
wt-start P C Ts mxl_0 τs $\wedge$
$(\forall\, pc <$ *size is*. $P,T_r,mxs,size\ is,xt \vdash is!pc,pc :: \tau s)$

— A program is welltyped if it is wellformed and all methods are welltyped
definition *wf-jvm-prog-phi* :: $ty_P \Rightarrow 'addr\ jvm\text{-}prog \Rightarrow bool$ ($\langle wf'\text{-}jvm'\text{-}prog_\text{-}\rangle$)
where
$wf\text{-}jvm\text{-}prog_\Phi \equiv$
wf-prog ($\lambda P\ C\ (M,Ts,T_r,(mxs,mxl_0,is,xt))$.
wt-method P C Ts T_r *mxs* mxl_0 *is xt* (Φ *C M*))

definition *wf-jvm-prog* :: $'addr\ jvm\text{-}prog \Rightarrow bool$
where
wf-jvm-prog P $\equiv \exists \Phi$. $wf\text{-}jvm\text{-}prog_\Phi$ *P*

lemma *wt-jvm-progD*:
wf-jvm-prog$_\Phi$ $P \Longrightarrow \exists\, wt.$ *wf-prog* wt P
lemma *wt-jvm-prog-impl-wt-instr*:
$[\![$ *wf-jvm-prog*$_\Phi$ P;
$P \vdash C$ *sees* M:$Ts \to T = \lfloor(mxs,mxl_0,ins,xt)\rfloor$ *in* C; $pc <$ *size ins* $]\!]$
$\Longrightarrow P,T,mxs,$*size ins*$,xt \vdash ins!pc,pc ::\ \Phi\ C\ M$
lemma *wt-jvm-prog-impl-wt-start*:
$[\![$ *wf-jvm-prog*$_\Phi$ P;
$P \vdash C$ *sees* M:$Ts \to T = \lfloor(mxs,mxl_0,ins,xt)\rfloor$ *in* C $]\!] \Longrightarrow$
$0 <$ *size ins* $\wedge$ *wt-start* $P\ C\ Ts\ mxl_0\ (\Phi\ C\ M)$
end

6.4 BV Type Safety Invariant

theory *BVConform*
imports
BVSpec
../JVM/JVMExec
begin

context *JVM-heap-base* **begin**

definition *confT* :: *'c prog* $\Rightarrow$ *'heap* $\Rightarrow$ *'addr val* $\Rightarrow$ *ty err* $\Rightarrow$ *bool*
(‹-,- ⊢ - :≤⊤ -› [51,51,51,51] 50)
where
$P,h \vdash v :\leq_\top E \equiv$ *case* E *of Err* $\Rightarrow$ *True* | *OK* $T \Rightarrow P,h \vdash v :\leq T$

notation (*ASCII*)
confT (‹-,- |- - :<=T -› [51,51,51,51] 50)

abbreviation *confTs* :: *'c prog* $\Rightarrow$ *'heap* $\Rightarrow$ *'addr val list* $\Rightarrow$ ty_l $\Rightarrow$ *bool*
(‹-,- ⊢ - [:≤⊤] -› [51,51,51,51] 50)
where
$P,h \vdash vs\ [:\leq_\top]\ Ts \equiv$ *list-all2* (*confT* P h) vs Ts

notation (*ASCII*)
confTs (‹-,- |- - [:<=T] -› [51,51,51,51] 50)

definition *conf-f* :: *'addr jvm-prog* $\Rightarrow$ *'heap* $\Rightarrow$ ty_i $\Rightarrow$ *'addr bytecode* $\Rightarrow$ *'addr frame* $\Rightarrow$ *bool*
where
conf-f P $h \equiv \lambda(ST,LT)$ *is* $(stk,loc,C,M,pc).$ $P,h \vdash stk\ [:\leq]\ ST \wedge P,h \vdash loc\ [:\leq_\top]\ LT \wedge pc <$ *size is*

primrec *conf-fs* :: [*'addr jvm-prog*,*'heap*,ty_P,*mname*,*nat*,*ty*,*'addr frame list*] $\Rightarrow$ *bool*
where
conf-fs P h Φ M_0 n_0 T_0 [] = *True*

| *conf-fs* P h Φ M_0 n_0 T_0 $(f\#frs) =$
(*let* $(stk,loc,C,M,pc) = f$ *in*
($\exists\, ST\ LT\ Ts\ T\ mxs\ mxl_0$ *is* $xt.$
$\Phi\ C\ M\ !\ pc =$ *Some* $(ST,LT) \wedge$
$(P \vdash C$ *sees* M:$Ts \to T = \lfloor(mxs,mxl_0,is,xt)\rfloor$ *in* $C) \wedge$
($\exists\, Ts'\ T'\ D\ m\ D'.$
$is!pc =$ (*Invoke* $M_0\ n_0$) $\wedge$ *class-type-of'* $(ST!n_0) = \lfloor D\rfloor \wedge P \vdash D$ *sees* M_0:$Ts' \to T' = m$ *in* D'

$\wedge\ P \vdash T_0 \leq T') \wedge$
$\quad$ *conf-f P h (ST, LT) is f* $\wedge$ *conf-fs P h* Φ *M (size Ts) T frs*))

primrec *conf-xcp* :: *'addr jvm-prog* $\Rightarrow$ *'heap* $\Rightarrow$ *'addr option* $\Rightarrow$ *'addr instr* $\Rightarrow$ *bool* **where**
conf-xcp P h None i = True
| *conf-xcp P h* $\lfloor a \rfloor$ *i* = ($\exists$ *D. typeof-addr h a* = $\lfloor$*Class-type D*$\rfloor$ $\wedge$ *P* $\vdash$ *D* $\preceq^*$ *Throwable* $\wedge$
($\forall$ *D'. P* $\vdash$ *D* $\preceq^*$ *D'* $\longrightarrow$ *is-relevant-class i P D'*))

end

context *JVM-heap-conf-base* **begin**

definition *correct-state* :: [ty_P,*'thread-id*,(*'addr*, *'heap*) *jvm-state*] $\Rightarrow$ *bool*
where
correct-state Φ *t* $\equiv$ λ(*xp*,*h*,*frs*).
P,*h* $\vdash$ *t* $\surd$*t* $\wedge$ *hconf h* $\wedge$ *preallocated h* $\wedge$
(*case frs of*
[] $\Rightarrow$ *True*
| (*f*#*fs*) $\Rightarrow$
(*let* (*stk*,*loc*,*C*,*M*,*pc*) = *f*
in $\exists$ *Ts T mxs mxl*$_0$ *is xt* τ.
(*P* $\vdash$ *C sees M*:*Ts*$\rightarrow$*T* = $\lfloor$(*mxs*,*mxl*$_0$,*is*,*xt*)$\rfloor$ *in C*) $\wedge$
Φ *C M* ! *pc* = *Some* τ $\wedge$
conf-f P h τ *is f* $\wedge$ *conf-fs P h* Φ *M (size Ts) T fs* $\wedge$
conf-xcp P h xp (*is* ! *pc*)))

notation
correct-state (‹- $\vdash$ -:- $\surd$› [*61*,*0*,*0*] *61*)

notation (*ASCII*)
correct-state (‹- |– -:- [*ok*]› [*61*,*0*,*0*] *61*)

end

context *JVM-heap-base* **begin**

lemma *conf-f-def2*:
conf-f P h (*ST*,*LT*) *is* (*stk*,*loc*,*C*,*M*,*pc*) $\equiv$
P,*h* $\vdash$ *stk* [:$\leq$] *ST* $\wedge$ *P*,*h* $\vdash$ *loc* [:$\leq_\top$] *LT* $\wedge$ *pc* < *size is*
by (*simp add*: *conf-f-def*)

6.4.1 Values and $\top$

lemma *confT-Err* [*iff*]: *P*,*h* $\vdash$ *x* :$\leq_\top$ *Err*
by (*simp add*: *confT-def*)

lemma *confT-OK* [*iff*]: *P*,*h* $\vdash$ *x* :$\leq_\top$ *OK T* = (*P*,*h* $\vdash$ *x* :$\leq$ *T*)
by (*simp add*: *confT-def*)

lemma *confT-cases*:
P,*h* $\vdash$ *x* :$\leq_\top$ *X* = (*X* = *Err* $\vee$ ($\exists$ *T*. *X* = *OK T* $\wedge$ *P*,*h* $\vdash$ *x* :$\leq$ *T*))
by (*cases X*) *auto*

lemma *confT-widen* [*intro?*, *trans*]:

⟦ $P,h \vdash x :\leq_\top T$; $P \vdash T \leq_\top T'$ ⟧ $\Longrightarrow P,h \vdash x :\leq_\top T'$
by (*cases* T', *auto intro*: *conf-widen*)

end

context *JVM-heap* **begin**

lemma *confT-hext* [*intro?*, *trans*]:
⟦ $P,h \vdash x :\leq_\top T$; $h \trianglelefteq h'$ ⟧ $\Longrightarrow P,h' \vdash x :\leq_\top T$
by (*cases* T) (*blast intro*: *conf-hext*)+

end

6.4.2 Stack and Registers

context *JVM-heap-base* **begin**

lemma *confTs-Cons1* [*iff*]:
$P,h \vdash x \# xs\ [:\leq_\top]\ Ts = (\exists z\ zs.\ Ts = z \# zs \wedge P,h \vdash x :\leq_\top z \wedge$ *list-all2* (*confT* $P\ h$) $xs\ zs)$
by(*rule list-all2-Cons1*)

lemma *confTs-confT-sup*:
⟦ $P,h \vdash loc\ [:\leq_\top]\ LT$; $n <$ *size* LT; $LT!n = OK\ T$; $P \vdash T \leq T'$ ⟧
$\Longrightarrow P,h \vdash (loc!n) :\leq T'$
apply (*frule list-all2-lengthD*)
apply (*drule list-all2-nthD*, *simp*)
apply *simp*
apply (*erule conf-widen*, *assumption*+)
done

lemma *confTs-widen* [*intro?*, *trans*]:
$P,h \vdash loc\ [:\leq_\top]\ LT \Longrightarrow P \vdash LT\ [\leq_\top]\ LT' \Longrightarrow P,h \vdash loc\ [:\leq_\top]\ LT'$
by (*rule list-all2-trans*, *rule confT-widen*)

lemma *confTs-map* [*iff*]:
$(P,h \vdash vs\ [:\leq_\top]\ map\ OK\ Ts) = (P,h \vdash vs\ [:\leq]\ Ts)$
by (*induct Ts arbitrary*: *vs*) (*auto simp add*: *list-all2-Cons2*)

lemma (**in** −) *reg-widen-Err*:
$(P \vdash replicate\ n\ Err\ [\leq_\top]\ LT) = (LT = replicate\ n\ Err)$
by (*induct n arbitrary*: *LT*) (*auto simp add*: *list-all2-Cons1*)

declare *reg-widen-Err* [*iff*]

lemma *confTs-Err* [*iff*]:
$P,h \vdash replicate\ n\ v\ [:\leq_\top]\ replicate\ n\ Err$
by (*induct n*) *auto*

end

context *JVM-heap* **begin**

lemma *confTs-hext* [*intro?*]:
$P,h \vdash loc\ [:\leq_\top]\ LT \Longrightarrow h \trianglelefteq h' \Longrightarrow P,h' \vdash loc\ [:\leq_\top]\ LT$

by (*fast elim*: *list-all2-mono confT-hext*)

6.4.3 correct-frames

declare *fun-upd-apply*[*simp del*]

lemma *conf-f-hext*:
⟦ *conf-f P h* Φ *M f*; $h \trianglelefteq h'$ ⟧ ⟹ *conf-f P h'* Φ *M f*
by(*cases f*, *cases* Φ, *auto simp add*: *conf-f-def intro*: *confs-hext confTs-hext*)

lemma *conf-fs-hext*:
⟦ *conf-fs P h* Φ *M n* T_r *frs*; $h \trianglelefteq h'$ ⟧ ⟹ *conf-fs P h'* Φ *M n* T_r *frs*
apply (*induct frs arbitrary*: *M n* T_r)
apply *simp*
apply *clarify*
apply (*simp* (*no-asm-use*))
apply *clarify*
apply (*unfold conf-f-def*)
apply (*simp* (*no-asm-use*) *split*: *if-split-asm*)
apply (*fast elim*!: *confs-hext confTs-hext*)+
done

declare *fun-upd-apply*[*simp*]

lemma *conf-xcp-hext*:
⟦ *conf-xcp P h xcp i*; $h \trianglelefteq h'$ ⟧ ⟹ *conf-xcp P h' xcp i*
by(*cases xcp*)(*auto elim*: *typeof-addr-hext-mono*)

end

context *JVM-heap-conf-base* **begin**

lemmas *defs1* = *correct-state-def conf-f-def wt-instr-def eff-def norm-eff-def app-def xcpt-app-def*

lemma *correct-state-impl-Some-method*:
Φ ⊢ *t*: (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)√
⟹ ∃ *m Ts T*. *P* ⊢ *C sees M*:*Ts*→*T* = ⌊*m*⌋ *in C*
by(*fastforce simp add*: *defs1*)

end

context *JVM-heap-conf-base'* **begin**

lemma *correct-state-hext-mono*:
⟦ Φ ⊢ *t*: (*xcp*, *h*, *frs*) √; $h \trianglelefteq h'$; *hconf h'* ⟧ ⟹ Φ ⊢ *t*: (*xcp*, *h'*, *frs*) √
unfolding *correct-state-def*
by(*fastforce elim*: *tconf-hext-mono preallocated-hext conf-f-hext conf-fs-hext conf-xcp-hext split*: *list.split*)

end

end

6.5 BV Type Safety Proof

theory *BVSpecTypeSafe*
imports
BVConform
../Common/ExternalCallWF
begin

declare *listE-length* [*simp del*]

This theory contains proof that the specification of the bytecode verifier only admits type safe programs.

6.5.1 Preliminaries

Simp and intro setup for the type safety proof:

context *JVM-heap-conf-base* **begin**

lemmas *widen-rules* [*intro*] = *conf-widen confT-widen confs-widens confTs-widen*

end

6.5.2 Exception Handling

For the *Invoke* instruction the BV has checked all handlers that guard the current *pc*.

lemma *Invoke-handlers*:
match-ex-table P C pc xt = *Some* (*pc'*,*d'*) ⟹
∃(*f*,*t*,*D*,*h*,*d*) ∈ *set* (*relevant-entries P* (*Invoke n M*) *pc xt*).
(*case D of None* ⇒ *True* | *Some D'* ⇒ *P* ⊢ *C* ≼* *D'*) ∧ *pc* ∈ {*f*..<*t*} ∧ *pc'* = *h* ∧ *d'* = *d*
by (*induct xt*) (*auto simp add*: *relevant-entries-def matches-ex-entry-def*
is-relevant-entry-def split: *if-split-asm*)

lemma *match-is-relevant*:
assumes *rv*: ⋀*D'*. *P* ⊢ *D* ≼* *D'* ⟹ *is-relevant-class* (*ins* ! *i*) *P D'*
assumes *match*: *match-ex-table P D pc xt* = *Some* (*pc'*,*d'*)
shows ∃(*f*,*t*,*D'*,*h*,*d*) ∈ *set* (*relevant-entries P* (*ins* ! *i*) *pc xt*). (*case D' of None* ⇒ *True* | *Some D''*
⇒ *P* ⊢ *D* ≼* *D''*) ∧ *pc* ∈ {*f*..<*t*} ∧ *pc'* = *h* ∧ *d'* = *d*
using *rv match*
by(*fastforce simp add*: *relevant-entries-def is-relevant-entry-def matches-ex-entry-def dest*: *match-ex-table-SomeD*)

context *JVM-heap-conf-base* **begin**

lemma *exception-step-conform*:
fixes *σ'* :: (*'addr*, *'heap*) *jvm-state*
assumes *wtp*: *wf-jvm-prog*$_\Phi$ *P*
assumes *correct*: Φ ⊢ *t*:(⌊*xcp*⌋, *h*, *fr* # *frs*) √
shows Φ ⊢ *t*:*exception-step P xcp h fr frs* √
proof −
obtain *stk loc C M pc* **where** *fr*: *fr* = (*stk*, *loc*, *C*, *M*, *pc*) **by**(*cases fr*)
from *correct* **obtain** *Ts T mxs mxl*$_0$ *ins xt*
where *meth*: *P* ⊢ *C sees M*:*Ts* → *T* = ⌊(*mxs*,*mxl*$_0$,*ins*,*xt*)⌋ *in C*
by (*simp add*: *correct-state-def fr*) *blast*

from *correct meth fr* **obtain** *D*
where *hxcp*: *typeof-addr h xcp* = ⌊*Class-type D*⌋ **and** *DsubThrowable*: *P* ⊢ *D* ≼* *Throwable*
and *rv*: ⋀*D'*. *P* ⊢ *D* ≼* *D'* ⟹ *is-relevant-class* (*instrs-of P C M* ! *pc*) *P D'*
by(*fastforce simp add*: *correct-state-def dest*: *sees-method-fun*)

from *meth* **have** [*simp*]: *ex-table-of P C M* = *xt* **by** *simp*

from *correct* **have** *tconf*: *P*,*h* ⊢ *t* √*t* **by**(*simp add*: *correct-state-def*)

show *?thesis*
proof(*cases match-ex-table P D pc xt*)
case *None*
with *correct fr meth hxcp* **show** *?thesis*
by(*fastforce simp add*: *correct-state-def cname-of-def split*: *list.split*)
next
case (*Some pc-d*)
then obtain *pc'* *d'* **where** *pcd*: *pc-d* = (*pc'*, *d'*)
and *match*: *match-ex-table P D pc xt* = *Some* (*pc'*,*d'*) **by** (*cases pc-d*) *auto*
from *match-is-relevant*[*OF rv match*] *meth* **obtain** *f t D'*
where *rv*: (*f*, *t*, *D'*, *pc'*, *d'*) ∈ *set* (*relevant-entries P* (*ins* ! *pc*) *pc xt*)
and *DsubD'*: (*case D' of None* ⇒ *True* | *Some D''* ⇒ *P* ⊢ *D* ≼* *D''*) **and** *pc*: *pc* ∈ {*f*..<*t*}
by(*auto*)

from *correct meth* **obtain** *ST LT*
where *h-ok*: *hconf h*
and Φ-*pc*: Φ *C M* ! *pc* = *Some* (*ST*, *LT*)
and *frame*: *conf-f P h* (*ST*,*LT*) *ins* (*stk*,*loc*,*C*,*M*,*pc*)
and *frames*: *conf-fs P h* Φ *M* (*size Ts*) *T frs*
and *preh*: *preallocated h*
unfolding *correct-state-def fr* **by**(*auto dest*: *sees-method-fun*)

from *frame* **obtain** *stk*: *P*,*h* ⊢ *stk* [:≤] *ST*
and *loc*: *P*,*h* ⊢ *loc* [:≤⊤] *LT* **and** *pc*: *pc* < *size ins*
by (*unfold conf-f-def*) *auto*

from *stk* **have** [*simp*]: *size stk* = *size ST* **..**

from *wtp meth correct fr* **have** *wt*: *P*,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*
by (*auto simp add*: *correct-state-def conf-f-def*
dest: *sees-method-fun*
elim!: *wt-jvm-prog-impl-wt-instr*)

from *wt* Φ-*pc* **have**
eff: ∀ (*pc'*, *s'*)∈*set* (*xcpt-eff* (*ins*!*pc*) *P pc* (*ST*,*LT*) *xt*).
pc' < *size ins* ∧ *P* ⊢ *s'* ≤' Φ *C M*!*pc'*
by (*auto simp add*: *defs1*)

let *?stk'* = *Addr xcp* # *drop* (*length stk* − *d'*) *stk*
let *?f* = (*?stk'*, *loc*, *C*, *M*, *pc'*)

have *conf*: *P*,*h* ⊢ *Addr xcp* :≤ *Class* (*case D' of None* ⇒ *Throwable* | *Some D''* ⇒ *D''*)
using *DsubD' hxcp DsubThrowable* **by**(*auto simp add*: *conf-def*)

```
    obtain ST' LT' where
      Φ-pc': Φ C M ! pc' = Some (ST', LT') and
      pc':   pc' < size ins and
      less:  P ⊢ (Class D # drop (size ST − d') ST, LT) ≤_i (ST', LT')
    proof(cases D')
      case Some
      thus ?thesis using eff rv DsubD' conf that
        by(fastforce simp add: xcpt-eff-def sup-state-opt-any-Some intro: widen-trans[OF widen-subcls])
    next
      case None
      with that eff rv conf DsubThrowable show ?thesis
        by(fastforce simp add: xcpt-eff-def sup-state-opt-any-Some intro: widen-trans[OF widen-subcls])
    qed

    with conf loc stk hxcp have conf-f P h (ST',LT') ins ?f
      by (auto simp add: defs1 conf-def intro: list-all2-dropI)

    with meth h-ok frames Φ-pc' fr match hxcp tconf preh
    show ?thesis unfolding correct-state-def
      by(fastforce dest: sees-method-fun simp add: cname-of-def)
  qed
qed

end
```

6.5.3 Single Instructions

In this subsection we prove for each single (welltyped) instruction that the state after execution of the instruction still conforms. Since we have already handled raised exceptions above, we can now assume that no exception has been raised in this step.

```
context JVM-conf-read begin

declare defs1 [simp]

lemma Invoke-correct:
  fixes σ' :: ('addr, 'heap) jvm-state
  assumes wtprog: wf-jvm-prog_Φ P
  assumes meth-C: P ⊢ C sees M:Ts→T=⌊(mxs,mxl_0,ins,xt)⌋ in C
  assumes ins:    ins ! pc = Invoke M' n
  assumes wti:    P,T,mxs,size ins,xt ⊢ ins!pc,pc :: Φ C M
  assumes approx: Φ ⊢ t:(None, h, (stk,loc,C,M,pc)#frs)√
  assumes exec: (tas, σ) ∈ exec-instr (ins!pc) P t h stk loc C M pc frs
  shows Φ ⊢ t:σ √
proof −
  note split-paired-Ex [simp del]

  from wtprog obtain wfmb where wfprog: wf-prog wfmb P
    by (simp add: wf-jvm-prog-phi-def)

  from ins meth-C approx obtain ST LT where
    heap-ok: hconf h and
    tconf:   P,h ⊢ t √t and
    Φ-pc:    Φ C M!pc = Some (ST,LT) and
```

```
  frame:  conf-f P h (ST,LT) ins (stk,loc,C,M,pc) and
  frames: conf-fs P h Φ M (size Ts) T frs and
  preh:   preallocated h
  by (fastforce dest: sees-method-fun)

from ins wti Φ-pc
have n: n < size ST by simp

show ?thesis
proof(cases stk!n = Null)
  case True
  with ins heap-ok Φ-pc frame frames exec meth-C tconf preh show ?thesis
    by(fastforce elim: wf-preallocatedE[OF wfprog, where C=NullPointer])
next
  case False
  note Null = this
  have NT: ST!n ≠ NT
  proof
    assume ST!n = NT
    moreover from frame have P,h ⊢ stk [:≤] ST by simp
    with n have P,h ⊢ stk!n :≤ ST!n by (simp add: list-all2-conv-all-nth)
    ultimately have stk!n = Null by simp
    with Null show False by contradiction
  qed

  from frame obtain
    stk: P,h ⊢ stk [:≤] ST and
    loc: P,h ⊢ loc [:≤⊤] LT by simp

  from NT ins wti Φ-pc have pc': pc+1 < size ins by simp

  from NT ins wti Φ-pc obtain ST' LT'
    where pc': pc+1 < size ins
    and Φ': Φ C M ! (pc+1) = Some (ST', LT')
    and LT': P ⊢ LT [≤⊤] LT'
    by(auto simp add: neq-Nil-conv sup-state-opt-any-Some split: if-split-asm)
  with NT ins wti Φ-pc obtain D D' TTs TT m
    where D: class-type-of' (ST!n) = ⌊D⌋
    and m-D: P ⊢ D sees M': TTs→TT = m in D'
    and Ts:  P ⊢ rev (take n ST) [≤] TTs
    and ST': P ⊢ (TT # drop (n+1) ST) [≤] ST'
    by(auto)

  from n stk D have P,h ⊢ stk!n :≤ ST ! n
    by (auto simp add: list-all2-conv-all-nth)

  from ‹P,h ⊢ stk!n :≤ ST ! n› Null D
  obtain U a where
    Addr:   stk!n = Addr a and
    obj:    typeof-addr h a = Some U and
    UsubSTn: P ⊢ ty-of-htype U ≤ ST ! n
    by(cases stk ! n)(auto simp add: conf-def widen-Class)

  from D UsubSTn obtain C' where
```

```
  C': class-type-of' (ty-of-htype U) = ⌊C'⌋ and C'subD: P ⊢ C' ≼* D
  by(rule widen-is-class-type-of) simp

with wfprog m-D
obtain Ts' T' D'' meth' where
  m-C': P ⊢ C' sees M': Ts'→T' = meth' in D'' and
  T':   P ⊢ T' ≤ TT and
  Ts':  P ⊢ TTs [≤] Ts'
  by (auto dest: sees-method-mono)

from Ts n have [simp]: size TTs = n
  by (auto dest: list-all2-lengthD simp: min-def)
with Ts' have [simp]: size Ts' = n
  by (auto dest: list-all2-lengthD)

from m-C' wfprog
obtain mD'': P ⊢ D'' sees M':Ts'→T'=meth' in D''
  by (fast dest: sees-method-idemp)

{ fix mxs' mxl' ins' xt'
  assume [simp]: meth' = ⌊(mxs', mxl', ins', xt')⌋
  let ?loc' = Addr a # rev (take n stk) @ replicate mxl' undefined-value
  let ?f' = ([], ?loc', D'', M', 0)
  let ?f  = (stk, loc, C, M, pc)

  from Addr obj m-C' ins meth-C exec C' False
  have s': σ = (None, h, ?f' # ?f # frs) by(auto split: if-split-asm)

  moreover
  from wtprog mD''
  obtain start: wt-start P D'' Ts' mxl' (Φ D'' M') and ins': ins' ≠ []
    by (auto dest: wt-jvm-prog-impl-wt-start)
  then obtain LT₀ where LT₀: Φ D'' M' ! 0 = Some ([], LT₀)
    by (clarsimp simp add: wt-start-def defs1 sup-state-opt-any-Some)
  moreover
  have conf-f P h ([], LT₀) ins' ?f'
  proof −
    let ?LT = OK (Class D'') # (map OK Ts') @ (replicate mxl' Err)

    from stk have P,h ⊢ take n stk [:≤] take n ST ..
    hence P,h ⊢ rev (take n stk) [:≤] rev (take n ST) by simp
    also note Ts also note Ts' finally
    have P,h ⊢ rev (take n stk) [:≤⊤] map OK Ts' by simp
    also
    have P,h ⊢ replicate mxl' undefined-value [:≤⊤] replicate mxl' Err by simp
    also from m-C' have P ⊢ C' ≼* D'' by (rule sees-method-decl-above)
    from obj heap-ok have is-htype P U by (rule typeof-addr-is-type)
    with C' have P ⊢ ty-of-htype U ≤ Class C'
      by(cases U)(simp-all add: widen-array-object)
    with ‹P ⊢ C' ≼* D''› obj C' have P,h ⊢ Addr a :≤ Class D''
      by (auto simp add: conf-def intro: widen-trans)
    ultimately
    have P,h ⊢ ?loc' [:≤⊤] ?LT by simp
    also from start LT₀ have P ⊢ ... [≤⊤] LT₀ by (simp add: wt-start-def)
```

finally have $P,h \vdash ?loc' [:\leq_\top] LT_0$ **.**
thus *?thesis* **using** *ins'* **by** *simp*
qed
ultimately have *?thesis* **using** *s' Φ-pc approx meth-C m-D T' ins D tconf C' mD''*
by (*fastforce dest*: *sees-method-fun* [*of* - *C*]) **}**
moreover
{ assume [*simp*]: *meth' = Native*
with *wfprog m-C'* **have** $D''\bullet M'(Ts') :: T'$ **by**(*simp add*: *sees-wf-native*)
with *C' m-C'* **have** *nec*: *is-native P U M'* **by**(*auto intro*: *is-native.intros*)

from *ins n Addr obj exec m-C' C'*
obtain *va h' tas'* **where** *va*: (*tas'*, *va*, *h'*) ∈ *red-external-aggr P t a M'* (*rev* (*take n stk*)) *h*
and *σ*: *σ = extRet2JVM n h' stk loc C M pc frs va* **by**(*auto*)
from *va nec obj* **have** *hext*: *h ⊴ h'* **by**(*auto intro*: *red-external-aggr-hext*)
with *frames* **have** *frames'*: *conf-fs P h' Φ M* (*length Ts*) *T frs* **by**(*rule conf-fs-hext*)
from *preh hext* **have** *preh'*: *preallocated h'* **by**(*rule preallocated-hext*)
from *va nec obj tconf* **have** *tconf'*: $P,h' \vdash t\ \surd t$
by(*auto dest*: *red-external-aggr-preserves-tconf*)
from *hext obj* **have** *obj'*: *typeof-addr h' a = ⌊U⌋* **by**(*rule typeof-addr-hext-mono*)

from *stk* **have** $P,h \vdash take\ n\ stk\ [:\leq]\ take\ n\ ST$ **by**(*rule list-all2-takeI*)
then obtain *Us* **where** $map\ typeof_h\ (take\ n\ stk) = map\ Some\ Us\ P \vdash Us\ [\leq]\ take\ n\ ST$
by(*auto simp add*: *confs-conv-map*)
hence *Us*: $map\ typeof_h\ (rev\ (take\ n\ stk)) = map\ Some\ (rev\ Us)\ P \vdash rev\ Us\ [\leq]\ rev\ (take\ n\ ST)$
by− (*simp only*: *rev-map*[*symmetric*], *simp*)
from ‹$P \vdash rev\ Us\ [\leq]\ rev\ (take\ n\ ST)$› *Ts Ts'*
have $P \vdash rev\ Us\ [\leq]\ Ts'$ **by**(*blast intro*: *widens-trans*)
with *obj* ‹$map\ typeof_h\ (rev\ (take\ n\ stk)) = map\ Some\ (rev\ Us)$› *C' m-C'*
have *wtext'*: $P,h \vdash a\bullet M'(rev\ (take\ n\ stk)) : T'$ **by**(*simp add*: *external-WT'.intros*)
from *va* **have** *va'*: $P,t \vdash \langle a\bullet M'(rev\ (take\ n\ stk)),h\rangle\ -tas'\rightarrow ext\ \langle va,h'\rangle$
by(*unfold WT-red-external-list-conv*[*OF wfprog wtext' tconf*])
with *heap-ok wtext' tconf wfprog* **have** *heap-ok'*: *hconf h'* **by**(*auto dest*: *external-call-hconf*)

have *?thesis*
proof(*cases va*)
case (*RetExc a'*)
from *frame hext* **have** *conf-f P h'* (*ST*, *LT*) *ins* (*stk*, *loc*, *C*, *M*, *pc*) **by**(*rule conf-f-hext*)
with *σ tconf' heap-ok' meth-C Φ-pc frames' RetExc red-external-conf-extRet*[*OF wfprog va' wtext' heap-ok preh tconf*] *ins preh'*
show *?thesis* **by**(*fastforce simp add*: *conf-def widen-Class*)
next
case *RetStaySame*
from *frame hext* **have** *conf-f P h'* (*ST*, *LT*) *ins* (*stk*, *loc*, *C*, *M*, *pc*) **by**(*rule conf-f-hext*)
with *σ heap-ok' meth-C Φ-pc RetStaySame frames' tconf' preh'* **show** *?thesis* **by** *fastforce*
next
case (*RetVal v*)
with *σ* **have** *σ*: *σ* = (*None*, *h'*, (*v* # *drop* (*n+1*) *stk*, *loc*, *C*, *M*, *pc+1*) # *frs*) **by** *simp*
from *heap-ok wtext' va' RetVal preh tconf* **have** $P,h' \vdash v :\leq T'$
by(*auto dest*: *red-external-conf-extRet*[*OF wfprog*])
from *stk* **have** $P,h \vdash drop\ (n + 1)\ stk\ [:\leq]\ drop\ (n+1)\ ST$ **by**(*rule list-all2-dropI*)
hence $P,h' \vdash drop\ (n + 1)\ stk\ [:\leq]\ drop\ (n+1)\ ST$ **using** *hext* **by**(*rule confs-hext*)
with ‹$P,h' \vdash v :\leq T'$› **have** $P,h' \vdash v\ \#\ drop\ (n + 1)\ stk\ [:\leq]\ T'\ \#\ drop\ (n+1)\ ST$
by(*auto simp add*: *conf-def intro*: *widen-trans*)
also

```
      with NT ins wti Φ-pc Φ' nec False D m-D T'
      have P ⊢ (T' # drop (n + 1) ST) [≤] ST'
        by(auto dest: sees-method-fun intro: widen-trans)
      also from loc hext have P,h' ⊢ loc [:≤⊤] LT by(rule confTs-hext)
      hence P,h' ⊢ loc [:≤⊤] LT' using LT' by(rule confTs-widen)
      ultimately show ?thesis using ‹hconf h'› σ meth-C Φ' pc' frames' tconf' preh' by fastforce
    qed }
  ultimately show ?thesis by(cases meth') auto
 qed
qed

declare list-all2-Cons2 [iff]

lemma Return-correct:
 assumes wt-prog: wf-jvm-prog_Φ P
 assumes meth: P ⊢ C sees M:Ts→T=⌊(mxs,mxl_0,ins,xt)⌋ in C
 assumes ins: ins ! pc = Return
 assumes wt: P,T,mxs,size ins,xt ⊢ ins!pc,pc :: Φ C M
 assumes correct: Φ ⊢ t:(None, h, (stk,loc,C,M,pc)#frs)√
 assumes s': (tas, σ') ∈ exec P t (None, h, (stk,loc,C,M,pc)#frs)
 shows Φ ⊢ t:σ'√
proof −
 from wt-prog
 obtain wfmb where wf: wf-prog wfmb P by (simp add: wf-jvm-prog-phi-def)

 from meth ins s' correct
 have frs = [] ⟹ ?thesis by (simp add: correct-state-def)
 moreover
 { fix f frs' assume frs': frs = f#frs'
  moreover obtain stk' loc' C' M' pc' where
   f: f = (stk',loc',C',M',pc') by (cases f)
  moreover note meth ins s'
  ultimately
  have σ':
   σ' = (None,h,(hd stk#(drop (1+size Ts) stk'),loc',C',M',pc'+1)#frs')
   (is σ' = (None,h,?f'#frs'))
   by simp

  from correct meth
  obtain ST LT where
   h-ok:   hconf h and
   tconf: P,h ⊢ t √t and
   Φ-pc: Φ C M ! pc = Some (ST, LT) and
   frame:  conf-f P h (ST, LT) ins (stk,loc,C,M,pc) and
   frames: conf-fs P h Φ M (size Ts) T frs and
   preh: preallocated h
   by (auto dest: sees-method-fun)

  from Φ-pc ins wt
  obtain U ST_0 where ST = U # ST_0 P ⊢ U ≤ T
   by (simp add: wt-instr-def app-def) blast
  with wf frame
  have hd-stk: P,h ⊢ hd stk :≤ T by (auto simp add: conf-f-def)
```

```
    from f frs' frames
    obtain ST' LT' Ts'' T'' mxs' mxl₀' ins' xt' Ts' T' where
      Φ': Φ C' M' ! pc' = Some (ST', LT') and
      meth-C': P ⊢ C' sees M':Ts''→T''=⌊(mxs',mxl₀',ins',xt')⌋ in C' and
      ins': ins' ! pc' = Invoke M (size Ts) and
      D: ∃ D m D'. class-type-of' (ST' ! (size Ts)) = Some D ∧ P ⊢ D sees M: Ts'→T' = m in D' and
      T': P ⊢ T ≤ T' and
      frame': conf-f P h (ST',LT') ins' f and
      conf-fs: conf-fs P h Φ M' (size Ts'') T'' frs'
      by clarsimp blast

    from f frame' obtain
      stk': P,h ⊢ stk' [:≤] ST' and
      loc': P,h ⊢ loc' [:≤⊤] LT' and
      pc': pc' < size ins'
      by (simp add: conf-f-def)

    from wt-prog meth-C' pc'
    have wti: P,T'',mxs',size ins',xt' ⊢ ins'!pc',pc' :: Φ C' M'
      by (rule wt-jvm-prog-impl-wt-instr)

    obtain aTs ST'' LT'' where
      Φ-suc: Φ C' M' ! Suc pc' = Some (ST'', LT'') and
      less: P ⊢ (T' # drop (size Ts+1) ST', LT') ≤ᵢ (ST'', LT'') and
      suc-pc': Suc pc' < size ins'
      using ins' Φ' D T' wti
      by(fastforce simp add: sup-state-opt-any-Some split: if-split-asm)

    from hd-stk T' have hd-stk': P,h ⊢ hd stk :≤ T' ..

    have frame'':
      conf-f P h (ST'',LT'') ins' ?f'
    proof −
      from stk'
      have P,h ⊢ drop (1+size Ts) stk' [:≤] drop (1+size Ts) ST' ..
      moreover
      with hd-stk' less
      have P,h ⊢ hd stk # drop (1+size Ts) stk' [:≤] ST'' by auto
      moreover
      from wf loc' less have P,h ⊢ loc' [:≤⊤] LT'' by auto
      moreover note suc-pc'
      ultimately show ?thesis by (simp add: conf-f-def)
    qed

    with σ' frs' f meth h-ok hd-stk Φ-suc frames meth-C' Φ' tconf preh
    have ?thesis by (fastforce dest: sees-method-fun [of - C'])
  }
  ultimately
  show ?thesis by (cases frs) blast+
qed

declare sup-state-opt-any-Some [iff]
declare not-Err-eq [iff]
```

lemma *Load-correct*:
⟦ *wf-prog wt P*;
  $P \vdash C$ *sees* M:$Ts \to T = \lfloor (mxs, mxl_0, ins, xt) \rfloor$ *in* C;
  ins!*pc* = *Load idx*;
  $P, T, mxs, size\ ins, xt \vdash ins!pc, pc :: \Phi\ C\ M$;
  $\Phi \vdash t$:(*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$;
  (*tas*, σ') ∈ *exec P t* (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*) ⟧
⟹ $\Phi \vdash t$:σ' $\surd$
by (*fastforce dest*: *sees-method-fun* [*of* - *C*] *elim*!: *confTs-confT-sup*)

declare [[*simproc del*: *list-to-set-comprehension*]]

lemma *Store-correct*:
⟦ *wf-prog wt P*;
  $P \vdash C$ *sees* M:$Ts \to T = \lfloor (mxs, mxl_0, ins, xt) \rfloor$ *in* C;
  ins!*pc* = *Store idx*;
  $P, T, mxs, size\ ins, xt \vdash ins!pc, pc :: \Phi\ C\ M$;
  $\Phi \vdash t$:(*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$;
  (*tas*, σ') ∈ *exec P t* (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*) ⟧
⟹ $\Phi \vdash t$:$\sigma'\surd$
apply *clarsimp*
apply (*drule* (*1*) *sees-method-fun*)
apply *clarsimp*
apply (*blast intro*!: *list-all2-update-cong*)
done

lemma *Push-correct*:
⟦ *wf-prog wt P*;
  $P \vdash C$ *sees* M:$Ts \to T = \lfloor (mxs, mxl_0, ins, xt) \rfloor$ *in* C;
  ins!*pc* = *Push v*;
  $P, T, mxs, size\ ins, xt \vdash ins!pc, pc :: \Phi\ C\ M$;
  $\Phi \vdash t$:(*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$;
  (*tas*, σ') ∈ *exec P t* (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*) ⟧
⟹ $\Phi \vdash t$:$\sigma'\surd$
apply *clarsimp*
apply (*drule* (*1*) *sees-method-fun*)
apply *clarsimp*
apply (*blast dest*: *typeof-lit-conf*)
done

declare [[*simproc add*: *list-to-set-comprehension*]]

lemma *Checkcast-correct*:
⟦ *wf-jvm-prog*$_\Phi$ *P*;
  $P \vdash C$ *sees* M:$Ts \to T = \lfloor (mxs, mxl_0, ins, xt) \rfloor$ *in* C;
  ins!*pc* = *Checkcast D*;
  $P, T, mxs, size\ ins, xt \vdash ins!pc, pc :: \Phi\ C\ M$;
  $\Phi \vdash t$:(*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$;
  (*tas*, σ) ∈ *exec-instr* (*ins*!*pc*) *P t h stk loc C M pc frs* ⟧
⟹ $\Phi \vdash t$:σ $\surd$
using *wf-preallocatedD*[*of* λ*P C* (*M*, *Ts*, T_r, *mxs*, mxl_0, *is*, *xt*). *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* (Φ *C M*) *P h ClassCast*]
apply (*clarsimp simp add*: *wf-jvm-prog-phi-def split*: *if-split-asm*)
apply(*drule* (*1*) *sees-method-fun*)

apply(*fastforce simp add*: *conf-def intro*: *widen-trans*)
apply (*drule* (*1*) *sees-method-fun*)
apply(*fastforce simp add*: *conf-def intro*: *widen-trans*)
done

lemma *Instanceof-correct*:
⟦ *wf-jvm-prog*$_\Phi$ *P*;
P ⊢ *C sees M*:*Ts*→*T*=⌊(*mxs*,*mxl*$_0$,*ins*,*xt*)⌋ *in C*;
ins!*pc* = *Instanceof Ty*;
P,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*;
Φ ⊢ *t*:(*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)√;
(*tas*, *σ*) ∈ *exec-instr* (*ins*!*pc*) *P t h stk loc C M pc frs* ⟧
⟹ Φ ⊢ *t*:*σ* √
apply (*clarsimp simp add*: *wf-jvm-prog-phi-def split*: *if-split-asm*)
apply (*drule* (*1*) *sees-method-fun*)
apply *fastforce*
done

declare *split-paired-All* [*simp del*]

end

lemma *widens-Cons* [*iff*]:
P ⊢ (*T* # *Ts*) [≤] *Us* = (∃*z zs*. *Us* = *z* # *zs* ∧ *P* ⊢ *T* ≤ *z* ∧ *P* ⊢ *Ts* [≤] *zs*)
by(*rule list-all2-Cons1*)

context *heap-conf-base* **begin**

end

context *JVM-conf-read* **begin**

lemma *Getfield-correct*:
assumes *wf*: *wf-prog wt P*
assumes *mC*: *P* ⊢ *C sees M*:*Ts*→*T*=⌊(*mxs*,*mxl*$_0$,*ins*,*xt*)⌋ *in C*
assumes *i*: *ins*!*pc* = *Getfield F D*
assumes *wt*: *P*,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*
assumes *cf*: Φ ⊢ *t*:(*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)√
assumes *xc*: (*tas*, *σ*′) ∈ *exec-instr* (*ins*!*pc*) *P t h stk loc C M pc frs*

shows Φ ⊢ *t*:*σ*′√
proof −
from *mC cf* **obtain** *ST LT* **where**
h√: *hconf h* **and**
tconf: *P*,*h* ⊢ *t* √*t* **and**
Φ: Φ *C M* ! *pc* = *Some* (*ST*,*LT*) **and**
stk: *P*,*h* ⊢ *stk* [:≤] *ST* **and** *loc*: *P*,*h* ⊢ *loc* [:≤$_\top$] *LT* **and**
pc: *pc* < *size ins* **and**
fs: *conf-fs P h* Φ *M* (*size Ts*) *T frs* **and**
preh: *preallocated h*
by (*fastforce dest*: *sees-method-fun*)

from *i* Φ *wt* **obtain** *oT ST*′′ *vT ST*′ *LT*′ *vT*′ *fm* **where**

```
    oT: P ⊢ oT ≤ Class D and
    ST: ST = oT # ST'' and
    F:  P ⊢ D sees F:vT (fm) in D and
    pc': pc+1 < size ins  and
    Φ': Φ C M ! (pc+1) = Some (vT'#ST', LT') and
    ST': P ⊢ ST'' [≤] ST' and LT': P ⊢ LT [≤⊤] LT' and
    vT': P ⊢ vT ≤ vT'
    by fastforce

  from stk ST obtain ref stk' where
    stk': stk = ref#stk' and
    ref:  P,h ⊢ ref :≤ oT and
    ST'': P,h ⊢ stk' [:≤] ST''
    by auto

  show ?thesis
  proof(cases ref = Null)
    case True
    with tconf h√ i xc stk' mC fs Φ ST'' ref ST loc pc'
      wf-preallocatedD[OF wf, of h NullPointer] preh
    show ?thesis by(fastforce)
  next
    case False
    from ref oT have P,h ⊢ ref :≤ Class D ..
    with False obtain a U' D' where a: ref = Addr a
      and h: typeof-addr h a = Some U'
      and U': D' = class-type-of U' and D': P ⊢ D' ≼* D
      by (blast dest: non-npD2)

    { fix v
      assume read: heap-read h a (CField D F) v
      from D' F have has-field: P ⊢ D' has F:vT (fm) in D
        by (blast intro: has-field-mono has-visible-field)
      with h have P,h ⊢ a@CField D F : vT unfolding U' ..
      with read have v: P,h ⊢ v :≤ vT using h√
        by(rule heap-read-conf)

      from ST'' ST' have P,h ⊢ stk' [:≤] ST' ..
      moreover
      from v vT' have P,h ⊢ v :≤ vT' by blast
      moreover
      from loc LT' have P,h ⊢ loc [:≤⊤] LT' ..
      moreover
      note h√ mC Φ' pc' v fs tconf preh
      ultimately have Φ ⊢ t:(None, h, (v#stk',loc,C,M,pc+1)#frs) √ by fastforce }
    with a h i mC stk' xc
    show ?thesis by auto
  qed
qed

lemma Putfield-correct:
  assumes wf: wf-prog wt P
  assumes mC: P ⊢ C sees M:Ts→T=⌊(mxs,mxl0,ins,xt)⌋ in C
  assumes i:  ins!pc = Putfield F D
```

assumes *wt*: $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
assumes *cf*: $\Phi \vdash t{:}(None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$
assumes *xc*: $(tas,\ \sigma') \in exec\text{-}instr\ (ins!pc)\ P\ t\ h\ stk\ loc\ C\ M\ pc\ frs$
shows $\Phi \vdash t{:}\sigma'\ \surd$
proof −
from *mC cf* **obtain** *ST LT* **where**
$h\surd$: *hconf h* **and**
tconf: $P,h \vdash t\ \surd t$ **and**
Φ: $\Phi\ C\ M\ !\ pc = Some\ (ST,LT)$ **and**
stk: $P,h \vdash stk\ [:\leq]\ ST$ **and** *loc*: $P,h \vdash loc\ [:\leq_\top]\ LT$ **and**
pc: $pc < size\ ins$ **and**
fs: *conf-fs P h* Φ *M* (*size Ts*) *T frs* **and**
preh: *preallocated h*
by (*fastforce dest*: *sees-method-fun*)

from *i* Φ *wt* **obtain** *vT vT′ oT ST″ ST′ LT′ fm* **where**
ST: $ST = vT\ \#\ oT\ \#\ ST''$ **and**
field: $P \vdash D\ sees\ F{:}vT'\ (fm)\ in\ D$ **and**
oT: $P \vdash oT \leq Class\ D$ **and** vT: $P \vdash vT \leq vT'$ **and**
pc': $pc+1 < size\ ins$ **and**
Φ′: $\Phi\ C\ M!(pc+1) = Some\ (ST',LT')$ **and**
ST': $P \vdash ST''\ [\leq]\ ST'$ **and** LT': $P \vdash LT\ [\leq_\top]\ LT'$
by *clarsimp*

from *stk ST* **obtain** *v ref stk′* **where**
stk': $stk = v\#ref\#stk'$ **and**
v: $P,h \vdash v :\leq vT$ **and**
ref: $P,h \vdash ref :\leq oT$ **and**
ST'': $P,h \vdash stk'\ [:\leq]\ ST''$
by *auto*

show *?thesis*
proof(*cases ref* = *Null*)
case *True*
with *tconf* $h\surd$ *i xc stk′ mC fs* Φ *ST″ ref ST loc pc′ v*
wf-preallocatedD[*OF wf*, *of h NullPointer*] *preh*
show *?thesis* **by**(*fastforce*)
next
case *False*
from *ref oT* **have** $P,h \vdash ref :\leq Class\ D$ **..**
with *False* **obtain** *a U′ D′* **where**
a: $ref = Addr\ a$ **and** *h*: *typeof-addr h a* = *Some U′*
and U': $D' = class\text{-}type\text{-}of\ U'$ **and** D': $P \vdash D' \preceq^* D$
by (*blast dest*: *non-npD2*)

from *v vT* **have** vT': $P,h \vdash v :\leq vT'$ **..**

from *field D′* **have** *has-field*: $P \vdash D'\ has\ F{:}vT'\ (fm)\ in\ D$
by (*blast intro*: *has-field-mono has-visible-field*)
with *h* **have** *al*: $P,h \vdash a@CField\ D\ F : vT'$ **unfolding** *U′* **..**
let *?f′* = $(stk',loc,C,M,pc+1)$

{ fix *h′*
assume *write*: *heap-write h a* (*CField D F*) *v h′*

```
      hence hext: h ⊴ h' by(rule hext-heap-write)
      with preh have preallocated h' by(rule preallocated-hext)
      moreover
      from write h√ al vT' have hconf h' by(rule hconf-heap-write-mono)
      moreover
      from ST'' ST' have P,h ⊢ stk' [:≤] ST' ..
      from this hext have P,h' ⊢ stk' [:≤] ST' by (rule confs-hext)
      moreover
      from loc LT' have P,h ⊢ loc [:≤⊤] LT' ..
      from this hext have P,h' ⊢ loc [:≤⊤] LT' by (rule confTs-hext)
      moreover
      from fs hext
      have conf-fs P h' Φ M (size Ts) T frs by (rule conf-fs-hext)
      moreover
      note mC Φ' pc'
      moreover
      from tconf hext have P,h' ⊢ t √t by(rule tconf-hext-mono)
      ultimately have Φ ⊢ t:(None, h', ?f'#frs) √ by fastforce }
    with a h i mC stk' xc show ?thesis by(auto simp del: correct-state-def)
  qed
qed

lemma CAS-correct:
  assumes wf: wf-prog wt P
  assumes mC: P ⊢ C sees M:Ts→T=⌊(mxs,mxl0,ins,xt)⌋ in C
  assumes i:  ins!pc = CAS F D
  assumes wt: P,T,mxs,size ins,xt ⊢ ins!pc,pc :: Φ C M
  assumes cf: Φ ⊢ t:(None, h, (stk,loc,C,M,pc)#frs)√
  assumes xc: (tas, σ') ∈ exec-instr (ins!pc) P t h stk loc C M pc frs
  shows Φ ⊢ t:σ' √
proof −
  from mC cf obtain ST LT where
    h√: hconf h and
    tconf: P,h ⊢ t √t and
    Φ: Φ C M ! pc = Some (ST,LT) and
    stk: P,h ⊢ stk [:≤] ST and loc: P,h ⊢ loc [:≤⊤] LT and
    pc: pc < size ins and
    fs: conf-fs P h Φ M (size Ts) T frs and
    preh: preallocated h
    by (fastforce dest: sees-method-fun)

  from i Φ wt obtain T1 T2 T3 T' ST'' ST' LT' fm where
    ST: ST = T3 # T2 # T1 # ST'' and
    field: P ⊢ D sees F:T' (fm) in D and
    oT: P ⊢ T1 ≤ Class D and T2: P ⊢ T2 ≤ T' and T3: P ⊢ T3 ≤ T' and
    pc': pc+1 < size ins and
    Φ': Φ C M!(pc+1) = Some (Boolean # ST',LT') and
    ST': P ⊢ ST'' [≤] ST' and LT': P ⊢ LT [≤⊤] LT'
    by clarsimp

  from stk ST obtain v'' v' v stk' where
    stk': stk = v''#v'#v#stk' and
    v:    P,h ⊢ v :≤ T1 and
    v':   P,h ⊢ v' :≤ T2 and
```

```
    v'': P,h ⊢ v'' :≤ T3 and
    ST'': P,h ⊢ stk' [:≤] ST''
    by auto

  show ?thesis
  proof(cases v = Null)
    case True
    with tconf h√ i xc stk' mC fs Φ ST'' v ST loc pc' v' v''
      wf-preallocatedD[OF wf, of h NullPointer] preh
    show ?thesis by(fastforce)
  next
    case False
    from v oT have P,h ⊢ v :≤ Class D ..
    with False obtain a U' D' where
      a: v = Addr a and h: typeof-addr h a = Some U'
      and U': D' = class-type-of U' and D': P ⊢ D' ≼* D
      by (blast dest: non-npD2)

    from v' T2 have vT': P,h ⊢ v' :≤ T' ..
    from v'' T3 have vT'': P,h ⊢ v'' :≤ T' ..

    from field D' have has-field: P ⊢ D' has F:T' (fm) in D
      by (blast intro: has-field-mono has-visible-field)
    with h have al: P,h ⊢ a@CField D F : T' unfolding U' ..

    from ST'' ST' have stk'': P,h ⊢ stk' [:≤] ST' ..
    from loc LT' have loc': P,h ⊢ loc [:≤⊤] LT' ..
    { fix h'
      assume write: heap-write h a (CField D F) v'' h'
      hence hext: h ◁ h' by(rule hext-heap-write)
      with preh have preallocated h' by(rule preallocated-hext)
      moreover
      from write h√ al vT'' have hconf h' by(rule hconf-heap-write-mono)
      moreover
      from stk'' hext have P,h' ⊢ stk' [:≤] ST' by (rule confs-hext)
      moreover
      from loc' hext have P,h' ⊢ loc [:≤⊤] LT' by (rule confTs-hext)
      moreover
      from fs hext
      have conf-fs P h' Φ M (size Ts) T frs by (rule conf-fs-hext)
      moreover
      note mC Φ' pc'
      moreover
      let ?f' = (Bool True # stk',loc,C,M,pc+1)
      from tconf hext have P,h' ⊢ t √t by(rule tconf-hext-mono)
      ultimately have Φ ⊢ t:(None, h', ?f'#frs) √ by fastforce
    } moreover {
      let ?f' = (Bool False # stk',loc,C,M,pc+1)
      have Φ ⊢ t:(None, h, ?f'#frs) √ using tconf h√ preh mC Φ' stk'' loc' pc' fs
        by fastforce
    } ultimately show ?thesis using a h i mC stk' xc by(auto simp del: correct-state-def)
  qed
qed
```

lemma *New-correct*:
assumes *wf*: *wf-prog wt P*
assumes *meth*: $P \vdash C$ *sees* M:*Ts*$\rightarrow T = \lfloor (mxs, mxl_0, ins, xt) \rfloor$ *in* C
assumes *ins*: *ins*!*pc* = *New X*
assumes *wt*: $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
assumes *conf*: $\Phi \vdash t$:(*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$
assumes *no-x*: (*tas*, σ) $\in$ *exec-instr* (*ins*!*pc*) *P t h stk loc C M pc frs*
shows $\Phi \vdash t{:}\sigma\ \surd$
proof −
from *ins conf meth*
obtain *ST LT* **where**
heap-ok: *hconf h* **and**
tconf: $P,h \vdash t\ \surd t$ **and**
Φ-*pc*: $\Phi\ C\ M!pc = Some\ (ST,LT)$ **and**
frame: *conf-f P h* (*ST*,*LT*) *ins* (*stk*,*loc*,*C*,*M*,*pc*) **and**
frames: *conf-fs P h* Φ *M* (*size Ts*) *T frs* **and**
preh: *preallocated h*
by (*auto dest*: *sees-method-fun*)

from Φ-*pc ins wt*
obtain *ST′ LT′* **where**
is-class-X: *is-class P X* **and**
mxs: *size ST* < *mxs* **and**
suc-pc: *pc*+1 < *size ins* **and**
Φ-*suc*: Φ *C M*!(*pc*+1) = *Some* (*ST′*, *LT′*) **and**
less: $P \vdash (Class\ X\ \#\ ST,\ LT) \leq_i (ST',\ LT')$
by *auto*
show *?thesis*
proof(*cases allocate h* (*Class-type X*) = {})
case *True*
with *frame frames tconf suc-pc no-x ins meth* Φ-*pc*
wf-preallocatedD[*OF wf*, *of h OutOfMemory*] *preh is-class-X heap-ok*
show *?thesis*
by(*fastforce intro*: *tconf-hext-mono confs-hext confTs-hext conf-fs-hext*)
next
case *False*
with *ins meth no-x* **obtain** *h′ oref*
where *new*: (*h′*, *oref*) ∈ *allocate h* (*Class-type X*)
and σ': σ = (*None*, *h′*, (*Addr oref*#*stk*,*loc*,*C*,*M*,*pc*+1)#*frs*) (**is** σ = (*None*, *h′*, *?f* # *frs*))
by *auto*

from *new* **have** *hext*: *h* ⊴ *h′* **by**(*rule hext-allocate*)
with *preh* **have** *preh′*: *preallocated h′* **by**(*rule preallocated-hext*)
from *new heap-ok is-class-X* **have** *heap-ok′*: *hconf h′*
by(*auto intro*: *hconf-allocate-mono*)

with *new is-class-X* **have** *h′*: *typeof-addr h′ oref* = ⌊*Class-type X*⌋ **by**(*auto dest*: *allocate-SomeD*)

note *heap-ok′* σ'
moreover
from *frame less suc-pc wf h′ hext*
have *conf-f P h′* (*ST′*, *LT′*) *ins ?f*
apply (*clarsimp simp add*: *fun-upd-apply conf-def split-beta*)
apply (*auto intro*: *confs-hext confTs-hext*)

```
    done
  moreover
  from frames hext have conf-fs P h' Φ M (size Ts) T frs by (rule conf-fs-hext)
  moreover from tconf hext have P,h' ⊢ t √t by(rule tconf-hext-mono)
  ultimately
  show ?thesis using meth Φ-suc preh' by fastforce
 qed
qed

lemma Goto-correct:
⟦ wf-prog wt P;
  P ⊢ C sees M:Ts→T=⌊(mxs,mxl0,ins,xt)⌋ in C;
  ins ! pc = Goto branch;
  P,T,mxs,size ins,xt ⊢ ins!pc,pc :: Φ C M;
  Φ ⊢ t:(None, h, (stk,loc,C,M,pc)#frs)√;
  (tas, σ') ∈ exec P t (None, h, (stk,loc,C,M,pc)#frs) ⟧
⟹ Φ ⊢ t:σ' √
apply clarsimp
apply (drule (1) sees-method-fun)
apply fastforce
done

declare [[simproc del: list-to-set-comprehension]]

lemma IfFalse-correct:
⟦ wf-prog wt P;
  P ⊢ C sees M:Ts→T=⌊(mxs,mxl0,ins,xt)⌋ in C;
  ins ! pc = IfFalse branch;
  P,T,mxs,size ins,xt ⊢ ins!pc,pc :: Φ C M;
  Φ ⊢ t:(None, h, (stk,loc,C,M,pc)#frs)√;
  (tas, σ') ∈ exec P t (None, h, (stk,loc,C,M,pc)#frs) ⟧
⟹ Φ ⊢ t:σ'√
apply clarsimp
apply (drule (1) sees-method-fun)
apply fastforce
done

declare [[simproc add: list-to-set-comprehension]]

lemma BinOp-correct:
⟦ wf-prog wt P;
  P ⊢ C sees M:Ts→T=⌊(mxs,mxl0,ins,xt)⌋ in C;
  ins ! pc = BinOpInstr bop;
  P,T,mxs,size ins,xt ⊢ ins!pc,pc :: Φ C M;
  Φ ⊢ t:(None, h, (stk,loc,C,M,pc)#frs)√;
  (tas, σ') ∈ exec P t (None, h, (stk,loc,C,M,pc)#frs) ⟧
⟹ Φ ⊢ t:σ'√
apply clarsimp
apply (drule (1) sees-method-fun)
apply(clarsimp simp add: conf-def)
apply(drule (2) WTrt-binop-widen-mono)
apply clarsimp
apply(frule (2) binop-progress)
apply(clarsimp split: sum.split-asm)
```

apply(*frule* (*5*) *binop-type*)
apply(*fastforce intro*: *widen-trans simp add*: *conf-def*)
apply(*frule* (*5*) *binop-type*)
apply(*clarsimp simp add*: *conf-def*)
apply(*clarsimp simp add*: *widen-Class*)
apply(*fastforce intro*: *widen-trans dest*: *binop-relevant-class simp add*: *cname-of-def conf-def*)
done

lemma *Pop-correct*:
⟦ *wf-prog wt P*;
P ⊢ *C sees M*:*Ts*→*T*=⌊(*mxs*,*mxl*$_0$,*ins*,*xt*)⌋ *in C*;
ins ! *pc* = *Pop*;
P,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*;
Φ ⊢ *t*:(*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)√;
(*tas*, *σ*′) ∈ *exec P t* (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*) ⟧
⟹ Φ ⊢ *t*:*σ*′√
apply *clarsimp*
apply (*drule* (*1*) *sees-method-fun*)
apply *fastforce*
done

lemma *Dup-correct*:
⟦ *wf-prog wt P*;
P ⊢ *C sees M*:*Ts*→*T*=⌊(*mxs*,*mxl*$_0$,*ins*,*xt*)⌋ *in C*;
ins ! *pc* = *Dup*;
P,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*;
Φ ⊢ *t*:(*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)√;
(*tas*, *σ*′) ∈ *exec P t* (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*) ⟧
⟹ Φ ⊢ *t*:*σ*′√
apply *clarsimp*
apply (*drule* (*1*) *sees-method-fun*)
apply *fastforce*
done

lemma *Swap-correct*:
⟦ *wf-prog wt P*;
P ⊢ *C sees M*:*Ts*→*T*=⌊(*mxs*,*mxl*$_0$,*ins*,*xt*)⌋ *in C*;
ins ! *pc* = *Swap*;
P,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*;
Φ ⊢ *t*:(*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)√;
(*tas*, *σ*′) ∈ *exec P t* (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*) ⟧
⟹ Φ ⊢ *t*:*σ*′√
apply *clarsimp*
apply (*drule* (*1*) *sees-method-fun*)
apply *fastforce*
done

declare [[*simproc del*: *list-to-set-comprehension*]]

lemma *Throw-correct*:
⟦ *wf-prog wt P*;
P ⊢ *C sees M*:*Ts*→*T*=⌊(*mxs*,*mxl*$_0$,*ins*,*xt*)⌋ *in C*;
ins ! *pc* = *ThrowExc*;
P,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*;

```
  Φ ⊢ t:(None, h, (stk,loc,C,M,pc)#frs)√;
  (tas, σ') ∈ exec-instr (ins!pc) P t h stk loc C M pc frs ⟧
⟹ Φ ⊢ t:σ'√
using wf-preallocatedD[of wt P h NullPointer]
apply(clarsimp)
apply(drule (1) sees-method-fun)
apply(auto)
  apply fastforce
 apply fastforce
apply(drule (1) non-npD)
apply fastforce+
done

declare [[simproc add: list-to-set-comprehension]]

lemma NewArray-correct:
  assumes wf:   wf-prog wt P
  assumes meth: P ⊢ C sees M:Ts→T=⌊(mxs,mxl0,ins,xt)⌋ in C
  assumes ins:  ins!pc = NewArray X
  assumes wt:   P,T,mxs,size ins,xt ⊢ ins!pc,pc :: Φ C M
  assumes conf: Φ ⊢ t:(None, h, (stk,loc,C,M,pc)#frs)√
  assumes no-x: (tas, σ) ∈ exec-instr (ins!pc) P t h stk loc C M pc frs
  shows Φ ⊢ t:σ √
proof −
  from ins conf meth
  obtain ST LT where
    heap-ok: hconf h and
    tconf: P,h ⊢ t √t and
    Φ-pc:    Φ C M!pc = Some (ST,LT) and
    stk: P,h ⊢ stk [:≤] ST and loc: P,h ⊢ loc [:≤⊤] LT and
    pc: pc < size ins and
    frame:   conf-f P h (ST,LT) ins (stk,loc,C,M,pc) and
    frames:  conf-fs P h Φ M (size Ts) T frs and
    preh:    preallocated h
    by (auto dest: sees-method-fun)

  from ins Φ-pc wt obtain ST'' X' ST' LT' where
    ST: ST = Integer # ST'' and
    pc': pc+1 < size ins  and
    Φ': Φ C M ! (pc+1) = Some (X'#ST', LT') and
    ST': P ⊢ ST'' [≤] ST' and LT': P ⊢ LT [≤⊤] LT' and
    XX': P ⊢ X⌊⌋ ≤ X' and
    suc-pc:     pc+1 < size ins and
    is-type-X: is-type P (X⌊⌋)
    by(fastforce dest: Array-widen)

  from stk ST obtain si stk' where si: stk = Intg si # stk'
    by(auto simp add: conf-def)

  show ?thesis
  proof(cases si <s 0 ∨ allocate h (Array-type X (nat (sint si))) = {})
    case True
    with frame frames tconf heap-ok suc-pc no-x ins meth Φ-pc si preh
      wf-preallocatedD[OF wf, of h OutOfMemory] wf-preallocatedD[OF wf, of h NegativeArraySize]
```

```
    show ?thesis
      by(fastforce intro: tconf-hext-mono confs-hext confTs-hext conf-fs-hext split: if-split-asm)+
  next
    case False
    with ins meth si no-x obtain h' oref
      where new: (h', oref) ∈ allocate h (Array-type X (nat (sint si)))
      and σ': σ = (None, h', (Addr oref#tl stk,loc,C,M,pc+1)#frs) (is σ = (None, h', ?f # frs))
      by(auto split: if-split-asm)
    from new have hext: h ⊴ h' by(rule hext-allocate)
    with preh have preh': preallocated h' by(rule preallocated-hext)
    from new heap-ok is-type-X have heap-ok': hconf h' by(auto intro: hconf-allocate-mono)
    from False have si': 0 <=s si by auto
    with new is-type-X have h': typeof-addr h' oref = ⌊Array-type X (nat (sint si))⌋
      by(auto dest: allocate-SomeD)

    note σ' heap-ok'
    moreover
    from frame ST' ST LT' suc-pc wf XX' h' hext
    have conf-f P h' (X' # ST', LT') ins ?f
      by(clarsimp simp add: fun-upd-apply conf-def split-beta)(auto intro: confs-hext confTs-hext)
    moreover
    from frames hext have conf-fs P h' Φ M (size Ts) T frs by (rule conf-fs-hext)
    moreover from tconf hext have P,h' ⊢ t √t by(rule tconf-hext-mono)
    ultimately
    show ?thesis using meth Φ' preh' by fastforce
  qed
qed

lemma ALoad-correct:
  assumes wf:   wf-prog wt P
  assumes meth: P ⊢ C sees M:Ts→T=⌊(mxs,mxl0,ins,xt)⌋ in C
  assumes ins:  ins!pc = ALoad
  assumes wt:   P,T,mxs,size ins,xt ⊢ ins!pc,pc :: Φ C M
  assumes conf: Φ ⊢ t: (None, h, (stk,loc,C,M,pc)#frs)√
  assumes no-x: (tas, σ) ∈ exec-instr (ins!pc) P t h stk loc C M pc frs
  shows Φ ⊢ t:σ √
proof −
  from ins conf meth
  obtain ST LT where
    heap-ok: hconf h and
    tconf:   P,h ⊢ t √t and
    Φ-pc:    Φ C M!pc = Some (ST,LT) and
    stk: P,h ⊢ stk [:≤] ST and loc: P,h ⊢ loc [:≤⊤] LT and
    pc: pc < size ins and
    frame:   conf-f P h (ST,LT) ins (stk,loc,C,M,pc) and
    frames:  conf-fs P h Φ M (size Ts) T frs and
    preh:    preallocated h
    by (auto dest: sees-method-fun)

  from ins wt Φ-pc have lST: length ST > 1 by(auto)

  show ?thesis
  proof(cases hd (tl stk) = Null)
    case True
```

with *ins no-x heap-ok tconf* Φ*-pc stk loc frame frames meth wf-preallocatedD*[*OF wf*, *of h Null-Pointer*] *preh*
show *?thesis* **by**(*fastforce*)
next
case *False*
note *stkNN* = *this*
have *STNN*: *hd* (*tl ST*) ≠ *NT*
proof
assume *hd* (*tl ST*) = *NT*
moreover
from *frame* **have** *P*,*h* ⊢ *stk* [:≤] *ST* **by** *simp*
with *lST* **have** *P*,*h* ⊢ *hd* (*tl stk*) :≤ *hd* (*tl ST*)
by (*cases ST*, *auto*, *case-tac list*, *auto*)
ultimately
have *hd* (*tl stk*) = *Null* **by** *simp*
with *stkNN* **show** *False* **by** *contradiction*
qed

with *stkNN ins* Φ*-pc wt* **obtain** *ST*′′ *X X*′ *ST*′ *LT*′ **where**
ST: *ST* = *Integer* # *X*⌊⌋ # *ST*′′ **and**
pc′: *pc*+1 < *size ins* **and**
Φ′: Φ *C M* ! (*pc*+1) = *Some* (*X*′#*ST*′, *LT*′) **and**
ST′: *P* ⊢ *ST*′′ [≤] *ST*′ **and** *LT*′: *P* ⊢ *LT* [≤⊤] *LT*′ **and**
XX′: *P* ⊢ *X* ≤ *X*′ **and**
suc-pc: *pc*+1 < *size ins*
by(*fastforce*)

from *stk ST* **obtain** *ref idx stk*′ **where**
stk′: *stk* = *idx*#*ref*#*stk*′ **and**
idx: *P*,*h* ⊢ *idx* :≤ *Integer* **and**
ref: *P*,*h* ⊢ *ref* :≤ *X*⌊⌋ **and**
ST′′: *P*,*h* ⊢ *stk*′ [:≤] *ST*′′
by *auto*

from *stkNN stk*′ **have** *ref* ≠ *Null* **by**(*simp*)
with *ref* **obtain** *a Xel n*
where *a*: *ref* = *Addr a*
and *ha*: *typeof-addr h a* = ⌊*Array-type Xel n*⌋
and *Xel*: *P* ⊢ *Xel* ≤ *X*
by(*cases ref*)(*fastforce simp add*: *conf-def widen-Array*)+

from *idx* **obtain** *idxI* **where** *idxI*: *idx* = *Intg idxI*
by(*auto simp add*: *conf-def*)
show *?thesis*
proof(*cases* 0 <=*s idxI* ∧ *sint idxI* < *int n*)
case *True*
hence *si*′: 0 <=*s idxI sint idxI* < *int n* **by** *auto*
hence *nat* (*sint idxI*) < *n*
by (*simp add*: *word-sle-eq nat-less-iff*)
with *ha* **have** *al*: *P*,*h* ⊢ *a*@*ACell* (*nat* (*sint idxI*)) : *Xel* **..**

{ fix *v*
assume *read*: *heap-read h a* (*ACell* (*nat* (*sint idxI*))) *v*
hence *v*: *P*,*h* ⊢ *v* :≤ *Xel* **using** *al heap-ok* **by**(*rule heap-read-conf*)

```
      let ?f = (v # stk', loc, C, M, pc + 1)

      from frame ST' ST LT' suc-pc wf XX' Xel idxI si' v ST''
      have conf-f P h (X' # ST', LT') ins ?f
        by(auto intro: widen-trans simp add: conf-def)
      hence Φ ⊢ t:(None, h, ?f # frs) √
        using meth Φ' heap-ok Φ-pc frames tconf preh by fastforce }
     with ins meth si' stk' a ha no-x idxI idx
     show ?thesis by(auto simp del: correct-state-def split: if-split-asm)
   next
     case False
     with stk' idxI ins no-x heap-ok tconf meth a ha Xel Φ-pc frame frames
       wf-preallocatedD[OF wf, of h ArrayIndexOutOfBounds]
     show ?thesis
       by (fastforce simp: preh split: if-split-asm simp del: Listn.lesub-list-impl-same-size)
   qed
 qed
qed


lemma AStore-correct:
 assumes wf:   wf-prog wt P
 assumes meth: P ⊢ C sees M:Ts→T=⌊(mxs,mxl0,ins,xt)⌋ in C
 assumes ins:  ins!pc = AStore
 assumes wt:   P,T,mxs,size ins,xt ⊢ ins!pc,pc :: Φ C M
 assumes conf: Φ ⊢ t: (None, h, (stk,loc,C,M,pc)#frs)√
 assumes no-x: (tas, σ) ∈ exec-instr (ins!pc) P t h stk loc C M pc frs
 shows Φ ⊢ t: σ √
proof −
 from ins conf meth
 obtain ST LT where
   heap-ok: hconf h and
   tconf: P,h ⊢ t √t and
   Φ-pc:    Φ C M!pc = Some (ST,LT) and
   stk: P,h ⊢ stk [:≤] ST and loc: P,h ⊢ loc [:≤⊤] LT and
   pc: pc < size ins and
   frame:   conf-f P h (ST,LT) ins (stk,loc,C,M,pc) and
   frames:  conf-fs P h Φ M (size Ts) T frs and
   preh:    preallocated h
   by (auto dest: sees-method-fun)

 from ins wt Φ-pc have lST: length ST > 2 by(auto)

 show ?thesis
 proof(cases hd (tl (tl stk)) = Null)
   case True
    with ins no-x heap-ok tconf Φ-pc stk loc frame frames meth wf-preallocatedD[OF wf, of h Null-
Pointer] preh
   show ?thesis by(fastforce)
 next
   case False
   note stkNN = this
   have STNN: hd (tl (tl ST)) ≠ NT
```

proof
  assume *hd (tl (tl ST)) = NT*
  moreover
  from *frame* **have** *P,h ⊢ stk [:≤] ST* **by** *simp*
  with *lST* **have** *P,h ⊢ hd (tl (tl stk)) :≤ hd (tl (tl ST))*
    by (*cases ST*, *auto*, *case-tac list*, *auto*, *case-tac lista*, *auto*)
  ultimately
  have *hd (tl (tl stk)) = Null* **by** *simp*
  with *stkNN* **show** *False* **by** *contradiction*
qed

with *ins stkNN Φ-pc wt* **obtain** *ST'' Y X ST' LT'* **where**
  ST: *ST = Y # Integer # X⌊⌉ # ST''* **and**
  pc': *pc+1 < size ins* **and**
  Φ': *Φ C M ! (pc+1) = Some (ST', LT')* **and**
  ST': *P ⊢ ST'' [≤] ST'* **and** *LT'*: *P ⊢ LT [≤⊤] LT'* **and**
  suc-pc: *pc+1 < size ins*
  by(*fastforce*)

from *stk ST* **obtain** *ref e idx stk'* **where**
  stk': *stk = e#idx#ref#stk'* **and**
  idx: *P,h ⊢ idx :≤ Integer* **and**
  ref: *P,h ⊢ ref :≤ X⌊⌉* **and**
  e: *P,h ⊢ e :≤ Y* **and**
  ST'': *P,h ⊢ stk' [:≤] ST''*
  by *auto*

from *stkNN stk'* **have** *ref ≠ Null* **by**(*simp*)
with *ref* **obtain** *a Xel n*
  where *a*: *ref = Addr a*
  and *ha*: *typeof-addr h a = ⌊Array-type Xel n⌋*
  and *Xel*: *P ⊢ Xel ≤ X*
  by(*cases ref*)(*fastforce simp add*: *conf-def widen-Array*)+

from *idx* **obtain** *idxI* **where** *idxI*: *idx = Intg idxI*
  by(*auto simp add*: *conf-def*)

show *?thesis*
proof(*cases 0 <=s idxI ∧ sint idxI < int n*)
  case *True*
  hence *si'*: *0 <=s idxI sint idxI < int n* **by** *simp-all*

  from *e* **obtain** *Te* **where** *Te*: *typeof$_h$ e = ⌊Te⌋ P ⊢ Te ≤ Y*
    by(*auto simp add*: *conf-def*)

  show *?thesis*
  proof(*cases P ⊢ Te ≤ Xel*)
    case *True*
    with *Te* **have** *eXel*: *P,h ⊢ e :≤ Xel*
      by(*auto simp add*: *conf-def intro*: *widen-trans*)

    { fix *h'*
      assume *write*: *heap-write h a (ACell (nat (sint idxI))) e h'*
      hence *hext*: *h ◁ h'* **by**(*rule hext-heap-write*)

```
        with preh have preh': preallocated h' by(rule preallocated-hext)

        let ?f = (stk', loc, C, M, pc + 1)

        from si' have nat (sint idxI) < n
          by (simp add: word-sle-eq nat-less-iff)
        with ha have P,h ⊢ a@ACell (nat (sint idxI)) : Xel ..
        with write heap-ok have heap-ok': hconf h' using eXel
          by(rule hconf-heap-write-mono)
        moreover
        from ST stk stk' ST' have P,h ⊢ stk' [:≤] ST' by auto
        with hext have stk'': P,h' ⊢ stk' [:≤] ST'
          by− (rule confs-hext)
        moreover
        from loc LT' have P,h ⊢ loc [:≤⊤] LT' ..
        with hext have P,h' ⊢ loc [:≤⊤] LT' by − (rule confTs-hext)
        moreover
        with frame ST' ST LT' suc-pc wf Xel idxI si' stk''
        have conf-f P h' (ST', LT') ins ?f
          by(clarsimp)
        with frames hext have conf-fs P h' Φ M (size Ts) T frs by− (rule conf-fs-hext)
        moreover from tconf hext have P,h' ⊢ t √t by(rule tconf-hext-mono)
        ultimately have Φ ⊢ t:(None, h', ?f # frs) √ using meth Φ' Φ-pc suc-pc preh'
          by(fastforce) }
      with True si' ins meth stk' a ha no-x idxI idx Te
      show ?thesis
        by(auto split: if-split-asm simp del: correct-state-def intro: widen-trans)
    next
      case False
      with stk' idxI ins no-x heap-ok tconf meth a ha Xel Te Φ-pc frame frames si'
        wf-preallocatedD[OF wf, of h ArrayStore]
      show ?thesis
        by (fastforce split: if-splits list.splits simp: preh
            simp del: Listn.lesub-list-impl-same-size)
    qed
  next
    case False
    with stk' idxI ins no-x heap-ok tconf meth a ha Xel Φ-pc frame frames preh
      wf-preallocatedD[OF wf, of h ArrayIndexOutOfBounds]
    show ?thesis by(fastforce split: if-split-asm)
  qed
 qed
qed

lemma ALength-correct:
 assumes wf:   wf-prog wt P
 assumes meth: P ⊢ C sees M:Ts→T=⌊(mxs,mxl0,ins,xt)⌋ in C
 assumes ins:  ins!pc = ALength
 assumes wt:   P,T,mxs,size ins,xt ⊢ ins!pc,pc :: Φ C M
 assumes conf: Φ ⊢ t: (None, h, (stk,loc,C,M,pc)#frs)√
 assumes no-x: (tas, σ) ∈ exec-instr (ins!pc) P t h stk loc C M pc frs
 shows Φ ⊢ t: σ √
proof −
 from ins conf meth
```

obtain *ST LT* **where**
heap-ok: *hconf h* **and**
tconf: *P*,*h* ⊢ *t* √*t* **and**
Φ-*pc*: Φ *C M*!*pc* = *Some* (*ST*,*LT*) **and**
stk: *P*,*h* ⊢ *stk* [:≤] *ST* **and** *loc*: *P*,*h* ⊢ *loc* [:≤$_\top$] *LT* **and**
pc: *pc* < *size ins* **and**
frame: *conf-f P h* (*ST*,*LT*) *ins* (*stk*,*loc*,*C*,*M*,*pc*) **and**
frames: *conf-fs P h* Φ *M* (*size Ts*) *T frs* **and**
preh: *preallocated h*
by (*auto dest*: *sees-method-fun*)

from *ins wt* Φ-*pc* **have** *lST*: *length ST* > *0* **by**(*auto*)

show *?thesis*
proof(*cases hd stk* = *Null*)
case *True*
with *ins no-x heap-ok tconf* Φ-*pc stk loc frame frames meth wf-preallocatedD*[*OF wf*, *of h Null-Pointer*] *preh*
show *?thesis* **by**(*fastforce*)
next
case *False*
note *stkNN* = *this*
have *STNN*: *hd ST* ≠ *NT*
proof
assume *hd ST* = *NT*
moreover
from *frame* **have** *P*,*h* ⊢ *stk* [:≤] *ST* **by** *simp*
with *lST* **have** *P*,*h* ⊢ *hd stk* :≤ *hd ST*
by (*cases ST*, *auto*)
ultimately
have *hd stk* = *Null* **by** *simp*
with *stkNN* **show** *False* **by** *contradiction*
qed

with *stkNN ins* Φ-*pc wt* **obtain** *ST*′′ *X ST*′ *LT*′ **where**
ST: *ST* = (*X*⌊⌋) # *ST*′′ **and**
pc′: *pc*+*1* < *size ins* **and**
Φ′: Φ *C M* ! (*pc*+*1*) = *Some* (*ST*′, *LT*′) **and**
ST′: *P* ⊢ (*Integer* # *ST*′′) [≤] *ST*′ **and** *LT*′: *P* ⊢ *LT* [≤$_\top$] *LT*′ **and**
suc-pc: *pc*+*1* < *size ins*
by(*fastforce*)

from *stk ST* **obtain** *ref stk*′ **where**
stk′: *stk* = *ref*#*stk*′ **and**
ref: *P*,*h* ⊢ *ref* :≤ *X*⌊⌋ **and**
ST′′: *P*,*h* ⊢ *stk*′ [:≤] *ST*′′
by *auto*

from *stkNN stk*′ **have** *ref* ≠ *Null* **by**(*simp*)
with *ref* **obtain** *a Xel n*
where *a*: *ref* = *Addr a*
and *ha*: *typeof-addr h a* = ⌊*Array-type Xel n*⌋
and *Xel*: *P* ⊢ *Xel* ≤ *X*
by(*cases ref*)(*fastforce simp add*: *conf-def widen-Array*)+

```
    from ins meth stk' a ha no-x have σ':
      σ = (None, h, (Intg (word-of-int (int n)) # stk', loc, C, M, pc + 1) # frs)
      (is σ = (None, h, ?f # frs))
      by(auto)
    moreover
    from ST stk stk' ST' have P,h ⊢ Intg si # stk' [:≤] ST' by(auto)
    with frame ST' ST LT' suc-pc wf
    have conf-f P h (ST', LT') ins ?f
      by(fastforce intro: widen-trans)
    ultimately show ?thesis using meth Φ' heap-ok Φ-pc frames tconf preh by fastforce
  qed
qed


lemma MEnter-correct:
  assumes wf:   wf-prog wt P
  assumes meth: P ⊢ C sees M:Ts→T=⌊(mxs,mxl₀,ins,xt)⌋ in C
  assumes ins:  ins!pc = MEnter
  assumes wt:   P,T,mxs,size ins,xt ⊢ ins!pc,pc :: Φ C M
  assumes conf: Φ ⊢ t: (None, h, (stk,loc,C,M,pc)#frs)√
  assumes no-x: (tas, σ) ∈ exec-instr (ins!pc) P t h stk loc C M pc frs
  shows Φ ⊢ t: σ √
proof −
  from ins conf meth
  obtain ST LT where
    heap-ok: hconf h and
    tconf:   P,h ⊢ t √t and
    Φ-pc:    Φ C M!pc = Some (ST,LT) and
    stk: P,h ⊢ stk [:≤] ST and loc: P,h ⊢ loc [:≤⊤] LT and
    pc: pc < size ins and
    frame:   conf-f P h (ST,LT) ins (stk,loc,C,M,pc) and
    frames:  conf-fs P h Φ M (size Ts) T frs and
    preh:    preallocated h
    by (auto dest: sees-method-fun)

  from ins wt Φ-pc have lST: length ST > 0 by(auto)

  show ?thesis
  proof(cases hd stk = Null)
    case True
    with ins no-x heap-ok tconf Φ-pc stk loc frame frames meth wf-preallocatedD[OF wf, of h Null-
Pointer] preh
    show ?thesis by(fastforce)
  next
    case False
    note stkNN = this
    have STNN: hd ST ≠ NT
    proof
      assume hd ST = NT
      moreover
      from frame have P,h ⊢ stk [:≤] ST by simp
      with lST have P,h ⊢ hd stk :≤ hd ST
        by (cases ST, auto)
```

```
    ultimately
    have hd stk = Null by simp
    with stkNN show False by contradiction
  qed

  with stkNN ins Φ-pc wt obtain ST'' X ST' LT' where
    ST: ST = X # ST'' and
    refT: is-refT X and
    pc': pc+1 < size ins  and
    Φ': Φ C M ! (pc+1) = Some (ST', LT') and
    ST': P ⊢ ST'' [≤] ST' and LT': P ⊢ LT [≤⊤] LT' and
    suc-pc:    pc+1 < size ins
    by(fastforce)

  from stk ST obtain ref stk' where
    stk': stk = ref#stk' and
    ref:  P,h ⊢ ref :≤ X
    by auto

  from stkNN stk' have ref ≠ Null by(simp)
  moreover
  from loc LT' have P,h ⊢ loc [:≤⊤] LT' ..
  moreover
  from ST stk stk' ST'
  have P,h ⊢ stk' [:≤] ST' by(auto)
   ultimately show ?thesis using meth Φ' heap-ok Φ-pc suc-pc frames loc LT' no-x ins stk' ST'
tconf preh
    by(fastforce)
 qed
qed

lemma MExit-correct:
 assumes wf:   wf-prog wt P
 assumes meth: P ⊢ C sees M:Ts→T=⌊(mxs,mxl0,ins,xt)⌋ in C
 assumes ins:  ins!pc = MExit
 assumes wt:   P,T,mxs,size ins,xt ⊢ ins!pc,pc :: Φ C M
 assumes conf: Φ ⊢ t: (None, h, (stk,loc,C,M,pc)#frs)√
 assumes no-x: (tas, σ) ∈ exec-instr (ins!pc) P t h stk loc C M pc frs
 shows Φ ⊢ t: σ √
proof −
 from ins conf meth
 obtain ST LT where
   heap-ok: hconf h and
   tconf:   P,h ⊢ t √t and
   Φ-pc:    Φ C M!pc = Some (ST,LT) and
   stk: P,h ⊢ stk [:≤] ST and loc: P,h ⊢ loc [:≤⊤] LT and
   pc: pc < size ins and
   frame:   conf-f P h (ST,LT) ins (stk,loc,C,M,pc) and
   frames:  conf-fs P h Φ M (size Ts) T frs and
   preh:    preallocated h
   by (auto dest: sees-method-fun)

 from ins wt Φ-pc have lST: length ST > 0 by(auto)
```

```
  show ?thesis
  proof(cases hd stk = Null)
    case True
     with ins no-x heap-ok tconf Φ-pc stk loc frame frames meth wf-preallocatedD[OF wf, of h Null-
Pointer] preh
    show ?thesis by(fastforce)
  next
    case False
    note stkNN = this
    have STNN: hd ST ≠ NT
    proof
      assume hd ST = NT
      moreover
      from frame have P,h ⊢ stk [:≤] ST by simp
      with lST have P,h ⊢ hd stk :≤ hd ST
        by (cases ST, auto)
      ultimately
      have hd stk = Null by simp
      with stkNN show False by contradiction
    qed

    with stkNN ins Φ-pc wt obtain ST'' X ST' LT' where
      ST: ST = X # ST'' and
      refT: is-refT X and
      pc': pc+1 < size ins  and
      Φ': Φ C M ! (pc+1) = Some (ST', LT') and
      ST': P ⊢ ST'' [≤] ST' and LT': P ⊢ LT [≤⊤] LT' and
      suc-pc:     pc+1 < size ins
      by(fastforce)

    from stk ST obtain ref stk' where
      stk': stk = ref#stk' and
      ref:  P,h ⊢ ref :≤ X
      by auto

    from stkNN stk' have ref ≠ Null by(simp)
    moreover
    from loc LT' have P,h ⊢ loc [:≤⊤] LT' ..
    moreover
    from ST stk stk' ST'
    have P,h ⊢ stk' [:≤] ST' by(auto)
    ultimately
    show ?thesis using meth Φ' heap-ok Φ-pc suc-pc frames loc LT' no-x ins stk' ST' tconf frame preh
      wf-preallocatedD[OF wf, of h IllegalMonitorState]
      by(fastforce)
  qed
qed
```

The next theorem collects the results of the sections above, i.e. exception handling and the execution step for each instruction. It states type safety for single step execution: in welltyped programs, a conforming state is transformed into another conforming state when one instruction is executed.

```
theorem instr-correct:
⟦ wf-jvm-prog_Φ P;
```

$P \vdash C$ *sees* M:$Ts \to T = \lfloor (mxs, mxl_0, ins, xt) \rfloor$ *in* C;
$(tas, \sigma') \in$ *exec* $P\ t$ (*None*, h, (stk,loc,C,M,pc)#frs);
$\Phi \vdash t$: (*None*, h, (stk,loc,C,M,pc)#frs)$\surd$ ⟧
$\Longrightarrow \Phi \vdash t$: $\sigma'\surd$
apply (*subgoal-tac* P,T,mxs,*size* ins,$xt \vdash ins!pc, pc :: \Phi\ C\ M$)
prefer *2*
apply (*erule wt-jvm-prog-impl-wt-instr*, *assumption*)
apply *clarsimp*
apply (*drule* (*1*) *sees-method-fun*)
apply *simp*
apply(*unfold exec.simps Let-def set-map*)
apply (*frule wt-jvm-progD*, *erule exE*)
apply (*cases ins* ! *pc*)
apply (*rule Load-correct*, *assumption*+, *fastforce*)
apply (*rule Store-correct*, *assumption*+, *fastforce*)
apply (*rule Push-correct*, *assumption*+, *fastforce*)
apply (*rule New-correct*, *assumption*+, *fastforce*)
apply (*rule NewArray-correct*, *assumption*+, *fastforce*)
apply (*rule ALoad-correct*, *assumption*+, *fastforce*)
apply (*rule AStore-correct*, *assumption*+, *fastforce*)
apply (*rule ALength-correct*, *assumption*+, *fastforce*)
apply (*rule Getfield-correct*, *assumption*+, *fastforce*)
apply (*rule Putfield-correct*, *assumption*+, *fastforce*)
apply (*rule CAS-correct*, *assumption*+, *fastforce*)
apply (*rule Checkcast-correct*, *assumption*+, *fastforce*)
apply (*rule Instanceof-correct*, *assumption*+, *fastforce*)
apply (*rule Invoke-correct*, *assumption*+, *fastforce*)
apply (*rule Return-correct*, *assumption*+, *fastforce simp add*: *split-beta*)
apply (*rule Pop-correct*, *assumption*+, *fastforce*)
apply (*rule Dup-correct*, *assumption*+, *fastforce*)
apply (*rule Swap-correct*, *assumption*+, *fastforce*)
apply (*rule BinOp-correct*, *assumption*+, *fastforce*)
apply (*rule Goto-correct*, *assumption*+, *fastforce*)
apply (*rule IfFalse-correct*, *assumption*+, *fastforce*)
apply (*rule Throw-correct*, *assumption*+, *fastforce*)
apply (*rule MEnter-correct*, *assumption*+, *fastforce*)
apply (*rule MExit-correct*, *assumption*+, *fastforce*)
done

declare *defs1* [*simp del*]

end

6.5.4 Main

lemma (**in** *JVM-conf-read*) *BV-correct-1* [*rule-format*]:
$\bigwedge \sigma$. ⟦ *wf-jvm-prog*$_\Phi$ P; $\Phi \vdash t$: $\sigma\surd$ ⟧ $\Longrightarrow P, t \vdash \sigma\ -tas-jvm\to\ \sigma' \longrightarrow \Phi \vdash t$: $\sigma'\surd$
apply (*simp only*: *split-tupled-all exec-1-iff*)
apply (*rename-tac xp h frs*)
apply (*case-tac xp*)
apply (*case-tac frs*)
apply *simp*
apply (*simp only*: *split-tupled-all*)
apply *hypsubst*

```
 apply (frule correct-state-impl-Some-method)
 apply clarify
 apply (rule instr-correct)
 apply assumption+
apply clarify
apply(case-tac frs)
 apply simp
apply(clarsimp simp only: exec.simps set-simps)
apply(erule (1) exception-step-conform)
done

theorem (in JVM-progress) progress:
  assumes wt: wf-jvm-prog_Φ P
  and cs: Φ ⊢ t: (xcp, h, f # frs)√
  shows ∃ ta σ'. P,t ⊢ (xcp, h, f # frs) −ta−jvm→ σ'
proof −
  obtain stk loc C M pc where f: f = (stk, loc, C, M, pc) by(cases f)
  with cs obtain Ts T mxs mxl_0 is xt ST LT
    where hconf: hconf h
    and sees: P ⊢ C sees M: Ts→T = ⌊(mxs, mxl_0, is, xt)⌋ in C
    and Φ-pc: Φ C M ! pc = ⌊(ST, LT)⌋
    and ST: P,h ⊢ stk [:≤] ST
    and LT: P,h ⊢ loc [:≤_⊤] LT
    and pc: pc < length is
    by(auto simp add: defs1)
  show ?thesis
  proof(cases xcp)
    case Some thus ?thesis
      unfolding f exec-1-iff by auto
  next
    case [simp]: None
    note [simp del] = split-paired-Ex
    note [simp] = defs1 list-all2-Cons2


    from wt obtain wf-md where wf: wf-prog wf-md P by(auto dest: wt-jvm-progD)
    from wt sees pc have wt: P,T,mxs,size is,xt ⊢ is!pc,pc :: Φ C M
      by(rule wt-jvm-prog-impl-wt-instr)


    have ∃ ta σ'. (ta, σ') ∈ exec-instr (is ! pc) P t h stk loc C M pc frs
    proof(cases is ! pc)
      case [simp]: ALoad
      with wt Φ-pc have lST: length ST > 1 by(auto)
      show ?thesis
      proof(cases hd (tl stk) = Null)
        case True thus ?thesis by simp
      next
        case False
        have STNN: hd (tl ST) ≠ NT
        proof
          assume hd (tl ST) = NT
          moreover
          from ST lST have P,h ⊢ hd (tl stk) :≤ hd (tl ST)
```

```
      by (cases ST)(auto, case-tac list, auto)
    ultimately have hd (tl stk) = Null by simp
    with False show False by contradiction
  qed

  with False Φ-pc wt obtain ST'' X where ST = Integer # X⌊⌉ # ST'' by auto
  with ST obtain ref idx stk' where stk': stk = idx#ref#stk' and idx: P,h ⊢ idx :≤ Integer
    and ref:  P,h ⊢ ref :≤ X⌊⌉ by(auto)

  from False stk' have ref ≠ Null by(simp)
  with ref obtain a Xel n where a: ref = Addr a
    and ha: typeof-addr h a = ⌊Array-type Xel n⌋
    and Xel: P ⊢ Xel ≤ X
    by(cases ref)(fastforce simp add: conf-def widen-Array)+

  from idx obtain idxI where idxI: idx = Intg idxI
    by(auto simp add: conf-def)
  show ?thesis
  proof(cases 0 <=s idxI ∧ sint idxI < int n)
    case True
    hence si': 0 <=s idxI sint idxI < int n by auto
    hence nat (sint idxI) < n
      by (simp add: word-sle-eq nat-less-iff)
    with ha have al: P,h ⊢ a@ACell (nat (sint idxI)) : Xel ..
    from heap-read-total[OF hconf this] True False ha stk' idxI a
    show ?thesis by auto
  next
    case False with ha stk' idxI a show ?thesis by auto
  qed
 qed
next
 case [simp]: AStore
 from wt Φ-pc have lST: length ST > 2 by(auto)

 show ?thesis
 proof(cases hd (tl (tl stk)) = Null)
   case True thus ?thesis by(fastforce)
 next
   case False
   note stkNN = this
   have STNN: hd (tl (tl ST)) ≠ NT
   proof
     assume hd (tl (tl ST)) = NT
     moreover
     from ST lST have P,h ⊢ hd (tl (tl stk)) :≤ hd (tl (tl ST))
       by (cases ST, auto, case-tac list, auto, case-tac lista, auto)
     ultimately have hd (tl (tl stk)) = Null by simp
     with stkNN show False by contradiction
   qed

   with stkNN Φ-pc wt obtain ST'' Y X
     where ST = Y # Integer # X⌊⌉ # ST'' by(fastforce)

   with ST obtain ref e idx stk' where stk': stk = e#idx#ref#stk'
```

```
      and idx: P,h ⊢ idx :≤ Integer and ref:  P,h ⊢ ref :≤ X⌊⌉
      and e: P,h ⊢ e :≤ Y by auto

    from stkNN stk' have ref ≠ Null by(simp)
    with ref obtain a Xel n where a: ref = Addr a
      and ha: typeof-addr h a = ⌊Array-type Xel n⌋
      and Xel: P ⊢ Xel ≤ X
      by(cases ref)(fastforce simp add: conf-def widen-Array)+

    from idx obtain idxI where idxI: idx = Intg idxI
      by(auto simp add: conf-def)

    show ?thesis
    proof(cases 0 <=s idxI ∧ sint idxI < int n)
      case True
      hence si': 0 <=s idxI sint idxI < int n by simp-all
      hence nat (sint idxI) < n
        by (simp add: word-sle-eq nat-less-iff)
      with ha have adal: P,h ⊢ a@ACell (nat (sint idxI)) : Xel ..

      show ?thesis
      proof(cases P ⊢ the (typeof_h e) ≤ Xel)
        case False
        with ha stk' idxI a show ?thesis by auto
      next
        case True
        hence P,h ⊢ e :≤ Xel using e by(auto simp add: conf-def)
        from heap-write-total[OF hconf adal this] ha stk' idxI a show ?thesis by auto
      qed
    next
      case False with ha stk' idxI a show ?thesis by auto
    qed
  qed
next
  case [simp]: (Getfield F D)

  from Φ-pc wt obtain oT ST'' vT fm where oT: P ⊢ oT ≤ Class D
    and ST = oT # ST'' and F: P ⊢ D sees F:vT (fm) in D
    by fastforce

  with ST obtain ref stk' where stk': stk = ref#stk'
    and ref:  P,h ⊢ ref :≤ oT by auto

  show ?thesis
  proof(cases ref = Null)
    case True thus ?thesis using stk' by auto
  next
    case False
    from ref oT have P,h ⊢ ref :≤ Class D ..
    with False obtain a U' D' where
      a: ref = Addr a and h: typeof-addr h a = Some U'
      and U': D' = class-type-of U' and D': P ⊢ D' ≼* D
      by (blast dest: non-npD2)
```

```
    from D' F have has-field: P ⊢ D' has F:vT (fm) in D
      by (blast intro: has-field-mono has-visible-field)
    with h have P,h ⊢ a@CField D F : vT unfolding U' ..
    from heap-read-total[OF hconf this]
    show ?thesis using stk' a by auto
  qed
next
  case [simp]: (Putfield F D)

  from Φ-pc wt obtain vT vT' oT ST'' fm where ST = vT # oT # ST''
    and field: P ⊢ D sees F:vT' (fm) in D
    and oT: P ⊢ oT ≤ Class D
    and vT': P ⊢ vT ≤ vT' by fastforce
  with ST obtain v ref stk' where stk': stk = v#ref#stk'
    and ref:  P,h ⊢ ref :≤ oT
    and v: P,h ⊢ v :≤ vT by auto

  show ?thesis
  proof(cases ref = Null)
    case True with stk' show ?thesis by auto
  next
    case False
    from ref oT have P,h ⊢ ref :≤ Class D ..
    with False obtain a U' D' where
      a: ref = Addr a and h: typeof-addr h a = Some U' and
      U': D' = class-type-of U' and D': P ⊢ D' ≼* D
      by (blast dest: non-npD2)

    from field D' have has-field: P ⊢ D' has F:vT' (fm) in D
      by (blast intro: has-field-mono has-visible-field)
    with h have al: P,h ⊢ a@CField D F : vT' unfolding U' ..
    from v vT' have P,h ⊢ v :≤ vT' by auto
    from heap-write-total[OF hconf al this] v a stk' h show ?thesis by auto
  qed
next
  case [simp]: (CAS F D)
  from Φ-pc wt obtain T' T1 T2 T3 ST'' fm where ST = T3 # T2 # T1 # ST''
    and field: P ⊢ D sees F:T' (fm) in D
    and oT: P ⊢ T1 ≤ Class D
    and vT': P ⊢ T2 ≤ T' P ⊢ T3 ≤ T' by fastforce
  with ST obtain v v' v'' stk' where stk': stk = v''#v'#v#stk'
    and v:  P,h ⊢ v :≤ T1
    and v': P,h ⊢ v' :≤ T2
    and v'': P,h ⊢ v'' :≤ T3 by auto
  show ?thesis
  proof(cases v= Null)
    case True with stk' show ?thesis by auto
  next
    case False
    from v oT have P,h ⊢ v :≤ Class D ..
    with False obtain a U' D' where
      a: v = Addr a and h: typeof-addr h a = Some U' and
      U': D' = class-type-of U' and D': P ⊢ D' ≼* D
      by (blast dest: non-npD2)
```

```
      from field D' have has-field: P ⊢ D' has F:T' (fm) in D
        by (blast intro: has-field-mono has-visible-field)
      with h have al: P,h ⊢ a@CField D F : T' unfolding U' ..
      from v' vT' have P,h ⊢ v' :≤ T' by auto
      from heap-read-total[OF hconf al] obtain v''' where v''': heap-read h a (CField D F) v''' by
blast
      show ?thesis
      proof(cases v''' = v')
        case True
        from v'' vT' have P,h ⊢ v'' :≤ T' by auto
        from heap-write-total[OF hconf al this] v a stk' h v''' True show ?thesis by auto
      next
        case False
        from v''' v a stk' h False show ?thesis by auto
      qed
    qed
  next
    case [simp]: (Invoke M' n)

    from wt Φ-pc have n: n < size ST by simp

    show ?thesis
    proof(cases stk!n = Null)
      case True thus ?thesis by simp
    next
      case False
      note Null = this
      have NT: ST!n ≠ NT
      proof
        assume ST!n = NT
        moreover from ST n have P,h ⊢ stk!n :≤ ST!n by (simp add: list-all2-conv-all-nth)
        ultimately have stk!n = Null by simp
        with Null show False by contradiction
      qed

      from NT wt Φ-pc obtain D D' Ts T m
        where D: class-type-of' (ST!n) = Some D
        and m-D: P ⊢ D sees M': Ts→T = m in D'
        and Ts: P ⊢ rev (take n ST) [≤] Ts
        by auto

      from n ST D have P,h ⊢ stk!n :≤ ST!n
        by (auto simp add: list-all2-conv-all-nth)

      from ‹P,h ⊢ stk!n :≤ ST!n› Null D
      obtain a T' where
        Addr:  stk!n = Addr a and
        obj:   typeof-addr h a = Some T' and
        T'subSTn: P ⊢ ty-of-htype T' ≤ ST ! n
        by(cases stk ! n)(auto simp add: conf-def widen-Class)

      from D T'subSTn obtain C' where
        C': class-type-of' (ty-of-htype T') = ⌊C'⌋ and C'subD: P ⊢ C' ≼* D
```

```
        by(rule widen-is-class-type-of) simp

      from Call-lemma[OF m-D C'subD wf]
      obtain D' Ts' T' m'
        where Call': P ⊢ C' sees M': Ts'→T' = m' in D' P ⊢ Ts [≤] Ts'
          P ⊢ T' ≤ T P ⊢ C' ≼* D' is-type P T' ∀ T∈set Ts'. is-type P T
        by blast

      show ?thesis
      proof(cases m')
        case Some with Call' C' obj Addr C' C'subD show ?thesis by(auto)
      next
        case [simp]: None
        from ST have P,h ⊢ take n stk [:≤] take n ST by(rule list-all2-takeI)
        then obtain Us where map typeof_h (take n stk) = map Some Us P ⊢ Us [≤] take n ST
          by(auto simp add: confs-conv-map)
         hence Us: map typeof_h (rev (take n stk)) = map Some (rev Us) P ⊢ rev Us [≤] rev (take n
ST)
          by− (simp only: rev-map[symmetric], simp)
        with Ts ‹P ⊢ Ts [≤] Ts'› have P ⊢ rev Us [≤] Ts' by(blast intro: widens-trans)
        with obj Us Call' C' have P,h ⊢ a•M'(rev (take n stk)) : T'
          by(auto intro!: external-WT'.intros)
        from external-call-progress[OF wf this hconf, of t] obj Addr Call' C'
        show ?thesis by(auto dest!: red-external-imp-red-external-aggr)
      qed
    qed
  qed(auto 4 4 simp add: split-beta split: if-split-asm)
  thus ?thesis using sees None
    unfolding f exec-1-iff by(simp del: split-paired-Ex)
 qed
qed

lemma (in JVM-heap-conf) BV-correct-initial:
 shows ⟦ wf-jvm-prog_Φ P; start-heap-ok; P ⊢ C sees M:Ts→T = ⌊m⌋ in D; P,start-heap ⊢ vs [:≤]
Ts ⟧
 ⟹ Φ ⊢ start-tid:JVM-start-state' P C M vs √
 apply (cases m)
 apply (unfold JVM-start-state'-def)
 apply (unfold correct-state-def)
 apply (clarsimp)
 apply (frule wt-jvm-progD)
 apply (erule exE)
 apply (frule wf-prog-wf-syscls)
 apply (rule conjI)
  apply(erule (1) tconf-start-heap-start-tid)
 apply(rule conjI)
  apply (simp add: wf-jvm-prog-phi-def hconf-start-heap)
 apply(frule sees-method-idemp)
 apply (frule wt-jvm-prog-impl-wt-start, assumption+)
 apply (unfold conf-f-def wt-start-def)
 apply(auto simp add: sup-state-opt-any-Some)
  apply(erule preallocated-start-heap)
 apply(rule exI conjI|assumption)+
 apply(auto simp add: list-all2-append1)
```

apply(*auto dest*: *list-all2-lengthD intro*!: *exI*)
done

end

6.6 Welltyped Programs produce no Type Errors

theory *BVNoTypeError*
imports
../JVM/JVMDefensive
BVSpecTypeSafe
begin

lemma *wt-jvm-prog-states*:
⟦ *wf-jvm-prog*$_\Phi$ *P*; *P* ⊢ *C sees M*: *Ts*→*T* = ⌊(*mxs*, *mxl*, *ins*, *et*)⌋ *in C*;
Φ *C M* ! *pc* = τ; *pc* < *size ins* ⟧
⟹ *OK* τ ∈ *states P mxs* (*1*+*size Ts*+*mxl*)
context *JVM-heap-conf-base′* **begin**

declare *is-IntgI* [*simp*, *intro*]
declare *is-BoolI* [*simp*, *intro*]
declare *is-RefI* [*simp*]

The main theorem: welltyped programs do not produce type errors if they are started in a conformant state.

theorem *no-type-error*:
assumes *welltyped*: *wf-jvm-prog*$_\Phi$ *P* **and** *conforms*: Φ ⊢ *t*:σ √
shows *exec-d P t* σ ≠ *TypeError*

6.7 Progress result for both of the multithreaded JVMs

theory *BVProgressThreaded*
imports
../Framework/FWProgress
../Framework/FWLTS
BVNoTypeError
../JVM/JVMThreaded
begin

lemma (**in** *JVM-heap-conf-base′*) *mexec-eq-mexecd*:
⟦ *wf-jvm-prog*$_\Phi$ *P*; Φ ⊢ *t*: (*xcp*, *h*, *frs*) √ ⟧ ⟹ *mexec P t* ((*xcp*, *frs*), *h*) = *mexecd P t* ((*xcp*, *frs*), *h*)
apply(*auto intro*!: *ext*)
apply(*unfold exec-1-iff*)
apply(*drule no-type-error*)
apply(*assumption*)
apply(*clarify*)
apply(*rule exec-1-d-NormalI*)
apply(*assumption*)
apply(*simp add*: *exec-d-def split*: *if-split-asm*)
apply(*erule jvmd-NormalE*, *auto*)
done

context *JVM-heap-conf-base* **begin**

abbreviation
correct-state-ts :: $ty_P \Rightarrow$ ('*addr*,'*thread-id*,'*addr jvm-thread-state*) *thread-info* $\Rightarrow$ '*heap* $\Rightarrow$ *bool*
where
correct-state-ts $\Phi \equiv$ *ts-ok* (λt (*xcp*, *frstls*) *h*. $\Phi \vdash t$: (*xcp*, *h*, *frstls*) $\surd$)

lemma *correct-state-ts-thread-conf*:
correct-state-ts Φ (*thr s*) (*shr s*) $\Longrightarrow$ *thread-conf P* (*thr s*) (*shr s*)
by(*erule ts-ok-mono*)(*auto simp add*: *correct-state-def*)

lemma *invoke-new-thread*:
assumes *wf-jvm-prog*$_\Phi$ *P*
and *P* $\vdash$ *C sees M*:*Ts*$\rightarrow$*T*=$\lfloor$(*mxs*,*mxl0*,*ins*,*xt*)$\rfloor$ *in C*
and *ins* ! *pc* = *Invoke Type.start 0*
and *P*,*T*,*mxs*,*size ins*,*xt* $\vdash$ *ins*!*pc*,*pc* :: Φ *C M*
and $\Phi \vdash t$: (*None*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs*) $\surd$
and *typeof-addr h* (*thread-id2addr a*) = $\lfloor$*Class-type D*$\rfloor$
and *P* $\vdash$ *D* $\preceq^*$ *Thread*
and *P* $\vdash$ *D sees run*:[]$\rightarrow$*Void*=$\lfloor$(*mxs'*, *mxl0'*, *ins'*,*xt'*)$\rfloor$ *in D'*
shows $\Phi \vdash$ *a*: (*None*, *h*, [([], *Addr* (*thread-id2addr a*) # *replicate mxl0'* *undefined-value*, *D'*, *run*, *0*)]) $\surd$
proof −
from ‹$\Phi \vdash t$: (*None*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs*) $\surd$›
have *hconf h* **and** *preallocated h* **by**(*simp-all add*: *correct-state-def*)
moreover
from ‹*P* $\vdash$ *D sees run*:[]$\rightarrow$*Void*=$\lfloor$(*mxs'*, *mxl0'*, *ins'*,*xt'*)$\rfloor$ *in D'*›
have *P* $\vdash$ *D' sees run*:[]$\rightarrow$*Void*=$\lfloor$(*mxs'*, *mxl0'*, *ins'*,*xt'*)$\rfloor$ *in D'*
by(*rule sees-method-idemp*)
with ‹*wf-jvm-prog*$_\Phi$ *P*›
have *wt-start P D'* [] *mxl0'* (Φ *D' run*) **and** *ins'* $\neq$ []
by(*auto dest*: *wt-jvm-prog-impl-wt-start*)
then obtain *LT'* **where** *LT'*: Φ *D' run* ! *0* = *Some* ([], *LT'*)
by (*clarsimp simp add*: *wt-start-def defs1 sup-state-opt-any-Some*)
moreover
have *conf-f P h* ([], *LT'*) *ins'* ([], *Addr* (*thread-id2addr a*) # *replicate mxl0'* *undefined-value*, *D'*, *run*, *0*)
proof −
let *?LT* = *OK* (*Class D'*) # (*replicate mxl0' Err*)
have *P*,*h* $\vdash$ *replicate mxl0'* *undefined-value* [:$\leq_\top$] *replicate mxl0' Err* **by** *simp*
also from ‹*P* $\vdash$ *D sees run*:[]$\rightarrow$*Void*=$\lfloor$(*mxs'*, *mxl0'*, *ins'*,*xt'*)$\rfloor$ *in D'*›
have *P* $\vdash$ *D* $\preceq^*$ *D'* **by**(*rule sees-method-decl-above*)
with ‹*typeof-addr h* (*thread-id2addr a*) = $\lfloor$*Class-type D*$\rfloor$›
have *P*,*h* $\vdash$ *Addr* (*thread-id2addr a*) :$\leq$ *Class D'*
by(*simp add*: *conf-def*)
ultimately have *P*,*h* $\vdash$ *Addr* (*thread-id2addr a*) # *replicate mxl0'* *undefined-value* [:$\leq_\top$] *?LT*
by(*simp*)
also from ‹*wt-start P D'* [] *mxl0'* (Φ *D' run*)› *LT'*
have *P* $\vdash$ ... [$\leq_\top$] *LT'* **by**(*simp add*: *wt-start-def*)
finally have *P*,*h* $\vdash$ *Addr* (*thread-id2addr a*) # *replicate mxl0'* *undefined-value* [:$\leq_\top$] *LT'* .
with ‹*ins'* $\neq$ []› **show** *?thesis* **by**(*simp add*: *conf-f-def*)
qed
moreover from ‹*typeof-addr h* (*thread-id2addr a*) = $\lfloor$*Class-type D*$\rfloor$› ‹*P* $\vdash$ *D* $\preceq^*$ *Thread*›

have $P,h \vdash a\ \surd t$ **by**(*rule tconfI*)
ultimately show *?thesis* **using** ‹$P \vdash D'$ *sees run*:[]→*Void*=⌊(*mxs'*, *mxl0'*, *ins'*,*xt'*)⌋ *in* D'›
by(*fastforce simp add*: *correct-state-def*)
qed

lemma *exec-new-threadE*:
assumes *wf-jvm-prog*$_\Phi$ P
and $P,t \vdash$ *Normal* σ $-ta-jvmd\rightarrow$ *Normal* σ'
and $\Phi \vdash t$: $\sigma\ \surd$
and $\{\!|ta|\!\}_t \neq []$
obtains *h frs a stk loc C M pc Ts T mxs mxl0 ins xt M' n Ta ta' va Us Us' U m' D'*
where σ = (*None*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs*)
and (*ta*, σ') ∈ *exec P t* (*None*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs*)
and $P \vdash C$ *sees M*: *Ts*→*T* = ⌊(*mxs*, *mxl0*, *ins*, *xt*)⌋ *in C*
and *stk* ! *n* = *Addr a*
and *ins* ! *pc* = *Invoke M' n*
and *n* < *length stk*
and *typeof-addr h a* = ⌊*Ta*⌋
and *is-native P Ta M'*
and *ta* = *extTA2JVM P ta'*
and σ' = *extRet2JVM n m' stk loc C M pc frs va*
and (*ta'*, *va*, *m'*) ∈ *red-external-aggr P t a M'* (*rev* (*take n stk*)) *h*
and *map typeof*$_h$ (*rev* (*take n stk*)) = *map Some Us*
and $P \vdash$ *class-type-of Ta sees M'*:*Us'*→*U* = *Native in D'*
and *D'*·*M'*(*Us'*) :: *U*
and $P \vdash$ *Us* [≤] *Us'*
proof −
from ‹$P,t \vdash$ *Normal* σ $-ta-jvmd\rightarrow$ *Normal* σ'› **obtain** *h f Frs xcp*
where *check*: *check P* σ
and *exec*: (*ta*, σ') ∈ *exec P t* σ
and [*simp*]: σ = (*xcp*, *h*, *f* # *Frs*)
by(*rule jvmd-NormalE*)
obtain *stk loc C M pc* **where** [*simp*]: *f* = (*stk*, *loc*, *C*, *M*, *pc*)
by(*cases f*, *blast*)
from ‹$\{\!|ta|\!\}_t \neq []$› *exec* **have** [*simp*]: *xcp* = *None* **by**(*cases xcp*) *auto*
from ‹$\Phi \vdash t$: $\sigma\ \surd$›
obtain *Ts T mxs mxl0 ins xt ST LT*
where *hconf h preallocated h*
and *sees*: $P \vdash C$ *sees M*: *Ts*→*T* = ⌊(*mxs*, *mxl0*, *ins*, *xt*)⌋ *in C*
and Φ *C M* ! *pc* = ⌊(*ST*, *LT*)⌋
and *conf-f P h* (*ST*, *LT*) *ins* (*stk*, *loc*, *C*, *M*, *pc*)
and *conf-fs P h* Φ *M* (*length Ts*) *T Frs*
by(*fastforce simp add*: *correct-state-def*)
from *check* ‹Φ *C M* ! *pc* = ⌊(*ST*, *LT*)⌋› *sees*
have *checkins*: *check-instr* (*ins* ! *pc*) *P h stk loc C M pc Frs*
by(*clarsimp simp add*: *check-def*)
from *sees* ‹$\{\!|ta|\!\}_t \neq []$› *exec* **obtain** *M' n* **where** [*simp*]: *ins* ! *pc* = *Invoke M' n*
by(*cases ins* ! *pc*, *auto split*: *if-split-asm simp add*: *split-beta ta-upd-simps*)
from ‹*wf-jvm-prog*$_\Phi$ *P*› **obtain** *wfmd* **where** *wfp*: *wf-prog wfmd P* **by**(*auto dest*: *wt-jvm-progD*)

from *checkins* **have** *n* < *length stk is-Ref* (*stk* ! *n*) **by** *auto*
moreover from *exec sees* ‹$\{\!|ta|\!\}_t \neq []$› **have** *stk* ! *n* ≠ *Null* **by** *auto*
with ‹*is-Ref* (*stk* ! *n*)› **obtain** *a* **where** *stk* ! *n* = *Addr a*
by(*auto simp add*: *is-Ref-def elim*: *is-AddrE*)

moreover with *checkins* **obtain** *Ta* **where** *Ta*: *typeof-addr h a* = ⌊*Ta*⌋ **by**(*fastforce*)
moreover with *checkins exec sees* ‹*n* < *length stk*› ‹{|*ta*|}$_t$ ≠ []› ‹*stk* ! *n* = *Addr a*›
obtain *Us Us′ U D′* **where** *map typeof*$_h$ (*rev* (*take n stk*)) = *map Some Us*
and *P* ⊢ *class-type-of Ta sees M′*:*Us′*→*U* = *Native in D′* **and** *D′*•*M′*(*Us′*) :: *U*
and *P* ⊢ *Us* [≤] *Us′*
by(*auto simp add*: *confs-conv-map min-def split-beta has-method-def external-WT′-iff split*: *if-split-asm*)
moreover with ‹*typeof-addr h a* = ⌊*Ta*⌋› ‹*n* < *length stk*› *exec sees* ‹*stk* ! *n* = *Addr a*›
obtain *ta′ va h′* **where** *ta* = *extTA2JVM P ta′ σ′* = *extRet2JVM n h′ stk loc C M pc Frs va*
(*ta′*, *va*, *h′*) ∈ *red-external-aggr P t a M′* (*rev* (*take n stk*)) *h*
by(*fastforce simp add*: *min-def*)
ultimately show *thesis* **using** *exec sees*
by−(*rule that*, *auto intro*!: *is-native.intros*)
qed

end

context *JVM-conf-read* **begin**

lemma *correct-state-new-thread*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *red*: *P*,*t* ⊢ *Normal σ* −*ta*−*jvmd*→ *Normal σ′*
and *cs*: Φ ⊢ *t*: *σ* √
and *nt*: *NewThread t″* (*xcp*, *frs*) *h″* ∈ *set* {|*ta*|}$_t$
shows Φ ⊢ *t″*: (*xcp*, *h″*, *frs*) √
proof −
from *wf* **obtain** *wt* **where** *wfp*: *wf-prog wt P* **by**(*blast dest*: *wt-jvm-progD*)
from *nt* **have** {|*ta*|}$_t$ ≠ [] **by** *auto*
with *wf red cs*
obtain *h Frs a stk loc C M pc Ts T mxs mxl0 ins xt M′ n Ta ta′ va h′ Us Us′ U D′*
where [*simp*]: *σ* = (*None*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *Frs*)
and *exec*: (*ta*, *σ′*) ∈ *exec P t* (*None*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *Frs*)
and *sees*: *P* ⊢ *C sees M*: *Ts*→*T* = ⌊(*mxs*, *mxl0*, *ins*, *xt*)⌋ *in C*
and [*simp*]: *stk* ! *n* = *Addr a*
and [*simp*]: *ins* ! *pc* = *Invoke M′ n*
and *n*: *n* < *length stk*
and *Ta*: *typeof-addr h a* = ⌊*Ta*⌋
and *iec*: *is-native P Ta M′*
and *ta*: *ta* = *extTA2JVM P ta′*
and *σ′*: *σ′* = *extRet2JVM n h′ stk loc C M pc Frs va*
and *rel*: (*ta′*, *va*, *h′*) ∈ *red-external-aggr P t a M′* (*rev* (*take n stk*)) *h*
and *Us*: *map typeof*$_h$ (*rev* (*take n stk*)) = *map Some Us*
and *wtext*: *P* ⊢ *class-type-of Ta sees M′*:*Us′*→*U* = *Native in D′ D′*•*M′*(*Us′*) :: *U*
and *sub*: *P* ⊢ *Us* [≤] *Us′*
by(*rule exec-new-threadE*)
from *cs* **have** *hconf*: *hconf h* **and** *preh*: *preallocated h*
and *tconf*: *P*,*h* ⊢ *t* √*t* **by**(*auto simp add*: *correct-state-def*)
from *Ta Us wtext sub* **have** *wtext′*: *P*,*h* ⊢ *a*•*M′*(*rev* (*take n stk*)) : *U*
by(*auto intro*!: *external-WT′.intros*)
from *rel* **have** *red*: *P*,*t* ⊢ ⟨*a*•*M′*(*rev* (*take n stk*)), *h*⟩ −*ta′*→*ext* ⟨*va*, *h′*⟩
by(*unfold WT-red-external-list-conv*[*OF wfp wtext′ tconf*])
from *ta nt* **obtain** *D M″ a′* **where** *nt′*: *NewThread t″* (*D*, *M″*, *a′*) *h″* ∈ *set* {|*ta′*|}$_t$
(*xcp*, *frs*) = *extNTA2JVM P* (*D*, *M″*, *a′*) **by** *auto*
with *red* **have** [*simp*]: *h″* = *h′* **by**−(*rule red-ext-new-thread-heap*)
from *red-external-new-thread-sub-thread*[*OF red nt′*(*1*)]

have $h't''$: *typeof-addr* h' $a' = \lfloor$*Class-type* $D\rfloor$ $P \vdash D \preceq^*$ *Thread* **and** [*simp*]: $M'' =$ *run* **by** *auto*
from *red-external-new-thread-exists-thread-object*[*OF red* $nt'(1)$]
have *tconf'*: $P,h' \vdash t'' \surd t$ **by**(*auto intro*: *tconfI*)
from *sub-Thread-sees-run*[*OF wfp* ‹$P \vdash D \preceq^*$ *Thread*›] **obtain** *mxs' mxl0' ins' xt' D'*
where *seesrun*: $P \vdash D$ *sees run*: $[]\rightarrow$*Void* = $\lfloor$(*mxs'*, *mxl0'*, *ins'*, *xt'*)$\rfloor$ *in* D' **by** *auto*
with *nt' ta nt* **have** *xcp* = *None frs* = [([],*Addr a'* # *replicate mxl0' undefined-value*,*D'*,*run*,*0*)]
by(*auto simp add*: *extNTA2JVM-def split-beta*)
moreover
have $\Phi \vdash t''$: (*None*, *h'*, [([], *Addr a'* # *replicate mxl0' undefined-value*, *D'*, *run*, *0*)]) $\surd$
proof −
from *red wtext'* ‹*hconf h*› **have** *hconf h'*
by(*rule external-call-hconf*)
moreover from *red* **have** $h \trianglelefteq h'$ **by**(*rule red-external-hext*)
with *preh* **have** *preallocated h'* **by**(*rule preallocated-hext*)
moreover from *seesrun*
have *seesrun'*: $P \vdash D'$ *sees run*: $[]\rightarrow$*Void* = $\lfloor$(*mxs'*, *mxl0'*, *ins'*, *xt'*)$\rfloor$ *in* D'
by(*rule sees-method-idemp*)
moreover with ‹*wf-jvm-prog*$_\Phi$ *P*›
obtain *wt-start P D'* [] *mxl0'* (Φ *D' run*) *ins'* ≠ []
by (*auto dest*: *wt-jvm-prog-impl-wt-start*)
then obtain *LT'* **where** Φ *D' run* ! *0* = *Some* ([], *LT'*)
by (*clarsimp simp add*: *wt-start-def defs1 sup-state-opt-any-Some*)
moreover
have *conf-f P h'* ([], *LT'*) *ins'* ([], *Addr a'* # *replicate mxl0' undefined-value*, *D'*, *run*, *0*)
proof −
let *?LT* = *OK* (*Class D'*) # (*replicate mxl0' Err*)
from *seesrun* **have** $P \vdash D \preceq^* D'$ **by**(*rule sees-method-decl-above*)
hence $P,h' \vdash$ *Addr a'* # *replicate mxl0' undefined-value* [:≤$_\top$] *?LT*
using $h't''$ **by**(*simp add*: *conf-def*)
also from ‹*wt-start P D'* [] *mxl0'* (Φ *D' run*)› ‹Φ *D' run* ! *0* = *Some* ([], *LT'*)›
have $P \vdash$ *?LT* [≤$_\top$] *LT'* **by**(*simp add*: *wt-start-def*)
finally have $P,h' \vdash$ *Addr a'* # *replicate mxl0' undefined-value* [:≤$_\top$] *LT'* .
with ‹*ins'* ≠ []› **show** *?thesis* **by**(*simp add*: *conf-f-def*)
qed
ultimately show *?thesis* **using** *tconf'* **by**(*fastforce simp add*: *correct-state-def*)
qed
ultimately show *?thesis* **by**(*clarsimp*)
qed

lemma *correct-state-heap-change*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *red*: $P,t \vdash$ *Normal* (*xcp*, *h*, *frs*) −*ta*−*jvmd*→ *Normal* (*xcp'*, *h'*, *frs'*)
and *cs*: $\Phi \vdash t$: (*xcp*, *h*, *frs*) $\surd$
and *cs''*: $\Phi \vdash t''$: (*xcp''*, *h*, *frs''*) $\surd$
shows $\Phi \vdash t''$: (*xcp''*, *h'*, *frs''*) $\surd$
proof(*cases xcp*)
case *None*
from *cs* **have** $P,h \vdash t \surd t$ **by**(*simp add*: *correct-state-def*)
with *red* **have** *hext h h'* **by** (*auto intro*: *exec-1-d-hext simp add*: *tconf-def*)
from ‹*wf-jvm-prog*$_\Phi$ *P*› *cs red* **have** $\Phi \vdash t$: (*xcp'*, *h'*, *frs'*) $\surd$
by(*auto elim*!: *jvmd-NormalE intro*: *BV-correct-1 simp add*: *exec-1-iff*)
from *cs''* **have** $P,h \vdash t'' \surd t$ **by**(*simp add*: *correct-state-def*)
with ‹$h \trianglelefteq h'$› **have** *tconf'*: $P,h' \vdash t'' \surd t$ **by**−(*rule tconf-hext-mono*)

```
  from ‹Φ ⊢ t: (xcp', h', frs') √›
  have hconf': hconf h' preallocated h' by(simp-all add: correct-state-def)

  show ?thesis
  proof(cases frs'')
    case Nil thus ?thesis using tconf' hconf' by(simp add: correct-state-def)
  next
    case (Cons f'' Frs'')
    obtain stk'' loc'' C0'' M0'' pc''
      where f'' = (stk'', loc'', C0'', M0'', pc'')
      by(cases f'', blast)
    with ‹frs'' = f'' # Frs''› cs''
    obtain Ts'' T'' mxs'' mxl₀'' ins'' xt'' ST'' LT''
      where hconf h
      and sees'': P ⊢ C0'' sees M0'': Ts''→T'' = ⌊(mxs'', mxl₀'', ins'', xt'')⌋ in C0''
      and Φ C0'' M0'' ! pc'' = ⌊(ST'', LT'')⌋
      and conf-f P h (ST'', LT'') ins'' (stk'', loc'', C0'', M0'', pc'')
      and conf-fs P h Φ M0'' (length Ts'') T'' Frs''
      by(fastforce simp add: correct-state-def)

    show ?thesis using Cons ‹Φ ⊢ t'': (xcp'', h, frs'') √› ‹hext h h'› hconf' tconf'
      apply(cases xcp'')
      apply(auto simp add: correct-state-def)
      apply(blast dest: hext-objD intro: conf-fs-hext conf-f-hext)+
      done
  qed
next
  case (Some a)
  with ‹P,t ⊢ Normal (xcp, h, frs) −ta−jvmd→ Normal (xcp', h', frs')›
  have h = h' by(auto elim!: jvmd-NormalE)
  with ‹Φ ⊢ t'': (xcp'', h, frs'') √› show ?thesis by simp
qed

lemma lifting-wf-correct-state-d:
  wf-jvm-progΦ P ⟹ lifting-wf JVM-final (mexecd P) (λt (xcp, frs) h. Φ ⊢ t: (xcp, h, frs) √)
by(unfold-locales)(auto intro: BV-correct-d-1 correct-state-new-thread correct-state-heap-change)

lemma lifting-wf-correct-state:
  assumes wf: wf-jvm-progΦ P
  shows lifting-wf JVM-final (mexec P) (λt (xcp, frs) h. Φ ⊢ t: (xcp, h, frs) √)
proof(unfold-locales)
  fix t x m ta x' m'
  assume mexec P t (x, m) ta (x', m')
    and (λ(xcp, frs) h. Φ ⊢ t: (xcp, h, frs) √) x m
  with wf show (λ(xcp, frs) h. Φ ⊢ t: (xcp, h, frs) √) x' m'
   by(cases x)(cases x', simp add: welltyped-commute[symmetric, OF ‹wf-jvm-progΦ P›], rule BV-correct-d-1)
next
  fix t x m ta x' m' t'' x''
  assume mexec P t (x, m) ta (x', m')
    and (λ(xcp, frs) h. Φ ⊢ t: (xcp, h, frs) √) x m
    and NewThread t'' x'' m' ∈ set {|ta|}t
  with wf show (λ(xcp, frs) h. Φ ⊢ t'': (xcp, h, frs) √) x'' m'
    apply(cases x, cases x', cases x'', clarify, unfold welltyped-commute[symmetric, OF ‹wf-jvm-progΦ
P›])
```

by(*rule correct-state-new-thread*)
next
fix *t x m ta x′ m′ t″ x″*
assume *mexec P t* (*x*, *m*) *ta* (*x′*, *m′*)
and (λ(*xcp*, *frs*) *h*. Φ ⊢ *t*: (*xcp*, *h*, *frs*) √) *x m*
and (λ(*xcp*, *frs*) *h*. Φ ⊢ *t″*: (*xcp*, *h*, *frs*) √) *x″ m*
with *wf* **show** (λ(*xcp*, *frs*) *h*. Φ ⊢ *t″*: (*xcp*, *h*, *frs*) √) *x″ m′*
by(*cases x*)(*cases x′*, *cases x″*, *clarify*, *unfold welltyped-commute*[*symmetric*, *OF* ‹*wf-jvm-prog*$_\Phi$ *P*›], *rule correct-state-heap-change*)
qed

lemmas *preserves-correct-state* = *FWLiftingSem.lifting-wf.RedT-preserves*[*OF lifting-wf-correct-state*]
lemmas *preserves-correct-state-d* = *FWLiftingSem.lifting-wf.RedT-preserves*[*OF lifting-wf-correct-state-d*]

end

context *JVM-heap-conf-base* **begin**

definition *correct-jvm-state* :: *ty*$_P$ ⇒ (′*addr*,′*thread-id*,′*addr jvm-thread-state*,′*heap*,′*addr*) *state set*
where
correct-jvm-state Φ
= {*s*. *correct-state-ts* Φ (*thr s*) (*shr s*) ∧ *lock-thread-ok* (*locks s*) (*thr s*)}

end

context *JVM-heap-conf* **begin**

lemma *correct-jvm-state-initial*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *wf-start*: *wf-start-state P C M vs*
shows *JVM-start-state P C M vs* ∈ *correct-jvm-state* Φ
proof −
from *wf-start* **obtain** *Ts T m D*
where *start-heap-ok* **and** *P* ⊢ *C sees M*:*Ts*→*T* = ⌊*m*⌋ *in D*
and *P*,*start-heap* ⊢ *vs* [:≤] *Ts* **by** *cases*
with *wf BV-correct-initial*[*OF wf this*] **show** *?thesis*
by(*cases m*)(*auto simp add*: *correct-jvm-state-def start-state-def JVM-start-state′-def intro*: *lock-thread-okI ts-okI split*: *if-split-asm*)
qed

end

context *JVM-conf-read* **begin**

lemma *invariant3p-correct-jvm-state-mexecdT*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
shows *invariant3p* (*mexecdT P*) (*correct-jvm-state* Φ)
unfolding *correct-jvm-state-def*
apply(*rule invariant3pI*)
apply *safe*
apply(*erule* (*1*) *lifting-wf.redT-preserves*[*OF lifting-wf-correct-state-d*[*OF wf*]])
apply(*erule* (*1*) *execd-mthr.redT-preserves-lock-thread-ok*)
done

lemma *invariant3p-correct-jvm-state-mexecT*:
assumes *wf*: *wf-jvm-prog$_\Phi$ P*
shows *invariant3p* (*mexecT P*) (*correct-jvm-state* Φ)
unfolding *correct-jvm-state-def*
apply(*rule invariant3pI*)
apply *safe*
apply(*erule* (*1*) *lifting-wf.redT-preserves*[*OF lifting-wf-correct-state*[*OF wf*]])
apply(*erule* (*1*) *exec-mthr.redT-preserves-lock-thread-ok*)
done

lemma *correct-jvm-state-preserved*:
assumes *wf*: *wf-jvm-prog$_\Phi$ P*
and *correct*: *s* $\in$ *correct-jvm-state* Φ
and *red*: *P* $\vdash$ *s* $-\triangleright ttas\rightarrow_{jvm*}$ *s'*
shows *s'* $\in$ *correct-jvm-state* Φ
using *wf red correct* **unfolding** *exec-mthr.RedT-def*
by(*rule invariant3p-rtrancl3p*[*OF invariant3p-correct-jvm-state-mexecT*])

theorem *jvm-typesafe*:
assumes *wf*: *wf-jvm-prog$_\Phi$ P*
and *start*: *wf-start-state P C M vs*
and *exec*: *P* $\vdash$ *JVM-start-state P C M vs* $-\triangleright ttas\rightarrow_{jvm*}$ *s'*
shows *s'* $\in$ *correct-jvm-state* Φ
by(*rule correct-jvm-state-preserved*[*OF wf - exec*])(*rule correct-jvm-state-initial*[*OF wf start*])

end

declare (**in** *JVM-typesafe*) *split-paired-Ex* [*simp del*]

context *JVM-heap-conf-base'* **begin**

lemma *execd-NewThread-Thread-Object*:
assumes *wf*: *wf-jvm-prog$_\Phi$ P*
and *conf*: $\Phi \vdash t'$: $\sigma\ \surd$
and *red*: *P,t'* $\vdash$ *Normal* σ $-ta-jvmd\rightarrow$ *Normal* σ'
and *nt*: *NewThread t x m* $\in$ *set* $\{\!|ta|\!\}_t$
shows $\exists$ *C. typeof-addr* (*fst* (*snd* σ')) (*thread-id2addr t*) = $\lfloor$*Class-type C*$\rfloor$ $\wedge$ *P* $\vdash$ *Class C* $\leq$ *Class Thread*
proof −
from *wf* **obtain** *wfmd* **where** *wfp*: *wf-prog wfmd P* **by**(*blast dest*: *wt-jvm-progD*)
from *red* **obtain** *h f Frs xcp*
where *check*: *check P* σ
and *exec*: (*ta*, σ') $\in$ *exec P t'* σ
and [*simp*]: σ = (*xcp*, *h*, *f* # *Frs*)
by(*rule jvmd-NormalE*)
obtain *xcp' h' frs'* **where** [*simp*]: σ' = (*xcp'*, *h'*, *frs'*) **by**(*cases* σ', *auto*)
obtain *stk loc C M pc* **where** [*simp*]: *f* = (*stk*, *loc*, *C*, *M*, *pc*) **by**(*cases f*, *blast*)
from *exec nt* **have** [*simp*]: *xcp* = *None* **by**(*cases xcp*, *auto*)
from ‹$\Phi \vdash t'$: $\sigma\ \surd$› **obtain** *Ts T mxs mxl0 ins xt ST LT*
where *hconf h*
and *P,h* $\vdash$ *t'* $\surd t$
and *sees*: *P* $\vdash$ *C sees M*: *Ts*$\rightarrow$*T* = $\lfloor$(*mxs*, *mxl0*, *ins*, *xt*)$\rfloor$ *in C*
and Φ *C M* ! *pc* = $\lfloor$(*ST*, *LT*)$\rfloor$

and *conf-f P h* (*ST*, *LT*) *ins* (*stk*, *loc*, *C*, *M*, *pc*)
and *conf-fs P h* Φ *M* (*length Ts*) *T Frs*
by(*fastforce simp add*: *correct-state-def*)
from *wf red conf nt*
obtain *h frs a stk loc C M pc M' n ta' va h'*
where *ha*: *typeof-addr h a* ≠ *None* **and** *ta*: *ta* = *extTA2JVM P ta'*
and *σ'*: *σ'* = *extRet2JVM n h' stk loc C M pc frs va*
and *rel*: (*ta'*, *va*, *h'*) ∈ *red-external-aggr P t' a M'* (*rev* (*take n stk*)) *h*
by −(*erule* (*2*) *exec-new-threadE*, *fastforce*+)
from *nt ta* **obtain** *x'* **where** *NewThread t x' m* ∈ *set* ⦃*ta'*⦄*t* **by** *auto*
from *red-external-aggr-new-thread-exists-thread-object*[*OF rel ha this*] *σ'*
show *?thesis* **by**(*cases va*) *auto*
qed

lemma *mexecdT-NewThread-Thread-Object*:
⟦ *wf-jvm-prog*Φ *P*; *correct-state-ts* Φ (*thr s*) (*shr s*); *P* ⊢ *s* −*t'*▷*ta*→*jvmd s'*; *NewThread t x m* ∈ *set* ⦃*ta*⦄*t* ⟧
⟹ ∃ *C*. *typeof-addr* (*shr s'*) (*thread-id2addr t*) = ⌊*Class-type C*⌋ ∧ *P* ⊢ *C* ≼* *Thread*
apply(*frule correct-state-ts-thread-conf*)
apply(*erule execd-mthr.redT.cases*)
apply(*hypsubst*)
apply(*frule* (*2*) *execd-tconf.redT-updTs-preserves*[**where** *ln'*=*redT-updLns* (*locks s*) *t' no-wait-locks* ⦃*ta*⦄*l*])
apply *clarsimp*
apply(*clarsimp*)
apply(*drule execd-NewThread-Thread-Object*)
apply(*drule* (*1*) *ts-okD*)
apply(*fastforce*)
apply(*assumption*)
apply(*fastforce*)
apply(*clarsimp*)
apply(*simp*)
done

end

context *JVM-heap* **begin**

lemma *exec-ta-satisfiable*:
assumes *P*,*t* ⊢ *s* −*ta*−*jvm*→ *s'*
shows ∃ *s*. *exec-mthr.actions-ok s t ta*
proof −
obtain *xcp h frs* **where** [*simp*]: *s* = (*xcp*, *h*, *frs*) **by**(*cases s*)
from *assms* **obtain** *stk loc C M pc frs'* **where** [*simp*]: *frs* = (*stk*, *loc*, *C*, *M*, *pc*) # *frs'*
by(*cases frs*)(*auto simp add*: *exec-1-iff*)
show *?thesis*
proof(*cases xcp*)
case *Some* **with** *assms* **show** *?thesis* **by**(*auto simp add*: *exec-1-iff lock-ok-las-def finfun-upd-apply split-paired-Ex*)
next
case *None*
with *assms* **show** *?thesis*
apply(*cases instrs-of P C M* ! *pc*)
apply(*auto simp add*: *exec-1-iff lock-ok-las-def finfun-upd-apply split-beta final-thread.actions-ok-iff*

split: *if-split-asm dest*: *red-external-aggr-ta-satisfiable*[**where** *final=JVM-final*])
apply(*fastforce simp add*: *final-thread.actions-ok-iff lock-ok-las-def dest*: *red-external-aggr-ta-satisfiable*[**where** *final=JVM-final*])
apply(*fastforce simp add*: *finfun-upd-apply intro*: *exI*[**where** *x=K$ None*] *exI*[**where** *x=K$* ⌊(*t*, *0*)⌋] *may-lock.intros*)+
done
qed
qed

end

context *JVM-typesafe* **begin**

lemma *execd-wf-progress*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
shows *progress JVM-final* (*mexecd P*) (*execd-mthr.wset-Suspend-ok P* (*correct-jvm-state* Φ))
(**is** *progress - - ?wf-state*)
proof
{
fix *s t x ta x' m' w*
assume *mexecd*: *mexecd P t* (*x*, *shr s*) *ta* (*x'*, *m'*)
and *Suspend*: *Suspend w* ∈ *set* ⦃*ta*⦄$_w$
from *mexecd-Suspend-Invoke*[*OF mexecd Suspend*]
show ¬ *JVM-final x'* **by** *auto*
}
note *Suspend-final* = *this*
{
fix *s*
assume *s*: *s* ∈ *?wf-state*
hence *lock-thread-ok* (*locks s*) (*thr s*)
by(*auto dest*: *execd-mthr.wset-Suspend-okD1 simp add*: *correct-jvm-state-def*)
moreover
have *exec-mthr.wset-final-ok* (*wset s*) (*thr s*)
proof(*rule exec-mthr.wset-final-okI*)
fix *t w*
assume *wset s t* = ⌊*w*⌋
from *execd-mthr.wset-Suspend-okD2*[*OF s this*]
obtain *x0 ta x m1 w' ln''* **and** *s0* :: ('*addr*, '*thread-id*, '*addr option* × '*addr frame list*, '*heap*, '*addr*) *state*
where *mexecd*: *mexecd P t* (*x0*, *shr s0*) *ta* (*x*, *m1*)
and *Suspend*: *Suspend w'* ∈ *set* ⦃*ta*⦄$_w$
and *tst*: *thr s t* = ⌊(*x*, *ln''*)⌋ **by** *blast*
from *Suspend-final*[*OF mexecd Suspend*] *tst*
show ∃*x ln*. *thr s t* = ⌊(*x*, *ln*)⌋ ∧ ¬ *JVM-final x* **by** *blast*
qed
ultimately show *lock-thread-ok* (*locks s*) (*thr s*) ∧ *exec-mthr.wset-final-ok* (*wset s*) (*thr s*) **..**
}
next
fix *s t x ta x' m'*
assume *wfs*: *s* ∈ *?wf-state*
and *thr s t* = ⌊(*x*, *no-wait-locks*)⌋
and *mexecd P t* (*x*, *shr s*) *ta* (*x'*, *m'*)
and *wait*: ¬ *waiting* (*wset s t*)
moreover obtain *ls ts h ws is* **where** *s* [*simp*]: *s* = (*ls*, (*ts*, *h*), *ws*, *is*) **by**(*cases s*) *fastforce*

ultimately have *ts t = ⌊(x, no-wait-locks)⌋ mexecd P t (x, h) ta (x′, m′)* **by** *auto*
from *wfs* **have** *correct-state-ts Φ ts h* **by**(*auto dest*: *execd-mthr.wset-Suspend-okD1 simp add*: *correct-jvm-state-def*)
from *wf* **obtain** *wfmd* **where** *wfp*: *wf-prog wfmd P* **by**(*auto dest*: *wt-jvm-progD*)

from ‹*ts t = ⌊(x, no-wait-locks)⌋*› ‹*mexecd P t (x, h) ta (x′, m′)*›
obtain *xcp frs xcp′ frs′*
where *P,t ⊢ Normal (xcp, h, frs) −ta−jvmd→ Normal (xcp′, m′, frs′)*
and [*simp*]: *x = (xcp, frs) x′ = (xcp′, frs′)*
by(*cases x, auto*)
then obtain *f Frs*
where *check*: *check P (xcp, h, f # Frs)*
and [*simp*]: *frs = f # Frs*
and *exec*: *(ta, xcp′, m′, frs′) ∈ exec P t (xcp, h, f # Frs)*
by(*auto elim*: *jvmd-NormalE*)
with ‹*ts t = ⌊(x, no-wait-locks)⌋*› ‹*correct-state-ts Φ ts h*›
have *correct*: *Φ ⊢ t: (xcp, h, f # Frs) √* **by**(*auto dest*: *ts-okD*)
obtain *stk loc C M pc* **where** *f* [*simp*]: *f = (stk, loc, C, M, pc)* **by** (*cases f*)
from *correct* **obtain** *Ts T mxs mxl0 ins xt ST LT*
where *hconf*: *hconf h*
and *tconf*: *P, h ⊢ t √t*
and *sees*: *P ⊢ C sees M:Ts→T = ⌊(mxs, mxl0, ins, xt)⌋ in C*
and *wt*: *Φ C M ! pc = ⌊(ST, LT)⌋*
and *conf-f*: *conf-f P h (ST, LT) ins (stk, loc, C, M, pc)*
and *confs*: *conf-fs P h Φ M (length Ts) T Frs*
and *confxcp*: *conf-xcp P h xcp (ins ! pc)*
and *preh*: *preallocated h*
by(*fastforce simp add*: *correct-state-def*)

have *∃ta′ σ′. P,t ⊢ Normal (xcp, h, (stk, loc, C, M, pc) # Frs) −ta′−jvmd→ Normal σ′ ∧*
(final-thread.actions-ok JVM-final (ls, (ts, h), ws, is) t ta′ ∨
final-thread.actions-ok′ (ls, (ts, h), ws, is) t ta′ ∧ final-thread.actions-subset ta′ ta)
proof(*cases final-thread.actions-ok′ (ls, (ts, h), ws, is) t ta*)
case *True*
have *final-thread.actions-subset ta ta* **..**
with *True* ‹*P,t ⊢ Normal (xcp, h, frs) −ta−jvmd→ Normal (xcp′, m′, frs′)*›
show *?thesis* **by** *auto*
next
case *False*
note *naok = this*
have *ws*: *wset s t = None ∨*
(∃ n a T w. ins ! pc = Invoke wait n ∧ stk ! n = Addr a ∧ typeof-addr h a = ⌊T⌋ ∧ is-native P T wait ∧ wset s t = ⌊PostWS w⌋ ∧ xcp = None)
proof(*cases wset s t*)
case *None* **thus** *?thesis* **..**
next
case (*Some w*)
from *execd-mthr.wset-Suspend-okD2*[*OF wfs this*] ‹*ts t = ⌊(x, no-wait-locks)⌋*›
obtain *xcp0 frs0 h0 ta0 w′ s1 tta1*
where *red0*: *mexecd P t ((xcp0, frs0), h0) ta0 ((xcp, frs), shr s1)*
and *Suspend*: *Suspend w′ ∈ set {|ta0|}w*
and *s1*: *P ⊢ s1 −▷tta1→jvmd* s*
by *auto*
from *mexecd-Suspend-Invoke*[*OF red0 Suspend*] *sees*

```
    obtain n a T where [simp]: ins ! pc = Invoke wait n xcp = None stk ! n = Addr a
      and type: typeof-addr h0 a = ⌊T⌋
      and iec: is-native P T wait
      by(auto simp add: is-native.simps) blast

    from red0 have h0 ⊴ shr s1 by(auto dest: exec-1-d-hext)
    also from s1 have shr s1 ⊴ shr s by(rule Execd-hext)
    finally have typeof-addr (shr s) a = ⌊T⌋ using type
      by(rule typeof-addr-hext-mono)
    moreover from Some wait s obtain w' where ws t = ⌊PostWS w'⌋
      by(auto simp add: not-waiting-iff)
    ultimately show ?thesis using iec s by auto
  qed

  from ws naok exec sees
  show ?thesis
  proof(cases ins ! pc)
    case (Invoke M' n)
    from ws Invoke check exec sees naok obtain a Ts U Ta Us D D'
      where a: stk ! n = Addr a
      and n: n < length stk
      and Ta: typeof-addr h a = ⌊Ta⌋
      and wtext: P ⊢ class-type-of Ta sees M':Us→U = Native in D' D'·M'(Us)::U
      and sub: P ⊢ Ts [≤] Us
      and Ts: map typeof_h (rev (take n stk)) = map Some Ts
      and [simp]: xcp = None
      apply(cases xcp)
    apply(simp add: is-Ref-def has-method-def external-WT'-iff check-def lock-ok-las'-def confs-conv-map
split-beta split: if-split-asm option.splits)
      apply(auto simp add: lock-ok-las'-def)[2]
      apply(fastforce simp add: is-native.simps lock-ok-las'-def dest: sees-method-fun)+
      done
    from exec Ta n a sees Invoke wtext obtain ta' va m''
      where exec': (ta', va, m'') ∈ red-external-aggr P t a M' (rev (take n stk)) h
      and ta: ta = extTA2JVM P ta'
      and va: (xcp', m', frs') = extRet2JVM n m'' stk loc C M pc Frs va
      by(auto)
    from va have [simp]: m'' = m' by(cases va) simp-all
    from Ta Ts wtext sub have wtext': P,h ⊢ a·M'(rev (take n stk)) : U
      by(auto intro!: external-WT'.intros simp add: is-native.simps)
    with wfp exec' tconf have red: P,t ⊢ ⟨a·M'(rev (take n stk)), h⟩ −ta'→ext ⟨va, m'⟩
      by(simp add: WT-red-external-list-conv)
    from ws Invoke have wset s t = None ∨ M' = wait ∧ (∃ w. wset s t = ⌊PostWS w⌋) by auto
    with wfp red tconf hconf obtain ta'' va' h''
      where red': P,t ⊢ ⟨a·M'(rev (take n stk)),h⟩ −ta''→ext ⟨va',h''⟩
        and ok': final-thread.actions-ok JVM-final s t ta'' ∨ final-thread.actions-ok' s t ta'' ∧ fi-
nal-thread.actions-subset ta'' ta'
      by(rule red-external-wf-red)
    from red' a n Ta Invoke sees wtext
    have (extTA2JVM P ta'', extRet2JVM n h'' stk loc C M pc Frs va') ∈ exec P t (xcp, h, f # Frs)
      by(auto intro: red-external-imp-red-external-aggr)
    with check have P,t ⊢ Normal (xcp, h, (stk, loc, C, M, pc) # Frs) −extTA2JVM P ta''−jvmd→
Normal (extRet2JVM n h'' stk loc C M pc Frs va')
      by −(rule exec-1-d.exec-1-d-NormalI, auto simp add: exec-d-def)
```

```
      moreover from ok' ta
      have final-thread.actions-ok JVM-final (ls, (ts, h), ws, is) t (extTA2JVM P ta'') ∨
        final-thread.actions-ok' (ls, (ts, h), ws, is) t (extTA2JVM P ta'') ∧ final-thread.actions-subset
(extTA2JVM P ta'') ta
        by(auto simp add: final-thread.actions-ok'-convert-extTA elim: final-thread.actions-subset.cases
del: subsetI)
      ultimately show ?thesis by blast
    next
      case MEnter
      with exec sees naok ws have False
        by(cases xcp)(auto split: if-split-asm simp add: lock-ok-las'-def finfun-upd-apply ta-upd-simps)
      thus ?thesis ..
    next
      case MExit
     with exec sees False check ws obtain a where [simp]: hd stk = Addr a xcp = None ws t = None
       and ta: ta = {|Unlock→a, SyncUnlock a|} ∨ ta = {|UnlockFail→a|}
        by(cases xcp)(fastforce split: if-split-asm simp add: lock-ok-las'-def finfun-upd-apply is-Ref-def
check-def)+
      from ta show ?thesis
      proof(rule disjE)
        assume ta: ta = {|Unlock→a, SyncUnlock a|}
        let ?ta' = {|UnlockFail→a|}
        from ta exec sees MExit obtain σ'
          where (?ta', σ') ∈ exec P t (xcp, h, f # Frs) by auto
        with check have P,t ⊢ Normal (xcp, h, (stk, loc, C, M, pc) # Frs) −?ta'−jvmd→ Normal σ'
          by −(rule exec-1-d.exec-1-d-NormalI, auto simp add: exec-d-def)
        moreover from False ta have has-locks (ls $ a) t = 0
          by(auto simp add: lock-ok-las'-def finfun-upd-apply ta-upd-simps)
        hence final-thread.actions-ok' (ls, (ts, h), ws, is) t ?ta'
          by(auto simp add: lock-ok-las'-def finfun-upd-apply ta-upd-simps)
        moreover from ta have final-thread.actions-subset ?ta' ta
        by(auto simp add: final-thread.actions-subset-iff collect-locks'-def finfun-upd-apply ta-upd-simps)
        ultimately show ?thesis by(fastforce simp add: ta-upd-simps)
      next
        assume ta: ta = {|UnlockFail→a|}
        let ?ta' = {|Unlock→a, SyncUnlock a|}
        from ta exec sees MExit obtain σ'
          where (?ta', σ') ∈ exec P t (xcp, h, f # Frs) by auto
        with check have P,t ⊢ Normal (xcp, h, (stk, loc, C, M, pc) # Frs) −?ta'−jvmd→ Normal σ'
          by −(rule exec-1-d.exec-1-d-NormalI, auto simp add: exec-d-def)
        moreover from False ta have has-lock (ls $ a) t
          by(auto simp add: lock-ok-las'-def finfun-upd-apply ta-upd-simps)
        hence final-thread.actions-ok' (ls, (ts, h), ws, is) t ?ta'
          by(auto simp add: lock-ok-las'-def finfun-upd-apply ta-upd-simps)
        moreover from ta have final-thread.actions-subset ?ta' ta
        by(auto simp add: final-thread.actions-subset-iff collect-locks'-def finfun-upd-apply ta-upd-simps)
        ultimately show ?thesis by(fastforce simp add: ta-upd-simps)
      qed
    qed(case-tac [!] xcp, auto simp add: split-beta lock-ok-las'-def split: if-split-asm)
  qed
  thus ∃ta' x' m'. mexecd P t (x, shr s) ta' (x', m') ∧
                   (final-thread.actions-ok JVM-final s t ta' ∨
                    final-thread.actions-ok' s t ta' ∧ final-thread.actions-subset ta' ta)
    by fastforce
```

next
fix *s t x*
assume *wfs*: *s* ∈ *?wf-state*
and *tst*: *thr s t* = ⌊(*x*, *no-wait-locks*)⌋
and ¬ *JVM-final x*
from *wfs* **have** *correct*: *correct-state-ts* Φ (*thr s*) (*shr s*)
by(*auto dest*: *execd-mthr.wset-Suspend-okD1 simp add*: *correct-jvm-state-def*)
obtain *xcp frs* **where** *x*: *x* = (*xcp*, *frs*) **by** (*cases x*, *auto*)
with ‹¬ *JVM-final x*› **obtain** *f Frs* **where** *frs* = *f* # *Frs*
by(*fastforce simp add*: *neq-Nil-conv*)
with *tst correct x* **have** Φ ⊢ *t*: (*xcp*, *shr s*, *f* # *Frs*) √ **by**(*auto dest*: *ts-okD*)
with ‹*wf-jvm-prog*$_\Phi$ *P*›
have *exec-d P t* (*xcp*, *shr s*, *f* # *Frs*) ≠ *TypeError* **by**(*auto dest*: *no-type-error*)
then obtain Σ **where** *exec-d P t* (*xcp*, *shr s*, *f* # *Frs*) = *Normal* Σ **by**(*auto*)
hence *exec P t* (*xcp*, *shr s*, *f* # *Frs*) = Σ
by(*auto simp add*: *exec-d-def check-def split*: *if-split-asm*)
with *progress*[*OF wf* ‹Φ ⊢ *t*: (*xcp*, *shr s*, *f* # *Frs*) √›]
obtain *ta σ* **where** (*ta*, *σ*) ∈ Σ **unfolding** *exec-1-iff* **by** *blast*
with ‹*x* = (*xcp*, *frs*)› ‹*frs* = *f* # *Frs*› ‹Φ ⊢ *t*: (*xcp*, *shr s*, *f* # *Frs*) √›
‹*wf-jvm-prog*$_\Phi$ *P*› ‹*exec-d P t* (*xcp*, *shr s*, *f* # *Frs*) = *Normal* Σ›
show ∃ *ta x′ m′*. *mexecd P t* (*x*, *shr s*) *ta* (*x′*, *m′*)
by(*cases ta*, *cases σ*)(*fastforce simp add*: *split-paired-Ex intro*: *exec-1-d-NormalI*)
qed(*fastforce dest*: *defensive-imp-aggressive-1 mexec-instr-Wakeup-no-Join exec-ta-satisfiable*)+

end

context *JVM-conf-read* **begin**

lemma *mexecT-eq-mexecdT*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *cs*: *correct-state-ts* Φ (*thr s*) (*shr s*)
shows *P* ⊢ *s* −*t*▷*ta*→$_{jvm}$ *s′* = *P* ⊢ *s* −*t*▷*ta*→$_{jvmd}$ *s′*
proof(*rule iffI*)
assume *P* ⊢ *s* −*t*▷*ta*→$_{jvm}$ *s′*
thus *P* ⊢ *s* −*t*▷*ta*→$_{jvmd}$ *s′*
proof(*cases rule*: *exec-mthr.redT-elims*[*consumes 1*, *case-names normal acquire*])
case (*normal x x′ m′*)
obtain *xcp frs* **where** *x* [*simp*]: *x* = (*xcp*, *frs*) **by**(*cases x*, *auto*)
from ‹*thr s t* = ⌊(*x*, *no-wait-locks*)⌋› *cs*
have Φ ⊢ *t*: (*xcp*, *shr s*, *frs*) √ **by**(*auto dest*: *ts-okD*)
from *mexec-eq-mexecd*[*OF wf* ‹Φ ⊢ *t*: (*xcp*, *shr s*, *frs*) √›] ‹*mexec P t* (*x*, *shr s*) *ta* (*x′*, *m′*)›
have ∗: *mexecd P t* (*x*, *shr s*) *ta* (*x′*, *m′*) **by** *simp*
with *lifting-wf.redT-updTs-preserves*[*OF lifting-wf-correct-state-d*[*OF wf*] *cs*, *OF this* ‹*thr s t* = ⌊(*x*, *no-wait-locks*)⌋›] ‹*thread-oks* (*thr s*) ⦃*ta*⦄$_t$›
have *correct-state-ts* Φ ((*redT-updTs* (*thr s*) ⦃*ta*⦄$_t$)(*t* ↦ (*x′*, *redT-updLns* (*locks s*) *t no-wait-locks* ⦃*ta*⦄$_l$))) *m′* **by** *simp*
with ∗ **show** *?thesis* **using** *normal*
by(*cases s′*)(*erule execd-mthr.redT-normal*, *auto*)
next
case *acquire* **thus** *?thesis*
apply(*cases s′*, *clarify*)
apply(*rule execd-mthr.redT-acquire*, *assumption*+)
by(*auto*)
qed

next
assume $P \vdash s\ -t\triangleright ta\rightarrow_{jvmd}\ s'$
thus $P \vdash s\ -t\triangleright ta\rightarrow_{jvm}\ s'$
proof(*cases rule*: *execd-mthr.redT-elims*[*consumes 1*, *case-names normal acquire*])
case (*normal x x' m'*)
obtain *xcp frs* **where** *x* [*simp*]: $x = (xcp, frs)$ **by**(*cases x*, *auto*)
from ‹*thr s t* = ⌊(*x*, *no-wait-locks*)⌋› *cs*
have $\Phi \vdash t$: (*xcp*, *shr s*, *frs*) $\surd$ **by**(*auto dest*: *ts-okD*)
from *mexec-eq-mexecd*[*OF wf* ‹$\Phi \vdash t$: (*xcp*, *shr s*, *frs*) $\surd$›] ‹*mexecd P t* (*x*, *shr s*) *ta* (*x'*, *m'*)›
have *mexec P t* (*x*, *shr s*) *ta* (*x'*, *m'*) **by** *simp*
moreover from *lifting-wf.redT-updTs-preserves*[*OF lifting-wf-correct-state-d*[*OF wf*] *cs*, *OF* ‹*mexecd P t* (*x*, *shr s*) *ta* (*x'*, *m'*)› ‹*thr s t* = ⌊(*x*, *no-wait-locks*)⌋›] ‹*thread-oks* (*thr s*) $\{\!|ta|\!\}_t$›
have *correct-state-ts* Φ ((*redT-updTs* (*thr s*) $\{\!|ta|\!\}_t$)($t \mapsto$ (*x'*, *redT-updLns* (*locks s*) *t no-wait-locks* $\{\!|ta|\!\}_l$))) *m'* **by** *simp*
ultimately show *?thesis* **using** *normal*
by(*cases s'*)(*erule exec-mthr.redT-normal*, *auto*)
next
case *acquire* **thus** *?thesis*
apply(*cases s'*, *clarify*)
apply(*rule exec-mthr.redT-acquire*, *assumption*+)
by(*auto*)
qed
qed

lemma *mExecT-eq-mExecdT*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *ct*: *correct-state-ts* Φ (*thr s*) (*shr s*)
shows $P \vdash s\ -\triangleright ttas\rightarrow_{jvm}{}^*\ s' = P \vdash s\ -\triangleright ttas\rightarrow_{jvmd}{}^*\ s'$
proof
assume *Red*: $P \vdash s\ -\triangleright ttas\rightarrow_{jvm}{}^*\ s'$
thus $P \vdash s\ -\triangleright ttas\rightarrow_{jvmd}{}^*\ s'$ **using** *ct*
proof(*induct rule*: *exec-mthr.RedT-induct*[*consumes 1*, *case-names refl step*])
case *refl* **thus** *?case* **by** *auto*
next
case (*step s ttas s' t ta s''*)
hence $P \vdash s\ -\triangleright ttas\rightarrow_{jvmd}{}^*\ s'$ **by** *blast*
moreover from ‹*correct-state-ts* Φ (*thr s*) (*shr s*)› ‹$P \vdash s\ -\triangleright ttas\rightarrow_{jvm}{}^*\ s'$›
have *correct-state-ts* Φ (*thr s'*) (*shr s'*)
by(*auto dest*: *preserves-correct-state*[*OF wf*])
with ‹$P \vdash s'\ -t\triangleright ta\rightarrow_{jvm}\ s''$› **have** $P \vdash s'\ -t\triangleright ta\rightarrow_{jvmd}\ s''$
by(*unfold mexecT-eq-mexecdT*[*OF wf*])
ultimately show *?case*
by(*blast intro*: *execd-mthr.RedTI rtrancl3p-step elim*: *execd-mthr.RedTE*)
qed
next
assume *Red*: $P \vdash s\ -\triangleright ttas\rightarrow_{jvmd}{}^*\ s'$
thus $P \vdash s\ -\triangleright ttas\rightarrow_{jvm}{}^*\ s'$ **using** *ct*
proof(*induct rule*: *execd-mthr.RedT-induct*[*consumes 1*, *case-names refl step*])
case *refl* **thus** *?case* **by** *auto*
next
case (*step s ttas s' t ta s''*)
hence $P \vdash s\ -\triangleright ttas\rightarrow_{jvm}{}^*\ s'$ **by** *blast*
moreover from ‹*correct-state-ts* Φ (*thr s*) (*shr s*)› ‹$P \vdash s\ -\triangleright ttas\rightarrow_{jvmd}{}^*\ s'$›

```
    have correct-state-ts Φ (thr s') (shr s')
      by(auto dest: preserves-correct-state-d[OF wf])
    with ‹P ⊢ s' −t▷ta→jvmd s''› have P ⊢ s' −t▷ta→jvm s''
      by(unfold mexecT-eq-mexecdT[OF wf])
    ultimately show ?case
      by(blast intro: exec-mthr.RedTI rtrancl3p-step elim: exec-mthr.RedTE)
  qed
qed

lemma mexecT-preserves-thread-conf:
  ⟦ wf-jvm-prog_Φ P; correct-state-ts Φ (thr s) (shr s);
    P ⊢ s −t'▷ta→jvm s'; thread-conf P (thr s) (shr s) ⟧
  ⟹ thread-conf P (thr s') (shr s')
by(simp only: mexecT-eq-mexecdT)(rule execd-tconf.redT-preserves)

lemma mExecT-preserves-thread-conf:
  ⟦ wf-jvm-prog_Φ P; correct-state-ts Φ (thr s) (shr s);
    P ⊢ s −▷tta→jvm* s'; thread-conf P (thr s) (shr s) ⟧
  ⟹ thread-conf P (thr s') (shr s')
by(simp only: mExecT-eq-mExecdT)(rule execd-tconf.RedT-preserves)

lemma wset-Suspend-ok-mexecd-mexec:
  assumes wf: wf-jvm-prog_Φ P
 shows exec-mthr.wset-Suspend-ok P (correct-jvm-state Φ) = execd-mthr.wset-Suspend-ok P (correct-jvm-state
Φ)
apply(safe)
 apply(rule execd-mthr.wset-Suspend-okI)
  apply(erule exec-mthr.wset-Suspend-okD1)
 apply(drule (1) exec-mthr.wset-Suspend-okD2)
 apply(subst (asm) (2) split-paired-Ex)
 apply(elim bexE exE conjE)
 apply(subst (asm) mexec-eq-mexecd[OF wf])
  apply(simp add: correct-jvm-state-def)
  apply(blast dest: ts-okD)
 apply(subst (asm) mexecT-eq-mexecdT[OF wf])
  apply(simp add: correct-jvm-state-def)
 apply(subst (asm) mExecT-eq-mExecdT[OF wf])
  apply(simp add: correct-jvm-state-def)
 apply(rule bexI exI|erule conjI|assumption)+
apply(rule exec-mthr.wset-Suspend-okI)
 apply(erule execd-mthr.wset-Suspend-okD1)
apply(drule (1) execd-mthr.wset-Suspend-okD2)
apply(subst (asm) (2) split-paired-Ex)
apply(elim bexE exE conjE)
apply(subst (asm) mexec-eq-mexecd[OF wf, symmetric])
 apply(simp add: correct-jvm-state-def)
 apply(blast dest: ts-okD)
apply(subst (asm) mexecT-eq-mexecdT[OF wf, symmetric])
 apply(simp add: correct-jvm-state-def)
apply(subst (asm) mExecT-eq-mExecdT[OF wf, symmetric])
 apply(simp add: correct-jvm-state-def)
apply(rule bexI exI|erule conjI|assumption)+
done
```

end

context *JVM-typesafe* **begin**

lemma *exec-wf-progress*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
shows *progress JVM-final* (*mexec P*) (*exec-mthr.wset-Suspend-ok P* (*correct-jvm-state* Φ))
(**is** *progress - - ?wf-state*)
proof −
interpret *progress*: *progress JVM-final mexecd P convert-RA ?wf-state*
using *assms* **unfolding** *wset-Suspend-ok-mexecd-mexec*[*OF wf*] **by**(*rule execd-wf-progress*)
show *?thesis*
proof(*unfold-locales*)
fix *s*
assume *s* ∈ *?wf-state*
thus *lock-thread-ok* (*locks s*) (*thr s*) ∧ *exec-mthr.wset-final-ok* (*wset s*) (*thr s*)
by(*rule progress.wf-stateD*)
next
fix *s t x ta x′ m′*
assume *wfs*: *s* ∈ *?wf-state*
and *tst*: *thr s t* = ⌊(*x*, *no-wait-locks*)⌋
and *exec*: *mexec P t* (*x*, *shr s*) *ta* (*x′*, *m′*)
and *wait*: ¬ *waiting* (*wset s t*)
from *wfs tst* **have** *correct*: Φ ⊢ *t*: (*fst x*, *shr s*, *snd x*) √
by(*auto dest*!: *exec-mthr.wset-Suspend-okD1 ts-okD simp add*: *correct-jvm-state-def*)
with *exec* **have** *mexecd P t* (*x*, *shr s*) *ta* (*x′*, *m′*)
by(*cases x*)(*simp only*: *mexec-eq-mexecd*[*OF wf*] *fst-conv snd-conv*)
from *progress.wf-red*[*OF wfs tst this wait*] *correct*
show ∃ *ta′ x′ m′*. *mexec P t* (*x*, *shr s*) *ta′* (*x′*, *m′*) ∧
(*final-thread.actions-ok JVM-final s t ta′* ∨
final-thread.actions-ok′ s t ta′ ∧ *final-thread.actions-subset ta′ ta*)
by(*cases x*)(*simp only*: *fst-conv snd-conv mexec-eq-mexecd*[*OF wf*])
next
fix *s t x ta x′ m′ w*
assume *wfs*: *s* ∈ *?wf-state*
and *tst*: *thr s t* = ⌊(*x*, *no-wait-locks*)⌋
and *exec*: *mexec P t* (*x*, *shr s*) *ta* (*x′*, *m′*)
and *wait*: ¬ *waiting* (*wset s t*)
and *Suspend*: *Suspend w* ∈ *set* {|*ta*|}$_w$
from *wfs tst* **have** *correct*: Φ ⊢ *t*: (*fst x*, *shr s*, *snd x*) √
by(*auto dest*!: *exec-mthr.wset-Suspend-okD1 ts-okD simp add*: *correct-jvm-state-def*)
with *exec* **have** *mexecd P t* (*x*, *shr s*) *ta* (*x′*, *m′*)
by(*cases x*)(*simp only*: *mexec-eq-mexecd*[*OF wf*] *fst-conv snd-conv*)
with *wfs tst* **show** ¬ *JVM-final x′* **using** *wait Suspend* **by**(*rule progress.red-wait-set-not-final*)
next
fix *s t x*
assume *wfs*: *s* ∈ *?wf-state*
and *tst*: *thr s t* = ⌊(*x*, *no-wait-locks*)⌋
and ¬ *JVM-final x*
from *progress.wf-progress*[*OF this*]
show ∃ *ta x′ m′*. *mexec P t* (*x*, *shr s*) *ta* (*x′*, *m′*)
by(*auto dest*: *defensive-imp-aggressive-1 simp add*: *split-beta*)
qed(*fastforce dest*: *mexec-instr-Wakeup-no-Join exec-ta-satisfiable*)+
qed

theorem *mexecd-TypeSafety*:
fixes *ln* :: *'addr* ⇒*f nat*
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *s*: *s* ∈ *execd-mthr.wset-Suspend-ok P* (*correct-jvm-state* Φ)
and *Exec*: *P* ⊢ *s* −▷*ttas*→$_{jvmd}$* *s'*
and ¬ (∃ *t ta s''*. *P* ⊢ *s'* −*t*▷*ta*→$_{jvmd}$ *s''*)
and *ts't*: *thr s' t* = ⌊((*xcp*, *frs*), *ln*)⌋
shows *frs* ≠ [] ∨ *ln* ≠ *no-wait-locks* ⟹ *t* ∈ *execd-mthr.deadlocked P s'*
and Φ ⊢ *t*: (*xcp*, *shr s'*, *frs*) √
proof −
interpret *progress JVM-final mexecd P convert-RA execd-mthr.wset-Suspend-ok P* (*correct-jvm-state* Φ)
by(*rule execd-wf-progress*) *fact*+

from *Exec s* **have** *wfs'*: *s'* ∈ *execd-mthr.wset-Suspend-ok P* (*correct-jvm-state* Φ)
unfolding *execd-mthr.RedT-def*
by(*blast intro*: *invariant3p-rtrancl3p execd-mthr.invariant3p-wset-Suspend-ok invariant3p-correct-jvm-state-mexecdT*[*OF wf*])

with *ts't* **show** *cst*: Φ ⊢ *t*: (*xcp*, *shr s'*, *frs*) √
by(*auto dest*: *ts-okD execd-mthr.wset-Suspend-okD1 simp add*: *correct-jvm-state-def*)
assume *nfin*: *frs* ≠ [] ∨ *ln* ≠ *no-wait-locks*
from *nfin* ‹*thr s' t* = ⌊((*xcp*, *frs*), *ln*)⌋› **have** *exec-mthr.not-final-thread s' t*
by(*auto simp*: *exec-mthr.not-final-thread-iff*)
from ‹¬ (∃ *t ta s''*. *P* ⊢ *s'* −*t*▷*ta*→$_{jvmd}$ *s''*)›
show *t* ∈ *execd-mthr.deadlocked P s'*
proof(*rule contrapos-np*)
assume *t* ∉ *execd-mthr.deadlocked P s'*
with ‹*exec-mthr.not-final-thread s' t*› **have** ¬ *execd-mthr.deadlocked' P s'*
by(*auto simp add*: *execd-mthr.deadlocked'-def*)
hence ¬ *execd-mthr.deadlock P s'* **unfolding** *execd-mthr.deadlock-eq-deadlocked'* .
thus ∃ *t ta s''*. *P* ⊢ *s'* −*t*▷*ta*→$_{jvmd}$ *s''* **by**(*rule redT-progress*[*OF wfs'*])
qed
qed

theorem *mexec-TypeSafety*:
fixes *ln* :: *'addr* ⇒*f nat*
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *s*: *s* ∈ *exec-mthr.wset-Suspend-ok P* (*correct-jvm-state* Φ)
and *Exec*: *P* ⊢ *s* −▷*ttas*→$_{jvm}$* *s'*
and ¬ (∃ *t ta s''*. *P* ⊢ *s'* −*t*▷*ta*→$_{jvm}$ *s''*)
and *ts't*: *thr s' t* = ⌊((*xcp*, *frs*), *ln*)⌋
shows *frs* ≠ [] ∨ *ln* ≠ *no-wait-locks* ⟹ *t* ∈ *multithreaded-base.deadlocked JVM-final* (*mexec P*) *s'*
and Φ ⊢ *t*: (*xcp*, *shr s'*, *frs*) √
proof −
interpret *progress JVM-final mexec P convert-RA exec-mthr.wset-Suspend-ok P* (*correct-jvm-state* Φ)
by(*rule exec-wf-progress*) *fact*+

from *Exec s* **have** *wfs'*: *s'* ∈ *exec-mthr.wset-Suspend-ok P* (*correct-jvm-state* Φ)
unfolding *exec-mthr.RedT-def*
by(*blast intro*: *invariant3p-rtrancl3p exec-mthr.invariant3p-wset-Suspend-ok invariant3p-correct-jvm-state-mexecT*[*OF wf*])

```
  with ts't show cst: Φ ⊢ t: (xcp, shr s', frs) √
    by(auto dest: ts-okD exec-mthr.wset-Suspend-okD1 simp add: correct-jvm-state-def)

  assume nfin: frs ≠ [] ∨ ln ≠ no-wait-locks
  from nfin ‹thr s' t = ⌊((xcp, frs), ln)⌋› have exec-mthr.not-final-thread s' t
    by(auto simp: exec-mthr.not-final-thread-iff)
  from ‹¬ (∃ t ta s''. P ⊢ s' −t▷ta→jvm s'')›
  show t ∈ exec-mthr.deadlocked P s'
  proof(rule contrapos-np)
    assume t ∉ exec-mthr.deadlocked P s'
    with ‹exec-mthr.not-final-thread s' t› have ¬ exec-mthr.deadlocked' P s'
      by(auto simp add: exec-mthr.deadlocked'-def)
    hence ¬ exec-mthr.deadlock P s' unfolding exec-mthr.deadlock-eq-deadlocked' .
    thus ∃ t ta s''. P ⊢ s' −t▷ta→jvm s'' by(rule redT-progress[OF wfs'])
  qed
qed

lemma start-mexec-mexecd-commute:
  assumes wf: wf-jvm-prog_Φ P
  and start: wf-start-state P C M vs
  shows P ⊢ JVM-start-state P C M vs −▷ttas→jvmd* s ⟷ P ⊢ JVM-start-state P C M vs
−▷ttas→jvm* s
using correct-jvm-state-initial[OF assms]
by(clarsimp simp add: correct-jvm-state-def)(rule mExecT-eq-mExecdT[symmetric, OF wf])

theorem mRtrancl-eq-mRtrancld:
  assumes wf: wf-jvm-prog_Φ P
  and ct: correct-state-ts Φ (thr s) (shr s)
  shows exec-mthr.mthr.Rtrancl3p P s ttas ⟷ execd-mthr.mthr.Rtrancl3p P s ttas (is ?lhs ⟷ ?rhs)
proof
  show ?lhs if ?rhs using that ct
  proof(coinduction arbitrary: s ttas)
    case Rtrancl3p
    interpret lifting-wf JVM-final mexecd P convert-RA λt (xcp, frs) h. Φ ⊢ t: (xcp, h, frs) √
      using wf by(rule lifting-wf-correct-state-d)
    from Rtrancl3p(1) show ?case
    proof cases
      case stop: Rtrancl3p-stop
      then show ?thesis using mexecT-eq-mexecdT[OF wf Rtrancl3p(2)] by clarsimp
    next
      case (Rtrancl3p-into-Rtrancl3p s' ttas' tta)
      then show ?thesis using mexecT-eq-mexecdT[OF wf Rtrancl3p(2)] Rtrancl3p(2)
        by(cases tta; cases s')(fastforce simp add: split-paired-Ex dest: redT-preserves)
    qed
  qed

  show ?rhs if ?lhs using that ct
  proof(coinduction arbitrary: s ttas)
    case Rtrancl3p
    interpret lifting-wf JVM-final mexec P convert-RA λt (xcp, frs) h. Φ ⊢ t: (xcp, h, frs) √
      using wf by(rule lifting-wf-correct-state)
    from Rtrancl3p(1) show ?case
    proof cases
```

case *stop*: *Rtrancl3p-stop*
then show *?thesis* **using** *mexecT-eq-mexecdT*[*OF wf Rtrancl3p*(*2*)] **by** *clarsimp*
next
case (*Rtrancl3p-into-Rtrancl3p s' ttas' tta*)
then show *?thesis* **using** *mexecT-eq-mexecdT*[*OF wf Rtrancl3p*(*2*)] *Rtrancl3p*(*2*)
by(*cases tta*; *cases s'*)(*fastforce simp add*: *split-paired-Ex dest*: *redT-preserves*)
qed
qed
qed

lemma *start-mRtrancl-mRtrancld-commute*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *start*: *wf-start-state P C M vs*
shows *exec-mthr.mthr.Rtrancl3p P* (*JVM-start-state P C M vs*) *ttas* $\longleftrightarrow$ *execd-mthr.mthr.Rtrancl3p P* (*JVM-start-state P C M vs*) *ttas*
using *correct-jvm-state-initial*[*OF assms*] **by**(*clarsimp simp add*: *correct-jvm-state-def mRtrancl-eq-mRtrancld*[*OF wf*])

end

6.7.1 Determinism

context *JVM-heap-conf* **begin**

lemma *exec-instr-deterministic*:
assumes *wf*: *wf-prog wf-md P*
and *det*: *deterministic-heap-ops*
and *exec1*: (*ta'*, *σ'*) ∈ *exec-instr i P t* (*shr s*) *stk loc C M pc frs*
and *exec2*: (*ta''*, *σ''*) ∈ *exec-instr i P t* (*shr s*) *stk loc C M pc frs*
and *check*: *check-instr i P* (*shr s*) *stk loc C M pc frs*
and *aok1*: *final-thread.actions-ok final s t ta'*
and *aok2*: *final-thread.actions-ok final s t ta''*
and *tconf*: *P*,*shr s* ⊢ *t* √*t*
shows *ta'* = *ta''* ∧ *σ'* = *σ''*
using *exec1 exec2 aok1 aok2*
proof(*cases i*)
case (*Invoke M' n*)
{ fix *T ta''' ta'''' va' va'' h' h''*
assume *T*: *typeof-addr* (*shr s*) (*the-Addr* (*stk* ! *n*)) = ⌊*T*⌋
and *method*: *snd* (*snd* (*snd* (*method P* (*class-type-of T*) *M'*))) = *None P* ⊢ *class-type-of T has M'*
and *params*: *P*,*shr s* ⊢ *rev* (*take n stk*) [:≤] *fst* (*snd* (*method P* (*class-type-of T*) *M'*))
and *red1*: (*ta'''*, *va'*, *h'*) ∈ *red-external-aggr P t* (*the-Addr* (*stk* ! *n*)) *M'* (*rev* (*take n stk*)) (*shr s*)
and *red2*: (*ta''''*, *va''*, *h''*) ∈ *red-external-aggr P t* (*the-Addr* (*stk* ! *n*)) *M'* (*rev* (*take n stk*)) (*shr s*)
and *ta'*: *ta'* = *extTA2JVM P ta'''*
and *ta''*: *ta''* = *extTA2JVM P ta''''*
from *T method params* **obtain** *T'* **where** *P*,*shr s* ⊢ *the-Addr* (*stk* ! *n*)·*M'*(*rev* (*take n stk*)) : *T'*
by(*fastforce simp add*: *has-method-def confs-conv-map external-WT'-iff*)
hence *P*,*t* ⊢ ⟨*the-Addr* (*stk* ! *n*)·*M'*(*rev* (*take n stk*)), *shr s*⟩ −*ta'''*→*ext* ⟨*va'*, *h'*⟩
and *P*,*t* ⊢ ⟨*the-Addr* (*stk* ! *n*)·*M'*(*rev* (*take n stk*)), *shr s*⟩ −*ta''''*→*ext* ⟨*va''*, *h''*⟩
using *red1 red2 tconf*
by−(*rule WT-red-external-aggr-imp-red-external*[*OF wf*], *assumption*+)+
moreover from *aok1 aok2 ta' ta''*

have *final-thread.actions-ok final s t ta'''*
and *final-thread.actions-ok final s t ta''''*
by(*auto simp add*: *final-thread.actions-ok-iff*)
ultimately have $ta''' = ta'''' \wedge va' = va'' \wedge h' = h''$
by(*rule red-external-deterministic*[*OF det*]) **}**
with *assms Invoke* **show** *?thesis*
by(*clarsimp simp add*: *split-beta split*: *if-split-asm*) *blast*
next
case *MExit*
{ assume *final-thread.actions-ok final s t* {⌊*UnlockFail→the-Addr* (*hd stk*)⌋}
and *final-thread.actions-ok final s t* {⌊*Unlock→the-Addr* (*hd stk*), *SyncUnlock* (*the-Addr* (*hd stk*))⌋}
hence *False*
by(*auto simp add*: *final-thread.actions-ok-iff lock-ok-las-def finfun-upd-apply elim*!: *allE*[**where** *x=the-Addr* (*hd stk*)]) **}**
with *assms MExit* **show** *?thesis* **by**(*auto split*: *if-split-asm*)
qed(*auto simp add*: *split-beta split*: *if-split-asm dest*: *deterministic-heap-ops-readD*[*OF det*] *deterministic-heap-ops-writeD*[*OF det*] *deterministic-heap-ops-allocateD*[*OF det*])

lemma *exec-1-deterministic*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *det*: *deterministic-heap-ops*
and *exec1*: $P,t \vdash (xcp, shr\ s, frs) -ta'-jvm\rightarrow \sigma'$
and *exec2*: $P,t \vdash (xcp, shr\ s, frs) -ta''-jvm\rightarrow \sigma''$
and *aok1*: *final-thread.actions-ok final s t ta'*
and *aok2*: *final-thread.actions-ok final s t ta''*
and *conf*: $\Phi \vdash t:(xcp, shr\ s, frs)\ \surd$
shows $ta' = ta'' \wedge \sigma' = \sigma''$
proof −
from *wf* **obtain** *wf-md* **where** *wf'*: *wf-prog wf-md P* **by**(*blast dest*: *wt-jvm-progD*)
from *conf* **have** *tconf*: $P,shr\ s \vdash t\ \surd t$ **by**(*simp add*: *correct-state-def*)
from *exec1 conf* **have** $P,t \vdash Normal\ (xcp, shr\ s, frs) -ta'-jvmd\rightarrow Normal\ \sigma'$
by(*simp add*: *welltyped-commute*[*OF wf*])
hence *check P* (*xcp*, *shr s*, *frs*) **by**(*rule jvmd-NormalE*)
with *exec1 exec2 aok1 aok2 tconf* **show** *?thesis*
by(*cases xcp*)(*case-tac* [!] *frs*, *auto elim*!: *exec-1.cases dest*: *exec-instr-deterministic*[*OF wf' det*] *simp add*: *check-def split-beta*)
qed

end

context *JVM-conf-read* **begin**

lemma *invariant3p-correct-state-ts*:
assumes *wf-jvm-prog*$_\Phi$ *P*
shows *invariant3p* (*mexecT P*) {*s*. *correct-state-ts* Φ (*thr s*) (*shr s*)}
using *assms* **by**(*rule lifting-wf.invariant3p-ts-ok*[*OF lifting-wf-correct-state*])

lemma *mexec-deterministic*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *det*: *deterministic-heap-ops*
shows *exec-mthr.deterministic P* {*s*. *correct-state-ts* Φ (*thr s*) (*shr s*)}
proof(*rule exec-mthr.determisticI*)
fix *s t x ta' x' m' ta'' x'' m''*
assume *tst*: *thr s t* = ⌊(*x*, *no-wait-locks*)⌋

and *red*: *mexec P t* (*x*, *shr s*) *ta′* (*x′*, *m′*) *mexec P t* (*x*, *shr s*) *ta″* (*x″*, *m″*)
and *aok*: *exec-mthr.actions-ok s t ta′ exec-mthr.actions-ok s t ta″*
and *correct* [*simplified*]: *s* ∈ {*s*. *correct-state-ts* Φ (*thr s*) (*shr s*)}
moreover obtain *xcp frs* **where** [*simp*]: *x* = (*xcp*, *frs*) **by**(*cases x*)
moreover obtain *xcp′ frs′* **where** [*simp*]: *x′* = (*xcp′*, *frs′*) **by**(*cases x′*)
moreover obtain *xcp″ frs″* **where** [*simp*]: *x″* = (*xcp″*, *frs″*) **by**(*cases x″*)
ultimately have *exec1*: *P*,*t* ⊢ (*xcp*, *shr s*, *frs*) −*ta′*−*jvm*→ (*xcp′*, *m′*, *frs′*)
and *exec1*: *P*,*t* ⊢ (*xcp*, *shr s*, *frs*) −*ta″*−*jvm*→ (*xcp″*, *m″*, *frs″*)
by *simp-all*
moreover note *aok*
moreover from *correct tst* **have** Φ ⊢ *t*:(*xcp*, *shr s*, *frs*)√
by(*auto dest*: *ts-okD*)
ultimately have *ta′* = *ta″* ∧ (*xcp′*, *m′*, *frs′*) = (*xcp″*, *m″*, *frs″*)
by(*rule exec-1-deterministic*[*OF wf det*])
thus *ta′* = *ta″* ∧ *x′* = *x″* ∧ *m′* = *m″* **by** *simp*
qed(*rule invariant3p-correct-state-ts*[*OF wf*])

end

end

6.8 Preservation of deadlock for the JVMs

theory *JVMDeadlocked*
imports
BVProgressThreaded
begin

context *JVM-progress* **begin**

lemma *must-sync-preserved-d*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *ml*: *execd-mthr.must-sync P t* (*xcp*, *frs*) *h*
and *hext*: *hext h h′*
and *hconf′*: *hconf h′*
and *cs*: Φ ⊢ *t*: (*xcp*, *h*, *frs*) √
shows *execd-mthr.must-sync P t* (*xcp*, *frs*) *h′*
proof(*rule execd-mthr.must-syncI*)
from *ml* **obtain** *ta xcp′ frs′ m′*
where *red*: *P*,*t* ⊢ *Normal* (*xcp*, *h*, *frs*) −*ta*−*jvmd*→ *Normal* (*xcp′*, *m′*, *frs′*)
by(*auto elim*: *execd-mthr.must-syncE*)
then obtain *f Frs*
where *check*: *check P* (*xcp*, *h*, *frs*)
and *exec*: (*ta*, *xcp′*, *m′*, *frs′*) ∈ *exec P t* (*xcp*, *h*, *frs*)
and [*simp*]: *frs* = *f* # *Frs*
by(*auto elim*: *jvmd-NormalE*)
from *cs hext hconf′* **have** *cs′*: Φ ⊢ *t*: (*xcp*, *h′*, *frs*) √
by(*rule correct-state-hext-mono*)
then obtain *ta σ′* **where** *exec*: *P*,*t* ⊢ (*xcp*, *h′*, *frs*) −*ta*−*jvm*→ *σ′*
by(*auto dest*: *progress*[*OF wf*])
hence *P*,*t* ⊢ *Normal* (*xcp*, *h′*, *frs*) −*ta*−*jvmd*→ *Normal σ′*
unfolding *welltyped-commute*[*OF wf cs′*] **.**
moreover from *exec* **have** ∃ *s*. *exec-mthr.actions-ok s t ta* **by**(*rule exec-ta-satisfiable*)

ultimately show $\exists ta\ x'\ m'$ *s. mexecd P t* $((xcp, frs), h')$ *ta* $(x', m') \wedge$ *exec-mthr.actions-ok s t ta*
by(*cases* σ')(*fastforce simp del*: *split-paired-Ex*)
qed

lemma *can-sync-devreserp-d*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *cl'*: *execd-mthr.can-sync P t* (xcp, frs) h' *L*
and *cs*: $\Phi \vdash t$: (xcp, h, frs) $\surd$
and *hext*: *hext h h'*
and *hconf'*: *hconf h'*
shows $\exists L' \subseteq L$. *execd-mthr.can-sync P t* (xcp, frs) *h L'*
proof −
from *cl'* **obtain** *ta xcp' frs' m'*
where *red*: *P*,*t* ⊢ *Normal* (xcp, h', frs) −*ta*−*jvmd*→ *Normal* (xcp', m', frs')
and *L*: *L* = *collect-locks* $\{\!|ta|\!\}_l$ <+> *collect-cond-actions* $\{\!|ta|\!\}_c$ <+> *collect-interrupts* $\{\!|ta|\!\}_i$
by −(*erule execd-mthr.can-syncE*, *auto*)
then obtain *f Frs*
where *check*: *check P* (xcp, h', frs)
and *exec*: $(ta, xcp', m', frs') \in$ *exec P t* (xcp, h', frs)
and [*simp*]: *frs* = *f* # *Frs*
by(*auto elim*: *jvmd-NormalE simp add*: *finfun-upd-apply*)
obtain *stk loc C M pc* **where** [*simp*]: *f* = (stk, loc, C, M, pc) **by** (*cases f*, *blast*)
from *cs* **obtain** *ST LT Ts T mxs mxl ins xt* **where**
hconf: *hconf h* **and**
tconf: *P*,*h* ⊢ *t* $\surd$*t* **and**
meth: *P* ⊢ *C sees M*:*Ts*→*T* = ⌊(*mxs*, *mxl*, *ins*, *xt*)⌋ *in C* **and**
Φ: Φ *C M* ! *pc* = *Some* (*ST*,*LT*) **and**
frame: *conf-f P h* (*ST*,*LT*) *ins* (*stk*,*loc*,*C*,*M*,*pc*) **and**
frames: *conf-fs P h* Φ *M* (*size Ts*) *T Frs*
by (*fastforce simp add*: *correct-state-def dest*: *sees-method-fun*)
from *cs* **have** *exec P t* $(xcp, h, f \# Frs) \neq \{\}$
by(*auto dest*!: *progress*[*OF wf*] *simp add*: *exec-1-iff*)
with *no-type-error*[*OF wf cs*] **have** *check'*: *check P* (xcp, h, frs)
by(*auto simp add*: *exec-d-def split*: *if-split-asm*)
from *wf* **obtain** *wfmd* **where** *wfp*: *wf-prog wfmd P* **by**(*auto dest*: *wt-jvm-progD*)
from *tconf hext* **have** *tconf'*: *P*,*h'* ⊢ *t* $\surd$*t* **by**(*rule tconf-hext-mono*)
show *?thesis*
proof(*cases xcp*)
case [*simp*]: (*Some a*)
with *exec* **have** [*simp*]: *m'* = *h'* **by**(*auto*)
from ‹Φ ⊢ *t*: (xcp, h, frs) $\surd$› **obtain** *D* **where** *D*: *typeof-addr h a* = ⌊*Class-type D*⌋
by(*auto simp add*: *correct-state-def*)
with *hext* **have** *cname-of h a* = *cname-of h' a* **by**(*auto dest*: *hext-objD simp add*: *cname-of-def*)
with *exec* **have** $(ta, xcp', h, frs') \in$ *exec P t* (xcp, h, frs) **by** *auto*
moreover from *check D hext* **have** *check P* (xcp, h, frs)
by(*auto simp add*: *check-def check-xcpt-def dest*: *hext-objD*)
ultimately have *P*,*t* ⊢ *Normal* (xcp, h, frs) −*ta*−*jvmd*→ *Normal* (xcp', h, frs')
by −(*rule exec-1-d-NormalI*, *simp only*: *exec-d-def if-True*)
with *L* **have** *execd-mthr.can-sync P t* (xcp, frs) *h L*
by(*auto intro*: *execd-mthr.can-syncI*)
thus *?thesis* **by** *auto*
next
case [*simp*]: *None*

note [*simp*] = *defs1 list-all2-Cons2*

from *frame* **have** *ST*: $P,h \vdash stk\ [:\leq]\ ST$
and *LT*: $P,h \vdash loc\ [:\leq_\top]\ LT$
and *pc*: $pc < length\ ins$ **by** *simp-all*
from *wf meth pc* **have** *wt*: $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
by(*rule wt-jvm-prog-impl-wt-instr*)

from $\langle \Phi \vdash t\text{: } (xcp,\ h,\ frs)\ \surd \rangle$
have $\exists ta\ \sigma'.\ P,t \vdash (xcp,\ h,\ f\ \#\ Frs)\ -ta-jvm\rightarrow\ \sigma'$
by(*auto dest*: *progress*[*OF wf*] *simp del*: *correct-state-def split-paired-Ex*)
with *exec meth* **have** $\exists ta'\ \sigma'.\ (ta',\ \sigma') \in exec\ P\ t\ (xcp,\ h,\ frs) \wedge collect\text{-}locks\ \{\!|ta'|\!\}_l \subseteq collect\text{-}locks\ \{\!|ta|\!\}_l \wedge collect\text{-}cond\text{-}actions\ \{\!|ta'|\!\}_c \subseteq collect\text{-}cond\text{-}actions\ \{\!|ta|\!\}_c \wedge collect\text{-}interrupts\ \{\!|ta'|\!\}_i \subseteq collect\text{-}interrupts\ \{\!|ta|\!\}_i$
proof(*cases ins* ! *pc*)
case (*Invoke M' n*)
show *?thesis*
proof(*cases stk* ! *n* = *Null*)
case *True* **with** *Invoke exec meth* **show** *?thesis* **by** *simp*
next
case *False*
with *check meth* **obtain** *a* **where** *a*: *stk* ! *n* = *Addr a* **and** *n*: *n* < *length stk*
by(*auto simp add*: *check-def is-Ref-def Invoke*)
from *frame* **have** *stk*: $P,h \vdash stk\ [:\leq]\ ST$ **by**(*auto simp add*: *conf-f-def*)
hence $P,h \vdash stk\ !\ n :\leq ST\ !\ n$ **using** *n* **by**(*rule list-all2-nthD*)
with *a* **obtain** *ao Ta* **where** *Ta*: *typeof-addr h a* = $\lfloor Ta \rfloor$
by(*auto simp add*: *conf-def*)
from *hext Ta* **have** *Ta'*: *typeof-addr h' a* = $\lfloor Ta \rfloor$ **by**(*rule typeof-addr-hext-mono*)
with *check a meth Invoke False* **obtain** *D Ts' T' meth D'*
where *C*: *D* = *class-type-of Ta*
and *sees'*: $P \vdash D\ sees\ M'\text{:}Ts' \rightarrow T' = meth\ in\ D'$
and *params*: $P,h' \vdash rev\ (take\ n\ stk)\ [:\leq]\ Ts'$
by(*auto simp add*: *check-def has-method-def*)
show *?thesis*
proof(*cases meth*)
case *Some*
with *exec meth a Ta Ta' Invoke n sees' C* **show** *?thesis* **by**(*simp add*: *split-beta*)
next
case *None*
with *exec meth a Ta Ta' Invoke n sees' C*
obtain *ta' va h''* **where** *ta'*: *ta* = *extTA2JVM P ta'*
and *va*: (*xcp'*, *m'*, *frs'*) = *extRet2JVM n h'' stk loc C M pc Frs va*
and *exec'*: (*ta'*, *va*, *h''*) ∈ *red-external-aggr P t a M'* (*rev* (*take n stk*)) *h'*
by(*fastforce*)
from *va* **have** [*simp*]: *h''* = *m'* **by**(*cases va*) *simp-all*
note *Ta* **moreover from** *None sees' wfp* **have** $D'\cdot M'(Ts') :: T'$ **by**(*auto intro*: *sees-wf-native*)
with *C sees' params Ta' None* **have** $P,h' \vdash a\cdot M'(rev\ (take\ n\ stk)) : T'$
by(*auto simp add*: *external-WT'-iff confs-conv-map*)
with *wfp exec' tconf'* **have** *red*: $P,t \vdash \langle a\cdot M'(rev\ (take\ n\ stk)),\ h' \rangle\ -ta' \rightarrow ext\ \langle va,\ m' \rangle$
by(*simp add*: *WT-red-external-list-conv*)

from *stk* **have** $P,h \vdash take\ n\ stk\ [:\leq]\ take\ n\ ST$ **by**(*rule list-all2-takeI*)
then obtain *Ts* **where** *map typeof*$_h$ (*take n stk*) = *map Some Ts*
by(*auto simp add*: *confs-conv-map*)

```
      hence map typeof_h (rev (take n stk)) = map Some (rev Ts) by(simp only: rev-map[symmetric])
       moreover hence map typeof_h' (rev (take n stk)) = map Some (rev Ts) using hext by(rule
map-typeof-hext-mono)
        with ‹P,h' ⊢ a•M'(rev (take n stk)) : T'› ‹D'•M'(Ts') :: T'› sees' C Ta' Ta
        have P ⊢ rev Ts [≤] Ts' by cases (auto dest: sees-method-fun)
        ultimately have P,h ⊢ a•M'(rev (take n stk)) : T'
         using Ta C sees' params None ‹D'•M'(Ts') :: T'›
         by(auto simp add: external-WT'-iff confs-conv-map)
        from red-external-wt-hconf-hext[OF wfp red hext this tconf hconf]
        obtain ta'' va' h''' where P,t ⊢ ⟨a•M'(rev (take n stk)),h⟩ −ta''→ext ⟨va',h'''⟩
         and ta'': collect-locks {|ta''|}_l = collect-locks {|ta'|}_l
         collect-cond-actions {|ta''|}_c = collect-cond-actions {|ta'|}_c
         collect-interrupts {|ta''|}_i = collect-interrupts {|ta'|}_i
         by auto
        with None a Ta Invoke meth Ta' n C sees'
        have (extTA2JVM P ta'', extRet2JVM n h''' stk loc C M pc Frs va') ∈ exec P t (xcp, h, frs)
         by(force intro: red-external-imp-red-external-aggr simp del: split-paired-Ex)
        with ta'' ta' show ?thesis by(fastforce simp del: split-paired-Ex)
       qed
      qed
    qed(auto 4 4 split: if-split-asm simp add: split-beta ta-upd-simps exec-1-iff intro: rev-image-eqI simp
del: split-paired-Ex)
     with check' have ∃ta' σ'. P,t ⊢ Normal (xcp, h, frs) −ta'−jvmd→ Normal σ' ∧ collect-locks
{|ta'|}_l ⊆ collect-locks {|ta|}_l ∧
     collect-cond-actions {|ta'|}_c ⊆ collect-cond-actions {|ta|}_c ∧ collect-interrupts {|ta'|}_i ⊆ collect-interrupts
{|ta|}_i
      apply clarify
      apply(rule exI conjI)+
      apply(rule exec-1-d.exec-1-d-NormalI, auto simp add: exec-d-def)
      done
     with L show ?thesis
      apply −
      apply(erule exE conjE|rule exI conjI)+
      prefer 2
      apply(rule-tac x'=(fst σ', snd (snd σ')) and m'=fst (snd σ') in execd-mthr.can-syncI)
      apply auto
      done
  qed
qed

end

context JVM-typesafe begin

lemma execd-preserve-deadlocked:
  assumes wf: wf-jvm-prog_Φ P
  shows preserve-deadlocked JVM-final (mexecd P) convert-RA (correct-jvm-state Φ)
proof(unfold-locales)
  show invariant3p (mexecdT P) (correct-jvm-state Φ)
    by(rule invariant3p-correct-jvm-state-mexecdT[OF wf])
next
  fix s t' ta' s' t x ln
  assume s: s ∈ correct-jvm-state Φ
    and red: P ⊢ s −t'▷ta'→_jvmd s'
```

```
    and tst: thr s t = ⌊(x, ln)⌋
    and execd-mthr.must-sync P t x (shr s)
  moreover obtain xcp frs where x [simp]: x = (xcp, frs) by(cases x, auto)
  ultimately have ml: execd-mthr.must-sync P t (xcp, frs) (shr s) by simp
  moreover from s have cs': correct-state-ts Φ (thr s) (shr s) by(simp add: correct-jvm-state-def)
  with tst have Φ ⊢ t: (xcp, shr s, frs) √ by(auto dest: ts-okD)
  moreover from red have hext (shr s) (shr s') by(rule execd-hext)
  moreover from wf red cs' have correct-state-ts Φ (thr s') (shr s')
    by(rule lifting-wf.redT-preserves[OF lifting-wf-correct-state-d])
  from red tst have thr s' t ≠ None
    by(cases s)(cases s', rule notI, auto dest: execd-mthr.redT-thread-not-disappear)
  with ‹correct-state-ts Φ (thr s') (shr s')› have hconf (shr s')
    by(auto dest: ts-okD simp add: correct-state-def)
  ultimately have execd-mthr.must-sync P t (xcp, frs) (shr s')
    by−(rule must-sync-preserved-d[OF wf])
  thus execd-mthr.must-sync P t x (shr s') by simp
next
  fix s t' ta' s' t x ln L
  assume s: s ∈ correct-jvm-state Φ
    and red: P ⊢ s −t'▷ta'→jvmd s'
    and tst: thr s t = ⌊(x, ln)⌋
    and execd-mthr.can-sync P t x (shr s') L
  moreover obtain xcp frs where x [simp]: x = (xcp, frs) by(cases x, auto)
  ultimately have ml: execd-mthr.can-sync P t (xcp, frs) (shr s') L by simp
  moreover from s have cs': correct-state-ts Φ (thr s) (shr s) by(simp add: correct-jvm-state-def)
  with tst have Φ ⊢ t: (xcp, shr s, frs) √ by(auto dest: ts-okD)
  moreover from red have hext (shr s) (shr s') by(rule execd-hext)
  moreover from red tst have thr s' t ≠ None
    by(cases s)(cases s', rule notI, auto dest: execd-mthr.redT-thread-not-disappear)
  from red cs' have correct-state-ts Φ (thr s') (shr s')
    by(rule lifting-wf.redT-preserves[OF lifting-wf-correct-state-d[OF wf]])
  with ‹thr s' t ≠ None› have hconf (shr s')
    by(auto dest: ts-okD simp add: correct-state-def)
  ultimately have ∃ L' ⊆ L. execd-mthr.can-sync P t (xcp, frs) (shr s) L'
    by−(rule can-sync-devreserp-d[OF wf])
  thus ∃ L' ⊆ L. execd-mthr.can-sync P t x (shr s) L' by simp
qed

end
```

and now everything again for the aggresive VM

```
context JVM-heap-conf-base' begin

lemma must-lock-d-eq-must-lock:
  ⟦ wf-jvm-prog_Φ P; Φ ⊢ t: (xcp, h, frs) √ ⟧
  ⟹ execd-mthr.must-sync P t (xcp, frs) h = exec-mthr.must-sync P t (xcp, frs) h
apply(rule iffI)
 apply(rule exec-mthr.must-syncI)
 apply(erule execd-mthr.must-syncE)
 apply(simp only: mexec-eq-mexecd)
 apply(blast)
apply(rule execd-mthr.must-syncI)
apply(erule exec-mthr.must-syncE)
apply(simp only: mexec-eq-mexecd[symmetric])
```

apply(*blast*)
done

lemma *can-lock-d-eq-can-lock*:
⟦ *wf-jvm-prog*$_\Phi$ *P*; Φ ⊢ *t*: (*xcp*, *h*, *frs*) √ ⟧
⟹ *execd-mthr.can-sync P t* (*xcp*, *frs*) *h L* = *exec-mthr.can-sync P t* (*xcp*, *frs*) *h L*
apply(*rule iffI*)
apply(*erule execd-mthr.can-syncE*)
apply(*rule exec-mthr.can-syncI*)
apply(*simp only*: *mexec-eq-mexecd*)
apply(*assumption*)+
apply(*erule exec-mthr.can-syncE*)
apply(*rule execd-mthr.can-syncI*)
by(*simp only*: *mexec-eq-mexecd*)

end

context *JVM-typesafe* **begin**

lemma *exec-preserve-deadlocked*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
shows *preserve-deadlocked JVM-final* (*mexec P*) *convert-RA* (*correct-jvm-state* Φ)
proof −
interpret *preserve-deadlocked JVM-final mexecd P convert-RA correct-jvm-state* Φ
by(*rule execd-preserve-deadlocked*) *fact*+

{ **fix** *s t' ta' s' t x*
assume *s*: *s* ∈ *correct-jvm-state* Φ
and *red*: *P* ⊢ *s* −*t'*▷*ta'*→$_{jvm}$ *s'*
and *tst*: *thr s t* = ⌊(*x*, *no-wait-locks*)⌋
obtain *xcp frs* **where** *x* [*simp*]: *x* = (*xcp*, *frs*) **by**(*cases x*, *auto*)
from *s* **have** *css*: *correct-state-ts* Φ (*thr s*) (*shr s*) **by**(*simp add*: *correct-jvm-state-def*)
with *red* **have** *redd*: *P* ⊢ *s* −*t'*▷*ta'*→$_{jvmd}$ *s'* **by**(*simp add*: *mexecT-eq-mexecdT*[*OF wf*])
from *css tst* **have** *cst*: Φ ⊢ *t*: (*xcp*, *shr s*, *frs*) √ **by**(*auto dest*: *ts-okD*)
from *redd* **have** *cst'*: Φ ⊢ *t*: (*xcp*, *shr s'*, *frs*) √
proof(*cases rule*: *execd-mthr.redT-elims*)
case *acquire* **with** *cst* **show** *?thesis* **by** *simp*
next
case (*normal X X' M' ws'*)
obtain *XCP FRS* **where** *X* [*simp*]: *X* = (*XCP*, *FRS*) **by**(*cases X*, *auto*)
obtain *XCP' FRS'* **where** *X'* [*simp*]: *X'* = (*XCP'*, *FRS'*) **by**(*cases X'*, *auto*)
from ‹*mexecd P t'* (*X*, *shr s*) *ta'* (*X'*, *M'*)›
have *P*,*t'* ⊢ *Normal* (*XCP*, *shr s*, *FRS*) −*ta'*−*jvmd*→ *Normal* (*XCP'*, *M'*, *FRS'*) **by** *simp*
moreover from ‹*thr s t'* = ⌊(*X*, *no-wait-locks*)⌋› *css*
have Φ ⊢ *t'*: (*XCP*, *shr s*, *FRS*) √ **by**(*auto dest*: *ts-okD*)
ultimately have Φ ⊢ *t'*: (*XCP*, *M'*, *FRS*) √ **by** −(*rule correct-state-heap-change*[*OF wf*])
moreover from *lifting-wf.redT-updTs-preserves*[*OF lifting-wf-correct-state-d*[*OF wf*] *css*, *OF* ‹*mexecd P t'* (*X*, *shr s*) *ta'* (*X'*, *M'*)› ‹*thr s t'* = ⌊(*X*, *no-wait-locks*)⌋›, *of no-wait-locks*] ‹*thread-oks* (*thr s*) ⦃*ta'*⦄$_t$›
have *correct-state-ts* Φ ((*redT-updTs* (*thr s*) ⦃*ta'*⦄$_t$)(*t'* ↦ (*X'*, *no-wait-locks*))) *M'* **by** *simp*
ultimately have *correct-state-ts* Φ (*redT-updTs* (*thr s*) ⦃*ta'*⦄$_t$) *M'*
using ‹*thr s t'* = ⌊(*X*, *no-wait-locks*)⌋› ‹*thread-oks* (*thr s*) ⦃*ta'*⦄$_t$›
apply(*auto intro*!: *ts-okI dest*: *ts-okD*)
apply(*case-tac t*=*t'*)

apply(*fastforce dest*: *redT-updTs-Some*)
apply(*drule-tac t*=*t* **in** *ts-okD*, *fastforce*+)
done
hence *correct-state-ts* Φ (*redT-updTs* (*thr s*) {|*ta'*|}$_t$) (*shr s'*)
using ‹*s'* = (*redT-updLs* (*locks s*) *t'* {|*ta'*|}$_l$, ((*redT-updTs* (*thr s*) {|*ta'*|}$_t$)(*t'* ↦ (*X'*, *redT-updLns* (*locks s*) *t' no-wait-locks* {|*ta'*|}$_l$)), *M'*), *ws'*, *redT-updIs* (*interrupts s*) {|*ta'*|}$_i$)›
by *simp*
moreover from *tst* ‹*thread-oks* (*thr s*) {|*ta'*|}$_t$›
have *redT-updTs* (*thr s*) {|*ta'*|}$_t$ *t* = ⌊(*x*, *no-wait-locks*)⌋ **by**(*auto intro*: *redT-updTs-Some*)
ultimately show *?thesis* **by**(*auto dest*: *ts-okD*)
qed
{ assume *exec-mthr.must-sync P t x* (*shr s*)
hence *ml*: *exec-mthr.must-sync P t* (*xcp*, *frs*) (*shr s*) **by** *simp*
with *cst* **have** *execd-mthr.must-sync P t* (*xcp*, *frs*) (*shr s*)
by(*auto dest*: *must-lock-d-eq-must-lock*[*OF wf*])
with *s redd tst* **have** *execd-mthr.must-sync P t x* (*shr s'*)
unfolding *x* **by**(*rule can-lock-preserved*)
with *cst'* **have** *exec-mthr.must-sync P t x* (*shr s'*)
by(*auto dest*: *must-lock-d-eq-must-lock*[*OF wf*]) **}**
note *ml* = *this*
{ fix *L*
assume *exec-mthr.can-sync P t x* (*shr s'*) *L*
hence *cl*: *exec-mthr.can-sync P t* (*xcp*, *frs*) (*shr s'*) *L* **by** *simp*
with *cst'* **have** *execd-mthr.can-sync P t* (*xcp*, *frs*) (*shr s'*) *L*
by(*auto dest*: *can-lock-d-eq-can-lock*[*OF wf*])
with *s redd tst*
have ∃ *L'* ⊆ *L*. *execd-mthr.can-sync P t x* (*shr s*) *L'*
unfolding *x* **by**(*rule can-lock-devreserp*)
then obtain *L'* **where** *execd-mthr.can-sync P t x* (*shr s*) *L'*
and *L'*: *L'*⊆ *L* **by** *blast*
with *cst* **have** *exec-mthr.can-sync P t x* (*shr s*) *L'*
by(*auto dest*: *can-lock-d-eq-can-lock*[*OF wf*])
with *L'* **have** ∃ *L'* ⊆ *L*. *exec-mthr.can-sync P t x* (*shr s*) *L'*
by(*blast*) **}**
note *this ml* **}**
moreover have *invariant3p* (*mexecT P*) (*correct-jvm-state* Φ) **by**(*rule invariant3p-correct-jvm-state-mexecT*[*OF wf*])
ultimately show *?thesis* **by**(*unfold-locales*)
qed

end

end

6.9 Monotonicity of eff and app

theory *EffectMono*
imports
Effect
begin

declare *not-Err-eq* [*iff*]

declare *widens-trans*[*trans*]

lemma app_i*-mono*:
  assumes *wf*: *wf-prog p P*
  assumes *less*: $P \vdash \tau \leq_i \tau'$
  shows $app_i\ (i,P,mxs,mpc,rT,\tau') \Longrightarrow app_i\ (i,P,mxs,mpc,rT,\tau)$
proof −
  assume *app*: $app_i\ (i,P,mxs,mpc,rT,\tau')$

  obtain *ST LT ST' LT'* **where**
    [*simp*]: $\tau = (ST,LT)$ **and**
    [*simp*]: $\tau' = (ST',LT')$
    by (*cases* τ, *cases* τ')

  from *less* **have** [*simp*]: *size ST* = *size ST'* **and** [*simp*]: *size LT* = *size LT'*
    by (*auto dest*: *list-all2-lengthD*)

  note [*iff*] = *list-all2-Cons2 widen-Class*
  note [*simp*] = *fun-of-def*

  from *app less* **show** $app_i\ (i,P,mxs,mpc,rT,\tau)$
  proof (*cases i*)
    case *Load*
    with *app less* **show** *?thesis* **by** (*auto dest*!: *list-all2-nthD*)
  next
    case (*Invoke M n*)
    with *app* **have** *n*: $n <$ *size ST'* **by** *simp*

    { assume *ST*!*n* = *NT* **hence** *?thesis* **using** *n app Invoke* **by** *simp* **}**
    moreover {
      assume *ST'*!*n* = *NT*
      moreover with *n less* **have** *ST*!*n* = *NT*
        by (*auto dest*: *list-all2-nthD*)
      ultimately have *?thesis* **using** *n app Invoke* **by** *simp* **}**
    moreover {
      assume *ST*: *ST*!*n* ≠ *NT* **and** *ST'*: *ST'*!*n* ≠ *NT*

      from *ST' app Invoke*
      obtain *D Ts T m C'*
        where *D*: *class-type-of'* (*ST'* ! *n*) = ⌊*D*⌋
        and *Ts*: $P \vdash$ *rev* (*take n ST'*) [≤] *Ts*
        and *D-M*: $P \vdash D$ *sees M*: *Ts*→*T* = *m in C'*
        by *fastforce*

      from *less* **have** $P \vdash$ *ST*!*n* ≤ *ST'*!*n*
        by(*auto dest*: *list-all2-nthD2*[*OF* - *n*])
      with *D* **obtain** *D'* **where** *D'*: *class-type-of'* (*ST* ! *n*) = ⌊*D'*⌋
        and *DsubC*: $P \vdash D' \preceq^* D$
        using *ST* **by**(*rule widen-is-class-type-of*)
      from *wf D-M DsubC* **obtain** *Ts' T' m' C''* **where**
        D'-M: $P \vdash D'$ *sees M*: *Ts'*→*T'* = *m' in C''* **and**
        Ts': $P \vdash$ *Ts* [≤] *Ts'*
        by (*blast dest*: *sees-method-mono*)
      from *less* **have** $P \vdash$ *rev* (*take n ST*) [≤] *rev* (*take n ST'*) **by** *simp*

```
    also note Ts also note Ts'
    finally have P ⊢ rev (take n ST) [≤] Ts' .
    with D'-M D' app less Invoke D have ?thesis by(auto)
  }
  ultimately show ?thesis by blast
next
  case Getfield
  with app less show ?thesis
    by(fastforce simp add: sees-field-def widen-Array dest: has-fields-fun)
next
  case Putfield
  with app less show ?thesis
  by (fastforce intro: widen-trans rtrancl-trans simp add: sees-field-def widen-Array dest: has-fields-fun)
next
  case CAS
  with app less show ?thesis
  by (fastforce intro: widen-trans rtrancl-trans simp add: sees-field-def widen-Array dest: has-fields-fun)
next
  case Return
  with app less show ?thesis by (fastforce intro: widen-trans)
next
  case ALoad
  with app less show ?thesis by(auto simp add: widen-Array)
next
  case AStore
  with app less show ?thesis by(auto simp add: widen-Array)
next
  case ALength
  with app less show ?thesis by(auto simp add: widen-Array)
next
  case (Checkcast T)
  with app less show ?thesis
    by(auto elim!: refTE simp: widen-Array)
next
  case (Instanceof T)
  with app less show ?thesis
    by(auto elim!: refTE simp: widen-Array)
next
  case ThrowExc
  with app less show ?thesis
    by(auto elim!: refTE simp: widen-Array)
next
  case MEnter
  with app less show ?thesis
    by(auto elim!: refTE simp: widen-Array)
next
  case MExit
  with app less show ?thesis
    by(auto elim!: refTE simp: widen-Array)
next
  case (BinOpInstr bop)
  with app less show ?thesis by(force dest: WTrt-binop-widen-mono)
next
  case Dup
```

```
    with app less show ?thesis
      by(auto dest: list-all2-lengthD)
  next
    case Swap
    with app less show ?thesis
      by(auto dest: list-all2-lengthD)
  qed (auto elim!: refTE not-refTE)
qed

lemma succs-mono:
  assumes wf: wf-prog p P and app_i: app_i (i,P,mxs,mpc,rT,τ')
  shows P ⊢ τ ≤_i τ' ⟹ set (succs i τ pc) ⊆ set (succs i τ' pc)
proof (cases i)
  case (Invoke M n)
  obtain ST LT ST' LT' where
    [simp]: τ = (ST,LT) and [simp]: τ' = (ST',LT') by (cases τ, cases τ')
  assume P ⊢ τ ≤_i τ'
  moreover
  with app_i Invoke have n < size ST by (auto dest: list-all2-lengthD)
  ultimately
  have P ⊢ ST!n ≤ ST'!n by (auto simp add: fun-of-def dest: list-all2-nthD)
  with Invoke show ?thesis by auto
next
  case ALoad
  obtain ST LT ST' LT' where
    [simp]: τ = (ST,LT) and [simp]: τ' = (ST',LT') by (cases τ, cases τ')
  assume P ⊢ τ ≤_i τ'
  moreover
  with app_i ALoad have 1 < size ST by (auto dest: list-all2-lengthD)
  ultimately
  have P ⊢ ST!1 ≤ ST'!1 by (auto simp add: fun-of-def dest: list-all2-nthD)
  with ALoad show ?thesis by auto
next
  case AStore
  obtain ST LT ST' LT' where
    [simp]: τ = (ST,LT) and [simp]: τ' = (ST',LT') by (cases τ, cases τ')
  assume P ⊢ τ ≤_i τ'
  moreover
  with app_i AStore have 2 < size ST by (auto dest: list-all2-lengthD)
  ultimately
  have P ⊢ ST!2 ≤ ST'!2 by (auto simp add: fun-of-def dest: list-all2-nthD)
  with AStore show ?thesis by auto
next
  case ALength
  obtain ST LT ST' LT' where
    [simp]: τ = (ST,LT) and [simp]: τ' = (ST',LT') by (cases τ, cases τ')
  assume P ⊢ τ ≤_i τ'
  moreover
  with app_i ALength have 0 < size ST by (auto dest: list-all2-lengthD)
  ultimately
  have P ⊢ ST!0 ≤ ST'!0 by (auto simp add: fun-of-def dest: list-all2-nthD)
  with ALength show ?thesis by auto
next
  case MEnter
```

obtain *ST LT ST′ LT′* **where**
[*simp*]: $\tau = (ST,LT)$ **and** [*simp*]: $\tau' = (ST',LT')$ **by** (*cases* τ, *cases* τ')
assume $P \vdash \tau \leq_i \tau'$
moreover
with app_i *MEnter* **have** *0 < size ST* **by** (*auto dest*: *list-all2-lengthD*)
ultimately
have $P \vdash ST!0 \leq ST'!0$ **by** (*auto simp add*: *fun-of-def dest*: *list-all2-nthD*)
with *MEnter* **show** *?thesis* **by** *auto*
next
case *MExit*
obtain *ST LT ST′ LT′* **where**
[*simp*]: $\tau = (ST,LT)$ **and** [*simp*]: $\tau' = (ST',LT')$ **by** (*cases* τ, *cases* τ')
assume $P \vdash \tau \leq_i \tau'$
moreover
with app_i *MExit* **have** *0 < size ST* **by** (*auto dest*: *list-all2-lengthD*)
ultimately
have $P \vdash ST!0 \leq ST'!0$ **by** (*auto simp add*: *fun-of-def dest*: *list-all2-nthD*)
with *MExit* **show** *?thesis* **by** *auto*
qed *auto*

lemma *app-mono*:
assumes *wf*: *wf-prog p P*
assumes *less′*: $P \vdash \tau \leq' \tau'$
shows *app i P m rT pc mpc xt* $\tau' \Longrightarrow$ *app i P m rT pc mpc xt* τ
proof (*cases* τ)
case *None* **thus** *?thesis* **by** *simp*
next
case (*Some* τ_1)
moreover
with *less′* **obtain** τ_2 **where** τ_2: $\tau' =$ *Some* τ_2 **by** (*cases* τ') *auto*
ultimately have *less*: $P \vdash \tau_1 \leq_i \tau_2$ **using** *less′* **by** *simp*

assume *app i P m rT pc mpc xt* τ'
with *Some* τ_2 **obtain**
app_i: app_i (i, P, pc, m, rT, τ_2) **and**
xcpt: *xcpt-app i P pc m xt* τ_2 **and**
succs: $\forall (pc',s') \in set$ (*eff i P pc xt* (*Some* τ_2)). $pc' < mpc$
by (*auto simp add*: *app-def*)

from *wf less* app_i **have** app_i (i, P, pc, m, rT, τ_1) **by** (*rule* app_i*-mono*)
moreover
from *less* **have** *size* (*fst* τ_1) = *size* (*fst* τ_2)
by (*cases* τ_1, *cases* τ_2) (*auto dest*: *list-all2-lengthD*)
with *xcpt* **have** *xcpt-app i P pc m xt* τ_1 **by** (*simp add*: *xcpt-app-def*)
moreover
from *wf* app_i *less* **have** $\forall pc.$ *set* (*succs i* τ_1 *pc*) $\subseteq$ *set* (*succs i* τ_2 *pc*)
by (*blast dest*: *succs-mono*)
with *succs*
have $\forall (pc',s') \in set$ (*eff i P pc xt* (*Some* τ_1)). $pc' < mpc$
by (*cases* τ_1, *cases* τ_2)
(*auto simp add*: *eff-def norm-eff-def xcpt-eff-def dest*: *bspec*)
ultimately
show *?thesis* **using** *Some* **by** (*simp add*: *app-def*)
qed

lemma *eff$_i$-mono*:
assumes *wf*: *wf-prog p P*
assumes *less*: $P \vdash \tau \leq_i \tau'$
assumes *app$_i$*: *app i P m rT pc mpc xt* (*Some* τ')
assumes *succs*: *succs i* τ *pc* $\neq$ [] *succs i* τ' *pc* $\neq$ []
shows $P \vdash$ *eff$_i$* (*i*,*P*,τ) $\leq_i$ *eff$_i$* (*i*,*P*,τ')
proof −
obtain *ST LT ST′ LT′* **where**
[*simp*]: τ = (*ST*,*LT*) **and**
[*simp*]: τ' = (*ST′*,*LT′*)
by (*cases* τ, *cases* τ')

note [*simp*] = *eff-def app-def fun-of-def*

from *less* **have** $P \vdash$ (*Some* τ) $\leq'$ (*Some* τ') **by** *simp*
from *wf this app$_i$*
have *app*: *app i P m rT pc mpc xt* (*Some* τ) **by** (*rule app-mono*)

from *less app app$_i$* **show** *?thesis*
proof (*cases i*)
case *ThrowExc* **with** *succs* **have** *False* **by** *simp*
thus *?thesis* **..**
next
case *Return* **with** *succs* **have** *False* **by** *simp*
thus *?thesis* **..**
next
case (*Load i*)
from *Load app* **obtain** *y* **where**
y: *i* < *size LT LT*![*i*] = *OK y* **by** *clarsimp*
from *Load app$_i$* **obtain** *y′* **where**
y′: *i* < *size LT′ LT′*![*i*] = *OK y′* **by** *clarsimp*

from *less* **have** $P \vdash LT\ [\leq_\top]\ LT'$ **by** *simp*
with *y y′* **have** $P \vdash y \leq y'$ **by** (*auto dest*: *list-all2-nthD*)
with *Load less y y′ app app$_i$*
show *?thesis* **by** *auto*
next
case *Store* **with** *less app app$_i$*
show *?thesis* **by** (*auto simp add*: *list-all2-update-cong*)
next
case (*Invoke M n*)
with *app$_i$* **have** *n*: *n* < *size ST′* **by** *simp*
from *less* **have** [*simp*]: *size ST* = *size ST′*
by (*auto dest*: *list-all2-lengthD*)

from *Invoke succs* **have** *ST*: *ST*![*n*] $\neq$ *NT* **and** *ST′*: *ST′*![*n*] $\neq$ *NT* **by** (*auto*)

from *ST′ app$_i$ Invoke* **obtain** *D Ts T m C′*
where *D*: *class-type-of′* (*ST′* ! *n*) = ⌊*D*⌋
and *Ts*: *P* ⊢ *rev* (*take n ST′*) [≤] *Ts*
and *D-M*: *P* ⊢ *D sees M*: *Ts*→*T* = *m in C′*
by *fastforce*

```
    from less have P ⊢ ST!n ≤ ST'!n by(auto dest: list-all2-nthD2[OF - n])
    with D obtain D' where D': class-type-of' (ST ! n) = ⌊D'⌋
      and DsubC: P ⊢ D' ≼* D
      using ST by(rule widen-is-class-type-of)

    from wf D-M DsubC obtain Ts' T' m' C'' where
      D'-M: P ⊢ D' sees M: Ts'→T' = m' in C'' and
      Ts': P ⊢ Ts [≤] Ts' and P ⊢ T' ≤ T by (blast dest: sees-method-mono)

    show ?thesis using Invoke n D D' D-M less D'-M Ts' ‹P ⊢ T' ≤ T›
      by(auto intro: list-all2-dropI)
  next
    case ALoad with less app app_i succs
    show ?thesis by(auto split: if-split-asm dest: Array-Array-widen)
  next
    case AStore with less app app_i succs
    show ?thesis by(auto split: if-split-asm dest: Array-Array-widen)
  next
    case (BinOpInstr bop)
    with less app app_i succs show ?thesis
      by auto(force dest: WTrt-binop-widen-mono WTrt-binop-fun)
  qed auto
qed

end
```

6.10 The Typing Framework for the JVM

```
theory TF-JVM
imports
  ../DFA/Typing-Framework-err
  EffectMono
  BVSpec
  ../Common/ExternalCallWF
begin

definition exec :: 'addr jvm-prog ⇒ nat ⇒ ty ⇒ ex-table ⇒ 'addr instr list ⇒ ty_i' err step-type
where
  exec G maxs rT et bs ≡
   err-step (size bs) (λpc. app (bs!pc) G maxs rT pc (size bs) et) (λpc. eff (bs!pc) G pc et)

locale JVM-sl =
  fixes P :: 'addr jvm-prog and mxs and mxl_0
  fixes Ts :: ty list and is :: 'addr instr list and xt and T_r

  fixes mxl and A and r and f and app and eff and step
  defines [simp]: mxl ≡ 1+size Ts+mxl_0
  defines [simp]: A   ≡ states P mxs mxl
  defines [simp]: r   ≡ JVM-SemiType.le P mxs mxl
  defines [simp]: f   ≡ JVM-SemiType.sup P mxs mxl

  defines [simp]: app ≡ λpc. Effect.app (is!pc) P mxs T_r pc (size is) xt
  defines [simp]: eff ≡ λpc. Effect.eff (is!pc) P pc xt
```

defines [*simp*]: *step* ≡ *err-step* (*size is*) *app eff*

locale *start-context* = *JVM-sl* +
fixes *p* **and** *C*
assumes *wf*: *wf-prog p P*
assumes *C*: *is-class P C*
assumes *Ts*: *set Ts* ⊆ *types P*

fixes *first* :: ty_i' **and** *start*
defines [*simp*]:
first ≡ *Some* ([],*OK* (*Class C*) # *map OK Ts* @ *replicate* mxl_0 *Err*)
defines [*simp*]:
start ≡ *OK first* # *replicate* (*size is* − *1*) (*OK None*)

6.10.1 Connecting JVM and Framework

lemma (**in** *JVM-sl*) *step-def-exec*: *step* ≡ *exec P mxs* T_r *xt is*
by (*simp add*: *exec-def*)

lemma *special-ex-swap-lemma* [*iff*]:
(∃ *X*. (∃ *n*. *X* = *A n* ∧ *P n*) ∧ *Q X*) = (∃ *n*. *Q*(*A n*) ∧ *P n*)
by *blast*

lemma *ex-in-list* [*iff*]:
(∃ *n*. *ST* ∈ *list n A* ∧ *n* ≤ *mxs*) = (*set ST* ⊆ *A* ∧ *size ST* ≤ *mxs*)
by (*unfold list-def*) *auto*

lemma *singleton-list*:
(∃ *n*. [*Class C*] ∈ *list n* (*types P*) ∧ *n* ≤ *mxs*) = (*is-class P C* ∧ *0* < *mxs*)
by(*auto*)

lemma *set-drop-subset*:
set xs ⊆ *A* ⟹ *set* (*drop n xs*) ⊆ *A*
by (*auto dest*: *in-set-dropD*)

lemma *Suc-minus-minus-le*:
n < *mxs* ⟹ *Suc* (*n* − (*n* − *b*)) ≤ *mxs*
by *arith*

lemma *in-listE*:
⟦ *xs* ∈ *list n A*; ⟦*size xs* = *n*; *set xs* ⊆ *A*⟧ ⟹ *P* ⟧ ⟹ *P*
by (*unfold list-def*) *blast*

declare *is-relevant-entry-def* [*simp*]
declare *set-drop-subset* [*simp*]

lemma (**in** *start-context*) [*simp*, *intro*!]: *is-class P Throwable*
apply(*rule converse-subcls-is-class*[*OF wf*])
apply(*rule xcpt-subcls-Throwable*[*OF - wf*])
prefer *2*
apply(*rule is-class-xcpt*[*OF - wf*])
apply(*fastforce simp add*: *sys-xcpts-def sys-xcpts-list-def*)+
done

declare *option.splits*[*split del*]
declare *option.case-cong*[*cong*]
declare *is-type-array* [*simp del*]

theorem (**in** *start-context*) *exec-pres-type*:
pres-type step (*size is*) *A*
declare *option.case-cong-weak*[*cong*]
declare *option.splits*[*split*]
declare *is-type-array*[*simp*]

declare *is-relevant-entry-def* [*simp del*]
declare *set-drop-subset* [*simp del*]

lemma *lesubstep-type-simple*:
xs [$\sqsubseteq_{Product.le\ (=)\ r}$] *ys* $\Longrightarrow$ *set xs* {$\sqsubseteq_r$} *set ys*
declare *is-relevant-entry-def* [*simp del*]

lemma *conjI2*: ⟦ *A*; *A* $\Longrightarrow$ *B* ⟧ $\Longrightarrow$ *A* $\wedge$ *B* **by** *blast*

lemma (**in** *JVM-sl*) *eff-mono*:
⟦*wf-prog p P*; *pc* < *length is*; *s* $\sqsubseteq_{sup\text{-}state\text{-}opt\ P}$ *t*; *app pc t*⟧
$\Longrightarrow$ *set* (*eff pc s*) {$\sqsubseteq_{sup\text{-}state\text{-}opt\ P}$} *set* (*eff pc t*)
lemma (**in** *JVM-sl*) *bounded-step*: *bounded step* (*size is*)
theorem (**in** *JVM-sl*) *step-mono*:
wf-prog wf-mb P $\Longrightarrow$ *mono r step* (*size is*) *A*

lemma (**in** *start-context*) *first-in-A* [*iff*]: *OK first* $\in$ *A*
using *Ts C* **by** (*force intro*!: *list-appendI simp add*: *JVM-states-unfold*)

lemma (**in** *JVM-sl*) *wt-method-def2*:
wt-method P C' Ts T_r *mxs* mxl_0 *is xt* τs =
(*is* $\neq$ [] $\wedge$
size τs = *size is* $\wedge$
OK ' *set* τs $\subseteq$ *states P mxs mxl* $\wedge$
wt-start P C' Ts mxl_0 τs $\wedge$
wt-app-eff (*sup-state-opt P*) *app eff* τs)

end

6.11 LBV for the JVM

theory *LBVJVM*
imports
../DFA/Abstract-BV
TF-JVM
begin

type-synonym *prog-cert* = *cname* $\Rightarrow$ *mname* $\Rightarrow$ ty_i' *err list*

definition *check-cert* :: *'addr jvm-prog* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ ty_i' *err list* $\Rightarrow$ *bool*

where
check-cert P mxs mxl n cert ≡ *check-types P mxs mxl cert* ∧ *size cert* = *n*+1 ∧
(∀ *i*<*n*. *cert*!*i* ≠ *Err*) ∧ *cert*!*n* = *OK None*

definition *lbvjvm* :: *'addr jvm-prog* ⇒ *nat* ⇒ *nat* ⇒ *ty* ⇒ *ex-table* ⇒
ty$_i$' *err list* ⇒ *'addr instr list* ⇒ *ty*$_i$' *err* ⇒ *ty*$_i$' *err*
where
lbvjvm P mxs maxr T_r *et cert bs* ≡
wtl-inst-list bs cert (*JVM-SemiType.sup P mxs maxr*) (*JVM-SemiType.le P mxs maxr*) *Err* (*OK None*) (*exec P mxs* T_r *et bs*) *0*

definition *wt-lbv* :: *'addr jvm-prog* ⇒ *cname* ⇒ *ty list* ⇒ *ty* ⇒ *nat* ⇒ *nat* ⇒
ex-table ⇒ *ty*$_i$' *err list* ⇒ *'addr instr list* ⇒ *bool*
where
wt-lbv P C Ts T_r *mxs* mxl_0 *et cert ins* ≡
check-cert P mxs (*1*+*size Ts*+mxl_0) (*size ins*) *cert* ∧
0 < *size ins* ∧
(*let start* = *Some* ([],(*OK* (*Class C*))#((*map OK Ts*))@(*replicate* mxl_0 *Err*));
result = *lbvjvm P mxs* (*1*+*size Ts*+mxl_0) T_r *et cert ins* (*OK start*)
in result ≠ *Err*)

definition *wt-jvm-prog-lbv* :: *'addr jvm-prog* ⇒ *prog-cert* ⇒ *bool*
where
wt-jvm-prog-lbv P cert ≡
wf-prog (λ*P C* (*mn*,*Ts*,T_r,(*mxs*,mxl_0,*b*,*et*)). *wt-lbv P C Ts* T_r *mxs* mxl_0 *et* (*cert C mn*) *b*) *P*

definition *mk-cert* :: *'addr jvm-prog* ⇒ *nat* ⇒ *ty* ⇒ *ex-table* ⇒ *'addr instr list*
⇒ *ty*$_m$ ⇒ *ty*$_i$' *err list*
where
mk-cert P mxs T_r *et bs phi* ≡ *make-cert* (*exec P mxs* T_r *et bs*) (*map OK phi*) (*OK None*)

definition *prg-cert* :: *'addr jvm-prog* ⇒ *ty*$_P$ ⇒ *prog-cert*
where
prg-cert P phi C mn ≡ *let* (*C*,*Ts*,T_r,*meth*) = *method P C mn*; (*mxs*,mxl_0,*ins*,*et*) = *the meth*
in mk-cert P mxs T_r *et ins* (*phi C mn*)

lemma *check-certD* [*intro?*]:
check-cert P mxs mxl n cert ⟹ *cert-ok cert n Err* (*OK None*) (*states P mxs mxl*)
by (*unfold cert-ok-def check-cert-def check-types-def*) *auto*

lemma (**in** *start-context*) *wt-lbv-wt-step*:
assumes *lbv*: *wt-lbv P C Ts* T_r *mxs* mxl_0 *xt cert is*
shows ∃ τ*s* ∈ *list* (*size is*) *A*. *wt-step r Err step* τ*s* ∧ *OK first* ⊑$_r$ τ*s*!*0*

lemma (**in** *start-context*) *wt-lbv-wt-method*:
assumes *lbv*: *wt-lbv P C Ts* T_r *mxs* mxl_0 *xt cert is*
shows ∃ τ*s*. *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* τ*s*

lemma (**in** *start-context*) *wt-method-wt-lbv*:
assumes *wt*: *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* τ*s*
defines [*simp*]: *cert* ≡ *mk-cert P mxs* T_r *xt is* τ*s*

shows *wt-lbv P C Ts* T_r *mxs* mxl_0 *xt cert is*

theorem *jvm-lbv-correct*:
wt-jvm-prog-lbv P Cert $\Longrightarrow$ *wf-jvm-prog P*
theorem *jvm-lbv-complete*:
assumes *wt*: *wf-jvm-prog*$_\Phi$ *P*
shows *wt-jvm-prog-lbv P* (*prg-cert P* Φ)
end

6.12 Kildall for the JVM

theory *BVExec*
imports
../DFA/Abstract-BV
TF-JVM
begin

definition *kiljvm* :: *'addr jvm-prog* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *ty* $\Rightarrow$
'addr instr list $\Rightarrow$ *ex-table* $\Rightarrow$ *ty*$_i$*' err list* $\Rightarrow$ *ty*$_i$*' err list*
where
kiljvm P mxs mxl T_r *is xt* $\equiv$
kildall (*JVM-SemiType.le P mxs mxl*) (*JVM-SemiType.sup P mxs mxl*)
(*exec P mxs* T_r *xt is*)

definition *wt-kildall* :: *'addr jvm-prog* $\Rightarrow$ *cname* $\Rightarrow$ *ty list* $\Rightarrow$ *ty* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$
'addr instr list $\Rightarrow$ *ex-table* $\Rightarrow$ *bool*
where
wt-kildall P C' Ts T_r *mxs* mxl_0 *is xt* $\equiv$
0 $<$ *size is* $\wedge$
(*let first* $=$ *Some* ([],[*OK* (*Class C'*)]@(*map OK Ts*)@(*replicate* mxl_0 *Err*));
start $=$ *OK first*#(*replicate* (*size is* $-$ *1*) (*OK None*));
result $=$ *kiljvm P mxs* (*1*+*size Ts*+mxl_0) T_r *is xt start*
in $\forall$ *n* $<$ *size is*. *result*!*n* $\neq$ *Err*)

definition *wf-jvm-prog*$_k$ :: *'addr jvm-prog* $\Rightarrow$ *bool*
where
wf-jvm-prog$_k$ *P* $\equiv$
wf-prog (λ*P C'* (*M*,*Ts*,T_r,(*mxs*,mxl_0,*is*,*xt*)). *wt-kildall P C' Ts* T_r *mxs* mxl_0 *is xt*) *P*

theorem (**in** *start-context*) *is-bcv-kiljvm*:
is-bcv r Err step (*size is*) *A* (*kiljvm P mxs mxl* T_r *is xt*)

lemma *subset-replicate* [*intro?*]: *set* (*replicate n x*) $\subseteq$ {*x*}
by (*induct n*) *auto*

lemma *in-set-replicate*:
assumes *x* $\in$ *set* (*replicate n y*)
shows *x* $=$ *y*
lemma (**in** *start-context*) *start-in-A* [*intro?*]:
0 $<$ *size is* $\Longrightarrow$ *start* $\in$ *list* (*size is*) *A*
using *Ts C*

theorem (**in** *start-context*) *wt-kil-correct*:

assumes *wtk*: *wt-kildall P C Ts* T_r *mxs* mxl_0 *is xt*
shows $\exists \tau s.$ *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* τs

theorem (**in** *start-context*) *wt-kil-complete*:
assumes *wtm*: *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* τs
shows *wt-kildall P C Ts* T_r *mxs* mxl_0 *is xt*

theorem *jvm-kildall-correct*:
$wf\text{-}jvm\text{-}prog_k$ *P* = *wf-jvm-prog P*
end

6.13 Code generation for the byte code verifier

theory *BCVExec*
imports
BVNoTypeError
BVExec
begin

lemmas [*code-unfold*] = *exec-lub-def*

lemmas [*code*] = *JVM-le-unfold*[*THEN meta-eq-to-obj-eq*]

lemma *err-code* [*code*]:
Err.err A = *Collect* (*case-err True* ($\lambda x.$ $x \in A$))
by(*auto simp add*: *err-def split*: *err.split*)

lemma *list-code* [*code*]:
list n A = {*xs*. *size xs* = *n* ∧ *list-all* ($\lambda x.$ $x \in A$) *xs*}
unfolding *list-def*
by(*auto intro*!: *ext simp add*: *list-all-iff*)

lemma *opt-code* [*code*]:
opt A = *Collect* (*case-option True* ($\lambda x.$ $x \in A$))
by(*auto simp add*: *opt-def split*: *option.split*)

lemma *Times-code* [*code-unfold*]:
Sigma A (%-. *B*) = {(*a*, *b*). $a \in A \wedge b \in B$}
by *auto*

lemma *upto-esl-code* [*code*]:
upto-esl m (*A*, *r*, *f*) = (*Union* (($\lambda n.$ *list n A*) ‘ {..*m*}), *Listn.le r*, *Listn.sup f*)
by(*auto simp add*: *upto-esl-def*)

lemmas [*code*] = *lesub-def plussub-def*

lemma *JVM-sup-unfold* [*code*]:
JVM-SemiType.sup S m n =
lift2 (*Opt.sup* (*Product.sup* (*Listn.sup* (*SemiType.sup S*)) ($\lambda x\ y.$ *OK* (*map2* (*lift2* (*SemiType.sup S*)) *x y*))))
unfolding *JVM-SemiType.sup-def JVM-SemiType.sl-def Opt.esl-def Err.sl-def*
stk-esl-def loc-sl-def Product.esl-def Listn.sl-def upto-esl-def
SemiType.esl-def Err.esl-def

by *simp*

declare *sup-fun-def* [*code*]

lemma [*code*]: *states P mxs mxl* = *fst*(*sl P mxs mxl*)
unfolding *states-def* **..**

lemma *check-types-code* [*code*]:
check-types P mxs mxl τs = (*list-all* (λ*x*. *x* ∈ (*states P mxs mxl*)) *τs*)
unfolding *check-types-def* **by**(*auto simp add*: *list-all-iff*)

lemma *wf-jvm-prog-code* [*code-unfold*]:
wf-jvm-prog = *wf-jvm-prog*$_k$
by(*simp add*: *fun-eq-iff jvm-kildall-correct*)

definition *wf-jvm-prog'* = *wf-jvm-prog*

ML-val ‹@{*code wf-jvm-prog'*}›

end
theory *BV-Main*
imports
JVMDeadlocked
LBVJVM
BCVExec
begin

end

600

Chapter 7

Compilation

7.1 Method calls in expressions

theory *CallExpr* **imports**
../J/Expr
begin

fun *inline-call* :: ('a,'b,'addr) *exp* ⇒ ('a,'b,'addr) *exp* ⇒ ('a,'b,'addr) *exp*
and *inline-calls* :: ('a,'b,'addr) *exp* ⇒ ('a,'b,'addr) *exp list* ⇒ ('a,'b,'addr) *exp list*
where
inline-call f (*new C*) = *new C*
| *inline-call f* (*newA T*⌊*e*⌉) = *newA T*⌊*inline-call f e*⌉
| *inline-call f* (*Cast C e*) = *Cast C* (*inline-call f e*)
| *inline-call f* (*e instanceof T*) = (*inline-call f e*) *instanceof T*
| *inline-call f* (*Val v*) = *Val v*
| *inline-call f* (*Var V*) = *Var V*
| *inline-call f* (*V*:=*e*) = *V* := *inline-call f e*
| *inline-call f* (*e* «*bop*» *e*′) = (*if is-val e then* (*e* «*bop*» *inline-call f e*′) *else* (*inline-call f e* «*bop*» *e*′))
| *inline-call f* (*a*⌊*i*⌉) = (*if is-val a then a*⌊*inline-call f i*⌉ *else* (*inline-call f a*)⌊*i*⌉)
| *inline-call f* (*AAss a i e*) =
(*if is-val a then if is-val i then AAss a i* (*inline-call f e*) *else AAss a* (*inline-call f i*) *e*
else AAss (*inline-call f a*) *i e*)
| *inline-call f* (*a*·*length*) = *inline-call f a*·*length*
| *inline-call f* (*e*·*F*{*D*}) = *inline-call f e*·*F*{*D*}
| *inline-call f* (*FAss e F D e*′) = (*if is-val e then FAss e F D* (*inline-call f e*′) *else FAss* (*inline-call f e*) *F D e*′)
| *inline-call f* (*CompareAndSwap e D F e*′ *e*′′) =
(*if is-val e then if is-val e*′ *then CompareAndSwap e D F e*′ (*inline-call f e*′′)
else CompareAndSwap e D F (*inline-call f e*′) *e*′′
else CompareAndSwap (*inline-call f e*) *D F e*′ *e*′′)
| *inline-call f* (*e*·*M*(*es*)) =
(*if is-val e then if is-vals es* ∧ *is-addr e then f else e*·*M*(*inline-calls f es*) *else inline-call f e*·*M*(*es*))
| *inline-call f* ({*V*:*T*=*vo*; *e*}) = {*V*:*T*=*vo*; *inline-call f e*}
| *inline-call f* (*sync*$_V$ (*o*′) *e*) = *sync*$_V$ (*inline-call f o*′) *e*
| *inline-call f* (*insync*$_V$ (*a*) *e*) = *insync*$_V$ (*a*) (*inline-call f e*)
| *inline-call f* (*e*;;*e*′) = *inline-call f e*;;*e*′
| *inline-call f* (*if* (*b*) *e else e*′) = (*if* (*inline-call f b*) *e else e*′)
| *inline-call f* (*while* (*b*) *e*) = *while* (*b*) *e*
| *inline-call f* (*throw e*) = *throw* (*inline-call f e*)
| *inline-call f* (*try e1 catch*(*C V*) *e2*) = *try inline-call f e1 catch*(*C V*) *e2*

| *inline-calls f* [] = []
| *inline-calls f* (*e*#*es*) = (*if is-val e then e* # *inline-calls f es else inline-call f e* # *es*)

fun *collapse* :: *'addr expr* × *'addr expr list* ⇒ *'addr expr* **where**
collapse (*e*, []) = *e*
| *collapse* (*e*, (*e'* # *es*)) = *collapse* (*inline-call e e'*, *es*)

definition *is-call* :: (*'a*, *'b*, *'addr*) *exp* ⇒ *bool*
where *is-call e* = (*call e* ≠ *None*)

definition *is-calls* :: (*'a*, *'b*, *'addr*) *exp list* ⇒ *bool*
where *is-calls es* = (*calls es* ≠ *None*)

lemma *inline-calls-map-Val-append* [*simp*]:
inline-calls f (*map Val vs* @ *es*) = *map Val vs* @ *inline-calls f es*
by(*induct vs*, *auto*)

lemma *inline-call-eq-Val-aux*:
inline-call e E = *Val v* ⟹ *call E* = ⌊*aMvs*⌋ ⟹ *e* = *Val v*
by(*induct E*)(*auto split*: *if-split-asm*)

lemmas *inline-call-eq-Val* [*dest*] = *inline-call-eq-Val-aux inline-call-eq-Val-aux*[*OF sym*, *THEN sym*]

lemma *inline-calls-eq-empty* [*simp*]: *inline-calls e es* = [] ⟷ *es* = []
by(*cases es*, *auto*)

lemma *inline-calls-map-Val* [*simp*]: *inline-calls e* (*map Val vs*) = *map Val vs*
by(*induct vs*) *auto*

lemma **fixes** *E* :: (*'a*,*'b*, *'addr*) *exp* **and** *Es* :: (*'a*,*'b*, *'addr*) *exp list*
shows *inline-call-eq-Throw* [*dest*]: *inline-call e E* = *Throw a* ⟹ *call E* = ⌊*aMvs*⌋ ⟹ *e* = *Throw a* ∨ *e* = *addr a*
by(*induct E rule*: *exp.induct*)(*fastforce split*:*if-split-asm*)+

lemma *Throw-eq-inline-call-eq* [*dest*]:
inline-call e E = *Throw a* ⟹ *call E* = ⌊*aMvs*⌋ ⟹ *Throw a* = *e* ∨ *addr a* = *e*
by(*auto dest*: *inline-call-eq-Throw*[*OF sym*])

lemma *is-vals-inline-calls* [*dest*]:
⟦ *is-vals* (*inline-calls e es*); *calls es* = ⌊*aMvs*⌋ ⟧ ⟹ *is-val e*
by(*induct es*, *auto split*: *if-split-asm*)

lemma [*dest*]: ⟦ *inline-calls e es* = *map Val vs*; *calls es* = ⌊*aMvs*⌋ ⟧ ⟹ *is-val e*
⟦ *map Val vs* = *inline-calls e es*; *calls es* = ⌊*aMvs*⌋ ⟧ ⟹ *is-val e*
by(*fastforce intro*!: *is-vals-inline-calls del*: *is-val.intros simp add*: *is-vals-conv elim*: *sym*)+

lemma *inline-calls-eq-Val-Throw* [*dest*]:
⟦ *inline-calls e es* = *map Val vs* @ *Throw a* # *es'*; *calls es* = ⌊*aMvs*⌋ ⟧ ⟹ *e* = *Throw a* ∨ *is-val e*
apply(*induct es arbitrary*: *vs a es'*)
apply(*auto simp add*: *Cons-eq-append-conv split*: *if-split-asm*)
done

lemma *Val-Throw-eq-inline-calls* [*dest*]:
⟦ *map Val vs* @ *Throw a* # *es′* = *inline-calls e es*; *calls es* = ⌊*aMvs*⌋ ⟧ ⟹ *Throw a* = *e* ∨ *is-val e*
by(*auto dest*: *inline-calls-eq-Val-Throw*[*OF sym*])

declare *option.split* [*split del*] *if-split-asm* [*split*] *if-split* [*split del*]

lemma *call-inline-call* [*simp*]:
call e = ⌊*aMvs*⌋ ⟹ *call* (*inline-call* {*v*:*T*=*vo*; *e′*} *e*) = *call e′*
calls es = ⌊*aMvs*⌋ ⟹ *calls* (*inline-calls* {*v*:*T*=*vo*;*e′*} *es*) = *call e′*
apply(*induct e* **and** *es rule*: *call.induct calls.induct*)
apply(*fastforce*)
apply(*fastforce*)
apply(*fastforce*)
apply(*fastforce*)
apply(*fastforce*)
apply(*fastforce split*: *if-split*)
apply(*fastforce*)
apply(*fastforce*)
apply(*fastforce split*: *if-split*)
apply(*clarsimp*)
apply(*fastforce split*: *if-split*)
apply(*fastforce split*: *if-split*)
apply(*fastforce*)
apply(*fastforce*)
apply(*fastforce split*: *if-split*)
apply(*fastforce split*: *if-split*)
apply(*fastforce split*: *if-split*)
apply(*fastforce*)
apply(*fastforce*)
apply(*fastforce*)
apply(*fastforce*)
apply(*fastforce*)
apply(*fastforce*)
apply(*fastforce*)
apply(*fastforce*)
apply(*fastforce*)
apply(*fastforce split*: *if-split*)
done

declare *option.split* [*split*] *if-split* [*split*] *if-split-asm* [*split del*]

lemma *fv-inline-call*: *fv* (*inline-call e′ e*) ⊆ *fv e* ∪ *fv e′*
and *fvs-inline-calls*: *fvs* (*inline-calls e′ es*) ⊆ *fvs es* ∪ *fv e′*
by(*induct e* **and** *es rule*: *call.induct calls.induct*)(*fastforce split*: *if-split-asm*)+

lemma *contains-insync-inline-call-conv*:
contains-insync (*inline-call e e′*) ⟷ *contains-insync e* ∧ *call e′* ≠ *None* ∨ *contains-insync e′*
and *contains-insyncs-inline-calls-conv*:
contains-insyncs (*inline-calls e es′*) ⟷ *contains-insync e* ∧ *calls es′* ≠ *None* ∨ *contains-insyncs es′*
by(*induct e′* **and** *es′ rule*: *call.induct calls.induct*)(*auto split*: *if-split-asm simp add*: *is-vals-conv*)

lemma *contains-insync-inline-call* [*simp*]:
call e′ = ⌊*aMvs*⌋ ⟹ *contains-insync* (*inline-call e e′*) ⟷ *contains-insync e* ∨ *contains-insync e′*

and *contains-insyncs-inline-calls* [*simp*]:
calls es′ = ⌊*aMvs*⌋ ⟹ *contains-insyncs* (*inline-calls e es′*) ⟷ *contains-insync e* ∨ *contains-insyncs es′*
by(*simp-all add*: *contains-insync-inline-call-conv contains-insyncs-inline-calls-conv*)

lemma *collapse-append* [*simp*]:
collapse (*e*, *es* @ *es′*) = *collapse* (*collapse* (*e*, *es*), *es′*)
by(*induct es arbitrary*: *e*, *auto*)

lemma *collapse-conv-foldl*:
collapse (*e*, *es*) = *foldl inline-call e es*
by(*induct es arbitrary*: *e*) *simp-all*

lemma *fv-collapse*: ∀ *e* ∈ *set es*. *is-call e* ⟹ *fv* (*collapse* (*e*, *es*)) ⊆ *fvs* (*e* # *es*)
apply(*induct es arbitrary*: *e*)
apply(*insert fv-inline-call*)
apply(*fastforce dest*: *subsetD*)+
done

lemma *final-inline-callD*: ⟦ *final* (*inline-call E e*); *is-call e* ⟧ ⟹ *final E*
by(*induct e*)(*auto simp add*: *is-call-def split*: *if-split-asm*)

lemma *collapse-finalD*: ⟦ *final* (*collapse* (*e*, *es*)); ∀ *e*∈*set es*. *is-call e* ⟧ ⟹ *final e*
by(*induct es arbitrary*: *e*)(*auto dest*: *final-inline-callD*)

context *heap-base* **begin**

definition *synthesized-call* :: *′m prog* ⇒ *′heap* ⇒ (*′addr* × *mname* × *′addr val list*) ⇒ *bool*
where
synthesized-call P h =
(λ(*a*, *M*, *vs*). ∃ *T Ts Tr D*. *typeof-addr h a* = ⌊*T*⌋ ∧ *P* ⊢ *class-type-of T sees M*:*Ts*→*Tr* = *Native in D*)

lemma *synthesized-call-conv*:
synthesized-call P h (*a*, *M*, *vs*) =
(∃ *T Ts Tr D*. *typeof-addr h a* = ⌊*T*⌋ ∧ *P* ⊢ *class-type-of T sees M*:*Ts*→*Tr* = *Native in D*)
by(*simp add*: *synthesized-call-def*)

end

end

7.2 The JinjaThreads source language with explicit call stacks

theory *J0* **imports**
../J/WWellForm
../J/WellType
../J/Threaded
../Framework/FWBisimulation
CallExpr
begin

declare *widen-refT* [*elim*]

abbreviation *final-expr0* :: *'addr expr* × *'addr expr list* ⇒ *bool* **where**
final-expr0 ≡ λ(*e*, *es*). *final e* ∧ *es* = []

type-synonym
(*'addr*, *'thread-id*, *'heap*) *J0-thread-action* =
(*'addr*, *'thread-id*, *'addr expr* × *'addr expr list*,*'heap*) *Jinja-thread-action*

type-synonym
(*'addr*, *'thread-id*, *'heap*) *J0-state* = (*'addr*,*'thread-id*,*'addr expr* × *'addr expr list*,*'heap*,*'addr*) *state*

print-translation ‹
let
fun tr'
[*a1*, *t*
, *Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax exp*}, -) $
Const (@{*type-syntax String.literal*}, -) $ *Const* (@{*type-syntax unit*}, -) $ *a2*) $
(*Const* (@{*type-syntax list*}, -) $
(*Const* (@{*type-syntax exp*}, -) $
Const (@{*type-syntax String.literal*}, -) $
Const (@{*type-syntax unit*}, -) $ *a3*))
, *h*] =
if a1 = *a2 andalso a2* = *a3 then Syntax.const* @{*type-syntax J0-thread-action*} $ *a1* $ *t* $ *h*
else raise Match;
in [(@{*type-syntax Jinja-thread-action*}, *K tr'*)]
end
›
typ (*'addr*,*'thread-id*,*'heap*) *J0-thread-action*

print-translation ‹
let
fun tr'
[*a1*, *t*
, *Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax exp*}, -) $
Const (@{*type-syntax String.literal*}, -) $ *Const* (@{*type-syntax unit*}, -) $ *a2*) $
(*Const* (@{*type-syntax list*}, -) $
(*Const* (@{*type-syntax exp*}, -) $
Const (@{*type-syntax String.literal*}, -) $
Const (@{*type-syntax unit*}, -) $ *a3*))
, *h*, *a4*] =
if a1 = *a2 andalso a2* = *a3 then Syntax.const* @{*type-syntax J0-state*} $ *a1* $ *t* $ *h*
else raise Match;
in [(@{*type-syntax state*}, *K tr'*)]
end
›
typ (*'addr*, *'thread-id*, *'heap*) *J0-state*

definition *extNTA2J0* :: *'addr J-prog* ⇒ (*cname* × *mname* × *'addr*) ⇒ (*'addr expr* × *'addr expr list*)
where
extNTA2J0 P = (λ(*C*, *M*, *a*). *let* (*D*, -, -, *meth*) = *method P C M*; (-, *body*) = *the meth*

in ({*this*:*Class D*=⌊*Addr a*⌋; *body*}, []))

lemma *extNTA2J0-iff* [*simp*]:
extNTA2J0 P (*C*, *M*, *a*) =
({*this*:*Class* (*fst* (*method P C M*))=⌊*Addr a*⌋; *snd* (*the* (*snd* (*snd* (*snd* (*method P C M*)))))}, [])
by(*simp add*: *extNTA2J0-def split-def*)

abbreviation *extTA2J0* ::
'addr J-prog ⇒ (*'addr*, *'thread-id*, *'heap*) *external-thread-action* ⇒ (*'addr*, *'thread-id*, *'heap*) *J0-thread-action*
where *extTA2J0 P* ≡ *convert-extTA* (*extNTA2J0 P*)

lemma *obs-a-extTA2J-eq-obs-a-extTA2J0* [*simp*]: {|*extTA2J P ta*|}$_o$ = {|*extTA2J0 P ta*|}$_o$
by(*cases ta*)(*simp add*: *ta-upd-simps*)

lemma *extTA2J0-ε*: *extTA2J0 P ε* = *ε*
by(*simp*)

context *J-heap-base* **begin**

definition *no-call* :: *'m prog* ⇒ *'heap* ⇒ (*'a*, *'b*, *'addr*) *exp* ⇒ *bool*
where *no-call P h e* = (∀ *aMvs*. *call e* = ⌊*aMvs*⌋ ⟶ *synthesized-call P h aMvs*)

definition *no-calls* :: *'m prog* ⇒ *'heap* ⇒ (*'a*, *'b*, *'addr*) *exp list* ⇒ *bool*
where *no-calls P h es* = (∀ *aMvs*. *calls es* = ⌊*aMvs*⌋ ⟶ *synthesized-call P h aMvs*)

inductive *red0* ::
'addr J-prog ⇒ *'thread-id* ⇒ *'addr expr* ⇒ *'addr expr list* ⇒ *'heap*
⇒ (*'addr*, *'thread-id*, *'heap*) *J0-thread-action* ⇒ *'addr expr* ⇒ *'addr expr list* ⇒ *'heap* ⇒ *bool*
(‹-,- ⊢*0* ((*1*⟨-'/-,/-⟩) -*-*→/ (*1*⟨-'/-,/-⟩))› [*51*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*] *81*)
for *P* :: *'addr J-prog* **and** *t* :: *'thread-id*
where

red0Red:
⟦ *extTA2J0 P*,*P*,*t* ⊢ ⟨*e*, (*h*, *Map.empty*)⟩ −*ta*→ ⟨*e'*, (*h'*, *xs'*)⟩;
∀ *aMvs*. *call e* = ⌊*aMvs*⌋ ⟶ *synthesized-call P h aMvs* ⟧
⟹ *P*,*t* ⊢*0* ⟨*e*/*es*, *h*⟩ −*ta*→ ⟨*e'*/*es*, *h'*⟩

| *red0Call*:
⟦ *call e* = ⌊(*a*, *M*, *vs*)⌋; *typeof-addr h a* = ⌊*U*⌋;
P ⊢ *class-type-of U sees M*:*Ts*→*T* = ⌊(*pns*, *body*)⌋ *in D*;
size vs = *size pns*; *size Ts* = *size pns* ⟧
⟹ *P*,*t* ⊢*0* ⟨*e*/*es*, *h*⟩ −*ε*→ ⟨*blocks* (*this* # *pns*) (*Class D* # *Ts*) (*Addr a* # *vs*) *body*/*e*#*es*, *h*⟩

| *red0Return*:
final e' ⟹ *P*,*t* ⊢*0* ⟨*e'*/*e*#*es*, *h*⟩ −*ε*→ ⟨*inline-call e' e*/*es*, *h*⟩

abbreviation *J0-start-state* :: *'addr J-prog* ⇒ *cname* ⇒ *mname* ⇒ *'addr val list* ⇒ (*'addr*, *'thread-id*, *'heap*) *J0-state*
where
J0-start-state ≡
start-state (λ*C M Ts T* (*pns*, *body*) *vs*. (*blocks* (*this* # *pns*) (*Class C* # *Ts*) (*Null* # *vs*) *body*, []))

abbreviation *mred0* ::
'addr J-prog ⇒ (*'addr*,*'thread-id*,*'addr expr* × *'addr expr list*,*'heap*,*'addr*,(*'addr*, *'thread-id*) *obs-event*)

semantics
where *mred0 P* ≡ (λ*t* ((*e*, *es*), *h*) *ta* ((*e′*, *es′*), *h′*). *red0 P t e es h ta e′ es′ h′*)

end

declare *domIff*[*iff*, *simp del*]

context *J-heap-base* **begin**

lemma assumes *wf*: *wwf-J-prog P*
shows *red-fv-subset*: *extTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩ ⟹ *fv e′* ⊆ *fv e*
and *reds-fvs-subset*: *extTA*,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es′*, *s′*⟩ ⟹ *fvs es′* ⊆ *fvs es*
proof(*induct rule*: *red-reds.inducts*)
case (*RedCall s a U M Ts T pns body D vs*)
hence *fv body* ⊆ {*this*} ∪ *set pns*
using *wf* **by**(*fastforce dest*!:*sees-wf-mdecl simp*:*wf-mdecl-def*)
with *RedCall* **show** *?case* **by** *fastforce*
next
case *RedCallExternal* **thus** *?case* **by**(*auto simp add*: *extRet2J-def split*: *extCallRet.split-asm*)
qed(*fastforce*)+

end

declare *domIff*[*iff del*]

context *J-heap-base* **begin**

lemma assumes *wwf*: *wwf-J-prog P*
shows *red-fv-ok*: ⟦ *extTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩; *fv e* ⊆ *dom* (*lcl s*) ⟧ ⟹ *fv e′* ⊆ *dom* (*lcl s′*)
and *reds-fvs-ok*: ⟦ *extTA*,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es′*, *s′*⟩; *fvs es* ⊆ *dom* (*lcl s*) ⟧ ⟹ *fvs es′* ⊆ *dom* (*lcl s′*)
proof(*induct rule*: *red-reds.inducts*)
case (*RedCall s a U M Ts T pns body D vs*)
from ‹*P* ⊢ *class-type-of U sees M*: *Ts*→*T* = ⌊(*pns*, *body*)⌋ *in D*› **have** *wwf-J-mdecl P D* (*M*,*Ts*,*T*,*pns*,*body*)
by(*auto dest*!: *sees-wf-mdecl*[*OF wwf*] *simp add*: *wf-mdecl-def*)
with *RedCall* **show** *?case* **by**(*auto*)
next
case *RedCallExternal* **thus** *?case* **by**(*auto simp add*: *extRet2J-def split*: *extCallRet.split-asm*)
next
case (*BlockRed e h x V vo ta e′ h′ x′ T*)
note *red* = ‹*extTA*,*P*,*t* ⊢ ⟨*e*,(*h*, *x*(*V* := *vo*))⟩ −*ta*→ ⟨*e′*,(*h′*, *x′*)⟩›
hence *fv e′* ⊆ *fv e* **by**(*auto dest*: *red-fv-subset*[*OF wwf*] *del*: *subsetI*)
moreover from ‹ *fv* {*V*:*T*=*vo*; *e*} ⊆ *dom* (*lcl* (*h*, *x*))›
have *fv e* − {*V*} ⊆ *dom x* **by**(*simp*)
ultimately have *fv e′* − {*V*} ⊆ *dom x* − {*V*} **by**(*auto*)
moreover from *red* **have** *dom* (*x*(*V* := *vo*)) ⊆ *dom x′*
by(*auto dest*: *red-lcl-incr del*: *subsetI*)
ultimately have *fv e′* − {*V*} ⊆ *dom x′* − {*V*}
by(*auto split*: *if-split-asm*)
thus *?case* **by**(*auto simp del*: *fun-upd-apply*)
qed(*fastforce dest*: *red-lcl-incr del*: *subsetI*)+

lemma *is-call-red-state-unchanged*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩; *call e* = ⌊*aMvs*⌋; ¬ *synthesized-call P* (*hp s*) *aMvs* ⟧ ⟹ *s′* = *s*

$\wedge$ *ta* = ε

and *is-calls-reds-state-unchanged*:
⟦ *extTA,P,t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *calls es* = ⌊*aMvs*⌋; ¬ *synthesized-call P* (*hp s*) *aMvs* ⟧ ⟹ *s'* = *s* ∧ *ta* = ε
apply(*induct rule*: *red-reds.inducts*)
apply(*fastforce split*: *if-split-asm simp add*: *synthesized-call-def*)+
done

lemma *called-methodD*:
⟦ *extTA,P,t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; *call e* = ⌊(*a*, *M*, *vs*)⌋; ¬ *synthesized-call P* (*hp s*) (*a*, *M*, *vs*) ⟧
⟹ ∃ *hT D Us U pns body*. *hp s'* = *hp s* ∧ *typeof-addr* (*hp s*) *a* = ⌊*hT*⌋ ∧
P ⊢ *class-type-of hT sees M*: *Us*→*U* = ⌊(*pns*, *body*)⌋ *in D* ∧
length vs = *length pns* ∧ *length Us* = *length pns*

and *called-methodsD*:
⟦ *extTA,P,t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *calls es* = ⌊(*a*, *M*, *vs*)⌋; ¬ *synthesized-call P* (*hp s*) (*a*, *M*, *vs*) ⟧
⟹ ∃ *hT D Us U pns body*. *hp s'* = *hp s* ∧ *typeof-addr* (*hp s*) *a* = ⌊*hT*⌋ ∧
P ⊢ *class-type-of hT sees M*: *Us*→*U* = ⌊(*pns*, *body*)⌋ *in D* ∧
length vs = *length pns* ∧ *length Us* = *length pns*
apply(*induct rule*: *red-reds.inducts*)
apply(*auto split*: *if-split-asm simp add*: *synthesized-call-def*)
apply(*fastforce*)
done

7.2.1 Silent moves

primrec τ*move0* :: *'m prog* ⇒ *'heap* ⇒ (*'a*, *'b*, *'addr*) *exp* ⇒ *bool*
and τ*moves0* :: *'m prog* ⇒ *'heap* ⇒ (*'a*, *'b*, *'addr*) *exp list* ⇒ *bool*
where
τ*move0 P h* (*new C*) ⟷ *False*
| τ*move0 P h* (*newA T*⌊*e*⌉) ⟷ τ*move0 P h e* ∨ (∃ *a*. *e* = *Throw a*)
| τ*move0 P h* (*Cast U e*) ⟷ τ*move0 P h e* ∨ (∃ *a*. *e* = *Throw a*) ∨ (∃ *v*. *e* = *Val v*)
| τ*move0 P h* (*e instanceof T*) ⟷ τ*move0 P h e* ∨ (∃ *a*. *e* = *Throw a*) ∨ (∃ *v*. *e* = *Val v*)
| τ*move0 P h* (*e* «*bop*» *e'*) ⟷ τ*move0 P h e* ∨ (∃ *a*. *e* = *Throw a*) ∨ (∃ *v*. *e* = *Val v* ∧
(τ*move0 P h e'* ∨ (∃ *a*. *e'* = *Throw a*) ∨ (∃ *v*. *e'* = *Val v*)))
| τ*move0 P h* (*Val v*) ⟷ *False*
| τ*move0 P h* (*Var V*) ⟷ *True*
| τ*move0 P h* (*V* := *e*) ⟷ τ*move0 P h e* ∨ (∃ *a*. *e* = *Throw a*) ∨ (∃ *v*. *e* = *Val v*)
| τ*move0 P h* (*a*⌊*i*⌉) ⟷ τ*move0 P h a* ∨ (∃ *ad*. *a* = *Throw ad*) ∨ (∃ *v*. *a* = *Val v* ∧ (τ*move0 P h i* ∨ (∃ *a*. *i* = *Throw a*)))
| τ*move0 P h* (*AAss a i e*) ⟷ τ*move0 P h a* ∨ (∃ *ad*. *a* = *Throw ad*) ∨ (∃ *v*. *a* = *Val v* ∧
(τ*move0 P h i* ∨ (∃ *a*. *i* = *Throw a*) ∨ (∃ *v*. *i* = *Val v* ∧ (τ*move0 P h e* ∨ (∃ *a*. *e* = *Throw a*)))))
| τ*move0 P h* (*a*·*length*) ⟷ τ*move0 P h a* ∨ (∃ *ad*. *a* = *Throw ad*)
| τ*move0 P h* (*e*·*F*{*D*}) ⟷ τ*move0 P h e* ∨ (∃ *a*. *e* = *Throw a*)
| τ*move0 P h* (*FAss e F D e'*) ⟷ τ*move0 P h e* ∨ (∃ *a*. *e* = *Throw a*) ∨ (∃ *v*. *e* = *Val v* ∧ (τ*move0 P h e'* ∨ (∃ *a*. *e'* = *Throw a*)))
| τ*move0 P h* (*e*·*compareAndSwap*(*D*·*F*, *e'*, *e''*)) ⟷ τ*move0 P h e* ∨ (∃ *a*. *e* = *Throw a*) ∨ (∃ *v*. *e* = *Val v* ∧
(τ*move0 P h e'* ∨ (∃ *a*. *e'* = *Throw a*) ∨ (∃ *v*. *e'* = *Val v* ∧ (τ*move0 P h e''* ∨ (∃ *a*. *e''* = *Throw a*)))))
| τ*move0 P h* (*e*·*M*(*es*)) ⟷ τ*move0 P h e* ∨ (∃ *a*. *e* = *Throw a*) ∨ (∃ *v*. *e* = *Val v* ∧
((τ*moves0 P h es* ∨ (∃ *vs a es'*. *es* = *map Val vs* @ *Throw a* # *es'*)) ∨

(∃ *vs*. *es* = *map Val vs* ∧ (*v* = *Null* ∨ (∀ *T C Ts Tr D*. *typeof*$_h$ *v* = ⌊*T*⌋ ⟶ *class-type-of*′ *T* = ⌊*C*⌋ ⟶ *P* ⊢ *C sees M*:*Ts*→*Tr* = *Native in D* ⟶ τ*external-defs D M*)))))
| τ*move0 P h* ({*V*:*T*=*vo*; *e*}) ⟷ τ*move0 P h e* ∨ ((∃ *a*. *e* = *Throw a*) ∨ (∃ *v*. *e* = *Val v*))
| τ*move0 P h* (*sync*$_{V'}$(*e*) *e*′) ⟷ τ*move0 P h e* ∨ (∃ *a*. *e* = *Throw a*)
| τ*move0 P h* (*insync*$_{V'}$(*ad*) *e*) ⟷ τ*move0 P h e*
| τ*move0 P h* (*e*;;*e*′) ⟷ τ*move0 P h e* ∨ (∃ *a*. *e* = *Throw a*) ∨ (∃ *v*. *e* = *Val v*)
| τ*move0 P h* (*if* (*e*) *e*′ *else e*″) ⟷ τ*move0 P h e* ∨ (∃ *a*. *e* = *Throw a*) ∨ (∃ *v*. *e* = *Val v*)
| τ*move0 P h* (*while* (*e*) *e*′) = *True*
| — *Throw a* is no τ*move0* because there is no reduction for it. If it were, most defining equations would be simpler. However, *insync*$_{V'}$ (*ad*) *Throw ad* must not be a τ*move0*, but would be if *Throw a* was.
τ*move0 P h* (*throw e*) ⟷ τ*move0 P h e* ∨ (∃ *a*. *e* = *Throw a*) ∨ *e* = *null*
| τ*move0 P h* (*try e catch*(*C V*) *e*′) ⟷ τ*move0 P h e* ∨ (∃ *a*. *e* = *Throw a*) ∨ (∃ *v*. *e* = *Val v*)

| τ*moves0 P h* [] ⟷ *False*
| τ*moves0 P h* (*e* # *es*) ⟷ τ*move0 P h e* ∨ (∃ *v*. *e* = *Val v* ∧ τ*moves0 P h es*)

abbreviation τ*MOVE* :: *'m prog* ⇒ ((*'addr expr* × *'addr locals*) × *'heap*, (*'addr*, *'thread-id*, *'heap*) *J-thread-action*) *trsys*
where τ*MOVE* ≡ λ*P* ((*e*, *x*), *h*) *ta s*′. τ*move0 P h e* ∧ *ta* = ε

primrec τ*Move0* :: *'m prog* ⇒ *'heap* ⇒ (*'addr expr* × *'addr expr list*) ⇒ *bool*
where
τ*Move0 P h* (*e*, *exs*) = (τ*move0 P h e* ∨ *final e*)

abbreviation τ*MOVE0* :: *'m prog* ⇒ ((*'addr expr* × *'addr expr list*) × *'heap*, (*'addr*, *'thread-id*, *'heap*) *J0-thread-action*) *trsys*
where τ*MOVE0* ≡ λ*P* (*es*, *h*) *ta s*. τ*Move0 P h es* ∧ *ta* = ε

definition τ*red0* ::
(*'addr*, *'thread-id*, *'heap*) *external-thread-action* ⇒ (*'addr*, *'thread-id*, *'x*,*'heap*) *Jinja-thread-action*)
⇒ *'addr J-prog* ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr* × *'addr locals*) ⇒ (*'addr expr* × *'addr locals*) ⇒ *bool*
where
τ*red0 extTA P t h exs e*′*xs*′ =
(*extTA*,*P*,*t* ⊢ ⟨*fst exs*, (*h*, *snd exs*)⟩ −ε→ ⟨*fst e*′*xs*′, (*h*, *snd e*′*xs*′)⟩ ∧ τ*move0 P h* (*fst exs*) ∧ *no-call P h* (*fst exs*))

definition τ*reds0* ::
(*'addr*, *'thread-id*, *'heap*) *external-thread-action* ⇒ (*'addr*, *'thread-id*, *'x*,*'heap*) *Jinja-thread-action*)
⇒ *'addr J-prog* ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr list* × *'addr locals*) ⇒ (*'addr expr list* × *'addr locals*) ⇒ *bool*
where
τ*reds0 extTA P t h esxs es*′*xs*′ =
(*extTA*,*P*,*t* ⊢ ⟨*fst esxs*, (*h*, *snd esxs*)⟩ [−ε→] ⟨*fst es*′*xs*′, (*h*, *snd es*′*xs*′)⟩ ∧ τ*moves0 P h* (*fst esxs*) ∧ *no-calls P h* (*fst esxs*))

abbreviation τ*red0t* ::
(*'addr*, *'thread-id*, *'heap*) *external-thread-action* ⇒ (*'addr*, *'thread-id*, *'x*,*'heap*) *Jinja-thread-action*)
⇒ *'addr J-prog* ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr* × *'addr locals*) ⇒ (*'addr expr* × *'addr locals*) ⇒ *bool*
where τ*red0t extTA P t h* ≡ (τ*red0 extTA P t h*)^++

abbreviation τ*reds0t* ::

((*'addr*, *'thread-id*, *'heap*) *external-thread-action* ⇒ (*'addr*, *'thread-id*, *'x*,*'heap*) *Jinja-thread-action*)
⇒ *'addr J-prog* ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr list* × *'addr locals*) ⇒ (*'addr expr list* × *'addr locals*) ⇒ *bool*
where *τreds0t extTA P t h* ≡ (*τreds0 extTA P t h*)^++

abbreviation *τred0r* ::
((*'addr*, *'thread-id*, *'heap*) *external-thread-action* ⇒ (*'addr*, *'thread-id*, *'x*,*'heap*) *Jinja-thread-action*)
⇒ *'addr J-prog* ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr* × *'addr locals*) ⇒ (*'addr expr* × *'addr locals*) ⇒ *bool*
where *τred0r extTA P t h* ≡ (*τred0 extTA P t h*)^**

abbreviation *τreds0r* ::
((*'addr*, *'thread-id*, *'heap*) *external-thread-action* ⇒ (*'addr*, *'thread-id*, *'x*,*'heap*) *Jinja-thread-action*)
⇒ *'addr J-prog* ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr list* × *'addr locals*) ⇒ (*'addr expr list* × *'addr locals*) ⇒ *bool*
where *τreds0r extTA P t h* ≡ (*τreds0 extTA P t h*)^**

definition *τRed0* ::
'addr J-prog ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr* × *'addr expr list*) ⇒ (*'addr expr* × *'addr expr list*) ⇒ *bool*
where *τRed0 P t h ees e'es'* = (*P*,*t* ⊢*0* ⟨*fst ees*/*snd ees*, *h*⟩ −*ε*→ ⟨*fst e'es'*/*snd e'es'*, *h*⟩ ∧ *τMove0 P h ees*)

abbreviation *τRed0r* ::
'addr J-prog ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr* × *'addr expr list*) ⇒ (*'addr expr* × *'addr expr list*) ⇒ *bool*
where *τRed0r P t h* ≡ (*τRed0 P t h*)^**

abbreviation *τRed0t* ::
'addr J-prog ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr* × *'addr expr list*) ⇒ (*'addr expr* × *'addr expr list*) ⇒ *bool*
where *τRed0t P t h* ≡ (*τRed0 P t h*)^++

lemma *τmove0-τmoves0-intros*:
fixes *e e1 e2 e'* :: (*'a*, *'b*, *'addr*) *exp* **and** *es* :: (*'a*, *'b*, *'addr*) *exp list*
shows *τmove0NewArray*: *τmove0 P h e* ⟹ *τmove0 P h* (*newA T*⌊*e*⌉)
and *τmove0Cast*: *τmove0 P h e* ⟹ *τmove0 P h* (*Cast U e*)
and *τmove0CastRed*: *τmove0 P h* (*Cast U* (*Val v*))
and *τmove0InstanceOf*: *τmove0 P h e* ⟹ *τmove0 P h* (*e instanceof T*)
and *τmove0InstanceOfRed*: *τmove0 P h* ((*Val v*) *instanceof T*)
and *τmove0BinOp1*: *τmove0 P h e* ⟹ *τmove0 P h* (*e*«*bop*»*e'*)
and *τmove0BinOp2*: *τmove0 P h e* ⟹ *τmove0 P h* (*Val v*«*bop*»*e*)
and *τmove0BinOp*: *τmove0 P h* (*Val v*«*bop*»*Val v'*)
and *τmove0Var*: *τmove0 P h* (*Var V*)
and *τmove0LAss*: *τmove0 P h e* ⟹ *τmove0 P h* (*V* := *e*)
and *τmove0LAssRed*: *τmove0 P h* (*V* := *Val v*)
and *τmove0AAcc1*: *τmove0 P h e* ⟹ *τmove0 P h* (*e*⌊*e'*⌉)
and *τmove0AAcc2*: *τmove0 P h e* ⟹ *τmove0 P h* (*Val v*⌊*e*⌉)
and *τmove0AAss1*: *τmove0 P h e* ⟹ *τmove0 P h* (*e*⌊*e1*⌉ := *e2*)
and *τmove0AAss2*: *τmove0 P h e* ⟹ *τmove0 P h* (*Val v*⌊*e*⌉ := *e'*)
and *τmove0AAss3*: *τmove0 P h e* ⟹ *τmove0 P h* (*Val v*⌊*Val v'*⌉ := *e*)
and *τmove0ALength*: *τmove0 P h e* ⟹ *τmove0 P h* (*e*•*length*)
and *τmove0FAcc*: *τmove0 P h e* ⟹ *τmove0 P h* (*e*•*F*{*D*})
and *τmove0FAss1*: *τmove0 P h e* ⟹ *τmove0 P h* (*FAss e F D e'*)

and $\tau move0FAss2$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (Val\ v{\cdot}F\{D\} := e)$
and $\tau move0CAS1$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e''))$
and $\tau move0CAS2$: $\tau move0\ P\ h\ e' \Longrightarrow \tau move0\ P\ h\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e''))$
and $\tau move0CAS3$: $\tau move0\ P\ h\ e'' \Longrightarrow \tau move0\ P\ h\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v',\ e''))$
and $\tau move0CallObj$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (e{\cdot}M(es))$
and $\tau move0CallParams$: $\tau moves0\ P\ h\ es \Longrightarrow \tau move0\ P\ h\ (Val\ v{\cdot}M(es))$
and $\tau move0Call$: $(\bigwedge T\ C\ Ts\ Tr\ D.\ [\![\ typeof_h\ v = \lfloor T \rfloor;\ class\text{-}type\text{-}of'\ T = \lfloor C \rfloor;\ P \vdash C\ sees\ M{:}Ts{\rightarrow}Tr = Native\ in\ D\ ]\!] \Longrightarrow \tau external\text{-}defs\ D\ M) \Longrightarrow \tau move0\ P\ h\ (Val\ v{\cdot}M(map\ Val\ vs))$
and $\tau move0Block$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ \{V{:}T{=}vo;\ e\}$
and $\tau move0BlockRed$: $\tau move0\ P\ h\ \{V{:}T{=}vo;\ Val\ v\}$
and $\tau move0Sync$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (sync_{V'}\ (e)\ e')$
and $\tau move0InSync$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (insync_{V'}\ (a)\ e)$
and $\tau move0Seq$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (e;;e')$
and $\tau move0SeqRed$: $\tau move0\ P\ h\ (Val\ v;;\ e')$
and $\tau move0Cond$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (if\ (e)\ e1\ else\ e2)$
and $\tau move0CondRed$: $\tau move0\ P\ h\ (if\ (Val\ v)\ e1\ else\ e2)$
and $\tau move0WhileRed$: $\tau move0\ P\ h\ (while\ (e)\ e')$
and $\tau move0Throw$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (throw\ e)$
and $\tau move0ThrowNull$: $\tau move0\ P\ h\ (throw\ null)$
and $\tau move0Try$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (try\ e\ catch(C\ V)\ e')$
and $\tau move0TryRed$: $\tau move0\ P\ h\ (try\ Val\ v\ catch(C\ V)\ e)$
and $\tau move0TryThrow$: $\tau move0\ P\ h\ (try\ Throw\ a\ catch(C\ V)\ e)$
and $\tau move0NewArrayThrow$: $\tau move0\ P\ h\ (newA\ T\lfloor Throw\ a \rceil)$
and $\tau move0CastThrow$: $\tau move0\ P\ h\ (Cast\ T\ (Throw\ a))$
and $\tau move0CInstanceOfThrow$: $\tau move0\ P\ h\ ((Throw\ a)\ instanceof\ T)$
and $\tau move0BinOpThrow1$: $\tau move0\ P\ h\ (Throw\ a\ «bop»\ e')$
and $\tau move0BinOpThrow2$: $\tau move0\ P\ h\ (Val\ v\ «bop»\ Throw\ a)$
and $\tau move0LAssThrow$: $\tau move0\ P\ h\ (V{:=}(Throw\ a))$
and $\tau move0AAccThrow1$: $\tau move0\ P\ h\ (Throw\ a\lfloor e \rceil)$
and $\tau move0AAccThrow2$: $\tau move0\ P\ h\ (Val\ v\lfloor Throw\ a \rceil)$
and $\tau move0AAssThrow1$: $\tau move0\ P\ h\ (AAss\ (Throw\ a)\ e\ e')$
and $\tau move0AAssThrow2$: $\tau move0\ P\ h\ (AAss\ (Val\ v)\ (Throw\ a)\ e')$
and $\tau move0AAssThrow3$: $\tau move0\ P\ h\ (AAss\ (Val\ v)\ (Val\ v')\ (Throw\ a))$
and $\tau move0ALengthThrow$: $\tau move0\ P\ h\ (Throw\ a{\cdot}length)$
and $\tau move0FAccThrow$: $\tau move0\ P\ h\ (Throw\ a{\cdot}F\{D\})$
and $\tau move0FAssThrow1$: $\tau move0\ P\ h\ (Throw\ a{\cdot}F\{D\} := e)$
and $\tau move0FAssThrow2$: $\tau move0\ P\ h\ (FAss\ (Val\ v)\ F\ D\ (Throw\ a))$
and $\tau move0CallThrowObj$: $\tau move0\ P\ h\ (Throw\ a{\cdot}M(es))$
and $\tau move0CallThrowParams$: $\tau move0\ P\ h\ (Val\ v{\cdot}M(map\ Val\ vs\ @\ Throw\ a\ \#\ es))$
and $\tau move0BlockThrow$: $\tau move0\ P\ h\ \{V{:}T{=}vo;\ Throw\ a\}$
and $\tau move0SyncThrow$: $\tau move0\ P\ h\ (sync_{V'}\ (Throw\ a)\ e)$
and $\tau move0SeqThrow$: $\tau move0\ P\ h\ (Throw\ a;;e)$
and $\tau move0CondThrow$: $\tau move0\ P\ h\ (if\ (Throw\ a)\ e1\ else\ e2)$
and $\tau move0ThrowThrow$: $\tau move0\ P\ h\ (throw\ (Throw\ a))$

and $\tau moves0Hd$: $\tau move0\ P\ h\ e \Longrightarrow \tau moves0\ P\ h\ (e\ \#\ es)$
and $\tau moves0Tl$: $\tau moves0\ P\ h\ es \Longrightarrow \tau moves0\ P\ h\ (Val\ v\ \#\ es)$
by *auto*

lemma $\tau moves0\text{-}map\text{-}Val$ [*iff*]:
$\neg\ \tau moves0\ P\ h\ (map\ Val\ vs)$
by(*induct vs*) *auto*

lemma $\tau moves0\text{-}map\text{-}Val\text{-}append$ [*simp*]:

$\tau moves0\ P\ h\ (map\ Val\ vs\ @\ es) = \tau moves0\ P\ h\ es$
by(*induct vs*)(*auto*)

lemma *no-reds-map-Val-Throw* [*simp*]:
$extTA,P,t \vdash \langle map\ Val\ vs\ @\ Throw\ a\ \#\ es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle = False$
by(*induct vs arbitrary*: *es'*)(*auto elim*: *reds.cases*)

lemma assumes [*simp*]: $extTA\ \varepsilon = \varepsilon$
shows *red-τ-taD*: $[\![\ extTA,P,t \vdash \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle;\ \tau move0\ P\ (hp\ s)\ e\ ]\!] \Longrightarrow ta = \varepsilon$
and *reds-τ-taD*: $[\![\ extTA,P,t \vdash \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle;\ \tau moves0\ P\ (hp\ s)\ es\ ]\!] \Longrightarrow ta = \varepsilon$
apply(*induct rule*: *red-reds.inducts*)
apply(*fastforce simp add*: *map-eq-append-conv τexternal'-def τexternal-def dest*: *τexternal'-red-external-TA-empty*
done

lemma *τmove0-heap-unchanged*: $[\![\ extTA,P,t \vdash \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle;\ \tau move0\ P\ (hp\ s)\ e\ ]\!] \Longrightarrow hp\ s'$
$= hp\ s$
and *τmoves0-heap-unchanged*: $[\![\ extTA,P,t \vdash \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle;\ \tau moves0\ P\ (hp\ s)\ es\ ]\!] \Longrightarrow hp$
$s' = hp\ s$
apply(*induct rule*: *red-reds.inducts*)
apply(*auto*)
apply(*fastforce simp add*: *map-eq-append-conv τexternal'-def τexternal-def dest*: *τexternal'-red-external-heap-unch*
done

lemma *τMove0-iff*:
$\tau Move0\ P\ h\ ees \longleftrightarrow (let\ (e,\ \text{-}) = ees\ in\ \tau move0\ P\ h\ e \vee final\ e)$
by(*cases ees*)(*simp*)

lemma *no-call-simps* [*simp*]:
$no\text{-}call\ P\ h\ (new\ C) = True$
$no\text{-}call\ P\ h\ (newA\ T\lfloor e\rceil) = no\text{-}call\ P\ h\ e$
$no\text{-}call\ P\ h\ (Cast\ T\ e) = no\text{-}call\ P\ h\ e$
$no\text{-}call\ P\ h\ (e\ instanceof\ T) = no\text{-}call\ P\ h\ e$
$no\text{-}call\ P\ h\ (Val\ v) = True$
$no\text{-}call\ P\ h\ (Var\ V) = True$
$no\text{-}call\ P\ h\ (V := e) = no\text{-}call\ P\ h\ e$
$no\text{-}call\ P\ h\ (e\ «bop»\ e') = (if\ is\text{-}val\ e\ then\ no\text{-}call\ P\ h\ e'\ else\ no\text{-}call\ P\ h\ e)$
$no\text{-}call\ P\ h\ (a\lfloor i\rceil) = (if\ is\text{-}val\ a\ then\ no\text{-}call\ P\ h\ i\ else\ no\text{-}call\ P\ h\ a)$
$no\text{-}call\ P\ h\ (AAss\ a\ i\ e) = (if\ is\text{-}val\ a\ then\ (if\ is\text{-}val\ i\ then\ no\text{-}call\ P\ h\ e\ else\ no\text{-}call\ P\ h\ i)\ else$
$no\text{-}call\ P\ h\ a)$
$no\text{-}call\ P\ h\ (a{\cdot}length) = no\text{-}call\ P\ h\ a$
$no\text{-}call\ P\ h\ (e{\cdot}F\{D\}) = no\text{-}call\ P\ h\ e$
$no\text{-}call\ P\ h\ (FAss\ e\ F\ D\ e') = (if\ is\text{-}val\ e\ then\ no\text{-}call\ P\ h\ e'\ else\ no\text{-}call\ P\ h\ e)$
$no\text{-}call\ P\ h\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e'')) = (if\ is\text{-}val\ e\ then\ (if\ is\text{-}val\ e'\ then\ no\text{-}call\ P\ h\ e''\ else$
$no\text{-}call\ P\ h\ e')\ else\ no\text{-}call\ P\ h\ e)$
$no\text{-}call\ P\ h\ (e{\cdot}M(es)) = (if\ is\text{-}val\ e\ then\ (if\ is\text{-}vals\ es \wedge is\text{-}addr\ e\ then\ synthesized\text{-}call\ P\ h\ (THE\ a.$
$e = addr\ a,\ M,\ THE\ vs.\ es = map\ Val\ vs)\ else\ no\text{-}calls\ P\ h\ es)\ else\ no\text{-}call\ P\ h\ e)$
$no\text{-}call\ P\ h\ (\{V{:}T{=}vo;\ e\}) = no\text{-}call\ P\ h\ e$
$no\text{-}call\ P\ h\ (sync_{V'}\ (e)\ e') = no\text{-}call\ P\ h\ e$
$no\text{-}call\ P\ h\ (insync_{V'}\ (ad)\ e) = no\text{-}call\ P\ h\ e$
$no\text{-}call\ P\ h\ (e;;e') = no\text{-}call\ P\ h\ e$
$no\text{-}call\ P\ h\ (if\ (e)\ e1\ else\ e2) = no\text{-}call\ P\ h\ e$
$no\text{-}call\ P\ h\ (while(e)\ e') = True$
$no\text{-}call\ P\ h\ (throw\ e) = no\text{-}call\ P\ h\ e$
$no\text{-}call\ P\ h\ (try\ e\ catch(C\ V)\ e') = no\text{-}call\ P\ h\ e$

by(*auto simp add*: *no-call-def no-calls-def*)

lemma *no-calls-simps* [*simp*]:
no-calls P h [] = *True*
no-calls P h (*e* # *es*) = (*if is-val e then no-calls P h es else no-call P h e*)
by(*simp-all add*: *no-call-def no-calls-def*)

lemma *no-calls-map-Val* [*simp*]:
no-calls P h (*map Val vs*)
by(*induct vs*) *simp-all*

lemma assumes *nfin*: ¬ *final e'*
shows *inline-call-τmove0-inv*: *call e* = ⌊*aMvs*⌋ ⟹ *τmove0 P h* (*inline-call e' e*) = *τmove0 P h e'*
and *inline-calls-τmoves0-inv*: *calls es* = ⌊*aMvs*⌋ ⟹ *τmoves0 P h* (*inline-calls e' es*) = *τmove0 P h e'*
apply(*induct e* **and** *es rule*: *τmove0.induct τmoves0.induct*)
apply(*insert nfin*)
apply *simp-all*
apply *auto*
done

lemma *τred0-iff* [*iff*]:
τred0 extTA P t h (*e*, *xs*) (*e'*, *xs'*) = (*extTA*,*P*,*t* ⊢ ⟨*e*, (*h*, *xs*)⟩ −*ε*→ ⟨*e'*, (*h*, *xs'*)⟩ ∧ *τmove0 P h e* ∧ *no-call P h e*)
by(*simp add*: *τred0-def*)

lemma *τreds0-iff* [*iff*]:
τreds0 extTA P t h (*es*, *xs*) (*es'*, *xs'*) =
(*extTA*,*P*,*t* ⊢ ⟨*es*, (*h*, *xs*)⟩ [−*ε*→] ⟨*es'*, (*h*, *xs'*)⟩ ∧ *τmoves0 P h es* ∧ *no-calls P h es*)
by(*simp add*: *τreds0-def*)

lemma *τred0t-1step*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*e*, (*h*, *xs*)⟩ −*ε*→ ⟨*e'*, (*h*, *xs'*)⟩; *τmove0 P h e*; *no-call P h e* ⟧
⟹ *τred0t extTA P t h* (*e*, *xs*) (*e'*, *xs'*)
by(*blast intro*: *tranclp.r-into-trancl*)

lemma *τred0t-2step*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*e*, (*h*, *xs*)⟩ −*ε*→ ⟨*e'*, (*h*, *xs'*)⟩; *τmove0 P h e*; *no-call P h e*;
extTA,*P*,*t* ⊢ ⟨*e'*, (*h*, *xs'*)⟩ −*ε*→ ⟨*e''*, (*h*, *xs''*)⟩; *τmove0 P h e'*; *no-call P h e'* ⟧
⟹ *τred0t extTA P t h* (*e*, *xs*) (*e''*, *xs''*)
by(*blast intro*: *tranclp.trancl-into-trancl*[*OF τred0t-1step*])

lemma *τred1t-3step*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*e*, (*h*, *xs*)⟩ −*ε*→ ⟨*e'*, (*h*, *xs'*)⟩; *τmove0 P h e*; *no-call P h e*;
extTA,*P*,*t* ⊢ ⟨*e'*, (*h*, *xs'*)⟩ −*ε*→ ⟨*e''*, (*h*, *xs''*)⟩; *τmove0 P h e'*; *no-call P h e'*;
extTA,*P*,*t* ⊢ ⟨*e''*, (*h*, *xs''*)⟩ −*ε*→ ⟨*e'''*, (*h*, *xs'''*)⟩; *τmove0 P h e''*; *no-call P h e''* ⟧
⟹ *τred0t extTA P t h* (*e*, *xs*) (*e'''*, *xs'''*)
by(*blast intro*: *tranclp.trancl-into-trancl*[*OF τred0t-2step*])

lemma *τreds0t-1step*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*es*, (*h*, *xs*)⟩ [−*ε*→] ⟨*es'*, (*h*, *xs'*)⟩; *τmoves0 P h es*; *no-calls P h es* ⟧
⟹ *τreds0t extTA P t h* (*es*, *xs*) (*es'*, *xs'*)
by(*blast intro*: *tranclp.r-into-trancl*)

lemma *τreds0t-2step*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*es*, (*h*, *xs*)⟩ [−*ε*→] ⟨*es′*, (*h*, *xs′*)⟩; *τmoves0 P h es*; *no-calls P h es*;
extTA,*P*,*t* ⊢ ⟨*es′*, (*h*, *xs′*)⟩ [−*ε*→] ⟨*es″*, (*h*, *xs″*)⟩; *τmoves0 P h es′*; *no-calls P h es′* ⟧
⟹ *τreds0t extTA P t h* (*es*, *xs*) (*es″*, *xs″*)
by(*blast intro*: *tranclp.trancl-into-trancl*[*OF τreds0t-1step*])

lemma *τreds0t-3step*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*es*, (*h*, *xs*)⟩ [−*ε*→] ⟨*es′*, (*h*, *xs′*)⟩; *τmoves0 P h es*; *no-calls P h es*;
extTA,*P*,*t* ⊢ ⟨*es′*, (*h*, *xs′*)⟩ [−*ε*→] ⟨*es″*, (*h*, *xs″*)⟩; *τmoves0 P h es′*; *no-calls P h es′*;
extTA,*P*,*t* ⊢ ⟨*es″*, (*h*, *xs″*)⟩ [−*ε*→] ⟨*es‴*, (*h*, *xs‴*)⟩; *τmoves0 P h es″*; *no-calls P h es″* ⟧
⟹ *τreds0t extTA P t h* (*es*, *xs*) (*es‴*, *xs‴*)
by(*blast intro*: *tranclp.trancl-into-trancl*[*OF τreds0t-2step*])

lemma *τred0r-1step*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*e*, (*h*, *xs*)⟩ −*ε*→ ⟨*e′*, (*h*, *xs′*)⟩; *τmove0 P h e*; *no-call P h e* ⟧
⟹ *τred0r extTA P t h* (*e*, *xs*) (*e′*, *xs′*)
by(*blast intro*: *r-into-rtranclp*)

lemma *τred0r-2step*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*e*, (*h*, *xs*)⟩ −*ε*→ ⟨*e′*, (*h*, *xs′*)⟩; *τmove0 P h e*; *no-call P h e*;
extTA,*P*,*t* ⊢ ⟨*e′*, (*h*, *xs′*)⟩ −*ε*→ ⟨*e″*, (*h*, *xs″*)⟩; *τmove0 P h e′*; *no-call P h e′* ⟧
⟹ *τred0r extTA P t h* (*e*, *xs*) (*e″*, *xs″*)
by(*blast intro*: *rtranclp.rtrancl-into-rtrancl*[*OF τred0r-1step*])

lemma *τred0r-3step*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*e*, (*h*, *xs*)⟩ −*ε*→ ⟨*e′*, (*h*, *xs′*)⟩; *τmove0 P h e*; *no-call P h e*;
extTA,*P*,*t* ⊢ ⟨*e′*, (*h*, *xs′*)⟩ −*ε*→ ⟨*e″*, (*h*, *xs″*)⟩; *τmove0 P h e′*; *no-call P h e′*;
extTA,*P*,*t* ⊢ ⟨*e″*, (*h*, *xs″*)⟩ −*ε*→ ⟨*e‴*, (*h*, *xs‴*)⟩; *τmove0 P h e″*; *no-call P h e″* ⟧
⟹ *τred0r extTA P t h* (*e*, *xs*) (*e‴*, *xs‴*)
by(*blast intro*: *rtranclp.rtrancl-into-rtrancl*[*OF τred0r-2step*])

lemma *τreds0r-1step*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*es*, (*h*, *xs*)⟩ [−*ε*→] ⟨*es′*, (*h*, *xs′*)⟩; *τmoves0 P h es*; *no-calls P h es* ⟧
⟹ *τreds0r extTA P t h* (*es*, *xs*) (*es′*, *xs′*)
by(*blast intro*: *r-into-rtranclp*)

lemma *τreds0r-2step*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*es*, (*h*, *xs*)⟩ [−*ε*→] ⟨*es′*, (*h*, *xs′*)⟩; *τmoves0 P h es*; *no-calls P h es*;
extTA,*P*,*t* ⊢ ⟨*es′*, (*h*, *xs′*)⟩ [−*ε*→] ⟨*es″*, (*h*, *xs″*)⟩; *τmoves0 P h es′*; *no-calls P h es′* ⟧
⟹ *τreds0r extTA P t h* (*es*, *xs*) (*es″*, *xs″*)
by(*blast intro*: *rtranclp.rtrancl-into-rtrancl*[*OF τreds0r-1step*])

lemma *τreds0r-3step*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*es*, (*h*, *xs*)⟩ [−*ε*→] ⟨*es′*, (*h*, *xs′*)⟩; *τmoves0 P h es*; *no-calls P h es*;
extTA,*P*,*t* ⊢ ⟨*es′*, (*h*, *xs′*)⟩ [−*ε*→] ⟨*es″*, (*h*, *xs″*)⟩; *τmoves0 P h es′*; *no-calls P h es′*;
extTA,*P*,*t* ⊢ ⟨*es″*, (*h*, *xs″*)⟩ [−*ε*→] ⟨*es‴*, (*h*, *xs‴*)⟩; *τmoves0 P h es″*; *no-calls P h es″* ⟧
⟹ *τreds0r extTA P t h* (*es*, *xs*) (*es‴*, *xs‴*)
by(*blast intro*: *rtranclp.rtrancl-into-rtrancl*[*OF τreds0r-2step*])

lemma *τred0t-inj-τreds0t*:
τred0t extTA P t h (*e*, *xs*) (*e′*, *xs′*)
⟹ *τreds0t extTA P t h* (*e* # *es*, *xs*) (*e′* # *es*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl ListRed1*)

lemma *τreds0t-cons-τreds0t*:
τreds0t extTA P t h (*es*, *xs*) (*es′*, *xs′*)
⟹ *τreds0t extTA P t h* (*Val v* # *es*, *xs*) (*Val v* # *es′*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl ListRed2*)

lemma *τred0r-inj-τreds0r*:
τred0r extTA P t h (*e*, *xs*) (*e′*, *xs′*)
⟹ *τreds0r extTA P t h* (*e* # *es*, *xs*) (*e′* # *es*, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl ListRed1*)

lemma *τreds0r-cons-τreds0r*:
τreds0r extTA P t h (*es*, *xs*) (*es′*, *xs′*)
⟹ *τreds0r extTA P t h* (*Val v* # *es*, *xs*) (*Val v* # *es′*, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl ListRed2*)

lemma *NewArray-τred0t-xt*:
τred0t extTA P t h (*e*, *xs*) (*e′*, *xs′*)
⟹ *τred0t extTA P t h* (*newA T*⌊*e*⌉, *xs*) (*newA T*⌊*e′*⌉, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl NewArrayRed*)

lemma *Cast-τred0t-xt*:
τred0t extTA P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred0t extTA P t h* (*Cast T e*, *xs*) (*Cast T e′*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl CastRed*)

lemma *InstanceOf-τred0t-xt*:
τred0t extTA P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred0t extTA P t h* (*e instanceof T*, *xs*) (*e′ instanceof T*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl InstanceOfRed*)

lemma *BinOp-τred0t-xt1*:
τred0t extTA P t h (*e1*, *xs*) (*e1′*, *xs′*) ⟹ *τred0t extTA P t h* (*e1* «*bop*» *e2*, *xs*) (*e1′* «*bop*» *e2*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl BinOpRed1*)

lemma *BinOp-τred0t-xt2*:
τred0t extTA P t h (*e2*, *xs*) (*e2′*, *xs′*) ⟹ *τred0t extTA P t h* (*Val v* «*bop*» *e2*, *xs*) (*Val v* «*bop*» *e2′*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl BinOpRed2*)

lemma *LAss-τred0t*:
τred0t extTA P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred0t extTA P t h* (*V* := *e*, *xs*) (*V* := *e′*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl LAssRed*)

lemma *AAcc-τred0t-xt1*:
τred0t extTA P t h (*a*, *xs*) (*a′*, *xs′*) ⟹ *τred0t extTA P t h* (*a*⌊*i*⌉, *xs*) (*a′*⌊*i*⌉, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl AAccRed1*)

lemma *AAcc-τred0t-xt2*:
τred0t extTA P t h (*i*, *xs*) (*i′*, *xs′*) ⟹ *τred0t extTA P t h* (*Val a*⌊*i*⌉, *xs*) (*Val a*⌊*i′*⌉, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl AAccRed2*)

lemma *AAss-τred0t-xt1*:
τred0t extTA P t h (*a*, *xs*) (*a′*, *xs′*) ⟹ *τred0t extTA P t h* (*a*⌊*i*⌉ := *e*, *xs*) (*a′*⌊*i*⌉ := *e*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl AAssRed1*)

lemma *AAss-τred0t-xt2*:

$\tau red0t\ extTA\ P\ t\ h\ (i,\ xs)\ (i',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (Val\ a\lfloor i\rceil := e,\ xs)\ (Val\ a\lfloor i'\rceil := e,\ xs')$
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl AAssRed2*)

lemma *AAss-τred0t-xt3*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (Val\ a\lfloor Val\ i\rceil := e,\ xs)\ (Val\ a\lfloor Val\ i\rceil := e',\ xs')$
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl AAssRed3*)

lemma *ALength-τred0t-xt*:
$\tau red0t\ extTA\ P\ t\ h\ (a,\ xs)\ (a',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (a{\cdot}length,\ xs)\ (a'{\cdot}length,\ xs')$
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl ALengthRed*)

lemma *FAcc-τred0t-xt*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (e{\cdot}F\{D\},\ xs)\ (e'{\cdot}F\{D\},\ xs')$
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl FAccRed*)

lemma *FAss-τred0t-xt1*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (e{\cdot}F\{D\} := e2,\ xs)\ (e'{\cdot}F\{D\} := e2,\ xs')$
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl FAssRed1*)

lemma *FAss-τred0t-xt2*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (Val\ v{\cdot}F\{D\} := e,\ xs)\ (Val\ v{\cdot}F\{D\} := e',\ xs')$
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl FAssRed2*)

lemma *CAS-τred0t-xt1*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3),\ xs)\ (e'{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3),\ xs')$
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl CASRed1*)

lemma *CAS-τred0t-xt2*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ e,\ e3),\ xs)\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e3),\ xs')$
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl CASRed2*)

lemma *CAS-τred0t-xt3*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v',\ e),\ xs)\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v',\ e'),\ xs')$
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl CASRed3*)

lemma *Call-τred0t-obj*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (e{\cdot}M(ps),\ xs)\ (e'{\cdot}M(ps),\ xs')$
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl CallObj*)

lemma *Call-τred0t-param*:
$\tau reds0t\ extTA\ P\ t\ h\ (es,\ xs)\ (es',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (Val\ v{\cdot}M(es),\ xs)\ (Val\ v{\cdot}M(es'),\ xs')$
by(*induct rule*: *tranclp-induct2*)(*fastforce intro*: *tranclp.trancl-into-trancl CallParams*)+

lemma *Block-τred0t-xt*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs(V := vo))\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (\{V{:}T{=}vo;\ e\},\ xs)\ (\{V{:}T{=}xs'\ V;\ e'\},\ xs'(V := xs\ V))$
proof(*induct rule*: *tranclp-induct2*)
case *base* **thus** *?case* **by**(*auto intro*: *BlockRed simp del*: *fun-upd-apply*)
next

case (*step e′ xs′ e″ xs″*)
from ‹*τred0 extTA P t h* (*e′*, *xs′*) (*e″*, *xs″*)›
have *extTA*,*P*,*t* ⊢ ⟨*e′*,(*h*, *xs′*)⟩ −*ε*→ ⟨*e″*,(*h*, *xs″*)⟩ *τmove0 P h e′ no-call P h e′* **by** *auto*
hence *extTA*,*P*,*t* ⊢ ⟨*e′*,(*h*, *xs′*(*V* := *xs V*, *V* := *xs′ V*))⟩ −*ε*→ ⟨*e″*,(*h*, *xs″*)⟩ **by** *simp*
from *BlockRed*[*OF this*, *of T*] ‹*τmove0 P h e′*› ‹*no-call P h e′*›
have *τred0 extTA P t h* ({*V*:*T*=*xs′ V*; *e′*}, *xs′*(*V* := *xs V*)) ({*V*:*T*=*xs″ V*; *e″*}, *xs″*(*V* := *xs V*))
by(*auto*)
with ‹*τred0t extTA P t h* ({*V*:*T*=*vo*; *e*}, *xs*) ({*V*:*T*=*xs′ V*; *e′*}, *xs′*(*V* := *xs V*))› **show** *?case* **..**
qed

lemma *Sync-τred0t-xt*:
τred0t extTA P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred0t extTA P t h* (*sync*$_V$ (*e*) *e2*, *xs*) (*sync*$_V$ (*e′*) *e2*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl SynchronizedRed1*)

lemma *InSync-τred0t-xt*:
τred0t extTA P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred0t extTA P t h* (*insync*$_V$ (*a*) *e*, *xs*) (*insync*$_V$ (*a*) *e′*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl SynchronizedRed2*)

lemma *Seq-τred0t-xt*:
τred0t extTA P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred0t extTA P t h* (*e*;;*e2*, *xs*) (*e′*;;*e2*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl SeqRed*)

lemma *Cond-τred0t-xt*:
τred0t extTA P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred0t extTA P t h* (*if* (*e*) *e1 else e2*, *xs*) (*if* (*e′*) *e1 else e2*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl CondRed*)

lemma *Throw-τred0t-xt*:
τred0t extTA P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred0t extTA P t h* (*throw e*, *xs*) (*throw e′*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl ThrowRed*)

lemma *Try-τred0t-xt*:
τred0t extTA P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred0t extTA P t h* (*try e catch*(*C V*) *e2*, *xs*) (*try e′ catch*(*C V*) *e2*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl TryRed*)

lemma *NewArray-τred0r-xt*:
τred0r extTA P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred0r extTA P t h* (*newA T*⌊*e*⌉, *xs*) (*newA T*⌊*e′*⌉, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl NewArrayRed*)

lemma *Cast-τred0r-xt*:
τred0r extTA P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred0r extTA P t h* (*Cast T e*, *xs*) (*Cast T e′*, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl CastRed*)

lemma *InstanceOf-τred0r-xt*:
τred0r extTA P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred0r extTA P t h* (*e instanceof T*, *xs*) (*e′ instanceof T*, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl InstanceOfRed*)

lemma *BinOp-τred0r-xt1*:
τred0r extTA P t h (*e1*, *xs*) (*e1′*, *xs′*) ⟹ *τred0r extTA P t h* (*e1 «bop» e2*, *xs*) (*e1′ «bop» e2*, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl BinOpRed1*)

lemma *BinOp-τred0r-xt2*:
τred0r extTA P t h (*e2*, *xs*) (*e2′*, *xs′*) ⟹ *τred0r extTA P t h* (*Val v «bop» e2*, *xs*) (*Val v «bop»*

e2', *xs'*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl BinOpRed2*)

lemma *LAss-τred0r*:
τred0r extTA P t h (*e*, *xs*) (*e'*, *xs'*) ⟹ *τred0r extTA P t h* (*V* := *e*, *xs*) (*V* := *e'*, *xs'*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl LAssRed*)

lemma *AAcc-τred0r-xt1*:
τred0r extTA P t h (*a*, *xs*) (*a'*, *xs'*) ⟹ *τred0r extTA P t h* (*a*⌊*i*⌉, *xs*) (*a'*⌊*i*⌉, *xs'*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl AAccRed1*)

lemma *AAcc-τred0r-xt2*:
τred0r extTA P t h (*i*, *xs*) (*i'*, *xs'*) ⟹ *τred0r extTA P t h* (*Val a*⌊*i*⌉, *xs*) (*Val a*⌊*i'*⌉, *xs'*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl AAccRed2*)

lemma *AAss-τred0r-xt1*:
τred0r extTA P t h (*a*, *xs*) (*a'*, *xs'*) ⟹ *τred0r extTA P t h* (*a*⌊*i*⌉ := *e*, *xs*) (*a'*⌊*i*⌉ := *e*, *xs'*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl AAssRed1*)

lemma *AAss-τred0r-xt2*:
τred0r extTA P t h (*i*, *xs*) (*i'*, *xs'*) ⟹ *τred0r extTA P t h* (*Val a*⌊*i*⌉ := *e*, *xs*) (*Val a*⌊*i'*⌉ := *e*, *xs'*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl AAssRed2*)

lemma *AAss-τred0r-xt3*:
τred0r extTA P t h (*e*, *xs*) (*e'*, *xs'*) ⟹ *τred0r extTA P t h* (*Val a*⌊*Val i*⌉ := *e*, *xs*) (*Val a*⌊*Val i*⌉ := *e'*, *xs'*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl AAssRed3*)

lemma *ALength-τred0r-xt*:
τred0r extTA P t h (*a*, *xs*) (*a'*, *xs'*) ⟹ *τred0r extTA P t h* (*a*·*length*, *xs*) (*a'*·*length*, *xs'*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl ALengthRed*)

lemma *FAcc-τred0r-xt*:
τred0r extTA P t h (*e*, *xs*) (*e'*, *xs'*) ⟹ *τred0r extTA P t h* (*e*·*F*{*D*}, *xs*) (*e'*·*F*{*D*}, *xs'*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl FAccRed*)

lemma *FAss-τred0r-xt1*:
τred0r extTA P t h (*e*, *xs*) (*e'*, *xs'*) ⟹ *τred0r extTA P t h* (*e*·*F*{*D*} := *e2*, *xs*) (*e'*·*F*{*D*} := *e2*, *xs'*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl FAssRed1*)

lemma *FAss-τred0r-xt2*:
τred0r extTA P t h (*e*, *xs*) (*e'*, *xs'*) ⟹ *τred0r extTA P t h* (*Val v*·*F*{*D*} := *e*, *xs*) (*Val v*·*F*{*D*} := *e'*, *xs'*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl FAssRed2*)

lemma *CAS-τred0r-xt1*:
τred0r extTA P t h (*e*, *xs*) (*e'*, *xs'*) ⟹ *τred0r extTA P t h* (*e*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *xs*) (*e'*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *xs'*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl CASRed1*)

lemma *CAS-τred0r-xt2*:
τred0r extTA P t h (*e*, *xs*) (*e'*, *xs'*) ⟹ *τred0r extTA P t h* (*Val v*·*compareAndSwap*(*D*·*F*, *e*, *e3*), *xs*) (*Val v*·*compareAndSwap*(*D*·*F*, *e'*, *e3*), *xs'*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl CASRed2*)

lemma *CAS-τred0r-xt3*:
τred0r extTA P t h (e, xs) (e′, xs′) ⟹ τred0r extTA P t h (Val v·compareAndSwap(D·F, Val v′, e), xs) (Val v·compareAndSwap(D·F, Val v′, e′), xs′)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl CASRed3*)

lemma *Call-τred0r-obj*:
τred0r extTA P t h (e, xs) (e′, xs′) ⟹ τred0r extTA P t h (e·M(ps), xs) (e′·M(ps), xs′)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl CallObj*)

lemma *Call-τred0r-param*:
τreds0r extTA P t h (es, xs) (es′, xs′) ⟹ τred0r extTA P t h (Val v·M(es), xs) (Val v·M(es′), xs′)
by(*induct rule*: *rtranclp-induct2*)(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl CallParams*)+

lemma *Block-τred0r-xt*:
τred0r extTA P t h (e, xs(V := vo)) (e′, xs′) ⟹ τred0r extTA P t h ({V:T=vo; e}, xs) ({V:T=xs′ V; e′}, xs′(V := xs V))
proof(*induct rule*: *rtranclp-induct2*)
case *refl* **thus** *?case* **by**(*simp del*: *fun-upd-apply*)(*auto simp add*: *fun-upd-apply*)
next
case (*step e′ xs′ e″ xs″*)
from ‹*τred0 extTA P t h (e′, xs′) (e″, xs″)*›
have *extTA,P,t ⊢ ⟨e′,(h, xs′)⟩ −ε→ ⟨e″,(h, xs″)⟩ τmove0 P h e′ no-call P h e′* **by** *auto*
hence *extTA,P,t ⊢ ⟨e′,(h, xs′(V := xs V, V := xs′ V))⟩ −ε→ ⟨e″,(h, xs″)⟩* **by** *simp*
from *BlockRed*[*OF this, of T*] ‹*τmove0 P h e′*› ‹*no-call P h e′*›
have *τred0 extTA P t h ({V:T=xs′ V; e′}, xs′(V := xs V)) ({V:T=xs″ V; e″}, xs″(V := xs V))*
by *auto*
with ‹*τred0r extTA P t h ({V:T=vo; e}, xs) ({V:T=xs′ V; e′}, xs′(V := xs V))*› **show** *?case* **..**
qed

lemma *Sync-τred0r-xt*:
τred0r extTA P t h (e, xs) (e′, xs′) ⟹ τred0r extTA P t h (sync$_V$ (e) e2, xs) (sync$_V$ (e′) e2, xs′)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl SynchronizedRed1*)

lemma *InSync-τred0r-xt*:
τred0r extTA P t h (e, xs) (e′, xs′) ⟹ τred0r extTA P t h (insync$_V$ (a) e, xs) (insync$_V$ (a) e′, xs′)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl SynchronizedRed2*)

lemma *Seq-τred0r-xt*:
τred0r extTA P t h (e, xs) (e′, xs′) ⟹ τred0r extTA P t h (e;;e2, xs) (e′;;e2, xs′)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl SeqRed*)

lemma *Cond-τred0r-xt*:
τred0r extTA P t h (e, xs) (e′, xs′) ⟹ τred0r extTA P t h (if (e) e1 else e2, xs) (if (e′) e1 else e2, xs′)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl CondRed*)

lemma *Throw-τred0r-xt*:
τred0r extTA P t h (e, xs) (e′, xs′) ⟹ τred0r extTA P t h (throw e, xs) (throw e′, xs′)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl ThrowRed*)

lemma *Try-τred0r-xt*:
τred0r extTA P t h (e, xs) (e′, xs′) ⟹ τred0r extTA P t h (try e catch(C V) e2, xs) (try e′ catch(C V) e2, xs′)

by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl TryRed*)

lemma *τRed0-conv* [*iff*]:
  τRed0 P t h (*e*, *es*) (*e′*, *es′*) = (*P*,*t* ⊢*0* ⟨*e*/*es*, *h*⟩ −*ε*→ ⟨*e′*/*es′*, *h*⟩ ∧ *τMove0 P h* (*e*, *es*))
by(*simp add*: *τRed0-def*)

lemma *τred0r-lcl-incr*:
  τred0r extTA P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *dom xs* ⊆ *dom xs′*
by(*induct rule*: *rtranclp-induct2*)(*auto dest*: *red-lcl-incr del*: *subsetI*)

lemma *τred0t-lcl-incr*:
  τred0t extTA P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *dom xs* ⊆ *dom xs′*
by(*rule τred0r-lcl-incr*)(*rule tranclp-into-rtranclp*)

lemma *τred0r-dom-lcl*:
  assumes *wwf*: *wwf-J-prog P*
  shows *τred0r extTA P t h* (*e*, *xs*) (*e′*, *xs′*) ⟹ *dom xs′* ⊆ *dom xs* ∪ *fv e*
apply(*induct rule*: *converse-rtranclp-induct2*)
 apply *blast*
apply(*clarsimp del*: *subsetI*)
apply(*frule red-dom-lcl*)
 apply(*drule red-fv-subset*[*OF wwf*])
apply *auto*
done

lemma *τred0t-dom-lcl*:
  assumes *wwf*: *wwf-J-prog P*
  shows *τred0t extTA P t h* (*e*, *xs*) (*e′*, *xs′*) ⟹ *dom xs′* ⊆ *dom xs* ∪ *fv e*
by(*rule τred0r-dom-lcl*[*OF wwf*])(*rule tranclp-into-rtranclp*)

lemma *τred0r-fv-subset*:
  assumes *wwf*: *wwf-J-prog P*
  shows *τred0r extTA P t h* (*e*, *xs*) (*e′*, *xs′*) ⟹ *fv e′* ⊆ *fv e*
by(*induct rule*: *converse-rtranclp-induct2*)(*auto dest*: *red-fv-subset*[*OF wwf*])

lemma *τred0t-fv-subset*:
  assumes *wwf*: *wwf-J-prog P*
  shows *τred0t extTA P t h* (*e*, *xs*) (*e′*, *xs′*) ⟹ *fv e′* ⊆ *fv e*
by(*rule τred0r-fv-subset*[*OF wwf*])(*rule tranclp-into-rtranclp*)

lemma fixes *e* :: (*'a*, *'b*, *'addr*) *exp* **and** *es* :: (*'a*, *'b*, *'addr*) *exp list*
  shows *τmove0-callD*: *call e* = ⌊(*a*, *M*, *vs*)⌋ ⟹ *τmove0 P h e* ⟷ (*synthesized-call P h* (*a*, *M*, *vs*) ⟶ *τexternal′ P h a M*)
  and *τmoves0-callsD*: *calls es* = ⌊(*a*, *M*, *vs*)⌋ ⟹ *τmoves0 P h es* ⟷ (*synthesized-call P h* (*a*, *M*, *vs*) ⟶ *τexternal′ P h a M*)
apply(*induct e* **and** *es rule*: *call.induct calls.induct*)
apply(*auto split*: *if-split-asm simp add*: *is-vals-conv*)
apply(*fastforce simp add*: *synthesized-call-def map-eq-append-conv τexternal′-def τexternal-def dest*: *sees-method-fun*)+
done

lemma fixes *e* :: (*'a*, *'b*, *'addr*) *exp* **and** *es* :: (*'a*, *'b*, *'addr*) *exp list*
  shows *τmove0-not-call*: ⟦ *τmove0 P h e*; *call e* = ⌊(*a*, *M*, *vs*)⌋; *synthesized-call P h* (*a*, *M*, *vs*) ⟧ ⟹ *τexternal′ P h a M*

and *τmoves0-not-calls*: ⟦ *τmoves0 P h es*; *calls es* = ⌊(*a*, *M*, *vs*)⌋; *synthesized-call P h* (*a*, *M*, *vs*) ⟧
⟹ *τexternal′ P h a M*
apply(*drule τmove0-callD*[**where** *P*=*P* **and** *h*=*h*], *simp*)
apply(*drule τmoves0-callsD*[**where** *P*=*P* **and** *h*=*h*], *simp*)
done

lemma *τred0-into-τRed0*:
assumes *red*: *τred0* (*extTA2J0 P*) *P t h* (*e*, *Map.empty*) (*e′*, *xs′*)
shows *τRed0 P t h* (*e*, *es*) (*e′*, *es*)
proof −
from *red* **have** *red*: *extTA2J0 P*,*P*,*t* ⊢ ⟨*e*, (*h*, *Map.empty*)⟩ −*ε*→ ⟨*e′*, (*h*, *xs′*)⟩
and *τmove0 P h e* **and** *no-call P h e* **by** *auto*
hence *P*,*t* ⊢*0* ⟨*e*/*es*,*h*⟩ −*ε*→ ⟨*e′*/*es*,*h*⟩
by−(*erule red0Red*,*auto simp add*: *no-call-def*)
thus *?thesis* **using** ‹*τmove0 P h e*› **by**(*auto*)
qed

lemma *τred0r-into-τRed0r*:
assumes *wwf*: *wwf-J-prog P*
shows
⟦ *τred0r* (*extTA2J0 P*) *P t h* (*e*, *Map.empty*) (*e″*, *Map.empty*); *fv e* = {} ⟧
⟹ *τRed0r P t h* (*e*, *es*) (*e″*, *es*)
proof(*induct e xs*≡*Map.empty* :: *′addr locals rule*: *converse-rtranclp-induct2*)
case *refl* **show** *?case* **by** *blast*
next
case (*step e e′ xs′*)
from ‹*τred0* (*extTA2J0 P*) *P t h* (*e*, *Map.empty*) (*e′*, *xs′*)›
have *red*: *extTA2J0 P*,*P*,*t* ⊢ ⟨*e*, (*h*, *Map.empty*)⟩ −*ε*→ ⟨*e′*, (*h*, *xs′*)⟩
and *τmove0 P h e* **and** *no-call P h e* **by** *auto*
from *red-dom-lcl*[*OF red*] ‹*fv e* = {}›
have *dom xs′* = {} **by**(*auto split*:*if-split-asm*)
hence *xs′* = *Map.empty* **by**(*auto*)
moreover
from *wwf red* **have** *fv e′* ⊆ *fv e* **by**(*rule red-fv-subset*)
with ‹*fv e* = {}› **have** *fv e′* = {} **by** *blast*
ultimately have *τRed0r P t h* (*e′*, *es*) (*e″*, *es*) **by**(*rule step*)
moreover from *red* ‹*τmove0 P h e*› ‹*xs′* = *Map.empty*› ‹*no-call P h e*›
have *τRed0 P t h* (*e*, *es*) (*e′*, *es*) **by**(*auto simp add*: *no-call-def intro*!: *red0Red*)
ultimately show *?case* **by**(*blast intro*: *converse-rtranclp-into-rtranclp*)
qed

lemma *τred0t-into-τRed0t*:
assumes *wwf*: *wwf-J-prog P*
shows
⟦ *τred0t* (*extTA2J0 P*) *P t h* (*e*, *Map.empty*) (*e″*, *Map.empty*); *fv e* = {} ⟧
⟹ *τRed0t P t h* (*e*, *es*) (*e″*, *es*)
proof(*induct e xs*≡*Map.empty* :: *′addr locals rule*: *converse-tranclp-induct2*)
case *base* **thus** *?case*
by(*blast intro*!: *tranclp.r-into-trancl τred0-into-τRed0*)
next
case (*step e e′ xs′*)
from ‹*τred0* (*extTA2J0 P*) *P t h* (*e*, *Map.empty*) (*e′*, *xs′*)›
have *red*: *extTA2J0 P*,*P*,*t* ⊢ ⟨*e*, (*h*, *Map.empty*)⟩ −*ε*→ ⟨*e′*, (*h*, *xs′*)⟩ **and** *τmove0 P h e* **and** *no-call*

P h e **by** *auto*
from *red-dom-lcl*[*OF red*] ‹*fv e* = {}›
have *dom xs'* = {} **by**(*auto split*:*if-split-asm*)
hence *xs'* = *Map.empty* **by** *auto*
moreover from *wwf red* **have** *fv e'* ⊆ *fv e* **by**(*rule red-fv-subset*)
with ‹*fv e* = {}› **have** *fv e'* = {} **by** *blast*
ultimately have *τRed0t P t h* (*e'*, *es*) (*e''*, *es*) **by**(*rule step*)
moreover from *red* ‹*τmove0 P h e*› ‹*xs'* = *Map.empty*› ‹*no-call P h e*›
have *τRed0 P t h* (*e*, *es*) (*e'*, *es*) **by**(*auto simp add*: *no-call-def intro*!: *red0Red*)
ultimately show *?case* **by**(*blast intro*: *tranclp-into-tranclp2*)
qed

lemma *τred0r-Val*:
τred0r extTA P t h (*Val v*, *xs*) *s'* ⟷ *s'* = (*Val v*, *xs*)
proof
assume *τred0r extTA P t h* (*Val v*, *xs*) *s'*
thus *s'* = (*Val v*, *xs*) **by** *induct*(*auto*)
qed *auto*

lemma *τred0t-Val*:
τred0t extTA P t h (*Val v*, *xs*) *s'* ⟷ *False*
proof
assume *τred0t extTA P t h* (*Val v*, *xs*) *s'*
thus *False* **by** *induct auto*
qed *auto*

lemma *τreds0r-map-Val*:
τreds0r extTA P t h (*map Val vs*, *xs*) *s'* ⟷ *s'* = (*map Val vs*, *xs*)
proof
assume *τreds0r extTA P t h* (*map Val vs*, *xs*) *s'*
thus *s'* = (*map Val vs*, *xs*) **by** *induct auto*
qed *auto*

lemma *τreds0t-map-Val*:
τreds0t extTA P t h (*map Val vs*, *xs*) *s'* ⟷ *False*
proof
assume *τreds0t extTA P t h* (*map Val vs*, *xs*) *s'*
thus *False* **by** *induct auto*
qed *auto*

lemma *Red-Suspend-is-call*:
⟦ *P*,*t* ⊢*0* ⟨*e*/*exs*, *h*⟩ −*ta*→ ⟨*e'*/*exs'*, *h'*⟩; *Suspend w* ∈ *set* {|*ta*|}*w* ⟧ ⟹ *is-call e'*
by(*auto elim*!: *red0.cases dest*: *red-Suspend-is-call simp add*: *is-call-def*)

lemma *red0-mthr*: *multithreaded final-expr0* (*mred0 P*)
by(*unfold-locales*)(*auto elim*!: *red0.cases dest*: *red-new-thread-heap*)

lemma *red0-τmthr-wf*: *τmultithreaded-wf final-expr0* (*mred0 P*) (*τMOVE0 P*)
proof −
interpret *multithreaded final-expr0 mred0 P* **by**(*rule red0-mthr*)
show *?thesis*
proof
fix *x1 m1 t ta1 x1' m1'*

```
    assume mred0 P t (x1, m1) ta1 (x1', m1') τMOVE0 P (x1, m1) ta1 (x1', m1')
    thus m1 = m1' by(cases x1)(fastforce elim!: red0.cases dest: τmove0-heap-unchanged)
  qed(simp add: split-beta)
qed

lemma red-τmthr-wf: τmultithreaded-wf final-expr (mred P) (τMOVE P)
proof
  fix x1 m1 t ta1 x1' m1'
  assume mred P t (x1, m1) ta1 (x1', m1') τMOVE P (x1, m1) ta1 (x1', m1')
  thus m1 = m1' by(auto dest: τmove0-heap-unchanged simp add: split-def)
qed(simp add: split-beta)

end

sublocale J-heap-base < red-mthr:
  τmultithreaded-wf
    final-expr
    mred P
    convert-RA
    τMOVE P
  for P
by(rule red-τmthr-wf)

sublocale J-heap-base < red0-mthr:
  τmultithreaded-wf
    final-expr0
    mred0 P
    convert-RA
    τMOVE0 P
  for P
by(rule red0-τmthr-wf)

context J-heap-base begin

lemma τRed0r-into-red0-τmthr-silent-moves:
  τRed0r P t h (e, es) (e'', es'') ⟹ red0-mthr.silent-moves P t ((e, es), h) ((e'', es''), h)
apply(induct rule: rtranclp-induct2)
 apply blast
apply(erule rtranclp.rtrancl-into-rtrancl)
apply(simp add: red0-mthr.silent-move-iff)
done

lemma τRed0t-into-red0-τmthr-silent-movet:
  τRed0t P t h (e, es) (e'', es'') ⟹ red0-mthr.silent-movet P t ((e, es), h) ((e'', es''), h)
apply(induct rule: tranclp-induct2)
apply(fastforce simp add: red0-mthr.silent-move-iff elim: tranclp.trancl-into-trancl)+
done

end

end
```

7.3 Bisimulation proof for between source code small step semantics with and without callstacks for single threads

theory *J0Bisim* **imports**
J0
../J/JWellForm
../Common/ExternalCallWF
begin

inductive *wf-state* :: *'addr expr* × *'addr expr list* ⇒ *bool*
where
⟦ *fvs* (*e* # *es*) = {}; ∀ *e* ∈ *set es*. *is-call e* ⟧
⟹ *wf-state* (*e*, *es*)

inductive *bisim-red-red0* :: (*'addr expr* × *'addr locals*) × *'heap* ⇒ (*'addr expr* × *'addr expr list*) × *'heap* ⇒ *bool*
where
wf-state ees ⟹ *bisim-red-red0* ((*collapse ees*, *Map.empty*), *h*) (*ees*, *h*)

abbreviation *ta-bisim0* :: (*'addr*, *'thread-id*, *'heap*) *J-thread-action* ⇒ (*'addr*, *'thread-id*, *'heap*) *J0-thread-action* ⇒ *bool*
where *ta-bisim0* ≡ *ta-bisim* (λ*t*. *bisim-red-red0*)

lemma *wf-state-iff* [*simp*, *code*]:
wf-state (*e*, *es*) ⟷ *fvs* (*e* # *es*) = {} ∧ (∀ *e* ∈ *set es*. *is-call e*)
by(*simp add*: *wf-state.simps*)

lemma *bisim-red-red0I* [*intro*]:
⟦ *e'* = *collapse ees*; *xs* = *Map.empty*; *h'* = *h*; *wf-state ees* ⟧ ⟹ *bisim-red-red0* ((*e'*, *xs*), *h'*) (*ees*, *h*)
by(*simp add*: *bisim-red-red0.simps del*: *split-paired-Ex*)

lemma *bisim-red-red0-final0D*:
⟦ *bisim-red-red0* (*x1*, *m1*) (*x2*, *m2*); *final-expr0 x2* ⟧ ⟹ *final-expr x1*
by(*erule bisim-red-red0.cases*) *auto*

context *J-heap-base* **begin**

lemma *red0-preserves-wf-state*:
assumes *wf*: *wwf-J-prog P*
and *red*: *P*,*t* ⊢0 ⟨*e* / *es*, *h*⟩ −*ta*→ ⟨*e'* / *es'*, *h'*⟩
and *wf-state*: *wf-state* (*e*, *es*)
shows *wf-state* (*e'*, *es'*)
using *wf-state*
proof(*cases*)
assume *fvs* (*e* # *es*) = {} **and** *icl*: ∀ *e* ∈ *set es*. *is-call e*
hence *fv*: *fv e* = {} *fvs es* = {} **by** *auto*
show *?thesis*
proof
from *red* **show** *fvs* (*e'* # *es'*) = {}
proof *cases*
case (*red0Red xs'*)
hence [*simp*]: *es'* = *es*
and *red*: *extTA2J0 P*,*P*,*t* ⊢ ⟨*e*,(*h*, *Map.empty*)⟩ −*ta*→ ⟨*e'*,(*h'*, *xs'*)⟩ **by** *auto*
from *red-fv-subset*[*OF wf red*] *fv* **have** *fv e'* = {} **by** *auto*

```
      with fv show ?thesis by simp
    next
      case (red0Call a M vs U Ts T pns body D)
      hence [simp]: ta = ε
        e' = blocks (this # pns) (Class D # Ts) (Addr a # vs) body
        es' = e # es h' = h
        and sees: P ⊢ class-type-of U sees M: Ts→T = ⌊(pns, body)⌋ in D by auto
      from sees-wf-mdecl[OF wf sees]
      have fv body ⊆ insert this (set pns) length Ts = length pns by(simp-all add: wf-mdecl-def)
      thus ?thesis using fv ‹length vs = length pns› by auto
    next
      case (red0Return E)
      with fv-inline-call[of e E] show ?thesis using fv by auto
    qed
  next
    from red icl show ∀ e∈set es'. is-call e
      by cases(simp-all add: is-call-def)
  qed
qed

lemma new-thread-bisim0-extNTA2J-extNTA2J0:
  assumes wf: wwf-J-prog P
  and red: P,t ⊢ ⟨a'•M'(vs), h⟩ −ta→ext ⟨va, h'⟩
  and nt: NewThread t' CMa m ∈ set {|ta|}_t
  shows bisim-red-red0 (extNTA2J P CMa, m) (extNTA2J0 P CMa, m)
proof −
  obtain C M a where CMa [simp]: CMa = (C, M, a) by(cases CMa)
  from red nt have [simp]: m = h' by(rule red-ext-new-thread-heap)
  from red-external-new-thread-sees[OF wf red nt[unfolded CMa]]
  obtain T pns body D where h'a: typeof-addr h' a = ⌊Class-type C⌋
    and sees: P ⊢ C sees M: []→T = ⌊(pns, body)⌋ in D by auto
  from sees-wf-mdecl[OF wf sees] have fv body ⊆ {this} by(auto simp add: wf-mdecl-def)
  with red nt h'a sees show ?thesis by(fastforce simp add: is-call-def intro: bisim-red-red0.intros)
qed

lemma ta-bisim0-extNTA2J-extNTA2J0:
  ⟦ wwf-J-prog P; P,t ⊢ ⟨a'•M'(vs), h⟩ −ta→ext ⟨va, h'⟩ ⟧
  ⟹ ta-bisim0 (extTA2J P ta) (extTA2J0 P ta)
apply(auto simp add: ta-bisim-def intro!: list-all2-all-nthI)
apply(case-tac {|ta|}_t ! n)
apply(simp-all)
apply(erule (1) new-thread-bisim0-extNTA2J-extNTA2J0)
apply(auto simp add: in-set-conv-nth)
done

lemma assumes wf: wwf-J-prog P
  shows red-red0-tabisim0:
  P,t ⊢ ⟨e, s⟩ −ta→ ⟨e', s'⟩ ⟹ ∃ ta'. extTA2J0 P,P,t ⊢ ⟨e, s⟩ −ta'→ ⟨e', s'⟩ ∧ ta-bisim0 ta ta'
  and reds-reds0-tabisim0:
  P,t ⊢ ⟨es, s⟩ [−ta→] ⟨es', s'⟩ ⟹ ∃ ta'. extTA2J0 P,P,t ⊢ ⟨es, s⟩ [−ta'→] ⟨es', s'⟩ ∧ ta-bisim0 ta ta'
proof(induct rule: red-reds.inducts)
  case (RedCallExternal s a T M Ts Tr D vs ta va h' ta' e' s')
  note red = ‹P,t ⊢ ⟨a•M(vs),hp s⟩ −ta→ext ⟨va,h'⟩›
  note T = ‹typeof-addr (hp s) a = ⌊T⌋›
```

from *T* ‹*P* ⊢ *class-type-of T sees M*: *Ts*→*Tr* = *Native in D*› *red*
have *extTA2J0 P*,*P*,*t* ⊢ ⟨*addr a*·*M*(*map Val vs*),*s*⟩ −*extTA2J0 P ta*→ ⟨*e*′,(*h*′, *lcl s*)⟩
by(*rule red-reds.RedCallExternal*)(*simp-all add*: ‹*e*′ = *extRet2J* (*addr a*·*M*(*map Val vs*)) *va*›)
moreover from ‹*ta*′ = *extTA2J P ta*› *T red wf*
have *ta-bisim0 ta*′ (*extTA2J0 P ta*) **by**(*auto intro*: *ta-bisim0-extNTA2J-extNTA2J0*)
ultimately show *?case* **unfolding** ‹*s*′ = (*h*′, *lcl s*)› **by** *blast*
next
case *RedTryFail* **thus** *?case* **by**(*force intro*: *red-reds.RedTryFail*)
qed(*fastforce intro*: *red-reds.intros simp add*: *ta-bisim-def ta-upd-simps*)+

lemma assumes *wf*: *wwf-J-prog P*
shows *red0-red-tabisim0*:
extTA2J0 P,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩ ⟹ ∃ *ta*′. *P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*′→ ⟨*e*′, *s*′⟩ ∧ *ta-bisim0 ta*′ *ta*
and *reds0-reds-tabisim0*:
extTA2J0 P,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩ ⟹ ∃ *ta*′. *P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*′→] ⟨*es*′, *s*′⟩ ∧ *ta-bisim0 ta*′ *ta*
proof(*induct rule*: *red-reds.inducts*)
case (*RedCallExternal s a T M Ts Tr D vs ta va h*′ *ta*′ *e*′ *s*′)
note *red* = ‹*P*,*t* ⊢ ⟨*a*·*M*(*vs*),*hp s*⟩ −*ta*→*ext* ⟨*va*,*h*′⟩›
note *T* = ‹*typeof-addr* (*hp s*) *a* = ⌊*T*⌋›
from *T* ‹*P* ⊢ *class-type-of T sees M*: *Ts*→*Tr* = *Native in D*› *red*
have *P*,*t* ⊢ ⟨*addr a*·*M*(*map Val vs*),*s*⟩ −*extTA2J P ta*→ ⟨*e*′,(*h*′, *lcl s*)⟩
by(*rule red-reds.RedCallExternal*)(*simp-all add*: ‹*e*′ = *extRet2J* (*addr a*·*M*(*map Val vs*)) *va*›)
moreover from ‹*ta*′ = *extTA2J0 P ta*› *T red wf*
have *ta-bisim0* (*extTA2J P ta*) *ta*′ **by**(*auto intro*: *ta-bisim0-extNTA2J-extNTA2J0*)
ultimately show *?case* **unfolding** ‹*s*′ = (*h*′, *lcl s*)› **by** *blast*
next
case *RedTryFail* **thus** *?case* **by**(*force intro*: *red-reds.RedTryFail*)
qed(*fastforce intro*: *red-reds.intros simp add*: *ta-bisim-def ta-upd-simps*)+

lemma *red-inline-call-red*:
assumes *red*: *P*,*t* ⊢ ⟨*e*, (*h*, *Map.empty*)⟩ −*ta*→ ⟨*e*′, (*h*′, *Map.empty*)⟩
shows *call E* = ⌊*aMvs*⌋ ⟹ *P*,*t* ⊢ ⟨*inline-call e E*, (*h*, *x*)⟩ −*ta*→ ⟨*inline-call e*′ *E*, (*h*′, *x*)⟩
(**is** - ⟹ *?concl E x*)

and
calls Es = ⌊*aMvs*⌋ ⟹ *P*,*t* ⊢ ⟨*inline-calls e Es*, (*h*, *x*)⟩ [−*ta*→] ⟨*inline-calls e*′ *Es*, (*h*′, *x*)⟩
(**is** - ⟹ *?concls Es x*)
proof(*induct E* **and** *Es arbitrary*: *x* **and** *x rule*: *call.induct calls.induct*)
case (*Call obj M pns x*)
note *IHobj* = ‹⋀*x*. *call obj* = ⌊*aMvs*⌋ ⟹ *?concl obj x*›
note *IHpns* = ‹⋀*x*. *calls pns* = ⌊*aMvs*⌋ ⟹ *?concls pns x*›
obtain *a M*′ *vs* **where** [*simp*]: *aMvs* = (*a*, *M*′, *vs*) **by**(*cases aMvs*, *auto*)
from ‹*call* (*obj*·*M*(*pns*)) = ⌊*aMvs*⌋› **have** *call* (*obj*·*M*(*pns*)) = ⌊(*a*, *M*′, *vs*)⌋ **by** *simp*
thus *?case*
proof(*induct rule*: *call-callE*)
case *CallObj*
with *IHobj*[*of x*] **show** *?case* **by**(*fastforce intro*: *red-reds.CallObj*)
next
case (*CallParams v*′′)
with *IHpns*[*of x*] **show** *?case* **by**(*fastforce intro*: *red-reds.CallParams*)
next
case *Call*
from *red-lcl-add*[*OF red*, **where** *?l0.0*=*x*]
have *P*,*t* ⊢ ⟨*e*,(*h*, *x*)⟩ −*ta*→ ⟨*e*′, (*h*′, *x*)⟩ **by** *simp*

```
    with Call show ?case by(fastforce dest: BlockRed)
  qed
next
  case (Block V T' vo exp x)
  note IH = ‹⋀x. call exp = ⌊aMvs⌋ ⟹ ?concl exp x›
  from IH[of x(V := vo)] ‹call {V:T'=vo; exp} = ⌊aMvs⌋›
  show ?case by(clarsimp simp del: fun-upd-apply)(drule BlockRed, auto)
next
  case (Cons-exp exp exps x)
  show ?case
  proof(cases is-val exp)
    case True
    with ‹calls (exp # exps) = ⌊aMvs⌋› have calls exps = ⌊aMvs⌋ by auto
    with ‹calls exps = ⌊aMvs⌋ ⟹ ?concls exps x› True
    show ?thesis by(fastforce intro: ListRed2)
  next
    case False
    with ‹calls (exp # exps) = ⌊aMvs⌋› have call exp = ⌊aMvs⌋ by auto
    with ‹call exp = ⌊aMvs⌋ ⟹ ?concl exp x›
    show ?thesis by(fastforce intro: ListRed1)
  qed
qed(fastforce intro: red-reds.intros)+

lemma
  assumes P ⊢ class-type-of T sees M:Us→U = ⌊(pns, body)⌋ in D length vs = length pns length Us
= length pns
  shows is-call-red-inline-call:
  ⟦ call e = ⌊(a, M, vs)⌋; typeof-addr (hp s) a = ⌊T⌋ ⟧
  ⟹ P,t ⊢ ⟨e, s⟩ −ε→ ⟨inline-call (blocks (this # pns) (Class D # Us) (Addr a # vs) body) e, s⟩
  (is - ⟹ - ⟹ ?red e s)
  and is-calls-reds-inline-calls:
  ⟦ calls es = ⌊(a, M, vs)⌋; typeof-addr (hp s) a = ⌊T⌋ ⟧
  ⟹ P,t ⊢ ⟨es, s⟩ [−ε→] ⟨inline-calls (blocks (this # pns) (Class D # Us) (Addr a # vs) body) es, s⟩
  (is - ⟹ - ⟹ ?reds es s)
proof(induct e and es arbitrary: s and s rule: call.induct calls.induct)
  case (Call obj M' params s)
  note IHObj = ‹⋀s. ⟦call obj = ⌊(a, M, vs)⌋; typeof-addr (hp s) a = ⌊T⌋ ⟧ ⟹ ?red obj s›
  note IHParams = ‹⋀s. ⟦ calls params = ⌊(a, M, vs)⌋; typeof-addr (hp s) a = ⌊T⌋ ⟧ ⟹ ?reds params
s›
  from ‹call (obj·M'(params)) = ⌊(a, M, vs)⌋›
  show ?case
  proof(induct rule: call-callE)
    case CallObj
    from IHObj[OF CallObj] ‹typeof-addr (hp s) a = ⌊T⌋› have ?red obj s by blast
    moreover from CallObj have ¬ is-val obj by auto
    ultimately show ?case by(auto intro: red-reds.CallObj)
  next
    case (CallParams v)
    from IHParams[OF ‹calls params = ⌊(a, M, vs)⌋›] ‹typeof-addr (hp s) a = ⌊T⌋›
    have ?reds params s by blast
    moreover from CallParams have ¬ is-vals params by auto
    ultimately show ?case using ‹obj = Val v› by(auto intro: red-reds.CallParams)
  next
    case Call
```

with *RedCall*[**where** *s=s*, *simplified*, *OF* ‹*typeof-addr* (*hp s*) *a* = ⌊*T*⌋› ‹*P* ⊢ *class-type-of T sees M*:*Us*→*U* = ⌊(*pns*, *body*)⌋ *in D*› ‹*length vs* = *length pns*› ‹*length Us* = *length pns*›]
show *?thesis* **by**(*simp*)
qed
next
case (*Block V ty vo exp s*)
note *IH* = ‹⋀*s*. ⟦*call exp* = ⌊(*a*, *M*, *vs*)⌋; *typeof-addr* (*hp s*) *a* = ⌊*T*⌋ ⟧ ⟹ *?red exp s*›
from ‹*call* {*V*:*ty*=*vo*; *exp*} = ⌊(*a*, *M*, *vs*)⌋› *IH*[*of* (*hp s*, (*lcl s*)(*V* := *vo*))] ‹*typeof-addr* (*hp s*) *a* = ⌊*T*⌋›
show *?case* **by**(*cases s*, *simp del*: *fun-upd-apply*)(*drule red-reds.BlockRed*, *simp*)
qed(*fastforce intro*: *red-reds.intros*)+

lemma *red-inline-call-red′*:
assumes *fv*: *fv ee* = {}
and *eefin*: ¬ *final ee*
shows ⟦ *call E* = ⌊*aMvs*⌋; *P*,*t* ⊢ ⟨*inline-call ee E*, (*h*, *x*)⟩ −*ta*→ ⟨*E′*, (*h′*, *x′*)⟩ ⟧
⟹ ∃*ee′*. *E′* = *inline-call ee′ E* ∧ *P*,*t* ⊢ ⟨*ee*, (*h*, *Map.empty*)⟩ −*ta*→ ⟨*ee′*, (*h′*, *Map.empty*)⟩ ∧ *x* = *x′*
(**is** ⟦ -; - ⟧ ⟹ *?concl E E′ x x′*)
and ⟦ *calls Es* = ⌊*aMvs*⌋; *P*,*t* ⊢ ⟨*inline-calls ee Es*, (*h*, *x*)⟩ [−*ta*→] ⟨*Es′*, (*h′*, *x′*)⟩ ⟧
⟹ ∃*ee′*. *Es′* = *inline-calls ee′ Es* ∧ *P*,*t* ⊢ ⟨*ee*, (*h*, *Map.empty*)⟩ −*ta*→ ⟨*ee′*, (*h′*, *Map.empty*)⟩ ∧ *x* = *x′*
(**is** ⟦ -; - ⟧ ⟹ *?concls Es Es′ x x′*)
proof(*induct E* **and** *Es arbitrary*: *E′ x x′* **and** *Es′ x x′ rule*: *call.induct calls.induct*)
case *new* **thus** *?case* **by** *simp*
next
case (*newArray T exp E′ x x′*)
thus *?case* **using** *eefin* **by**(*auto elim*!: *red-cases*)
next
case *Cast* **thus** *?case* **using** *eefin* **by**(*auto elim*!:*red-cases*)
next
case *InstanceOf* **thus** *?case* **using** *eefin* **by**(*auto elim*!:*red-cases*)
next
case *Val* **thus** *?case* **by** *simp*
next
case *Var* **thus** *?case* **by** *simp*
next
case *LAss*
thus *?case* **using** *eefin* **by**(*auto elim*!: *red-cases*)
next
case *BinOp*
thus *?case* **using** *eefin* **by**(*auto elim*!: *red-cases split*: *if-split-asm*)
next
case *AAcc*
thus *?case* **using** *eefin* **by**(*auto elim*!: *red-cases split*: *if-split-asm*)
next
case *AAss* **thus** *?case* **using** *eefin* **by**(*auto elim*!: *red-cases split*: *if-split-asm*)
next
case *ALen* **thus** *?case* **using** *eefin* **by**(*auto elim*!: *red-cases split*: *if-split-asm*)
next
case *FAcc* **thus** *?case* **using** *eefin* **by**(*auto elim*!: *red-cases*)
next
case *FAss* **thus** *?case* **using** *eefin* **by**(*auto elim*!: *red-cases split*: *if-split-asm*)
next

```
  case CompareAndSwap thus ?case using eefin by(auto elim!: red-cases split: if-split-asm)
next
  case (Call obj M pns E' x x')
  note IHobj = ‹⋀x E' x'. ⟦call obj = ⌊aMvs⌋; P,t ⊢ ⟨inline-call ee obj,(h, x)⟩ −ta→ ⟨E',(h', x')⟩⟧
        ⟹ ?concl obj E' x x'›
  note IHpns = ‹⋀Es' x x'. ⟦calls pns = ⌊aMvs⌋; P,t ⊢ ⟨inline-calls ee pns,(h, x)⟩ [−ta→] ⟨Es',(h',
x')⟩⟧
        ⟹ ?concls pns Es' x x'›
  note red = ‹P,t ⊢ ⟨inline-call ee (obj·M(pns)),(h, x)⟩ −ta→ ⟨E',(h', x')⟩›
  obtain a M' vs where [simp]: aMvs = (a, M', vs) by(cases aMvs, auto)
  from ‹call (obj·M(pns)) = ⌊aMvs⌋› have call (obj·M(pns)) = ⌊(a,M',vs)⌋ by simp
  thus ?case
  proof(cases rule: call-callE)
    case CallObj
    hence ¬ is-val obj by auto
    with red CallObj eefin obtain obj' where E' = obj'·M(pns)
      and red': P,t ⊢ ⟨inline-call ee obj,(h, x)⟩ −ta→ ⟨obj',(h', x')⟩
      by(auto elim!: red-cases)
    from IHobj[OF - red'] CallObj obtain ee'
      where inline-call ee' obj = obj' x = x'
      and P,t ⊢ ⟨ee,(h, Map.empty)⟩ −ta→ ⟨ee',(h', Map.empty)⟩ by(auto simp del: fun-upd-apply)
    with ‹E' = obj'·M(pns)› CallObj red' show ?thesis by(fastforce simp del: fun-upd-apply)
  next
    case (CallParams v'')
    hence ¬ is-vals pns by auto
    with red CallParams eefin obtain pns' where E' = obj·M(pns')
      and red': P,t ⊢ ⟨inline-calls ee pns,(h, x)⟩ [−ta→] ⟨pns',(h', x')⟩
      by(auto elim!: red-cases)
    from IHpns[OF - red'] CallParams obtain ee'
      where inline-calls ee' pns = pns' x = x'
      and P,t ⊢ ⟨ee,(h, Map.empty)⟩ −ta→ ⟨ee',(h', Map.empty)⟩
      by(auto simp del: fun-upd-apply)
    with ‹E' = obj·M(pns')› CallParams red' ‹¬ is-vals pns›
    show ?thesis by(auto simp del: fun-upd-apply)
  next
    case Call
    with red have red': P,t ⊢ ⟨ee,(h, x)⟩ −ta→ ⟨E',(h', x')⟩ by(auto)
    from red-lcl-sub[OF red', of {}] fv
    have P,t ⊢ ⟨ee,(h, Map.empty)⟩ −ta→ ⟨E',(h', Map.empty)⟩ by simp
    moreover have x' = x
    proof(rule ext)
      fix V
      from red-notfree-unchanged[OF red', of V] fv
      show x' V = x V by simp
    qed
    ultimately show ?thesis using Call by simp
  qed
next
  case (Block V ty voo exp E' x x')
  note IH = ‹⋀x E' x'. ⟦call exp = ⌊aMvs⌋; P,t ⊢ ⟨inline-call ee exp,(h, x)⟩ −ta→ ⟨E',(h', x')⟩⟧
        ⟹ ?concl exp E' x x'›
  from ‹call {V:ty=voo; exp} = ⌊aMvs⌋› have ic: call exp = ⌊aMvs⌋ by simp
  note red = ‹P,t ⊢ ⟨inline-call ee {V:ty=voo; exp},(h, x)⟩ −ta→ ⟨E',(h', x')⟩›
  hence P,t ⊢ ⟨{V:ty=voo; inline-call ee exp},(h, x)⟩ −ta→ ⟨E',(h', x')⟩ by simp
```

```
  with ic eefin obtain exp' x'' where E' = {V:ty=x'' V; exp'}
    and red': P,t ⊢ ⟨inline-call ee exp,(h, fun-upd x V voo)⟩ −ta→ ⟨exp',(h', x'')⟩
    and x' = fun-upd x'' V (x V)
    by −(erule red.cases,auto dest: inline-call-eq-Val)
  from IH[OF ic red'] obtain ee' vo'
    where icl: inline-call ee' exp = exp' x'' = fun-upd x V voo
    and red'': P,t ⊢ ⟨ee,(h, Map.empty)⟩ −ta→ ⟨ee',(h', Map.empty)⟩ by blast
  from ‹x'' = fun-upd x V voo› have x'' V = voo by(simp add: fun-eq-iff)
  with icl red'' ‹E' = {V:ty=x'' V; exp'}› ‹x' = fun-upd x'' V (x V)› red'
  show ?case by(auto simp del: fun-upd-apply)
next
  case Synchronized thus ?case using eefin by(auto elim!: red-cases)
next
  case InSynchronized thus ?case using eefin by(auto elim!: red-cases)
next
  case Seq
  thus ?case using eefin by(auto elim!: red-cases)
next
  case Cond thus ?case using eefin by(auto elim!: red-cases)
next
  case While thus ?case by simp
next
  case throw
  thus ?case using eefin by(auto elim!: red-cases)
next
  case TryCatch
  thus ?case using eefin by(auto elim!: red-cases)
next
  case Nil-exp thus ?case by simp
next
  case Cons-exp
  thus ?case using eefin by(auto elim!: reds-cases split: if-split-asm)
qed

lemma assumes sees: P ⊢ class-type-of T sees M:Us→U = ⌊(pns, body)⌋ in D
  shows is-call-red-inline-callD:
  ⟦ P,t ⊢ ⟨e, s⟩ −ta→ ⟨e', s'⟩; call e = ⌊(a, M, vs)⌋; typeof-addr (hp s) a = ⌊T⌋ ⟧
  ⟹ e' = inline-call (blocks (this # pns) (Class D # Us) (Addr a # vs) body) e
  and is-calls-reds-inline-callsD:
  ⟦ P,t ⊢ ⟨es, s⟩ [−ta→] ⟨es', s'⟩; calls es = ⌊(a, M, vs)⌋; typeof-addr (hp s) a = ⌊T⌋ ⟧
  ⟹ es' = inline-calls (blocks (this # pns) (Class D # Us) (Addr a # vs) body) es
proof(induct rule: red-reds.inducts)
  case RedCall with sees show ?case by(auto dest: sees-method-fun)
next
  case RedCallExternal
  with sees show ?case by(auto dest: sees-method-fun)
next
  case (BlockRed e h x V vo ta e' h' x' T')
  from ‹call {V:T'=vo; e} = ⌊(a, M, vs)⌋› ‹typeof-addr (hp (h, x)) a = ⌊T⌋› sees
  have call e = ⌊(a, M, vs)⌋ and ¬ synthesized-call P h (a, M, vs)
    by(auto simp add: synthesized-call-conv dest: sees-method-fun)
  with ‹P,t ⊢ ⟨e,(h, x(V := vo))⟩ −ta→ ⟨e',(h', x')⟩›
  have x(V := vo) = x' by(auto dest: is-call-red-state-unchanged)
  hence x' V = vo by auto
```

with *BlockRed* **show** *?case* **by**(*simp*)
qed(*fastforce split*: *if-split-asm*)+

lemma (**in** −) *wf-state-ConsD*: *wf-state* (*e*, *e′* # *es*) ⟹ *wf-state* (*e′*, *es*)
by(*simp*)

lemma *red-fold-exs*:
⟦ *P*,*t* ⊢ ⟨*e*,(*h*, *Map.empty*)⟩ −*ta*→ ⟨*e′*,(*h′*, *Map.empty*)⟩; *wf-state* (*e*, *es*) ⟧
⟹ *P*,*t* ⊢ ⟨*collapse* (*e*, *es*), (*h*, *Map.empty*)⟩ −*ta*→ ⟨*collapse* (*e′*, *es*), (*h′*, *Map.empty*)⟩
(**is** ⟦ -; - ⟧ ⟹ *?concl e e′ es*)
proof(*induction es arbitrary*: *e e′*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons E es*)
note *wf* = ‹*wf-state* (*e*, *E* # *es*)›
note *red* = ‹*P*,*t* ⊢ ⟨*e*,(*h*, *Map.empty*)⟩ −*ta*→ ⟨*e′*,(*h′*, *Map.empty*)⟩›
from *wf* **obtain** *a M vs arrobj* **where** *call*: *call E* = ⌊(*a*, *M*, *vs*)⌋
by(*auto simp add*: *is-call-def*)
from *red call* **have** *P*,*t* ⊢ ⟨*inline-call e E*,(*h*, *Map.empty*)⟩ −*ta*→ ⟨*inline-call e′ E*,(*h′*, *Map.empty*)⟩
by(*rule red-inline-call-red*)
hence *P*,*t* ⊢ ⟨*collapse* (*inline-call e E*, *es*),(*h*, *Map.empty*)⟩ −*ta*→ ⟨*collapse* (*inline-call e′ E*, *es*),(*h′*, *Map.empty*)⟩
proof(*rule Cons.IH*)
from *wf* **have** *fv E* = {} *fv e* = {} **by** *auto*
with *fv-inline-call*[*of e E*] **have** *fv* (*inline-call e E*) = {} **by** *auto*
thus *wf-state* (*inline-call e E*, *es*) **using** *wf* **by** *auto*
qed
thus *?case* **by** *simp*
qed

lemma *red-fold-exs′*:
⟦ *P*,*t* ⊢ ⟨*collapse* (*e*, *es*), (*h*, *Map.empty*)⟩ −*ta*→ ⟨*e′*, (*h′*, *x′*)⟩; *wf-state* (*e*, *es*); ¬ *final e* ⟧
⟹ ∃ *E′*. *e′* = *collapse* (*E′*, *es*) ∧ *P*,*t* ⊢ ⟨*e*, (*h*, *Map.empty*)⟩ −*ta*→ ⟨*E′*, (*h′*, *Map.empty*)⟩
(**is** ⟦ -; -; - ⟧ ⟹ *?concl e es*)
proof(*induction es arbitrary*: *e*)
case *Nil*
hence *red′*: *P*,*t* ⊢ ⟨*e*,(*h*, *Map.empty*)⟩ −*ta*→ ⟨*e′*,(*h′*, *x′*)⟩ **by** *simp*
with *red-dom-lcl*[*OF this*] ‹*wf-state* (*e*, [])› **show** *?case* **by** *auto*
next
case (*Cons E es*)
note *wf* = ‹*wf-state* (*e*, *E* # *es*)›
note *nfin* = ‹¬ *final e*›
from *wf* **have** *fv e* = {} **by** *simp*
from *wf* **obtain** *a M vs* **where** *call*: *call E* = ⌊(*a*, *M*, *vs*)⌋ **by**(*auto simp add*: *is-call-def*)
from ‹*P*,*t* ⊢ ⟨*collapse* (*e*, *E* # *es*),(*h*, *Map.empty*)⟩ −*ta*→ ⟨*e′*,(*h′*, *x′*)⟩›
have *P*,*t* ⊢ ⟨*collapse* (*inline-call e E*, *es*),(*h*, *Map.empty*)⟩ −*ta*→ ⟨*e′*,(*h′*, *x′*)⟩ **by** *simp*
moreover from *wf fv-inline-call*[*of e E*] **have** *wf-state* (*inline-call e E*, *es*) **by** *auto*
moreover from *nfin call* **have** ¬ *final* (*inline-call e E*) **by**(*auto elim*!: *final.cases*)
ultimately have *?concl* (*inline-call e E*) *es* **by**(*rule Cons.IH*)
then obtain *E′* **where** *e′*: *e′* = *collapse* (*E′*, *es*)
and *red′*: *P*,*t* ⊢ ⟨*inline-call e E*,(*h*, *Map.empty*)⟩ −*ta*→ ⟨*E′*,(*h′*, *Map.empty*)⟩ **by** *blast*
from *red-inline-call-red′*(*1*)[*OF* ‹*fv e* = {}› *nfin* ‹*call E* = ⌊(*a*, *M*, *vs*)⌋› *red′*]
obtain *e′* **where** *E′* = *inline-call e′ E P*,*t* ⊢ ⟨*e*,(*h*, *Map.empty*)⟩ −*ta*→ ⟨*e′*,(*h′*, *Map.empty*)⟩ **by** *auto*
thus *?case* **using** *e′* **by** *auto*

qed

lemma *τRed0r-inline-call-not-final*:
  $\exists\, e'\ es'.\ \tau Red0r\ P\ t\ h\ (e,\ es)\ (e',\ es') \wedge (final\ e' \longrightarrow es' = []) \wedge collapse\ (e,\ es) = collapse\ (e',\ es')$
proof(*induct es arbitrary*: *e*)
  case *Nil* **thus** *?case* **by** *blast*
next
  case (*Cons e es E*) **show** *?case*
  proof(*cases final E*)
    case *True*
    hence *τRed0 P t h* (*E*, *e # es*) (*inline-call E e*, *es*) **by**(*auto intro*: *red0Return*)
    moreover from *Cons*[*of inline-call E e*] **obtain** *e' es'*
      where *τRed0r P t h* (*inline-call E e*, *es*) (*e'*, *es'*) *final e' ⟶ es' = []*
      collapse (*inline-call E e*, *es*) = *collapse* (*e'*, *es'*) **by** *blast*
    ultimately show *?thesis* **unfolding** *collapse.simps* **by**(*blast intro*: *converse-rtranclp-into-rtranclp*)
  qed *blast*
qed

lemma *τRed0-preserves-wf-state*:
  ⟦ *wwf-J-prog P*; *τRed0 P t h* (*e*, *es*) (*e'*, *es'*); *wf-state* (*e*, *es*) ⟧ ⟹ *wf-state* (*e'*, *es'*)
by(*auto simp del*: *wf-state-iff intro*: *red0-preserves-wf-state*)

lemma *τRed0r-preserves-wf-state*:
  assumes *wf*: *wwf-J-prog P*
  shows ⟦ *τRed0r P t h* (*e*, *es*) (*e'*, *es'*); *wf-state* (*e*, *es*) ⟧ ⟹ *wf-state* (*e'*, *es'*)
by(*induct rule*: *rtranclp-induct2*)(*blast intro*: *τRed0-preserves-wf-state*[*OF wf*])+

lemma *τRed0t-preserves-wf-state*:
  assumes *wf*: *wwf-J-prog P*
  shows ⟦ *τRed0t P t h* (*e*, *es*) (*e'*, *es'*); *wf-state* (*e*, *es*) ⟧ ⟹ *wf-state* (*e'*, *es'*)
by(*induct rule*: *tranclp-induct2*)(*blast intro*: *τRed0-preserves-wf-state*[*OF wf*])+

lemma *collapse-τmove0-inv*:
  ⟦ ∀ *e*∈*set es*. *is-call e*; ¬ *final e* ⟧ ⟹ *τmove0 P h* (*collapse* (*e*, *es*)) = *τmove0 P h e*
proof(*induction es arbitrary*: *e*)
  case *Nil* **thus** *?case* **by** *clarsimp*
next
  case (*Cons e es e''*)
  from ‹∀ *a*∈*set* (*e # es*). *is-call a*› **obtain** *aMvs* **where** *calls*: ∀ *e*∈*set es*. *is-call e*
    and *call*: *call e* = ⌊*aMvs*⌋ **by**(*auto simp add*: *is-call-def*)
  note *calls* **moreover**
  from ‹¬ *final e''*› *call* **have** ¬ *final* (*inline-call e'' e*) **by**(*auto simp add*: *final-iff*)
  ultimately have *τmove0 P h* (*collapse* (*inline-call e'' e*, *es*)) = *τmove0 P h* (*inline-call e'' e*)
    by(*rule Cons.IH*)
  also from *call* ‹¬ *final e''*› **have** ... = *τmove0 P h e''* **by**(*auto simp add*: *inline-call-τmove0-inv*)
  finally show *?case* **by** *simp*
qed

lemma *τRed0r-into-silent-moves*:
  τRed0r P t h (*e*, *es*) (*e'*, *es'*) ⟹ *red0-mthr.silent-moves P t* ((*e*, *es*), *h*) ((*e'*, *es'*), *h*)
by(*induct rule*: *rtranclp-induct2*)(*fastforce intro*: *τtrsys.silent-move.intros elim*!: *rtranclp.rtrancl-into-rtrancl*)+

lemma *τRed0t-into-silent-movet*:
  τRed0t P t h (*e*, *es*) (*e'*, *es'*) ⟹ *red0-mthr.silent-movet P t* ((*e*, *es*), *h*) ((*e'*, *es'*), *h*)

by(*induct rule*: *tranclp-induct2*)(*fastforce intro*: *τtrsys.silent-move.intros elim*!: *tranclp.trancl-into-trancl*)+

lemma *red-simulates-red0*:
assumes *wwf*: *wwf-J-prog P*
and *sim*: *bisim-red-red0 s1 s2 mred0 P t s2 ta2 s2' ¬ τMOVE0 P s2 ta2 s2'*
shows ∃ *s1' ta1*. *mred P t s1 ta1 s1' ∧ ¬ τMOVE P s1 ta1 s1' ∧ bisim-red-red0 s1' s2' ∧ ta-bisim0 ta1 ta2*
proof −
note *sim*
moreover obtain *e1 h1 x1* **where** *s1*: *s1* = ((*e1*, *x1*), *h1*) **by**(*cases s1*, *auto*)
moreover obtain *e' es' h2'* **where** *s2'*: *s2'* = ((*e'*, *es'*), *h2'*) **by**(*cases s2'*, *auto*)
moreover obtain *e es h2* **where** *s2*: *s2* = ((*e*, *es*), *h2*) **by**(*cases s2*, *auto*)
ultimately have *bisim*: *bisim-red-red0* ((*e1*, *x1*), *h1*) ((*e*, *es*), *h2*)
and *red*: *P*,*t* ⊢*0* ⟨*e*/*es*, *h2*⟩ −*ta2*→ ⟨*e'*/*es'*, *h2'*⟩
and *τ*: ¬ *τmove0 P h2 e* ∧ ¬ *final e* ∨ *ta2* ≠ *ε* **by** *auto*
from *red τ* **have** *τ*: ¬ *τmove0 P h2 e* **and** *nfin*: ¬ *final e*
by(*cases*, *auto dest*: *red-τ-taD*[**where** *extTA*=*extTA2J0 P*, *OF extTA2J0-ε*])+
from *bisim* **have** *heap*: *h1* = *h2* **and** *fold*: *e1* = *collapse* (*e*, *es*)
and *x1*: *x1* = *Map.empty* **and** *wf-state*: *wf-state* (*e*, *es*)
by(*auto elim*!: *bisim-red-red0.cases*)
from *red wf-state* **have** *wf-state'*: *wf-state* (*e'*, *es'*) **by**(*rule red0-preserves-wf-state*[*OF wwf*])
from *red* **show** *?thesis*
proof(*cases*)
case (*red0Red xs'*)
hence [*simp*]: *es'* = *es*
and *extTA2J0 P*,*P*,*t* ⊢ ⟨*e*, (*h2*, *Map.empty*)⟩ −*ta2*→ ⟨*e'*, (*h2'*, *xs'*)⟩ **by** *auto*
from *red0-red-tabisim0*[*OF wwf this*(*2*)] **obtain** *ta1*
where *red'*: *P*,*t* ⊢ ⟨*e*,(*h2*, *Map.empty*)⟩ −*ta1*→ ⟨*e'*,(*h2'*, *xs'*)⟩
and *tasim*: *ta-bisim0 ta1 ta2* **by** *auto*
moreover from *wf-state* **have** *fv e* = {} **by** *auto*
with *red-dom-lcl*[*OF red'*] *red-fv-subset*[*OF wwf red'*] **have** *xs'* = *Map.empty* **by** *auto*
ultimately have *P*,*t* ⊢ ⟨*e*,(*h2*, *Map.empty*)⟩ −*ta1*→ ⟨*e'*,(*h2'*, *Map.empty*)⟩ **by** *simp*
with *wf-state* **have** *P*,*t* ⊢ ⟨*collapse* (*e*, *es*),(*h2*, *Map.empty*)⟩ −*ta1*→ ⟨*collapse* (*e'*, *es*),(*h2'*, *Map.empty*)⟩
by −(*erule red-fold-exs*, *auto*)
moreover from *τ wf-state fold nfin* **have** ¬ *τmove0 P h2 e1* **by**(*auto simp add*: *collapse-τmove0-inv*)
hence ¬ *τMOVE P* ((*collapse* (*e*, *es*), *Map.empty*), *h2*) *ta1* ((*collapse* (*e'*, *es*), *Map.empty*), *h2'*)
unfolding *fold* **by** *auto*
moreover from *wf-state'* **have** *bisim-red-red0* ((*collapse* (*e'*, *es*), *Map.empty*), *h2'*) *s2'*
unfolding *s2'* **by**(*auto*)
ultimately show *?thesis* **unfolding** *heap s1 s2 s2' fold x1*
using *tasim* **by**(*fastforce intro*!: *exI rtranclp.rtrancl-refl*)
next
case *red0Call*
with *τ* **have** *False*
by(*auto simp add*: *synthesized-call-def τexternal'-def dest*!: *τmove0-callD*[**where** *P*=*P* **and** *h*=*h2*] *dest*: *sees-method-fun*)
thus *?thesis* **..**
next
case *red0Return* **with** *nfin* **show** *?thesis* **by** *simp*
qed
qed

lemma *delay-bisimulation-measure-red-red0*:

assumes *wf*: *wwf-J-prog P*
shows *delay-bisimulation-measure* (*mred P t*) (*mred0 P t*) *bisim-red-red0 ta-bisim0* (*τMOVE P*) (*τMOVE0 P*) (λ*e e'*. *False*) (λ((*e*, *es*), *h*) ((*e*, *es'*), *h*). *length es* < *length es'*)
proof
show *wfP* (λ*e e'*. *False*) **by** *auto*
next
have *wf* {(*x* :: *nat*, *y*). *x* < *y*} **by**(*rule wf-less*)
hence *wf* (*inv-image* {(*x* :: *nat*, *y*). *x* < *y*} (*length o snd o fst*)) **by**(*rule wf-inv-image*)
also have *inv-image* {(*x* :: *nat*, *y*). *x* < *y*} (*length o snd o fst*) = {(*x*, *y*). (λ((*e*, *es*), *h*) ((*e*, *es'*), *h*). *length es* < *length es'*) *x y*} **by** *auto*
finally show *wfP* (λ((*e*, *es*), *h*) ((*e*, *es'*), *h*). *length es* < *length es'*)
unfolding *wfp-def* **.**
next
fix *s1 s2 s1'*
assume *bisim-red-red0 s1 s2* **and** *red-mthr.silent-move P t s1 s1'*
moreover obtain *e1 h1 x1* **where** *s1*: *s1* = ((*e1*, *x1*), *h1*) **by**(*cases s1*, *auto*)
moreover obtain *e1' h1' x1'* **where** *s1'*: *s1'* = ((*e1'*, *x1'*), *h1'*) **by**(*cases s1'*, *auto*)
moreover obtain *e es h2* **where** *s2*: *s2* = ((*e*, *es*), *h2*) **by**(*cases s2*, *auto*)
ultimately have *bisim*: *bisim-red-red0* ((*e1*, *x1*), *h1*) ((*e*, *es*), *h2*)
and *red*: *P*,*t* ⊢ ⟨*e1*, (*h1*, *x1*)⟩ −*ε*→ ⟨*e1'*, (*h1'*, *x1'*)⟩
and *τ*: *τmove0 P h1 e1* **by**(*auto elim*: *τtrsys.silent-move.cases*)
from *bisim* **have** *heap*: *h1* = *h2*
and *fold*: *e1* = *collapse* (*e*, *es*)
and *x1*: *x1* = *Map.empty*
and *wf-state*: *wf-state* (*e*, *es*)
by(*auto elim*!: *bisim-red-red0.cases*)
from *τRed0r-inline-call-not-final*[*of P t h1 e es*]
obtain *e' es'* **where** *red1*: *τRed0r P t h1* (*e*, *es*) (*e'*, *es'*)
and *final e'* ⟹ *es'* = []
and *feq*: *collapse* (*e*, *es*) = *collapse* (*e'*, *es'*) **by** *blast*
have *nfin*: ¬ *final e'*
proof
assume *fin*: *final e'*
hence *es'* = [] **by**(*rule* ‹*final e'* ⟹ *es'* = []›)
with *fold fin feq* **have** *final e1* **by** *simp*
with *red* **show** *False* **by** *auto*
qed
from *red1 wf-state* **have** *wf-state'*: *wf-state* (*e'*, *es'*) **by**(*rule τRed0r-preserves-wf-state*[*OF wf*])
hence *fv*: *fvs* (*e'* # *es'*) = {} **and** *icl*: ∀ *e*∈*set es'*. *is-call e* **by** *auto*
from *red-fold-exs'*[*OF red*[*unfolded fold x1 feq*] *wf-state' nfin*]
obtain *E'* **where** *e1'*: *e1'* = *collapse* (*E'*, *es'*)
and *red'*: *P*,*t* ⊢ ⟨*e'*,(*h1*, *Map.empty*)⟩ −*ε*→ ⟨*E'*,(*h1'*, *Map.empty*)⟩ **by** *auto*
from *fv fv-collapse*[*of es e*] *wf-state fold feq* **have** *fv e1* = {} **by**(*auto*)
with *red-dom-lcl*[*OF red*] *x1* **have** *x1'*: *x1'* = *Map.empty* **by** *simp*
from *red-red0-tabisim0*[*OF wf red'*]
have *red''*: *extTA2J0 P*,*P*,*t* ⊢ ⟨*e'*,(*h1*, *Map.empty*)⟩ −*ε*→ ⟨*E'*,(*h1'*, *Map.empty*)⟩ **by** *simp*
show *bisim-red-red0 s1' s2* ∧ (λ*e e'*. *False*)^++ *s1' s1* ∨
(∃ *s2'*. *red0-mthr.silent-movet P t s2 s2'* ∧ *bisim-red-red0 s1' s2'*)
proof(*cases no-call P h1 e'*)
case *True*
with *red''* **have** *P*,*t* ⊢*0* ⟨*e'*/*es'*, *h1*⟩ −*ε*→ ⟨*E'*/*es'*, *h1'*⟩ **unfolding** *no-call-def* **by**(*rule red0Red*)
moreover from *red τ* **have** [*simp*]: *h1'* = *h1* **by**(*auto dest*: *τmove0-heap-unchanged*)
moreover from *τ fold feq icl nfin* **have** *τmove0 P h1 e'* **by**(*simp add*: *collapse-τmove0-inv*)
ultimately have *τRed0 P t h1* (*e'*, *es'*) (*E'*, *es'*) **using** ‹*τmove0 P h1 e'*› **by** *auto*

```
    with red1 have τRed0t P t h1 (e, es) (E', es') by(rule rtranclp-into-tranclp1)
    moreover hence wf-state (E', es') using wf-state by(rule τRed0t-preserves-wf-state[OF wf])
    hence bisim-red-red0 ((e1', x1'), h1) ((E', es'), h1) unfolding x1' e1' by(auto)
    ultimately show ?thesis using s1 s1' s2 heap by simp(blast intro: τRed0t-into-silent-movet)
  next
    case False
    then obtain a M vs where call: call e' = ⌊(a, M, vs)⌋
      and notsynth: ¬ synthesized-call P h1 (a, M, vs) by(auto simp add: no-call-def)
    from notsynth called-methodD[OF red'' call] obtain T D Us U pns body
      where h1' = h1
      and ha: typeof-addr h1 a = ⌊T⌋
      and sees: P ⊢ class-type-of T sees M: Us→U = ⌊(pns, body)⌋ in D
      and length: length vs = length pns length Us = length pns
      by(auto)
    let ?e = blocks (this # pns) (Class D # Us) (Addr a # vs) body
    from call ha have P,t ⊢0 ⟨e'/es',h1⟩ −ε→ ⟨?e/e' # es',h1⟩
      using sees length by(rule red0Call)
    moreover from τ fold feq icl nfin False have τmove0 P h1 e' by(simp add: collapse-τmove0-inv)
    ultimately have τRed0 P t h1 (e', es') (?e, e' # es') by auto
    with red1 have τRed0t P t h1 (e, es) (?e, e' # es') by(rule rtranclp-into-tranclp1)
    moreover {
      from ‹P,t ⊢0 ⟨e'/es',h1⟩ −ε→ ⟨?e/e' # es',h1⟩› have wf-state (?e, e' # es')
        using wf-state' by(rule red0-preserves-wf-state[OF wf])
      moreover from is-call-red-inline-callD[OF sees red' call] ha
      have E' = inline-call ?e e' by auto
      ultimately have bisim-red-red0 s1' ((?e, e' # es'), h1') unfolding s1' e1' x1'
        by(auto del: wf-state.cases wf-state.intros) }
    moreover from red' call notsynth have h1 = h1'
      by(auto dest: is-call-red-state-unchanged)
    ultimately show ?thesis unfolding heap x1' x1 s2 s1' ‹h1' = h1›
      by(blast intro: τRed0t-into-silent-movet)
  qed
next
  fix s1 s2 s2'
  assume bisim-red-red0 s1 s2 and red0-mthr.silent-move P t s2 s2'
  moreover obtain e1 h1 x1 where s1: s1 = ((e1, x1), h1) by(cases s1, auto)
  moreover obtain e' es' h2' where s2': s2' = ((e', es'), h2') by(cases s2', auto)
  moreover obtain e es h2 where s2: s2 = ((e, es), h2) by(cases s2, auto)
  ultimately have bisim: bisim-red-red0 ((e1, x1), h1) ((e, es), h2)
    and red: P,t ⊢0 ⟨e/es, h2⟩ −ε→ ⟨e'/es', h2'⟩
    and τ: τmove0 P h2 e ∨ final e by(auto elim: τtrsys.silent-move.cases)
  from bisim have heap: h1 = h2
    and fold: e1 = collapse (e, es)
    and x1: x1 = Map.empty and wf-state: wf-state (e, es)
    by(auto elim!: bisim-red-red0.cases)
  from red wf-state have wf-state': wf-state (e', es') by(rule red0-preserves-wf-state[OF wf])
  from red show bisim-red-red0 s1 s2' ∧ (λ((e, es), h) ((e, es'), h). length es < length es')^++ s2' s2
∨
      (∃ s1'. red-mthr.silent-movet P t s1 s1' ∧ bisim-red-red0 s1' s2')
  proof cases
    case (red0Red xs')
    hence [simp]: es' = es
      and extTA2J0 P,P,t ⊢ ⟨e, (h2, Map.empty)⟩ −ε→ ⟨e', (h2', xs')⟩ by auto
    from red0-red-tabisim0[OF wf this(2)] have red': P,t ⊢ ⟨e,(h2, Map.empty)⟩ −ε→ ⟨e',(h2', xs')⟩
```

by *auto*
moreover from *wf-state* **have** *fv e* = {} **by** *auto*
with *red-dom-lcl*[*OF red'*] *red-fv-subset*[*OF wf red'*] **have** *xs'* = *Map.empty* **by** *auto*
ultimately have *P*,*t* ⊢ ⟨*e*,(*h2*, *Map.empty*)⟩ −*ε*→ ⟨*e'*,(*h2'*, *Map.empty*)⟩ **by** *simp*
hence *P*,*t* ⊢ ⟨*collapse* (*e*, *es*),(*h2*, *Map.empty*)⟩ −*ε*→ ⟨*collapse* (*e'*, *es*),(*h2'*, *Map.empty*)⟩
using *wf-state* **by**(*rule red-fold-exs*)
moreover from *red'* **have** ¬ *final e* **by** *auto*
with *τ wf-state fold* **have** *τmove0 P h2 e1* **by**(*auto simp add*: *collapse-τmove0-inv*)
ultimately have *red-mthr.silent-movet P t s1* ((*collapse* (*e'*, *es*), *Map.empty*), *h2'*)
using *s1 fold τ x1 heap* **by**(*auto intro*: *τtrsys.silent-move.intros*)
moreover from *wf-state'* **have** *bisim-red-red0* ((*collapse* (*e'*, *es*), *Map.empty*), *h2'*) *s2'*
unfolding *s2'* **by**(*auto*)
ultimately show *?thesis* **by** *blast*
next
case (*red0Call a M vs U Ts T pns body D*)
hence [*simp*]: *es'* = *e* # *es h2'* = *h2 e'* = *blocks* (*this* # *pns*) (*Class D* # *Ts*) (*Addr a* # *vs*) *body*
and *call*: *call e* = ⌊(*a*, *M*, *vs*)⌋
and *ha*: *typeof-addr h2 a* = ⌊*U*⌋
and *sees*: *P* ⊢ *class-type-of U sees M*: *Ts*→*T* = ⌊(*pns*, *body*)⌋ *in D*
and *len*: *length vs* = *length pns length Ts* = *length pns* **by** *auto*
from *is-call-red-inline-call*(*1*)[*OF sees len call*, *of* (*h2*, *Map.empty*)] *ha*
have *P*,*t* ⊢ ⟨*e*,(*h2*, *Map.empty*)⟩ −*ε*→ ⟨*inline-call e' e*, (*h2*, *Map.empty*)⟩ **by** *simp*
hence *P*,*t* ⊢ ⟨*collapse* (*e*, *es*), (*h2*, *Map.empty*)⟩ −*ε*→ ⟨*collapse* (*inline-call e' e*, *es*), (*h2*, *Map.empty*)⟩
using *wf-state* **by**(*rule red-fold-exs*)
moreover from *call ha wf-state τ* **have** *τmove0 P h2* (*collapse* (*e*, *es*))
by(*subst collapse-τmove0-inv*) *auto*
hence *τMOVE P* ((*collapse* (*e*, *es*), *Map.empty*), *h2*) *ε* ((*collapse* (*inline-call e' e*, *es*), *Map.empty*), *h2*) **by** *auto*
moreover from *wf-state'*
have *bisim-red-red0* ((*collapse* (*inline-call e' e*, *es*), *Map.empty*), *h2*) ((*e'*, *es'*), *h2'*)
by(*auto*)
ultimately show *?thesis* **unfolding** *s1 s2 s2' fold heap x1* **by**(*fastforce*)
next
case (*red0Return E*)
hence [*simp*]: *es* = *E* # *es' e'* = *inline-call e E h2'* = *h2* **by** *auto*
from *fold wf-state'*
have *bisim-red-red0* ((*e1*, *Map.empty*), *h1*) ((*inline-call e E*, *es'*), *h2*)
unfolding *heap* **by**(*auto*)
thus *?thesis* **using** *s1 s2' s2 x1* **by** *auto*
qed
next
fix *s1 s2 ta1 s1'*
assume *bisim-red-red0 s1 s2* **and** *mred P t s1 ta1 s1'* **and** ¬ *τMOVE P s1 ta1 s1'*
moreover obtain *e1 h1 x1* **where** *s1*: *s1* = ((*e1*, *x1*), *h1*) **by**(*cases s1*, *auto*)
moreover obtain *e1' h1' x1'* **where** *s1'*: *s1'* = ((*e1'*, *x1'*), *h1'*) **by**(*cases s1'*, *auto*)
moreover obtain *e es h2* **where** *s2*: *s2* = ((*e*, *es*), *h2*) **by**(*cases s2*, *auto*)
ultimately have *bisim*: *bisim-red-red0* ((*e1*, *x1*), *h1*) ((*e*, *es*), *h2*)
and *red*: *P*,*t* ⊢ ⟨*e1*, (*h1*, *x1*)⟩ −*ta1*→ ⟨*e1'*, (*h1'*, *x1'*)⟩
and *τ*: ¬ *τmove0 P h1 e1* **by**(*auto dest*: *red-τ-taD*[**where** *extTA*=*extTA2J P*, *OF extTA2J-ε*])
from *bisim* **have** *heap*: *h1* = *h2*
and *fold*: *e1* = *collapse* (*e*, *es*)
and *x1*: *x1* = *Map.empty*
and *wf-state*: *wf-state* (*e*, *es*)

```
    by(auto elim!: bisim-red-red0.cases)
  from τRed0r-inline-call-not-final[of P t h1 e es]
  obtain e' es' where red1: τRed0r P t h1 (e, es) (e', es')
    and final e' ⟹ es' = [] and feq: collapse (e, es) = collapse (e', es') by blast
  hence red1': red0-mthr.silent-moves P t ((e, es), h2) ((e', es'), h2)
    unfolding heap by −(rule τRed0r-into-silent-moves)
  have nfin: ¬ final e'
  proof
    assume fin: final e'
    hence es' = [] by(rule ‹final e' ⟹ es' = []›)
    with fold fin feq have final e1 by simp
    with red show False by auto
  qed
  from red1 wf-state have wf-state': wf-state (e', es') by(rule τRed0r-preserves-wf-state[OF wf])
  hence fv: fvs (e' # es') = {} and icl: ∀ e ∈ set es'. is-call e by auto
  from red-fold-exs'[OF red[unfolded fold x1 feq] wf-state' nfin]
  obtain E' where e1': e1' = collapse (E', es')
    and red': P,t ⊢ ⟨e',(h1', Map.empty)⟩ −ta1→ ⟨E',(h1', Map.empty)⟩ by auto
  from fv fv-collapse[OF icl, of e'] fold feq have fv e1 = {} by(auto)
  with red-dom-lcl[OF red] x1 have x1': x1' = Map.empty by simp
  from red-red0-tabisim0[OF wf red'] obtain ta2
    where red'': extTA2J0 P,P,t ⊢ ⟨e',(h1', Map.empty)⟩ −ta2→ ⟨E',(h1', Map.empty)⟩
    and tasim: ta-bisim0 ta1 ta2 by auto
  from τ fold feq icl nfin have ¬ τmove0 P h1 e' by(simp add: collapse-τmove0-inv)
  hence ∀ aMvs. call e' = ⌊aMvs⌋ ⟶ synthesized-call P h1 aMvs
    by(auto dest: τmove0-callD)
  with red'' have red''': P,t ⊢0 ⟨e'/es', h1'⟩ −ta2→ ⟨E'/es', h1''⟩ by(rule red0Red)
  moreover from τ fold feq icl nfin have ¬ τmove0 P h1 e' by(simp add: collapse-τmove0-inv)
  hence ¬ τMOVE0 P ((e', es'), h1') ta2 ((E', es'), h1'') using nfin by auto
  moreover from red''' wf-state' have wf-state (E', es') by(rule red0-preserves-wf-state[OF wf])
  hence bisim-red-red0 s1' ((E', es'), h1'') unfolding s1' e1' x1' by(auto)
  ultimately show ∃ s2' s2'' ta2. red0-mthr.silent-moves P t s2 s2' ∧ mred0 P t s2' ta2 s2'' ∧
                  ¬ τMOVE0 P s2' ta2 s2'' ∧ bisim-red-red0 s1' s2'' ∧ ta-bisim0 ta1 ta2
    using tasim red1' heap unfolding s1' s2 by −(rule exI conjI|assumption|auto)+
next
  fix s1 s2 ta2 s2'
  assume bisim-red-red0 s1 s2 and mred0 P t s2 ta2 s2' ¬ τMOVE0 P s2 ta2 s2'
  from red-simulates-red0[OF wf this]
  show ∃ s1' s1'' ta1. red-mthr.silent-moves P t s1 s1' ∧ mred P t s1' ta1 s1'' ∧
                  ¬ τMOVE P s1' ta1 s1'' ∧ bisim-red-red0 s1'' s2' ∧ ta-bisim0 ta1 ta2
    by(blast intro: rtranclp.rtrancl-refl)
qed

lemma delay-bisimulation-diverge-red-red0:
  assumes wwf-J-prog P
  shows delay-bisimulation-diverge (mred P t) (mred0 P t) bisim-red-red0 ta-bisim0 (τMOVE P)
(τMOVE0 P)
proof −
  interpret delay-bisimulation-measure
    mred P t mred0 P t bisim-red-red0 ta-bisim0 τMOVE P τMOVE0 P
    λe e'. False λ((e, es), h) ((e, es'), h). length es < length es'
    using assms by(rule delay-bisimulation-measure-red-red0)
  show ?thesis by unfold-locales
qed
```

lemma *bisim-red-red0-finalD*:
assumes *bisim*: *bisim-red-red0* (*x1*, *m1*) (*x2*, *m2*)
and *final-expr x1*
shows ∃ *x2'*. *red0-mthr.silent-moves P t* (*x2*, *m2*) (*x2'*, *m2*) ∧ *bisim-red-red0* (*x1*, *m1*) (*x2'*, *m2*) ∧ *final-expr0 x2'*
proof −
from *bisim*
obtain *e' e es* **where** *wf-state*: *wf-state* (*e*, *es*)
and [*simp*]: *x1* = (*e'*, *Map.empty*) *x2* = (*e*, *es*) *e'* = *collapse* (*e*, *es*) *m2* = *m1*
by *cases*(*cases x2*, *auto*)
from ‹*final-expr x1*› **have** *final* (*collapse* (*e*, *es*)) **by** *simp*
moreover from *wf-state* **have** ∀ *e*∈*set es*. *is-call e* **by** *auto*
ultimately have *red0-mthr.silent-moves P t* ((*e*, *es*), *m1*) ((*collapse* (*e*, *es*), []), *m1*)
proof(*induction es arbitrary*: *e*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons e' es*)
from ‹*final* (*collapse* (*e*, *e'* # *es*))› **have** *final* (*collapse* (*inline-call e e'*, *es*)) **by** *simp*
moreover from ‹∀ *e*∈*set* (*e'* # *es*). *is-call e*› **have** ∀ *e*∈*set es*. *is-call e* **by** *simp*
ultimately have *red0-mthr.silent-moves P t* ((*inline-call e e'*, *es*), *m1*) ((*collapse* (*inline-call e e'*, *es*), []), *m1*)
by(*rule Cons.IH*)
moreover from ‹*final* (*collapse* (*e*, *e'* # *es*))› ‹∀ *e*∈*set* (*e'* # *es*). *is-call e*›
have *final e* **by**(*rule collapse-finalD*)
hence *P*,*t* ⊢*0* ⟨*e*/*e'*#*es*, *m1*⟩ −*ε*→ ⟨*inline-call e e'*/*es*, *m1*⟩ **by**(*rule red0Return*)
with ‹*final e*› **have** *red0-mthr.silent-move P t* ((*e*, *e'*#*es*), *m1*) ((*inline-call e e'*, *es*), *m1*) **by** *auto*
ultimately show *?case* **by** −(*erule converse-rtranclp-into-rtranclp*, *simp*)
qed
moreover have *bisim-red-red0* ((*collapse* (*e*, *es*), *Map.empty*), *m1*) ((*collapse* (*e*, *es*), []), *m1*)
using ‹*final* (*collapse* (*e*, *es*))› **by**(*auto intro*!: *bisim-red-red0I*)
ultimately show *?thesis* **using** ‹*final* (*collapse* (*e*, *es*))› **by** *auto*
qed

lemma *red0-simulates-red-not-final*:
assumes *wwf*: *wwf-J-prog P*
assumes *bisim*: *bisim-red-red0* ((*e*, *xs*), *h*) ((*e0*, *es0*), *h0*)
and *red*: *P*,*t* ⊢ ⟨*e*, (*h*, *xs*)⟩ −*ta*→ ⟨*e'*, (*h'*, *xs'*)⟩
and *fin*: ¬ *final e0*
and *nτ*: ¬ *τmove0 P h e*
shows ∃ *e0' ta0*. *P*,*t* ⊢*0* ⟨*e0*/*es0*, *h*⟩ −*ta0*→ ⟨*e0'*/*es0*, *h'*⟩ ∧ *bisim-red-red0* ((*e'*, *xs'*), *h'*) ((*e0'*, *es0*), *h'*) ∧ *ta-bisim0 ta ta0*
proof −
from *bisim* **have** [*simp*]: *xs* = *Map.empty h0* = *h* **and** *e*: *e* = *collapse* (*e0*, *es0*)
and *wfs*: *wf-state* (*e0*, *es0*) **by**(*auto elim*!: *bisim-red-red0.cases*)
with *red* **have** *P*,*t* ⊢ ⟨*collapse* (*e0*, *es0*), (*h*, *Map.empty*)⟩ −*ta*→ ⟨*e'*, (*h'*, *xs'*)⟩ **by** *simp*
from *wfs red-fold-exs'*[*OF this*] *fin* **obtain** *e0'* **where** *e'*: *e'* = *collapse* (*e0'*, *es0*)
and *red'*: *P*,*t* ⊢ ⟨*e0*,(*h*, *Map.empty*)⟩ −*ta*→ ⟨*e0'*,(*h'*, *Map.empty*)⟩ **by**(*auto*)
from *wfs fv-collapse*[*of es0*, *of e0*] *e* **have** *fv e* = {} **by**(*auto*)
with *red-dom-lcl*[*OF red*] **have** [*simp*]: *xs'* = *Map.empty* **by** *simp*
from *red-red0-tabisim0*[*OF wwf red'*] **obtain** *ta0*
where *red''*: *extTA2J0 P*,*P*,*t* ⊢ ⟨*e0*,(*h*, *Map.empty*)⟩ −*ta0*→ ⟨*e0'*,(*h'*, *Map.empty*)⟩
and *tasim*: *ta-bisim0 ta ta0* **by** *auto*
from *nτ e wfs fin* **have** ¬ *τmove0 P h e0* **by**(*auto simp add*: *collapse-τmove0-inv*)

hence $\forall$ *aMvs. call e0* = $\lfloor$*aMvs*$\rfloor$ $\longrightarrow$ *synthesized-call P h aMvs*
by(*auto dest*: τ*move0-callD*)
with *red''* **have** *red'''*: *P*,*t* $\vdash$ *0* $\langle$*e0*/*es0*, *h*$\rangle$ $-$*ta0*$\rightarrow$ $\langle$*e0'*/*es0*, *h'*$\rangle$ **by**(*rule red0Red*)
moreover from *red'''* *wfs* **have** *wf-state* (*e0'*, *es0*) **by**(*rule red0-preserves-wf-state*[*OF wwf*])
hence *bisim-red-red0* ((*e'*, *xs'*), *h'*) ((*e0'*, *es0*), *h'*) **unfolding** *e'* **by**(*auto*)
ultimately show *?thesis* **using** *tasim* **by**(*auto simp del*: *split-paired-Ex*)
qed

lemma *red-red0-FWbisim*:
assumes *wf*: *wwf-J-prog P*
shows *FWdelay-bisimulation-diverge final-expr* (*mred P*) *final-expr0* (*mred0 P*)
(*λt. bisim-red-red0*) (*λexs* (*e0*, *es0*). ¬ *final e0*) (τ*MOVE P*) (τ*MOVE0 P*)
proof −
interpret *delay-bisimulation-diverge mred P t mred0 P t bisim-red-red0 ta-bisim0* τ*MOVE P* τ*MOVE0 P*
for *t* **by**(*rule delay-bisimulation-diverge-red-red0*[*OF wf*])
show *?thesis*
proof
fix *t* **and** *s1* :: ((*'addr expr* × *'addr locals*) × *'heap*) **and** *s2* :: ((*'addr expr* × *'addr expr list*) × *'heap*)
assume *bisim-red-red0 s1 s2* (*λ*(*x1*, *m*). *final-expr x1*) *s1*
moreover obtain *x1 m1* **where** [*simp*]: *s1* = (*x1*, *m1*) **by**(*cases s1*)
moreover obtain *x2 m2* **where** [*simp*]: *s2* = (*x2*, *m2*) **by**(*cases s2*)
ultimately have *bisim-red-red0* (*x1*, *m1*) (*x2*, *m2*) *final-expr x1* **by** *simp-all*
from *bisim-red-red0-finalD*[*OF this*, *of P t*]
show ∃ *s2'*. *red0-mthr.silent-moves P t s2 s2'* ∧ *bisim-red-red0 s1 s2'* ∧ (*λ*(*x2*, *m*). *final-expr0 x2*) *s2'* **by** *auto*
next
fix *t* **and** *s1* :: ((*'addr expr* × *'addr locals*) × *'heap*) **and** *s2* :: ((*'addr expr* × *'addr expr list*) × *'heap*)
assume *bisim-red-red0 s1 s2* (*λ*(*x2*, *m*). *final-expr0 x2*) *s2*
moreover obtain *x1 m1* **where** [*simp*]: *s1* = (*x1*, *m1*) **by**(*cases s1*)
moreover obtain *x2 m2* **where** [*simp*]: *s2* = (*x2*, *m2*) **by**(*cases s2*)
ultimately have *bisim-red-red0* (*x1*, *m1*) (*x2*, *m2*) *final-expr0 x2* **by** *simp-all*
moreover hence *final-expr x1* **by**(*rule bisim-red-red0-final0D*)
ultimately show ∃ *s1'*. *red-mthr.silent-moves P t s1 s1'* ∧ *bisim-red-red0 s1' s2* ∧ (*λ*(*x1*, *m*). *final-expr x1*) *s1'* **by** *auto*
next
fix *t' x m1 xx m2 t x1 x2 x1' ta1 x1'' m1' x2' ta2 x2'' m2'*
assume *b*: *bisim-red-red0* (*x*, *m1*) (*xx*, *m2*) **and** *bo*: *bisim-red-red0* (*x1*, *m1*) (*x2*, *m2*)
and *red-mthr.silent-moves P t* (*x1*, *m1*) (*x1'*, *m1*)
and *red1*: *mred P t* (*x1'*, *m1*) *ta1* (*x1''*, *m1'*) **and** ¬ τ*MOVE P* (*x1'*, *m1*) *ta1* (*x1''*, *m1'*)
and *red0-mthr.silent-moves P t* (*x2*, *m2*) (*x2'*, *m2*)
and *red2*: *mred0 P t* (*x2'*, *m2*) *ta2* (*x2''*, *m2'*) **and** ¬ τ*MOVE0 P* (*x2'*, *m2*) *ta2* (*x2''*, *m2'*)
and *bo'*: *bisim-red-red0* (*x1''*, *m1'*) (*x2''*, *m2'*)
and *tb*: *ta-bisim0 ta1 ta2*
from *b* **have** *m1* = *m2* **by**(*auto elim*: *bisim-red-red0.cases*)
moreover from *bo'* **have** *m1'* = *m2'* **by**(*auto elim*: *bisim-red-red0.cases*)
ultimately show *bisim-red-red0* (*x*, *m1'*) (*xx*, *m2'*) **using** *b*
by(*auto elim*: *bisim-red-red0.cases*)
next
fix *t x1 m1 x2 m2 x1' ta1 x1'' m1' x2' ta2 x2'' m2' w*
assume *b*: *bisim-red-red0* (*x1*, *m1*) (*x2*, *m2*)

and *red-mthr.silent-moves P t (x1, m1) (x1', m1)*
and *red1*: *mred P t (x1', m1) ta1 (x1'', m1')* **and** ¬ *τMOVE P (x1', m1) ta1 (x1'', m1')*
and *red0-mthr.silent-moves P t (x2, m2) (x2', m2)*
and *red2*: *mred0 P t (x2', m2) ta2 (x2'', m2')* **and** ¬ *τMOVE0 P (x2', m2) ta2 (x2'', m2')*
and *b'*: *bisim-red-red0 (x1'', m1') (x2'', m2')* **and** *ta-bisim0 ta1 ta2*
and *Suspend*: *Suspend w ∈ set {|ta1|}_w Suspend w ∈ set {|ta2|}_w*
hence *(λexs (e0, es0). is-call e0) x1'' x2''*
by(*cases x1'*)(*cases x2', auto dest*: *Red-Suspend-is-call simp add*: *final-iff*)
thus *(λexs (e0, es0). ¬ final e0) x1'' x2''* **by**(*auto simp add*: *final-iff is-call-def*)
next
fix *t x1 m1 x2 m2 ta1 x1' m1'*
assume *b*: *bisim-red-red0 (x1, m1) (x2, m2)*
and *c*: *(λ(e0, es0). ¬ final e0) x2*
and *red1*: *mred P t (x1, m1) ta1 (x1', m1')*
and *wakeup*: *Notified ∈ set {|ta1|}_w ∨ WokenUp ∈ set {|ta1|}_w*
from *c* **have** ¬ *final (fst x2)* **by**(*auto simp add*: *is-call-def*)
moreover from *red1 wakeup* **have** ¬ *τmove0 P m1 (fst x1)*
by(*cases x1*)(*auto dest*: *red-τ-taD*[**where** *extTA=extTA2J P*, *simplified*] *simp add*: *ta-upd-simps*)
moreover from *b* **have** *m2 = m1* **by**(*cases*) *auto*
ultimately obtain *e0' ta0* **where** *P,t ⊢0 ⟨fst x2/snd x2,m2⟩ −ta0→ ⟨e0'/snd x2,m1'⟩*
bisim-red-red0 ((fst x1', snd x1'), m1') ((e0', snd x2), m1') ta-bisim0 ta1 ta0
using *red0-simulates-red-not-final*[*OF wf*, *of fst x1 snd x1 m1 fst x2 snd x2 m2 t ta1 fst x1' m1' snd x1'*]
using *b red1* **by**(*auto simp add*: *split-beta*)
thus *∃ ta2 x2' m2'. mred0 P t (x2, m2) ta2 (x2', m2') ∧ bisim-red-red0 (x1', m1') (x2', m2') ∧ ta-bisim0 ta1 ta2*
by(*cases ta0*)(*fastforce simp add*: *split-beta*)
next
fix *t x1 m1 x2 m2 ta2 x2' m2'*
assume *b*: *bisim-red-red0 (x1, m1) (x2, m2)*
and *c*: *(λ(e0, es0). ¬ final e0) x2*
and *red2*: *mred0 P t (x2, m2) ta2 (x2', m2')*
and *wakeup*: *Notified ∈ set {|ta2|}_w ∨ WokenUp ∈ set {|ta2|}_w*
from *b* **have** [*simp*]: *m1 = m2* **by** *cases auto*
with *red-simulates-red0*[*OF wf b red2*] *wakeup* **obtain** *s1' ta1*
where *mred P t (x1, m1) ta1 s1' bisim-red-red0 s1' (x2', m2') ta-bisim0 ta1 ta2*
by(*fastforce simp add*: *split-paired-Ex*)
moreover from ‹*bisim-red-red0 s1' (x2', m2')*› **have** *m2' = snd s1'* **by** *cases auto*
ultimately
show *∃ ta1 x1' m1'. mred P t (x1, m1) ta1 (x1', m1') ∧ bisim-red-red0 (x1', m1') (x2', m2') ∧ ta-bisim0 ta1 ta2*
by(*cases ta1*)(*fastforce simp add*: *split-beta*)
next
show *(∃ x. final-expr x) ⟷ (∃ x. final-expr0 x)*
by(*auto simp add*: *final-iff*)
qed
qed

end

sublocale *J-heap-base* < *red-red0*:
FWdelay-bisimulation-base
final-expr
mred P

final-expr0
mred0 P
convert-RA
λt. bisim-red-red0
λexs (e0, es0). ¬ final e0
τMOVE P τMOVE0 P
for *P*
by(*unfold-locales*)

context *J-heap-base* **begin**

lemma *bisim-J-J0-start*:
assumes *wf*: *wwf-J-prog P*
and *wf-start*: *wf-start-state P C M vs*
shows *red-red0.mbisim* (*J-start-state P C M vs*) (*J0-start-state P C M vs*)
proof −
from *wf-start* **obtain** *Ts T pns body D*
where *sees*: *P* ⊢ *C sees M*:*Ts*→*T*=⌊(*pns*,*body*)⌋ *in D*
and *conf*: *P*,*start-heap* ⊢ *vs* [:≤] *Ts*
by *cases auto*

from *conf* **have** *vs*: *length vs* = *length Ts* **by**(*rule list-all2-lengthD*)
from *sees-wf-mdecl*[*OF wf sees*]
have *wwfCM*: *wwf-J-mdecl P D* (*M*, *Ts*, *T*, *pns*, *body*)
and *len*: *length pns* = *length Ts* **by**(*auto simp add*: *wf-mdecl-def*)
from *wwfCM* **have** *fvbody*: *fv body* ⊆ {*this*} ∪ *set pns*
and *pns*: *length pns* = *length Ts* **by** *simp-all*
with *vs len* **have** *fv*: *fv* (*blocks pns Ts vs body*) ⊆ {*this*} **by** *auto*
with *len vs sees* **show** *?thesis* **unfolding** *start-state-def*
by(*auto intro*!: *red-red0.mbisimI*)(*auto intro*!: *bisim-red-red0.intros wset-thread-okI simp add*: *is-call-def split*: *if-split-asm*)
qed

end

end

7.4 The intermediate language J1

theory *J1State* **imports**
../J/State
CallExpr
begin

type-synonym
'addr expr1 = (*nat*, *nat*, *'addr*) *exp*

type-synonym
'addr J1-prog = *'addr expr1 prog*

type-synonym
'addr locals1 = *'addr val list*

translations
(*type*) *'addr expr1* <= (*type*) (*nat*, *nat*, *'addr*) *exp*
(*type*) *'addr J1-prog* <= (*type*) *'addr expr1 prog*

type-synonym
'addr J1state = (*'addr expr1* × *'addr locals1*) *list*

type-synonym
(*'addr*, *'thread-id*, *'heap*) *J1-thread-action* =
(*'addr*, *'thread-id*, (*'addr expr1* × *'addr locals1*) × (*'addr expr1* × *'addr locals1*) *list*,*'heap*) *Jinja-thread-action*

type-synonym
(*'addr*, *'thread-id*, *'heap*) *J1-state* =
(*'addr*,*'thread-id*,(*'addr expr1* × *'addr locals1*) × (*'addr expr1* × *'addr locals1*) *list*,*'heap*,*'addr*) *state*

print-translation ‹
let
fun tr'
[*a1*, *t*
, *Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax exp*}, -) $ *Const* (@{*type-syntax nat*}, -) $ *Const* (@{*type-syntax nat*}, -) $ *a2*) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax val*}, -) $ *a3*))) $
(*Const* (@{*type-syntax list*}, -) $
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax exp*}, -) $ *Const* (@{*type-syntax nat*}, -) $ *Const* (@{*type-syntax nat*}, -) $ *a4*) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax val*}, -) $ *a5*))))
, *h*] =
if a1 = *a2 andalso a2* = *a3 andalso a3* = *a4 andalso a4* = *a5*
then Syntax.const @{*type-syntax J1-thread-action*} $ *a1* $ *t* $ *h*
else raise Match;
in [(@{*type-syntax Jinja-thread-action*}, *K tr'*)]
end
›
typ (*'addr*,*'thread-id*,*'heap*) *J1-thread-action*

print-translation ‹
let
fun tr'
[*a1*, *t*
, *Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax exp*}, -) $ *Const* (@{*type-syntax nat*}, -) $ *Const* (@{*type-syntax nat*}, -) $ *a2*) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax val*}, -) $ *a3*))) $
(*Const* (@{*type-syntax list*}, -) $
(*Const* (@{*type-syntax prod*}, -) $
(*Const* (@{*type-syntax exp*}, -) $ *Const* (@{*type-syntax nat*}, -) $ *Const* (@{*type-syntax nat*}, -) $ *a4*) $
(*Const* (@{*type-syntax list*}, -) $ (*Const* (@{*type-syntax val*}, -) $ *a5*))))

```
      , h, a6] =
      if a1 = a2 andalso a2 = a3 andalso a3 = a4 andalso a4 = a5 andalso a5 = a6
      then Syntax.const @{type-syntax J1-state} $ a1 $ t $ h
      else raise Match;
    in [(@{type-syntax state}, K tr')]
  end
›
typ ('addr, 'thread-id, 'heap) J1-state

fun blocks1 :: nat ⇒ ty list ⇒ (nat,'b,'addr) exp ⇒ (nat,'b,'addr) exp
where
  blocks1 n [] e = e
| blocks1 n (T#Ts) e = {n:T=None; blocks1 (Suc n) Ts e}

primrec max-vars:: ('a,'b,'addr) exp ⇒ nat
  and max-varss:: ('a,'b,'addr) exp list ⇒ nat
where
  max-vars (new C) = 0
| max-vars (newA T⌊e⌉) = max-vars e
| max-vars (Cast C e) = max-vars e
| max-vars (e instanceof T) = max-vars e
| max-vars (Val v) = 0
| max-vars (e «bop» e') = max (max-vars e) (max-vars e')
| max-vars (Var V) = 0
| max-vars (V:=e) = max-vars e
| max-vars (a⌊i⌉) = max (max-vars a) (max-vars i)
| max-vars (AAss a i e) = max (max (max-vars a) (max-vars i)) (max-vars e)
| max-vars (a•length) = max-vars a
| max-vars (e•F{D}) = max-vars e
| max-vars (FAss e1 F D e2) = max (max-vars e1) (max-vars e2)
| max-vars (e•compareAndSwap(D•F, e', e'')) = max (max (max-vars e) (max-vars e')) (max-vars e'')
| max-vars (e•M(es)) = max (max-vars e) (max-varss es)
| max-vars ({V:T=vo; e}) = max-vars e + 1
— sync and insync will need an extra local variable when compiling to bytecode to store the object
that is being synchronized on until its release
| max-vars (sync_V (e') e) = max (max-vars e') (max-vars e + 1)
| max-vars (insync_V (a) e) = max-vars e + 1
| max-vars (e1;;e2) = max (max-vars e1) (max-vars e2)
| max-vars (if (e) e1 else e2) =
   max (max-vars e) (max (max-vars e1) (max-vars e2))
| max-vars (while (b) e) = max (max-vars b) (max-vars e)
| max-vars (throw e) = max-vars e
| max-vars (try e1 catch(C V) e2) = max (max-vars e1) (max-vars e2 + 1)

| max-varss [] = 0
| max-varss (e#es) = max (max-vars e) (max-varss es)

— Indices in blocks increase by 1

primrec ℬ :: 'addr expr1 ⇒ nat ⇒ bool
  and ℬs :: 'addr expr1 list ⇒ nat ⇒ bool
where
  ℬ (new C) i = True
| ℬ (newA T⌊e⌉) i = ℬ e i
```

$| \mathcal{B}\ (Cast\ C\ e)\ i = \mathcal{B}\ e\ i$
$| \mathcal{B}\ (e\ instanceof\ T)\ i = \mathcal{B}\ e\ i$
$| \mathcal{B}\ (Val\ v)\ i = True$
$| \mathcal{B}\ (e1\ «bop»\ e2)\ i = (\mathcal{B}\ e1\ i \wedge \mathcal{B}\ e2\ i)$
$| \mathcal{B}\ (Var\ j)\ i = True$
$| \mathcal{B}\ (j{:=}e)\ i = \mathcal{B}\ e\ i$
$| \mathcal{B}\ (a\lfloor j\rceil)\ i = (\mathcal{B}\ a\ i \wedge \mathcal{B}\ j\ i)$
$| \mathcal{B}\ (a\lfloor j\rceil{:=}e)\ i = (\mathcal{B}\ a\ i \wedge \mathcal{B}\ j\ i \wedge \mathcal{B}\ e\ i)$
$| \mathcal{B}\ (a{\cdot}length)\ i = \mathcal{B}\ a\ i$
$| \mathcal{B}\ (e{\cdot}F\{D\})\ i = \mathcal{B}\ e\ i$
$| \mathcal{B}\ (e1{\cdot}F\{D\} := e2)\ i = (\mathcal{B}\ e1\ i \wedge \mathcal{B}\ e2\ i)$
$| \mathcal{B}\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e''))\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}\ e'\ i \wedge \mathcal{B}\ e''\ i)$
$| \mathcal{B}\ (e{\cdot}M(es))\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}s\ es\ i)$
$| \mathcal{B}\ (\{j{:}T{=}vo;\ e\})\ i = (i = j \wedge \mathcal{B}\ e\ (i{+}1))$
$| \mathcal{B}\ (sync_V\ (o')\ e)\ i = (i = V \wedge \mathcal{B}\ o'\ i \wedge \mathcal{B}\ e\ (i{+}1))$
$| \mathcal{B}\ (insync_V\ (a)\ e)\ i = (i = V \wedge \mathcal{B}\ e\ (i{+}1))$
$| \mathcal{B}\ (e1;;e2)\ i = (\mathcal{B}\ e1\ i \wedge \mathcal{B}\ e2\ i)$
$| \mathcal{B}\ (if\ (e)\ e1\ else\ e2)\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}\ e1\ i \wedge \mathcal{B}\ e2\ i)$
$| \mathcal{B}\ (throw\ e)\ i = \mathcal{B}\ e\ i$
$| \mathcal{B}\ (while\ (e)\ c)\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}\ c\ i)$
$| \mathcal{B}\ (try\ e1\ catch(C\ j)\ e2)\ i = (\mathcal{B}\ e1\ i \wedge i{=}j \wedge \mathcal{B}\ e2\ (i{+}1))$

$| \mathcal{B}s\ []\ i = True$
$| \mathcal{B}s\ (e\#es)\ i = (\mathcal{B}\ e\ i \wedge \mathcal{B}s\ es\ i)$

Variables for monitor addresses do not occur freely in synchonization blocks

primrec *syncvars* :: $('a, 'a, 'addr)\ exp \Rightarrow bool$
and *syncvarss* :: $('a, 'a, 'addr)\ exp\ list \Rightarrow bool$
where
$syncvars\ (new\ C) = True$
$| syncvars\ (newA\ T\lfloor e\rceil) = syncvars\ e$
$| syncvars\ (Cast\ T\ e) = syncvars\ e$
$| syncvars\ (e\ instanceof\ T) = syncvars\ e$
$| syncvars\ (Val\ v) = True$
$| syncvars\ (e1\ «bop»\ e2) = (syncvars\ e1 \wedge syncvars\ e2)$
$| syncvars\ (Var\ V) = True$
$| syncvars\ (V{:=}e) = syncvars\ e$
$| syncvars\ (a\lfloor i\rceil) = (syncvars\ a \wedge syncvars\ i)$
$| syncvars\ (a\lfloor i\rceil := e) = (syncvars\ a \wedge syncvars\ i \wedge syncvars\ e)$
$| syncvars\ (a{\cdot}length) = syncvars\ a$
$| syncvars\ (e{\cdot}F\{D\}) = syncvars\ e$
$| syncvars\ (e{\cdot}F\{D\} := e2) = (syncvars\ e \wedge syncvars\ e2)$
$| syncvars\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e'')) = (syncvars\ e \wedge syncvars\ e' \wedge syncvars\ e'')$
$| syncvars\ (e{\cdot}M(es)) = (syncvars\ e \wedge syncvarss\ es)$
$| syncvars\ \{V{:}T{=}vo;e\} = syncvars\ e$
$| syncvars\ (sync_V\ (e1)\ e2) = (syncvars\ e1 \wedge syncvars\ e2 \wedge V \notin fv\ e2)$
$| syncvars\ (insync_V\ (a)\ e) = (syncvars\ e \wedge V \notin fv\ e)$
$| syncvars\ (e1;;e2) = (syncvars\ e1 \wedge syncvars\ e2)$
$| syncvars\ (if\ (b)\ e1\ else\ e2) = (syncvars\ b \wedge syncvars\ e1 \wedge syncvars\ e2)$
$| syncvars\ (while\ (b)\ c) = (syncvars\ b \wedge syncvars\ c)$
$| syncvars\ (throw\ e) = syncvars\ e$
$| syncvars\ (try\ e1\ catch(C\ V)\ e2) = (syncvars\ e1 \wedge syncvars\ e2)$

$| syncvarss\ [] = True$

```
| syncvarss (e#es) = (syncvars e ∧ syncvarss es)

definition bsok :: 'addr expr1 ⇒ nat ⇒ bool
where bsok e n ≡ B e n ∧ expr-locks e = (λad. 0)

definition bsoks :: 'addr expr1 list ⇒ nat ⇒ bool
where bsoks es n ≡ Bs es n ∧ expr-lockss es = (λad. 0)

primrec call1 :: ('a, 'b, 'addr) exp ⇒ ('addr × mname × 'addr val list) option
  and calls1 :: ('a, 'b, 'addr) exp list ⇒ ('addr × mname × 'addr val list) option
where
  call1 (new C) = None
| call1 (newA T⌊e⌉) = call1 e
| call1 (Cast C e) = call1 e
| call1 (e instanceof T) = call1 e
| call1 (Val v) = None
| call1 (Var V) = None
| call1 (V:=e) = call1 e
| call1 (e «bop» e') = (if is-val e then call1 e' else call1 e)
| call1 (a⌊i⌉) = (if is-val a then call1 i else call1 a)
| call1 (AAss a i e) = (if is-val a then (if is-val i then call1 e else call1 i) else call1 a)
| call1 (a·length) = call1 a
| call1 (e·F{D}) = call1 e
| call1 (FAss e F D e') = (if is-val e then call1 e' else call1 e)
| call1 (CompareAndSwap e D F e' e'') = (if is-val e then (if is-val e' then call1 e'' else call1 e') else
call1 e)
| call1 (e·M(es)) = (if is-val e then
                  (if is-vals es ∧ is-addr e then ⌊(THE a. e = addr a, M, THE vs. es = map Val vs)⌋
else calls1 es)
                  else call1 e)
| call1 ({V:T=vo; e}) = (case vo of None ⇒ call1 e | Some v ⇒ None)
| call1 (sync_V (o') e) = call1 o'
| call1 (insync_V (a) e) = call1 e
| call1 (e;;e') = call1 e
| call1 (if (e) e1 else e2) = call1 e
| call1 (while(b) e) = None
| call1 (throw e) = call1 e
| call1 (try e1 catch(C V) e2) = call1 e1

| calls1 [] = None
| calls1 (e#es) = (if is-val e then calls1 es else call1 e)


lemma expr-locks-blocks1 [simp]:
  expr-locks (blocks1 n Ts e) = expr-locks e
by(induct n Ts e rule: blocks1.induct) simp-all

lemma max-varss-append [simp]:
  max-varss (es @ es') = max (max-varss es) (max-varss es')
by(induct es, auto)

lemma max-varss-map-Val [simp]: max-varss (map Val vs) = 0
by(induct vs) auto
```

lemma *blocks1-max-vars*:
max-vars (*blocks1 n Ts e*) = *max-vars e* + *length Ts*
by(*induct n Ts e rule*: *blocks1.induct*)(*auto*)

lemma *blocks-max-vars*:
⟦ *length vs* = *length pns*; *length Ts* = *length pns* ⟧
⟹ *max-vars* (*blocks pns Ts vs e*) = *max-vars e* + *length pns*
by(*induct pns Ts vs e rule*: *blocks.induct*)(*auto*)

lemma *Bs-append* [*simp*]: $\mathcal{B}s$ (*es* @ *es'*) *n* ⟷ $\mathcal{B}s$ *es n* ∧ $\mathcal{B}s$ *es' n*
by(*induct es*) *auto*

lemma *Bs-map-Val* [*simp*]: $\mathcal{B}s$ (*map Val vs*) *n*
by(*induct vs*) *auto*

lemma *B-blocks1* [*intro*]: $\mathcal{B}$ *body* (*n* + *length Ts*) ⟹ $\mathcal{B}$ (*blocks1 n Ts body*) *n*
by(*induct n Ts body rule*: *blocks1.induct*)(*auto*)

lemma *B-extRet2J* [*simp*]: $\mathcal{B}$ *e n* ⟹ $\mathcal{B}$ (*extRet2J e va*) *n*
by(*cases va*) *auto*

lemma *B-inline-call*: ⟦ $\mathcal{B}$ *e n*; ⋀*n*. $\mathcal{B}$ *e' n* ⟧ ⟹ $\mathcal{B}$ (*inline-call e' e*) *n*
and *Bs-inline-calls*: ⟦ $\mathcal{B}s$ *es n*; ⋀*n*. $\mathcal{B}$ *e' n* ⟧ ⟹ $\mathcal{B}s$ (*inline-calls e' es*) *n*
by(*induct e* **and** *es arbitrary*: *n* **and** *n rule*: *call.induct calls.induct*) *auto*

lemma *syncvarss-append* [*simp*]: *syncvarss* (*es* @ *es'*) ⟷ *syncvarss es* ∧ *syncvarss es'*
by(*induct es*) *auto*

lemma *syncvarss-map-Val* [*simp*]: *syncvarss* (*map Val vs*)
by(*induct vs*) *auto*

lemma *bsok-simps* [*simp*]:
bsok (*new C*) *n* = *True*
bsok (*newA T*⌊*e*⌉) *n* = *bsok e n*
bsok (*Cast T e*) *n* = *bsok e n*
bsok (*e instanceof T*) *n* = *bsok e n*
bsok (*e1* «*bop*» *e2*) *n* = (*bsok e1 n* ∧ *bsok e2 n*)
bsok (*Var V*) *n* = *True*
bsok (*Val v*) *n* = *True*
bsok (*V* := *e*) *n* = *bsok e n*
bsok (*a*⌊*i*⌉) *n* = (*bsok a n* ∧ *bsok i n*)
bsok (*a*⌊*i*⌉ := *e*) *n* = (*bsok a n* ∧ *bsok i n* ∧ *bsok e n*)
bsok (*a*·*length*) *n* = *bsok a n*
bsok (*e*·*F*{*D*}) *n* = *bsok e n*
bsok (*e*·*F*{*D*} := *e'*) *n* = (*bsok e n* ∧ *bsok e' n*)
bsok (*e*·*compareAndSwap*(*D*·*F*, *e'*, *e''*)) *n* = (*bsok e n* ∧ *bsok e' n* ∧ *bsok e'' n*)
bsok (*e*·*M*(*ps*)) *n* = (*bsok e n* ∧ *bsoks ps n*)
bsok {*V*:*T*=*vo*; *e*} *n* = (*bsok e* (*Suc n*) ∧ *V* = *n*)
bsok (*sync*$_V$ (*e*) *e'*) *n* = (*bsok e n* ∧ *bsok e'* (*Suc n*) ∧ *V* = *n*)
bsok (*insync*$_V$ (*ad*) *e*) *n* = *False*
bsok (*e*;; *e'*) *n* = (*bsok e n* ∧ *bsok e' n*)
bsok (*if* (*e*) *e1 else e2*) *n* = (*bsok e n* ∧ *bsok e1 n* ∧ *bsok e2 n*)
bsok (*while* (*b*) *c*) *n* = (*bsok b n* ∧ *bsok c n*)
bsok (*throw e*) *n* = *bsok e n*

bsok (*try e catch*(*C V*) *e'*) *n* = (*bsok e n* ∧ *bsok e'* (*Suc n*) ∧ *V* = *n*)
and *bsoks-simps* [*simp*]:
bsoks [] *n* = *True*
bsoks (*e* # *es*) *n* = (*bsok e n* ∧ *bsoks es n*)
by(*auto simp add*: *bsok-def bsoks-def fun-eq-iff*)

lemma *call1-callE*:
assumes *call1* (*obj*·*M*(*pns*)) = ⌊(*a*, *M'*, *vs*)⌋
obtains (*CallObj*) *call1 obj* = ⌊(*a*, *M'*, *vs*)⌋
| (*CallParams*) *v* **where** *obj* = *Val v calls1 pns* = ⌊(*a*, *M'*, *vs*)⌋
| (*Call*) *obj* = *addr a pns* = *map Val vs M* = *M'*
using *assms* **by**(*auto split*: *if-split-asm simp add*: *is-vals-conv*)

lemma *calls1-map-Val-append* [*simp*]:
calls1 (*map Val vs* @ *es*) = *calls1 es*
by(*induct vs*) *simp-all*

lemma *calls1-map-Val* [*simp*]:
calls1 (*map Val vs*) = *None*
by(*induct vs*) *simp-all*

lemma fixes *e* :: ('*a*, '*b*, '*addr*) *exp* **and** *es* :: ('*a*, '*b*, '*addr*) *exp list*
shows *call1-imp-call*: *call1 e* = ⌊*aMvs*⌋ ⟹ *call e* = ⌊*aMvs*⌋
and *calls1-imp-calls*: *calls1 es* = ⌊*aMvs*⌋ ⟹ *calls es* = ⌊*aMvs*⌋
by(*induct e* **and** *es rule*: *call1.induct calls1.induct*) *auto*

lemma *max-vars-inline-call*: *max-vars* (*inline-call e' e*) ≤ *max-vars e* + *max-vars e'*
and *max-varss-inline-calls*: *max-varss* (*inline-calls e' es*) ≤ *max-varss es* + *max-vars e'*
by(*induct e* **and** *es rule*: *call1.induct calls1.induct*) *auto*

lemmas *inline-call-max-vars1* = *max-vars-inline-call*
lemmas *inline-calls-max-varss1* = *max-varss-inline-calls*

end

7.5 Abstract heap locales for J1 programs

theory *J1Heap* **imports**
J1State
../Common/Conform
begin

locale *J1-heap-base* = *heap-base* +
constrains *addr2thread-id* :: ('*addr* :: *addr*) ⇒ '*thread-id*
and *thread-id2addr* :: '*thread-id* ⇒ '*addr*
and *sc-spurious-wakeups* :: *bool*
and *empty-heap* :: '*heap*
and *allocate* :: '*heap* ⇒ *htype* ⇒ ('*heap* × '*addr*) *set*
and *typeof-addr* :: '*heap* ⇒ '*addr* ⇀ *htype*
and *heap-read* :: '*heap* ⇒ '*addr* ⇒ *addr-loc* ⇒ '*addr val* ⇒ *bool*
and *heap-write* :: '*heap* ⇒ '*addr* ⇒ *addr-loc* ⇒ '*addr val* ⇒ '*heap* ⇒ *bool*

locale *J1-heap* = *heap* +

constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *sc-spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *P* :: *'addr J1-prog*

sublocale *J1-heap* < *J1-heap-base* .

locale *J1-heap-conf-base* = *heap-conf-base* +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *sc-spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr J1-prog*

sublocale *J1-heap-conf-base* < *J1-heap-base* .

locale *J1-heap-conf* =
J1-heap-conf-base +
heap-conf +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *sc-spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr J1-prog*

sublocale *J1-heap-conf* < *J1-heap* **by**(*unfold-locales*)

locale *J1-conf-read* =
J1-heap-conf +
heap-conf-read +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *sc-spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*

and *P* :: *'addr J1-prog*

end

7.6 Semantics of the intermediate language

theory *J1* **imports**
J1State
J1Heap
../Framework/FWBisimulation
begin

abbreviation *final-expr1* :: (*'addr expr1* × *'addr locals1*) × (*'addr expr1* × *'addr locals1*) *list* ⇒ *bool*
where
final-expr1 ≡ λ(*ex*, *exs*). *final* (*fst ex*) ∧ *exs* = []

definition *extNTA2J1* ::
'addr J1-prog ⇒ (*cname* × *mname* × *'addr*) ⇒ ((*'addr expr1* × *'addr locals1*) × (*'addr expr1* × *'addr locals1*) *list*)
where
extNTA2J1 P = (λ(*C*, *M*, *a*). *let* (*D*, -, -, *meth*) = *method P C M*; *body* = *the meth*
in (({*0*:*Class D*=*None*; *body*}, *Addr a* # *replicate* (*max-vars body*) *undefined-value*), []))

lemma *extNTA2J1-iff* [*simp*]:
extNTA2J1 P (*C*, *M*, *a*) = (({*0*:*Class* (*fst* (*method P C M*))=*None*; *the* (*snd* (*snd* (*snd* (*method P C M*))))}, *Addr a* # *replicate* (*max-vars* (*the* (*snd* (*snd* (*snd* (*method P C M*)))))) *undefined-value*), [])
by(*simp add*: *extNTA2J1-def split-beta*)

abbreviation *extTA2J1* ::
'addr J1-prog ⇒ (*'addr*, *'thread-id*, *'heap*) *external-thread-action* ⇒ (*'addr*, *'thread-id*, *'heap*) *J1-thread-action*
where *extTA2J1 P* ≡ *convert-extTA* (*extNTA2J1 P*)

abbreviation (*input*) *extRet2J1* :: *'addr expr1* ⇒ *'addr extCallRet* ⇒ *'addr expr1*
where *extRet2J1* ≡ *extRet2J*

lemma *max-vars-extRet2J1* [*simp*]:
max-vars e = *0* ⟹ *max-vars* (*extRet2J1 e va*) = *0*
by(*cases va*) *simp-all*

context *J1-heap-base* **begin**

abbreviation *J1-start-state* :: *'addr J1-prog* ⇒ *cname* ⇒ *mname* ⇒ *'addr val list* ⇒ (*'addr*, *'thread-id*, *'heap*) *J1-state*
where
J1-start-state ≡
start-state (λ*C M Ts T body vs*. ((*blocks1 0* (*Class C* # *Ts*) *body*, *Null* # *vs* @ *replicate* (*max-vars body*) *undefined-value*), []))

inductive *red1* ::
bool ⇒ *'addr J1-prog* ⇒ *'thread-id* ⇒ *'addr expr1* ⇒ *'heap* × *'addr locals1*
⇒ (*'addr*, *'thread-id*, *'heap*) *external-thread-action* ⇒ *'addr expr1* ⇒ *'heap* × *'addr locals1* ⇒ *bool*

(‹-,-,- ⊢1 ((1⟨-,/-⟩) −−→/ (1⟨-,/-⟩))› [51,51,0,0,0,0,0,0] 81)
and *reds1* ::
bool ⇒ *'addr J1-prog* ⇒ *'thread-id* ⇒ *'addr expr1 list* ⇒ *'heap* × *'addr locals1*
⇒ (*'addr*, *'thread-id*, *'heap*) *external-thread-action* ⇒ *'addr expr1 list* ⇒ *'heap* × *'addr locals1* ⇒ *bool*
(‹-,-,- ⊢1 ((1⟨-,/-⟩) [−−→]/ (1⟨-,/-⟩))› [51,51,0,0,0,0,0,0] 81)
for *uf* :: *bool* **and** *P* :: *'addr J1-prog* **and** *t* :: *'thread-id*
where
Red1New:
(*h'*, *a*) ∈ *allocate h* (*Class-type C*)
⟹ *uf*,*P*,*t* ⊢1 ⟨*new C*, (*h*, *l*)⟩ −{|*NewHeapElem a* (*Class-type C*)|}→ ⟨*addr a*, (*h'*, *l*)⟩

| *Red1NewFail*:
allocate h (*Class-type C*) = {}
⟹ *uf*,*P*,*t* ⊢1 ⟨*new C*, (*h*, *l*)⟩ −ε→ ⟨*THROW OutOfMemory*, (*h*, *l*)⟩

| *New1ArrayRed*:
uf,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩
⟹ *uf*,*P*,*t* ⊢1 ⟨*newA T*⌊*e*⌉, *s*⟩ −*ta*→ ⟨*newA T*⌊*e'*⌉, *s'*⟩

| *Red1NewArray*:
⟦ 0 <=*s i*; (*h'*, *a*) ∈ *allocate h* (*Array-type T* (*nat* (*sint i*))) ⟧
⟹ *uf*,*P*,*t* ⊢1 ⟨*newA T*⌊*Val* (*Intg i*)⌉, (*h*, *l*)⟩ −{|*NewHeapElem a* (*Array-type T* (*nat* (*sint i*)))|}→ ⟨*addr a*, (*h'*, *l*)⟩

| *Red1NewArrayNegative*:
i <*s* 0 ⟹ *uf*,*P*,*t* ⊢1 ⟨*newA T*⌊*Val* (*Intg i*)⌉, *s*⟩ −ε→ ⟨*THROW NegativeArraySize*, *s*⟩

| *Red1NewArrayFail*:
⟦ 0 <=*s i*; *allocate h* (*Array-type T* (*nat* (*sint i*))) = {} ⟧
⟹ *uf*,*P*,*t* ⊢1 ⟨*newA T*⌊*Val* (*Intg i*)⌉, (*h*, *l*)⟩ −ε→ ⟨*THROW OutOfMemory*, (*h*, *l*)⟩

| *Cast1Red*:
uf,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩
⟹ *uf*,*P*,*t* ⊢1 ⟨*Cast C e*, *s*⟩ −*ta*→ ⟨*Cast C e'*, *s'*⟩

| *Red1Cast*:
⟦ *typeof*$_{hp\ s}$ *v* = ⌊*U*⌋; *P* ⊢ *U* ≤ *T* ⟧
⟹ *uf*,*P*,*t* ⊢1 ⟨*Cast T* (*Val v*), *s*⟩ −ε→ ⟨*Val v*, *s*⟩

| *Red1CastFail*:
⟦ *typeof*$_{hp\ s}$ *v* = ⌊*U*⌋; ¬ *P* ⊢ *U* ≤ *T* ⟧
⟹ *uf*,*P*,*t* ⊢1 ⟨*Cast T* (*Val v*), *s*⟩ −ε→ ⟨*THROW ClassCast*, *s*⟩

| *InstanceOf1Red*:
uf,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩ ⟹ *uf*,*P*,*t* ⊢1 ⟨*e instanceof T*, *s*⟩ −*ta*→ ⟨*e' instanceof T*, *s'*⟩

| *Red1InstanceOf*:
⟦ *typeof*$_{hp\ s}$ *v* = ⌊*U*⌋; *b* ⟷ *v* ≠ *Null* ∧ *P* ⊢ *U* ≤ *T* ⟧
⟹ *uf*,*P*,*t* ⊢1 ⟨(*Val v*) *instanceof T*, *s*⟩ −ε→ ⟨*Val* (*Bool b*), *s*⟩

| *Bin1OpRed1*:
uf,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩ ⟹ *uf*,*P*,*t* ⊢1 ⟨*e* «*bop*» *e2*, *s*⟩ −*ta*→ ⟨*e'* «*bop*» *e2*, *s'*⟩

| *Bin1OpRed2*:
uf,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩ ⟹ *uf*,*P*,*t* ⊢1 ⟨(*Val v*) «*bop*» *e*, *s*⟩ −*ta*→ ⟨(*Val v*) «*bop*» *e*′, *s*′⟩

| *Red1BinOp*:
binop bop v1 v2 = *Some* (*Inl v*) ⟹
uf,*P*,*t* ⊢1 ⟨(*Val v1*) «*bop*» (*Val v2*), *s*⟩ −*ε*→ ⟨*Val v*, *s*⟩

| *Red1BinOpFail*:
binop bop v1 v2 = *Some* (*Inr a*) ⟹
uf,*P*,*t* ⊢1 ⟨(*Val v1*) «*bop*» (*Val v2*), *s*⟩ −*ε*→ ⟨*Throw a*, *s*⟩

| *Red1Var*:
⟦ (*lcl s*)!*V* = *v*; *V* < *size* (*lcl s*) ⟧
⟹ *uf*,*P*,*t* ⊢1 ⟨*Var V*, *s*⟩ −*ε*→ ⟨*Val v*, *s*⟩

| *LAss1Red*:
uf,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩
⟹ *uf*,*P*,*t* ⊢1 ⟨*V*:=*e*, *s*⟩ −*ta*→ ⟨*V*:=*e*′, *s*′⟩

| *Red1LAss*:
V < *size l*
⟹ *uf*,*P*,*t* ⊢1 ⟨*V*:=(*Val v*), (*h*, *l*)⟩ −*ε*→ ⟨*unit*, (*h*, *l*[*V* := *v*])⟩

| *AAcc1Red1*:
uf,*P*,*t* ⊢1 ⟨*a*, *s*⟩ −*ta*→ ⟨*a*′, *s*′⟩ ⟹ *uf*,*P*,*t* ⊢1 ⟨*a*⌊*i*⌉, *s*⟩ −*ta*→ ⟨*a*′⌊*i*⌉, *s*′⟩

| *AAcc1Red2*:
uf,*P*,*t* ⊢1 ⟨*i*, *s*⟩ −*ta*→ ⟨*i*′, *s*′⟩ ⟹ *uf*,*P*,*t* ⊢1 ⟨(*Val a*)⌊*i*⌉, *s*⟩ −*ta*→ ⟨(*Val a*)⌊*i*′⌉, *s*′⟩

| *Red1AAccNull*:
uf,*P*,*t* ⊢1 ⟨*null*⌊*Val i*⌉, *s*⟩ −*ε*→ ⟨*THROW NullPointer*, *s*⟩

| *Red1AAccBounds*:
⟦ *typeof-addr* (*hp s*) *a* = ⌊*Array-type T n*⌋; *i* <*s* 0 ∨ *sint i* ≥ *int n* ⟧
⟹ *uf*,*P*,*t* ⊢1 ⟨(*addr a*)⌊*Val* (*Intg i*)⌉, *s*⟩ −*ε*→ ⟨*THROW ArrayIndexOutOfBounds*, *s*⟩

| *Red1AAcc*:
⟦ *typeof-addr h a* = ⌊*Array-type T n*⌋; 0 <=*s i*; *sint i* < *int n*;
heap-read h a (*ACell* (*nat* (*sint i*))) *v* ⟧
⟹ *uf*,*P*,*t* ⊢1 ⟨(*addr a*)⌊*Val* (*Intg i*)⌉, (*h*, *xs*)⟩ −{*ReadMem a* (*ACell* (*nat* (*sint i*))) *v*}→ ⟨*Val v*, (*h*, *xs*)⟩

| *AAss1Red1*:
uf,*P*,*t* ⊢1 ⟨*a*, *s*⟩ −*ta*→ ⟨*a*′, *s*′⟩ ⟹ *uf*,*P*,*t* ⊢1 ⟨*a*⌊*i*⌉ := *e*, *s*⟩ −*ta*→ ⟨*a*′⌊*i*⌉ := *e*, *s*′⟩

| *AAss1Red2*:
uf,*P*,*t* ⊢1 ⟨*i*, *s*⟩ −*ta*→ ⟨*i*′, *s*′⟩ ⟹ *uf*,*P*,*t* ⊢1 ⟨(*Val a*)⌊*i*⌉ := *e*, *s*⟩ −*ta*→ ⟨(*Val a*)⌊*i*′⌉ := *e*, *s*′⟩

| *AAss1Red3*:
uf,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩ ⟹ *uf*,*P*,*t* ⊢1 ⟨*AAss* (*Val a*) (*Val i*) *e*, *s*⟩ −*ta*→ ⟨(*Val a*)⌊*Val i*⌉ := *e*′, *s*′⟩

| *Red1AAssNull*:
uf,*P*,*t* ⊢1 ⟨*AAss null* (*Val i*) (*Val e*), *s*⟩ −*ε*→ ⟨*THROW NullPointer*, *s*⟩

| *Red1AAssBounds*:
⟦ *typeof-addr* (*hp s*) *a* = ⌊*Array-type T n*⌋; *i* <*s* 0 ∨ *sint i* ≥ *int n* ⟧
⟹ *uf*,*P*,*t* ⊢*1* ⟨*AAss* (*addr a*) (*Val* (*Intg i*)) (*Val e*), *s*⟩ −*ε*→ ⟨*THROW ArrayIndexOutOfBounds*, *s*⟩

| *Red1AAssStore*:
⟦ *typeof-addr* (*hp s*) *a* = ⌊*Array-type T n*⌋; 0 <=*s i*; *sint i* < *int n*;
typeof$_{hp\ s}$ *w* = ⌊*U*⌋; ¬ (*P* ⊢ *U* ≤ *T*) ⟧
⟹ *uf*,*P*,*t* ⊢*1* ⟨*AAss* (*addr a*) (*Val* (*Intg i*)) (*Val w*), *s*⟩ −*ε*→ ⟨*THROW ArrayStore*, *s*⟩

| *Red1AAss*:
⟦ *typeof-addr h a* = ⌊*Array-type T n*⌋; 0 <=*s i*; *sint i* < *int n*; *typeof*$_h$ *w* = *Some U*; *P* ⊢ *U* ≤ *T*;
heap-write h a (*ACell* (*nat* (*sint i*))) *w h*′ ⟧
⟹ *uf*,*P*,*t* ⊢*1* ⟨*AAss* (*addr a*) (*Val* (*Intg i*)) (*Val w*), (*h*, *l*)⟩ −{|*WriteMem a* (*ACell* (*nat* (*sint i*))) *w*|}→ ⟨*unit*, (*h*′, *l*)⟩

| *ALength1Red*:
uf,*P*,*t* ⊢*1* ⟨*a*, *s*⟩ −*ta*→ ⟨*a*′, *s*′⟩ ⟹ *uf*,*P*,*t* ⊢*1* ⟨*a*·*length*, *s*⟩ −*ta*→ ⟨*a*′·*length*, *s*′⟩

| *Red1ALength*:
typeof-addr h a = ⌊*Array-type T n*⌋
⟹ *uf*,*P*,*t* ⊢*1* ⟨*addr a*·*length*, (*h*, *xs*)⟩ −*ε*→ ⟨*Val* (*Intg* (*word-of-nat n*)), (*h*, *xs*)⟩

| *Red1ALengthNull*:
uf,*P*,*t* ⊢*1* ⟨*null*·*length*, *s*⟩ −*ε*→ ⟨*THROW NullPointer*, *s*⟩

| *FAcc1Red*:
uf,*P*,*t* ⊢*1* ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩ ⟹ *uf*,*P*,*t* ⊢*1* ⟨*e*·*F*{*D*}, *s*⟩ −*ta*→ ⟨*e*′·*F*{*D*}, *s*′⟩

| *Red1FAcc*:
heap-read h a (*CField D F*) *v*
⟹ *uf*,*P*,*t* ⊢*1* ⟨(*addr a*)·*F*{*D*}, (*h*, *xs*)⟩ −{|*ReadMem a* (*CField D F*) *v*|}→ ⟨*Val v*, (*h*, *xs*)⟩

| *Red1FAccNull*:
uf,*P*,*t* ⊢*1* ⟨*null*·*F*{*D*}, *s*⟩ −*ε*→ ⟨*THROW NullPointer*, *s*⟩

| *FAss1Red1*:
uf,*P*,*t* ⊢*1* ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩ ⟹ *uf*,*P*,*t* ⊢*1* ⟨*e*·*F*{*D*}:=*e2*, *s*⟩ −*ta*→ ⟨*e*′·*F*{*D*}:=*e2*, *s*′⟩

| *FAss1Red2*:
uf,*P*,*t* ⊢*1* ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩ ⟹ *uf*,*P*,*t* ⊢*1* ⟨*FAss* (*Val v*) *F D e*, *s*⟩ −*ta*→ ⟨*Val v*·*F*{*D*}:=*e*′, *s*′⟩

| *Red1FAss*:
heap-write h a (*CField D F*) *v h*′ ⟹
uf,*P*,*t* ⊢*1* ⟨*FAss* (*addr a*) *F D* (*Val v*), (*h*, *l*)⟩ −{|*WriteMem a* (*CField D F*) *v*|}→ ⟨*unit*, (*h*′, *l*)⟩

| *Red1FAssNull*:
uf,*P*,*t* ⊢*1* ⟨*FAss null F D* (*Val v*), *s*⟩ −*ε*→ ⟨*THROW NullPointer*, *s*⟩

| *CAS1Red1*:
uf,*P*,*t* ⊢*1* ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩ ⟹
uf,*P*,*t* ⊢*1* ⟨*e*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *s*⟩ −*ta*→ ⟨*e*′·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *s*′⟩

| *CAS1Red2*:

$uf,P,t \vdash 1\ \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle \Longrightarrow$
$uf,P,t \vdash 1\ \langle Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ e,\ e3),\ s\rangle\ -ta\rightarrow\ \langle Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e3),\ s'\rangle$

| *CAS1Red3*:
$uf,P,t \vdash 1\ \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle \Longrightarrow$
$uf,P,t \vdash 1\ \langle Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v',\ e),\ s\rangle\ -ta\rightarrow\ \langle Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v',\ e'),\ s'\rangle$

| *CAS1Null*:
$uf,P,t \vdash 1\ \langle null{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v,\ Val\ v'),\ s\rangle\ -\varepsilon\rightarrow\ \langle THROW\ NullPointer,\ s\rangle$

| *Red1CASSucceed*:
$[\![\ heap\text{-}read\ h\ a\ (CField\ D\ F)\ v;\ heap\text{-}write\ h\ a\ (CField\ D\ F)\ v'\ h'\ ]\!] \Longrightarrow$
$uf,P,t \vdash 1\ \langle addr\ a{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v,\ Val\ v'),\ (h,\ l)\rangle$
$-\{\!|ReadMem\ a\ (CField\ D\ F)\ v,\ WriteMem\ a\ (CField\ D\ F)\ v'|\!\}\rightarrow$
$\langle true,\ (h',\ l)\rangle$

| *Red1CASFail*:
$[\![\ heap\text{-}read\ h\ a\ (CField\ D\ F)\ v'';\ v \neq v''\ ]\!] \Longrightarrow$
$uf,P,t \vdash 1\ \langle addr\ a{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v,\ Val\ v'),\ (h,\ l)\rangle$
$-\{\!|ReadMem\ a\ (CField\ D\ F)\ v''|\!\}\rightarrow$
$\langle false,\ (h,\ l)\rangle$

| *Call1Obj*:
$uf,P,t \vdash 1\ \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle \Longrightarrow uf,P,t \vdash 1\ \langle e{\cdot}M(es),\ s\rangle\ -ta\rightarrow\ \langle e'{\cdot}M(es),\ s'\rangle$

| *Call1Params*:
$uf,P,t \vdash 1\ \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',s'\rangle \Longrightarrow$
$uf,P,t \vdash 1\ \langle(Val\ v){\cdot}M(es),s\rangle\ -ta\rightarrow\ \langle(Val\ v){\cdot}M(es'),s'\rangle$

| *Red1CallExternal*:
$[\![\ typeof\text{-}addr\ (hp\ s)\ a = \lfloor T\rfloor;\ P \vdash class\text{-}type\text{-}of\ T\ sees\ M{:}Ts\rightarrow Tr = Native\ in\ D;\ P,t \vdash \langle a{\cdot}M(vs),\ hp\ s\rangle\ -ta\rightarrow ext\ \langle va,\ h'\rangle;$
$e' = extRet2J1\ ((addr\ a){\cdot}M(map\ Val\ vs))\ va;\ s' = (h',\ lcl\ s)\ ]\!]$
$\Longrightarrow uf,P,t \vdash 1\ \langle(addr\ a){\cdot}M(map\ Val\ vs),\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$

| *Red1CallNull*:
$uf,P,t \vdash 1\ \langle null{\cdot}M(map\ Val\ vs),\ s\rangle\ -\varepsilon\rightarrow\ \langle THROW\ NullPointer,\ s\rangle$

| *Block1Some*:
$V < length\ x \Longrightarrow uf,P,t \vdash 1\ \langle\{V{:}T{=}\lfloor v\rfloor;\ e\},\ (h,\ x)\rangle\ -\varepsilon\rightarrow\ \langle\{V{:}T{=}None;\ e\},\ (h,\ x[V := v])\rangle$

| *Block1Red*:
$uf,P,t \vdash 1\ \langle e,\ (h,\ x)\rangle\ -ta\rightarrow\ \langle e',\ (h',\ x')\rangle$
$\Longrightarrow uf,P,t \vdash 1\ \langle\{V{:}T{=}None;\ e\},\ (h,\ x)\rangle\ -ta\rightarrow\ \langle\{V{:}T{=}None;\ e'\},\ (h',\ x')\rangle$

| *Red1Block*:
$uf,P,t \vdash 1\ \langle\{V{:}T{=}None;\ Val\ u\},\ s\rangle\ -\varepsilon\rightarrow\ \langle Val\ u,\ s\rangle$

| *Synchronized1Red1*:
$uf,P,t \vdash 1\ \langle o',\ s\rangle\ -ta\rightarrow\ \langle o'',\ s'\rangle \Longrightarrow uf,P,t \vdash 1\ \langle sync_V\ (o')\ e,\ s\rangle\ -ta\rightarrow\ \langle sync_V\ (o'')\ e,\ s'\rangle$

| *Synchronized1Null*:
$V < length\ xs \Longrightarrow uf,P,t \vdash 1\ \langle sync_V\ (null)\ e,\ (h,\ xs)\rangle\ -\varepsilon\rightarrow\ \langle THROW\ NullPointer,\ (h,\ xs[V :=$

Null])⟩

| *Lock1Synchronized*:
V < *length xs* ⟹ *uf*,*P*,*t* ⊢1 ⟨*sync*$_V$ (*addr a*) *e*, (*h*, *xs*)⟩ −{|*Lock*→*a*, *SyncLock a*|}→ ⟨*insync*$_V$ (*a*) *e*, (*h*, *xs*[*V* := *Addr a*])⟩

| *Synchronized1Red2*:
uf,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩ ⟹ *uf*,*P*,*t* ⊢1 ⟨*insync*$_V$ (*a*) *e*, *s*⟩ −*ta*→ ⟨*insync*$_V$ (*a*) *e*′, *s*′⟩

| *Unlock1Synchronized*:
⟦ *xs* ! *V* = *Addr a*′; *V* < *length xs* ⟧ ⟹ *uf*,*P*,*t* ⊢1 ⟨*insync*$_V$ (*a*) (*Val v*), (*h*, *xs*)⟩ −{|*Unlock*→*a*′, *SyncUnlock a*′|}→ ⟨*Val v*, (*h*, *xs*)⟩

| *Unlock1SynchronizedNull*:
⟦ *xs* ! *V* = *Null*; *V* < *length xs* ⟧ ⟹ *uf*,*P*,*t* ⊢1 ⟨*insync*$_V$ (*a*) (*Val v*), (*h*, *xs*)⟩ −*ε*→ ⟨*THROW NullPointer*, (*h*, *xs*)⟩

| *Unlock1SynchronizedFail*:
⟦ *uf*; *xs* ! *V* = *Addr a*′; *V* < *length xs* ⟧
⟹ *uf*,*P*,*t* ⊢1 ⟨*insync*$_V$ (*a*) (*Val v*), (*h*, *xs*)⟩ −{|*UnlockFail*→*a*′|}→ ⟨*THROW IllegalMonitorState*, (*h*, *xs*)⟩

| *Seq1Red*:
uf,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩ ⟹ *uf*,*P*,*t* ⊢1 ⟨*e*;;*e2*, *s*⟩ −*ta*→ ⟨*e*′;;*e2*, *s*′⟩

| *Red1Seq*:
uf,*P*,*t* ⊢1 ⟨*Seq* (*Val v*) *e*, *s*⟩ −*ε*→ ⟨*e*, *s*⟩

| *Cond1Red*:
uf,*P*,*t* ⊢1 ⟨*b*, *s*⟩ −*ta*→ ⟨*b*′, *s*′⟩ ⟹ *uf*,*P*,*t* ⊢1 ⟨*if* (*b*) *e1 else e2*, *s*⟩ −*ta*→ ⟨*if* (*b*′) *e1 else e2*, *s*′⟩

| *Red1CondT*:
uf,*P*,*t* ⊢1 ⟨*if* (*true*) *e1 else e2*, *s*⟩ −*ε*→ ⟨*e1*, *s*⟩

| *Red1CondF*:
uf,*P*,*t* ⊢1 ⟨*if* (*false*) *e1 else e2*, *s*⟩ −*ε*→ ⟨*e2*, *s*⟩

| *Red1While*:
uf,*P*,*t* ⊢1 ⟨*while*(*b*) *c*, *s*⟩ −*ε*→ ⟨*if* (*b*) (*c*;;*while*(*b*) *c*) *else unit*, *s*⟩

| *Throw1Red*:
uf,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩ ⟹ *uf*,*P*,*t* ⊢1 ⟨*throw e*, *s*⟩ −*ta*→ ⟨*throw e*′, *s*′⟩

| *Red1ThrowNull*:
uf,*P*,*t* ⊢1 ⟨*throw null*, *s*⟩ −*ε*→ ⟨*THROW NullPointer*, *s*⟩

| *Try1Red*:
uf,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩ ⟹ *uf*,*P*,*t* ⊢1 ⟨*try e catch*(*C V*) *e2*, *s*⟩ −*ta*→ ⟨*try e*′ *catch*(*C V*) *e2*, *s*′⟩

| *Red1Try*:
uf,*P*,*t* ⊢1 ⟨*try* (*Val v*) *catch*(*C V*) *e2*, *s*⟩ −*ε*→ ⟨*Val v*, *s*⟩

| *Red1TryCatch*:

⟦ *typeof-addr h a* = ⌊*Class-type D*⌋; *P* ⊢ *D* ⪯* *C*; *V* < *length x* ⟧
⟹ *uf*,*P*,*t* ⊢1 ⟨*try* (*Throw a*) *catch*(*C V*) *e2*, (*h*, *x*)⟩ −ε→ ⟨{*V*:*Class C*=*None*; *e2*}, (*h*, *x*[*V* := *Addr a*])⟩

| *Red1TryFail*:
⟦ *typeof-addr* (*hp s*) *a* = ⌊*Class-type D*⌋; ¬ *P* ⊢ *D* ⪯* *C* ⟧
⟹ *uf*,*P*,*t* ⊢1 ⟨*try* (*Throw a*) *catch*(*C V*) *e2*, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩

| *List1Red1*:
uf,*P*,*t* ⊢1 ⟨*e*,*s*⟩ −*ta*→ ⟨*e*′,*s*′⟩ ⟹
uf,*P*,*t* ⊢1 ⟨*e*#*es*,*s*⟩ [−*ta*→] ⟨*e*′#*es*,*s*′⟩

| *List1Red2*:
uf,*P*,*t* ⊢1 ⟨*es*,*s*⟩ [−*ta*→] ⟨*es*′,*s*′⟩ ⟹
uf,*P*,*t* ⊢1 ⟨*Val v* # *es*,*s*⟩ [−*ta*→] ⟨*Val v* # *es*′,*s*′⟩

| *New1ArrayThrow*: *uf*,*P*,*t* ⊢1 ⟨*newA T*⌊*Throw a*⌉, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *Cast1Throw*: *uf*,*P*,*t* ⊢1 ⟨*Cast C* (*Throw a*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *InstanceOf1Throw*: *uf*,*P*,*t* ⊢1 ⟨(*Throw a*) *instanceof T*, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *Bin1OpThrow1*: *uf*,*P*,*t* ⊢1 ⟨(*Throw a*) «*bop*» *e*2, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *Bin1OpThrow2*: *uf*,*P*,*t* ⊢1 ⟨(*Val v*1) «*bop*» (*Throw a*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *LAss1Throw*: *uf*,*P*,*t* ⊢1 ⟨*V*:=(*Throw a*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *AAcc1Throw1*: *uf*,*P*,*t* ⊢1 ⟨(*Throw a*)⌊*i*⌉, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *AAcc1Throw2*: *uf*,*P*,*t* ⊢1 ⟨(*Val v*)⌊*Throw a*⌉, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *AAss1Throw1*: *uf*,*P*,*t* ⊢1 ⟨(*Throw a*)⌊*i*⌉ := *e*, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *AAss1Throw2*: *uf*,*P*,*t* ⊢1 ⟨(*Val v*)⌊*Throw a*⌉ := *e*, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *AAss1Throw3*: *uf*,*P*,*t* ⊢1 ⟨*AAss* (*Val v*) (*Val i*) (*Throw a*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *ALength1Throw*: *uf*,*P*,*t* ⊢1 ⟨(*Throw a*)·*length*, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *FAcc1Throw*: *uf*,*P*,*t* ⊢1 ⟨(*Throw a*)·*F*{*D*}, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *FAss1Throw1*: *uf*,*P*,*t* ⊢1 ⟨(*Throw a*)·*F*{*D*}:=*e*2, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *FAss1Throw2*: *uf*,*P*,*t* ⊢1 ⟨*FAss* (*Val v*) *F D* (*Throw a*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *CAS1Throw*: *uf*,*P*,*t* ⊢1 ⟨*Throw a*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *CAS1Throw2*: *uf*,*P*,*t* ⊢1 ⟨*Val v*·*compareAndSwap*(*D*·*F*, *Throw a*, *e3*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *CAS1Throw3*: *uf*,*P*,*t* ⊢1 ⟨*Val v*·*compareAndSwap*(*D*·*F*, *Val v*′, *Throw a*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *Call1ThrowObj*: *uf*,*P*,*t* ⊢1 ⟨(*Throw a*)·*M*(*es*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *Call1ThrowParams*: ⟦ *es* = *map Val vs* @ *Throw a* # *es*′ ⟧ ⟹ *uf*,*P*,*t* ⊢1 ⟨(*Val v*)·*M*(*es*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *Block1Throw*: *uf*,*P*,*t* ⊢1 ⟨{*V*:*T*=*None*; *Throw a*}, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *Synchronized1Throw1*: *uf*,*P*,*t* ⊢1 ⟨*sync*$_V$ (*Throw a*) *e*, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *Synchronized1Throw2*:
⟦ *xs* ! *V* = *Addr a*′; *V* < *length xs* ⟧
⟹ *uf*,*P*,*t* ⊢1 ⟨*insync*$_V$ (*a*) *Throw ad*, (*h*, *xs*)⟩ −{|*Unlock*→*a*′, *SyncUnlock a*′|}→ ⟨*Throw ad*, (*h*, *xs*)⟩
| *Synchronized1Throw2Fail*:
⟦ *uf*; *xs* ! *V* = *Addr a*′; *V* < *length xs* ⟧
⟹ *uf*,*P*,*t* ⊢1 ⟨*insync*$_V$ (*a*) *Throw ad*, (*h*, *xs*)⟩ −{|*UnlockFail*→*a*′|}→ ⟨*THROW IllegalMonitorState*, (*h*, *xs*)⟩
| *Synchronized1Throw2Null*:
⟦ *xs* ! *V* = *Null*; *V* < *length xs* ⟧
⟹ *uf*,*P*,*t* ⊢1 ⟨*insync*$_V$ (*a*) *Throw ad*, (*h*, *xs*)⟩ −ε→ ⟨*THROW NullPointer*, (*h*, *xs*)⟩
| *Seq1Throw*: *uf*,*P*,*t* ⊢1 ⟨(*Throw a*);;*e*2, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *Cond1Throw*: *uf*,*P*,*t* ⊢1 ⟨*if* (*Throw a*) *e*1 *else e*2, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *Throw1Throw*: *uf*,*P*,*t* ⊢1 ⟨*throw*(*Throw a*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩

inductive-cases *red1-cases*:

uf,*P*,*t* ⊢1 ⟨*new C*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
uf,*P*,*t* ⊢1 ⟨*new T*⌊*e*⌉, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
uf,*P*,*t* ⊢1 ⟨*e* «*bop*» *e′*, *s*⟩ −*ta*→ ⟨*e″*, *s′*⟩
uf,*P*,*t* ⊢1 ⟨*Var V*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
uf,*P*,*t* ⊢1 ⟨*V*:=*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
uf,*P*,*t* ⊢1 ⟨*a*⌊*i*⌉, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
uf,*P*,*t* ⊢1 ⟨*a*⌊*i*⌉ := *e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
uf,*P*,*t* ⊢1 ⟨*a*·*length*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
uf,*P*,*t* ⊢1 ⟨*e*·*F*{*D*}, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
uf,*P*,*t* ⊢1 ⟨*e*·*F*{*D*} := *e2*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
uf,*P*,*t* ⊢1 ⟨*e*·*compareAndSwap*(*D*·*F*, *e′*, *e″*), *s*⟩ −*ta*→ ⟨*e‴*, *s′*⟩
uf,*P*,*t* ⊢1 ⟨*e*·*M*(*es*), *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
uf,*P*,*t* ⊢1 ⟨{*V*:*T*=*vo*; *e*}, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
uf,*P*,*t* ⊢1 ⟨*sync*$_V$ (*o′*) *e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
uf,*P*,*t* ⊢1 ⟨*insync*$_V$ (*a*) *e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
uf,*P*,*t* ⊢1 ⟨*e*;;*e′*, *s*⟩ −*ta*→ ⟨*e″*, *s′*⟩
uf,*P*,*t* ⊢1 ⟨*throw e*, *s* ⟩ −*ta*→ ⟨*e′*, *s′*⟩
uf,*P*,*t* ⊢1 ⟨*try e catch*(*C V*) *e″*, *s* ⟩ −*ta*→ ⟨*e′*, *s′*⟩

inductive *Red1* ::
bool ⇒ *'addr J1-prog* ⇒ *'thread-id* ⇒ (*'addr expr1* × *'addr locals1*) ⇒ (*'addr expr1* × *'addr locals1*) *list* ⇒ *'heap*
⇒ (*'addr*, *'thread-id*, *'heap*) *J1-thread-action*
⇒ (*'addr expr1* × *'addr locals1*) ⇒ (*'addr expr1* × *'addr locals1*) *list* ⇒ *'heap* ⇒ *bool*
(‹-,-,- ⊢1 ((1⟨-′/-,/-⟩) −-→/ (1⟨-′/-,/-⟩))› [51,51,0,0,0,0,0,0,0,0] 81)
for *uf* :: *bool* **and** *P* :: *'addr J1-prog* **and** *t* :: *'thread-id*
where

red1Red:
uf,*P*,*t* ⊢1 ⟨*e*, (*h*, *x*)⟩ −*ta*→ ⟨*e′*, (*h′*, *x′*)⟩
⟹ *uf*,*P*,*t* ⊢1 ⟨(*e*, *x*)/*exs*, *h*⟩ −*extTA2J1 P ta*→ ⟨(*e′*, *x′*)/*exs*, *h′*⟩

| *red1Call*:
⟦ *call1 e* = ⌊(*a*, *M*, *vs*)⌋; *typeof-addr h a* = ⌊*U*⌋;
P ⊢ *class-type-of U sees M*:*Ts*→*T* = ⌊*body*⌋ *in D*;
size vs = *size Ts* ⟧
⟹ *uf*,*P*,*t* ⊢1 ⟨(*e*, *x*)/*exs*, *h*⟩ −*ε*→ ⟨(*blocks1 0* (*Class D*#*Ts*) *body*, *Addr a* # *vs* @ *replicate* (*max-vars body*) *undefined-value*)/(*e*, *x*)#*exs*, *h*⟩

| *red1Return*:
final e′ ⟹ *uf*,*P*,*t* ⊢1 ⟨(*e′*, *x′*)/(*e*, *x*)#*exs*, *h*⟩ −*ε*→ ⟨(*inline-call e′ e*, *x*)/*exs*, *h*⟩

abbreviation *mred1g* :: *bool* ⇒ *'addr J1-prog* ⇒ (*'addr*,*'thread-id*,(*'addr expr1* × *'addr locals1*) × (*'addr expr1* × *'addr locals1*) *list*,*'heap*,*'addr*,(*'addr*, *'thread-id*) *obs-event*) *semantics*
where *mred1g uf P* ≡ λ*t* ((*ex*, *exs*), *h*) *ta* ((*ex′*, *exs′*), *h′*). *uf*,*P*,*t* ⊢1 ⟨*ex*/*exs*, *h*⟩ −*ta*→ ⟨*ex′*/*exs′*, *h′*⟩

abbreviation *mred1′* ::
'addr J1-prog ⇒ (*'addr*,*'thread-id*,(*'addr expr1* × *'addr locals1*) × (*'addr expr1* × *'addr locals1*) *list*,*'heap*,*'addr*,(*'addr*, *'thread-id*) *obs-event*) *semantics*
where *mred1′* ≡ *mred1g False*

abbreviation *mred1* ::
'addr J1-prog ⇒ (*'addr*,*'thread-id*,(*'addr expr1* × *'addr locals1*) × (*'addr expr1* × *'addr locals1*) *list*,*'heap*,*'addr*,(*'addr*, *'thread-id*) *obs-event*) *semantics*

where *mred1* ≡ *mred1g True*

lemma *red1-preserves-len*: *uf*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩ ⟹ *length* (*lcl s*′) = *length* (*lcl s*)
and *reds1-preserves-len*: *uf*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩ ⟹ *length* (*lcl s*′) = *length* (*lcl s*)
by(*induct rule*: *red1-reds1.inducts*)(*auto*)

lemma *reds1-preserves-elen*: *uf*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩ ⟹ *length es*′ = *length es*
by(*induct es arbitrary*: *es*′)(*auto elim*: *reds1.cases*)

lemma *red1-Val-iff* [*iff*]:
¬ *uf*,*P*,*t* ⊢1 ⟨*Val v*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩
by(*auto elim*: *red1.cases*)

lemma *red1-Throw-iff* [*iff*]:
¬ *uf*,*P*,*t* ⊢1 ⟨*Throw a*, *xs*⟩ −*ta*→ ⟨*e*′, *s*′⟩
by(*auto elim*: *red1.cases*)

lemma *reds1-Nil-iff* [*iff*]:
¬ *uf*,*P*,*t* ⊢1 ⟨[], *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩
by(*auto elim*: *reds1.cases*)

lemma *reds1-Val-iff* [*iff*]:
¬ *uf*,*P*,*t* ⊢1 ⟨*map Val vs*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩
by(*induct vs arbitrary*: *es*′)(*auto elim*: *reds1.cases*)

lemma *reds1-map-Val-Throw-iff* [*iff*]:
¬ *uf*,*P*,*t* ⊢1 ⟨*map Val vs* @ *Throw a* # *es*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩
by(*induct vs arbitrary*: *es*′)(*auto elim*: *reds1.cases elim*!: *red1-cases*)

lemma *red1-max-vars-decr*: *uf*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩ ⟹ *max-vars e*′ ≤ *max-vars e*
and *reds1-max-varss-decr*: *uf*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩ ⟹ *max-varss es*′ ≤ *max-varss es*
by(*induct rule*: *red1-reds1.inducts*)(*auto*)

lemma *red1-new-thread-heap*: ⟦*uf*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩; *NewThread t*′ *ex h* ∈ *set* {|*ta*|}*t* ⟧ ⟹ *h* = *hp s*′
and *reds1-new-thread-heap*: ⟦*uf*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩; *NewThread t*′ *ex h* ∈ *set* {|*ta*|}*t* ⟧ ⟹ *h* = *hp s*′
apply(*induct rule*: *red1-reds1.inducts*)
apply(*fastforce dest*: *red-ext-new-thread-heap simp add*: *ta-upd-simps*)+
done

lemma *red1-new-threadD*:
⟦ *uf*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩; *NewThread t*′ *x H* ∈ *set* {|*ta*|}*t* ⟧
⟹ ∃ *a M vs va T Ts Tr D*. *P*,*t* ⊢ ⟨*a*·*M*(*vs*), *hp s*⟩ −*ta*→*ext* ⟨*va*, *hp s*′⟩ ∧ *typeof-addr* (*hp s*) *a* = ⌊*T*⌋ ∧ *P* ⊢ *class-type-of T sees M*:*Ts*→*Tr* = *Native in D*
and *reds1-new-threadD*:
⟦ *uf*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩; *NewThread t*′ *x H* ∈ *set* {|*ta*|}*t* ⟧
⟹ ∃ *a M vs va T Ts Tr D*. *P*,*t* ⊢ ⟨*a*·*M*(*vs*), *hp s*⟩ −*ta*→*ext* ⟨*va*, *hp s*′⟩ ∧ *typeof-addr* (*hp s*) *a* = ⌊*T*⌋ ∧ *P* ⊢ *class-type-of T sees M*:*Ts*→*Tr* = *Native in D*
by(*induct rule*: *red1-reds1.inducts*)(*fastforce simp add*: *ta-upd-simps*)+

lemma *red1-call-synthesized*: ⟦ *uf*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩; *call1 e* = ⌊*aMvs*⌋ ⟧ ⟹ *synthesized-call P* (*hp s*) *aMvs*
and *reds1-calls-synthesized*: ⟦ *uf*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩; *calls1 es* = ⌊*aMvs*⌋ ⟧ ⟹ *synthe-*

sized-call P (hp s) aMvs
apply(*induct rule*: *red1-reds1.inducts*)
apply(*auto split*: *if-split-asm simp add*: *is-vals-conv append-eq-map-conv synthesized-call-conv*)
apply *blast*
done

lemma *red1-preserves-B*: ⟦ *uf*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; ℬ *e n*⟧ ⟹ ℬ *e' n*
and *reds1-preserves-Bs*: ⟦ *uf*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; ℬ*s es n*⟧ ⟹ ℬ*s es' n*
by(*induct arbitrary*: *n* **and** *n rule*: *red1-reds1.inducts*)(*auto*)

end

context *J1-heap* **begin**

lemma *red1-hext-incr*: *uf*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩ ⟹ *hext* (*hp s*) (*hp s'*)
and *reds1-hext-incr*: *uf*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩ ⟹ *hext* (*hp s*) (*hp s'*)
by(*induct rule*: *red1-reds1.inducts*)(*auto intro*: *hext-heap-ops red-external-hext*)

lemma *Red1-hext-incr*: *uf*,*P*,*t* ⊢1 ⟨*ex*/*exs*,*h*⟩ −*ta*→ ⟨*ex'*/*exs'*,*h'*⟩ ⟹ *h* ⊴ *h'*
by(*auto elim*!: *Red1.cases dest*: *red1-hext-incr*)

end

7.6.1 Silent moves

context *J1-heap-base* **begin**

primrec *τmove1* :: *'m prog* ⇒ *'heap* ⇒ (*'a*, *'b*, *'addr*) *exp* ⇒ *bool*
and *τmoves1* :: *'m prog* ⇒ *'heap* ⇒ (*'a*, *'b*, *'addr*) *exp list* ⇒ *bool*
where
τmove1 P h (*new C*) ⟷ *False*
| *τmove1 P h* (*newA T*⌊*e*⌋) ⟷ *τmove1 P h e* ∨ (∃ *a*. *e* = *Throw a*)
| *τmove1 P h* (*Cast U e*) ⟷ *τmove1 P h e* ∨ *final e*
| *τmove1 P h* (*e instanceof T*) ⟷ *τmove1 P h e* ∨ *final e*
| *τmove1 P h* (*e* «*bop*» *e'*) ⟷ *τmove1 P h e* ∨ (∃ *a*. *e* = *Throw a*) ∨ (∃ *v*. *e* = *Val v* ∧ (*τmove1 P h e'* ∨ *final e'*))
| *τmove1 P h* (*Val v*) ⟷ *False*
| *τmove1 P h* (*Var V*) ⟷ *True*
| *τmove1 P h* (*V* := *e*) ⟷ *τmove1 P h e* ∨ *final e*
| *τmove1 P h* (*a*⌊*i*⌋) ⟷ *τmove1 P h a* ∨ (∃ *ad*. *a* = *Throw ad*) ∨ (∃ *v*. *a* = *Val v* ∧ (*τmove1 P h i* ∨ (∃ *a*. *i* = *Throw a*)))
| *τmove1 P h* (*AAss a i e*) ⟷ *τmove1 P h a* ∨ (∃ *ad*. *a* = *Throw ad*) ∨ (∃ *v*. *a* = *Val v* ∧ (*τmove1 P h i* ∨ (∃ *a*. *i* = *Throw a*) ∨ (∃ *v*. *i* = *Val v* ∧ (*τmove1 P h e* ∨ (∃ *a*. *e* = *Throw a*))))))
| *τmove1 P h* (*a*·*length*) ⟷ *τmove1 P h a* ∨ (∃ *ad*. *a* = *Throw ad*)
| *τmove1 P h* (*e*·*F*{*D*}) ⟷ *τmove1 P h e* ∨ (∃ *a*. *e* = *Throw a*)
| *τmove1 P h* (*FAss e F D e'*) ⟷ *τmove1 P h e* ∨ (∃ *a*. *e* = *Throw a*) ∨ (∃ *v*. *e* = *Val v* ∧ (*τmove1 P h e'* ∨ (∃ *a*. *e'* = *Throw a*)))
| *τmove1 P h* (*e*·*compareAndSwap*(*D*·*F*, *e'*, *e''*)) ⟷ *τmove1 P h e* ∨ (∃ *a*. *e* = *Throw a*) ∨ (∃ *v*. *e* = *Val v* ∧
τmove1 P h e' ∨ (∃ *a*. *e'* = *Throw a*) ∨ (∃ *v*. *e'* = *Val v* ∧ (*τmove1 P h e''* ∨ (∃ *a*. *e''* = *Throw a*)))))
| *τmove1 P h* (*e*·*M*(*es*)) ⟷ *τmove1 P h e* ∨ (∃ *a*. *e* = *Throw a*) ∨ (∃ *v*. *e* = *Val v* ∧
(*τmoves1 P h es* ∨ (∃ *vs a es'*. *es* = *map Val vs* @ *Throw a* # *es'*) ∨
(∃ *vs*. *es* = *map Val vs* ∧ (*v* = *Null* ∨ (∀ *T C Ts Tr D*. *typeof*$_h$ *v* = ⌊*T*⌋ ⟶ *class-type-of' T* =

$\lfloor C \rfloor \longrightarrow P \vdash C$ *sees* M:$Ts \to Tr$ = *Native in* $D \longrightarrow \tau$*external-defs* $D\ M$)))))
| τ*move1* $P\ h$ ({ V:T=vo; e}) $\longleftrightarrow$ $vo \neq$ *None* $\vee\ \tau$*move1* $P\ h\ e \vee$ *final* e
| τ*move1* $P\ h$ ($sync_{V'}(e)\ e'$) $\longleftrightarrow\ \tau$*move1* $P\ h\ e \vee (\exists\, a.\ e =$ *Throw* $a)$
| τ*move1* $P\ h$ ($insync_{V'}(ad)\ e$) $\longleftrightarrow\ \tau$*move1* $P\ h\ e$
| τ*move1* $P\ h$ (e;;e') $\longleftrightarrow\ \tau$*move1* $P\ h\ e \vee$ *final* e
| τ*move1* $P\ h$ (*if* (e) e' *else* e'') $\longleftrightarrow\ \tau$*move1* $P\ h\ e \vee$ *final* e
| τ*move1* $P\ h$ (*while* (e) e') = *True*
| τ*move1* $P\ h$ (*throw* e) $\longleftrightarrow\ \tau$*move1* $P\ h\ e \vee (\exists\, a.\ e =$ *Throw* $a) \vee e =$ *null*
| τ*move1* $P\ h$ (*try* e *catch*($C\ V$) e') $\longleftrightarrow\ \tau$*move1* $P\ h\ e \vee$ *final* e

| τ*moves1* $P\ h$ [] $\longleftrightarrow$ *False*
| τ*moves1* $P\ h$ (e # es) $\longleftrightarrow\ \tau$*move1* $P\ h\ e \vee (\exists\, v.\ e =$ *Val* $v \wedge\ \tau$*moves1* $P\ h\ es$)

fun τ*Move1* :: $'m$ *prog* $\Rightarrow$ $'heap \Rightarrow ((('a, 'b, 'addr)\ exp \times 'c) \times ((('a, 'b, 'addr)\ exp \times 'd)\ list \Rightarrow bool$
where
τ*Move1* $P\ h\ ((e, x), exs) = (\tau$*move1* $P\ h\ e \vee$ *final* $e)$

definition τ*red1g* :: *bool* $\Rightarrow$ $'addr$ *J1-prog* $\Rightarrow$ $'thread\text{-}id \Rightarrow 'heap \Rightarrow ('addr\ expr1 \times 'addr\ locals1) \Rightarrow$ $('addr\ expr1 \times 'addr\ locals1) \Rightarrow$ *bool*
where τ*red1g uf P t h exs e'xs'* = (*uf*,*P*,*t* $\vdash$1 $\langle$*fst exs*, (*h*, *snd exs*)$\rangle$ $-\varepsilon\to$ $\langle$*fst e'xs'*, (*h*, *snd e'xs'*)$\rangle \wedge$ τ*move1 P h* (*fst exs*))

definition τ*reds1g* ::
bool $\Rightarrow$ $'addr$ *J1-prog* $\Rightarrow$ $'thread\text{-}id \Rightarrow 'heap \Rightarrow ('addr\ expr1\ list \times 'addr\ locals1) \Rightarrow ('addr\ expr1\ list$ $\times$ $'addr\ locals1) \Rightarrow$ *bool*
where
τ*reds1g uf P t h esxs es'xs'* =
(*uf*,*P*,*t* $\vdash$1 $\langle$*fst esxs*, (*h*, *snd esxs*)$\rangle$ [$-\varepsilon\to$] $\langle$*fst es'xs'*, (*h*, *snd es'xs'*)$\rangle \wedge\ \tau$*moves1 P h* (*fst esxs*))

abbreviation τ*red1gt* ::
bool $\Rightarrow$ $'addr$ *J1-prog* $\Rightarrow$ $'thread\text{-}id \Rightarrow 'heap \Rightarrow ('addr\ expr1 \times 'addr\ locals1) \Rightarrow ('addr\ expr1 \times 'addr$ *locals1*) $\Rightarrow$ *bool*
where τ*red1gt uf P t h* $\equiv$ (τ*red1g uf P t h*)^++

abbreviation τ*reds1gt* ::
bool $\Rightarrow$ $'addr$ *J1-prog* $\Rightarrow$ $'thread\text{-}id \Rightarrow 'heap \Rightarrow ('addr\ expr1\ list \times 'addr\ locals1) \Rightarrow ('addr\ expr1\ list$ $\times$ $'addr\ locals1) \Rightarrow$ *bool*
where τ*reds1gt uf P t h* $\equiv$ (τ*reds1g uf P t h*)^++

abbreviation τ*red1gr* ::
bool $\Rightarrow$ $'addr$ *J1-prog* $\Rightarrow$ $'thread\text{-}id \Rightarrow 'heap \Rightarrow ('addr\ expr1 \times 'addr\ locals1) \Rightarrow ('addr\ expr1 \times 'addr$ *locals1*) $\Rightarrow$ *bool*
where τ*red1gr uf P t h* $\equiv$ (τ*red1g uf P t h*)^**

abbreviation τ*reds1gr* ::
bool $\Rightarrow$ $'addr$ *J1-prog* $\Rightarrow$ $'thread\text{-}id \Rightarrow 'heap \Rightarrow ('addr\ expr1\ list \times 'addr\ locals1) \Rightarrow ('addr\ expr1\ list$ $\times$ $'addr\ locals1) \Rightarrow$ *bool*
where τ*reds1gr uf P t h* $\equiv$ (τ*reds1g uf P t h*)^**

definition τ*Red1g* ::
bool $\Rightarrow$ $'addr$ *J1-prog* $\Rightarrow$ $'thread\text{-}id \Rightarrow 'heap \Rightarrow ('addr\ expr1 \times 'addr\ locals1) \times (('addr\ expr1 \times 'addr$ *locals1*) *list*)
$\Rightarrow$ ($'addr\ expr1 \times 'addr\ locals1$) $\times$ (($'addr\ expr1 \times 'addr\ locals1$) *list*) $\Rightarrow$ *bool*
where τ*Red1g uf P t h exexs ex'exs'* = (*uf*,*P*,*t* $\vdash$1 $\langle$*fst exexs*/*snd exexs*, *h*$\rangle$ $-\varepsilon\to$ $\langle$*fst ex'exs'*/*snd*

ex'exs', *h*⟩ ∧ *τMove1 P h exexs*)

abbreviation *τRed1gt* ::
bool ⇒ *'addr J1-prog* ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr1* × *'addr locals1*) × ((*'addr expr1* × *'addr locals1*) *list*)
⇒ (*'addr expr1* × *'addr locals1*) × ((*'addr expr1* × *'addr locals1*) *list*) ⇒ *bool*
where *τRed1gt uf P t h* ≡ (*τRed1g uf P t h*)^++

abbreviation *τRed1gr* ::
bool ⇒ *'addr J1-prog* ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr1* × *'addr locals1*) × ((*'addr expr1* × *'addr locals1*) *list*)
⇒ (*'addr expr1* × *'addr locals1*) × ((*'addr expr1* × *'addr locals1*) *list*) ⇒ *bool*
where *τRed1gr uf P t h* ≡ (*τRed1g uf P t h*)^**

abbreviation *τred1* ::
'addr J1-prog ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr1* × *'addr locals1*) ⇒ (*'addr expr1* × *'addr locals1*)
⇒ *bool*
where *τred1* ≡ *τred1g True*

abbreviation *τreds1* ::
'addr J1-prog ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr1 list* × *'addr locals1*) ⇒ (*'addr expr1 list* × *'addr locals1*) ⇒ *bool*
where *τreds1* ≡ *τreds1g True*

abbreviation *τred1t* ::
'addr J1-prog ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr1* × *'addr locals1*) ⇒ (*'addr expr1* × *'addr locals1*)
⇒ *bool*
where *τred1t* ≡ *τred1gt True*

abbreviation *τreds1t* ::
'addr J1-prog ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr1 list* × *'addr locals1*) ⇒ (*'addr expr1 list* × *'addr locals1*) ⇒ *bool*
where *τreds1t* ≡ *τreds1gt True*

abbreviation *τred1r* ::
'addr J1-prog ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr1* × *'addr locals1*) ⇒ (*'addr expr1* × *'addr locals1*)
⇒ *bool*
where *τred1r* ≡ *τred1gr True*

abbreviation *τreds1r* ::
'addr J1-prog ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr1 list* × *'addr locals1*) ⇒ (*'addr expr1 list* × *'addr locals1*) ⇒ *bool*
where *τreds1r* ≡ *τreds1gr True*

abbreviation *τRed1* ::
'addr J1-prog ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr1* × *'addr locals1*) × ((*'addr expr1* × *'addr locals1*) *list*)
⇒ (*'addr expr1* × *'addr locals1*) × ((*'addr expr1* × *'addr locals1*) *list*) ⇒ *bool*
where *τRed1* ≡ *τRed1g True*

abbreviation *τRed1t* ::
'addr J1-prog ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr1* × *'addr locals1*) × ((*'addr expr1* × *'addr locals1*) *list*)
⇒ (*'addr expr1* × *'addr locals1*) × ((*'addr expr1* × *'addr locals1*) *list*) ⇒ *bool*

where $\tau Red1t \equiv \tau Red1gt\ True$

abbreviation $\tau Red1r$ ::
$'addr\ J1\text{-}prog \Rightarrow 'thread\text{-}id \Rightarrow 'heap \Rightarrow ('addr\ expr1 \times 'addr\ locals1) \times (('addr\ expr1 \times 'addr\ locals1)\ list)$
$\Rightarrow ('addr\ expr1 \times 'addr\ locals1) \times (('addr\ expr1 \times 'addr\ locals1)\ list) \Rightarrow bool$
where $\tau Red1r \equiv \tau Red1gr\ True$

abbreviation $\tau red1'$ ::
$'addr\ J1\text{-}prog \Rightarrow 'thread\text{-}id \Rightarrow 'heap \Rightarrow ('addr\ expr1 \times 'addr\ locals1) \Rightarrow ('addr\ expr1 \times 'addr\ locals1) \Rightarrow bool$
where $\tau red1' \equiv \tau red1g\ False$

abbreviation $\tau reds1'$ ::
$'addr\ J1\text{-}prog \Rightarrow 'thread\text{-}id \Rightarrow 'heap \Rightarrow ('addr\ expr1\ list \times 'addr\ locals1) \Rightarrow ('addr\ expr1\ list \times 'addr\ locals1) \Rightarrow bool$
where $\tau reds1' \equiv \tau reds1g\ False$

abbreviation $\tau red1't$ ::
$'addr\ J1\text{-}prog \Rightarrow 'thread\text{-}id \Rightarrow 'heap \Rightarrow ('addr\ expr1 \times 'addr\ locals1) \Rightarrow ('addr\ expr1 \times 'addr\ locals1) \Rightarrow bool$
where $\tau red1't \equiv \tau red1gt\ False$

abbreviation $\tau reds1't$ ::
$'addr\ J1\text{-}prog \Rightarrow 'thread\text{-}id \Rightarrow 'heap \Rightarrow ('addr\ expr1\ list \times 'addr\ locals1) \Rightarrow ('addr\ expr1\ list \times 'addr\ locals1) \Rightarrow bool$
where $\tau reds1't \equiv \tau reds1gt\ False$

abbreviation $\tau red1'r$ ::
$'addr\ J1\text{-}prog \Rightarrow 'thread\text{-}id \Rightarrow 'heap \Rightarrow ('addr\ expr1 \times 'addr\ locals1) \Rightarrow ('addr\ expr1 \times 'addr\ locals1) \Rightarrow bool$
where $\tau red1'r \equiv \tau red1gr\ False$

abbreviation $\tau reds1'r$ ::
$'addr\ J1\text{-}prog \Rightarrow 'thread\text{-}id \Rightarrow 'heap \Rightarrow ('addr\ expr1\ list \times 'addr\ locals1) \Rightarrow ('addr\ expr1\ list \times 'addr\ locals1) \Rightarrow bool$
where $\tau reds1'r \equiv \tau reds1gr\ False$

abbreviation $\tau Red1'$ ::
$'addr\ J1\text{-}prog \Rightarrow 'thread\text{-}id \Rightarrow 'heap \Rightarrow ('addr\ expr1 \times 'addr\ locals1) \times (('addr\ expr1 \times 'addr\ locals1)\ list)$
$\Rightarrow ('addr\ expr1 \times 'addr\ locals1) \times (('addr\ expr1 \times 'addr\ locals1)\ list) \Rightarrow bool$
where $\tau Red1' \equiv \tau Red1g\ False$

abbreviation $\tau Red1't$ ::
$'addr\ J1\text{-}prog \Rightarrow 'thread\text{-}id \Rightarrow 'heap \Rightarrow ('addr\ expr1 \times 'addr\ locals1) \times (('addr\ expr1 \times 'addr\ locals1)\ list)$
$\Rightarrow ('addr\ expr1 \times 'addr\ locals1) \times (('addr\ expr1 \times 'addr\ locals1)\ list) \Rightarrow bool$
where $\tau Red1't \equiv \tau Red1gt\ False$

abbreviation $\tau Red1'r$ ::
$'addr\ J1\text{-}prog \Rightarrow 'thread\text{-}id \Rightarrow 'heap \Rightarrow ('addr\ expr1 \times 'addr\ locals1) \times (('addr\ expr1 \times 'addr\ locals1)\ list)$
$\Rightarrow ('addr\ expr1 \times 'addr\ locals1) \times (('addr\ expr1 \times 'addr\ locals1)\ list) \Rightarrow bool$

where $\tau Red1'r \equiv \tau Red1gr\ False$

abbreviation $\tau MOVE1$ ::
$'m\ prog \Rightarrow ((('addr\ expr1 \times 'addr\ locals1) \times ('addr\ expr1 \times 'addr\ locals1)\ list) \times 'heap, ('addr, 'thread\text{-}id, 'heap)\ J1\text{-}thread\text{-}action)\ trsys$
where $\tau MOVE1\ P \equiv \lambda(exexs, h)\ ta\ s.\ \tau Move1\ P\ h\ exexs \wedge ta = \varepsilon$

lemma $\tau move1\text{-}\tau moves1\text{-}intros$:
fixes $e :: ('a, 'b, 'addr)\ exp$ **and** $es :: ('a, 'b, 'addr)\ exp\ list$
shows $\tau move1NewArray$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (newA\ T\lfloor e\rceil)$
and $\tau move1Cast$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (Cast\ U\ e)$
and $\tau move1CastRed$: $\tau move1\ P\ h\ (Cast\ U\ (Val\ v))$
and $\tau move1InstanceOf$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (e\ instanceof\ U)$
and $\tau move1InstanceOfRed$: $\tau move1\ P\ h\ ((Val\ v)\ instanceof\ U)$
and $\tau move1BinOp1$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (e\text{«}bop\text{»}e')$
and $\tau move1BinOp2$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (Val\ v\text{«}bop\text{»}e)$
and $\tau move1BinOp$: $\tau move1\ P\ h\ (Val\ v\text{«}bop\text{»}Val\ v')$
and $\tau move1Var$: $\tau move1\ P\ h\ (Var\ V)$
and $\tau move1LAss$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (V := e)$
and $\tau move1LAssRed$: $\tau move1\ P\ h\ (V := Val\ v)$
and $\tau move1AAcc1$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (e\lfloor e'\rceil)$
and $\tau move1AAcc2$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (Val\ v\lfloor e\rceil)$
and $\tau move1AAss1$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (AAss\ e\ e'\ e'')$
and $\tau move1AAss2$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (AAss\ (Val\ v)\ e\ e')$
and $\tau move1AAss3$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (AAss\ (Val\ v)\ (Val\ v')\ e)$
and $\tau move1ALength$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (e{\cdot}length)$
and $\tau move1FAcc$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (e{\cdot}F\{D\})$
and $\tau move1FAss1$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (FAss\ e\ F\ D\ e')$
and $\tau move1FAss2$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (FAss\ (Val\ v)\ F\ D\ e)$
and $\tau move1CAS1$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e''))$
and $\tau move1CAS2$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ e,\ e''))$
and $\tau move1CAS3$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v',\ e))$
and $\tau move1CallObj$: $\tau move1\ P\ h\ obj \Longrightarrow \tau move1\ P\ h\ (obj{\cdot}M(ps))$
and $\tau move1CallParams$: $\tau moves1\ P\ h\ ps \Longrightarrow \tau move1\ P\ h\ (Val\ v{\cdot}M(ps))$
and $\tau move1Call$: $(\bigwedge T\ C\ Ts\ Tr\ D.\ [\![\ typeof_h\ v = \lfloor T\rfloor;\ class\text{-}type\text{-}of'\ T = \lfloor C\rfloor;\ P \vdash C\ sees\ M{:}Ts{\to}Tr = Native\ in\ D\ ]\!] \Longrightarrow \tau external\text{-}defs\ D\ M) \Longrightarrow \tau move1\ P\ h\ (Val\ v{\cdot}M(map\ Val\ vs))$
and $\tau move1BlockSome$: $\tau move1\ P\ h\ \{V{:}T{=}\lfloor v\rfloor;\ e\}$
and $\tau move1Block$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ \{V{:}T{=}None;\ e\}$
and $\tau move1BlockRed$: $\tau move1\ P\ h\ \{V{:}T{=}None;\ Val\ v\}$
and $\tau move1Sync$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (sync_{V'}\ (e)\ e')$
and $\tau move1InSync$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (insync_{V'}\ (a)\ e)$
and $\tau move1Seq$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (e;;e')$
and $\tau move1SeqRed$: $\tau move1\ P\ h\ (Val\ v;;\ e)$
and $\tau move1Cond$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (if\ (e)\ e1\ else\ e2)$
and $\tau move1CondRed$: $\tau move1\ P\ h\ (if\ (Val\ v)\ e1\ else\ e2)$
and $\tau move1WhileRed$: $\tau move1\ P\ h\ (while\ (c)\ e)$
and $\tau move1Throw$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (throw\ e)$
and $\tau move1ThrowNull$: $\tau move1\ P\ h\ (throw\ null)$
and $\tau move1Try$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (try\ e\ catch(C\ V)\ e'')$
and $\tau move1TryRed$: $\tau move1\ P\ h\ (try\ Val\ v\ catch(C\ V)\ e)$
and $\tau move1TryThrow$: $\tau move1\ P\ h\ (try\ Throw\ a\ catch(C\ V)\ e)$
and $\tau move1NewArrayThrow$: $\tau move1\ P\ h\ (newA\ T\lfloor Throw\ a\rceil)$
and $\tau move1CastThrow$: $\tau move1\ P\ h\ (Cast\ T\ (Throw\ a))$
and $\tau move1InstanceOfThrow$: $\tau move1\ P\ h\ ((Throw\ a)\ instanceof\ T)$

and *τmove1BinOpThrow1*: *τmove1 P h (Throw a «bop» e2)*
and *τmove1BinOpThrow2*: *τmove1 P h (Val v «bop» Throw a)*
and *τmove1LAssThrow*: *τmove1 P h (V:=(Throw a))*
and *τmove1AAccThrow1*: *τmove1 P h (Throw a⌊e⌉)*
and *τmove1AAccThrow2*: *τmove1 P h (Val v⌊Throw a⌉)*
and *τmove1AAssThrow1*: *τmove1 P h (AAss (Throw a) e e′)*
and *τmove1AAssThrow2*: *τmove1 P h (AAss (Val v) (Throw a) e′)*
and *τmove1AAssThrow3*: *τmove1 P h (AAss (Val v) (Val v′) (Throw a))*
and *τmove1ALengthThrow*: *τmove1 P h (Throw a·length)*
and *τmove1FAccThrow*: *τmove1 P h (Throw a·F{D})*
and *τmove1FAssThrow1*: *τmove1 P h (Throw a·F{D} := e)*
and *τmove1FAssThrow2*: *τmove1 P h (FAss (Val v) F D (Throw a))*
and *τmove1CASThrow1*: *τmove1 P h (CompareAndSwap (Throw a) D F e e′)*
and *τmove1CASThrow2*: *τmove1 P h (CompareAndSwap (Val v) D F (Throw a) e′)*
and *τmove1CASThrow3*: *τmove1 P h (CompareAndSwap (Val v) D F (Val v′) (Throw a))*
and *τmove1CallThrowObj*: *τmove1 P h (Throw a·M(es))*
and *τmove1CallThrowParams*: *τmove1 P h (Val v·M(map Val vs @ Throw a # es))*
and *τmove1BlockThrow*: *τmove1 P h {V:T=None; Throw a}*
and *τmove1SyncThrow*: *τmove1 P h (sync$_{V'}$ (Throw a) e)*
and *τmove1SeqThrow*: *τmove1 P h (Throw a;;e)*
and *τmove1CondThrow*: *τmove1 P h (if (Throw a) e1 else e2)*
and *τmove1ThrowThrow*: *τmove1 P h (throw (Throw a))*

and *τmoves1Hd*: *τmove1 P h e ⟹ τmoves1 P h (e # es)*
and *τmoves1Tl*: *τmoves1 P h es ⟹ τmoves1 P h (Val v # es)*
by *fastforce+*

lemma *τmoves1-map-Val* [*dest*!]:
τmoves1 P h (map Val es) ⟹ False
by(*induct es*)(*auto*)

lemma *τmoves1-map-Val-ThrowD* [*simp*]: *τmoves1 P h (map Val vs @ Throw a # es) = False*
by(*induct vs*)(*fastforce*)+

lemma fixes *e* :: (*'a*, *'b*, *'addr*) *exp* **and** *es* :: (*'a*, *'b*, *'addr*) *exp list*
shows *τmove1-not-call1*:
call1 e = ⌊(a, M, vs)⌋ ⟹ τmove1 P h e ⟷ (synthesized-call P h (a, M, vs) ⟶ τexternal′ P h a M)
and *τmoves1-not-calls1*:
calls1 es = ⌊(a, M, vs)⌋ ⟹ τmoves1 P h es ⟷ (synthesized-call P h (a, M, vs) ⟶ τexternal′ P h a M)
apply(*induct e* **and** *es rule*: *call1.induct calls1.induct*)
apply(*auto split*: *if-split-asm simp add*: *is-vals-conv*)
apply(*fastforce simp add*: *synthesized-call-def map-eq-append-conv τexternal′-def τexternal-def dest*: *sees-method-fun*)+
done

lemma *red1-τ-taD*: ⟦ *uf*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩; *τmove1 P (hp s) e* ⟧ ⟹ *ta* = *ε*
and *reds1-τ-taD*: ⟦ *uf*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es′*, *s′*⟩; *τmoves1 P (hp s) es* ⟧ ⟹ *ta* = *ε*
apply(*induct rule*: *red1-reds1.inducts*)
apply(*fastforce simp add*: *map-eq-append-conv τexternal′-def τexternal-def dest*: *τexternal′-red-external-TA-empty*)+
done

lemma *τmove1-heap-unchanged*: ⟦ *uf*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩; *τmove1 P (hp s) e* ⟧ ⟹ *hp s′* =

hp s
and *τmoves1-heap-unchanged*: ⟦ *uf*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *τmoves1 P* (*hp s*) *es* ⟧ ⟹ *hp s'* = *hp s*
apply(*induct rule*: *red1-reds1.inducts*)
apply(*auto*)
apply(*fastforce simp add*: *map-eq-append-conv τexternal'-def τexternal-def dest*: *τexternal'-red-external-heap-unch*
done

lemma *τMove1-iff*:
τMove1 P h exexs ⟷ (*let* ((*e*, -), -) = *exexs in τmove1 P h e* ∨ *final e*)
by(*cases exexs*)(*auto*)

lemma *τred1-iff* [*iff*]:
τred1g uf P t h (*e*, *xs*) (*e'*, *xs'*) = (*uf*,*P*,*t* ⊢1 ⟨*e*, (*h*, *xs*)⟩ −*ε*→ ⟨*e'*, (*h*, *xs'*)⟩ ∧ *τmove1 P h e*)
by(*simp add*: *τred1g-def*)

lemma *τreds1-iff* [*iff*]:
τreds1g uf P t h (*es*, *xs*) (*es'*, *xs'*) = (*uf*,*P*,*t* ⊢1 ⟨*es*, (*h*, *xs*)⟩ [−*ε*→] ⟨*es'*, (*h*, *xs'*)⟩ ∧ *τmoves1 P h es*)
by(*simp add*: *τreds1g-def*)

lemma *τred1t-1step*:
⟦ *uf*,*P*,*t* ⊢1 ⟨*e*, (*h*, *xs*)⟩ −*ε*→ ⟨*e'*, (*h*, *xs'*)⟩; *τmove1 P h e* ⟧
⟹ *τred1gt uf P t h* (*e*, *xs*) (*e'*, *xs'*)
by(*blast intro*: *tranclp.r-into-trancl*)

lemma *τred1t-2step*:
⟦ *uf*,*P*,*t* ⊢1 ⟨*e*, (*h*, *xs*)⟩ −*ε*→ ⟨*e'*, (*h*, *xs'*)⟩; *τmove1 P h e*;
uf,*P*,*t* ⊢1 ⟨*e'*, (*h*, *xs'*)⟩ −*ε*→ ⟨*e''*, (*h*, *xs''*)⟩; *τmove1 P h e'* ⟧
⟹ *τred1gt uf P t h* (*e*, *xs*) (*e''*, *xs''*)
by(*blast intro*: *tranclp.trancl-into-trancl*[*OF τred1t-1step*])

lemma *τred1t-3step*:
⟦ *uf*,*P*,*t* ⊢1 ⟨*e*, (*h*, *xs*)⟩ −*ε*→ ⟨*e'*, (*h*, *xs'*)⟩; *τmove1 P h e*;
uf,*P*,*t* ⊢1 ⟨*e'*, (*h*, *xs'*)⟩ −*ε*→ ⟨*e''*, (*h*, *xs''*)⟩; *τmove1 P h e'*;
uf,*P*,*t* ⊢1 ⟨*e''*, (*h*, *xs''*)⟩ −*ε*→ ⟨*e'''*, (*h*, *xs'''*)⟩; *τmove1 P h e''* ⟧
⟹ *τred1gt uf P t h* (*e*, *xs*) (*e'''*, *xs'''*)
by(*blast intro*: *tranclp.trancl-into-trancl*[*OF τred1t-2step*])

lemma *τreds1t-1step*:
⟦ *uf*,*P*,*t* ⊢1 ⟨*es*, (*h*, *xs*)⟩ [−*ε*→] ⟨*es'*, (*h*, *xs'*)⟩; *τmoves1 P h es* ⟧
⟹ *τreds1gt uf P t h* (*es*, *xs*) (*es'*, *xs'*)
by(*blast intro*: *tranclp.r-into-trancl*)

lemma *τreds1t-2step*:
⟦ *uf*,*P*,*t* ⊢1 ⟨*es*, (*h*, *xs*)⟩ [−*ε*→] ⟨*es'*, (*h*, *xs'*)⟩; *τmoves1 P h es*;
uf,*P*,*t* ⊢1 ⟨*es'*, (*h*, *xs'*)⟩ [−*ε*→] ⟨*es''*, (*h*, *xs''*)⟩; *τmoves1 P h es'* ⟧
⟹ *τreds1gt uf P t h* (*es*, *xs*) (*es''*, *xs''*)
by(*blast intro*: *tranclp.trancl-into-trancl*[*OF τreds1t-1step*])

lemma *τreds1t-3step*:
⟦ *uf*,*P*,*t* ⊢1 ⟨*es*, (*h*, *xs*)⟩ [−*ε*→] ⟨*es'*, (*h*, *xs'*)⟩; *τmoves1 P h es*;
uf,*P*,*t* ⊢1 ⟨*es'*, (*h*, *xs'*)⟩ [−*ε*→] ⟨*es''*, (*h*, *xs''*)⟩; *τmoves1 P h es'*;
uf,*P*,*t* ⊢1 ⟨*es''*, (*h*, *xs''*)⟩ [−*ε*→] ⟨*es'''*, (*h*, *xs'''*)⟩; *τmoves1 P h es''* ⟧

$\Longrightarrow \tau reds1gt\ uf\ P\ t\ h\ (es, xs)\ (es''', xs''')$
by(*blast intro*: *tranclp.trancl-into-trancl*[*OF* $\tau reds1t$-*2step*])

lemma $\tau red1r$-*1step*:
$[\![\ uf,P,t \vdash 1\ \langle e, (h, xs)\rangle\ -\varepsilon\rightarrow\ \langle e', (h, xs')\rangle;\ \tau move1\ P\ h\ e\ ]\!]$
$\Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (e, xs)\ (e', xs')$
by(*blast intro*: *r-into-rtranclp*)

lemma $\tau red1r$-*2step*:
$[\![\ uf,P,t \vdash 1\ \langle e, (h, xs)\rangle\ -\varepsilon\rightarrow\ \langle e', (h, xs')\rangle;\ \tau move1\ P\ h\ e;$
$uf,P,t \vdash 1\ \langle e', (h, xs')\rangle\ -\varepsilon\rightarrow\ \langle e'', (h, xs'')\rangle;\ \tau move1\ P\ h\ e'\ ]\!]$
$\Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (e, xs)\ (e'', xs'')$
by(*blast intro*: *rtranclp.rtrancl-into-rtrancl*[*OF* $\tau red1r$-*1step*])

lemma $\tau red1r$-*3step*:
$[\![\ uf,P,t \vdash 1\ \langle e, (h, xs)\rangle\ -\varepsilon\rightarrow\ \langle e', (h, xs')\rangle;\ \tau move1\ P\ h\ e;$
$uf,P,t \vdash 1\ \langle e', (h, xs')\rangle\ -\varepsilon\rightarrow\ \langle e'', (h, xs'')\rangle;\ \tau move1\ P\ h\ e';$
$uf,P,t \vdash 1\ \langle e'', (h, xs'')\rangle\ -\varepsilon\rightarrow\ \langle e''', (h, xs''')\rangle;\ \tau move1\ P\ h\ e''\ ]\!]$
$\Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (e, xs)\ (e''', xs''')$
by(*blast intro*: *rtranclp.rtrancl-into-rtrancl*[*OF* $\tau red1r$-*2step*])

lemma $\tau reds1r$-*1step*:
$[\![\ uf,P,t \vdash 1\ \langle es, (h, xs)\rangle\ [-\varepsilon\rightarrow]\ \langle es', (h, xs')\rangle;\ \tau moves1\ P\ h\ es\ ]\!]$
$\Longrightarrow \tau reds1gr\ uf\ P\ t\ h\ (es, xs)\ (es', xs')$
by(*blast intro*: *r-into-rtranclp*)

lemma $\tau reds1r$-*2step*:
$[\![\ uf,P,t \vdash 1\ \langle es, (h, xs)\rangle\ [-\varepsilon\rightarrow]\ \langle es', (h, xs')\rangle;\ \tau moves1\ P\ h\ es;$
$uf,P,t \vdash 1\ \langle es', (h, xs')\rangle\ [-\varepsilon\rightarrow]\ \langle es'', (h, xs'')\rangle;\ \tau moves1\ P\ h\ es'\ ]\!]$
$\Longrightarrow \tau reds1gr\ uf\ P\ t\ h\ (es, xs)\ (es'', xs'')$
by(*blast intro*: *rtranclp.rtrancl-into-rtrancl*[*OF* $\tau reds1r$-*1step*])

lemma $\tau reds1r$-*3step*:
$[\![\ uf,P,t \vdash 1\ \langle es, (h, xs)\rangle\ [-\varepsilon\rightarrow]\ \langle es', (h, xs')\rangle;\ \tau moves1\ P\ h\ es;$
$uf,P,t \vdash 1\ \langle es', (h, xs')\rangle\ [-\varepsilon\rightarrow]\ \langle es'', (h, xs'')\rangle;\ \tau moves1\ P\ h\ es';$
$uf,P,t \vdash 1\ \langle es'', (h, xs'')\rangle\ [-\varepsilon\rightarrow]\ \langle es''', (h, xs''')\rangle;\ \tau moves1\ P\ h\ es''\ ]\!]$
$\Longrightarrow \tau reds1gr\ uf\ P\ t\ h\ (es, xs)\ (es''', xs''')$
by(*blast intro*: *rtranclp.rtrancl-into-rtrancl*[*OF* $\tau reds1r$-*2step*])

lemma $\tau red1t$-*preserves-len*: $\tau red1gt\ uf\ P\ t\ h\ (e, xs)\ (e', xs') \Longrightarrow length\ xs' = length\ xs$
by(*induct rule*: *tranclp-induct2*)(*auto dest*: *red1-preserves-len*)

lemma $\tau red1r$-*preserves-len*: $\tau red1gr\ uf\ P\ t\ h\ (e, xs)\ (e', xs') \Longrightarrow length\ xs' = length\ xs$
by(*induct rule*: *rtranclp-induct2*)(*auto dest*: *red1-preserves-len*)

lemma $\tau red1t$-*inj*-$\tau reds1t$: $\tau red1gt\ uf\ P\ t\ h\ (e, xs)\ (e', xs') \Longrightarrow \tau reds1gt\ uf\ P\ t\ h\ (e\ \#\ es, xs)\ (e'\ \#\ es, xs')$
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl List1Red1* $\tau moves1Hd$)

lemma $\tau reds1t$-*cons*-$\tau reds1t$: $\tau reds1gt\ uf\ P\ t\ h\ (es, xs)\ (es', xs') \Longrightarrow \tau reds1gt\ uf\ P\ t\ h\ (Val\ v\ \#\ es, xs)\ (Val\ v\ \#\ es', xs')$
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl List1Red2* $\tau moves1Tl$)

lemma $\tau red1r$-*inj*-$\tau reds1r$: $\tau red1gr\ uf\ P\ t\ h\ (e, xs)\ (e', xs') \Longrightarrow \tau reds1gr\ uf\ P\ t\ h\ (e\ \#\ es, xs)\ (e'\ \#$

es, *xs'*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl List1Red1 τmoves1Hd*)

lemma *τreds1r-cons-τreds1r*: *τreds1gr uf P t h* (*es*, *xs*) (*es'*, *xs'*) ⟹ *τreds1gr uf P t h* (*Val v # es*, *xs*) (*Val v # es'*, *xs'*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl List1Red2 τmoves1Tl*)

lemma *NewArray-τred1t-xt*:
τred1gt uf P t h (*e*, *xs*) (*e'*, *xs'*) ⟹ *τred1gt uf P t h* (*newA T*⌊*e*⌉, *xs*) (*newA T*⌊*e'*⌉, *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl New1ArrayRed τmove1NewArray*)

lemma *Cast-τred1t-xt*:
τred1gt uf P t h (*e*, *xs*) (*e'*, *xs'*) ⟹ *τred1gt uf P t h* (*Cast T e*, *xs*) (*Cast T e'*, *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl Cast1Red τmove1Cast*)

lemma *InstanceOf-τred1t-xt*:
τred1gt uf P t h (*e*, *xs*) (*e'*, *xs'*) ⟹ *τred1gt uf P t h* (*e instanceof T*, *xs*) (*e' instanceof T*, *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl InstanceOf1Red τmove1InstanceOf*)

lemma *BinOp-τred1t-xt1*:
τred1gt uf P t h (*e1*, *xs*) (*e1'*, *xs'*) ⟹ *τred1gt uf P t h* (*e1 «bop» e2*, *xs*) (*e1' «bop» e2*, *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl Bin1OpRed1 τmove1BinOp1*)

lemma *BinOp-τred1t-xt2*:
τred1gt uf P t h (*e2*, *xs*) (*e2'*, *xs'*) ⟹ *τred1gt uf P t h* (*Val v «bop» e2*, *xs*) (*Val v «bop» e2'*, *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl Bin1OpRed2 τmove1BinOp2*)

lemma *LAss-τred1t*:
τred1gt uf P t h (*e*, *xs*) (*e'*, *xs'*) ⟹ *τred1gt uf P t h* (*V := e*, *xs*) (*V := e'*, *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl LAss1Red τmove1LAss*)

lemma *AAcc-τred1t-xt1*:
τred1gt uf P t h (*a*, *xs*) (*a'*, *xs'*) ⟹ *τred1gt uf P t h* (*a*⌊*i*⌉, *xs*) (*a'*⌊*i*⌉, *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl AAcc1Red1 τmove1AAcc1*)

lemma *AAcc-τred1t-xt2*:
τred1gt uf P t h (*i*, *xs*) (*i'*, *xs'*) ⟹ *τred1gt uf P t h* (*Val a*⌊*i*⌉, *xs*) (*Val a*⌊*i'*⌉, *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl AAcc1Red2 τmove1AAcc2*)

lemma *AAss-τred1t-xt1*:
τred1gt uf P t h (*a*, *xs*) (*a'*, *xs'*) ⟹ *τred1gt uf P t h* (*a*⌊*i*⌉ := *e*, *xs*) (*a'*⌊*i*⌉ := *e*, *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl AAss1Red1 τmove1AAss1*)

lemma *AAss-τred1t-xt2*:
τred1gt uf P t h (*i*, *xs*) (*i'*, *xs'*) ⟹ *τred1gt uf P t h* (*Val a*⌊*i*⌉ := *e*, *xs*) (*Val a*⌊*i'*⌉ := *e*, *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl AAss1Red2 τmove1AAss2*)

lemma *AAss-τred1t-xt3*:
τred1gt uf P t h (*e*, *xs*) (*e'*, *xs'*) ⟹ *τred1gt uf P t h* (*Val a*⌊*Val i*⌉ := *e*, *xs*) (*Val a*⌊*Val i*⌉ := *e'*, *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl AAss1Red3 τmove1AAss3*)

lemma *ALength-τred1t-xt*:
τred1gt uf P t h (*a*, *xs*) (*a'*, *xs'*) ⟹ *τred1gt uf P t h* (*a*•*length*, *xs*) (*a'*•*length*, *xs'*)

by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl ALength1Red* τ*move1ALength*)

lemma *FAcc-*τ*red1t-xt*:
τ*red1gt uf P t h* (*e*, *xs*) (*e'*, *xs'*) $\Longrightarrow$ τ*red1gt uf P t h* (*e*·*F*{*D*}, *xs*) (*e'*·*F*{*D*}, *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl FAcc1Red* τ*move1FAcc*)

lemma *FAss-*τ*red1t-xt1*:
τ*red1gt uf P t h* (*e*, *xs*) (*e'*, *xs'*) $\Longrightarrow$ τ*red1gt uf P t h* (*e*·*F*{*D*} := *e2*, *xs*) (*e'*·*F*{*D*} := *e2*, *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl FAss1Red1* τ*move1FAss1*)

lemma *FAss-*τ*red1t-xt2*:
τ*red1gt uf P t h* (*e*, *xs*) (*e'*, *xs'*) $\Longrightarrow$ τ*red1gt uf P t h* (*Val v*·*F*{*D*} := *e*, *xs*) (*Val v*·*F*{*D*} := *e'*, *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl FAss1Red2* τ*move1FAss2*)

lemma *CAS-*τ*red1t-xt1*:
τ*red1gt uf P t h* (*e*, *xs*) (*e'*, *xs'*) $\Longrightarrow$ τ*red1gt uf P t h* (*e*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *xs*) (*e'*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl CAS1Red1*)

lemma *CAS-*τ*red1t-xt2*:
τ*red1gt uf P t h* (*e*, *xs*) (*e'*, *xs'*) $\Longrightarrow$ τ*red1gt uf P t h* (*Val v*·*compareAndSwap*(*D*·*F*, *e*, *e3*), *xs*) (*Val v*·*compareAndSwap*(*D*·*F*, *e'*, *e3*), *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl CAS1Red2*)

lemma *CAS-*τ*red1t-xt3*:
τ*red1gt uf P t h* (*e*, *xs*) (*e'*, *xs'*) $\Longrightarrow$ τ*red1gt uf P t h* (*Val v*·*compareAndSwap*(*D*·*F*, *Val v'*, *e*), *xs*) (*Val v*·*compareAndSwap*(*D*·*F*, *Val v'*, *e'*), *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl CAS1Red3*)

lemma *Call-*τ*red1t-obj*:
τ*red1gt uf P t h* (*e*, *xs*) (*e'*, *xs'*) $\Longrightarrow$ τ*red1gt uf P t h* (*e*·*M*(*ps*), *xs*) (*e'*·*M*(*ps*), *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl Call1Obj* τ*move1CallObj*)

lemma *Call-*τ*red1t-param*:
τ*reds1gt uf P t h* (*es*, *xs*) (*es'*, *xs'*) $\Longrightarrow$ τ*red1gt uf P t h* (*Val v*·*M*(*es*), *xs*) (*Val v*·*M*(*es'*), *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl Call1Params* τ*move1CallParams*)

lemma *Block-None-*τ*red1t-xt*:
τ*red1gt uf P t h* (*e*, *xs*) (*e'*, *xs'*) $\Longrightarrow$ τ*red1gt uf P t h* ({*V*:*T*=*None*; *e*}, *xs*) ({*V*:*T*=*None*; *e'*}, *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl* τ*move1Block elim*!: *Block1Red*)

lemma *Block-*τ*red1t-Some*:
⟦ τ*red1gt uf P t h* (*e*, *xs*[*V* := *v*]) (*e'*, *xs'*); *V* < *length xs* ⟧
$\Longrightarrow$ τ*red1gt uf P t h* ({*V*:*Ty*=⌊*v*⌋; *e*}, *xs*) ({*V*:*Ty*=*None*; *e'*}, *xs'*)
by(*blast intro*: *tranclp-into-tranclp2 Block1Some* τ*move1BlockSome Block-None-*τ*red1t-xt*)

lemma *Sync-*τ*red1t-xt*:
τ*red1gt uf P t h* (*e*, *xs*) (*e'*, *xs'*) $\Longrightarrow$ τ*red1gt uf P t h* (*sync*$_V$ (*e*) *e2*, *xs*) (*sync*$_V$ (*e'*) *e2*, *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl Synchronized1Red1* τ*move1Sync*)

lemma *InSync-*τ*red1t-xt*:
τ*red1gt uf P t h* (*e*, *xs*) (*e'*, *xs'*) $\Longrightarrow$ τ*red1gt uf P t h* (*insync*$_V$ (*a*) *e*, *xs*) (*insync*$_V$ (*a*) *e'*, *xs'*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl Synchronized1Red2* τ*move1InSync*)

lemma *Seq-τred1t-xt*:
τred1gt uf P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred1gt uf P t h* (*e*;;*e2*, *xs*) (*e′*;;*e2*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl Seq1Red τmove1Seq*)

lemma *Cond-τred1t-xt*:
τred1gt uf P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred1gt uf P t h* (*if* (*e*) *e1 else e2*, *xs*) (*if* (*e′*) *e1 else e2*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl Cond1Red τmove1Cond*)

lemma *Throw-τred1t-xt*:
τred1gt uf P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred1gt uf P t h* (*throw e*, *xs*) (*throw e′*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl Throw1Red τmove1Throw*)

lemma *Try-τred1t-xt*:
τred1gt uf P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred1gt uf P t h* (*try e catch*(*C V*) *e2*, *xs*) (*try e′ catch*(*C V*) *e2*, *xs′*)
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *tranclp.trancl-into-trancl Try1Red τmove1Try*)

lemma *NewArray-τred1r-xt*:
τred1gr uf P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred1gr uf P t h* (*newA T*⌊*e*⌉, *xs*) (*newA T*⌊*e′*⌉, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl New1ArrayRed τmove1NewArray*)

lemma *Cast-τred1r-xt*:
τred1gr uf P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred1gr uf P t h* (*Cast T e*, *xs*) (*Cast T e′*, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl Cast1Red τmove1Cast*)

lemma *InstanceOf-τred1r-xt*:
τred1gr uf P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred1gr uf P t h* (*e instanceof T*, *xs*) (*e′ instanceof T*, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl InstanceOf1Red τmove1InstanceOf*)

lemma *BinOp-τred1r-xt1*:
τred1gr uf P t h (*e1*, *xs*) (*e1′*, *xs′*) ⟹ *τred1gr uf P t h* (*e1* «*bop*» *e2*, *xs*) (*e1′* «*bop*» *e2*, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl Bin1OpRed1 τmove1BinOp1*)

lemma *BinOp-τred1r-xt2*:
τred1gr uf P t h (*e2*, *xs*) (*e2′*, *xs′*) ⟹ *τred1gr uf P t h* (*Val v* «*bop*» *e2*, *xs*) (*Val v* «*bop*» *e2′*, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl Bin1OpRed2 τmove1BinOp2*)

lemma *LAss-τred1r*:
τred1gr uf P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred1gr uf P t h* (*V* := *e*, *xs*) (*V* := *e′*, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl LAss1Red τmove1LAss*)

lemma *AAcc-τred1r-xt1*:
τred1gr uf P t h (*a*, *xs*) (*a′*, *xs′*) ⟹ *τred1gr uf P t h* (*a*⌊*i*⌉, *xs*) (*a′*⌊*i*⌉, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl AAcc1Red1 τmove1AAcc1*)

lemma *AAcc-τred1r-xt2*:
τred1gr uf P t h (*i*, *xs*) (*i′*, *xs′*) ⟹ *τred1gr uf P t h* (*Val a*⌊*i*⌉, *xs*) (*Val a*⌊*i′*⌉, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl AAcc1Red2 τmove1AAcc2*)

lemma *AAss-τred1r-xt1*:

$\tau red1gr\ uf\ P\ t\ h\ (a,\ xs)\ (a',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (a\lfloor i\rceil := e,\ xs)\ (a'\lfloor i\rceil := e,\ xs')$
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl AAss1Red1 τmove1AAss1*)

lemma *AAss-τred1r-xt2*:
$\tau red1gr\ uf\ P\ t\ h\ (i,\ xs)\ (i',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (Val\ a\lfloor i\rceil := e,\ xs)\ (Val\ a\lfloor i'\rceil := e,\ xs')$
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl AAss1Red2 τmove1AAss2*)

lemma *AAss-τred1r-xt3*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (Val\ a\lfloor Val\ i\rceil := e,\ xs)\ (Val\ a\lfloor Val\ i\rceil := e',\ xs')$
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl AAss1Red3 τmove1AAss3*)

lemma *ALength-τred1r-xt*:
$\tau red1gr\ uf\ P\ t\ h\ (a,\ xs)\ (a',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (a{\cdot}length,\ xs)\ (a'{\cdot}length,\ xs')$
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl ALength1Red τmove1ALength*)

lemma *FAcc-τred1r-xt*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (e{\cdot}F\{D\},\ xs)\ (e'{\cdot}F\{D\},\ xs')$
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl FAcc1Red τmove1FAcc*)

lemma *FAss-τred1r-xt1*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (e{\cdot}F\{D\} := e2,\ xs)\ (e'{\cdot}F\{D\} := e2,\ xs')$
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl FAss1Red1 τmove1FAss1*)

lemma *FAss-τred1r-xt2*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (Val\ v{\cdot}F\{D\} := e,\ xs)\ (Val\ v{\cdot}F\{D\} := e',\ xs')$
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl FAss1Red2 τmove1FAss2*)

lemma *CAS-τred1r-xt1*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3),\ xs)$
$(e'{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3),\ xs')$
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl CAS1Red1*)

lemma *CAS-τred1r-xt2*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ e,\ e3),\ xs)\ (Val$
$v{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e3),\ xs')$
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl CAS1Red2*)

lemma *CAS-τred1r-xt3*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v',\ e),\ xs)$
$(Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v',\ e'),\ xs')$
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl CAS1Red3*)

lemma *Call-τred1r-obj*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (e{\cdot}M(ps),\ xs)\ (e'{\cdot}M(ps),\ xs')$
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl Call1Obj τmove1CallObj*)

lemma *Call-τred1r-param*:
$\tau reds1gr\ uf\ P\ t\ h\ (es,\ xs)\ (es',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (Val\ v{\cdot}M(es),\ xs)\ (Val\ v{\cdot}M(es'),\ xs')$
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl Call1Params τmove1Call-Params*)

lemma *Block-None-τred1r-xt*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (\{V{:}T{=}None;\ e\},\ xs)\ (\{V{:}T{=}None;\ e'\},\ xs')$
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl τmove1Block elim*!: *Block1Red*)

lemma *Block-τred1r-Some*:
⟦ *τred1gr uf P t h* (*e*, *xs*[*V* := *v*]) (*e′*, *xs′*); *V* < *length xs* ⟧
⟹ *τred1gr uf P t h* ({*V*:*Ty*=⌊*v*⌋; *e*}, *xs*) ({*V*:*Ty*=*None*; *e′*}, *xs′*)
by(*blast intro*: *converse-rtranclp-into-rtranclp Block1Some τmove1BlockSome Block-None-τred1r-xt*)

lemma *Sync-τred1r-xt*:
τred1gr uf P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred1gr uf P t h* (*sync*$_V$ (*e*) *e2*, *xs*) (*sync*$_V$ (*e′*) *e2*, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl Synchronized1Red1 τmove1Sync*)

lemma *InSync-τred1r-xt*:
τred1gr uf P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred1gr uf P t h* (*insync*$_V$ (*a*) *e*, *xs*) (*insync*$_V$ (*a*) *e′*, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl Synchronized1Red2 τmove1In-Sync*)

lemma *Seq-τred1r-xt*:
τred1gr uf P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred1gr uf P t h* (*e*;;*e2*, *xs*) (*e′*;;*e2*, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl Seq1Red τmove1Seq*)

lemma *Cond-τred1r-xt*:
τred1gr uf P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred1gr uf P t h* (*if* (*e*) *e1 else e2*, *xs*) (*if* (*e′*) *e1 else e2*, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl Cond1Red τmove1Cond*)

lemma *Throw-τred1r-xt*:
τred1gr uf P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred1gr uf P t h* (*throw e*, *xs*) (*throw e′*, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl Throw1Red τmove1Throw*)

lemma *Try-τred1r-xt*:
τred1gr uf P t h (*e*, *xs*) (*e′*, *xs′*) ⟹ *τred1gr uf P t h* (*try e catch*(*C V*) *e2*, *xs*) (*try e′ catch*(*C V*) *e2*, *xs′*)
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *rtranclp.rtrancl-into-rtrancl Try1Red τmove1Try*)

lemma *τred1t-ThrowD* [*dest*]: *τred1gt uf P t h* (*Throw a*, *xs*) (*e″*, *xs″*) ⟹ *e″* = *Throw a* ∧ *xs″* = *xs*
by(*induct rule*: *tranclp-induct2*)(*auto*)

lemma *τred1r-ThrowD* [*dest*]: *τred1gr uf P t h* (*Throw a*, *xs*) (*e″*, *xs″*) ⟹ *e″* = *Throw a* ∧ *xs″* = *xs*
by(*induct rule*: *rtranclp-induct2*)(*auto*)

lemma *τRed1-conv* [*iff*]:
τRed1g uf P t h (*ex*, *exs*) (*ex′*, *exs′*) = (*uf*,*P*,*t* ⊢1 ⟨*ex*/*exs*, *h*⟩ −*ε*→ ⟨*ex′*/*exs′*, *h*⟩ ∧ *τMove1 P h* (*ex*, *exs*))
by(*simp add*: *τRed1g-def*)

lemma *τred1t-into-τRed1t*:
τred1gt uf P t h (*e*, *xs*) (*e″*, *xs″*) ⟹ *τRed1gt uf P t h* ((*e*, *xs*), *exs*) ((*e″*, *xs″*), *exs*)
by(*induct rule*: *tranclp-induct2*)(*fastforce dest*: *red1Red intro*: *τmove1Block tranclp.intros*)+

lemma *τred1r-into-τRed1r*:
τred1gr uf P t h (*e*, *xs*) (*e″*, *xs″*) ⟹ *τRed1gr uf P t h* ((*e*, *xs*), *exs*) ((*e″*, *xs″*), *exs*)
by(*induct rule*: *rtranclp-induct2*)(*fastforce dest*: *red1Red intro*: *τmove1Block rtranclp.intros*)+

lemma *red1-max-vars*: *uf*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩ ⟹ *max-vars e′* ≤ *max-vars e*
and *reds1-max-varss*: *uf*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es′*, *s′*⟩ ⟹ *max-varss es′* ≤ *max-varss es*

by(*induct rule*: *red1-reds1.inducts*) *auto*

lemma *τred1t-max-vars*: *τred1gt uf P t h* (*e*, *xs*) (*e′*, *xs′*) ⟹ *max-vars e′* ≤ *max-vars e*
by(*induct rule*: *tranclp-induct2*)(*auto dest*: *red1-max-vars*)

lemma *τred1r-max-vars*: *τred1gr uf P t h* (*e*, *xs*) (*e′*, *xs′*) ⟹ *max-vars e′* ≤ *max-vars e*
by(*induct rule*: *rtranclp-induct2*)(*auto dest*: *red1-max-vars*)

lemma *τred1r-Val*:
τred1gr uf P t h (*Val v*, *xs*) *s′* ⟷ *s′* = (*Val v*, *xs*)
proof
assume *τred1gr uf P t h* (*Val v*, *xs*) *s′*
thus *s′* = (*Val v*, *xs*) **by** *induct*(*auto*)
qed *auto*

lemma *τred1t-Val*:
τred1gt uf P t h (*Val v*, *xs*) *s′* ⟷ *False*
proof
assume *τred1gt uf P t h* (*Val v*, *xs*) *s′*
thus *False* **by** *induct auto*
qed *auto*

lemma *τreds1r-map-Val*:
τreds1gr uf P t h (*map Val vs*, *xs*) *s′* ⟷ *s′* = (*map Val vs*, *xs*)
proof
assume *τreds1gr uf P t h* (*map Val vs*, *xs*) *s′*
thus *s′* = (*map Val vs*, *xs*) **by** *induct auto*
qed *auto*

lemma *τreds1t-map-Val*:
τreds1gt uf P t h (*map Val vs*, *xs*) *s′* ⟷ *False*
proof
assume *τreds1gt uf P t h* (*map Val vs*, *xs*) *s′*
thus *False* **by** *induct auto*
qed *auto*

lemma *τreds1r-map-Val-Throw*:
τreds1gr uf P t h (*map Val vs* @ *Throw a* # *es*, *xs*) *s′* ⟷ *s′* = (*map Val vs* @ *Throw a* # *es*, *xs*)
(**is** *?lhs* ⟷ *?rhs*)
proof
assume *?lhs* **thus** *?rhs* **by** *induct auto*
qed *auto*

lemma *τreds1t-map-Val-Throw*:
τreds1gt uf P t h (*map Val vs* @ *Throw a* # *es*, *xs*) *s′* ⟷ *False*
(**is** *?lhs* ⟷ *?rhs*)
proof
assume *?lhs* **thus** *?rhs* **by** *induct auto*
qed *auto*

lemma *τred1r-Throw*:
τred1gr uf P t h (*Throw a*, *xs*) *s′* ⟷ *s′* = (*Throw a*, *xs*) (**is** *?lhs* ⟷ *?rhs*)
proof
assume *?lhs* **thus** *?rhs* **by** *induct auto*

qed *simp*

lemma *τred1t-Throw*:
τred1gt uf P t h (*Throw a*, *xs*) *s′* ⟷ *False* (**is** *?lhs* ⟷ *?rhs*)
proof
assume *?lhs* **thus** *?rhs* **by** *induct auto*
qed *simp*

lemma *red1-False-into-red1-True*:
False,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩ ⟹ *True*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
and *reds1-False-into-reds1-True*:
False,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es′*, *s′*⟩ ⟹ *True*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es′*, *s′*⟩
by (*induct rule*: *red1-reds1.inducts*) (*auto intro*: *red1-reds1.intros*)

lemma *Red1-False-into-Red1-True*:
assumes *False*,*P*,*t* ⊢1 ⟨*ex*/*exs*,*shr s*⟩ −*ta*→ ⟨*ex′*/*exs′*,*m′*⟩
shows *True*,*P*,*t* ⊢1 ⟨*ex*/*exs*,*shr s*⟩ −*ta*→ ⟨*ex′*/*exs′*,*m′*⟩
using *assms*
by(*cases*)(*auto dest*: *Red1.intros red1-False-into-red1-True*)

lemma *red1-Suspend-is-call*:
⟦ *uf*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩; *Suspend w* ∈ *set* {|*ta*|}*w* ⟧ ⟹ *call1 e′* ≠ *None*
and *reds-Suspend-is-calls*:
⟦ *uf*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es′*, *s′*⟩; *Suspend w* ∈ *set* {|*ta*|}*w* ⟧ ⟹ *calls1 es′* ≠ *None*
by(*induct rule*: *red1-reds1.inducts*)(*auto dest*: *red-external-Suspend-StaySame*)

lemma *Red1-Suspend-is-call*:
⟦ *uf*,*P*,*t* ⊢1 ⟨(*e*, *xs*)/*exs*, *h*⟩ −*ta*→ ⟨(*e′*, *xs′*)/*exs′*, *h′*⟩; *Suspend w* ∈ *set* {|*ta*|}*w* ⟧ ⟹ *call1 e′* ≠ *None*
by(*auto elim*!: *Red1.cases dest*: *red1-Suspend-is-call*)

lemma *Red1-mthr*: *multithreaded final-expr1* (*mred1g uf P*)
by(*unfold-locales*)(*fastforce elim*!: *Red1.cases dest*: *red1-new-thread-heap*)+

lemma *red1-τmove1-heap-unchanged*: ⟦ *uf*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩; *τmove1 P* (*hp s*) *e* ⟧ ⟹ *hp s′* = *hp s*
and *red1-τmoves1-heap-unchanged*: ⟦ *uf*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es′*, *s′*⟩; *τmoves1 P* (*hp s*) *es* ⟧ ⟹ *hp s′* = *hp s*
apply(*induct rule*: *red1-reds1.inducts*)
apply(*fastforce simp add*: *map-eq-append-conv τexternal′-def τexternal-def dest*: *τexternal′-red-external-heap-unch*
done

lemma *Red1-τmthr-wf*: *τmultithreaded-wf final-expr1* (*mred1g uf P*) (*τMOVE1 P*)
proof −
interpret *multithreaded final-expr1 mred1g uf P convert-RA*
by(*rule Red1-mthr*)
show *?thesis*
proof
fix *x1 m1 t ta1 x1′ m1′*
assume *mred1g uf P t* (*x1*, *m1*) *ta1* (*x1′*, *m1′*) *τMOVE1 P* (*x1*, *m1*) *ta1* (*x1′*, *m1′*)
thus *m1* = *m1′* **by**(*cases x1*)(*fastforce elim*!: *Red1.cases dest*: *red1-τmove1-heap-unchanged*)
next
fix *s ta s′*
assume *τMOVE1 P s ta s′*
thus *ta* = *ε* **by**(*simp add*: *split-beta*)

qed
qed

end

sublocale *J1-heap-base* < *Red1-mthr*:
τmultithreaded-wf
final-expr1
mred1g uf P
convert-RA
τMOVE1 P
for *uf P*
by(*rule Red1-τmthr-wf*)

context *J1-heap-base* **begin**

lemma *τRed1't-into-Red1'-τmthr-silent-movet*:
τRed1gt uf P t h (*ex2*, *exs2*) (*ex2''*, *exs2''*)
⟹ *Red1-mthr.silent-movet uf P t* ((*ex2*, *exs2*), *h*) ((*ex2''*, *exs2''*), *h*)
apply(*induct rule*: *tranclp-induct2*)
apply *clarsimp*
apply(*rule tranclp.r-into-trancl*)
apply(*simp add*: *Red1-mthr.silent-move-iff*)
apply(*erule tranclp.trancl-into-trancl*)
apply(*simp add*: *Red1-mthr.silent-move-iff*)
done

lemma *τRed1t-into-Red1'-τmthr-silent-moves*:
τRed1gt uf P t h (*ex2*, *exs2*) (*ex2''*, *exs2''*)
⟹ *Red1-mthr.silent-moves uf P t* ((*ex2*, *exs2*), *h*) ((*ex2''*, *exs2''*), *h*)
by(*rule tranclp-into-rtranclp*)(*rule τRed1't-into-Red1'-τmthr-silent-movet*)

lemma *τRed1'r-into-Red1'-τmthr-silent-moves*:
τRed1gr uf P t h (*ex*, *exs*) (*ex'*, *exs'*) ⟹ *Red1-mthr.silent-moves uf P t* ((*ex*, *exs*), *h*) ((*ex'*, *exs'*), *h*)
apply(*induct rule*: *rtranclp-induct2*)
apply *blast*
apply(*erule rtranclp.rtrancl-into-rtrancl*)
apply(*simp add*: *Red1-mthr.silent-move-iff*)
done

lemma *τRed1r-rtranclpD*:
τRed1gr uf P t h s s' ⟹ *τtrsys.silent-moves* (*mred1g uf P t*) (*τMOVE1 P*) (*s*, *h*) (*s'*, *h*)
apply(*induct rule*: *rtranclp-induct*)
apply(*auto elim*!: *rtranclp.rtrancl-into-rtrancl intro*: *τtrsys.silent-move.intros*)
done

lemma *τRed1t-tranclpD*:
τRed1gt uf P t h s s' ⟹ *τtrsys.silent-movet* (*mred1g uf P t*) (*τMOVE1 P*) (*s*, *h*) (*s'*, *h*)
apply(*induct rule*: *tranclp-induct*)
apply(*rule tranclp.r-into-trancl*)
apply(*auto elim*!: *tranclp.trancl-into-trancl intro*!: *τtrsys.silent-move.intros simp*: *τRed1g-def split-def*)
done

lemma *τmreds1-Val-Nil*: *τtrsys.silent-moves* (*mred1g uf P t*) (*τMOVE1 P*) (((*Val v*, *xs*), []), *h*) *s* ⟷ *s* = (((*Val v*, *xs*), []), *h*)
proof
assume *τtrsys.silent-moves* (*mred1g uf P t*) (*τMOVE1 P*) (((*Val v*, *xs*), []), *h*) *s*
thus *s* = (((*Val v*, *xs*), []), *h*)
by *induct*(*auto elim*!: *Red1-mthr.silent-move.cases Red1.cases*)
qed *auto*

lemma *τmreds1-Throw-Nil*:
τtrsys.silent-moves (*mred1g uf P t*) (*τMOVE1 P*) (((*Throw a*, *xs*), []), *h*) *s* ⟷ *s* = (((*Throw a*, *xs*), []), *h*)
proof
assume *τtrsys.silent-moves* (*mred1g uf P t*) (*τMOVE1 P*) (((*Throw a*, *xs*), []), *h*) *s*
thus *s* = (((*Throw a*, *xs*), []), *h*)
by *induct*(*auto elim*!: *Red1-mthr.silent-move.cases Red1.cases*)
qed *auto*

end

end

7.7 Deadlock perservation for the intermediate language

theory *J1Deadlock* **imports**
J1
../Framework/FWDeadlock
../Common/ExternalCallWF
begin

context *J1-heap-base* **begin**

lemma *IUF-red-taD*:
True,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩
⟹ ∃ *e*′ *ta*′ *s*′. *False*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*′→ ⟨*e*′, *s*′⟩ ∧
collect-locks ⦃*ta*′⦄*l* ⊆ *collect-locks* ⦃*ta*⦄*l* ∧ *set* ⦃*ta*′⦄*c* ⊆ *set* ⦃*ta*⦄*c* ∧ *collect-interrupts* ⦃*ta*′⦄*i* ⊆ *collect-interrupts* ⦃*ta*⦄*i* ∧
(∃ *s*. *Red1-mthr.actions-ok s t ta*′)

and *IUFs-reds-taD*:
True,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩
⟹ ∃ *es*′ *ta*′ *s*′. *False*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*′→] ⟨*es*′, *s*′⟩ ∧
collect-locks ⦃*ta*′⦄*l* ⊆ *collect-locks* ⦃*ta*⦄*l* ∧ *set* ⦃*ta*′⦄*c* ⊆ *set* ⦃*ta*⦄*c* ∧ *collect-interrupts* ⦃*ta*′⦄*i* ⊆ *collect-interrupts* ⦃*ta*⦄*i* ∧
(∃ *s*. *Red1-mthr.actions-ok s t ta*′)
proof(*induct rule*: *red1-reds1.inducts*)
case *Red1InstanceOf* **thus** *?case*
using [[*hypsubst-thin* = *true*]]
by(*auto intro*!: *exI red1-reds1.Red1InstanceOf simp del*: *split-paired-Ex*)((*subst fst-conv snd-conv wset-def*)+, *simp*)
next
case *Red1CallExternal* **thus** *?case*
by(*fastforce simp del*: *split-paired-Ex dest*: *red-external-ta-satisfiable*[**where** *final*=*final-expr1* :: (*'addr expr1* × *'addr val list*) × (*'addr expr1* × *'addr val list*) *list* ⇒ *bool*] *intro*: *red1-reds1.Red1CallExternal*)

next
case *Lock1Synchronized* **thus** *?case*
by(*auto intro*!: *exI exI*[**where** *x*=(*K*$ *None*, (*Map.empty*, *undefined*), *Map.empty*, {})] *red1-reds1.Lock1Synchronized simp del*: *split-paired-Ex simp add*: *lock-ok-las-def finfun-upd-apply may-lock.intros*(*1*))
next
case (*Synchronized1Red2 e s ta e' s' V a*)
then obtain *e' ta' s'*
where *False*,*P*,*t* ⊢*1* ⟨*e*,*s*⟩ −*ta'*→ ⟨*e'*,*s'*⟩
and *L*: *collect-locks* ⦃*ta'*⦄*l* ⊆ *collect-locks* ⦃*ta*⦄*l* ∧ *set* ⦃*ta'*⦄*c* ⊆ *set* ⦃*ta*⦄*c* ∧ *collect-interrupts* ⦃*ta'*⦄*i* ⊆ *collect-interrupts* ⦃*ta*⦄*i*
and *aok*: ∃ *s*. *Red1-mthr.actions-ok s t ta'*
by *blast*
from ‹*False*,*P*,*t* ⊢*1* ⟨*e*,*s*⟩ −*ta'*→ ⟨*e'*,*s'*⟩› **have** *False*,*P*,*t* ⊢*1* ⟨*insync*$_V$ (*a*) *e*, *s*⟩ −*ta'*→ ⟨*insync*$_V$ (*a*) *e'*, *s'*⟩
by(*rule red1-reds1.Synchronized1Red2*)
thus *?case* **using** *L aok* **by** *blast*
next
case *Unlock1Synchronized* **thus** *?case*
by(*auto simp del*: *split-paired-Ex intro*!: *exI exI*[**where** *x*=(*K*$ ⌊(*t*, *0*)⌋, (*Map.empty*, *undefined*), *Map.empty*, {})] *red1-reds1.Unlock1Synchronized simp add*: *lock-ok-las-def finfun-upd-apply*)
next
case *Unlock1SynchronizedFail* **thus** *?case*
by(*auto simp del*: *split-paired-Ex intro*!: *exI exI*[**where** *x*=(*K*$ ⌊(*t*, *0*)⌋, (*Map.empty*, *undefined*), *Map.empty*, {})] *red1-reds1.Unlock1Synchronized simp add*: *lock-ok-las-def finfun-upd-apply collect-locks-def split*: *if-split-asm*)
next
case *Synchronized1Throw2* **thus** *?case*
by(*auto simp del*: *split-paired-Ex intro*!: *exI exI*[**where** *x*=(*K*$ ⌊(*t*, *0*)⌋, (*Map.empty*, *undefined*), *Map.empty*, {})] *red1-reds1.Synchronized1Throw2 simp add*: *lock-ok-las-def finfun-upd-apply*)
next
case *Synchronized1Throw2Fail* **thus** *?case*
by(*auto simp del*: *split-paired-Ex intro*!: *exI exI*[**where** *x*=(*K*$ ⌊(*t*, *0*)⌋, (*Map.empty*, *undefined*), *Map.empty*, {})] *red1-reds1.Synchronized1Throw2 simp add*: *lock-ok-las-def finfun-upd-apply collect-locks-def split*: *if-split-asm*)
qed(*fastforce intro*: *red1-reds1.intros*)+

lemma *IUF-Red1-taD*:
assumes *True*,*P*,*t* ⊢*1* ⟨*ex*/*exs*, *h*⟩ −*ta*→ ⟨*ex'*/*exs'*, *h'*⟩
shows ∃ *ex' exs' h' ta'*. *False*,*P*,*t* ⊢*1* ⟨*ex*/*exs*, *h*⟩ −*ta'*→ ⟨*ex'*/*exs'*, *h'*⟩ ∧
collect-locks ⦃*ta'*⦄*l* ⊆ *collect-locks* ⦃*ta*⦄*l* ∧ *set* ⦃*ta'*⦄*c* ⊆ *set* ⦃*ta*⦄*c* ∧ *collect-interrupts* ⦃*ta'*⦄*i* ⊆ *collect-interrupts* ⦃*ta*⦄*i* ∧
(∃ *s*. *Red1-mthr.actions-ok s t ta'*)
using *assms*
apply(*cases*)
apply(*safe dest*!: *IUF-red-taD*)
apply(*simp del*: *split-paired-Ex*)
apply(*rule exI conjI*)+
apply(*erule red1Red*)
apply *simp*
apply *blast*
apply(*rule exI conjI red1Call*)+
apply(*auto simp add*: *lock-ok-las-def*)
apply(*rule exI conjI red1Return*)+
apply *auto*

done

lemma *mred1'-mred1-must-sync-eq*:
Red1-mthr.must-sync False P t x (shr s) = Red1-mthr.must-sync True P t x (shr s)
proof
assume *Red1-mthr.must-sync False P t x (shr s)*
thus *Red1-mthr.must-sync True P t x (shr s)*
by(*rule Red1-mthr.must-syncE*)(*rule Red1-mthr.must-syncI*, *auto simp add*: *split-def simp del*: *split-paired-Ex intro*: *Red1-False-into-Red1-True*)
next
assume *Red1-mthr.must-sync True P t x (shr s)*
thus *Red1-mthr.must-sync False P t x (shr s)*
apply(*rule Red1-mthr.must-syncE*)
apply(*rule Red1-mthr.must-syncI*)
apply(*cases x*)
apply(*auto simp add*: *split-beta split-paired-Ex*)
apply(*drule IUF-Red1-taD*)
apply *simp*
apply *blast*
done
qed

lemma *Red1-Red1'-deadlock-inv*:
Red1-mthr.deadlock True P s = Red1-mthr.deadlock False P s
proof(*rule iffI*)
assume *dead*: *Red1-mthr.deadlock True P s*
show *Red1-mthr.deadlock False P s*
proof(*rule multithreaded-base.deadlockI*)
fix *t x*
assume *tst*: *thr s t = ⌊(x, no-wait-locks)⌋*
and *nfin*: *¬ final-expr1 x*
and *wst*: *wset s t = None*
with *dead* **obtain** *ms*: *Red1-mthr.must-sync True P t x (shr s)*
and *cs* [*rule-format*]: *∀ LT. Red1-mthr.can-sync True P t x (shr s) LT ⟶*
(∃ lt∈LT. final-thread.must-wait final-expr1 s t lt (dom (thr s)))
by(*rule Red1-mthr.deadlockD1*)
from *ms*[*folded mred1'-mred1-must-sync-eq*]
show *Red1-mthr.must-sync False P t x (shr s) ∧*
(∀ LT. Red1-mthr.can-sync False P t x (shr s) LT ⟶
(∃ lt∈LT. final-thread.must-wait final-expr1 s t lt (dom (thr s))))
proof
show *∀ LT. Red1-mthr.can-sync False P t x (shr s) LT ⟶*
(∃ lt∈LT. final-thread.must-wait final-expr1 s t lt (dom (thr s)))
proof(*intro strip*)
fix *LT*
assume *Red1-mthr.can-sync False P t x (shr s) LT*
then obtain *ta x' m'* **where** *mred1' P t (x, shr s) ta (x', m')*
and [*simp*]: *LT = collect-locks {|ta|}_l <+> collect-cond-actions {|ta|}_c <+> collect-interrupts {|ta|}_i*
by(*rule Red1-mthr.can-syncE*)
hence *mred1 P t (x, shr s) ta (x', m')* **by**(*auto simp add*: *split-beta intro*: *Red1-False-into-Red1-True*)
hence *Red1-mthr.can-sync True P t x (shr s) LT* **by**(*rule Red1-mthr.can-syncI*) *simp*
thus *∃ lt∈LT. final-thread.must-wait final-expr1 s t lt (dom (thr s))* **by**(*rule cs*)
qed

```
    qed
  next
    fix t x ln l
    assume thr s t = ⌊(x, ln)⌋ 0 < ln $ l ¬ waiting (wset s t)
    thus ∃ l t'. 0 < ln $ l ∧ t ≠ t' ∧ thr s t' ≠ None ∧ has-lock ((locks s) $ l) t'
      by(rule Red1-mthr.deadlockD2[OF dead]) blast
  next
    fix t x w
    assume thr s t = ⌊(x, no-wait-locks)⌋
    thus wset s t ≠ ⌊PostWS w⌋
      by(rule Red1-mthr.deadlockD3[OF dead, rule-format])
  qed
next
  assume dead: Red1-mthr.deadlock False P s
  show Red1-mthr.deadlock True P s
  proof(rule Red1-mthr.deadlockI)
    fix t x
    assume tst: thr s t = ⌊(x, no-wait-locks)⌋
      and nfin: ¬ final-expr1 x
      and wst: wset s t = None
    with dead obtain ms: Red1-mthr.must-sync False P t x (shr s)
      and cs [rule-format]: ∀ LT. Red1-mthr.can-sync False P t x (shr s) LT ⟶
            (∃ lt∈LT. final-thread.must-wait final-expr1 s t lt (dom (thr s)))
      by(rule Red1-mthr.deadlockD1)
    from ms[unfolded mred1'-mred1-must-sync-eq]
    show Red1-mthr.must-sync True P t x (shr s) ∧
          (∀ LT. Red1-mthr.can-sync True P t x (shr s) LT ⟶
                (∃ lt∈LT. final-thread.must-wait final-expr1 s t lt (dom (thr s))))
    proof
      show ∀ LT. Red1-mthr.can-sync True P t x (shr s) LT ⟶
        (∃ lt∈LT. final-thread.must-wait final-expr1 s t lt (dom (thr s)))
      proof(intro strip)
        fix LT
        assume Red1-mthr.can-sync True P t x (shr s) LT
        then obtain ta x' m' where mred1 P t (x, shr s) ta (x', m')
          and [simp]: LT = collect-locks {|ta|}_l <+> collect-cond-actions {|ta|}_c <+> collect-interrupts
{|ta|}_i
          by(rule Red1-mthr.can-syncE)
        then obtain e xs exs e' xs' exs' where x [simp]: x = ((e, xs), exs) x' = ((e', xs'), exs')
          and red: True,P,t ⊢1 ⟨(e, xs)/exs, shr s⟩ −ta→ ⟨(e', xs')/exs', m'⟩ by(cases x, cases x')
fastforce
        from IUF-Red1-taD[OF red] obtain ex'' exs'' h'' ta'
          where red': False,P,t ⊢1 ⟨(e, xs)/exs,shr s⟩ −ta'→ ⟨ex''/exs'',h''⟩
            and collect-locks {|ta'|}_l <+> collect-cond-actions {|ta'|}_c <+> collect-interrupts {|ta'|}_i ⊆
collect-locks {|ta|}_l <+> collect-cond-actions {|ta|}_c <+> collect-interrupts {|ta|}_i
          by auto blast
        then obtain LT' where cs': Red1-mthr.can-sync False P t x (shr s) LT'
          and LT': LT' ⊆ LT by(cases ex'')(fastforce intro!: Red1-mthr.can-syncI)
        with cs[of LT'] show ∃ lt∈LT. final-thread.must-wait final-expr1 s t lt (dom (thr s)) by auto
      qed
    qed
  next
    fix t x ln l
    assume thr s t = ⌊(x, ln)⌋ 0 < ln $ l ¬ waiting (wset s t)
```

```
    thus ∃ l t'. 0 < ln $ l ∧ t ≠ t' ∧ thr s t' ≠ None ∧ has-lock ((locks s) $ l) t'
      by(rule Red1-mthr.deadlockD2[OF dead]) blast
  next
    fix t x w
    assume thr s t = ⌊(x, no-wait-locks)⌋
    thus wset s t ≠ ⌊PostWS w⌋
      by(rule Red1-mthr.deadlockD3[OF dead, rule-format])
  qed
qed

end

end
```

7.8 Program Compilation

```
theory PCompiler
imports
  ../Common/WellForm
  ../Common/BinOp
  ../Common/Conform
begin

definition compM :: (mname ⇒ ty list ⇒ ty ⇒ 'a ⇒ 'b) ⇒ 'a mdecl' ⇒ 'b mdecl'
where compM f ≡ λ(M, Ts, T, m). (M, Ts, T, map-option (f M Ts T) m)

definition compC :: (cname ⇒ mname ⇒ ty list ⇒ ty ⇒ 'a ⇒ 'b) ⇒ 'a cdecl ⇒ 'b cdecl
where compC f ≡ λ(C,D,Fdecls,Mdecls). (C,D,Fdecls, map (compM (f C)) Mdecls)

primrec compP :: (cname ⇒ mname ⇒ ty list ⇒ ty ⇒ 'a ⇒ 'b) ⇒ 'a prog ⇒ 'b prog
where compP f (Program P) = Program (map (compC f) P)
```

Compilation preserves the program structure. Therfore lookup functions either commute with compilation (like method lookup) or are preserved by it (like the subclass relation).

```
lemma map-of-map4:
  map-of (map (λ(x,a,b,c).(x,a,b,f x a b c)) ts) =
  (λx. map-option (λ(a,b,c).(a,b,f x a b c)) (map-of ts x))
apply(induct ts)
 apply simp
apply(rule ext)
apply fastforce
done

lemma class-compP:
  class P C = Some (D, fs, ms)
  ⟹ class (compP f P) C = Some (D, fs, map (compM (f C)) ms)
by(cases P)(simp add:class-def compP-def compC-def map-of-map4)

lemma class-compPD:
  class (compP f P) C = Some (D, fs, cms)
  ⟹ ∃ms. class P C = Some(D,fs,ms) ∧ cms = map (compM (f C)) ms
by(cases P)(clarsimp simp add:class-def compP-def compC-def map-of-map4)
```

lemma [*simp*]: *is-class* (*compP f P*) *C* = *is-class P C*

lemma [*simp*]: *class* (*compP f P*) *C* = *map-option* (λ*c*. *snd*(*compC f* (*C*,*c*))) (*class P C*)
lemma *sees-methods-compP*:
P ⊢ *C sees-methods Mm* ⟹
compP f P ⊢ *C sees-methods* (λ*M*. *map-option* (λ((*Ts*,*T*,*m*),*D*). ((*Ts*,*T*,*map-option* (*f D M Ts T*) *m*),*D*)) (*Mm M*))

lemma *sees-method-compP*:
P ⊢ *C sees M*: *Ts*→*T* = *m in D* ⟹
compP f P ⊢ *C sees M*: *Ts*→*T* = *map-option* (*f D M Ts T*) *m in D*

lemma [*simp*]:
P ⊢ *C sees M*: *Ts*→*T* = *m in D* ⟹
method (*compP f P*) *C M* = (*D*,*Ts*,*T*,*map-option* (*f D M Ts T*) *m*)

lemma *sees-methods-compPD*:
⟦ *cP* ⊢ *C sees-methods Mm*′; *cP* = *compP f P* ⟧ ⟹
∃ *Mm*. *P* ⊢ *C sees-methods Mm* ∧
Mm′ = (λ*M*. *map-option* (λ((*Ts*,*T*,*m*),*D*). ((*Ts*,*T*,*map-option* (*f D M Ts T*) *m*),*D*)) (*Mm M*))

lemma *sees-method-compPD*:
compP f P ⊢ *C sees M*: *Ts*→*T* = *fm in D* ⟹
∃ *m*. *P* ⊢ *C sees M*: *Ts*→*T* = *m in D* ∧ *map-option* (*f D M Ts T*) *m* = *fm*
lemma *sees-method-native-compP* [*simp*]:
compP f P ⊢ *C sees M*:*Ts* → *T* = *Native in D* ⟷ *P* ⊢ *C sees M*:*Ts* → *T* = *Native in D*
by(*auto dest*: *sees-method-compPD sees-method-compP*)

lemma [*simp*]: *subcls1*(*compP f P*) = *subcls1 P*
by(*fastforce simp add*: *is-class-def compC-def intro*:*subcls1I order-antisym dest*:*subcls1D*)

lemma [*simp*]: *is-type* (*compP f P*) *T* = *is-type P T*
by(*induct T*)(*auto cong*: *ty.case-cong*)

lemma *is-type-compP* [*simp*]: *is-type* (*compP f P*) = *is-type P*
by *auto*

lemma *compP-widen*[*simp*]:
(*compP f P* ⊢ *T* ≤ *T*′) = (*P* ⊢ *T* ≤ *T*′)
by(*induct T*′ *arbitrary*: *T*)(*simp-all add*: *widen-Class widen-Array*)

lemma [*simp*]: (*compP f P* ⊢ *Ts* [≤] *Ts*′) = (*P* ⊢ *Ts* [≤] *Ts*′)
lemma *is-lub-compP* [*simp*]:
is-lub (*compP f P*) = *is-lub P*
by(*auto intro*!: *ext elim*!: *is-lub.cases intro*: *is-lub.intros*)

lemma [*simp*]:
fixes *f* :: *cname* ⇒ *mname* ⇒ *ty list* ⇒ *ty* ⇒ *'a* ⇒ *'b*
shows (*compP f P* ⊢ *C has-fields FDTs*) = (*P* ⊢ *C has-fields FDTs*)

lemma [*simp*]: *fields* (*compP f P*) *C* = *fields P C*

lemma [*simp*]: (*compP f P* ⊢ *C sees F*:*T* (*fm*) *in D*) = (*P* ⊢ *C sees F*:*T* (*fm*) *in D*)

lemma [*simp*]: *field* (*compP f P*) *F D* = *field P F D*

7.8.1 Invariance of *wf-prog* under compilation

lemma [*iff*]: *distinct-fst* (*classes* (*compP f P*)) = *distinct-fst* (*classes P*)

lemma [*iff*]: *distinct-fst* (*map* (*compM f*) *ms*) = *distinct-fst ms*

lemma [*iff*]: *wf-syscls* (*compP f P*) = *wf-syscls P*
unfolding *wf-syscls-def* **by** *auto*

lemma [*iff*]: *wf-fdecl* (*compP f P*) = *wf-fdecl P*

lemma *set-compP*:
(*class* (*compP f P*) *C* = ⌊(*D*,*fs*,*ms'*)⌋) ⟷
(∃ *ms*. *class P C* = ⌊(*D*,*fs*,*ms*)⌋ ∧ *ms'* = *map* (*compM* (*f C*)) *ms*)
by(*cases P*)(*auto simp add*: *compC-def image-iff map-of-map4*)

lemma *compP-has-method*: *compP f P* ⊢ *C has M* ⟷ *P* ⊢ *C has M*
unfolding *has-method-def*
by(*fastforce dest*: *sees-method-compPD intro*: *sees-method-compP*)

lemma *is-native-compP* [*simp*]: *is-native* (*compP f P*) = *is-native P*
by(*auto simp add*: *fun-eq-iff is-native.simps*)

lemma *τexternal-compP* [*simp*]:
τexternal (*compP f P*) = *τexternal P*
by(*auto intro*!: *ext simp add*: *τexternal-def*)

context *heap-base* **begin**

lemma *heap-clone-compP* [*simp*]:
heap-clone (*compP f P*) = *heap-clone P*
by(*intro ext*)(*auto elim*!: *heap-clone.cases intro*: *heap-clone.intros*)

lemma *red-external-compP* [*simp*]:
compP f P,*t* ⊢ ⟨*a*·*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h'*⟩ ⟷ *P*,*t* ⊢ ⟨*a*·*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h'*⟩
by(*auto elim*!: *red-external.cases intro*: *red-external.intros*)

lemma *τexternal'-compP* [*simp*]:
τexternal' (*compP f P*) = *τexternal' P*
by(*simp add*: *τexternal'-def* [*abs-def*])

end

lemma *wf-overriding-compP* [*simp*]: *wf-overriding* (*compP f P*) *D* (*compM* (*f C*) *m*) = *wf-overriding P D m*
by(*cases m*)(*fastforce intro*: *sees-method-compP*[**where** *f*=*f*] *dest*: *sees-method-compPD*[**where** *f*=*f*] *simp add*: *compM-def*)

lemma *wf-cdecl-compPI*:
assumes *wf1-imp-wf2*:
⋀*C M Ts T m*. ⟦ *wf-mdecl wf*$_1$ *P C* (*M*,*Ts*,*T*,⌊*m*⌋); *P* ⊢ *C sees M*:*Ts*→*T* = ⌊*m*⌋ *in C* ⟧

```
    ⟹ wf-mdecl wf₂ (compP f P) C (M,Ts,T, ⌊f C M Ts T m⌋)
  and wfcP1: ∀ C rest. class P C = ⌊rest⌋ ⟶ wf-cdecl wf₁ P (C, rest)
  and xcomp: class (compP f P) C = ⌊rest'⌋
  and wf: wf-prog p P
  shows wf-cdecl wf₂ (compP f P) (C, rest')
proof −
  obtain D fs ms' where x: rest' = (D, fs, ms') by(cases rest')
  with xcomp obtain ms where xsrc: class P C = ⌊(D,fs,ms)⌋
    and ms': ms' = map (compM (f C)) ms
    by(auto simp add: set-compP compC-def)
  from xsrc wfcP1 have wf1: wf-cdecl wf₁ P (C,D,fs,ms) by blast
  { fix field
    assume field ∈ set fs
    with wf1 have wf-fdecl (compP f P) field by(simp add: wf-cdecl-def)
  }
  moreover from wf1 have distinct-fst fs by(simp add: wf-cdecl-def)
  moreover
  { fix m
    assume mset': m ∈ set ms'
    obtain M Ts' T' body' where m: m = (M, Ts', T', body') by(cases m)
    with ms' obtain body where mf: body' = map-option (f C M Ts' T') body
      and mset: (M, Ts', T', body) ∈ set ms using mset'
      by(clarsimp simp add: image-iff compM-def)
    moreover from mset xsrc wfcP1 have wf-mdecl wf₁ P C (M,Ts',T',body)
      by(fastforce simp add: wf-cdecl-def)
    moreover from wf xsrc mset x have P ⊢ C sees M:Ts'→T' = body in C
      by(auto intro: mdecl-visible)
    ultimately have wf-mdecl wf₂ (compP f P) C m using m
      by(cases body)(simp add: wf-mdecl-def, auto intro: wf1-imp-wf2) }
  moreover from wf1 have distinct-fst ms by(simp add: wf-cdecl-def)
  with ms' have distinct-fst ms' by(auto)
  moreover
  { assume CObj: C ≠ Object
    with xsrc wfcP1
    have part1: is-class (compP f P) D ¬ compP f P ⊢ D ≼* C
      by(auto simp add: wf-cdecl-def)
    { fix m
      assume mset': m ∈ set ms'
      obtain M Ts T body' where m: m = (M, Ts, T, body') by(cases m)
      with mset' ms' obtain body where mf: body' = map-option (f C M Ts T) body
        and mset: (M, Ts, T, body) ∈ set ms
        by(clarsimp simp add: image-iff compM-def)
      from wf1 CObj mset
    have wf-overriding P D (M, Ts, T, body) by(auto simp add: wf-cdecl-def simp del: wf-overriding.simps)
      hence wf-overriding (compP f P) D m unfolding m mf
     by(subst (asm) wf-overriding-compP[symmetric, where f=f and C=C])(simp del: wf-overriding.simps
wf-overriding-compP add: compM-def) }
    note this part1 }
  moreover
  { assume C = Thread
    with wf1 ms' have ∃ m. (run, [], Void, m) ∈ set ms'
      by(fastforce simp add: wf-cdecl-def image-iff compM-def)+ }
  ultimately show ?thesis unfolding x wf-cdecl-def by blast
qed
```

lemma *wf-prog-compPI*:
assumes *lift*:
⋀*C M Ts T m*.
⟦ *P* ⊢ *C sees M*:*Ts*→*T* = ⌊*m*⌋ *in C*; *wf-mdecl wf*$_1$ *P C* (*M*,*Ts*,*T*,⌊*m*⌋) ⟧
⟹ *wf-mdecl wf*$_2$ (*compP f P*) *C* (*M*,*Ts*,*T*, ⌊*f C M Ts T m*⌋)
and *wf*: *wf-prog wf*$_1$ *P*
shows *wf-prog wf*$_2$ (*compP f P*)
using *wf*
apply (*clarsimp simp add*:*wf-prog-def2*)
apply(*rule wf-cdecl-compPI*[*OF lift*], *assumption*+)
apply(*auto intro*: *wf*)
done

lemma *wf-cdecl-compPD*:
assumes *wf1-imp-wf2*:
⋀*C M Ts T m*. ⟦ *wf-mdecl wf*$_1$ (*compP f P*) *C* (*M*,*Ts*,*T*,⌊*f C M Ts T m*⌋); *compP f P* ⊢ *C sees M*:*Ts*→*T* = ⌊*f C M Ts T m*⌋ *in C* ⟧
⟹ *wf-mdecl wf*$_2$ *P C* (*M*,*Ts*,*T*, ⌊*m*⌋)
and *wfcP1*: ∀ *C rest*. *class* (*compP f P*) *C* = ⌊*rest*⌋ ⟶ *wf-cdecl wf*$_1$ (*compP f P*) (*C*, *rest*)
and *xcomp*: *class P C* = ⌊*rest*⌋
and *wf*: *wf-prog wf-md* (*compP f P*)
shows *wf-cdecl wf*$_2$ *P* (*C*, *rest*)
proof −
obtain *D fs ms*′ **where** *x*: *rest* = (*D*, *fs*, *ms*′) **by**(*cases rest*)
with *xcomp* **have** *xsrc*: *class* (*compP f P*) *C* = ⌊(*D*,*fs*,*map* (*compM* (*f C*)) *ms*′)⌋
by(*auto simp add*: *set-compP compC-def*)
from *xsrc wfcP1* **have** *wf1*: *wf-cdecl wf*$_1$ (*compP f P*) (*C*,*D*,*fs*,*map* (*compM* (*f C*)) *ms*′) **by** *blast*
{ fix *field*
assume *field* ∈ *set fs*
with *wf1* **have** *wf-fdecl P field* **by**(*simp add*: *wf-cdecl-def*)
}
moreover from *wf1* **have** *distinct-fst fs* **by**(*simp add*: *wf-cdecl-def*)
moreover
{ fix *m*
assume *mset*′: *m* ∈ *set ms*′
obtain *M Ts*′ *T*′ *body*′ **where** *m*: *m* = (*M*, *Ts*′, *T*′, *body*′) **by**(*cases m*)
hence *mset*: (*M*, *Ts*′, *T*′, *map-option* (*f C M Ts*′ *T*′) *body*′) ∈ *set* (*map* (*compM* (*f C*)) *ms*′) **using** *mset*′
by(*auto simp add*: *image-iff compM-def intro*: *rev-bexI*)
moreover from *wf xsrc mset x* **have** *compP f P* ⊢ *C sees M*:*Ts*′→*T*′ = *map-option* (*f C M Ts*′ *T*′) *body*′ *in C*
by(*auto intro*: *mdecl-visible*)
moreover from *mset wfcP1*[*rule-format*, *OF xsrc*]
have *wf-mdecl wf*$_1$ (*compP f P*) *C* (*M*,*Ts*′,*T*′,*map-option* (*f C M Ts*′ *T*′) *body*′)
by(*auto simp add*: *wf-cdecl-def*)
ultimately have *wf-mdecl wf*$_2$ *P C m* **using** *m*
by(*cases body*′)(*simp add*: *wf-mdecl-def*, *auto intro*: *wf1-imp-wf2*) **}**
moreover from *wf1* **have** *distinct-fst ms*′ **by**(*simp add*: *wf-cdecl-def*)
moreover
{ assume *CObj*: *C* ≠ *Object*
with *xsrc wfcP1*
have *part1*: *is-class P D* ¬ *P* ⊢ *D* ≼* *C*
by(*auto simp add*: *wf-cdecl-def*)

```
    { fix m
      assume mset': m ∈ set ms'
      with wf1 CObj have wf-overriding (compP f P) D (compM (f C) m)
        by(simp add: wf-cdecl-def del: wf-overriding-compP)
      hence wf-overriding P D m by simp }
    note this part1 }
  moreover
  { assume C = Thread
    with wf1 have ∃ m. (run, [], Void, m) ∈ set ms'
      by(fastforce simp add: wf-cdecl-def image-iff compM-def)+ }
  ultimately show ?thesis unfolding x wf-cdecl-def by blast
qed

lemma wf-prog-compPD:
assumes wf: wf-prog wf1 (compP f P)
and lift:
  ⋀C M Ts T m.
    ⟦ compP f P ⊢ C sees M:Ts→T = ⌊f C M Ts T m⌋ in C; wf-mdecl wf1 (compP f P) C (M,Ts,T,
⌊f C M Ts T m⌋) ⟧
    ⟹ wf-mdecl wf2 P C (M,Ts,T,⌊m⌋)
shows wf-prog wf2 P
using wf
apply(clarsimp simp add:wf-prog-def2)
apply(rule wf-cdecl-compPD[OF lift], assumption+)
apply(auto intro: wf)
done

lemma WT-binop-compP [simp]: compP f P ⊢ T1 «bop» T2 :: T ⟷ P ⊢ T1 «bop» T2 :: T
by(cases bop)(fastforce)+

lemma WTrt-binop-compP [simp]: compP f P ⊢ T1 «bop» T2 : T ⟷ P ⊢ T1 «bop» T2 : T
by(cases bop)(fastforce)+

lemma binop-relevant-class-compP [simp]: binop-relevant-class bop (compP f P) = binop-relevant-class
bop P
by(cases bop) simp-all

lemma is-class-compP [simp]:
  is-class (compP f P) = is-class P
by(simp add: is-class-def fun-eq-iff)

lemma has-field-compP [simp]:
  compP f P ⊢ C has F:T (fm) in D ⟷ P ⊢ C has F:T (fm) in D
by(auto simp add: has-field-def)

context heap-base begin

lemma compP-addr-loc-type [simp]:
  addr-loc-type (compP f P) = addr-loc-type P
by(auto elim!: addr-loc-type.cases intro: addr-loc-type.intros intro!: ext)

lemma conf-compP [simp]:
  compP f P,h ⊢ v :≤ T ⟷ P,h ⊢ v :≤ T
by(simp add: conf-def)
```

lemma *compP-conf*: *conf* (*compP f P*) = *conf P*
by(*auto simp add*: *conf-def intro*!: *ext*)

lemma *compP-confs*: *compP f P*,*h* ⊢ *vs* [:≤] *Ts* ⟷ *P*,*h* ⊢ *vs* [:≤] *Ts*
by(*simp add*: *compP-conf*)

lemma *tconf-compP* [*simp*]: *compP f P*, *h* ⊢ *t* √*t* ⟷ *P*,*h* ⊢ *t* √*t*
by(*auto simp add*: *tconf-def*)

lemma *wf-start-state-compP* [*simp*]:
wf-start-state (*compP f P*) = *wf-start-state P*
by(*auto 4 6 simp add*: *fun-eq-iff wf-start-state.simps compP-conf dest*: *sees-method-compP*[**where** *f*=*f*] *sees-method-compPD*[**where** *f*=*f*])

end

lemma *compP-addr-conv*:
addr-conv addr2thread-id thread-id2addr typeof-addr (*compP f P*) = *addr-conv addr2thread-id thread-id2addr typeof-addr P*
unfolding *addr-conv-def*
by *simp*

lemma *compP-heap*:
heap addr2thead-id thread-id2addr allocate typeof-addr heap-write (*compP f P*) =
heap addr2thead-id thread-id2addr allocate typeof-addr heap-write P
unfolding *heap-def compP-addr-conv heap-axioms-def*
by *auto*

lemma *compP-heap-conf*:
heap-conf addr2thead-id thread-id2addr empty-heap allocate typeof-addr heap-write hconf (*compP f P*) =
heap-conf addr2thead-id thread-id2addr empty-heap allocate typeof-addr heap-write hconf P
unfolding *heap-conf-def heap-conf-axioms-def compP-heap*
unfolding *heap-base.compP-conf heap-base.compP-addr-loc-type is-type-compP is-class-compP*
by(*rule refl*)

lemma *compP-heap-conf-read*:
heap-conf-read addr2thead-id thread-id2addr empty-heap allocate typeof-addr heap-read heap-write hconf (*compP f P*) =
heap-conf-read addr2thead-id thread-id2addr empty-heap allocate typeof-addr heap-read heap-write hconf P
unfolding *heap-conf-read-def heap-conf-read-axioms-def*
unfolding *compP-heap-conf heap-base.compP-conf heap-base.compP-addr-loc-type*
by(*rule refl*)

compiler composition

lemma *compM-compM*:
compM f (*compM g md*) = *compM* (λ*M Ts T*. *f M Ts T* ∘ *g M Ts T*) *md*
by(*cases md*)(*simp add*: *compM-def option.map-comp o-def*)

lemma *compC-compC*:
compC f (*compC g cd*) = *compC* (λ*C M Ts T*. *f C M Ts T* ∘ *g C M Ts T*) *cd*
by(*simp add*: *compC-def split-beta compM-compM*)

lemma *compP-compP*:
compP f (*compP g P*) = *compP* (λ*C M Ts T*. *f C M Ts T* ∘ *g C M Ts T*) *P*
by(*cases P*)(*simp add*: *compC-compC*)

end

7.9 Compilation Stage 2

theory *Compiler2*
imports *PCompiler J1State ../JVM/JVMInstructions*
begin

primrec *compE2* :: *'addr expr1* ⇒ *'addr instr list*
and *compEs2* :: *'addr expr1 list* ⇒ *'addr instr list*
where
compE2 (*new C*) = [*New C*]
| *compE2* (*newA T*⌊*e*⌉) = *compE2 e* @ [*NewArray T*]
| *compE2* (*Cast T e*) = *compE2 e* @ [*Checkcast T*]
| *compE2* (*e instanceof T*) = *compE2 e* @ [*Instanceof T*]
| *compE2* (*Val v*) = [*Push v*]
| *compE2* (*e1* «*bop*» *e2*) = *compE2 e1* @ *compE2 e2* @ [*BinOpInstr bop*]
| *compE2* (*Var i*) = [*Load i*]
| *compE2* (*i*:=*e*) = *compE2 e* @ [*Store i*, *Push Unit*]
| *compE2* (*a*⌊*i*⌉) = *compE2 a* @ *compE2 i* @ [*ALoad*]
| *compE2* (*a*⌊*i*⌉ := *e*) = *compE2 a* @ *compE2 i* @ *compE2 e* @ [*AStore*, *Push Unit*]
| *compE2* (*a*·*length*) = *compE2 a* @ [*ALength*]
| *compE2* (*e*·*F*{*D*}) = *compE2 e* @ [*Getfield F D*]
| *compE2* (*e1*·*F*{*D*} := *e2*) = *compE2 e1* @ *compE2 e2* @ [*Putfield F D*, *Push Unit*]
| *compE2* (*e*·*compareAndSwap*(*D*·*F*, *e'*, *e''*)) = *compE2 e* @ *compE2 e'* @ *compE2 e''* @ [*CAS F D*]
| *compE2* (*e*·*M*(*es*)) = *compE2 e* @ *compEs2 es* @ [*Invoke M* (*size es*)]
| *compE2* ({*i*:*T*=*vo*; *e*}) = (*case vo of None* ⇒ [] | ⌊*v*⌋ ⇒ [*Push v*, *Store i*]) @ *compE2 e*
| *compE2* (*sync*$_V$ (*o'*) *e*) = *compE2 o'* @ [*Dup*, *Store V*, *MEnter*] @
compE2 e @ [*Load V*, *MExit*, *Goto 4*, *Load V*, *MExit*, *ThrowExc*]
| *compE2* (*insync*$_V$ (*a*) *e*) = [*Goto 1*] — Define insync sensibly
| *compE2* (*e1*;;*e2*) = *compE2 e1* @ [*Pop*] @ *compE2 e2*
| *compE2* (*if* (*e*) e_1 *else* e_2) =
(*let cnd* = *compE2 e*;
thn = *compE2* e_1;
els = *compE2* e_2;
test = *IfFalse* (*int*(*size thn* + *2*));
thnex = *Goto* (*int*(*size els* + *1*))
in cnd @ [*test*] @ *thn* @ [*thnex*] @ *els*)
| *compE2* (*while* (*e*) *c*) =
(*let cnd* = *compE2 e*;
bdy = *compE2 c*;
test = *IfFalse* (*int*(*size bdy* + *3*));
loop = *Goto* (−*int*(*size bdy* + *size cnd* + *2*))
in cnd @ [*test*] @ *bdy* @ [*Pop*] @ [*loop*] @ [*Push Unit*])
| *compE2* (*throw e*) = *compE2 e* @ [*ThrowExc*]
| *compE2* (*try e1 catch*(*C i*) *e2*) =
(*let catch* = *compE2 e2*
in compE2 e1 @ [*Goto* (*int*(*size catch*)+*2*), *Store i*] @ *catch*)

```
| compEs2 []     = []
| compEs2 (e#es) = compE2 e @ compEs2 es
```

Compilation of exception table. Is given start address of code to compute absolute addresses necessary in exception table.

```
fun compxE2 :: 'addr expr1       ⇒ pc ⇒ nat ⇒ ex-table
and compxEs2 :: 'addr expr1 list ⇒ pc ⇒ nat ⇒ ex-table
where
  compxE2 (new C) pc d = []
| compxE2 (newA T⌊e⌋) pc d = compxE2 e pc d
| compxE2 (Cast T e) pc d = compxE2 e pc d
| compxE2 (e instanceof T) pc d = compxE2 e pc d
| compxE2 (Val v) pc d = []
| compxE2 (e1 «bop» e2) pc d =
   compxE2 e1 pc d @ compxE2 e2 (pc + size(compE2 e1)) (d+1)
| compxE2 (Var i) pc d = []
| compxE2 (i:=e) pc d = compxE2 e pc d
| compxE2 (a⌊i⌋) pc d = compxE2 a pc d @ compxE2 i (pc + size (compE2 a)) (d + 1)
| compxE2 (a⌊i⌋ := e) pc d =
        (let pc1 = pc  + size (compE2 a);
             pc2 = pc1 + size (compE2 i)
         in compxE2 a pc d @ compxE2 i pc1 (d + 1) @ compxE2 e pc2 (d + 2))
| compxE2 (a·length) pc d = compxE2 a pc d
| compxE2 (e·F{D}) pc d = compxE2 e pc d
| compxE2 (e1·F{D} := e2) pc d = compxE2 e1 pc d @ compxE2 e2 (pc + size (compE2 e1)) (d +
1)
| compxE2 (e·compareAndSwap(D·F, e', e'')) pc d =
   (let pc1 = pc  + size (compE2 e);
        pc2 = pc1 + size (compE2 e')
    in compxE2 e pc d @ compxE2 e' pc1 (d + 1) @ compxE2 e'' pc2 (d + 2))
| compxE2 (e·M(es)) pc d = compxE2 e pc d @ compxEs2 es (pc + size(compE2 e)) (d+1)
| compxE2 ({i:T=vo; e}) pc d = compxE2 e (case vo of None ⇒ pc | ⌊v⌋ ⇒ Suc (Suc pc)) d
| compxE2 (sync_V (o') e) pc d =
         (let pc1 = pc + size (compE2 o') + 3;
              pc2 = pc1 + size(compE2 e)
          in compxE2 o' pc d @ compxE2 e pc1 d @ [(pc1, pc2, None, Suc (Suc (Suc pc2)), d)])
| compxE2 (insync_V (a) e) pc d = []
| compxE2 (e1;;e2) pc d =
   compxE2 e1 pc d @ compxE2 e2 (pc+size(compE2 e1)+1) d
| compxE2 (if (e) e1 else e2) pc d =
         (let pc1  = pc + size(compE2 e) + 1;
              pc2  = pc1 + size(compE2 e1) + 1
          in compxE2 e pc d @ compxE2 e1 pc1 d @ compxE2 e2 pc2 d)
| compxE2 (while (b) e) pc d =
   compxE2 b pc d @ compxE2 e (pc+size(compE2 b)+1) d
| compxE2 (throw e) pc d = compxE2 e pc d
| compxE2 (try e1 catch(C i) e2) pc d =
   (let pc1 = pc + size(compE2 e1)
    in compxE2 e1 pc d @ compxE2 e2 (pc1+2) d @ [(pc,pc1,Some C,pc1+1,d)])

| compxEs2 [] pc d    = []
| compxEs2 (e#es) pc d = compxE2 e pc d @ compxEs2 es (pc+size(compE2 e)) (d+1)
```

lemmas *compxE2-compxEs2-induct* =
compxE2-compxEs2.induct[
unfolded meta-all5-eq-conv meta-all4-eq-conv meta-all3-eq-conv meta-all2-eq-conv meta-all-eq-conv,
case-names
new NewArray Cast InstanceOf Val BinOp Var LAss AAcc AAss ALen FAcc FAss Call Block
Synchronized InSynchronized Seq Cond While throw TryCatch
Nil Cons]

lemma *compE2-neq-Nil* [*simp*]: *compE2 e* ≠ []
by(*induct e*) *auto*

declare *compE2-neq-Nil*[*symmetric*, *simp*]

lemma *compEs2-append* [*simp*]: *compEs2* (*es* @ *es'*) = *compEs2 es* @ *compEs2 es'*
by(*induct es*) *auto*

lemma *compEs2-eq-Nil-conv* [*simp*]: *compEs2 es* = [] ⟷ *es* = []
by(*cases es*) *auto*

lemma *compEs2-map-Val*: *compEs2* (*map Val vs*) = *map Push vs*
by(*induct vs*) *auto*

lemma *compE2-0th-neq-Invoke* [*simp*]:
compE2 e ! *0* ≠ *Invoke M n*
by(*induct e*)(*auto simp add*: *nth-append*)

declare *compE2-0th-neq-Invoke*[*symmetric*, *simp*]

lemma *compxEs2-append* [*simp*]:
compxEs2 (*es* @ *es'*) *pc d* = *compxEs2 es pc d* @ *compxEs2 es'* (*length* (*compEs2 es*) + *pc*) (*length es* + *d*)
by(*induct es arbitrary*: *pc d*)(*auto simp add*: *ac-simps*)

lemma *compxEs2-map-Val* [*simp*]: *compxEs2* (*map Val vs*) *pc d* = []
by(*induct vs arbitrary*: *d pc*) *auto*

lemma *compE2-blocks1* [*simp*]:
compE2 (*blocks1 n Ts body*) = *compE2 body*
by(*induct n Ts body rule*: *blocks1.induct*)(*auto*)

lemma *compxE2-blocks1* [*simp*]:
compxE2 (*blocks1 n Ts body*) = *compxE2 body*
by(*induct n Ts body rule*: *blocks1.induct*)(*auto intro*!: *ext*)

lemma fixes *e* :: *'addr expr1* **and** *es* :: *'addr expr1 list*
shows *compE2-not-Return*: *Return* ∉ *set* (*compE2 e*)
and *compEs2-not-Return*: *Return* ∉ *set* (*compEs2 es*)
by(*induct e* **and** *es rule*: *compE2.induct compEs2.induct*)(*auto*)

primrec *max-stack* :: *'addr expr1* ⇒ *nat*
and *max-stacks* :: *'addr expr1 list* ⇒ *nat*
where
max-stack (*new C*) = *1*
| *max-stack* (*newA T*⌊*e*⌋) = *max-stack e*

| *max-stack* (*Cast C e*) = *max-stack e*
| *max-stack* (*e instanceof T*) = *max-stack e*
| *max-stack* (*Val v*) = *1*
| *max-stack* (*e1* «*bop*» *e2*) = *max* (*max-stack e1*) (*max-stack e2*) + *1*
| *max-stack* (*Var i*) = *1*
| *max-stack* (*i*:=*e*) = *max-stack e*
| *max-stack* (*a*⌊*i*⌋) = *max* (*max-stack a*) (*max-stack i* + *1*)
| *max-stack* (*a*⌊*i*⌋ := *e*) = *max* (*max* (*max-stack a*) (*max-stack i* + *1*)) (*max-stack e* + *2*)
| *max-stack* (*a*•*length*) = *max-stack a*
| *max-stack* (*e*•*F*{*D*}) = *max-stack e*
| *max-stack* (*e1*•*F*{*D*} := *e2*) = *max* (*max-stack e1*) (*max-stack e2*) + *1*
| *max-stack* (*e*•*compareAndSwap*(*D*•*F*, *e′*, *e′′*)) = *max* (*max* (*max-stack e*) (*max-stack e′* + *1*)) (*max-stack e′′* + *2*)
| *max-stack* (*e*•*M*(*es*)) = *max* (*max-stack e*) (*max-stacks es*) + *1*
| *max-stack* ({*i*:*T*=*vo*; *e*}) = *max-stack e*
| *max-stack* (*sync*$_V$ (*o′*) *e*) = *max* (*max-stack o′*) (*max* (*max-stack e*) *2*)
| *max-stack* (*insync*$_V$ (*a*) *e*) = *1*
| *max-stack* (*e1*;;*e2*) = *max* (*max-stack e1*) (*max-stack e2*)
| *max-stack* (*if* (*e*) e_1 *else* e_2) =
  max (*max-stack e*) (*max* (*max-stack* e_1) (*max-stack* e_2))
| *max-stack* (*while* (*e*) *c*) = *max* (*max-stack e*) (*max-stack c*)
| *max-stack* (*throw e*) = *max-stack e*
| *max-stack* (*try e1 catch*(*C i*) *e2*) = *max* (*max-stack e1*) (*max-stack e2*)

| *max-stacks* [] = *0*
| *max-stacks* (*e*#*es*) = *max* (*max-stack e*) (*1* + *max-stacks es*)

lemma *max-stack1*: *1* ≤ *max-stack e*
lemma *max-stacks-ge-length*: *max-stacks es* ≥ *length es*
by(*induct es*, *auto*)

lemma *max-stack-blocks1* [*simp*]:
  max-stack (*blocks1 n Ts body*) = *max-stack body*
by(*induct n Ts body rule*: *blocks1.induct*) *auto*

definition *compMb2* :: *′addr expr1* ⇒ *′addr jvm-method*
where
  compMb2 ≡ λ*body*.
  let ins = *compE2 body* @ [*Return*];
      xt = *compxE2 body 0 0*
  in (*max-stack body*, *max-vars body*, *ins*, *xt*)

definition *compP2* :: *′addr J1-prog* ⇒ *′addr jvm-prog*
where *compP2* ≡ *compP* (λ*C M Ts T*. *compMb2*)

lemma *compMb2*:
  compMb2 e = (*max-stack e*, *max-vars e*, (*compE2 e* @ [*Return*]), *compxE2 e 0 0*)
by (*simp add*: *compMb2-def*)

end

7.10 Various Operations for Exception Tables

theory *Exception-Tables* **imports**
Compiler2
../Common/ExternalCallWF
../JVM/JVMExceptions
begin

definition *pcs* :: *ex-table* ⇒ *nat set*
where *pcs xt* ≡ ⋃(*f*,*t*,*C*,*h*,*d*) ∈ *set xt*. {*f* ..< *t*}

lemma *pcs-subset*:
fixes *e* :: *'addr expr1* **and** *es* :: *'addr expr1 list*
shows *pcs*(*compxE2 e pc d*) ⊆ {*pc*..<*pc*+*size*(*compE2 e*)}
and *pcs*(*compxEs2 es pc d*) ⊆ {*pc*..<*pc*+*size*(*compEs2 es*)}
apply(*induct e pc d* **and** *es pc d rule*: *compxE2-compxEs2-induct*)
apply (*simp-all add*:*pcs-def*)
apply (*fastforce*)+
done

lemma *pcs-Nil* [*simp*]: *pcs* [] = {}
by(*simp add*:*pcs-def*)

lemma *pcs-Cons* [*simp*]: *pcs* (*x*#*xt*) = {*fst x* ..< *fst*(*snd x*)} ∪ *pcs xt*
by(*auto simp add*: *pcs-def*)

lemma *pcs-append* [*simp*]: *pcs*(*xt*$_1$ @ *xt*$_2$) = *pcs xt*$_1$ ∪ *pcs xt*$_2$
by(*simp add*:*pcs-def*)

lemma [*simp*]: *pc* < *pc*$_0$ ∨ *pc*$_0$+*size*(*compE2 e*) ≤ *pc* ⟹ *pc* ∉ *pcs*(*compxE2 e pc*$_0$ *d*)
using *pcs-subset* **by** *fastforce*

lemma [*simp*]: *pc* < *pc0* ∨ *pc0*+*size*(*compEs2 es*) ≤ *pc* ⟹ *pc* ∉ *pcs*(*compxEs2 es pc0 d*)
using *pcs-subset* **by** *fastforce*

lemma [*simp*]: *pc1* + *size*(*compE2 e1*) ≤ *pc2* ⟹ *pcs*(*compxE2 e1 pc1 d1*) ∩ *pcs*(*compxE2 e2 pc2 d2*) = {}
using *pcs-subset* **by** *fastforce*

lemma [*simp*]: *pc*$_1$ + *size*(*compE2 e*) ≤ *pc*$_2$ ⟹ *pcs*(*compxE2 e pc*$_1$ *d*$_1$) ∩ *pcs*(*compxEs2 es pc*$_2$ *d*$_2$) = {}
using *pcs-subset* **by** *fastforce*

lemma *match-ex-table-append-not-pcs* [*simp*]:
pc ∉ *pcs xt0* ⟹ *match-ex-table P C pc* (*xt0* @ *xt1*) = *match-ex-table P C pc xt1*
by (*induct xt0*) (*auto simp*: *matches-ex-entry-def*)

lemma *outside-pcs-not-matches-entry* [*simp*]:
⟦ *x* ∈ *set xt*; *pc* ∉ *pcs xt* ⟧ ⟹ ¬ *matches-ex-entry P D pc x*
by(*auto simp*:*matches-ex-entry-def pcs-def*)

lemma *outside-pcs-compxE2-not-matches-entry* [*simp*]:
assumes *xe*: *xe* ∈ *set*(*compxE2 e pc d*)
and *outside*: *pc'* < *pc* ∨ *pc*+*size*(*compE2 e*) ≤ *pc'*

shows ¬ *matches-ex-entry P C pc′ xe*
proof
assume *matches-ex-entry P C pc′ xe*
with *xe* **have** *pc′* ∈ *pcs*(*compxE2 e pc d*)
by(*force simp add*:*matches-ex-entry-def pcs-def*)
with *outside* **show** *False* **by** *simp*
qed

lemma *outside-pcs-compxEs2-not-matches-entry* [*simp*]:
assumes *xe*: *xe* ∈ *set*(*compxEs2 es pc d*)
and *outside*: *pc′* < *pc* ∨ *pc*+*size*(*compEs2 es*) ≤ *pc′*
shows ¬ *matches-ex-entry P C pc′ xe*
proof
assume *matches-ex-entry P C pc′ xe*
with *xe* **have** *pc′* ∈ *pcs*(*compxEs2 es pc d*)
by(*force simp add*:*matches-ex-entry-def pcs-def*)
with *outside* **show** *False* **by** *simp*
qed

lemma *match-ex-table-app*[*simp*]:
∀ *xte* ∈ *set xt*$_1$. ¬ *matches-ex-entry P D pc xte* ⟹
match-ex-table P D pc (*xt*$_1$ @ *xt*) = *match-ex-table P D pc xt*
by(*induct xt*$_1$) *simp-all*

lemma *match-ex-table-eq-NoneI* [*simp*]:
∀ *x* ∈ *set xtab*. ¬ *matches-ex-entry P C pc x* ⟹
match-ex-table P C pc xtab = *None*
using *match-ex-table-app*[**where** *?xt* = []] **by** *fastforce*

lemma *match-ex-table-not-pcs-None*:
pc ∉ *pcs xt* ⟹ *match-ex-table P C pc xt* = *None*
by(*auto intro*: *match-ex-table-eq-NoneI*)

lemma *match-ex-entry*:
fixes *start* **shows**
matches-ex-entry P C pc (*start*, *end*, *catch-type*, *handler*) =
(*start* ≤ *pc* ∧ *pc* < *end* ∧ (*case catch-type of None* ⇒ *True* | ⌊*C′*⌋ ⇒ *P* ⊢ *C* ⪯* *C′*))
by(*simp add*:*matches-ex-entry-def*)

lemma *pcs-compxE2D* [*dest*]:
pc ∈ *pcs* (*compxE2 e pc′ d*) ⟹ *pc′* ≤ *pc* ∧ *pc* < *pc′* + *length* (*compE2 e*)
using *pcs-subset* **by**(*fastforce*)

lemma *pcs-compxEs2D* [*dest*]:
pc ∈ *pcs* (*compxEs2 es pc′ d*) ⟹ *pc′* ≤ *pc* ∧ *pc* < *pc′* + *length* (*compEs2 es*)
using *pcs-subset* **by**(*fastforce*)

definition *shift* :: *nat* ⇒ *ex-table* ⇒ *ex-table*
where
shift n xt ≡ *map* (λ(*from*,*to*,*C*,*handler*,*depth*). (*n*+*from*,*n*+*to*,*C*,*n*+*handler*,*depth*)) *xt*

lemma *shift-0* [*simp*]: *shift 0 xt* = *xt*
by(*induct xt*)(*auto simp*:*shift-def*)

lemma *shift-Nil* [*simp*]: *shift n* [] = []
by(*simp add*:*shift-def*)

lemma *shift-Cons-tuple* [*simp*]:
shift n ((*from*, *to*, *C*, *handler*, *depth*) # *xt*) = (*from* + *n*, *to* + *n*, *C*, *handler* + *n*, *depth*) # *shift n xt*
by(*simp add*: *shift-def*)

lemma *shift-append* [*simp*]: *shift n* (xt_1 @ xt_2) = *shift n* xt_1 @ *shift n* xt_2
by(*simp add*:*shift-def*)

lemma *shift-shift* [*simp*]: *shift m* (*shift n xt*) = *shift* (*m*+*n*) *xt*
by(*simp add*: *shift-def split-def*)

lemma fixes *e* :: *'addr expr1* **and** *es* :: *'addr expr1 list*
shows *shift-compxE2*: *shift pc* (*compxE2 e pc' d*) = *compxE2 e* (*pc'* + *pc*) *d*
and *shift-compxEs2*: *shift pc* (*compxEs2 es pc' d*) = *compxEs2 es* (*pc'* + *pc*) *d*
by(*induct e* **and** *es arbitrary*: *pc pc' d* **and** *pc pc' d rule*: *compE2.induct compEs2.induct*)
(*auto simp*:*shift-def ac-simps*)

lemma *compxE2-size-convs* [*simp*]: *n* ≠ *0* ⟹ *compxE2 e n d* = *shift n* (*compxE2 e 0 d*)
and *compxEs2-size-convs*: *n* ≠ *0* ⟹ *compxEs2 es n d* = *shift n* (*compxEs2 es 0 d*)
by(*simp-all add*:*shift-compxE2 shift-compxEs2*)

lemma *pcs-shift-conv* [*simp*]: *pcs* (*shift n xt*) = (+) *n* ‘ *pcs xt*
apply(*auto simp add*: *shift-def pcs-def*)
apply(*rule-tac x*=*x*−*n* **in** *image-eqI*)
apply(*auto*)
apply(*rule bexI*)
prefer *2*
apply(*assumption*)
apply(*auto*)
done

lemma *image-plus-const-conv* [*simp*]:
fixes *m* :: *nat*
shows *m* ∈ (+) *n* ‘ *A* ⟷ *m* ≥ *n* ∧ *m* − *n* ∈ *A*
by(*force*)

lemma *match-ex-table-shift-eq-None-conv* [*simp*]:
match-ex-table P C pc (*shift n xt*) = *None* ⟷ *pc* < *n* ∨ *match-ex-table P C* (*pc* − *n*) *xt* = *None*
by(*induct xt*)(*auto simp add*: *match-ex-entry split*: *if-split-asm*)

lemma *match-ex-table-shift-pc-None*:
pc ≥ *n* ⟹ *match-ex-table P C pc* (*shift n xt*) = *None* ⟷ *match-ex-table P C* (*pc* − *n*) *xt* = *None*
by(*simp add*: *match-ex-table-shift-eq-None-conv*)

lemma *match-ex-table-shift-eq-Some-conv* [*simp*]:
match-ex-table P C pc (*shift n xt*) = ⌊(*pc'*, *d*)⌋ ⟷
pc ≥ *n* ∧ *pc'* ≥ *n* ∧ *match-ex-table P C* (*pc* − *n*) *xt* = ⌊(*pc'* − *n*, *d*)⌋
by(*induct xt*)(*auto simp add*: *match-ex-entry split*: *if-split-asm*)

lemma *match-ex-table-shift*:
match-ex-table P C pc xt = ⌊(*pc'*, *d*)⌋ ⟹ *match-ex-table P C* (*n* + *pc*) (*shift n xt*) = ⌊(*n* + *pc'*, *d*)⌋

by(*simp add*: *match-ex-table-shift-eq-Some-conv*)

lemma *match-ex-table-shift-pcD*:
match-ex-table P C pc (*shift n xt*) = ⌊(*pc'*, *d*)⌋ ⟹ *pc* ≥ *n* ∧ *pc'* ≥ *n* ∧ *match-ex-table P C* (*pc* − *n*) *xt* = ⌊(*pc'* − *n*, *d*)⌋
by(*simp add*: *match-ex-table-shift-eq-Some-conv*)

lemma *match-ex-table-pcsD*: *match-ex-table P C pc xt* = ⌊(*pc'*, *D*)⌋ ⟹ *pc* ∈ *pcs xt*
by(*induct xt*)(*auto split*: *if-split-asm simp add*: *match-ex-entry*)

definition *stack-xlift* :: *nat* ⇒ *ex-table* ⇒ *ex-table*
where *stack-xlift n xt* ≡ *map* (λ(*from*,*to*,*C*,*handler*,*depth*). (*from*, *to*, *C*, *handler*, *n* + *depth*)) *xt*

lemma *stack-xlift-0* [*simp*]: *stack-xlift 0 xt* = *xt*
by(*induct xt*, *auto simp add*: *stack-xlift-def*)

lemma *stack-xlift-Nil* [*simp*]: *stack-xlift n* [] = []
by(*simp add*: *stack-xlift-def*)

lemma *stack-xlift-Cons-tuple* [*simp*]:
stack-xlift n ((*from*, *to*, *C*, *handler*, *depth*) # *xt*) = (*from*, *to*, *C*, *handler*, *depth* + *n*) # *stack-xlift n xt*
by(*simp add*: *stack-xlift-def*)

lemma *stack-xlift-append* [*simp*]: *stack-xlift n* (*xt* @ *xt'*) = *stack-xlift n xt* @ *stack-xlift n xt'*
by(*simp add*: *stack-xlift-def*)

lemma *stack-xlift-stack-xlift* [*simp*]: *stack-xlift n* (*stack-xlift m xt*) = *stack-xlift* (*n* + *m*) *xt*
by(*simp add*: *stack-xlift-def split-def*)

lemma fixes *e* :: *'addr expr1* **and** *es* :: *'addr expr1 list*
shows *stack-xlift-compxE2*: *stack-xlift n* (*compxE2 e pc d*) = *compxE2 e pc* (*n* + *d*)
and *stack-xlift-compxEs2*: *stack-xlift n* (*compxEs2 es pc d*) = *compxEs2 es pc* (*n* + *d*)
by(*induct e* **and** *es arbitrary*: *d pc* **and** *d pc rule*: *compE2.induct compEs2.induct*)
(*auto simp add*: *shift-compxE2 simp del*: *compxE2-size-convs*)

lemma *compxE2-stack-xlift-convs* [*simp*]: *d* > *0* ⟹ *compxE2 e pc d* = *stack-xlift d* (*compxE2 e pc 0*)
and *compxEs2-stack-xlift-convs* [*simp*]: *d* > *0* ⟹ *compxEs2 es pc d* = *stack-xlift d* (*compxEs2 es pc 0*)
by(*simp-all add*: *stack-xlift-compxE2 stack-xlift-compxEs2*)

lemma *stack-xlift-shift* [*simp*]: *stack-xlift d* (*shift n xt*) = *shift n* (*stack-xlift d xt*)
by(*induct xt*)(*auto*)

lemma *pcs-stack-xlift-conv* [*simp*]: *pcs* (*stack-xlift n xt*) = *pcs xt*
by(*auto simp add*: *pcs-def stack-xlift-def*)

lemma *match-ex-table-stack-xlift-eq-None-conv* [*simp*]:
match-ex-table P C pc (*stack-xlift d xt*) = *None* ⟷ *match-ex-table P C pc xt* = *None*
by(*induct xt*)(*auto simp add*: *match-ex-entry*)

lemma *match-ex-table-stack-xlift-eq-Some-conv* [*simp*]:

match-ex-table P C pc (stack-xlift n xt) = ⌊(pc', d)⌋ ⟷ d ≥ n ∧ match-ex-table P C pc xt = ⌊(pc', d − n)⌋
by(*induct xt*)(*auto simp add*: *match-ex-entry*)

lemma *match-ex-table-stack-xliftD*:
match-ex-table P C pc (stack-xlift n xt) = ⌊(pc', d)⌋ ⟹ d ≥ n ∧ match-ex-table P C pc xt = ⌊(pc', d − n)⌋
by(*simp*)

lemma *match-ex-table-stack-xlift*:
match-ex-table P C pc xt = ⌊(pc', d)⌋ ⟹ match-ex-table P C pc (stack-xlift n xt) = ⌊(pc', n + d)⌋
by *simp*

lemma *pcs-stack-xlift*: *pcs (stack-xlift n xt) = pcs xt*
by(*auto simp add*: *stack-xlift-def pcs-def*)

lemma *match-ex-table-None-append* [*simp*]:
match-ex-table P C pc xt = None
⟹ match-ex-table P C pc (xt @ xt') = match-ex-table P C pc xt'
by(*induct xt*, *auto*)

lemma *match-ex-table-Some-append* [*simp*]:
match-ex-table P C pc xt = ⌊(pc', d)⌋ ⟹ match-ex-table P C pc (xt @ xt') = ⌊(pc', d)⌋
by(*induct xt*)(*auto*)

lemma *match-ex-table-append*:
match-ex-table P C pc (xt @ xt') = (case match-ex-table P C pc xt of None ⇒ match-ex-table P C pc xt'
| Some pcd ⇒ Some pcd)
by(*auto*)

lemma *match-ex-table-pc-length-compE2*:
match-ex-table P a pc (compxE2 e pc' d) = ⌊pcd⌋ ⟹ pc' ≤ pc ∧ pc < length (compE2 e) + pc'

and *match-ex-table-pc-length-compEs2*:
match-ex-table P a pc (compxEs2 es pc' d) = ⌊pcd⌋ ⟹ pc' ≤ pc ∧ pc < length (compEs2 es) + pc'
using *pcs-subset* **by**(*cases pcd*, *fastforce dest*!: *match-ex-table-pcsD*)+

lemma *match-ex-table-compxE2-shift-conv*:
f > 0 ⟹ match-ex-table P C pc (compxE2 e f d) = ⌊(pc', d')⌋ ⟷ pc ≥ f ∧ pc' ≥ f ∧ match-ex-table P C (pc − f) (compxE2 e 0 d) = ⌊(pc' − f, d')⌋
by *simp*

lemma *match-ex-table-compxEs2-shift-conv*:
f > 0 ⟹ match-ex-table P C pc (compxEs2 es f d) = ⌊(pc', d')⌋ ⟷ pc ≥ f ∧ pc' ≥ f ∧ match-ex-table P C (pc − f) (compxEs2 es 0 d) = ⌊(pc' − f, d')⌋
by(*simp add*: *compxEs2-size-convs*)

lemma *match-ex-table-compxE2-stack-conv*:
d > 0 ⟹ match-ex-table P C pc (compxE2 e 0 d) = ⌊(pc', d')⌋ ⟷ d' ≥ d ∧ match-ex-table P C pc (compxE2 e 0 0) = ⌊(pc', d' − d)⌋
by *simp*

lemma *match-ex-table-compxEs2-stack-conv*:

$d > 0 \Longrightarrow$ *match-ex-table P C pc* (*compxEs2 es 0 d*) = $\lfloor$(*pc'*, *d'*)$\rfloor$ $\longleftrightarrow$ $d' \geq d \wedge$ *match-ex-table P C pc* (*compxEs2 es 0 0*) = $\lfloor$(*pc'*, $d' - d$)$\rfloor$
by(*simp add*: *compxEs2-stack-xlift-convs*)

lemma fixes *e* :: *'addr expr1* **and** *es* :: *'addr expr1 list*
shows *match-ex-table-compxE2-not-same*: *match-ex-table P C pc* (*compxE2 e n d*) = $\lfloor$(*pc'*, *d'*)$\rfloor$ $\Longrightarrow$ *pc* $\neq$ *pc'*
and *match-ex-table-compxEs2-not-same*:*match-ex-table P C pc* (*compxEs2 es n d*) = $\lfloor$(*pc'*, *d'*)$\rfloor$ $\Longrightarrow$ *pc* $\neq$ *pc'*
apply(*induct e n d* **and** *es n d rule*: *compxE2-compxEs2-induct*)
apply(*auto simp add*: *match-ex-table-append match-ex-entry simp del*: *compxE2-size-convs compxEs2-size-convs compxE2-stack-xlift-convs compxEs2-stack-xlift-convs split*: *if-split-asm*)
done

end

7.11 Type rules for the intermediate language

theory *J1WellType* **imports**
J1State
../Common/ExternalCallWF
../Common/SemiType
begin

declare *Listn.lesub-list-impl-same-size*[*simp del*] *listE-length* [*simp del*]

7.11.1 Well-Typedness

type-synonym
env1 = *ty list* — type environment indexed by variable number

inductive *WT1* :: *'addr J1-prog* $\Rightarrow$ *env1* $\Rightarrow$ *'addr expr1* $\Rightarrow$ *ty* $\Rightarrow$ *bool* (‹-,- $\vdash$1 - :: -› [*51*,*0*,*0*,*51*] *50*)
and *WTs1* :: *'addr J1-prog* $\Rightarrow$ *env1* $\Rightarrow$ *'addr expr1 list* $\Rightarrow$ *ty list* $\Rightarrow$ *bool* (‹-,- $\vdash$1 - [::] -› [*51*,*0*,*0*,*51*]*50*)
for *P* :: *'addr J1-prog*
where

WT1New:
is-class P C $\Longrightarrow$
P,*E* $\vdash$1 *new C* :: *Class C*

| *WT1NewArray*:
⟦ *P*,*E* $\vdash$1 *e* :: *Integer*; *is-type P* (*T*$\lfloor\rceil$) ⟧ $\Longrightarrow$
P,*E* $\vdash$1 *newA T*$\lfloor$*e*$\rceil$ :: *T*$\lfloor\rceil$

| *WT1Cast*:
⟦ *P*,*E* $\vdash$1 *e* :: *T*; *P* $\vdash$ *U* $\leq$ *T* $\vee$ *P* $\vdash$ *T* $\leq$ *U*; *is-type P U* ⟧
$\Longrightarrow$ *P*,*E* $\vdash$1 *Cast U e* :: *U*

| *WT1InstanceOf*:
⟦ *P*,*E* $\vdash$1 *e* :: *T*; *P* $\vdash$ *U* $\leq$ *T* $\vee$ *P* $\vdash$ *T* $\leq$ *U*; *is-type P U*; *is-refT U* ⟧
$\Longrightarrow$ *P*,*E* $\vdash$1 *e instanceof U* :: *Boolean*

| *WT1Val*:
typeof v = *Some T* $\Longrightarrow$

$P,E \vdash_1 Val\ v :: T$

| *WT1Var*:
$[\![E!V = T;\ V < size\ E]\!] \Longrightarrow$
$P,E \vdash_1 Var\ V :: T$

| *WT1BinOp*:
$[\![P,E \vdash_1 e1 :: T1;\ P,E \vdash_1 e2 :: T2;\ P \vdash T1 «bop» T2 :: T]\!]$
$\Longrightarrow P,E \vdash_1 e1 «bop» e2 :: T$

| *WT1LAss*:
$[\![E!i = T;\ \ i < size\ E;\ P,E \vdash_1 e :: T';\ \ P \vdash T' \leq T]\!]$
$\Longrightarrow P,E \vdash_1 i:=e :: Void$

| *WT1AAcc*:
$[\![P,E \vdash_1 a :: T\lfloor\rfloor;\ P,E \vdash_1 i :: Integer]\!]$
$\Longrightarrow P,E \vdash_1 a\lfloor i\rceil :: T$

| *WT1AAss*:
$[\![P,E \vdash_1 a :: T\lfloor\rfloor;\ P,E \vdash_1 i :: Integer;\ P,E \vdash_1 e :: T';\ P \vdash T' \leq T]\!]$
$\Longrightarrow P,E \vdash_1 a\lfloor i\rceil := e :: Void$

| *WT1ALength*:
$P,E \vdash_1 a :: T\lfloor\rfloor \Longrightarrow P,E \vdash_1 a{\cdot}length :: Integer$

| *WTFAcc1*:
$[\![P,E \vdash_1 e :: U;\ class\text{-}type\text{-}of'\ U = \lfloor C \rfloor;\ P \vdash C\ sees\ F{:}T\ (fm)\ in\ D]\!]$
$\Longrightarrow P,E \vdash_1 e{\cdot}F\{D\} :: T$

| *WTFAss1*:
$[\![P,E \vdash_1 e1 :: U;\ class\text{-}type\text{-}of'\ U = \lfloor C \rfloor;\ P \vdash C\ sees\ F{:}T\ (fm)\ in\ D;\ \ P,E \vdash_1 e2 :: T';\ \ P \vdash T' \leq T]\!]$
$\Longrightarrow P,E \vdash_1 e1{\cdot}F\{D\} := e2 :: Void$

| *WTCAS1*:
$[\![P,E \vdash_1 e1 :: U;\ class\text{-}type\text{-}of'\ U = \lfloor C \rfloor;\ P \vdash C\ sees\ F{:}T\ (fm)\ in\ D;\ volatile\ fm;$
$\ \ P,E \vdash_1 e2 :: T';\ P \vdash T' \leq T;\ P,E \vdash_1 e3 :: T'';\ P \vdash T'' \leq T]\!]$
$\Longrightarrow P,E \vdash_1 e1{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3) :: Boolean$

| *WT1Call*:
$[\![P,E \vdash_1 e :: U;\ class\text{-}type\text{-}of'\ U = \lfloor C \rfloor;\ P \vdash C\ sees\ M{:}Ts \to T = m\ in\ D;$
$\ \ P,E \vdash_1 es\ [::]\ Ts';\ P \vdash Ts'\ [\leq]\ Ts]\!]$
$\Longrightarrow P,E \vdash_1 e{\cdot}M(es) :: T$

| *WT1Block*:
$[\![is\text{-}type\ P\ T;\ \ P,E@[T] \vdash_1 e :: T';\ case\ vo\ of\ None \Rightarrow True \mid \lfloor v \rfloor \Rightarrow \exists\, T'.\ typeof\ v = \lfloor T' \rfloor \wedge P \vdash T' \leq T]\!]$
$\Longrightarrow\ P,E \vdash_1 \{V{:}T{=}vo;\ e\} :: T'$

| *WT1Synchronized*:
$[\![P,E \vdash_1 o' :: T;\ is\text{-}refT\ T;\ T \neq NT;\ P,E@[Class\ Object] \vdash_1 e :: T']\!]$
$\Longrightarrow P,E \vdash_1 sync_V\ (o')\ e :: T'$

| *WT1Seq*:

⟦ $P,E \vdash 1\ e_1 :: T_1$; $P,E \vdash 1\ e_2 :: T_2$ ⟧
⟹ $P,E \vdash 1\ e_1;;e_2 :: T_2$

| *WT1Cond*:
⟦ $P,E \vdash 1\ e :: Boolean$; $P,E \vdash 1\ e_1 :: T_1$; $P,E \vdash 1\ e_2 :: T_2$; $P \vdash lub(T_1,T_2) = T$ ⟧
⟹ $P,E \vdash 1\ if\ (e)\ e_1\ else\ e_2 :: T$

| *WT1While*:
⟦ $P,E \vdash 1\ e :: Boolean$; $P,E \vdash 1\ c :: T$ ⟧
⟹ $P,E \vdash 1\ while\ (e)\ c :: Void$

| *WT1Throw*:
⟦ $P,E \vdash 1\ e :: Class\ C$; $P \vdash C \preceq^* Throwable$ ⟧ ⟹
$P,E \vdash 1\ throw\ e :: Void$

| *WT1Try*:
⟦ $P,E \vdash 1\ e_1 :: T$; $P,E@[Class\ C] \vdash 1\ e_2 :: T$; *is-class* $P\ C$ ⟧
⟹ $P,E \vdash 1\ try\ e_1\ catch(C\ V)\ e_2 :: T$

| *WT1Nil*: $P,E \vdash 1\ []\ [::]\ []$

| *WT1Cons*: ⟦ $P,E \vdash 1\ e :: T$; $P,E \vdash 1\ es\ [::]\ Ts$ ⟧ ⟹ $P,E \vdash 1\ e\#es\ [::]\ T\#Ts$

declare *WT1-WTs1.intros*[*intro*!]
declare *WT1Nil*[*iff*]

inductive-cases *WT1-WTs1-cases*[*elim*!]:
$P,E \vdash 1\ Val\ v :: T$
$P,E \vdash 1\ Var\ i :: T$
$P,E \vdash 1\ Cast\ D\ e :: T$
$P,E \vdash 1\ e\ instanceof\ U :: T$
$P,E \vdash 1\ i:=e :: T$
$P,E \vdash 1\ \{i:U=vo;\ e\} :: T$
$P,E \vdash 1\ e1;;e2 :: T$
$P,E \vdash 1\ if\ (e)\ e1\ else\ e2 :: T$
$P,E \vdash 1\ while\ (e)\ c :: T$
$P,E \vdash 1\ throw\ e :: T$
$P,E \vdash 1\ try\ e1\ catch(C\ i)\ e2 :: T$
$P,E \vdash 1\ e{\cdot}F\{D\} :: T$
$P,E \vdash 1\ e1{\cdot}F\{D\}:=e2 :: T$
$P,E \vdash 1\ e{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e'') :: T$
$P,E \vdash 1\ e1\ «bop»\ e2 :: T$
$P,E \vdash 1\ new\ C :: T$
$P,E \vdash 1\ newA\ T'\lfloor e \rceil :: T$
$P,E \vdash 1\ a\lfloor i \rceil := e :: T$
$P,E \vdash 1\ a\lfloor i \rceil :: T$
$P,E \vdash 1\ a{\cdot}length :: T$
$P,E \vdash 1\ e{\cdot}M(es) :: T$
$P,E \vdash 1\ sync_V\ (o')\ e :: T$
$P,E \vdash 1\ insync_V\ (a)\ e :: T$
$P,E \vdash 1\ []\ [::]\ Ts$
$P,E \vdash 1\ e\#es\ [::]\ Ts$

lemma *WTs1-same-size*: $P,E \vdash 1\ es\ [::]\ Ts \Longrightarrow size\ es = size\ Ts$

by (*induct es arbitrary*: *Ts*) *auto*

lemma *WTs1-snoc-cases*:
assumes *wt*: *P*,*E* ⊢*1 es* @ [*e*] [::] *Ts*
obtains *T Ts′* **where** *P*,*E* ⊢*1 es* [::] *Ts′ P*,*E* ⊢*1 e* :: *T*
proof −
from *wt* **have** ∃ *T Ts′*. *P*,*E* ⊢*1 es* [::] *Ts′* ∧ *P*,*E* ⊢*1 e* :: *T*
by(*induct es arbitrary*: *Ts*) *auto*
thus *thesis* **by**(*auto intro*: *that*)
qed

lemma *WTs1-append*:
assumes *wt*: *P*,*Env* ⊢*1 es* @ *es′* [::] *Ts*
obtains *Ts′ Ts″* **where** *P*,*Env* ⊢*1 es* [::] *Ts′ P*,*Env* ⊢*1 es′* [::] *Ts″*
proof −
from *wt* **have** ∃ *Ts′ Ts″*. *P*,*Env* ⊢*1 es* [::] *Ts′* ∧ *P*,*Env* ⊢*1 es′* [::] *Ts″*
by(*induct es arbitrary*: *Ts*) *auto*
thus *?thesis* **by**(*auto intro*: *that*)
qed

lemma *WT1-not-contains-insync*: *P*,*E* ⊢*1 e* :: *T* ⟹ ¬ *contains-insync e*
and *WTs1-not-contains-insyncs*: *P*,*E* ⊢*1 es* [::] *Ts* ⟹ ¬ *contains-insyncs es*
by(*induct rule*: *WT1-WTs1.inducts*) *auto*

lemma *WT1-expr-locks*: *P*,*E* ⊢*1 e* :: *T* ⟹ *expr-locks e* = (λ*a*. *0*)
and *WTs1-expr-lockss*: *P*,*E* ⊢*1 es* [::] *Ts* ⟹ *expr-lockss es* = (λ*a*. *0*)
by(*induct rule*: *WT1-WTs1.inducts*)(*auto*)

lemma assumes *wf*: *wf-prog wfmd P*
shows *WT1-unique*: *P*,*E* ⊢*1 e* :: *T1* ⟹ *P*,*E* ⊢*1 e* :: *T2* ⟹ *T1* = *T2*
and *WTs1-unique*: *P*,*E* ⊢*1 es* [::] *Ts1* ⟹ *P*,*E* ⊢*1 es* [::] *Ts2* ⟹ *Ts1* = *Ts2*
apply(*induct arbitrary*: *T2* **and** *Ts2 rule*:*WT1-WTs1.inducts*)
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *clarsimp*
apply *blast*
apply(*blast dest*: *WT-binop-fun*)
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply (*blast dest*:*sees-field-idemp sees-field-fun*)
apply (*blast dest*: *sees-field-fun*)
apply *blast*

apply(*erule WT1-WTs1-cases*)
apply(*simp*)
apply (*blast dest*:*sees-method-idemp sees-method-fun*)

apply *blast*
apply *blast*
apply *blast*

apply(*blast dest*: *is-lub-unique*[*OF wf*])
apply *blast*
apply *blast*
apply *blast*
apply *blast*
apply *blast*
done

lemma assumes *wf*: *wf-prog p P*
  shows *WT1-is-type*: *P*,*E* ⊢*1 e* :: *T* ⟹ *set E* ⊆ *types P* ⟹ *is-type P T*
  and *WTs1-is-type*: *P*,*E* ⊢*1 es* [::] *Ts* ⟹ *set E* ⊆ *types P* ⟹ *set Ts* ⊆ *types P*
apply(*induct rule*:*WT1-WTs1.inducts*)
apply *simp*
apply *simp*
apply *simp*
apply *simp*
apply (*simp add*:*typeof-lit-is-type*)
apply (*fastforce intro*:*nth-mem*)
apply(*simp add*: *WT-binop-is-type*)
apply(*simp*)
apply(*simp del*: *is-type-array add*: *is-type-ArrayD*)
apply(*simp*)
apply(*simp*)
apply (*simp add*:*sees-field-is-type*[*OF* - *wf*])
apply *simp*
apply *simp*
apply(*fastforce dest*!: *sees-wf-mdecl*[*OF wf*] *simp*:*wf-mdecl-def*)
apply(*simp*)
apply(*simp add*: *is-class-Object*[*OF wf*])
apply *simp*
apply(*blast dest*: *is-lub-is-type*[*OF wf*])
apply *simp*
apply *simp*
apply *simp*
apply *simp*
apply(*simp*)
done

lemma *blocks1-WT*:
  ⟦ *P*,*Env* @ *Ts* ⊢*1 body* :: *T*; *set Ts* ⊆ *types P* ⟧ ⟹ *P*,*Env* ⊢*1 blocks1* (*length Env*) *Ts body* :: *T*
proof(*induct n*≡*length Env Ts body arbitrary*: *Env rule*: *blocks1.induct*)
  case *1* **thus** *?case* **by** *simp*
next
  case (*2 T′ Ts e*)
  note *IH* = ‹⋀*Env′*. ⟦*Suc* (*length Env*) = *length Env′*; *P*,*Env′* @ *Ts* ⊢*1 e* :: *T*; *set Ts* ⊆ *types P* ⟧
        ⟹ *P*,*Env′* ⊢*1 blocks1* (*length Env′*) *Ts e* :: *T*›
  from ‹*set* (*T′* # *Ts*) ⊆ *types P*› **have** *set Ts* ⊆ *types P is-type P T′* **by**(*auto*)
  moreover from ‹*P*,*Env* @ *T′* # *Ts* ⊢*1 e* :: *T*› **have** *P*,(*Env* @ [*T′*]) @ *Ts* ⊢*1 e* :: *T* **by** *simp*
  note *IH*[*OF* - *this*]
  ultimately show *?case* **by** *auto*
qed

lemma *WT1-fv*: ⟦ *P*,*E* ⊢*1 e* :: *T*; *B e* (*length E*); *syncvars e* ⟧ ⟹ *fv e* ⊆ {*0*..<*length E*}
  and *WTs1-fvs*: ⟦ *P*,*E* ⊢*1 es* [::] *Ts*; *Bs es* (*length E*); *syncvarss es* ⟧ ⟹ *fvs es* ⊆ {*0*..<*length E*}

proof(*induct rule*: *WT1-WTs1.inducts*)
case (*WT1Synchronized E e1 T e2 T′ V*)
note *IH1* = ‹⟦$\mathcal{B}$ *e1* (*length E*); *syncvars e1*⟧ ⟹ *fv e1* ⊆ {*0..<length E*}›
note *IH2* = ‹⟦$\mathcal{B}$ *e2* (*length* (*E* @ [*Class Object*])); *syncvars e2*⟧ ⟹ *fv e2* ⊆ {*0..<length* (*E* @ [*Class Object*])}›
from ‹$\mathcal{B}$ (*sync*$_V$ (*e1*) *e2*) (*length E*)› **have** [*simp*]: *V* = *length E*
and *B1*: $\mathcal{B}$ *e1* (*length E*) **and** *B2*: $\mathcal{B}$ *e2* (*Suc* (*length E*)) **by** *auto*
from ‹*syncvars* (*sync*$_V$ (*e1*) *e2*)› **have** *sync1*: *syncvars e1* **and** *sync2*: *syncvars e2* **and** *V*: *V* ∉ *fv e2* **by** *auto*
have *fv e2* ⊆ {*0..<length E*}
proof
fix *x*
assume *x*: *x* ∈ *fv e2*
with *V* **have** *x* ≠ *length E* **by** *auto*
moreover from *IH2 B2 sync2* **have** *fv e2* ⊆ {*0..<Suc* (*length E*)} **by** *auto*
with *x* **have** *x* < *Suc* (*length E*) **by** *auto*
ultimately show *x* ∈ {*0..<length E*} **by** *auto*
qed
with *IH1*[*OF B1 sync1*] **show** *?case* **by**(*auto*)
next
case (*WT1Cond E e e1 T1 e2 T2 T*)
thus *?case* **by**(*auto del*: *subsetI*)
qed *fastforce+*

end

7.12 Well-Formedness of Intermediate Language

theory *J1WellForm* **imports**
../J/DefAss
J1WellType
begin

7.12.1 Well-formedness

definition *wf-J1-mdecl* :: *'addr J1-prog* ⇒ *cname* ⇒ *'addr expr1 mdecl* ⇒ *bool*
where
wf-J1-mdecl P C ≡ λ(*M*,*Ts*,*T*,*body*).
(∃ *T′*. *P*,*Class C*#*Ts* ⊢*1 body* :: *T′* ∧ *P* ⊢ *T′* ≤ *T*) ∧
$\mathcal{D}$ *body* ⌊{*..size Ts*}⌋ ∧ $\mathcal{B}$ *body* (*size Ts* + *1*) ∧ *syncvars body*

lemma *wf-J1-mdecl*[*simp*]:
wf-J1-mdecl P C (*M*,*Ts*,*T*,*body*) ≡
((∃ *T′*. *P*,*Class C*#*Ts* ⊢*1 body* :: *T′* ∧ *P* ⊢ *T′* ≤ *T*) ∧
$\mathcal{D}$ *body* ⌊{*..size Ts*}⌋ ∧ $\mathcal{B}$ *body* (*size Ts* + *1*)) ∧ *syncvars body*
by (*simp add*:*wf-J1-mdecl-def*)

abbreviation *wf-J1-prog* :: *'addr J1-prog* ⇒ *bool*
where *wf-J1-prog* == *wf-prog wf-J1-mdecl*

end

7.13 Preservation of Well-Typedness in Stage 2

theory *TypeComp*
imports
Exception-Tables
J1WellForm
../BV/BVSpec
HOL−Library.Prefix-Order
HOL−Library.Sublist
begin

locale *TC0* =
fixes *P* :: *'addr J1-prog* **and** *mxl* :: *nat*
begin

definition *ty* :: *ty list* ⇒ *'addr expr1* ⇒ *ty*
where *ty E e* ≡ *THE T*. *P*,*E* ⊢*1 e* :: *T*

definition ty_l :: *ty list* ⇒ *nat set* ⇒ ty_l
where ty_l *E A'* ≡ *map* (λ*i*. *if i* ∈ *A'* ∧ *i* < *size E then OK*(*E*!*i*) *else Err*) [*0*..<*mxl*]

definition ty_i' :: *ty list* ⇒ *ty list* ⇒ *nat set option* ⇒ ty_i'
where ty_i' *ST E A* ≡ *case A of None* ⇒ *None* | ⌊*A'*⌋ ⇒ *Some*(*ST*, ty_l *E A'*)

definition *after* :: *ty list* ⇒ *nat set option* ⇒ *ty list* ⇒ *'addr expr1* ⇒ ty_i'
where *after E A ST e* ≡ ty_i' (*ty E e* # *ST*) *E* (*A* ⊔ $\mathcal{A}$ *e*)

end

locale *TC1* = *TC0* +
fixes *wfmd*
assumes *wf-prog*: *wf-prog wfmd P*
begin

lemma *ty-def2* [*simp*]: *P*,*E* ⊢*1 e* :: *T* ⟹ *ty E e* = *T*
apply(*unfold ty-def ty-def*)
apply(*blast intro*: *the-equality WT1-unique*[*OF wf-prog*])
done

end

context *TC0* **begin**

lemma ty_i'*-None* [*simp*]: ty_i' *ST E None* = *None*
by(*simp add*:ty_i'*-def*)

lemma ty_l*-app-diff*[*simp*]:
ty_l (*E*@[*T*]) (*A* − {*size E*}) = ty_l *E A*
by(*auto simp add*:ty_l*-def hyperset-defs*)

lemma ty_i'*-app-diff*[*simp*]:
ty_i' *ST* (*E* @ [*T*]) (*A* ⊖ *size E*) = ty_i' *ST E A*
by(*auto simp add*:ty_i'*-def hyperset-defs*)

lemma *ty$_l$-antimono*:
$A \subseteq A' \Longrightarrow P \vdash ty_l\ E\ A'\ [\leq_\top]\ ty_l\ E\ A$
by(*auto simp:ty$_l$-def list-all2-conv-all-nth*)

lemma *ty$_i$'-antimono*:
$A \subseteq A' \Longrightarrow P \vdash ty_i'\ ST\ E\ \lfloor A' \rfloor \leq' ty_i'\ ST\ E\ \lfloor A \rfloor$
by(*auto simp:ty$_i$'-def ty$_l$-def list-all2-conv-all-nth*)

lemma *ty$_l$-env-antimono*:
$P \vdash ty_l\ (E@[T])\ A\ [\leq_\top]\ ty_l\ E\ A$
by(*auto simp:ty$_l$-def list-all2-conv-all-nth*)

lemma *ty$_i$'-env-antimono*:
$P \vdash ty_i'\ ST\ (E@[T])\ A \leq' ty_i'\ ST\ E\ A$
by(*auto simp:ty$_i$'-def ty$_l$-def list-all2-conv-all-nth*)

lemma *ty$_i$'-incr*:
$P \vdash ty_i'\ ST\ (E\ @\ [T])\ \lfloor insert\ (size\ E)\ A \rfloor \leq' ty_i'\ ST\ E\ \lfloor A \rfloor$
by(*auto simp:ty$_i$'-def ty$_l$-def list-all2-conv-all-nth*)

lemma *ty$_l$-incr*:
$P \vdash ty_l\ (E\ @\ [T])\ (insert\ (size\ E)\ A)\ [\leq_\top]\ ty_l\ E\ A$
by(*auto simp*: *hyperset-defs ty$_l$-def list-all2-conv-all-nth*)

lemma *ty$_l$-in-types*:
$set\ E \subseteq types\ P \Longrightarrow ty_l\ E\ A \in list\ mxl\ (err\ (types\ P))$
by(*auto simp add:ty$_l$-def intro*!:*listI dest*!: *nth-mem*)

function *compT* :: *ty list* ⇒ *nat hyperset* ⇒ *ty list* ⇒ *'addr expr1* ⇒ *ty$_i$' list*
and *compTs* :: *ty list* ⇒ *nat hyperset* ⇒ *ty list* ⇒ *'addr expr1 list* ⇒ *ty$_i$' list*
where
$compT\ E\ A\ ST\ (new\ C) = []$
| $compT\ E\ A\ ST\ (newA\ T\lfloor e \rceil) = compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e]$
| $compT\ E\ A\ ST\ (Cast\ C\ e) = compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e]$
| $compT\ E\ A\ ST\ (e\ instanceof\ T) = compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e]$
| $compT\ E\ A\ ST\ (Val\ v) = []$
| $compT\ E\ A\ ST\ (e1\ «bop»\ e2) =$
$(let\ ST1 = ty\ E\ e1\#ST;\ A1 = A \sqcup \mathcal{A}\ e1\ in$
$compT\ E\ A\ ST\ e1\ @\ [after\ E\ A\ ST\ e1]\ @$
$compT\ E\ A1\ ST1\ e2\ @\ [after\ E\ A1\ ST1\ e2])$
| $compT\ E\ A\ ST\ (Var\ i) = []$
| $compT\ E\ A\ ST\ (i := e) = compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e,\ ty_i'\ ST\ E\ (A \sqcup \mathcal{A}\ e \sqcup \lfloor \{i\} \rfloor)]$
| $compT\ E\ A\ ST\ (a\lfloor i \rceil) =$
$(let\ ST1 = ty\ E\ a\ \#\ ST;\ A1 = A \sqcup \mathcal{A}\ a$
$in\ \ compT\ E\ A\ ST\ a\ @\ [after\ E\ A\ ST\ a]\ @\ compT\ E\ A1\ ST1\ i\ @\ [after\ E\ A1\ ST1\ i])$
| $compT\ E\ A\ ST\ (a\lfloor i \rceil := e) =$
$(let\ ST1 = ty\ E\ a\ \#\ ST;\ A1 = A \sqcup \mathcal{A}\ a;$
$ST2 = ty\ E\ i\ \#\ ST1;\ A2 = A1 \sqcup \mathcal{A}\ i;\ A3 = A2 \sqcup \mathcal{A}\ e$

```
   in compT E A ST a @ [after E A ST a] @ compT E A1 ST1 i @ [after E A1 ST1 i] @ compT E
A2 ST2 e @ [after E A2 ST2 e, ty_i' ST E A3])
| compT E A ST (a·length) = compT E A ST a @ [after E A ST a]
| compT E A ST (e·F{D}) = compT E A ST e @ [after E A ST e]
| compT E A ST (e1·F{D} := e2) =
  (let ST1 = ty E e1#ST; A1 = A ⊔ 𝒜 e1; A2 = A1 ⊔ 𝒜 e2
   in  compT E A ST e1 @ [after E A ST e1] @ compT E A1 ST1 e2 @ [after E A1 ST1 e2] @ [ty_i'
ST E A2])
| compT E A ST (e1·compareAndSwap(D·F, e2, e3)) =
  (let ST1 = ty E e1 # ST; A1 = A ⊔ 𝒜 e1; ST2 = ty E e2 # ST1; A2 = A1 ⊔ 𝒜 e2; A3 = A2
⊔ 𝒜 e3
   in  compT E A ST e1 @ [after E A ST e1] @ compT E A1 ST1 e2 @ [after E A1 ST1 e2] @ compT
E A2 ST2 e3 @ [after E A2 ST2 e3])
| compT E A ST (e·M(es)) =
   compT E A ST e @ [after E A ST e] @
   compTs E (A ⊔ 𝒜 e) (ty E e # ST) es
| compT E A ST {i:T=None; e} = compT (E@[T]) (A⊖i) ST e
| compT E A ST {i:T=⌊v⌋; e} =
   [after E A ST (Val v), ty_i' ST (E@[T]) (A ⊔ ⌊{i}⌋)] @ compT (E@[T]) (A ⊔ ⌊{i}⌋) ST e

| compT E A ST (sync_i (e1) e2) =
  (let A1 = A ⊔ 𝒜 e1 ⊔ ⌊{i}⌋; E1 = E @ [Class Object]; ST2 = ty E1 e2 # ST; A2 = A1 ⊔ 𝒜 e2
   in  compT E A ST e1 @
       [after E A ST e1,
        ty_i' (Class Object # Class Object # ST) E (A ⊔ 𝒜 e1),
        ty_i' (Class Object # ST) E1 A1,
        ty_i' ST E1 A1] @
       compT E1 A1 ST e2 @
       [ty_i' ST2 E1 A2, ty_i' (Class Object # ST2) E1 A2, ty_i' ST2 E1 A2,
        ty_i' (Class Throwable # ST) E1 A1,
        ty_i' (Class Object # Class Throwable # ST) E1 A1,
        ty_i' (Class Throwable # ST) E1 A1])
| compT E A ST (insync_i (a) e) = []

| compT E A ST (e1;;e2) =
  (let A1 = A ⊔ 𝒜 e1 in
   compT E A ST e1 @ [after E A ST e1, ty_i' ST E A1] @
   compT E A1 ST e2)
| compT E A ST (if (e) e1 else e2) =
   (let A0 = A ⊔ 𝒜 e; τ = ty_i' ST E A0 in
    compT E A ST e @ [after E A ST e, τ] @
    compT E A0 ST e1 @ [after E A0 ST e1, τ] @
    compT E A0 ST e2)
| compT E A ST (while (e) c) =
   (let A0 = A ⊔ 𝒜 e;  A1 = A0 ⊔ 𝒜 c; τ = ty_i' ST E A0 in
    compT E A ST e @ [after E A ST e, τ] @
    compT E A0 ST c @ [after E A0 ST c, ty_i' ST E A1, ty_i' ST E A0])
| compT E A ST (throw e) = compT E A ST e @ [after E A ST e]
| compT E A ST (try e1 catch(C i) e2) =
   compT E A ST e1 @ [after E A ST e1] @
   [ty_i' (Class C#ST) E A, ty_i' ST (E@[Class C]) (A ⊔ ⌊{i}⌋)] @
   compT (E@[Class C]) (A ⊔ ⌊{i}⌋) ST e2

| compTs E A ST [] = []
```

| *compTs E A ST* (*e*#*es*) = *compT E A ST e* @ [*after E A ST e*] @
compTs E (*A* ⊔ (*A* *e*)) (*ty E e* # *ST*) *es*
by *pat-completeness simp-all*
termination
apply(*relation case-sum* (λ*p*. *size* (*snd* (*snd* (*snd p*)))) (λ*p*. *size-list size* (*snd* (*snd* (*snd p*)))) <∗*mlex*∗>
{})
apply(*rule wf-mlex*[*OF wf-empty*])
apply(*rule mlex-less*, *simp*)+
done

lemmas *compT-compTs-induct* =
compT-compTs.induct[
unfolded meta-all5-eq-conv meta-all4-eq-conv meta-all3-eq-conv meta-all2-eq-conv meta-all-eq-conv,
case-names
new NewArray Cast InstanceOf Val BinOp Var LAss AAcc AAss ALen FAcc FAss CompareAndSwap
Call BlockNone BlockSome
Synchronized InSynchronized Seq Cond While throw TryCatch
Nil Cons]

definition *compTa* :: *ty list* ⇒ *nat hyperset* ⇒ *ty list* ⇒ *'addr expr1* ⇒ *ty*$_i$' *list*
where *compTa E A ST e* ≡ *compT E A ST e* @ [*after E A ST e*]

lemmas *compE2-not-Nil* = *compE2-neq-Nil*
declare *compE2-not-Nil*[*simp*]

lemma *compT-sizes*[*simp*]:
shows *size*(*compT E A ST e*) = *size*(*compE2 e*) − *1*
and *size*(*compTs E A ST es*) = *size*(*compEs2 es*)
apply(*induct E A ST e* **and** *E A ST es rule*: *compT-compTs-induct*)
apply(*auto split*:*nat-diff-split*)
done

lemma *compT-None-not-Some* [*simp*]: ⌊*τ*⌋ ∉ *set* (*compT E None ST e*)
and *compTs-None-not-Some* [*simp*]: ⌊*τ*⌋ ∉ *set* (*compTs E None ST es*)
by(*induct E A*≡*None* :: *nat hyperset ST e* **and** *E A*≡*None* :: *nat hyperset ST es rule*: *compT-compTs-induct*)
(*simp-all add*:*after-def*)

lemma *pair-eq-ty*$_i$'*-conv*:
(⌊(*ST*, *LT*)⌋ = *ty*$_i$' *ST*$_0$ *E A*) = (*case A of None* ⇒ *False* | *Some A* ⇒ (*ST* = *ST*$_0$ ∧ *LT* = *ty*$_l$ *E*
A))
by(*simp add*:*ty*$_i$'*-def*)

lemma *pair-conv-ty*$_i$': ⌊(*ST*, *ty*$_l$ *E A*)⌋ = *ty*$_i$' *ST E* ⌊*A*⌋
by(*simp add*:*ty*$_i$'*-def*)

lemma *ty*$_i$'*-antimono2*:
⟦ *E* ≤ *E'*; *A* ⊆ *A'* ⟧ ⟹ *P* ⊢ *ty*$_i$' *ST E'* ⌊*A'*⌋ ≤' *ty*$_i$' *ST E* ⌊*A*⌋
by(*auto simp*:*ty*$_i$'*-def ty*$_l$*-def list-all2-conv-all-nth less-eq-list-def prefix-def*)

declare *ty*$_i$'*-antimono* [*intro*!] *after-def*[*simp*] *pair-conv-ty*$_i$'[*simp*] *pair-eq-ty*$_i$'*-conv*[*simp*]

lemma *compT-LT-prefix*:
⟦ ⌊(*ST*,*LT*)⌋ ∈ *set*(*compT E A ST0 e*); *B e* (*size E*) ⟧ ⟹ *P* ⊢ ⌊(*ST*,*LT*)⌋ ≤' *ty*$_i$' *ST E A*
and *compTs-LT-prefix*:

⟦ ⌊(ST,LT)⌋ ∈ set(compTs E A ST0 es); Bs es (size E) ⟧ ⟹ P ⊢ ⌊(ST,LT)⌋ ≤' ty_i' ST E A
proof(*induct E A ST0 e* **and** *E A ST0 es rule*: *compT-compTs-induct*)
case *FAss* **thus** *?case* **by**(*fastforce simp*:*hyperset-defs elim*!:*sup-state-opt-trans*)
next
case *BinOp* **thus** *?case*
by(*fastforce simp*:*hyperset-defs elim*!:*sup-state-opt-trans split*:*bop.splits*)
next
case *Seq* **thus** *?case* **by**(*fastforce simp*:*hyperset-defs elim*!:*sup-state-opt-trans*)
next
case *While* **thus** *?case* **by**(*fastforce simp*:*hyperset-defs elim*!:*sup-state-opt-trans*)
next
case *Cond* **thus** *?case* **by**(*fastforce simp*:*hyperset-defs elim*!:*sup-state-opt-trans*)
next
case *BlockNone* **thus** *?case* **by**(*auto*)
next
case *BlockSome* **thus** *?case*
by(*clarsimp simp only*: ty_i'-*def*)(*fastforce intro*: ty_i'-*incr simp add*: *hyperset-defs elim*: *sup-state-opt-trans*)
next
case *Call* **thus** *?case* **by**(*fastforce simp*:*hyperset-defs elim*!:*sup-state-opt-trans*)
next
case *Cons* **thus** *?case*
by(*fastforce simp*:*hyperset-defs elim*!:*sup-state-opt-trans*)
next
case *TryCatch* **thus** *?case*
by(*fastforce simp*:*hyperset-defs intro*!: ty_i'-*incr elim*!:*sup-state-opt-trans*)
next
case *NewArray* **thus** *?case* **by**(*auto simp add*: *hyperset-defs*)
next
case *AAcc* **thus** *?case* **by**(*fastforce simp*:*hyperset-defs elim*!:*sup-state-opt-trans*)
next
case *AAss* **thus** *?case* **by**(*auto simp*:*hyperset-defs Un-ac elim*!:*sup-state-opt-trans*)
next
case *ALen* **thus** *?case* **by**(*auto simp add*: *hyperset-defs*)
next
case *CompareAndSwap* **thus** *?case* **by**(*auto simp*: *hyperset-defs Un-ac elim*!:*sup-state-opt-trans*)
next
case *Synchronized* **thus** *?case*
by(*fastforce simp add*: *hyperset-defs elim*: *sup-state-opt-trans intro*: *sup-state-opt-trans*[*OF* ty_i'-*incr*]
ty_i'-*antimono2*)
qed (*auto simp*:*hyperset-defs*)

declare ty_i'-*antimono* [*rule del*] *after-def*[*simp del*] *pair-conv-*ty_i'[*simp del*] *pair-eq-*ty_i'-*conv*[*simp del*]

lemma *OK-None-states* [*iff*]: *OK None* ∈ *states P mxs mxl*
by(*simp add*: *JVM-states-unfold*)

end

context *TC1* **begin**

lemma *after-in-states*:
⟦ *P*,*E* ⊢*1 e* :: *T*; *set E* ⊆ *types P*; *set ST* ⊆ *types P*; *size ST* + *max-stack e* ≤ *mxs* ⟧
⟹ *OK* (*after E A ST e*) ∈ *states P mxs mxl*
apply(*subgoal-tac size ST* + *1* ≤ *mxs*)

apply(*simp add*:*after-def* ty_i'*-def JVM-states-unfold* ty_l*-in-types*)
apply(*clarify intro*!: *exI*)
apply(*rule conjI*)
apply(*rule exI*[**where** *x*=*length ST + 1*], *fastforce*)
apply(*clarsimp*)
apply(*rule conjI*[*OF WT1-is-type*[*OF wf-prog*]], *auto intro*: *listI*)
using *max-stack1*[*of e*] **by** *simp*

end

context *TC0* **begin**

lemma *OK-*ty_i'*-in-statesI* [*simp*]:
⟦ *set E* ⊆ *types P*; *set ST* ⊆ *types P*; *size ST* ≤ *mxs* ⟧
⟹ *OK* (ty_i' *ST E A*) ∈ *states P mxs mxl*
apply(*simp add*:ty_i'*-def JVM-states-unfold* ty_l*-in-types*)
apply(*blast intro*!:*listI*)
done

end

lemma *is-class-type-aux*: *is-class P C* ⟹ *is-type P* (*Class C*)
by(*simp*)

context *TC1* **begin**

declare *is-type.simps*[*simp del*] *subsetI*[*rule del*]

theorem
shows *compT-states*:
⟦ *P*,*E* ⊢*1 e* :: *T*; *set E* ⊆ *types P*; *set ST* ⊆ *types P*;
size ST + max-stack e ≤ *mxs*; *size E + max-vars e* ≤ *mxl* ⟧
⟹ *OK* ‘ *set*(*compT E A ST e*) ⊆ *states P mxs mxl*
(**is** *PROP ?P e E T A ST*)

and *compTs-states*:
⟦ *P*,*E* ⊢*1 es*[::] *Ts*; *set E* ⊆ *types P*; *set ST* ⊆ *types P*;
size ST + max-stacks es ≤ *mxs*; *size E + max-varss es* ≤ *mxl* ⟧
⟹ *OK* ‘ *set*(*compTs E A ST es*) ⊆ *states P mxs mxl*
(**is** *PROP ?Ps es E Ts A ST*)
proof(*induct E A ST e* **and** *E A ST es arbitrary*: *T* **and** *Ts rule*: *compT-compTs-induct*)
case *new* **thus** *?case* **by**(*simp*)
next
case (*Cast C e*) **thus** *?case* **by** (*auto simp*:*after-in-states*)
next
case *InstanceOf* **thus** *?case* **by** (*auto simp*:*after-in-states*)
next
case *Val* **thus** *?case* **by**(*simp*)
next
case *Var* **thus** *?case* **by**(*simp*)
next
case *LAss* **thus** *?case* **by**(*auto simp*:*after-in-states*)
next
case *FAcc* **thus** *?case* **by**(*auto simp*:*after-in-states*)

next
case *FAss* **thus** *?case*
by(*auto simp*:*image-Un WT1-is-type*[*OF wf-prog*] *after-in-states*)
next
case *CompareAndSwap* **thus** *?case* **by**(*auto simp*:*image-Un WT1-is-type*[*OF wf-prog*] *after-in-states*)
next
case *Seq* **thus** *?case*
by(*auto simp*:*image-Un after-in-states*)
next
case *BinOp* **thus** *?case*
by(*auto simp*:*image-Un WT1-is-type*[*OF wf-prog*] *after-in-states*)
next
case *Cond* **thus** *?case*
by(*force simp*:*image-Un WT1-is-type*[*OF wf-prog*] *after-in-states*)
next
case *While* **thus** *?case*
by(*auto simp*:*image-Un WT1-is-type*[*OF wf-prog*] *after-in-states*)
next
case *BlockNone* **thus** *?case* **by** *auto*
next
case (*BlockSome E A ST i ty v exp*)
with *max-stack1*[*of exp*] **show** *?case* **by**(*auto intro*: *after-in-states*)
next
case (*TryCatch E A ST* e_1 *C i* e_2)
moreover have *size ST* + *1* ≤ *mxs* **using** *TryCatch.prems max-stack1*[*of* e_1] **by** *auto*
ultimately show *?case*
by(*auto simp*:*image-Un WT1-is-type*[*OF wf-prog*] *after-in-states*
is-class-type-aux)
next
case *Nil* **thus** *?case* **by** *simp*
next
case *Cons* **thus** *?case*
by(*auto simp*:*image-Un WT1-is-type*[*OF wf-prog*] *after-in-states*)
next
case *throw* **thus** *?case*
by(*auto simp*: *WT1-is-type*[*OF wf-prog*] *after-in-states*)
next
case *Call* **thus** *?case*
by(*auto simp*:*image-Un WT1-is-type*[*OF wf-prog*] *after-in-states*)
next
case *NewArray* **thus** *?case*
by(*auto simp*:*image-Un WT1-is-type*[*OF wf-prog*] *after-in-states*)
next
case *AAcc* **thus** *?case* **by**(*auto simp*:*image-Un WT1-is-type*[*OF wf-prog*] *after-in-states*)
next
case *AAss* **thus** *?case* **by**(*auto simp*:*image-Un WT1-is-type*[*OF wf-prog*] *after-in-states*)
next
case *ALen* **thus** *?case* **by**(*auto simp*:*image-Un WT1-is-type*[*OF wf-prog*] *after-in-states*)
next
case *InSynchronized* **thus** *?case* **by** *auto*
next
case (*Synchronized E A ST i exp1 exp2*)
from ‹*P*,*E* ⊢*1* $sync_i$ (*exp1*) *exp2* :: *T*› **obtain** *T1*
where *wt1*: *P*,*E* ⊢*1 exp1* :: *T1* **and** *T1*: *is-refT T1 T1* ≠ *NT*

and *wt2*: *P*,*E*@[*Class Object*] ⊢*1 exp2* :: *T* **by** *auto*
moreover note *E* = ‹*set E* ⊆ *types P*› **with** *wf-prog*
have *E'*: *set* (*E*@[*Class Object*]) ⊆ *types P* **by**(*auto simp add*: *is-type.simps*)
moreover from *wf-prog wt2 E'* **have** *T*: *is-type P T* **by**(*rule WT1-is-type*)
note *ST* = ‹*set ST* ⊆ *types P*› **with** *wf-prog*
have *ST'*: *set* (*Class Object* # *ST*) ⊆ *types P* **by**(*auto simp add*: *is-type.simps*)
moreover from *wf-prog* **have** *throwable*: *is-type P* (*Class Throwable*)
unfolding *is-type.simps* **by**(*rule is-class-Throwable*)
ultimately show *?case* **using** *Synchronized max-stack1*[*of exp2*] *T*
by(*auto simp add*: *image-Un after-in-states*)
qed

declare *is-type.simps*[*simp*] *subsetI*[*intro*!]

end

locale *TC2* = *TC0* +
fixes T_r :: *ty* **and** *mxs* :: *pc*
begin

definition
wt-instrs :: *'addr instr list* ⇒ *ex-table* ⇒ ty_i' *list* ⇒ *bool* (‹(⊢ -, - /[::]/ -)› [*0*,*0*,*51*] *50*)
where
⊢ *is*,*xt* [::] *τs* ≡ *size is* < *size τs* ∧ *pcs xt* ⊆ {*0*..<*size is*} ∧ (∀ *pc*< *size is*. *P*,T_r,*mxs*,*size τs*,*xt* ⊢ *is*!*pc*,*pc* :: *τs*)

lemmas *wt-defs* = *wt-instrs-def wt-instr-def app-def eff-def norm-eff-def*

lemma *wt-instrs-Nil* [*simp*]: *τs* ≠ [] ⟹ ⊢ [],[] [::] *τs*
by(*simp add*:*wt-defs*)

end

locale *TC3* = *TC1* + *TC2*

lemma *eff-None* [*simp*]: *eff i P pc et None* = []
by (*simp add*: *Effect.eff-def*)

declare *split-comp-eq*[*simp del*]

lemma *wt-instr-appR*:
⟦ *P*,*T*,*m*,*mpc*,*xt* ⊢ *is*!*pc*,*pc* :: *τs*;
pc < *size is*; *size is* < *size τs*; *mpc* ≤ *size τs*; *mpc* ≤ *mpc'* ⟧
⟹ *P*,*T*,*m*,*mpc'*,*xt* ⊢ *is*!*pc*,*pc* :: *τs*@*τs'*
by (*fastforce simp*:*wt-instr-def app-def*)

lemma *relevant-entries-shift* [*simp*]:
relevant-entries P i (*pc*+*n*) (*shift n xt*) = *shift n* (*relevant-entries P i pc xt*)
apply (*induct xt*)
apply (*unfold relevant-entries-def shift-def*)
apply *simp*
apply (*auto simp add*: *is-relevant-entry-def*)

done

lemma *xcpt-eff-shift* [*simp*]:
xcpt-eff i P (*pc*+*n*) *τ* (*shift n xt*) =
map (*λ*(*pc*,*τ*). (*pc* + *n*, *τ*)) (*xcpt-eff i P pc τ xt*)
apply(*simp add*: *xcpt-eff-def*)
apply(*cases τ*)
apply(*auto simp add*: *shift-def*)
done

lemma *eff-shift* [*simp*]:
app$_i$ (*i*, *P*, *pc*, *m*, *T*, *τ*) ⟹
eff i P (*pc*+*n*) (*shift n xt*) (*Some τ*) =
map (*λ*(*pc*,*τ*). (*pc*+*n*,*τ*)) (*eff i P pc xt* (*Some τ*))
apply(*simp add*:*eff-def norm-eff-def*)
apply(*cases i*,*auto*)
done

lemma *xcpt-app-shift* [*simp*]:
xcpt-app i P (*pc*+*n*) *m* (*shift n xt*) *τ* = *xcpt-app i P pc m xt τ*
by (*simp add*: *xcpt-app-def*) (*auto simp add*: *shift-def*)

lemma *wt-instr-appL*:
⟦ *P*,*T*,*m*,*mpc*,*xt* ⊢ *i*,*pc* :: *τs*; *pc* < *size τs*; *mpc* ≤ *size τs* ⟧
⟹ *P*,*T*,*m*,*mpc* + *size τs*′,*shift* (*size τs*′) *xt* ⊢ *i*,*pc*+*size τs*′ :: *τs*′@*τs*
apply(*clarsimp simp add*: *wt-instr-def app-def*)
apply(*auto*)
apply(*cases i*, *auto*)
done

lemma *wt-instr-Cons*:
⟦ *P*,*T*,*m*,*mpc* − *1*,[] ⊢ *i*,*pc* − *1* :: *τs*;
0 < *pc*; *0* < *mpc*; *pc* < *size τs* + *1*; *mpc* ≤ *size τs* + *1* ⟧
⟹ *P*,*T*,*m*,*mpc*,[] ⊢ *i*,*pc* :: *τ*#*τs*
apply(*drule wt-instr-appL*[**where** *τs*′ = [*τ*]])
apply *arith*
apply *arith*
apply (*simp split*:*nat-diff-split-asm*)
done

lemma *wt-instr-append*:
⟦ *P*,*T*,*m*,*mpc* − *size τs*′,[] ⊢ *i*,*pc* − *size τs*′ :: *τs*;
size τs′ ≤ *pc*; *size τs*′ ≤ *mpc*; *pc* < *size τs* + *size τs*′; *mpc* ≤ *size τs* + *size τs*′ ⟧
⟹ *P*,*T*,*m*,*mpc*,[] ⊢ *i*,*pc* :: *τs*′@*τs*
apply(*drule wt-instr-appL*[**where** *τs*′ = *τs*′])
apply *arith*
apply *arith*

apply (*simp split*:*nat-diff-split-asm*)
done

lemma *xcpt-app-pcs*:
pc ∉ *pcs xt* ⟹ *xcpt-app i P pc mxs xt τ*
by (*auto simp add*: *xcpt-app-def relevant-entries-def is-relevant-entry-def pcs-def*)

lemma *xcpt-eff-pcs*:
pc ∉ *pcs xt* ⟹ *xcpt-eff i P pc τ xt* = []
by (*cases τ*)
(*auto simp add*: *is-relevant-entry-def xcpt-eff-def relevant-entries-def pcs-def*
intro!: *filter-False*)

lemma *pcs-shift*:
pc < *n* ⟹ *pc* ∉ *pcs* (*shift n xt*)
by (*auto simp add*: *shift-def pcs-def*)

lemma *xcpt-eff-shift-pc-ge-n*: **assumes** *x* ∈ *set* (*xcpt-eff i P pc τ* (*shift n xt*))
shows *n* ≤ *pc*
proof −
{ assume *pc* < *n*
hence *pc* ∉ *pcs* (*shift n xt*) **by**(*rule pcs-shift*)
with *assms* **have** *False*
by(*auto simp add*: *pcs-def xcpt-eff-def is-relevant-entry-def relevant-entries-def split-beta cong*: *filter-cong*) **}**
thus *?thesis* **by**(*cases n* ≤ *pc*)(*auto*)
qed

lemma *wt-instr-appRx*:
⟦ *P*,*T*,*m*,*mpc*,*xt* ⊢ *is*!*pc*,*pc* :: *τs*; *pc* < *size is*; *size is* < *size τs*; *mpc* ≤ *size τs* ⟧
⟹ *P*,*T*,*m*,*mpc*,*xt* @ *shift* (*size is*) *xt*′ ⊢ *is*!*pc*,*pc* :: *τs*
apply(*clarsimp simp*:*wt-instr-def eff-def app-def*)
apply(*fastforce dest*: *xcpt-eff-shift-pc-ge-n intro*!: *xcpt-app-pcs*[*OF pcs-shift*])
done

lemma *wt-instr-appLx*:
⟦ *P*,*T*,*m*,*mpc*,*xt* ⊢ *i*,*pc* :: *τs*; *pc* ∉ *pcs xt*′ ⟧
⟹ *P*,*T*,*m*,*mpc*,*xt*′@*xt* ⊢ *i*,*pc* :: *τs*
by (*auto simp*:*wt-instr-def app-def eff-def xcpt-app-pcs xcpt-eff-pcs*)

context *TC2* **begin**

lemma *wt-instrs-extR*:
⊢ *is*,*xt* [::] *τs* ⟹ ⊢ *is*,*xt* [::] *τs* @ *τs*′
by(*auto simp add*:*wt-instrs-def wt-instr-appR*)

lemma *wt-instrs-ext*:
⟦ ⊢ *is*$_1$,*xt*$_1$ [::] *τs*$_1$@*τs*$_2$; ⊢ *is*$_2$,*xt*$_2$ [::] *τs*$_2$; *size τs*$_1$ = *size is*$_1$ ⟧
⟹ ⊢ *is*$_1$@*is*$_2$, *xt*$_1$ @ *shift* (*size is*$_1$) *xt*$_2$ [::] *τs*$_1$@*τs*$_2$

apply(*clarsimp simp*:*wt-instrs-def*)
apply(*rule conjI*, *fastforce*)
apply(*rule conjI*, *fastforce simp add*: *pcs-shift-conv*)
apply *clarsimp*
apply(*rule conjI*, *fastforce simp*:*wt-instr-appRx*)
apply *clarsimp*
apply(*erule-tac* $x = pc - size\ is_1$ **in** *allE*)+
apply(*thin-tac* $P \longrightarrow Q$ **for** $P\ Q$)
apply(*erule impE*, *arith*)
apply(*drule-tac* $\tau s' = \tau s_1$ **in** *wt-instr-appL*)
apply *arith*
apply *simp*
apply(*fastforce simp add*:*add.commute intro*!: *wt-instr-appLx*)
done

corollary *wt-instrs-ext2*:
$[\![\vdash is_2,xt_2\ [::]\ \tau s_2; \vdash is_1,xt_1\ [::]\ \tau s_1@\tau s_2;\ size\ \tau s_1 = size\ is_1\ ]\!]$
$\Longrightarrow \vdash is_1@is_2,\ xt_1\ @\ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2$
by(*rule wt-instrs-ext*)

corollary *wt-instrs-ext-prefix* [*trans*]:
$[\![\vdash is_1,xt_1\ [::]\ \tau s_1@\tau s_2; \vdash is_2,xt_2\ [::]\ \tau s_3;$
$size\ \tau s_1 = size\ is_1;\ \tau s_3 \leq \tau s_2\ ]\!]$
$\Longrightarrow \vdash is_1@is_2,\ xt_1\ @\ shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2$
by(*bestsimp simp*:*less-eq-list-def prefix-def elim*: *wt-instrs-ext dest*:*wt-instrs-extR*)

corollary *wt-instrs-app*:
assumes is_1: $\vdash is_1,xt_1\ [::]\ \tau s_1@[\tau]$
assumes is_2: $\vdash is_2,xt_2\ [::]\ \tau\#\tau s_2$
assumes *s*: $size\ \tau s_1 = size\ is_1$
shows $\vdash is_1@is_2,\ xt_1@shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau\#\tau s_2$
proof −
from is_1 **have** $\vdash is_1,xt_1\ [::]\ (\tau s_1@[\tau])@\tau s_2$
by (*rule wt-instrs-extR*)
hence $\vdash is_1,xt_1\ [::]\ \tau s_1@\tau\#\tau s_2$ **by** *simp*
from *this* is_2 *s* **show** *?thesis* **by** (*rule wt-instrs-ext*)
qed

corollary *wt-instrs-app-last*[*trans*]:
$[\![\vdash is_2,xt_2\ [::]\ \tau\#\tau s_2; \vdash is_1,xt_1\ [::]\ \tau s_1;$
$last\ \tau s_1 = \tau;\ \ size\ \tau s_1 = size\ is_1+1\ ]\!]$
$\Longrightarrow \vdash is_1@is_2,\ xt_1@shift\ (size\ is_1)\ xt_2\ [::]\ \tau s_1@\tau s_2$
apply(*cases* τs_1 *rule*:*rev-cases*)
apply *simp*
apply(*simp add*:*wt-instrs-app*)
done

corollary *wt-instrs-append-last*[*trans*]:
$[\![\vdash is,xt\ [::]\ \tau s;\ P,T_r,mxs,mpc,[] \vdash i,pc :: \tau s;$

```
    pc = size is; mpc = size τs; size is + 1 < size τs ⟧
  ⟹ ⊢ is@[i],xt [::] τs
apply(clarsimp simp add:wt-instrs-def)
apply(rule conjI, fastforce)
apply(fastforce intro!:wt-instr-appLx[where xt = [],simplified]
            dest!:less-antisym)
done


corollary wt-instrs-app2:
  ⟦ ⊢ (is₂ :: 'b instr list),xt₂ [::] τ'#τs₂;  ⊢ is₁,xt₁ [::] τ#τs₁@[τ'];
    xt' = xt₁ @ shift (size is₁) xt₂;  size τs₁+1 = size is₁ ⟧
  ⟹ ⊢ is₁@is₂,xt' [::] τ#τs₁@τ'#τs₂
using wt-instrs-app[where ?τs₁.0 = τ # τs₁ and ?'b = 'b] by simp


corollary wt-instrs-app2-simp[trans,simp]:
  ⟦ ⊢ (is₂ :: 'b instr list),xt₂ [::] τ'#τs₂;  ⊢ is₁,xt₁ [::] τ#τs₁@[τ']; size τs₁+1 = size is₁ ⟧
  ⟹ ⊢ is₁@is₂, xt₁@shift (size is₁) xt₂ [::] τ#τs₁@τ'#τs₂
using wt-instrs-app[where ?τs₁.0 = τ # τs₁ and ?'b = 'b] by simp


corollary wt-instrs-Cons[simp]:
  ⟦ τs ≠ []; ⊢ [i],[] [::] [τ,τ']; ⊢ is,xt [::] τ'#τs ⟧
  ⟹ ⊢ i#is,shift 1 xt [::] τ#τ'#τs
using wt-instrs-app2[where ?is₁.0 = [i] and ?τs₁.0 = [] and ?is₂.0 = is
                     and ?xt₁.0 = []]
by simp


corollary wt-instrs-Cons2[trans]:
  assumes τs: ⊢ is,xt [::] τs
  assumes i: P,T_r,mxs,mpc,[] ⊢ i,0 :: τ#τs
  assumes mpc: mpc = size τs + 1
  shows ⊢ i#is,shift 1 xt [::] τ#τs
proof −
  from τs have τs ≠ [] by (auto simp: wt-instrs-def)
  with mpc i have ⊢ [i],[] [::] [τ]@τs by (simp add: wt-instrs-def)
  with τs show ?thesis by (fastforce dest: wt-instrs-ext)
qed


lemma wt-instrs-last-incr[trans]:
  ⟦ ⊢ is,xt [::] τs@[τ]; P ⊢ τ ≤' τ' ⟧ ⟹ ⊢ is,xt [::] τs@[τ']
apply(clarsimp simp add:wt-instrs-def wt-instr-def)
apply(rule conjI)
apply(fastforce)
apply(clarsimp)
apply(rename-tac pc' tau')
apply(erule allE, erule (1) impE)
apply(clarsimp)
apply(drule (1) bspec)
apply(clarsimp)
apply(subgoal-tac pc' = size τs)
```

prefer *2*
apply(*clarsimp simp*:*app-def*)
apply(*drule* (*1*) *bspec*)
apply(*clarsimp*)
apply(*auto elim*!:*sup-state-opt-trans*)
done

end

lemma [*iff*]: *xcpt-app i P pc mxs* [] τ
by (*simp add*: *xcpt-app-def relevant-entries-def*)

lemma [*simp*]: *xcpt-eff i P pc* τ [] = []
by (*simp add*: *xcpt-eff-def relevant-entries-def*)

context *TC2* **begin**

lemma *wt-New*:
⟦ *is-class P C*; *size ST* < *mxs* ⟧ ⟹
⊢ [*New C*],[] [::] [ty_i' *ST E A*, ty_i' (*Class C*#*ST*) *E A*]
by(*simp add*:*wt-defs* ty_i'-*def*)

lemma *wt-Cast*:
is-type P T ⟹
⊢ [*Checkcast T*],[] [::] [ty_i' (*U* # *ST*) *E A*, ty_i' (*T* # *ST*) *E A*]
by(*simp add*: ty_i'-*def wt-defs*)

lemma *wt-Instanceof*:
⟦ *is-type P T*; *is-refT U* ⟧ ⟹
⊢ [*Instanceof T*],[] [::] [ty_i' (*U* # *ST*) *E A*, ty_i' (*Boolean* # *ST*) *E A*]
by(*simp add*: ty_i'-*def wt-defs*)

lemma *wt-Push*:
⟦ *size ST* < *mxs*; *typeof v* = *Some T* ⟧
⟹ ⊢ [*Push v*],[] [::] [ty_i' *ST E A*, ty_i' (*T*#*ST*) *E A*]
by(*simp add*: ty_i'-*def wt-defs*)

lemma *wt-Pop*:
⊢ [*Pop*],[] [::] (ty_i' (*T*#*ST*) *E A* # ty_i' *ST E A* # τs)
by(*simp add*: ty_i'-*def wt-defs*)

lemma *wt-BinOpInstr*:
P ⊢ *T1* «*bop*» *T2* :: *T* ⟹ ⊢ [*BinOpInstr bop*],[] [::] [ty_i' (*T2* # *T1* # *ST*) *E A*, ty_i' (*T* # *ST*) *E A*]
by(*auto simp*:ty_i'-*def wt-defs dest*: *WT-binop-WTrt-binop intro*: *list-all2-refl*)

lemma *wt-Load*:
⟦ *size ST* < *mxs*; *size E* ≤ *mxl*; *i* ∈∈ *A*; *i* < *size E* ⟧
⟹ ⊢ [*Load i*],[] [::] [ty_i' *ST E A*, ty_i' (*E*!*i* # *ST*) *E A*]
by(*auto simp add*:ty_i'-*def wt-defs* ty_l-*def hyperset-defs intro*: *widens-refl*)

lemma *wt-Store*:
⟦ $P \vdash T \leq E!i$; $i < size\ E$; $size\ E \leq mxl$ ⟧ ⟹
$\vdash [Store\ i],[]\ [::]\ [ty_i'\ (T\#ST)\ E\ A,\ ty_i'\ ST\ E\ (\lfloor\{i\}\rfloor \sqcup A)]$
by(*auto simp*:*hyperset-defs nth-list-update* ty_i'*-def wt-defs* ty_l*-def intro*:*list-all2-all-nthI*)

lemma *wt-Get*:
⟦ $P \vdash C$ *sees* F:T (*fm*) *in* D; *class-type-of'* $U = \lfloor C \rfloor$ ⟧ ⟹
$\vdash [Getfield\ F\ D],[]\ [::]\ [ty_i'\ (U\ \#\ ST)\ E\ A,\ ty_i'\ (T\ \#\ ST)\ E\ A]$
by(*cases U*)(*auto simp*: ty_i'*-def wt-defs dest*: *sees-field-idemp sees-field-decl-above intro*: *widens-refl widen-trans widen-array-object*)

lemma *wt-Put*:
⟦ $P \vdash C$ *sees* F:T (*fm*) *in* D; *class-type-of'* $U = \lfloor C \rfloor$; $P \vdash T' \leq T$ ⟧ ⟹
$\vdash [Putfield\ F\ D],[]\ [::]\ [ty_i'\ (T'\ \#\ U\ \#\ ST)\ E\ A,\ ty_i'\ ST\ E\ A]$
by(*cases U*)(*auto 4 3 intro*: *sees-field-idemp widen-trans widen-array-object dest*: *sees-field-decl-above simp*: ty_i'*-def wt-defs*)

lemma *wt-CAS*:
⟦ $P \vdash C$ *sees* F:T (*fm*) *in* D; *class-type-of'* $U' = \lfloor C \rfloor$; *volatile fm*; $P \vdash T2 \leq T$; $P \vdash T3 \leq T$ ⟧ ⟹
$\vdash [CAS\ F\ D],[]\ [::]\ [ty_i'\ (T3\ \#\ T2\ \#\ U'\ \#\ ST)\ E\ A,\ ty_i'\ (Boolean\ \#\ ST)\ E\ A]$
by(*cases U'*)(*auto 4 4 simp add*: ty_i'*-def wt-defs intro*: *sees-field-idemp widen-trans widen-array-object dest*: *sees-field-decl-above*)

lemma *wt-Throw*:
$P \vdash C \preceq^* Throwable \Longrightarrow \vdash [ThrowExc],[]\ [::]\ [ty_i'\ (Class\ C\ \#\ ST)\ E\ A,\ \tau']$
by(*simp add*: ty_i'*-def wt-defs*)

lemma *wt-IfFalse*:
⟦ $2 \leq i$; $nat\ i < size\ \tau s + 2$; $P \vdash ty_i'\ ST\ E\ A \leq' \tau s\ !\ nat(i - 2)$ ⟧
⟹ $\vdash [IfFalse\ i],[]\ [::]\ ty_i'\ (Boolean\ \#\ ST)\ E\ A\ \#\ ty_i'\ ST\ E\ A\ \#\ \tau s$
by(*auto simp add*: ty_i'*-def wt-defs eval-nat-numeral nat-diff-distrib*)

lemma *wt-Goto*:
⟦ $0 \leq int\ pc + i$; $nat\ (int\ pc + i) < size\ \tau s$; $size\ \tau s \leq mpc$;
$P \vdash \tau s!pc \leq' \tau s\ !\ nat\ (int\ pc + i)$ ⟧
⟹ $P,T,mxs,mpc,[] \vdash Goto\ i,pc :: \tau s$
by(*clarsimp simp add*: *wt-defs*)

end

context *TC3* **begin**

lemma *wt-Invoke*:
⟦ $size\ es = size\ Ts'$; *class-type-of'* $U = \lfloor C \rfloor$; $P \vdash C$ *sees* M: $Ts \to T = m$ *in* D; $P \vdash Ts'\ [\leq]\ Ts$ ⟧
⟹ $\vdash [Invoke\ M\ (size\ es)],[]\ [::]\ [ty_i'\ (rev\ Ts' @ U\ \#\ ST)\ E\ A,\ ty_i'\ (T\#ST)\ E\ A]$
apply(*clarsimp simp add*: ty_i'*-def wt-defs*)
apply *safe*
apply(*simp-all* (*no-asm-use*))
apply(*auto simp add*: *intro*: *widens-refl*)
done

end

declare *nth-append*[*simp del*]
declare [[*simproc del*: *list-to-set-comprehension*]]

context *TC2* **begin**

corollary *wt-instrs-app3*[*simp*]:
⟦ ⊢ (*is*$_2$ :: 'b *instr list*),[] [::] (τ' # τ*s*$_2$); ⊢ *is*$_1$,*xt*$_1$ [::] τ # τ*s*$_1$ @ [τ']; *size* τ*s*$_1$+*1* = *size is*$_1$⟧
⟹ ⊢ (*is*$_1$ @ *is*$_2$),*xt*$_1$ [::] τ # τ*s*$_1$ @ τ' # τ*s*$_2$
using *wt-instrs-app2*[**where** *?xt*$_2$*.0* = [] **and** *?'b* = 'b] **by** (*simp add*:*shift-def*)

corollary *wt-instrs-Cons3*[*simp*]:
⟦ τ*s* ≠ []; ⊢ [*i*],[] [::] [τ,τ']; ⊢ *is*,[] [::] τ'#τ*s* ⟧
⟹ ⊢ (*i* # *is*),[] [::] τ # τ' # τ*s*
using *wt-instrs-Cons*[**where** *?xt* = []]
by (*simp add*:*shift-def*)

lemma *wt-instrs-xapp*:
⟦ ⊢ *is*$_1$ @ *is*$_2$, *xt* [::] τ*s*$_1$ @ *ty*$_i$' (*Class D* # *ST*) *E A* # τ*s*$_2$;
∀ τ ∈ *set* τ*s*$_1$. ∀ *ST' LT'*. τ = *Some*(*ST'*,*LT'*) ⟶
size ST ≤ *size ST'* ∧ *P* ⊢ *Some* (*drop* (*size ST'* − *size ST*) *ST'*,*LT'*) ≤' *ty*$_i$' *ST E A*;
size is$_1$ = *size* τ*s*$_1$; *size ST* < *mxs*; *case Co of None* ⇒ *D* = *Throwable* | *Some C* ⇒ *D* = *C* ∧ *is-class P C* ⟧ ⟹
⊢ *is*$_1$ @ *is*$_2$, *xt* @ [(*0*,*size is*$_1$ − *Suc n*,*Co*,*size is*$_1$,*size ST*)] [::] τ*s*$_1$ @ *ty*$_i$' (*Class D* # *ST*) *E A* # τ*s*$_2$
apply(*simp add*:*wt-instrs-def split del*: *option.split-asm*)
apply(*rule conjI*)
apply(*clarsimp split del*: *option.split-asm*)
apply *arith*
apply(*clarsimp split del*: *option.split-asm*)
apply(*erule allE*, *erule* (*1*) *impE*)
apply(*clarsimp simp add*: *wt-instr-def app-def eff-def split del*: *option.split-asm*)
apply(*rule conjI*)
apply (*thin-tac* ∀ *x*∈ *A* ∪ *B*. *P x* **for** *A B P*)
apply (*thin-tac* ∀ *x*∈ *A* ∪ *B*. *P x* **for** *A B P*)
apply (*clarsimp simp add*: *xcpt-app-def relevant-entries-def split del*: *option.split-asm*)
apply (*simp add*: *nth-append is-relevant-entry-def split*: *if-split-asm split del*: *option.split-asm*)
apply (*drule-tac x*=τ*s*$_1$!*pc* **in** *bspec*)
apply (*blast intro*: *nth-mem*)
apply *fastforce*
apply *fastforce*
apply (*rule conjI*)
apply(*clarsimp split del*: *option.split-asm*)
apply (*erule disjE*, *blast*)
apply (*erule disjE*, *blast*)
apply (*clarsimp simp add*: *xcpt-eff-def relevant-entries-def split*: *if-split-asm*)
apply(*clarsimp split del*: *option.split-asm*)
apply (*erule disjE*, *blast*)
apply (*erule disjE*, *blast*)
apply (*clarsimp simp add*: *xcpt-eff-def relevant-entries-def split*: *if-split-asm split del*: *option.split-asm*)
apply (*simp add*: *nth-append is-relevant-entry-def split*: *if-split-asm split del*: *option.split-asm*)

apply (*drule-tac* $x = \tau s_1!pc$ **in** *bspec*)
apply (*blast intro*: *nth-mem*)
apply (*fastforce simp add*: ty_i'*-def*)
done

lemma *wt-instrs-xapp-Some*[*trans*]:
$[\![\vdash is_1 @ is_2, xt [::] \tau s_1 @ ty_i' (Class\ C \# ST)\ E\ A \# \tau s_2;$
$\forall \tau \in set\ \tau s_1.\ \forall ST'\ LT'.\ \tau = Some(ST',LT') \longrightarrow$
$size\ ST \leq size\ ST' \wedge P \vdash Some\ (drop\ (size\ ST' - size\ ST)\ ST',LT') \leq' ty_i'\ ST\ E\ A;$
$size\ is_1 = size\ \tau s_1;\ is\text{-}class\ P\ C;\ size\ ST < mxs\ ]\!] \Longrightarrow$
$\vdash is_1 @ is_2, xt @ [(0,size\ is_1 - Suc\ n,Some\ C,size\ is_1,size\ ST)] [::] \tau s_1 @ ty_i' (Class\ C \# ST)\ E\ A \# \tau s_2$
by(*erule* (*3*) *wt-instrs-xapp*) *simp*

lemma *wt-instrs-xapp-Any*:
$[\![\vdash is_1 @ is_2, xt [::] \tau s_1 @ ty_i' (Class\ Throwable \# ST)\ E\ A \# \tau s_2;$
$\forall \tau \in set\ \tau s_1.\ \forall ST'\ LT'.\ \tau = Some(ST',LT') \longrightarrow$
$size\ ST \leq size\ ST' \wedge P \vdash Some\ (drop\ (size\ ST' - size\ ST)\ ST',LT') \leq' ty_i'\ ST\ E\ A;$
$size\ is_1 = size\ \tau s_1;\ size\ ST < mxs\ ]\!] \Longrightarrow$
$\vdash is_1 @ is_2, xt @ [(0,size\ is_1 - Suc\ n,None,size\ is_1,size\ ST)] [::] \tau s_1 @ ty_i' (Class\ Throwable \# ST)\ E\ A \# \tau s_2$
by(*erule* (*3*) *wt-instrs-xapp*) *simp*

end

declare [[*simproc add*: *list-to-set-comprehension*]]
declare *nth-append*[*simp*]

lemma *drop-Cons-Suc*:
$\bigwedge xs.\ drop\ n\ xs = y\#ys \Longrightarrow drop\ (Suc\ n)\ xs = ys$
apply (*induct n*)
apply *simp*
apply (*simp add*: *drop-Suc*)
done

lemma *drop-mess*:
$[\![Suc\ (length\ xs_0) \leq length\ xs;\ drop\ (length\ xs - Suc\ (length\ xs_0))\ xs = x \# xs_0]\!]$
$\Longrightarrow drop\ (length\ xs - length\ xs_0)\ xs = xs_0$
apply (*cases xs*)
apply *simp*
apply (*simp add*: *Suc-diff-le*)
apply (*case-tac length list − length* xs_0)
apply *simp*
apply (*simp add*: *drop-Cons-Suc*)
done

lemma *drop-mess2*:
assumes *len*: *Suc* (*Suc* (*length xs0*)) ≤ *length xs*
and *drop*: *drop* (*length xs* − *Suc* (*Suc* (*length xs0*))) *xs* = *x1* # *x2* # *xs0*
shows *drop* (*length xs* − *length xs0*) *xs* = *xs0*
proof(*cases xs*)
case *Nil* **with** *assms* **show** *?thesis* **by** *simp*
next
case (*Cons x xs'*)

```
  note Cons[simp]
  show ?thesis
  proof(cases xs')
    case Nil with assms show ?thesis by(simp)
  next
    case (Cons x' xs'')
    note Cons[simp]
    show ?thesis
    proof(rule drop-mess)
      from len show Suc (length xs0) ≤ length xs by simp
    next
      have drop (length xs − length (x2 # xs0)) xs = x2 # xs0
      proof(rule drop-mess)
        from len show Suc (length (x2 # xs0)) ≤ length xs by(simp)
      next
        from drop show drop (length xs − Suc (length (x2 # xs0))) xs = x1 # x2 # xs0 by simp
      qed
      thus drop (length xs − Suc (length xs0)) xs = x2 # xs0 by(simp)
    qed
  qed
qed

abbreviation postfix :: 'a list ⇒ 'a list ⇒ bool  (‹(-/ ≫= -)› [51, 50] 50) where
  postfix xs ys ≡ suffix ys xs

lemma postfix-conv-eq-length-drop:
  ST' ≫= ST ⟷ length ST ≤ length ST' ∧ drop (length ST' − length ST) ST' = ST
apply(auto)
apply (metis append-eq-conv-conj append-take-drop-id diff-is-0-eq drop-0 linorder-not-less nat-le-linear
suffix-take)
apply (metis append-take-drop-id length-drop suffix-take same-append-eq size-list-def)
by (metis suffix-drop)

declare suffix-ConsI[simp]

context TC0 begin

declare after-def[simp] pair-eq-ty_i'-conv[simp]

lemma
  assumes ST0 ≫= ST'
  shows compT-ST-prefix:
  ⌊(ST,LT)⌋ ∈ set(compT E A ST0 e) ⟹ ST ≫= ST'

  and compTs-ST-prefix:
  ⌊(ST,LT)⌋ ∈ set(compTs E A ST0 es) ⟹ ST ≫= ST'
using assms
by(induct E A ST0 e and E A ST0 es rule: compT-compTs-induct) auto

declare after-def[simp del] pair-eq-ty_i'-conv[simp del]

end
declare suffix-ConsI[simp del]
```

lemma *fun-of-simp* [*simp*]: *fun-of S x y* = ((*x*,*y*) ∈ *S*)
by (*simp add*: *fun-of-def*)

declare *widens-refl* [*iff*]

context *TC3* **begin**

theorem *compT-wt-instrs*:
⟦ *P*,*E* ⊢*1 e* :: *T*; 𝒟 *e A*; ℬ *e* (*size E*); *size ST* + *max-stack e* ≤ *mxs*; *size E* + *max-vars e* ≤ *mxl*; *set E* ⊆ *types P* ⟧
⟹ ⊢ *compE2 e*, *compxE2 e 0* (*size ST*) [::] *ty*$_i$′ *ST E A* # *compT E A ST e* @ [*after E A ST e*]
(**is** *PROP ?P e E T A ST*)

and *compTs-wt-instrs*:
⟦ *P*,*E* ⊢*1 es*[::]*Ts*; 𝒟*s es A*; ℬ*s es* (*size E*); *size ST* + *max-stacks es* ≤ *mxs*; *size E* + *max-varss es* ≤ *mxl*; *set E* ⊆ *types P* ⟧
⟹ *let* τ*s* = *ty*$_i$′ *ST E A* # *compTs E A ST es*
in ⊢ *compEs2 es*,*compxEs2 es 0* (*size ST*) [::] τ*s* ∧ *last* τ*s* = *ty*$_i$′ (*rev Ts* @ *ST*) *E* (*A* ⊔ 𝒜*s es*)
(**is** *PROP ?Ps es E Ts A ST*)
proof(*induct E A ST e* **and** *E A ST es arbitrary*: *T* **and** *Ts rule*: *compT-compTs-induct*)
case (*TryCatch E A ST e*$_1$ *C i e*$_2$)
hence [*simp*]: *i* = *size E* **by** *simp*
have *wt*$_1$: *P*,*E* ⊢*1 e*$_1$ :: *T* **and** *wt*$_2$: *P*,*E*@[*Class C*] ⊢*1 e*$_2$ :: *T*
and *class*: *is-class P C* **using** *TryCatch* **by** *auto*
let *?A*$_1$ = *A* ⊔ 𝒜 *e*$_1$ **let** *?A*$_i$ = *A* ⊔ ⌊{*i*}⌋ **let** *?E*$_i$ = *E* @ [*Class C*]
let *?*τ = *ty*$_i$′ *ST E A* **let** *?*τ*s*$_1$ = *compT E A ST e*$_1$
let *?*τ$_1$ = *ty*$_i$′ (*T*#*ST*) *E ?A*$_1$ **let** *?*τ$_2$ = *ty*$_i$′ (*Class C*#*ST*) *E A*
let *?*τ$_3$ = *ty*$_i$′ *ST ?E*$_i$ *?A*$_i$ **let** *?*τ*s*$_2$ = *compT ?E*$_i$ *?A*$_i$ *ST e*$_2$
let *?*τ$_2$′ = *ty*$_i$′ (*T*#*ST*) *?E*$_i$ (*?A*$_i$ ⊔ 𝒜 *e*$_2$)
let *?*τ′ = *ty*$_i$′ (*T*#*ST*) *E* (*A* ⊔ 𝒜 *e*$_1$ ⊓ (𝒜 *e*$_2$ ⊖ *i*))
let *?go* = *Goto* (*int*(*size*(*compE2 e*$_2$)) + *2*)
have *PROP ?P e*$_2$ *?E*$_i$ *T ?A*$_i$ *ST* **by** *fact*
hence ⊢ *compE2 e*$_2$,*compxE2 e*$_2$ *0* (*size ST*) [::] (*?*τ$_3$ # *?*τ*s*$_2$) @ [*?*τ$_2$′]
using *TryCatch.prems class* **by**(*auto simp*:*after-def*)
also have *?A*$_i$ ⊔ 𝒜 *e*$_2$ = (*A* ⊔ 𝒜 *e*$_2$) ⊔ ⌊{*size E*}⌋
by(*fastforce simp*:*hyperset-defs*)
also have *P* ⊢ *ty*$_i$′ (*T*#*ST*) *?E*$_i$ … ≤′ *ty*$_i$′ (*T*#*ST*) *E* (*A* ⊔ 𝒜 *e*$_2$)
by(*simp add*:*hyperset-defs ty*$_l$*-incr ty*$_i$′*-def*)
also have *P* ⊢ … ≤′ *ty*$_i$′ (*T*#*ST*) *E* (*A* ⊔ 𝒜 *e*$_1$ ⊓ (𝒜 *e*$_2$ ⊖ *i*))
by(*auto intro*!: *ty*$_l$*-antimono simp*:*hyperset-defs ty*$_i$′*-def*)
also have (*?*τ$_3$ # *?*τ*s*$_2$) @ [*?*τ′] = *?*τ$_3$ # *?*τ*s*$_2$ @ [*?*τ′] **by** *simp*
also have ⊢ [*Store i*],[] [::] *?*τ$_2$ # [] @ [*?*τ$_3$]
using *TryCatch.prems*
by(*auto simp*:*nth-list-update wt-defs ty*$_i$′*-def ty*$_l$*-def*
list-all2-conv-all-nth hyperset-defs)
also have [] @ (*?*τ$_3$ # *?*τ*s*$_2$ @ [*?*τ′]) = (*?*τ$_3$ # *?*τ*s*$_2$ @ [*?*τ′]) **by** *simp*
also have *P*,*T*$_r$,*mxs*,*size*(*compE2 e*$_2$)+*3*,[] ⊢ *?go*,*0* :: *?*τ$_1$#*?*τ$_2$#*?*τ$_3$#*?*τ*s*$_2$ @ [*?*τ′]
by(*auto simp*: *hyperset-defs ty*$_i$′*-def wt-defs nth-Cons nat-add-distrib*
fun-of-def intro: *ty*$_l$*-antimono list-all2-refl split*:*nat.split*)
also have ⊢ *compE2 e*$_1$,*compxE2 e*$_1$ *0* (*size ST*) [::] *?*τ # *?*τ*s*$_1$ @ [*?*τ$_1$]
using *TryCatch* **by**(*auto simp*:*after-def*)
also have *?*τ # *?*τ*s*$_1$ @ *?*τ$_1$ # *?*τ$_2$ # *?*τ$_3$ # *?*τ*s*$_2$ @ [*?*τ′] =
(*?*τ # *?*τ*s*$_1$ @ [*?*τ$_1$]) @ *?*τ$_2$ # *?*τ$_3$ # *?*τ*s*$_2$ @ [*?*τ′] **by** *simp*

```
  also have compE2 e1 @ ?go  # [Store i] @ compE2 e2 =
            (compE2 e1 @ [?go]) @ (Store i # compE2 e2) by simp
  also
  let ?Q τ = ∀ ST' LT'. τ = ⌊(ST', LT')⌋ ⟶
    size ST ≤ size ST' ∧ P ⊢ Some (drop (size ST' − size ST) ST',LT') ≤' ty_i' ST E A
  {
    have ?Q (ty_i' ST E A) by(clarsimp simp add: ty_i'-def)
    moreover have ?Q (ty_i' (T # ST) E ?A_1)
      by (fastforce simp add: ty_i'-def hyperset-defs intro!: ty_l-antimono)
    moreover { fix τ
      assume τ: τ ∈ set (compT E A ST e_1)
        hence ∀ ST' LT'. τ = ⌊(ST', LT')⌋ ⟶ ST' ≫= ST by(auto intro: compT-ST-prefix[OF
suffix-order.order-refl])
      with τ have ?Q τ unfolding postfix-conv-eq-length-drop using ‹B (try e_1 catch(C i) e_2) (length
E)›
        by(fastforce dest!: compT-LT-prefix simp add: ty_i'-def) }
    ultimately
    have ∀ τ∈set (ty_i' ST E A # compT E A ST e_1 @ [ty_i' (T # ST) E ?A_1]). ?Q τ by auto
  }
  also from TryCatch.prems max-stack1[of e_1] have size ST + 1 ≤ mxs by auto
  ultimately show ?case using wt_1 wt_2 TryCatch.prems class
    by (simp add:after-def)(erule-tac x=0 in meta-allE, simp)
next
  case (Synchronized E A ST i e1 e2)
  note wt = ‹P,E ⊢1 sync_i (e1) e2 :: T›
  then obtain U where wt1: P,E ⊢1 e1 :: U
    and U: is-refT U U ≠ NT
    and wt2: P,E@[Class Object] ⊢1 e2 :: T by auto
  from ‹B (sync_i (e1) e2) (length E)› have [simp]: i = length E
    and B1: B e1 (length E) and B2: B e2 (length (E@[Class Object])) by auto

  note lenST = ‹length ST + max-stack (sync_i (e1) e2) ≤ mxs›
  note lenE = ‹length E + max-vars (sync_i (e1) e2) ≤ mxl›

  let ?A1 = A ⊔ A e1 let ?A2 = ?A1 ⊔ ⌊{i}⌋
  let ?A3 = ?A2 ⊔ A e2 let ?A4 = ?A1 ⊔ A e2
  let ?E1 = E @ [Class Object]
  let ?τ = ty_i' ST E A let ?τs1 = compT E A ST e1
  let ?τ1 = ty_i' (U#ST) E ?A1
  let ?τ1' = ty_i' (Class Object # Class Object # ST) E ?A1
  let ?τ1'' = ty_i' (Class Object#ST) ?E1 ?A2
  let ?τ1''' = ty_i' ST ?E1 ?A2
  let ?τs2 = compT ?E1 ?A2 ST e2
  let ?τ2 = ty_i' (T#ST) ?E1 ?A3 let ?τ2' = ty_i' (Class Object#T#ST) ?E1 ?A3
  let ?τ2'' = ?τ2
  let ?τ3 = ty_i' (Class Throwable#ST) ?E1 ?A2
  let ?τ3' = ty_i' (Class Object#Class Throwable#ST) ?E1 ?A2
  let ?τ3'' = ?τ3
  let ?τ' = ty_i' (T#ST) E ?A4

  from lenE lenST max-stack1[of e2] U
  have ⊢ [Load i, MExit, ThrowExc], [] [::] [?τ3, ?τ3', ?τ3'', ?τ']
    by(auto simp add: ty_i'-def ty_l-def wt-defs hyperset-defs nth-Cons split: nat.split)
  also have P,T_r,mxs,5,[] ⊢ Goto 4,0 :: [?τ2'', ?τ3, ?τ3', ?τ3'', ?τ']
```

by(*auto simp*: *hyperset-defs* ty_i'*-def wt-defs intro*: ty_l*-antimono* ty_l*-incr*)
also have $P,T_r,mxs,6,[] \vdash$ *MExit*,*0* :: [$?\tau 2'$, $?\tau 2''$, $?\tau 3$, $?\tau 3'$, $?\tau 3''$, $?\tau'$]
by(*auto simp*: *hyperset-defs* ty_i'*-def wt-defs intro*: ty_l*-antimono* ty_l*-incr*)
also from *lenE lenST max-stack1*[*of e2*]
have $P,T_r,mxs,7,[] \vdash$ *Load i*,*0* :: [$?\tau 2$, $?\tau 2'$, $?\tau 2''$, $?\tau 3$, $?\tau 3'$, $?\tau 3''$, $?\tau'$]
by(*auto simp*: *hyperset-defs* ty_i'*-def wt-defs* ty_l*-def intro*: ty_l*-antimono*)
also from ‹$\mathcal{D}$ ($sync_i$ (*e1*) *e2*) *A*› **have** $\mathcal{D}$ *e2* (*A* ⊔ $\mathcal{A}$ *e1* ⊔ ⌊{*length E*}⌋)
by(*auto elim*!: *D-mono*′ *simp add*: *hyperset-defs*)
with ‹*PROP ?P e2 ?E1 T ?A2 ST*› *Synchronized wt2 is-class-Object*[*OF wf-prog*]
have ⊢ *compE2 e2*, *compxE2 e2 0* (*size ST*) [::] $?\tau 1'''$#$?\tau s2$@[$?\tau 2$]
by(*auto simp add*: *after-def*)
finally have ⊢ (*compE2 e2* @ [*Load i*, *MExit*, *Goto 4*]) @ [*Load i*, *MExit*, *ThrowExc*], *compxE2 e2 0* (*size ST*) [::]
($?\tau 1'''$ # $?\tau s2$ @ [$?\tau 2$, $?\tau 2'$, $?\tau 2''$]) @ [$?\tau 3$, $?\tau 3'$, $?\tau 3''$, $?\tau'$]
by(*simp*)
hence ⊢ (*compE2 e2* @ [*Load i*, *MExit*, *Goto 4*]) @ [*Load i*, *MExit*, *ThrowExc*],
compxE2 e2 0 (*size ST*) @ [(*0*, *size* (*compE2 e2* @ [*Load i*, *MExit*, *Goto 4*]) − *Suc 2*, *None*, *size* (*compE2 e2* @ [*Load i*, *MExit*, *Goto 4*]), *size ST*)] [::]
($?\tau 1'''$ # $?\tau s2$ @ [$?\tau 2$, $?\tau 2'$, $?\tau 2''$]) @ [$?\tau 3$, $?\tau 3'$, $?\tau 3''$, $?\tau'$]
proof(*rule wt-instrs-xapp-Any*)
from *lenST* **show** *length ST* < *mxs* **by** *simp*
next
show ∀ τ∈*set* ($?\tau 1'''$ # $?\tau s2$ @ [$?\tau 2$, $?\tau 2'$, $?\tau 2''$]). ∀ *ST*′ *LT*′.
τ = ⌊(*ST*′, *LT*′)⌋ ⟶ *length ST* ≤ *length ST*′ ∧
P ⊢ ⌊(*drop* (*length ST*′ − *length ST*) *ST*′, *LT*′)⌋ ≤′ ty_i' *ST* (*E* @ [*Class Object*]) *?A2*
proof(*intro strip*)
fix τ *ST*′ *LT*′
assume τ∈*set* ($?\tau 1'''$ # $?\tau s2$ @ [$?\tau 2$, $?\tau 2'$, $?\tau 2''$]) τ = ⌊(*ST*′, *LT*′)⌋
hence τ: ⌊(*ST*′, *LT*′)⌋ ∈ *set* ($?\tau 1'''$ # $?\tau s2$ @ [$?\tau 2$, $?\tau 2'$, $?\tau 2''$]) **by** *simp*
show *length ST* ≤ *length ST*′ ∧ *P* ⊢ ⌊(*drop* (*length ST*′ − *length ST*) *ST*′, *LT*′)⌋ ≤′ ty_i' *ST* (*E* @ [*Class Object*]) *?A2*
proof(*cases* ⌊(*ST*′, *LT*′)⌋ ∈ *set* $?\tau s2$)
case *True*
from *compT-ST-prefix*[*OF suffix-order.order-refl this*] *compT-LT-prefix*[*OF this B2*]
show *?thesis* **unfolding** *postfix-conv-eq-length-drop* **by**(*simp add*: ty_i'*-def*)
next
case *False*
with τ **show** *?thesis*
by(*auto simp add*: ty_i'*-def hyperset-defs intro*: ty_l*-antimono*)
qed
qed
qed *simp*
hence ⊢ *compE2 e2* @ [*Load i*, *MExit*, *Goto 4*, *Load i*, *MExit*, *ThrowExc*],
compxE2 e2 0 (*size ST*) @ [(*0*, *size* (*compE2 e2*), *None*, *Suc* (*Suc* (*Suc* (*size* (*compE2 e2*))))), *size ST*)] [::]
$?\tau 1'''$ # $?\tau s2$ @ [$?\tau 2$, $?\tau 2'$, $?\tau 2''$, $?\tau 3$, $?\tau 3'$, $?\tau 3''$, $?\tau'$] **by** *simp*
also from *wt1* ‹*set E* ⊆ *types P*› **have** *is-type P U* **by**(*rule WT1-is-type*[*OF wf-prog*])
with *U* **have** *P* ⊢ *U* ≤ *Class Object* **by**(*auto elim*!: *is-refT.cases intro*: *subcls-C-Object*[*OF - wf-prog*] *widen-array-object*)
with *lenE lenST max-stack1*[*of e2*]
have ⊢ [*Dup*, *Store i*, *MEnter*], [] [::] [$?\tau 1$, $?\tau 1'$, $?\tau 1''$] @ [$?\tau 1'''$]
by(*auto simp add*: ty_i'*-def* ty_l*-def wt-defs hyperset-defs nth-Cons nth-list-update list-all2-conv-all-nth split*: *nat.split*)
finally have ⊢ *Dup* # *Store i* # *MEnter* # *compE2 e2* @ [*Load i*, *MExit*, *Goto 4*, *Load i*, *MExit*,

ThrowExc],
compxE2 e2 3 (*size ST*) @ [(*3*, *3* + *size* (*compE2 e2*), *None*, *6* + *size* (*compE2 e2*), *size ST*)]
[::] *?τ1* # *?τ1′* # *?τ1″* # *?τ1‴* # *?τs2* @ [*?τ2*, *?τ2′*, *?τ2″*, *?τ3*, *?τ3′*, *?τ3″*, *?τ′*]
by(*simp add*: *eval-nat-numeral shift-def*)
also from ‹*PROP ?P e1 E U A ST*› *wt1 B1* ‹$\mathcal{D}$ (*sync*$_i$ (*e1*) *e2*) *A*› *lenE lenST* ‹*set E* ⊆ *types P*›
have ⊢ *compE2 e1*, *compxE2 e1 0* (*size ST*) [::] *?τ*#*?τs1*@[*?τ1*]
by(*auto simp add*: *after-def*)
finally show *?case* **using** *wt1 wt2 wt* **by**(*simp add*: *after-def ac-simps shift-Cons-tuple hyperUn-assoc*)
next
case *new* **thus** *?case* **by**(*auto simp add*:*after-def wt-New*)
next
case (*BinOp E A ST e*$_1$ *bop e*$_2$)
have *T*: *P*,*E* ⊢*1 e*$_1$ «*bop*» *e*$_2$ :: *T* **by** *fact*
then obtain *T*$_1$ *T*$_2$ **where** *T*$_1$: *P*,*E* ⊢*1 e*$_1$ :: *T*$_1$ **and** *T*$_2$: *P*,*E* ⊢*1 e*$_2$ :: *T*$_2$ **and**
bopT: *P* ⊢ *T*$_1$«*bop*»*T*$_2$ :: *T* **by** *auto*
let *?A*$_1$ = *A* ⊔ $\mathcal{A}$ *e*$_1$ **let** *?A*$_2$ = *?A*$_1$ ⊔ $\mathcal{A}$ *e*$_2$
let *?τ* = *ty*$_i$′ *ST E A* **let** *?τs*$_1$ = *compT E A ST e*$_1$
let *?τ*$_1$ = *ty*$_i$′ (*T*$_1$#*ST*) *E ?A*$_1$ **let** *?τs*$_2$ = *compT E ?A*$_1$ (*T*$_1$#*ST*) *e*$_2$
let *?τ*$_2$ = *ty*$_i$′ (*T*$_2$#*T*$_1$#*ST*) *E ?A*$_2$ **let** *?τ′* = *ty*$_i$′ (*T*#*ST*) *E ?A*$_2$
from *bopT* **have** ⊢ [*BinOpInstr bop*],[] [::] [*?τ*$_2$,*?τ′*] **by**(*rule wt-BinOpInstr*)
also from *BinOp.hyps*(*2*)[*of T*$_2$] *BinOp.prems T*$_2$ *T*$_1$
have ⊢ *compE2 e*$_2$, *compxE2 e*$_2$ *0* (*size* (*ty E e*$_1$#*ST*)) [::] *?τ*$_1$#*?τs*$_2$@[*?τ*$_2$] **by** (*auto simp*: *after-def*)
also from *BinOp T*$_1$ **have** ⊢ *compE2 e*$_1$, *compxE2 e*$_1$ *0* (*size ST*) [::] *?τ*#*?τs*$_1$@[*?τ*$_1$]
by (*auto simp*: *after-def*)
finally show *?case* **using** *T T*$_1$ *T*$_2$ **by** (*simp add*: *after-def hyperUn-assoc*)
next
case (*Cons E A ST e es*)
have *P*,*E* ⊢*1 e* # *es* [::] *Ts* **by** *fact*
then obtain *T*$_e$ *Ts′* **where**
T$_e$: *P*,*E* ⊢*1 e* :: *T*$_e$ **and** *Ts′*: *P*,*E* ⊢*1 es* [::] *Ts′* **and**
Ts: *Ts* = *T*$_e$#*Ts′* **by** *auto*
let *?A*$_e$ = *A* ⊔ $\mathcal{A}$ *e*
let *?τ* = *ty*$_i$′ *ST E A* **let** *?τs*$_e$ = *compT E A ST e*
let *?τ*$_e$ = *ty*$_i$′ (*T*$_e$#*ST*) *E ?A*$_e$ **let** *?τs′* = *compTs E ?A*$_e$ (*T*$_e$#*ST*) *es*
let *?τs* = *?τ* # *?τs*$_e$ @ (*?τ*$_e$ # *?τs′*)
from *Cons.hyps*(*2*) *Cons.prems T*$_e$ *Ts′*
have ⊢ *compEs2 es*, *compxEs2 es 0* (*size* (*T*$_e$#*ST*)) [::] *?τ*$_e$#*?τs′* **by** (*simp add*: *after-def*)
also from *Cons T*$_e$ **have** ⊢ *compE2 e*, *compxE2 e 0* (*size ST*) [::] *?τ*#*?τs*$_e$@[*?τ*$_e$] **by** (*auto simp*: *after-def*)
moreover
from *Cons.hyps*(*2*)[*OF Ts′*] *Cons.prems T*$_e$ *Ts′ Ts*
have *last ?τs* = *ty*$_i$′ (*rev Ts*@*ST*) *E* (*?A*$_e$ ⊔ $\mathcal{A}$*s es*) **by** *simp*
ultimately show *?case* **using** *T*$_e$
by(*auto simp add*: *after-def hyperUn-assoc shift-compxEs2 stack-xlift-compxEs2 simp del*: *compxE2-size-convs compxEs2-size-convs compxEs2-stack-xlift-convs compxE2-stack-xlift-convs intro*: *wt-instrs-app2*)
next
case (*FAss E A ST e*$_1$ *F D e*$_2$)
hence *Void*: *P*,*E* ⊢*1 e*$_1$·*F*{*D*} := *e*$_2$ :: *Void* **by** *auto*
then obtain *U C T T′ fm* **where**
C: *P*,*E* ⊢*1 e*$_1$ :: *U* **and** *U*: *class-type-of′ U* = ⌊*C*⌋ **and** *sees*: *P* ⊢ *C sees F*:*T* (*fm*) *in D* **and**
T′: *P*,*E* ⊢*1 e*$_2$ :: *T′* **and** *T′*-*T*: *P* ⊢ *T′* ≤ *T* **by** *auto*
let *?A*$_1$ = *A* ⊔ $\mathcal{A}$ *e*$_1$ **let** *?A*$_2$ = *?A*$_1$ ⊔ $\mathcal{A}$ *e*$_2$
let *?τ* = *ty*$_i$′ *ST E A* **let** *?τs*$_1$ = *compT E A ST e*$_1$

let $?\tau_1 = ty_i'\ (U\#ST)\ E\ ?A_1$ **let** $?\tau s_2 = compT\ E\ ?A_1\ (U\#ST)\ e_2$
let $?\tau_2 = ty_i'\ (T'\#U\#ST)\ E\ ?A_2$ **let** $?\tau_3 = ty_i'\ ST\ E\ ?A_2$
let $?\tau' = ty_i'\ (Void\#ST)\ E\ ?A_2$
from *FAss.prems sees T'-T U*
have $\vdash$ [*Putfield F D*,*Push Unit*],[] [::] [$?\tau_2$,$?\tau_3$,$?\tau'$]
by (*fastforce simp add*: *wt-Push wt-Put*)
also from *FAss.hyps*(*2*)[*of T'*] *FAss.prems T' C*
have $\vdash$ *compE2* e_2, *compxE2* e_2 *0* (*size ST+1*) [::] $?\tau_1\#?\tau s_2@[?\tau_2]$
by (*auto simp add*: *after-def hyperUn-assoc*)
also from *FAss C* **have** $\vdash$ *compE2* e_1, *compxE2* e_1 *0* (*size ST*) [::] $?\tau\#?\tau s_1@[?\tau_1]$
by (*auto simp add*: *after-def*)
finally show *?case* **using** *Void C T'* **by** (*simp add*: *after-def hyperUn-assoc*)
next
case *Val* **thus** *?case* **by**(*auto simp*:*after-def wt-Push*)
next
case (*Cast T exp*) **thus** *?case* **by** (*auto simp*:*after-def wt-Cast*)
next
case (*InstanceOf E A ST e*) **thus** *?case*
by(*auto simp*:*after-def intro*!: *wt-Instanceof wt-instrs-app3 intro*: *widen-refT refT-widen*)
next
case (*BlockNone E A ST i Ti e*)
from ‹*P*,*E* $\vdash$*1* {*i*:*Ti*=*None*; *e*} :: *T*› **have** *wte*: *P*,*E*@[*Ti*] $\vdash$*1* *e* :: *T*
and *Ti*: *is-type P Ti* **by** *auto*
let $?\tau s = ty_i'\ ST\ E\ A\ \#\ compT\ (E\ @\ [Ti])\ (A\ominus i)\ ST\ e$
from *BlockNone wte Ti*
have $\vdash$ *compE2 e*, *compxE2 e 0* (*size ST*) [::] $?\tau s\ @\ [ty_i'\ (T\#ST)\ (E@[Ti])\ (A\ominus(size\ E) \sqcup \mathcal{A}\ e)]$
by(*auto simp add*: *after-def*)
also have $P \vdash ty_i'\ (T\ \#\ ST)\ (E@[Ti])\ (A \ominus size\ E \sqcup \mathcal{A}\ e) \leq' ty_i'\ (T\ \#\ ST)\ (E@[Ti])\ ((A \sqcup \mathcal{A}\ e) \ominus size\ E)$
by(*auto simp add*:*hyperset-defs intro*: ty_i'*-antimono*)
also have $\ldots = ty_i'\ (T\ \#\ ST)\ E\ (A \sqcup \mathcal{A}\ e)$ **by** *simp*
also have $P \vdash \ldots \leq' ty_i'\ (T\ \#\ ST)\ E\ (A \sqcup (\mathcal{A}\ e \ominus i))$
by(*auto simp add*:*hyperset-defs intro*: ty_i'*-antimono*)
finally show *?case* **using** *BlockNone.prems* **by**(*simp add*: *after-def*)
next
case (*BlockSome E A ST i Ti v e*)
from ‹*P*,*E* $\vdash$*1* {*i*:*Ti*=$\lfloor v \rfloor$; *e*} :: *T*› **obtain** *Tv*
where *Tv*: *P*,*E* $\vdash$*1* *Val v* :: *Tv* *P* $\vdash$ *Tv* $\leq$ *Ti*
and *wte*: *P*,*E*@[*Ti*] $\vdash$*1* *e* :: *T*
and *Ti*: *is-type P Ti* **by** *auto*
from ‹*length ST* + *max-stack* {*i*:*Ti*=$\lfloor v \rfloor$; *e*} $\leq$ *mxs*›
have *lenST*: *length ST* + *max-stack e* $\leq$ *mxs* **by** *simp*
from ‹*length E* + *max-vars* {*i*:*Ti*=$\lfloor v \rfloor$; *e*} $\leq$ *mxl*›
have *lenE*: *length* (*E*@[*Ti*]) + *max-vars e* $\leq$ *mxl* **by** *simp*
from ‹$\mathcal{B}$ {*i*:*Ti*=$\lfloor v \rfloor$; *e*} (*length E*)› **have** [*simp*]: *i* = *length E*
and *B*: $\mathcal{B}$ *e* (*length* (*E*@[*Ti*])) **by** *auto*

from *BlockSome wte*
have $\vdash$ *compE2 e*, *compxE2 e 0* (*size ST*) [::] $(ty_i'\ ST\ (E\ @\ [Ti])\ (A \sqcup \lfloor\{length\ E\}\rfloor)\ \#\ compT\ (E\ @\ [Ti])\ (A \sqcup \lfloor\{i\}\rfloor)\ ST\ e)\ @\ [ty_i'\ (T\#ST)\ (E@[Ti])\ (A \sqcup \lfloor\{size\ E\}\rfloor \sqcup \mathcal{A}\ e)]$
by(*auto simp add*: *after-def*)
also have $P \vdash ty_i'\ (T\ \#\ ST)\ (E\ @\ [Ti])\ (A \sqcup \lfloor\{length\ E\}\rfloor \sqcup \mathcal{A}\ e) \leq' ty_i'\ (T\ \#\ ST)\ (E\ @\ [Ti])\ ((A \sqcup \mathcal{A}\ e) \ominus length\ E)$

by(*auto simp add*: *hyperset-defs intro*: ty_i'-*antimono*)
also have $\ldots = ty_i'\ (T\ \#\ ST)\ E\ (A \sqcup \mathcal{A}\ e)$ **by** *simp*
also have $P \vdash \ldots \leq' ty_i'\ (T\ \#\ ST)\ E\ (A \sqcup (\mathcal{A}\ e \ominus i))$
by(*auto simp add*:*hyperset-defs intro*: ty_i'-*antimono*)
also note *append-Cons*
also {
from *lenST max-stack1*[*of e*] *Tv*
have $\vdash [Push\ v], [] [::] [ty_i'\ ST\ E\ A, ty_i'\ (ty\ E\ (Val\ v)\ \#\ ST)\ E\ A]$
by(*auto intro*: *wt-Push*)
moreover from *Tv lenE*
have $\vdash [Store\ (length\ E)], [] [::] [ty_i'\ (Tv\ \#\ ST)\ (E\ @\ [Ti])\ (A \ominus length\ E), ty_i'\ ST\ (E\ @\ [Ti])\ (\lfloor\{length\ E\}\rfloor \sqcup (A \ominus length\ E))]$
by −(*rule wt-Store*, *auto*)
moreover have $ty_i'\ (Tv\ \#\ ST)\ (E\ @\ [Ti])\ (A \ominus length\ E) = ty_i'\ (Tv\ \#\ ST)\ E\ A$ **by**(*simp add*: ty_i'-*def*)
moreover have $\lfloor\{length\ E\}\rfloor \sqcup (A \ominus length\ E) = A \sqcup \lfloor\{length\ E\}\rfloor$ **by**(*simp add*: *hyperset-defs*)
ultimately have $\vdash [Push\ v, Store\ (length\ E)], [] [::] [ty_i'\ ST\ E\ A, ty_i'\ (Tv\ \#\ ST)\ E\ A, ty_i'\ ST\ (E\ @\ [Ti])\ (A \sqcup \lfloor\{length\ E\}\rfloor)]$
using *Tv* **by**(*auto intro*: *wt-instrs-Cons3*)
}
finally show *?case* **using** *Tv* ‹$P,E \vdash 1\ \{i{:}Ti{=}\lfloor v\rfloor;\ e\} :: T$› *wte* **by**(*simp add*: *after-def*)
next
case *Var* **thus** *?case* **by**(*auto simp*:*after-def wt-Load*)
next
case *FAcc* **thus** *?case* **by**(*auto simp*:*after-def wt-Get*)
next
case (*LAss E A ST i e*) **thus** *?case* **using** *max-stack1*[*of e*]
by(*auto simp*: *hyper-insert-comm after-def wt-Store wt-Push simp del*: *hyperUn-comm hyperUn-leftComm*)
next
case *Nil* **thus** *?case* **by** *auto*
next
case *throw* **thus** *?case* **by**(*auto simp add*: *after-def wt-Throw*)
next
case (*While E A ST e c*)
obtain *Tc* **where** *wte*: $P,E \vdash 1\ e :: Boolean$ **and** *wtc*: $P,E \vdash 1\ c :: Tc$
and [*simp*]: $T = Void$ **using** *While* **by** *auto*
have [*simp*]: $ty\ E\ (while\ (e)\ c) = Void$ **using** *While* **by** *simp*
let $?A_0 = A \sqcup \mathcal{A}\ e$ **let** $?A_1 = ?A_0 \sqcup \mathcal{A}\ c$
let $?\tau = ty_i'\ ST\ E\ A$ **let** $?\tau s_e = compT\ E\ A\ ST\ e$
let $?\tau_e = ty_i'\ (Boolean\#ST)\ E\ ?A_0$ **let** $?\tau_1 = ty_i'\ ST\ E\ ?A_0$
let $?\tau s_c = compT\ E\ ?A_0\ ST\ c$ **let** $?\tau_c = ty_i'\ (Tc\#ST)\ E\ ?A_1$
let $?\tau_2 = ty_i'\ ST\ E\ ?A_1$ **let** $?\tau' = ty_i'\ (Void\#ST)\ E\ ?A_0$
let $?\tau s = (?\tau\ \#\ ?\tau s_e\ @\ [?\tau_e])\ @\ ?\tau_1\ \#\ ?\tau s_c\ @\ [?\tau_c, ?\tau_2, ?\tau_1, ?\tau']$
have $\vdash [],[]\ [::]\ []\ @\ ?\tau s$ **by**(*simp add*:*wt-instrs-def*)
also
from *While.hyps*(*1*)[*of Boolean*] *While.prems*
have $\vdash compE2\ e, compxE2\ e\ 0\ (size\ ST)\ [::]\ ?\tau\ \#\ ?\tau s_e\ @\ [?\tau_e]$
by (*auto simp*:*after-def*)
also
have $[]\ @\ ?\tau s = (?\tau\ \#\ ?\tau s_e)\ @\ ?\tau_e\ \#\ ?\tau_1\ \#\ ?\tau s_c\ @\ [?\tau_c, ?\tau_2, ?\tau_1, ?\tau']$ **by** *simp*
also
let $?n_e = size(compE2\ e)$ **let** $?n_c = size(compE2\ c)$
let $?if = IfFalse\ (int\ ?n_c + 3)$
have $\vdash [?if],[]\ [::]\ ?\tau_e\ \#\ ?\tau_1\ \#\ ?\tau s_c\ @\ [?\tau_c, ?\tau_2, ?\tau_1, ?\tau']$

```
    by(simp add: wt-instr-Cons wt-instr-append wt-IfFalse
                 nat-add-distrib split: nat-diff-split)
  also
  have (?τ # ?τs_e) @ (?τ_e # ?τ_1 # ?τs_c @ [?τ_c, ?τ_2, ?τ_1, ?τ']) = ?τs by simp
  also from While.hyps(2)[of Tc] While.prems wtc
  have ⊢ compE2 c,compxE2 c 0 (size ST) [::] ?τ_1 # ?τs_c @ [?τ_c]
    by (auto simp:after-def)
  also have ?τs = (?τ # ?τs_e @ [?τ_e,?τ_1] @ ?τs_c) @ [?τ_c,?τ_2,?τ_1,?τ'] by simp
  also have ⊢ [Pop],[] [::] [?τ_c, ?τ_2]  by(simp add:wt-Pop)
  also have (?τ # ?τs_e @ [?τ_e,?τ_1] @ ?τs_c) @ [?τ_c,?τ_2,?τ_1,?τ'] = ?τs by simp
  also let ?go = Goto (-int(?n_c+?n_e+2))
  have P ⊢ ?τ_2 ≤' ?τ by(fastforce intro: ty_i'-antimono simp: hyperset-defs)
  hence P,T_r,mxs,size ?τs,[] ⊢ ?go,?n_e+?n_c+2 :: ?τs
    by(simp add: wt-Goto split: nat-diff-split)
  also have ?τs = (?τ # ?τs_e @ [?τ_e,?τ_1] @ ?τs_c @ [?τ_c, ?τ_2]) @ [?τ_1, ?τ']
    by simp
  also have ⊢ [Push Unit],[] [::] [?τ_1,?τ']
    using While.prems max-stack1[of c] by(auto simp add:wt-Push)
  finally show ?case using wtc wte
    by (simp add:after-def)
next
  case (Cond E A ST e e_1 e_2)
  obtain T_1 T_2 where wte: P,E ⊢1 e :: Boolean
    and wt_1: P,E ⊢1 e_1 :: T_1 and wt_2: P,E ⊢1 e_2 :: T_2
    and sub_1: P ⊢ T_1 ≤ T and sub_2: P ⊢ T_2 ≤ T
    using Cond by(auto dest: is-lub-upper)
  have [simp]: ty E (if (e) e_1 else e_2) = T using Cond by simp
  let ?A_0 = A ⊔ 𝒜 e let ?A_2 = ?A_0 ⊔ 𝒜 e_2 let ?A_1 = ?A_0 ⊔ 𝒜 e_1
  let ?A' = ?A_0 ⊔ 𝒜 e_1 ⊓ 𝒜 e_2
  let ?τ_2 = ty_i' ST E ?A_0 let ?τ' = ty_i' (T#ST) E ?A'
  let ?τs_2 = compT E ?A_0 ST e_2
  have PROP ?P e_2 E T_2 ?A_0 ST by fact
  hence ⊢ compE2 e_2, compxE2 e_2 0 (size ST) [::] (?τ_2#?τs_2) @ [ty_i' (T_2#ST) E ?A_2]
    using Cond.prems wt_2 by(auto simp add:after-def)
  also have P ⊢ ty_i' (T_2#ST) E ?A_2 ≤' ?τ' using sub_2
    by(auto simp add: hyperset-defs ty_i'-def intro!: ty_l-antimono)
  also
  let ?τ_3 = ty_i' (T_1 # ST) E ?A_1
  let ?g_2 = Goto(int (size (compE2 e_2) + 1))
  from sub_1 have P,T_r,mxs,size(compE2 e_2)+2,[] ⊢ ?g_2,0 :: ?τ_3#(?τ_2#?τs_2)@[?τ']
    by(cases length (compE2 e_2))
      (auto simp: hyperset-defs wt-defs nth-Cons ty_i'-def neq-Nil-conv
            split:nat.split intro!: ty_l-antimono)
  also let ?τs_1 = compT E ?A_0 ST e_1
  have PROP ?P e_1 E T_1 ?A_0 ST by fact
  hence ⊢ compE2 e_1,compxE2 e_1 0 (size ST) [::] ?τ_2 # ?τs_1 @ [?τ_3]
    using Cond.prems wt_1 by(auto simp add:after-def)
  also
  let ?τs_12 = ?τ_2 # ?τs_1 @ ?τ_3 # (?τ_2 # ?τs_2) @ [?τ']
  let ?τ_1 = ty_i' (Boolean#ST) E ?A_0
  let ?g_1 = IfFalse(int (size (compE2 e_1) + 2))
  let ?code = compE2 e_1 @ ?g_2 # compE2 e_2
  have ⊢ [?g_1],[] [::] [?τ_1] @ ?τs_12
    by(simp add: wt-IfFalse nat-add-distrib split:nat-diff-split)
```

also (*wt-instrs-ext2*) **have** $[?\tau_1] @ ?\tau s_{12} = ?\tau_1 \# ?\tau s_{12}$ **by** *simp* **also**
let $?\tau = ty_i'\ ST\ E\ A$
have *PROP ?P e E Boolean A ST* **by** *fact*
hence $\vdash$ *compE2 e, compxE2 e 0* (*size ST*) [::] $?\tau \#$ *compT E A ST e* @ $[?\tau_1]$
using *Cond.prems wte* **by**(*auto simp add*:*after-def*)
finally show *?case* **using** *wte* wt_1 wt_2 **by**(*simp add*:*after-def hyperUn-assoc*)
next
case (*Call E A ST e M es*)
from ‹*P*,*E* $\vdash$*1* *e*•*M*(*es*) :: *T*›
obtain *U C D Ts m Ts'*
where *C*: *P*,*E* $\vdash$*1* *e* :: *U*
and *icto*: *class-type-of'* *U* = $\lfloor C \rfloor$
and *method*: *P* $\vdash$ *C sees M*:*Ts* $\rightarrow$ *T* = *m in D*
and *wtes*: *P*,*E* $\vdash$*1* *es* [::] *Ts'* **and** *subs*: *P* $\vdash$ *Ts'* [$\leq$] *Ts*
by(*cases*) *auto*
from *wtes* **have** *same-size*: *size es* = *size Ts'* **by**(*rule WTs1-same-size*)
let $?A_0 = A \sqcup \mathcal{A}\ e$ **let** $?A_1 = ?A_0 \sqcup \mathcal{A}s\ es$
let $?\tau = ty_i'\ ST\ E\ A$ **let** $?\tau s_e =$ *compT E A ST e*
let $?\tau_e = ty_i'\ (U \# ST)\ E\ ?A_0$
let $?\tau s_{es} =$ *compTs E* $?A_0$ $(U \# ST)$ *es*
let $?\tau_1 = ty_i'\ (rev\ Ts' @ U \# ST)\ E\ ?A_1$
let $?\tau' = ty_i'\ (T \# ST)\ E\ ?A_1$
have $\vdash$ [*Invoke M* (*size es*)],[] [::] $[?\tau_1, ?\tau']$
by(*rule wt-Invoke*[*OF same-size icto method subs*])
also
from *Call.hyps*(*2*)[*of Ts'*] *Call.prems wtes C*
have $\vdash$ *compEs2 es*,*compxEs2 es 0* (*size ST*+*1*) [::] $?\tau_e \# ?\tau s_{es}$
last $(?\tau_e \# ?\tau s_{es}) = ?\tau_1$
by(*auto simp add*:*after-def*)
also have $(?\tau_e \# ?\tau s_{es}) @ [?\tau'] = ?\tau_e \# ?\tau s_{es} @ [?\tau']$ **by** *simp*
also have $\vdash$ *compE2 e*,*compxE2 e 0* (*size ST*) [::] $?\tau \# ?\tau s_e @ [?\tau_e]$
using *Call C* **by**(*auto simp add*:*after-def*)
finally show *?case* **using** *Call.prems C*
by(*simp add*:*after-def hyperUn-assoc shift-compxEs2 stack-xlift-compxEs2 del*: *compxEs2-stack-xlift-convs compxEs2-size-convs*)
next
case *Seq* **thus** *?case*
by(*auto simp*:*after-def*)
(*fastforce simp*:*wt-Push wt-Pop hyperUn-assoc*
intro:*wt-instrs-app2 wt-instrs-Cons*)
next
case (*NewArray E A ST Ta e*)
from ‹*P*,*E* $\vdash$*1* *newA Ta*$\lfloor e \rceil$ :: *T*›
have $\vdash$ [*NewArray Ta*], [] [::] $[ty_i'\ (Integer \# ST)\ E\ (A \sqcup \mathcal{A}\ e),\ ty_i'\ (Ta\lfloor\rceil \# ST)\ E\ (A \sqcup \mathcal{A}\ e)]$
by(*auto simp*:*hyperset-defs* ty_i'*-def wt-defs* ty_l*-def*)
with *NewArray* **show** *?case* **by**(*auto simp*: *after-def intro*: *wt-instrs-app3*)
next
case (*ALen E A ST exp*)
{ fix *T*
have $\vdash$ [*ALength*], [] [::] $[ty_i'\ (T\lfloor\rceil \# ST)\ E\ (A \sqcup \mathcal{A}\ exp),\ ty_i'\ (Integer \# ST)\ E\ (A \sqcup \mathcal{A}\ exp)]$
by(*auto simp*:*hyperset-defs* ty_i'*-def wt-defs* ty_l*-def*) **}**
with *ALen* **show** *?case* **by**(*auto simp add*: *after-def*)(*rule wt-instrs-app2*, *auto*)
next
case (*AAcc E A ST a i*)

from ‹*P*,*E* ⊢*1* *a*⌊*i*⌉ :: *T*› **have** *wta*: *P*,*E* ⊢*1* *a* :: *T*⌊⌉ **and** *wti*: *P*,*E* ⊢*1* *i* :: *Integer* **by** *auto*
let *?A1* = *A* ⊔ 𝒜 *a* **let** *?A2* = *?A1* ⊔ 𝒜 *i*
let *?τ* = *ty*$_i$′ *ST E A* **let** *?τsa* = *compT E A ST a*
let *?τ1* = *ty*$_i$′ (*T*⌊⌉#*ST*) *E ?A1* **let** *?τsi* = *compT E ?A1* (*T*⌊⌉#*ST*) *i*
let *?τ2* = *ty*$_i$′ (*Integer*#*T*⌊⌉#*ST*) *E ?A2* **let** *?τ′* = *ty*$_i$′ (*T*#*ST*) *E ?A2*
have ⊢ [*ALoad*], [] [::] [*?τ2*,*?τ′*] **by**(*auto simp add*: *ty*$_i$′*-def wt-defs*)
also from *AAcc.hyps*(*2*)[*of Integer*] *AAcc.prems wti wta*
have ⊢ *compE2 i*, *compxE2 i 0* (*size ST*+*1*) [::] *?τ1*#*?τsi*@[*?τ2*]
by(*auto simp add*: *after-def*)
also from *wta AAcc* **have** ⊢ *compE2 a*, *compxE2 a 0* (*size ST*) [::] *?τ*#*?τsa*@[*?τ1*]
by(*auto simp add*: *after-def*)
finally show *?case* **using** *wta wti* ‹*P*,*E* ⊢*1* *a*⌊*i*⌉ :: *T*› **by**(*simp add*: *after-def hyperUn-assoc*)
next
case (*AAss E A ST a i e*)
note *wt* = ‹*P*,*E* ⊢*1* *a*⌊*i*⌉ := *e* :: *T*›
then obtain *Ta U* **where** *wta*: *P*,*E* ⊢*1* *a* :: *Ta*⌊⌉ **and** *wti*: *P*,*E* ⊢*1* *i* :: *Integer*
and *wte*: *P*,*E* ⊢*1* *e* :: *U* **and** *U*: *P* ⊢ *U* ≤ *Ta* **and** [*simp*]: *T* = *Void* **by** *auto*
let *?A1* = *A* ⊔ 𝒜 *a* **let** *?A2* = *?A1* ⊔ 𝒜 *i* **let** *?A3* = *?A2* ⊔ 𝒜 *e*
let *?τ* = *ty*$_i$′ *ST E A* **let** *?τsa* = *compT E A ST a*
let *?τ1* = *ty*$_i$′ (*Ta*⌊⌉#*ST*) *E ?A1* **let** *?τsi* = *compT E ?A1* (*Ta*⌊⌉#*ST*) *i*
let *?τ2* = *ty*$_i$′ (*Integer*#*Ta*⌊⌉#*ST*) *E ?A2* **let** *?τse* = *compT E ?A2* (*Integer*#*Ta*⌊⌉#*ST*) *e*
let *?τ3* = *ty*$_i$′ (*U*#*Integer*#*Ta*⌊⌉#*ST*) *E ?A3* **let** *?τ4* = *ty*$_i$′ *ST E ?A3*
let *?τ′* = *ty*$_i$′ (*Void*#*ST*) *E ?A3*
from ‹*length ST* + *max-stack* (*a*⌊*i*⌉ := *e*) ≤ *mxs*›
have ⊢ [*AStore*, *Push Unit*], [] [::] [*?τ3*,*?τ4*,*?τ′*]
by(*auto simp add*: *ty*$_i$′*-def wt-defs nth-Cons split*: *nat.split*)
also from *AAss.hyps*(*3*)[*of U*] *wte AAss.prems wta wti*
have ⊢ *compE2 e*, *compxE2 e 0* (*size ST*+*2*) [::] *?τ2*#*?τse*@[*?τ3*]
by(*auto simp add*: *after-def*)
also from *AAss.hyps*(*2*)[*of Integer*] *wti wta AAss.prems*
have ⊢ *compE2 i*, *compxE2 i 0* (*size ST*+*1*) [::] *?τ1*#*?τsi*@[*?τ2*]
by(*auto simp add*: *after-def*)
also from *wta AAss* **have** ⊢ *compE2 a*, *compxE2 a 0* (*size ST*) [::] *?τ*#*?τsa*@[*?τ1*]
by(*auto simp add*: *after-def*)
finally show *?case* **using** *wta wti wte* ‹*P*,*E* ⊢*1* *a*⌊*i*⌉ := *e* :: *T*›
by(*simp add*: *after-def hyperUn-assoc*)
next
case (*CompareAndSwap E A ST e1 D F e2 e3*)
note *wt* = ‹*P*,*E* ⊢*1* *e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*) :: *T*›
then obtain *T1 T2 T3 C fm T′* **where** [*simp*]: *T* = *Boolean*
and *wt1*: *P*,*E* ⊢*1* *e1* :: *T1 class-type-of*′ *T1* = ⌊*C*⌋ *P* ⊢ *C sees F*:*T′* (*fm*) *in D volatile fm*
and *wt2*: *P*,*E* ⊢*1* *e2* :: *T2 P* ⊢ *T2* ≤ *T′* **and** *wt3*: *P*,*E* ⊢*1* *e3* :: *T3 P* ⊢ *T3* ≤ *T′*
by *auto*
let *?A1* = *A* ⊔ 𝒜 *e1* **let** *?A2* = *?A1* ⊔ 𝒜 *e2* **let** *?A3* = *?A2* ⊔ 𝒜 *e3*
let *?τ* = *ty*$_i$′ *ST E A* **let** *?τs1* = *compT E A ST e1*
let *?τ1* = *ty*$_i$′ (*T1*#*ST*) *E ?A1* **let** *?τs2* = *compT E ?A1* (*T1*#*ST*) *e2*
let *?τ2* = *ty*$_i$′ (*T2*#*T1*#*ST*) *E ?A2* **let** *?τs3* = *compT E ?A2* (*T2*#*T1*#*ST*) *e3*
let *?τ3* = *ty*$_i$′ (*T3*#*T2*#*T1*#*ST*) *E ?A3*
let *?τ′* = *ty*$_i$′ (*Boolean*#*ST*) *E ?A3*
from ‹*length ST* + *max-stack* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) ≤ *mxs*›
have ⊢ [*CAS F D*], [] [::] [*?τ3*,*?τ′*] **using** *wt1 wt2 wt3*
by(*cases T1*)(*auto simp add*: *ty*$_i$′*-def wt-defs nth-Cons split*: *nat.split intro*: *sees-field-idemp widen-trans*[*OF widen-array-object*] *dest*: *sees-field-decl-above*)
also from *CompareAndSwap.hyps*(*3*)[*of T3*] *wt3 CompareAndSwap.prems wt1 wt2*

```
  have ⊢ compE2 e3, compxE2 e3 0 (size ST+2) [::] ?τ2#?τs3@[?τ3]
    by(auto simp add: after-def)
  also from CompareAndSwap.hyps(2)[of T2] wt2 wt1 CompareAndSwap.prems
  have ⊢ compE2 e2, compxE2 e2 0 (size ST+1) [::] ?τ1#?τs2@[?τ2]
    by(auto simp add: after-def)
  also from wt1 CompareAndSwap have ⊢ compE2 e1, compxE2 e1 0 (size ST) [::] ?τ#?τs1@[?τ1]
    by(auto simp add: after-def)
  also have ty E (e1·compareAndSwap(D·F, e2, e3)) = T using wt by(rule ty-def2)
  ultimately show ?case using wt1 wt2 wt3
    by(simp add: after-def hyperUn-assoc)
next
  case (InSynchronized i a exp) thus ?case by auto
qed

end

lemma states-compP [simp]: states (compP f P) mxs mxl = states P mxs mxl
by (simp add: JVM-states-unfold)

lemma [simp]: app_i (i, compP f P, pc, mpc, T, τ) = app_i (i, P, pc, mpc, T, τ)
proof −
  { fix ST LT
    have app_i (i, compP f P, pc, mpc, T, (ST, LT)) = app_i (i, P, pc, mpc, T, (ST, LT))
    proof(cases i)
      case (Invoke M n)
      have ⋀C Ts D. (∃ T m. compP f P ⊢ C sees M: Ts→T = m in D) ⟷ (∃ T m. P ⊢ C sees M:
Ts→T = m in D)
        by(auto dest!: sees-method-compPD dest: sees-method-compP)
      with Invoke show ?thesis by clarsimp
    qed(simp-all) }
  thus ?thesis by(cases τ) simp
qed


lemma [simp]: is-relevant-entry (compP f P) i = is-relevant-entry P i
  apply (rule ext)+
  apply (unfold is-relevant-entry-def)
  apply (cases i)
  apply auto
  done

lemma [simp]: relevant-entries (compP f P) i pc xt = relevant-entries P i pc xt
by (simp add: relevant-entries-def)

lemma [simp]: app i (compP f P) mpc T pc mxl xt τ = app i P mpc T pc mxl xt τ
  apply (simp add: app-def xcpt-app-def eff-def xcpt-eff-def norm-eff-def)
  apply (fastforce simp add: image-def)
  done

lemma [simp]: app i P mpc T pc mxl xt τ ⟹ eff i (compP f P) pc xt τ = eff i P pc xt τ
  apply (clarsimp simp add: eff-def norm-eff-def xcpt-eff-def app-def)
  apply (cases i)
  apply(auto)
  done
```

lemma [*simp*]: *widen* (*compP f P*) = *widen P*
apply (*rule ext*)+
apply (*simp*)
done

lemma [*simp*]: *compP f P* ⊢ $\tau \leq' \tau'$ = *P* ⊢ $\tau \leq' \tau'$
by (*simp add*: *sup-state-opt-def sup-state-def sup-ty-opt-def*)

lemma [*simp*]: *compP f P*,*T*,*mpc*,*mxl*,*xt* ⊢ *i*,*pc* :: τs = *P*,*T*,*mpc*,*mxl*,*xt* ⊢ *i*,*pc* :: τs
by (*simp add*: *wt-instr-def cong*: *conj-cong*)

declare *TC0.compT-sizes*[*simp*] *TC1.ty-def2*[*OF TC1.intro*, *simp*]

lemma *compT-method*:
fixes *e* **and** *A* **and** *C* **and** *Ts* **and** mxl_0
defines [*simp*]: *E* ≡ *Class C* # *Ts*
and [*simp*]: *A* ≡ ⌊{..*size Ts*}⌋
and [*simp*]: *A'* ≡ *A* ⊔ $\mathcal{A}$ *e*
and [*simp*]: *mxs* ≡ *max-stack e*
and [*simp*]: mxl_0 ≡ *max-vars e*
and [*simp*]: *mxl* ≡ *1* + *size Ts* + mxl_0
assumes *wf-prog*: *wf-prog p P*
shows ⟦ *P*,*E* ⊢*1 e* :: *T*; $\mathcal{D}$ *e A*; $\mathcal{B}$ *e* (*size E*); *set E* ⊆ *types P*; *P* ⊢ *T* ≤ *T'* ⟧ ⟹
wt-method (*compP2 P*) *C Ts T' mxs* mxl_0 (*compE2 e* @ [*Return*]) (*compxE2 e 0 0*)
(*TC0.ty*$_i$' *mxl* [] *E A* # *TC0.compTa P mxl E A* [] *e*)
using *wf-prog*
apply(*simp add*:*wt-method-def TC0.compTa-def TC0.after-def compP2-def compMb2-def*)
apply(*rule conjI*)
apply(*simp add*:*check-types-def TC0.OK-ty*$_i$'*-in-statesI*)
apply(*rule conjI*)
apply(*frule WT1-is-type*[*OF wf-prog*])
apply *simp*
apply(*insert max-stack1*[*of e*])
apply(*fastforce intro*!: *TC0.OK-ty*$_i$'*-in-statesI*)
apply(*erule* (*1*) *TC1.compT-states*[*OF TC1.intro*])
apply *simp*
apply *simp*
apply *simp*
apply *simp*
apply(*rule conjI*)
apply(*fastforce simp add*:*wt-start-def TC0.ty*$_i$'*-def TC0.ty*$_l$*-def list-all2-conv-all-nth nth-Cons split*:*nat.split dest*:*less-antisym*)
apply (*frule* (*1*) *TC3.compT-wt-instrs*[*OF TC3.intro*[*OF TC1.intro*], **where** *ST* = [] **and** *mxs* = *max-stack e* **and** *mxl* = *1* + *size Ts* + *max-vars e*])
apply *simp*
apply *simp*
apply *simp*
apply *simp*
apply *simp*
apply (*clarsimp simp*:*TC2.wt-instrs-def TC0.after-def*)
apply(*rule conjI*)
apply (*fastforce*)
apply(*clarsimp*)

apply(*drule* (*1*) *less-antisym*)
apply(*thin-tac* $\forall x.\ P\ x$ **for** *P*)
apply(*clarsimp simp*:*TC2.wt-defs xcpt-app-pcs xcpt-eff-pcs TC0.ty$_i$'-def*)
done

definition *compTP* :: *'addr J1-prog* $\Rightarrow$ *ty$_P$*
where
compTP P C M $\equiv$
let (*D*,*Ts*,*T*,*meth*) = *method P C M*;
e = *the meth*;
E = *Class C* # *Ts*;
A = $\lfloor${..*size Ts*}$\rfloor$;
mxl = *1* + *size Ts* + *max-vars e*
in (*TC0.ty$_i$' mxl* [] *E A* # *TC0.compTa P mxl E A* [] *e*)

theorem *wt-compTP-compP2*:
wf-J1-prog P $\Longrightarrow$ *wf-jvm-prog$_{compTP\ P}$* (*compP2 P*)
apply (*simp add*: *wf-jvm-prog-phi-def compP2-def compMb2-def*)
apply (*rule wf-prog-compPI*)
prefer *2* **apply** *assumption*
apply (*clarsimp simp add*: *wf-mdecl-def*)
apply (*simp add*: *compTP-def*)
apply (*rule compT-method* [*simplified compP2-def compMb2-def*, *simplified*])
apply *assumption+*
apply (*drule* (*1*) *sees-wf-mdecl*)
apply (*simp add*: *wf-mdecl-def*)
apply (*fastforce intro*: *sees-method-is-class*)
apply *assumption*
done

theorem *wt-compP2*:
wf-J1-prog P $\Longrightarrow$ *wf-jvm-prog* (*compP2 P*)
by(*auto simp add*: *wf-jvm-prog-def intro*: *wt-compTP-compP2*)

end

7.14 Unobservable steps for the JVM

theory *JVMTau* **imports**
TypeComp
../JVM/JVMThreaded
../Framework/FWLTS
begin

declare *nth-append* [*simp del*]
declare *Listn.lesub-list-impl-same-size*[*simp del*]
declare *listE-length* [*simp del*]

declare *match-ex-table-append-not-pcs*[*simp del*]
outside-pcs-not-matches-entry [*simp del*]

outside-pcs-compxE2-not-matches-entry [*simp del*]
outside-pcs-compxEs2-not-matches-entry [*simp del*]

context *JVM-heap-base* **begin**

primrec $\tau instr :: 'm\ prog \Rightarrow 'heap \Rightarrow 'addr\ val\ list \Rightarrow 'addr\ instr \Rightarrow bool$
where
$\tau instr\ P\ h\ stk\ (Load\ n) = True$
$| \tau instr\ P\ h\ stk\ (Store\ n) = True$
$| \tau instr\ P\ h\ stk\ (Push\ v) = True$
$| \tau instr\ P\ h\ stk\ (New\ C) = False$
$| \tau instr\ P\ h\ stk\ (NewArray\ T) = False$
$| \tau instr\ P\ h\ stk\ ALoad = False$
$| \tau instr\ P\ h\ stk\ AStore = False$
$| \tau instr\ P\ h\ stk\ ALength = False$
$| \tau instr\ P\ h\ stk\ (Getfield\ F\ D) = False$
$| \tau instr\ P\ h\ stk\ (Putfield\ F\ D) = False$
$| \tau instr\ P\ h\ stk\ (CAS\ F\ D) = False$
$| \tau instr\ P\ h\ stk\ (Checkcast\ T) = True$
$| \tau instr\ P\ h\ stk\ (Instanceof\ T) = True$
$| \tau instr\ P\ h\ stk\ (Invoke\ M\ n) =$
$(n < length\ stk \wedge$
$(stk\ !\ n = Null \vee$
$(\forall T\ Ts\ Tr\ D.\ typeof\text{-}addr\ h\ (the\text{-}Addr\ (stk\ !\ n)) = \lfloor T \rfloor \longrightarrow P \vdash class\text{-}type\text{-}of\ T\ sees\ M{:}Ts \rightarrow Tr = Native\ in\ D \longrightarrow \tau external\text{-}defs\ D\ M)))$
$| \tau instr\ P\ h\ stk\ Return = True$
$| \tau instr\ P\ h\ stk\ Pop = True$
$| \tau instr\ P\ h\ stk\ Dup = True$
$| \tau instr\ P\ h\ stk\ Swap = True$
$| \tau instr\ P\ h\ stk\ (BinOpInstr\ bop) = True$
$| \tau instr\ P\ h\ stk\ (Goto\ i) = True$
$| \tau instr\ P\ h\ stk\ (IfFalse\ i) = True$
$| \tau instr\ P\ h\ stk\ ThrowExc = True$
$| \tau instr\ P\ h\ stk\ MEnter = False$
$| \tau instr\ P\ h\ stk\ MExit = False$

inductive $\tau move2 :: 'm\ prog \Rightarrow 'heap \Rightarrow 'addr\ val\ list \Rightarrow 'addr\ expr1 \Rightarrow nat \Rightarrow 'addr\ option \Rightarrow bool$
and $\tau moves2 :: 'm\ prog \Rightarrow 'heap \Rightarrow 'addr\ val\ list \Rightarrow 'addr\ expr1\ list \Rightarrow nat \Rightarrow 'addr\ option \Rightarrow bool$
for $P :: 'm\ prog$ **and** $h :: 'heap$ **and** $stk :: 'addr\ val\ list$
where
$\tau move2xcp: pc < length\ (compE2\ e) \Longrightarrow \tau move2\ P\ h\ stk\ e\ pc\ \lfloor xcp \rfloor$

$| \tau move2NewArray: \tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (newA\ T\lfloor e \rceil)\ pc\ xcp$

$| \tau move2Cast: \tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (Cast\ T\ e)\ pc\ xcp$
$| \tau move2CastRed: \tau move2\ P\ h\ stk\ (Cast\ T\ e)\ (length\ (compE2\ e))\ None$

$| \tau move2InstanceOf: \tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (e\ instanceof\ T)\ pc\ xcp$
$| \tau move2InstanceOfRed: \tau move2\ P\ h\ stk\ (e\ instanceof\ T)\ (length\ (compE2\ e))\ None$

$| \tau move2Val: \tau move2\ P\ h\ stk\ (Val\ v)\ 0\ None$

$| \tau move2BinOp1:$
$\tau move2\ P\ h\ stk\ e1\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (e1 \guillemotleft bop \guillemotright e2)\ pc\ xcp$

| $\tau move2BinOp2$:
$\tau move2\ P\ h\ stk\ e2\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (e1 \mathbin{«} bop \mathbin{»} e2)\ (length\ (compE2\ e1) + pc)\ xcp$
| $\tau move2BinOp$:
$\tau move2\ P\ h\ stk\ (e1 \mathbin{«} bop \mathbin{»} e2)\ (length\ (compE2\ e1) + length\ (compE2\ e2))\ None$

| $\tau move2Var$:
$\tau move2\ P\ h\ stk\ (Var\ V)\ 0\ None$

| $\tau move2LAss$:
$\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (V{:=}e)\ pc\ xcp$
| $\tau move2LAssRed1$:
$\tau move2\ P\ h\ stk\ (V{:=}e)\ (length\ (compE2\ e))\ None$
| $\tau move2LAssRed2$: $\tau move2\ P\ h\ stk\ (V{:=}e)\ (Suc\ (length\ (compE2\ e)))\ None$

| $\tau move2AAcc1$: $\tau move2\ P\ h\ stk\ a\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (a\lfloor i \rceil)\ pc\ xcp$
| $\tau move2AAcc2$: $\tau move2\ P\ h\ stk\ i\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (a\lfloor i \rceil)\ (length\ (compE2\ a) + pc)\ xcp$

| $\tau move2AAss1$: $\tau move2\ P\ h\ stk\ a\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (a\lfloor i \rceil := e)\ pc\ xcp$
| $\tau move2AAss2$: $\tau move2\ P\ h\ stk\ i\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (a\lfloor i \rceil := e)\ (length\ (compE2\ a) + pc)\ xcp$
| $\tau move2AAss3$: $\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (a\lfloor i \rceil := e)\ (length\ (compE2\ a) + length\ (compE2\ i) + pc)\ xcp$
| $\tau move2AAssRed$: $\tau move2\ P\ h\ stk\ (a\lfloor i \rceil := e)\ (Suc\ (length\ (compE2\ a) + length\ (compE2\ i) + length\ (compE2\ e)))\ None$

| $\tau move2ALength$: $\tau move2\ P\ h\ stk\ a\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (a{\cdot}length)\ pc\ xcp$

| $\tau move2FAcc$: $\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (e{\cdot}F\{D\})\ pc\ xcp$

| $\tau move2FAss1$: $\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (e{\cdot}F\{D\} := e')\ pc\ xcp$
| $\tau move2FAss2$: $\tau move2\ P\ h\ stk\ e'\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (e{\cdot}F\{D\} := e')\ (length\ (compE2\ e) + pc)\ xcp$
| $\tau move2FAssRed$: $\tau move2\ P\ h\ stk\ (e{\cdot}F\{D\} := e')\ (Suc\ (length\ (compE2\ e) + length\ (compE2\ e')))\ None$

| $\tau move2CAS1$: $\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e''))\ pc\ xcp$
| $\tau move2CAS2$: $\tau move2\ P\ h\ stk\ e'\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e''))\ (length\ (compE2\ e) + pc)\ xcp$
| $\tau move2CAS3$: $\tau move2\ P\ h\ stk\ e''\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e''))\ (length\ (compE2\ e) + length\ (compE2\ e') + pc)\ xcp$

| $\tau move2CallObj$:
$\tau move2\ P\ h\ stk\ obj\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (obj{\cdot}M(ps))\ pc\ xcp$
| $\tau move2CallParams$:
$\tau moves2\ P\ h\ stk\ ps\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (obj{\cdot}M(ps))\ (length\ (compE2\ obj) + pc)\ xcp$
| $\tau move2Call$:
$[\![\ length\ ps < length\ stk;$
$stk\ !\ length\ ps = Null \vee$
$(\forall\ T\ Ts\ Tr\ D.\ typeof\text{-}addr\ h\ (the\text{-}Addr\ (stk\ !\ length\ ps)) = \lfloor T \rfloor \longrightarrow P \vdash class\text{-}type\text{-}of\ T\ sees\ M{:}Ts{\to}Tr = Native\ in\ D \longrightarrow \tau external\text{-}defs\ D\ M)]\!]$
$\Longrightarrow \tau move2\ P\ h\ stk\ (obj{\cdot}M(ps))\ (length\ (compE2\ obj) + length\ (compEs2\ ps))\ None$

| $\tau move2BlockSome1$:

$\tau move2\ P\ h\ stk\ \{V{:}T{=}\lfloor v\rfloor;\ e\}\ 0\ None$
| $\tau move2BlockSome2$:
$\tau move2\ P\ h\ stk\ \{V{:}T{=}\lfloor v\rfloor;\ e\}\ (Suc\ 0)\ None$
| $\tau move2BlockSome$:
$\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ \{V{:}T{=}\lfloor v\rfloor;\ e\}\ (Suc\ (Suc\ pc))\ xcp$
| $\tau move2BlockNone$:
$\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ \{V{:}T{=}None;\ e\}\ pc\ xcp$

| $\tau move2Sync1$:
$\tau move2\ P\ h\ stk\ o'\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (sync_V\ (o')\ e)\ pc\ xcp$
| $\tau move2Sync2$:
$\tau move2\ P\ h\ stk\ (sync_V\ (o')\ e)\ (length\ (compE2\ o'))\ None$
| $\tau move2Sync3$:
$\tau move2\ P\ h\ stk\ (sync_V\ (o')\ e)\ (Suc\ (length\ (compE2\ o')))\ None$
| $\tau move2Sync4$:
$\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (sync_V\ (o')\ e)\ (Suc\ (Suc\ (Suc\ (length\ (compE2\ o') + pc))))\ xcp$
| $\tau move2Sync5$:
$\tau move2\ P\ h\ stk\ (sync_V\ (o')\ e)\ (Suc\ (Suc\ (Suc\ (length\ (compE2\ o') + length\ (compE2\ e)))))\ None$
| $\tau move2Sync6$:
$\tau move2\ P\ h\ stk\ (sync_V\ (o')\ e)\ (5 + length\ (compE2\ o') + length\ (compE2\ e))\ None$
| $\tau move2Sync7$:
$\tau move2\ P\ h\ stk\ (sync_V\ (o')\ e)\ (6 + length\ (compE2\ o') + length\ (compE2\ e))\ None$
| $\tau move2Sync8$:
$\tau move2\ P\ h\ stk\ (sync_V\ (o')\ e)\ (8 + length\ (compE2\ o') + length\ (compE2\ e))\ None$

| $\tau move2InSync$: $\tau move2\ P\ h\ stk\ (insync_V\ (a)\ e)\ 0\ None$

| $\tau move2Seq1$:
$\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (e;;e')\ pc\ xcp$
| $\tau move2SeqRed$:
$\tau move2\ P\ h\ stk\ (e;;e')\ (length\ (compE2\ e))\ None$
| $\tau move2Seq2$:
$\tau move2\ P\ h\ stk\ e'\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (e;;e')\ (Suc\ (length\ (compE2\ e) + pc))\ xcp$

| $\tau move2Cond$:
$\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (if\ (e)\ e1\ else\ e2)\ pc\ xcp$
| $\tau move2CondRed$:
$\tau move2\ P\ h\ stk\ (if\ (e)\ e1\ else\ e2)\ (length\ (compE2\ e))\ None$
| $\tau move2CondThen$:
$\tau move2\ P\ h\ stk\ e1\ pc\ xcp$
$\Longrightarrow \tau move2\ P\ h\ stk\ (if\ (e)\ e1\ else\ e2)\ (Suc\ (length\ (compE2\ e) + pc))\ xcp$
| $\tau move2CondThenExit$:
$\tau move2\ P\ h\ stk\ (if\ (e)\ e1\ else\ e2)\ (Suc\ (length\ (compE2\ e) + length\ (compE2\ e1)))\ None$
| $\tau move2CondElse$:
$\tau move2\ P\ h\ stk\ e2\ pc\ xcp$
$\Longrightarrow \tau move2\ P\ h\ stk\ (if\ (e)\ e1\ else\ e2)\ (Suc\ (Suc\ (length\ (compE2\ e) + length\ (compE2\ e1) + pc)))\ xcp$

| $\tau move2While1$:
$\tau move2\ P\ h\ stk\ c\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (while\ (c)\ e)\ pc\ xcp$
| $\tau move2While2$:
$\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (while\ (c)\ e)\ (Suc\ (length\ (compE2\ c) + pc))\ xcp$

| $\tau move2While3$: — Jump back to condition
$\tau move2\ P\ h\ stk\ (while\ (c)\ e)\ (Suc\ (Suc\ (length\ (compE2\ c) + length\ (compE2\ e))))\ None$
| $\tau move2While4$: — last instruction: Push Unit
$\tau move2\ P\ h\ stk\ (while\ (c)\ e)\ (Suc\ (Suc\ (Suc\ (length\ (compE2\ c) + length\ (compE2\ e)))))\ None$
| $\tau move2While5$: — IfFalse instruction
$\tau move2\ P\ h\ stk\ (while\ (c)\ e)\ (length\ (compE2\ c))\ None$
| $\tau move2While6$: — Pop instruction
$\tau move2\ P\ h\ stk\ (while\ (c)\ e)\ (Suc\ (length\ (compE2\ c) + length\ (compE2\ e)))\ None$

| $\tau move2Throw1$:
$\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (throw\ e)\ pc\ xcp$
| $\tau move2Throw2$:
$\tau move2\ P\ h\ stk\ (throw\ e)\ (length\ (compE2\ e))\ None$

| $\tau move2Try1$:
$\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (try\ e\ catch(C\ V)\ e')\ pc\ xcp$
| $\tau move2TryJump$:
$\tau move2\ P\ h\ stk\ (try\ e\ catch(C\ V)\ e')\ (length\ (compE2\ e))\ None$
| $\tau move2TryCatch2$:
$\tau move2\ P\ h\ stk\ (try\ e\ catch(C\ V)\ e')\ (Suc\ (length\ (compE2\ e)))\ None$
| $\tau move2Try2$:
$\tau move2\ P\ h\ stk\ \{V{:}T{=}None;\ e'\}\ pc\ xcp$
$\Longrightarrow \tau move2\ P\ h\ stk\ (try\ e\ catch(C\ V)\ e')\ (Suc\ (Suc\ (length\ (compE2\ e) + pc)))\ xcp$

| $\tau moves2Hd$:
$\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau moves2\ P\ h\ stk\ (e\ \#\ es)\ pc\ xcp$
| $\tau moves2Tl$:
$\tau moves2\ P\ h\ stk\ es\ pc\ xcp \Longrightarrow \tau moves2\ P\ h\ stk\ (e\ \#\ es)\ (length\ (compE2\ e) + pc)\ xcp$

inductive-cases $\tau move2\text{-}cases$:
$\tau move2\ P\ h\ stk\ (new\ C)\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (newA\ T\lfloor e\rceil)\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (Cast\ T\ e)\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (e\ instanceof\ T)\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (Val\ v)\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (Var\ V)\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (e1\,«bop»\,e2)\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (V := e)\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (e1\lfloor e2\rceil)\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (e1\lfloor e2\rceil := e3)\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (e1{\cdot}length)\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (e1{\cdot}F\{D\})\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (e1{\cdot}F\{D\} := e3)\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (e1{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3))\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (e{\cdot}M(ps))\ pc\ xcp$
$\tau move2\ P\ h\ stk\ \{V{:}T{=}vo;\ e\}\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (sync_V\ (e1)\ e2)\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (e1;;e2)\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (if\ (e1)\ e2\ else\ e3)\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (while\ (e1)\ e2)\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (try\ e1\ catch(C\ V)\ e2)\ pc\ xcp$
$\tau move2\ P\ h\ stk\ (throw\ e)\ pc\ xcp$

lemma $\tau moves2xcp$: $pc < length\ (compEs2\ es) \Longrightarrow \tau moves2\ P\ h\ stk\ es\ pc\ \lfloor xcp \rfloor$

proof(*induct es arbitrary*: *pc*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons e es*)
note *IH* = ‹⋀*pc*. *pc* < *length* (*compEs2 es*) ⟹ τ*moves2 P h stk es pc* ⌊*xcp*⌋›
note *pc* = ‹*pc* < *length* (*compEs2* (*e* # *es*))›
show *?case*
proof(*cases pc* < *length* (*compE2 e*))
case *True*
thus *?thesis* **by**(*auto intro*: τ*moves2Hd* τ*move2xcp*)
next
case *False*
with *pc IH*[*of pc* − *length* (*compE2 e*)]
have τ*moves2 P h stk es* (*pc* − *length* (*compE2 e*)) ⌊*xcp*⌋ **by**(*simp*)
hence τ*moves2 P h stk* (*e* # *es*) (*length* (*compE2 e*) + (*pc* − *length* (*compE2 e*))) ⌊*xcp*⌋
by(*rule* τ*moves2Tl*)
with *False* **show** *?thesis* **by** *simp*
qed
qed

lemma τ*move2-intros′*:
shows τ*move2CastRed′*: *pc* = *length* (*compE2 e*) ⟹ τ*move2 P h stk* (*Cast T e*) *pc None*
and τ*move2InstanceOfRed′*: *pc* = *length* (*compE2 e*) ⟹ τ*move2 P h stk* (*e instanceof T*) *pc None*
and τ*move2BinOp2′*: ⟦ τ*move2 P h stk e2 pc xcp*; *pc′* = *length* (*compE2 e1*) + *pc* ⟧ ⟹ τ*move2 P h stk* (*e1*«*bop*»*e2*) *pc′ xcp*
and τ*move2BinOp′*: *pc* = *length* (*compE2 e1*) + *length* (*compE2 e2*) ⟹ τ*move2 P h stk* (*e1*«*bop*»*e2*) *pc None*
and τ*move2LAssRed1′*: *pc* = *length* (*compE2 e*) ⟹ τ*move2 P h stk* (*V*:=*e*) *pc None*
and τ*move2LAssRed2′*: *pc* = *Suc* (*length* (*compE2 e*)) ⟹ τ*move2 P h stk* (*V*:=*e*) *pc None*
and τ*move2AAcc2′*: ⟦ τ*move2 P h stk i pc xcp*; *pc′* = *length* (*compE2 a*) + *pc* ⟧ ⟹ τ*move2 P h stk* (*a*⌊*i*⌉) *pc′ xcp*
and τ*move2AAss2′*: ⟦ τ*move2 P h stk i pc xcp*; *pc′* = *length* (*compE2 a*) + *pc* ⟧ ⟹ τ*move2 P h stk* (*a*⌊*i*⌉ := *e*) *pc′ xcp*
and τ*move2AAss3′*: ⟦ τ*move2 P h stk e pc xcp*; *pc′* = *length* (*compE2 a*) + *length* (*compE2 i*) + *pc* ⟧ ⟹ τ*move2 P h stk* (*a*⌊*i*⌉ := *e*) *pc′ xcp*
and τ*move2AAssRed′*: *pc* = *Suc* (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)) ⟹ τ*move2 P h stk* (*a*⌊*i*⌉ := *e*) *pc None*
and τ*move2FAss2′*: ⟦ τ*move2 P h stk e′ pc xcp*; *pc′* = *length* (*compE2 e*) + *pc* ⟧ ⟹ τ*move2 P h stk* (*e*·*F*{*D*} := *e′*) *pc′ xcp*
and τ*move2FAssRed′*: *pc* = *Suc* (*length* (*compE2 e*) + *length* (*compE2 e′*)) ⟹ τ*move2 P h stk* (*e*·*F*{*D*} := *e′*) *pc None*
and τ*move2CAS2′*: ⟦ τ*move2 P h stk e2 pc xcp*; *pc′* = *length* (*compE2 e1*) + *pc* ⟧ ⟹ τ*move2 P h stk* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *pc′ xcp*
and τ*move2CAS3′*: ⟦ τ*move2 P h stk e3 pc xcp*; *pc′* = *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc* ⟧ ⟹ τ*move2 P h stk* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *pc′ xcp*
and τ*move2CallParams′*: ⟦ τ*moves2 P h stk ps pc xcp*; *pc′* = *length* (*compE2 obj*) + *pc* ⟧ ⟹ τ*move2 P h stk* (*obj*·*M*(*ps*)) *pc′ xcp*
and τ*move2Call′*: ⟦ *pc* = *length* (*compE2 obj*) + *length* (*compEs2 ps*); *length ps* < *length stk*;
stk ! *length ps* = *Null* ∨
(∀ *T Ts Tr D*. *typeof-addr h* (*the-Addr* (*stk* ! *length ps*)) = ⌊*T*⌋ ⟶ *P* ⊢ *class-type-of T sees M*:*Ts*→*Tr* = *Native in D* ⟶ τ*external-defs D M*) ⟧
⟹ τ*move2 P h stk* (*obj*·*M*(*ps*)) *pc None*
and τ*move2BlockSome2*: *pc* = *Suc* 0 ⟹ τ*move2 P h stk* {*V*:*T*=⌊*v*⌋; *e*} *pc None*
and τ*move2BlockSome′*: ⟦ τ*move2 P h stk e pc xcp*; *pc′* = *Suc* (*Suc pc*) ⟧ ⟹ τ*move2 P h stk*

$\{V{:}T{=}\lfloor v\rfloor;\ e\}\ pc'\ xcp$
and $\tau move2Sync2'$: $pc = length\ (compE2\ o') \Longrightarrow \tau move2\ P\ h\ stk\ (sync_V\ (o')\ e)\ pc\ None$
and $\tau move2Sync3'$: $pc = Suc\ (length\ (compE2\ o')) \Longrightarrow \tau move2\ P\ h\ stk\ (sync_V\ (o')\ e)\ pc\ None$
and $\tau move2Sync4'$: $[\![\ \tau move2\ P\ h\ stk\ e\ pc\ xcp;\ pc' = Suc\ (Suc\ (Suc\ (length\ (compE2\ o') + pc)))\ ]\!] \Longrightarrow \tau move2\ P\ h\ stk\ (sync_V\ (o')\ e)\ pc'\ xcp$
and $\tau move2Sync5'$: $pc = Suc\ (Suc\ (Suc\ (length\ (compE2\ o') + length\ (compE2\ e)))) \Longrightarrow \tau move2\ P\ h\ stk\ (sync_V\ (o')\ e)\ pc\ None$
and $\tau move2Sync6'$: $pc = 5 + length\ (compE2\ o') + length\ (compE2\ e) \Longrightarrow \tau move2\ P\ h\ stk\ (sync_V\ (o')\ e)\ pc\ None$
and $\tau move2Sync7'$: $pc = 6 + length\ (compE2\ o') + length\ (compE2\ e) \Longrightarrow \tau move2\ P\ h\ stk\ (sync_V\ (o')\ e)\ pc\ None$
and $\tau move2Sync8'$: $pc = 8 + length\ (compE2\ o') + length\ (compE2\ e) \Longrightarrow \tau move2\ P\ h\ stk\ (sync_V\ (o')\ e)\ pc\ None$
and $\tau move2SeqRed'$: $pc = length\ (compE2\ e) \Longrightarrow \tau move2\ P\ h\ stk\ (e;;e')\ pc\ None$
and $\tau move2Seq2'$: $[\![\ \tau move2\ P\ h\ stk\ e'\ pc\ xcp;\ pc' = Suc\ (length\ (compE2\ e) + pc)\ ]\!] \Longrightarrow \tau move2\ P\ h\ stk\ (e;;e')\ pc'\ xcp$
and $\tau move2CondRed'$: $pc = length\ (compE2\ e) \Longrightarrow \tau move2\ P\ h\ stk\ (if\ (e)\ e1\ else\ e2)\ pc\ None$
and $\tau move2CondThen'$: $[\![\ \tau move2\ P\ h\ stk\ e1\ pc\ xcp;\ pc' = Suc\ (length\ (compE2\ e) + pc)\ ]\!] \Longrightarrow \tau move2\ P\ h\ stk\ (if\ (e)\ e1\ else\ e2)\ pc'\ xcp$
and $\tau move2CondThenExit'$: $pc = Suc\ (length\ (compE2\ e) + length\ (compE2\ e1)) \Longrightarrow \tau move2\ P\ h\ stk\ (if\ (e)\ e1\ else\ e2)\ pc\ None$
and $\tau move2CondElse'$: $[\![\ \tau move2\ P\ h\ stk\ e2\ pc\ xcp;\ pc' = Suc\ (Suc\ (length\ (compE2\ e) + length\ (compE2\ e1) + pc))\ ]\!]$
$\Longrightarrow \tau move2\ P\ h\ stk\ (if\ (e)\ e1\ else\ e2)\ pc'\ xcp$
and $\tau move2While2'$: $[\![\ \tau move2\ P\ h\ stk\ e\ pc\ xcp;\ pc' = Suc\ (length\ (compE2\ c) + pc)\ ]\!] \Longrightarrow \tau move2\ P\ h\ stk\ (while\ (c)\ e)\ pc'\ xcp$
and $\tau move2While3'$: $pc = Suc\ (Suc\ (length\ (compE2\ c) + length\ (compE2\ e))) \Longrightarrow \tau move2\ P\ h\ stk\ (while\ (c)\ e)\ pc\ None$
and $\tau move2While4'$: $pc = Suc\ (Suc\ (Suc\ (length\ (compE2\ c) + length\ (compE2\ e)))) \Longrightarrow \tau move2\ P\ h\ stk\ (while\ (c)\ e)\ pc\ None$
and $\tau move2While5'$: $pc = length\ (compE2\ c) \Longrightarrow \tau move2\ P\ h\ stk\ (while\ (c)\ e)\ pc\ None$
and $\tau move2While6'$: $pc = Suc\ (length\ (compE2\ c) + length\ (compE2\ e)) \Longrightarrow \tau move2\ P\ h\ stk\ (while\ (c)\ e)\ pc\ None$
and $\tau move2Throw2'$: $pc = length\ (compE2\ e) \Longrightarrow \tau move2\ P\ h\ stk\ (throw\ e)\ pc\ None$
and $\tau move2TryJump'$: $pc = length\ (compE2\ e) \Longrightarrow \tau move2\ P\ h\ stk\ (try\ e\ catch(C\ V)\ e')\ pc\ None$
and $\tau move2TryCatch2'$: $pc = Suc\ (length\ (compE2\ e)) \Longrightarrow \tau move2\ P\ h\ stk\ (try\ e\ catch(C\ V)\ e')\ pc\ None$
and $\tau move2Try2'$: $[\![\ \tau move2\ P\ h\ stk\ \{V{:}T{=}None;\ e'\}\ pc\ xcp;\ pc' = Suc\ (Suc\ (length\ (compE2\ e) + pc))\ ]\!]$
$\Longrightarrow \tau move2\ P\ h\ stk\ (try\ e\ catch(C\ V)\ e')\ pc'\ xcp$
and $\tau moves2Tl'$: $[\![\ \tau moves2\ P\ h\ stk\ es\ pc\ xcp;\ pc' = length\ (compE2\ e) + pc\ ]\!] \Longrightarrow \tau moves2\ P\ h\ stk\ (e\ \#\ es)\ pc'\ xcp$
apply(*blast intro*: $\tau move2$-$\tau moves2.intros$)+
done

lemma $\tau move2$-*iff*: $\tau move2\ P\ h\ stk\ e\ pc\ xcp \longleftrightarrow pc < length\ (compE2\ e) \wedge (xcp = None \longrightarrow \tau instr\ P\ h\ stk\ (compE2\ e\ !\ pc))$ (**is** *?lhs1* $\longleftrightarrow$ *?rhs1*)
and $\tau moves2$-*iff*: $\tau moves2\ P\ h\ stk\ es\ pc\ xcp \longleftrightarrow pc < length\ (compEs2\ es) \wedge (xcp = None \longrightarrow \tau instr\ P\ h\ stk\ (compEs2\ es\ !\ pc))$ (**is** *?lhs2* $\longleftrightarrow$ *?rhs2*)
proof −
have *rhs1lhs1*: $[\![\ \tau instr\ P\ h\ stk\ (compE2\ e\ !\ pc);\ pc < length\ (compE2\ e)\ ]\!] \Longrightarrow \tau move2\ P\ h\ stk\ e\ pc\ None$
and *rhs2lhs2*: $[\![\ \tau instr\ P\ h\ stk\ (compEs2\ es\ !\ pc);\ pc < length\ (compEs2\ es)\ ]\!] \Longrightarrow \tau moves2\ P\ h\ stk\ es\ pc\ None$

apply(*induct e* **and** *es arbitrary*: *pc* **and** *pc rule*: *compE2.induct compEs2.induct*)
apply(*force intro*: *τmove2-τmoves2.intros τmove2-intros′ simp add*: *nth-append nth-Cons′ not-less-eq split*: *if-split-asm*)+
done

{ assume *pc* < *length* (*compE2 e*) *xcp* ≠ *None*
hence *τmove2 P h stk e pc xcp* **by**(*auto intro*: *τmove2xcp*) **}**
with *rhs1lhs1* **have** *?rhs1* ⟹ *?lhs1* **by**(*cases xcp*) *auto*
moreover {
assume *pc* < *length* (*compEs2 es*) *xcp* ≠ *None*
hence *τmoves2 P h stk es pc xcp* **by**(*auto intro*: *τmoves2xcp*) **}**
with *rhs2lhs2* **have** *?rhs2* ⟹ *?lhs2* **by**(*cases xcp*) *auto*
moreover have *?lhs1* ⟹ *?rhs1* **and** *?lhs2* ⟹ *?rhs2*
by(*induct rule*: *τmove2-τmoves2.inducts*)(*fastforce simp add*: *nth-append eval-nat-numeral*)+
ultimately show *?lhs1* ⟷ *?rhs1 ?lhs2* ⟷ *?rhs2* **by** *blast*+
qed

lemma *τmove2-pc-length-compE2*: *τmove2 P h stk e pc xcp* ⟹ *pc* < *length* (*compE2 e*)
and *τmoves2-pc-length-compEs2*: *τmoves2 P h stk es pc xcp* ⟹ *pc* < *length* (*compEs2 es*)
by(*simp-all add*: *τmove2-iff τmoves2-iff*)

lemma *τmove2-pc-length-compE2-conv*: *pc* ≥ *length* (*compE2 e*) ⟹ ¬ *τmove2 P h stk e pc xcp*
by(*auto dest*: *τmove2-pc-length-compE2*)

lemma *τmoves2-pc-length-compEs2-conv*: *pc* ≥ *length* (*compEs2 es*) ⟹ ¬ *τmoves2 P h stk es pc xcp*
by(*auto dest*: *τmoves2-pc-length-compEs2*)

lemma *τmoves2-append* [*elim*]:
τmoves2 P h stk es pc xcp ⟹ *τmoves2 P h stk* (*es* @ *es′*) *pc xcp*
by(*auto simp add*: *τmoves2-iff nth-append*)

lemma *append-τmoves2*:
τmoves2 P h stk es pc xcp ⟹ *τmoves2 P h stk* (*es′* @ *es*) (*length* (*compEs2 es′*) + *pc*) *xcp*
by(*simp add*: *τmoves2-iff*)

lemma [*dest*]:
shows *τmove2-NewArrayD*: ⟦ *τmove2 P h stk* (*newA T*⌊*e*⌉) *pc xcp*; *pc* < *length* (*compE2 e*) ⟧ ⟹ *τmove2 P h stk e pc xcp*
and *τmove2-CastD*: ⟦ *τmove2 P h stk* (*Cast T e*) *pc xcp*; *pc* < *length* (*compE2 e*) ⟧ ⟹ *τmove2 P h stk e pc xcp*
and *τmove2-InstanceOfD*: ⟦ *τmove2 P h stk* (*e instanceof T*) *pc xcp*; *pc* < *length* (*compE2 e*) ⟧ ⟹ *τmove2 P h stk e pc xcp*
and *τmove2-BinOp1D*: ⟦ *τmove2 P h stk* (*e1* «*bop*» *e2*) *pc′ xcp′*; *pc′* < *length* (*compE2 e1*) ⟧ ⟹ *τmove2 P h stk e1 pc′ xcp′*
and *τmove2-BinOp2D*:
⟦ *τmove2 P h stk* (*e1* «*bop*» *e2*) (*length* (*compE2 e1*) + *pc′*) *xcp′*; *pc′* < *length* (*compE2 e2*) ⟧ ⟹ *τmove2 P h stk e2 pc′ xcp′*
and *τmove2-LAssD*: ⟦ *τmove2 P h stk* (*V* := *e*) *pc xcp*; *pc* < *length* (*compE2 e*) ⟧ ⟹ *τmove2 P h stk e pc xcp*
and *τmove2-AAccD1*: ⟦ *τmove2 P h stk* (*a*⌊*i*⌉) *pc xcp*; *pc* < *length* (*compE2 a*) ⟧ ⟹ *τmove2 P h stk a pc xcp*
and *τmove2-AAccD2*: ⟦ *τmove2 P h stk* (*a*⌊*i*⌉) (*length* (*compE2 a*) + *pc*) *xcp*; *pc* < *length* (*compE2 i*) ⟧ ⟹ *τmove2 P h stk i pc xcp*
and *τmove2-AAssD1*: ⟦ *τmove2 P h stk* (*a*⌊*i*⌉ := *e*) *pc xcp*; *pc* < *length* (*compE2 a*) ⟧ ⟹ *τmove2*

P h stk a pc xcp
and *τmove2-AAssD2*: ⟦ *τmove2 P h stk* (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *pc*) *xcp*; *pc* < *length* (*compE2 i*) ⟧ ⟹ *τmove2 P h stk i pc xcp*
and *τmove2-AAssD3*:
⟦ *τmove2 P h stk* (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *length* (*compE2 i*) + *pc*) *xcp*; *pc* < *length* (*compE2 e*) ⟧ ⟹ *τmove2 P h stk e pc xcp*
and *τmove2-ALengthD*: ⟦ *τmove2 P h stk* (*a*·*length*) *pc xcp*; *pc* < *length* (*compE2 a*) ⟧ ⟹ *τmove2 P h stk a pc xcp*
and *τmove2-FAccD*: ⟦ *τmove2 P h stk* (*e*·*F*{*D*}) *pc xcp*; *pc* < *length* (*compE2 e*) ⟧ ⟹ *τmove2 P h stk e pc xcp*
and *τmove2-FAssD1*: ⟦ *τmove2 P h stk* (*e*·*F*{*D*} := *e*′) *pc xcp*; *pc* < *length* (*compE2 e*) ⟧ ⟹ *τmove2 P h stk e pc xcp*
and *τmove2-FAssD2*: ⟦ *τmove2 P h stk* (*e*·*F*{*D*} := *e*′) (*length* (*compE2 e*) + *pc*) *xcp*; *pc* < *length* (*compE2 e*′) ⟧ ⟹ *τmove2 P h stk e*′ *pc xcp*
and *τmove2-CASD1*: ⟦ *τmove2 P h stk* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *pc xcp*; *pc* < *length* (*compE2 e1*) ⟧ ⟹ *τmove2 P h stk e1 pc xcp*
and *τmove2-CASD2*: ⟦ *τmove2 P h stk* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *pc*) *xcp*; *pc* < *length* (*compE2 e2*) ⟧ ⟹ *τmove2 P h stk e2 pc xcp*
and *τmove2-CASD3*:
⟦ *τmove2 P h stk* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*) *xcp*; *pc* < *length* (*compE2 e3*) ⟧ ⟹ *τmove2 P h stk e3 pc xcp*
and *τmove2-CallObjD*: ⟦ *τmove2 P h stk* (*e*·*M*(*es*)) *pc xcp*; *pc* < *length* (*compE2 e*) ⟧ ⟹ *τmove2 P h stk e pc xcp*
and *τmove2-BlockNoneD*: *τmove2 P h stk* {*V*:*T*=*None*; *e*} *pc xcp* ⟹ *τmove2 P h stk e pc xcp*
and *τmove2-BlockSomeD*: *τmove2 P h stk* {*V*:*T*=⌊*v*⌋; *e*} (*Suc* (*Suc pc*)) *xcp* ⟹ *τmove2 P h stk e pc xcp*
and *τmove2-sync1D*: ⟦ *τmove2 P h stk* (*sync*$_V$ (*o*′) *e*) *pc xcp*; *pc* < *length* (*compE2 o*′) ⟧ ⟹ *τmove2 P h stk o*′ *pc xcp*
and *τmove2-sync2D*:
⟦ *τmove2 P h stk* (*sync*$_V$ (*o*′) *e*) (*Suc* (*Suc* (*Suc* (*length* (*compE2 o*′) + *pc*)))) *xcp*; *pc* < *length* (*compE2 e*) ⟧ ⟹ *τmove2 P h stk e pc xcp*
and *τmove2-Seq1D*: ⟦ *τmove2 P h stk* (*e1*;; *e2*) *pc xcp*; *pc* < *length* (*compE2 e1*) ⟧ ⟹ *τmove2 P h stk e1 pc xcp*
and *τmove2-Seq2D*: *τmove2 P h stk* (*e1*;; *e2*) (*Suc* (*length* (*compE2 e1*) + *pc*′)) *xcp*′ ⟹ *τmove2 P h stk e2 pc*′ *xcp*′
and *τmove2-IfCondD*: ⟦ *τmove2 P h stk* (*if* (*e*) *e1 else e2*) *pc xcp*; *pc* < *length* (*compE2 e*) ⟧ ⟹ *τmove2 P h stk e pc xcp*
and *τmove2-IfThenD*:
⟦ *τmove2 P h stk* (*if* (*e*) *e1 else e2*) (*Suc* (*length* (*compE2 e*) + *pc*′)) *xcp*′; *pc*′ < *length* (*compE2 e1*) ⟧ ⟹ *τmove2 P h stk e1 pc*′ *xcp*′
and *τmove2-IfElseD*:
τmove2 P h stk (*if* (*e*) *e1 else e2*) (*Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + *pc*′))) *xcp*′ ⟹ *τmove2 P h stk e2 pc*′ *xcp*′
and *τmove2-WhileCondD*: ⟦ *τmove2 P h stk* (*while* (*c*) *b*) *pc xcp*; *pc* < *length* (*compE2 c*) ⟧ ⟹ *τmove2 P h stk c pc xcp*
and *τmove2-ThrowD*: ⟦ *τmove2 P h stk* (*throw e*) *pc xcp*; *pc* < *length* (*compE2 e*) ⟧ ⟹ *τmove2 P h stk e pc xcp*
and *τmove2-Try1D*: ⟦ *τmove2 P h stk* (*try e1 catch*(*C*′ *V*) *e2*) *pc xcp*; *pc* < *length* (*compE2 e1*) ⟧ ⟹ *τmove2 P h stk e1 pc xcp*
apply(*auto elim*!: *τmove2-cases intro*: *τmove2xcp dest*: *τmove2-pc-length-compE2*)
done

lemma *τmove2-Invoke*:
⟦*τmove2 P h stk e pc None*; *compE2 e* ! *pc* = *Invoke M n* ⟧

$\Longrightarrow n < \mathit{length}\ \mathit{stk} \wedge (\mathit{stk}\ !\ n = \mathit{Null} \vee (\forall\ T\ \mathit{Ts}\ \mathit{Tr}\ D.\ \mathit{typeof\text{-}addr}\ h\ (\mathit{the\text{-}Addr}\ (\mathit{stk}\ !\ n)) = \lfloor T \rfloor \longrightarrow P \vdash \mathit{class\text{-}type\text{-}of}\ T\ \mathit{sees}\ M{:}\mathit{Ts}\rightarrow\mathit{Tr} = \mathit{Native}\ \mathit{in}\ D \longrightarrow \tau\mathit{external\text{-}defs}\ D\ M))$

and $\tau\mathit{moves2\text{-}Invoke}$:
$[\![\tau\mathit{moves2}\ P\ h\ \mathit{stk}\ \mathit{es}\ \mathit{pc}\ \mathit{None};\ \mathit{compEs2}\ \mathit{es}\ !\ \mathit{pc} = \mathit{Invoke}\ M\ n\ ]\!]$
$\Longrightarrow n < \mathit{length}\ \mathit{stk} \wedge (\mathit{stk}\ !\ n = \mathit{Null} \vee (\forall\ T\ C\ \mathit{Ts}\ \mathit{Tr}\ D.\ \mathit{typeof\text{-}addr}\ h\ (\mathit{the\text{-}Addr}\ (\mathit{stk}\ !\ n)) = \lfloor T \rfloor \longrightarrow P \vdash \mathit{class\text{-}type\text{-}of}\ T\ \mathit{sees}\ M{:}\mathit{Ts}\rightarrow\mathit{Tr} = \mathit{Native}\ \mathit{in}\ D \longrightarrow \tau\mathit{external\text{-}defs}\ D\ M))$
by(*simp-all add*: $\tau\mathit{move2\text{-}iff}\ \tau\mathit{moves2\text{-}iff}\ \mathit{split\text{-}beta}$)

lemmas $\tau\mathit{move2\text{-}compE2\text{-}not\text{-}Invoke} = \tau\mathit{move2\text{-}Invoke}$
lemmas $\tau\mathit{moves2\text{-}compEs2\text{-}not\text{-}Invoke} = \tau\mathit{moves2\text{-}Invoke}$

lemma $\tau\mathit{move2\text{-}blocks1}$ [*simp*]:
$\tau\mathit{move2}\ P\ h\ \mathit{stk}\ (\mathit{blocks1}\ n\ \mathit{Ts}\ \mathit{body})\ \mathit{pc}'\ \mathit{xcp}' = \tau\mathit{move2}\ P\ h\ \mathit{stk}\ \mathit{body}\ \mathit{pc}'\ \mathit{xcp}'$
by(*simp add*: $\tau\mathit{move2\text{-}iff}$)

lemma $\tau\mathit{instr\text{-}stk\text{-}append}$:
$\tau\mathit{instr}\ P\ h\ \mathit{stk}\ i \Longrightarrow \tau\mathit{instr}\ P\ h\ (\mathit{stk}\ @\ \mathit{vs})\ i$
by(*cases i*)(*auto simp add*: *nth-append*)

lemma $\tau\mathit{move2\text{-}stk\text{-}append}$:
$\tau\mathit{move2}\ P\ h\ \mathit{stk}\ e\ \mathit{pc}\ \mathit{xcp} \Longrightarrow \tau\mathit{move2}\ P\ h\ (\mathit{stk}\ @\ \mathit{vs})\ e\ \mathit{pc}\ \mathit{xcp}$
by(*simp add*: $\tau\mathit{move2\text{-}iff}\ \tau\mathit{instr\text{-}stk\text{-}append}$)

lemma $\tau\mathit{moves2\text{-}stk\text{-}append}$:
$\tau\mathit{moves2}\ P\ h\ \mathit{stk}\ \mathit{es}\ \mathit{pc}\ \mathit{xcp} \Longrightarrow \tau\mathit{moves2}\ P\ h\ (\mathit{stk}\ @\ \mathit{vs})\ \mathit{es}\ \mathit{pc}\ \mathit{xcp}$
by(*simp add*: $\tau\mathit{moves2\text{-}iff}\ \tau\mathit{instr\text{-}stk\text{-}append}$)

fun $\tau\mathit{Move2} :: 'addr\ \mathit{jvm\text{-}prog} \Rightarrow ('addr, 'heap)\ \mathit{jvm\text{-}state} \Rightarrow \mathit{bool}$
where
$\tau\mathit{Move2}\ P\ (\mathit{xcp}, h, []) = \mathit{False}$
$|\ \tau\mathit{Move2}\ P\ (\mathit{xcp}, h, (\mathit{stk}, \mathit{loc}, C, M, \mathit{pc})\ \#\ \mathit{frs}) =$
$\quad (\mathit{let}\ (\text{-},\text{-},\text{-},\mathit{meth}) = \mathit{method}\ P\ C\ M;\ (\text{-},\text{-},\mathit{ins},\mathit{xt}) = \mathit{the}\ \mathit{meth}$
$\quad\ \mathit{in}\ (\mathit{pc} < \mathit{length}\ \mathit{ins} \wedge (\mathit{xcp} = \mathit{None} \longrightarrow \tau\mathit{instr}\ P\ h\ \mathit{stk}\ (\mathit{ins}\ !\ \mathit{pc}))))$

lemma $\tau\mathit{Move2\text{-}iff}$:
$\tau\mathit{Move2}\ P\ \sigma = (\mathit{let}\ (\mathit{xcp}, h, \mathit{frs}) = \sigma$
$\quad\ \mathit{in\ case\ frs\ of}\ [] \Rightarrow \mathit{False}$
$\quad |\ (\mathit{stk}, \mathit{loc}, C, M, \mathit{pc})\ \#\ \mathit{frs}' \Rightarrow$
$\quad (\mathit{let}\ (\text{-},\text{-},\text{-},\mathit{meth}) = \mathit{method}\ P\ C\ M;\ (\text{-},\text{-},\mathit{ins},\mathit{xt}) = \mathit{the}\ \mathit{meth}$
$\quad\ \mathit{in}\ (\mathit{pc} < \mathit{length}\ \mathit{ins} \wedge (\mathit{xcp} = \mathit{None} \longrightarrow \tau\mathit{instr}\ P\ h\ \mathit{stk}\ (\mathit{ins}\ !\ \mathit{pc})))))$
by(*cases* σ)(*clarsimp split*: *list.splits simp add*: *fun-eq-iff split-beta*)

lemma $\tau\mathit{instr\text{-}compP}$ [*simp*]: $\tau\mathit{instr}\ (\mathit{compP}\ f\ P)\ h\ \mathit{stk}\ i \longleftrightarrow \tau\mathit{instr}\ P\ h\ \mathit{stk}\ i$
by(*cases i*) *auto*

lemma [*simp*]: **fixes** $e :: 'addr\ \mathit{expr1}$ **and** $\mathit{es} :: 'addr\ \mathit{expr1\ list}$
shows $\tau\mathit{move2\text{-}compP}$: $\tau\mathit{move2}\ (\mathit{compP}\ f\ P)\ h\ \mathit{stk}\ e = \tau\mathit{move2}\ P\ h\ \mathit{stk}\ e$
and $\tau\mathit{moves2\text{-}compP}$: $\tau\mathit{moves2}\ (\mathit{compP}\ f\ P)\ h\ \mathit{stk}\ \mathit{es} = \tau\mathit{moves2}\ P\ h\ \mathit{stk}\ \mathit{es}$
by(*auto simp add*: $\tau\mathit{move2\text{-}iff}\ \tau\mathit{moves2\text{-}iff}\ \mathit{fun\text{-}eq\text{-}iff}$)

lemma $\tau\mathit{Move2\text{-}compP2}$:
$P \vdash C\ \mathit{sees}\ M{:}\mathit{Ts}\rightarrow T{=}\lfloor \mathit{body} \rfloor\ \mathit{in}\ D \Longrightarrow$
$\quad \tau\mathit{Move2}\ (\mathit{compP2}\ P)\ (\mathit{xcp}, h, (\mathit{stk}, \mathit{loc}, C, M, \mathit{pc})\ \#\ \mathit{frs}) =$

(*case xcp of None* ⇒ *τmove2 P h stk body pc xcp* ∨ *pc* = *length* (*compE2 body*) | *Some a* ⇒ *pc* < *Suc* (*length* (*compE2 body*)))
by(*clarsimp simp add*: *τmove2-iff compP2-def compMb2-def nth-append nth-Cons' split*: *option.splits if-split-asm*)

abbreviation *τMOVE2* ::
'addr jvm-prog ⇒ ((*'addr option* × *'addr frame list*) × *'heap*, (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action*) *trsys*
where *τMOVE2 P* ≡ λ((*xcp*, *frs*), *h*) *ta s*. *τMove2 P* (*xcp*, *h*, *frs*) ∧ *ta* = ε

lemma *τjvmd-heap-unchanged*:
⟦ *P*,*t* ⊢ *Normal* (*xcp*, *h*, *frs*) −ε−*jvmd*→ *Normal* (*xcp'*, *h'*, *frs'*); *τMove2 P* (*xcp*, *h*, *frs*) ⟧
⟹ *h* = *h'*
apply(*erule jvmd-NormalE*)
apply(*clarsimp*)
apply(*cases xcp*)
apply(*rename-tac stk loc C M pc FRS M' Ts T meth mxs mxl ins xt*)
apply(*case-tac ins* ! *pc*)
prefer *19* — BinOpInstr
apply(*rename-tac bop*)
apply(*case-tac the* (*binop bop* (*hd* (*tl stk*)) (*hd stk*)))
apply(*auto simp add*: *split-beta τexternal-def split*: *if-split-asm*)
apply(*fastforce simp add*: *check-def has-method-def τexternal-def dest*: *τexternal-red-external-aggr-heap-unchanged*
done

lemma *mexecd-τmthr-wf*:
τmultithreaded-wf JVM-final (*mexecd P*) (*τMOVE2 P*)
proof
fix *t x h ta x' h'*
assume *mexecd P t* (*x*, *h*) *ta* (*x'*, *h'*)
and *τMOVE2 P* (*x*, *h*) *ta* (*x'*, *h'*)
thus *h* = *h'*
by(*cases x*)(*cases x'*, *auto dest*: *τjvmd-heap-unchanged*)
next
fix *s ta s'*
assume *τMOVE2 P s ta s'* **thus** *ta* = ε **by**(*simp add*: *split-beta*)
qed

end

sublocale *JVM-heap-base* < *execd-mthr*:
τmultithreaded-wf
JVM-final
mexecd P
convert-RA
τMOVE2 P
for *P*
by(*rule mexecd-τmthr-wf*)

context *JVM-heap-base* **begin**

lemma *τexec-1-taD*:
assumes *exec*: *exec-1-d P t* (*Normal* (*xcp*, *h*, *frs*)) *ta* (*Normal* (*xcp'*, *h'*, *frs'*))
and *τ*: *τMove2 P* (*xcp*, *h*, *frs*)

shows *ta* = *ε*
using *assms*
apply(*auto elim*!: *jvmd-NormalE simp add*: *split-beta*)
apply(*cases xcp*)
apply *auto*
apply(*rename-tac stk loc C M pc FRS*)
apply(*case-tac instrs-of P C M* ! *pc*)
apply(*simp-all split*: *if-split-asm*)
apply(*auto simp add*: *check-def has-method-def* τ*external-def dest*!: τ*external-red-external-aggr-TA-empty*)
done

end

end

7.15 JVM Semantics for the delay bisimulation proof from intermediate language to byte code

theory *Execs* **imports** *JVMTau* **begin**

declare *match-ex-table-app* [*simp del*]
match-ex-table-eq-NoneI [*simp del*]
compxE2-size-convs [*simp del*]
compxE2-stack-xlift-convs [*simp del*]
compxEs2-stack-xlift-convs [*simp del*]

type-synonym
('*addr*, '*heap*) *check-instr'* =
'*addr instr* ⇒ '*addr jvm-prog* ⇒ '*heap* ⇒ '*addr val list* ⇒ '*addr val list* ⇒ *cname* ⇒ *mname* ⇒ *pc*
⇒ '*addr frame list* ⇒ *bool*

primrec *check-instr'* :: ('*addr*, '*heap*) *check-instr'*
where
check-instr'-Load:
check-instr' (*Load n*) *P h stk loc C* M_0 *pc frs* =
True

| *check-instr'-Store*:
check-instr' (*Store n*) *P h stk loc* C_0 M_0 *pc frs* =
(*0* < *length stk*)

| *check-instr'-Push*:
check-instr' (*Push v*) *P h stk loc* C_0 M_0 *pc frs* =
True

| *check-instr'-New*:
check-instr' (*New C*) *P h stk loc* C_0 M_0 *pc frs* =
True

| *check-instr'-NewArray*:
check-instr' (*NewArray T*) *P h stk loc C0 M0 pc frs* =
(*0* < *length stk*)

| *check-instr′-ALoad*:
check-instr′ ALoad P h stk loc C0 M0 pc frs =
(*1* < *length stk*)

| *check-instr′-AStore*:
check-instr′ AStore P h stk loc C0 M0 pc frs =
(*2* < *length stk*)

| *check-instr′-ALength*:
check-instr′ ALength P h stk loc C0 M0 pc frs =
(*0* < *length stk*)

| *check-instr′-Getfield*:
check-instr′ (*Getfield F C*) *P h stk loc* C_0 M_0 *pc frs* =
(*0* < *length stk*)

| *check-instr′-Putfield*:
check-instr′ (*Putfield F C*) *P h stk loc* C_0 M_0 *pc frs* =
(*1* < *length stk*)

| *check-instr′-CAS*:
check-instr′ (*CAS F C*) *P h stk loc* C_0 M_0 *pc frs* =
(*2* < *length stk*)

| *check-instr′-Checkcast*:
check-instr′ (*Checkcast T*) *P h stk loc* C_0 M_0 *pc frs* =
(*0* < *length stk*)

| *check-instr′-Instanceof*:
check-instr′ (*Instanceof T*) *P h stk loc* C_0 M_0 *pc frs* =
(*0* < *length stk*)

| *check-instr′-Invoke*:
check-instr′ (*Invoke M n*) *P h stk loc* C_0 M_0 *pc frs* =
(*n* < *length stk*)

| *check-instr′-Return*:
check-instr′ Return P h stk loc C_0 M_0 *pc frs* =
(*0* < *length stk*)

| *check-instr′-Pop*:
check-instr′ Pop P h stk loc C_0 M_0 *pc frs* =
(*0* < *length stk*)

| *check-instr′-Dup*:
check-instr′ Dup P h stk loc C_0 M_0 *pc frs* =
(*0* < *length stk*)

| *check-instr′-Swap*:
check-instr′ Swap P h stk loc C_0 M_0 *pc frs* =
(*1* < *length stk*)

| *check-instr′-BinOpInstr*:
check-instr′ (*BinOpInstr bop*) *P h stk loc* C_0 M_0 *pc frs* =

(*1 < length stk*)

| *check-instr'-IfFalse*:
check-instr' (*IfFalse b*) *P h stk loc* C_0 M_0 *pc frs* =
(*0 < length stk* ∧ *0* ≤ *int pc+b*)

| *check-instr'-Goto*:
check-instr' (*Goto b*) *P h stk loc* C_0 M_0 *pc frs* =
(*0* ≤ *int pc+b*)

| *check-instr'-Throw*:
check-instr' ThrowExc P h stk loc C_0 M_0 *pc frs* =
(*0 < length stk*)

| *check-instr'-MEnter*:
check-instr' MEnter P h stk loc C_0 M_0 *pc frs* =
(*0 < length stk*)

| *check-instr'-MExit*:
check-instr' MExit P h stk loc C_0 M_0 *pc frs* =
(*0 < length stk*)

definition *ci-stk-offer* :: ('*addr*, '*heap*) *check-instr'* ⇒ *bool*
where
ci-stk-offer ci =
(∀ *ins P h stk stk' loc C M pc frs*. *ci ins P h stk loc C M pc frs* ⟶ *ci ins P h* (*stk* @ *stk'*) *loc C M pc frs*)

lemma *ci-stk-offerI*:
(⋀*ins P h stk stk' loc C M pc frs*. *ci ins P h stk loc C M pc frs* ⟹ *ci ins P h* (*stk* @ *stk'*) *loc C M pc frs*) ⟹ *ci-stk-offer ci*
unfolding *ci-stk-offer-def* **by** *blast*

lemma *ci-stk-offerD*:
⟦ *ci-stk-offer ci*; *ci ins P h stk loc C M pc frs* ⟧ ⟹ *ci ins P h* (*stk* @ *stk'*) *loc C M pc frs*
unfolding *ci-stk-offer-def* **by** *blast*

lemma *check-instr'-stk-offer*:
ci-stk-offer check-instr'
proof(*rule ci-stk-offerI*)
fix *ins P h stk stk' loc C M pc frs*
assume *check-instr' ins P h stk loc C M pc frs*
thus *check-instr' ins P h* (*stk* @ *stk'*) *loc C M pc frs*
by(*cases ins*) *auto*
qed

context *JVM-heap-base* **begin**

lemma *check-instr-imp-check-instr'*:
check-instr ins P h stk loc C M pc frs ⟹ *check-instr' ins P h stk loc C M pc frs*
by(*cases ins*) *auto*

lemma *check-instr-stk-offer*:

ci-stk-offer check-instr
proof(*rule ci-stk-offerI*)
fix *ins P h stk stk' loc C M pc frs*
assume *check-instr ins P h stk loc C M pc frs*
thus *check-instr ins P h (stk @ stk') loc C M pc frs*
by(*cases ins*)(*auto simp add*: *nth-append hd-append neq-Nil-conv tl-append split*: *list.split*)
qed

end

primrec *jump-ok* :: *'addr instr list ⇒ nat ⇒ nat ⇒ bool*
where *jump-ok* [] *n n' = True*
| *jump-ok* (*x # xs*) *n n' = (jump-ok xs* (*Suc n*) *n' ∧*
(case x of IfFalse m ⇒ − int n ≤ m ∧ m ≤ int (*n' + length xs*)
| Goto m ⇒ − int n ≤ m ∧ m ≤ int (*n' + length xs*)
| - ⇒ True))

lemma *jump-ok-append* [*simp*]:
jump-ok (*xs @ xs'*) *n n' ⟷ jump-ok xs n* (*n' + length xs'*) *∧ jump-ok xs'* (*n + length xs*) *n'*
apply(*induct xs arbitrary*: *n*)
apply(*simp*)
apply(*auto split*: *instr.split*)
done

lemma *jump-ok-GotoD*:
⟦ *jump-ok ins n n'*; *ins* ! *pc = Goto m*; *pc < length ins* ⟧ *⟹ − int* (*pc + n*) *≤ m ∧ m < int* (*length ins − pc + n'*)
apply(*induct ins arbitrary*: *n n' pc*)
apply(*simp*)
apply(*clarsimp*)
apply(*case-tac pc*)
apply(*fastforce*)+
done

lemma *jump-ok-IfFalseD*:
⟦ *jump-ok ins n n'*; *ins* ! *pc = IfFalse m*; *pc < length ins* ⟧ *⟹ − int* (*pc + n*) *≤ m ∧ m < int* (*length ins − pc + n'*)
apply(*induct ins arbitrary*: *n n' pc*)
apply(*simp*)
apply(*clarsimp*)
apply(*case-tac pc*)
apply(*fastforce*)+
done

lemma fixes *e* :: *'addr expr1* **and** *es* :: *'addr expr1 list*
shows *compE2-jump-ok* [*intro*!]: *jump-ok* (*compE2 e*) *n* (*Suc n'*)
and *compEs2-jump-ok* [*intro*!]: *jump-ok* (*compEs2 es*) *n* (*Suc n'*)
apply(*induct e* **and** *es arbitrary*: *n n'* **and** *n n' rule*: *compE2.induct compEs2.induct*)
apply(*auto split*: *bop.split*)
done

lemma fixes *e* :: *'addr expr1* **and** *es* :: *'addr expr1 list*
shows *compE1-Goto-not-same*: ⟦ *compE2 e* ! *pc = Goto i*; *pc < length* (*compE2 e*) ⟧ *⟹ nat* (*int*

pc + *i*) ≠ *pc*
and *compEs2-Goto-not-same*: ⟦ *compEs2 es* ! *pc* = *Goto i*; *pc* < *length* (*compEs2 es*) ⟧ ⟹ *nat* (*int pc* + *i*) ≠ *pc*
apply(*induct e* **and** *es arbitrary*: *pc i* **and** *pc i rule*: *compE2.induct compEs2.induct*)
apply(*auto simp add*: *nth-Cons nth-append split*: *if-split-asm bop.split-asm nat.splits*)
apply *fastforce*+
done

fun *ins-jump-ok* :: *'addr instr* ⇒ *nat* ⇒ *bool*
where
ins-jump-ok (*Goto m*) *l* = (− (*int l*) ≤ *m*)
| *ins-jump-ok* (*IfFalse m*) *l* = (− (*int l*) ≤ *m*)
| *ins-jump-ok* - - = *True*

definition *wf-ci* :: (*'addr*, *'heap*) *check-instr'* ⇒ *bool*
where
wf-ci ci ⟷
ci-stk-offer ci ∧ *ci* ≤ *check-instr'* ∧
(∀ *ins P h stk loc C M pc pc' frs*. *ci ins P h stk loc C M pc frs* ⟶ *ins-jump-ok ins pc'* ⟶ *ci ins P h stk loc C M pc' frs*)

lemma *wf-ciI*:
⟦ *ci-stk-offer ci*;
⋀*ins P h stk loc C M pc frs*. *ci ins P h stk loc C M pc frs* ⟹ *check-instr' ins P h stk loc C M pc frs*;
⋀*ins P h stk loc C M pc pc' frs*. ⟦ *ci ins P h stk loc C M pc frs*; *ins-jump-ok ins pc'* ⟧ ⟹ *ci ins P h stk loc C M pc' frs* ⟧
⟹ *wf-ci ci*
unfolding *wf-ci-def le-fun-def le-bool-def*
by *blast*

lemma *check-instr'-pc*:
⟦ *check-instr' ins P h stk loc C M pc frs*; *ins-jump-ok ins pc'* ⟧ ⟹ *check-instr' ins P h stk loc C M pc' frs*
by(*cases ins*) *auto*

lemma *wf-ci-check-instr'* [*iff*]:
wf-ci check-instr'
apply(*rule wf-ciI*)
apply(*rule check-instr'-stk-offer*)
apply(*assumption*)
apply(*erule* (*1*) *check-instr'-pc*)
done

lemma *jump-ok-ins-jump-ok*:
⟦ *jump-ok ins n n'*; *pc* < *length ins* ⟧ ⟹ *ins-jump-ok* (*ins* ! *pc*) (*pc* + *n*)
apply(*induct ins arbitrary*: *n n' pc*)
apply(*fastforce simp add*: *nth-Cons' gr0-conv-Suc split*: *instr.split-asm*)+
done

context *JVM-heap-base* **begin**

lemma *check-instr-pc*:
⟦ *check-instr ins P h stk loc C M pc frs*; *ins-jump-ok ins pc'* ⟧ ⟹ *check-instr ins P h stk loc C M*

pc′ frs
by(*cases ins*) *auto*

lemma *wf-ci-check-instr* [*iff*]:
  wf-ci check-instr
apply(*rule wf-ciI*)
  apply(*rule check-instr-stk-offer*)
 apply(*erule check-instr-imp-check-instr′*)
apply(*erule* (*1*) *check-instr-pc*)
done

end

lemma *wf-ciD1*: *wf-ci ci* ⟹ *ci-stk-offer ci*
unfolding *wf-ci-def* **by** *blast*

lemma *wf-ciD2*: ⟦ *wf-ci ci*; *ci ins P h stk loc C M pc frs* ⟧ ⟹ *check-instr′ ins P h stk loc C M pc frs*
unfolding *wf-ci-def le-fun-def le-bool-def*
by *blast*

lemma *wf-ciD3*: ⟦ *wf-ci ci*; *ci ins P h stk loc C M pc frs*; *ins-jump-ok ins pc′* ⟧ ⟹ *ci ins P h stk loc C M pc′ frs*
unfolding *wf-ci-def* **by** *blast*

lemma *check-instr′-ins-jump-ok*: *check-instr′ ins P h stk loc C M pc frs* ⟹ *ins-jump-ok ins pc*
by(*cases ins*) *auto*
lemma *wf-ci-ins-jump-ok*:
  assumes *wf*: *wf-ci ci*
  and *ci*: *ci ins P h stk loc C M pc frs*
  and *pc′*: *pc* ≤ *pc′*
  shows *ins-jump-ok ins pc′*
proof −
  from *wf ci* **have** *check-instr′ ins P h stk loc C M pc frs* **by**(*rule wf-ciD2*)
  with *pc′* **have** *check-instr′ ins P h stk loc C M pc′ frs* **by**(*cases ins*) *auto*
  thus *?thesis* **by**(*rule check-instr′-ins-jump-ok*)
qed

lemma *wf-ciD3′*: ⟦ *wf-ci ci*; *ci ins P h stk loc C M pc frs*; *pc* ≤ *pc′* ⟧ ⟹ *ci ins P h stk loc C M pc′ frs*
apply(*frule* (*2*) *wf-ci-ins-jump-ok*)
apply(*erule* (*2*) *wf-ciD3*)
done

typedef (*′addr*, *′heap*) *check-instr* = *Collect wf-ci* :: (*′addr*, *′heap*) *check-instr′ set*
  morphisms *ci-app Abs-check-instr*
by *auto*

lemma *ci-app-check-instr′* [*simp*]: *ci-app* (*Abs-check-instr check-instr′*) = *check-instr′*
by(*simp add*: *Abs-check-instr-inverse*)

lemma (**in** *JVM-heap-base*) *ci-app-check-instr* [*simp*]: *ci-app* (*Abs-check-instr check-instr*) = *check-instr*
by(*simp add*: *Abs-check-instr-inverse*)

lemma *wf-ci-stk-offerD*:
  ci-app ci ins P h stk loc C M pc frs ⟹ *ci-app ci ins P h* (*stk* @ *stk′*) *loc C M pc frs*

apply(*rule ci-stk-offerD*[*OF wf-ciD1*]) **back**
by(*rule ci-app* [*simplified*])

lemma *wf-ciD2-ci-app*:
ci-app ci ins P h stk loc C M pc frs ⟹ *check-instr' ins P h stk loc C M pc frs*
apply(*cases ci*)
apply(*simp add*: *Abs-check-instr-inverse*)
apply(*erule* (*1*) *wf-ciD2*)
done

lemma *wf-ciD3-ci-app*:
⟦ *ci-app ci ins P h stk loc C M pc frs*; *ins-jump-ok ins pc'* ⟧ ⟹ *ci-app ci ins P h stk loc C M pc' frs*
apply(*cases ci*)
apply(*simp add*: *Abs-check-instr-inverse*)
apply(*erule* (*2*) *wf-ciD3*)
done

lemma *wf-ciD3'-ci-app*: ⟦ *ci-app ci ins P h stk loc C M pc frs*; *pc* ≤ *pc'* ⟧ ⟹ *ci-app ci ins P h stk loc C M pc' frs*
apply(*cases ci*)
apply(*simp add*: *Abs-check-instr-inverse*)
apply(*erule* (*2*) *wf-ciD3'*)
done

context *JVM-heap-base* **begin**

inductive *exec-meth* ::
('*addr*, '*heap*) *check-instr* ⇒ '*addr jvm-prog* ⇒ '*addr instr list* ⇒ *ex-table* ⇒ '*thread-id*
⇒ '*heap* ⇒ ('*addr val list* × '*addr val list* × *pc* × '*addr option*) ⇒ ('*addr*, '*thread-id*, '*heap*) *jvm-thread-action*
⇒ '*heap* ⇒ ('*addr val list* × '*addr val list* × *pc* × '*addr option*) ⇒ *bool*
for *ci* :: ('*addr*, '*heap*) *check-instr* **and** *P* :: '*addr jvm-prog*
and *ins* :: '*addr instr list* **and** *xt* :: *ex-table* **and** *t* :: '*thread-id*
where
exec-instr:
⟦ (*ta*, *xcp*, *h'*, [(*stk'*, *loc'*, *undefined*, *undefined*, *pc'*)]) ∈ *exec-instr* (*ins* ! *pc*) *P t h stk loc undefined undefined pc* [];
pc < *length ins*;
ci-app ci (*ins* ! *pc*) *P h stk loc undefined undefined pc* [] ⟧
⟹ *exec-meth ci P ins xt t h* (*stk*, *loc*, *pc*, *None*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp*)

| *exec-catch*:
⟦ *match-ex-table P* (*cname-of h xcp*) *pc xt* = ⌊(*pc'*, *d*)⌋; *d* ≤ *length stk* ⟧
⟹ *exec-meth ci P ins xt t h* (*stk*, *loc*, *pc*, ⌊*xcp*⌋) ε *h* (*Addr xcp* # *drop* (*size stk* − *d*) *stk*, *loc*, *pc'*, *None*)

lemma *exec-meth-instr*:
exec-meth ci P ins xt t h (*stk*, *loc*, *pc*, *None*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp*) ⟷
(*ta*, *xcp*, *h'*, [(*stk'*, *loc'*, *undefined*, *undefined*, *pc'*)]) ∈ *exec-instr* (*ins* ! *pc*) *P t h stk loc undefined undefined pc* [] ∧ *pc* < *length ins* ∧ *ci-app ci* (*ins* ! *pc*) *P h stk loc undefined undefined pc* []
by(*auto elim*: *exec-meth.cases intro*: *exec-instr*)

lemma *exec-meth-xcpt*:
exec-meth ci P ins xt t h (*stk*, *loc*, *pc*, ⌊*xcp*⌋) *ta h* (*stk'*, *loc'*, *pc'*, *xcp'*) ⟷

(∃ *d*. *match-ex-table P* (*cname-of h xcp*) *pc xt* = ⌊(*pc'*, *d*)⌋ ∧ *ta* = ε ∧ *stk'* = (*Addr xcp* # *drop* (*size stk* − *d*) *stk*) ∧ *loc'* = *loc* ∧ *xcp'* = *None* ∧ *d* ≤ *length stk*)
by(*auto elim*: *exec-meth.cases intro*: *exec-catch*)

abbreviation *exec-meth-a*
where *exec-meth-a* ≡ *exec-meth* (*Abs-check-instr check-instr'*)

abbreviation *exec-meth-d*
where *exec-meth-d* ≡ *exec-meth* (*Abs-check-instr check-instr*)

lemma *exec-meth-length-compE2D* [*dest*]:
exec-meth ci P (*compE2 e*) (*compxE2 e 0 d*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h' s'* ⟹ *pc* < *length* (*compE2 e*)
apply(*erule exec-meth.cases*)
apply(*auto dest*: *match-ex-table-pc-length-compE2*)
done

lemma *exec-meth-length-compEs2D* [*dest*]:
exec-meth ci P (*compEs2 es*) (*compxEs2 es 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h' s'* ⟹ *pc* < *length* (*compEs2 es*)
apply(*erule exec-meth.cases*)
apply(*auto dest*: *match-ex-table-pc-length-compEs2*)
done

lemma *exec-instr-stk-offer*:
assumes *check*: *check-instr'* (*ins* ! *pc*) *P h stk loc C M pc frs*
and *exec*: (*ta'*, *xcp'*, *h'*, (*stk'*, *loc'*, *C*, *M*, *pc'*) # *frs*) ∈ *exec-instr* (*ins* ! *pc*) *P t h stk loc C M pc frs*
shows (*ta'*, *xcp'*, *h'*, (*stk'* @ *stk''*, *loc'*, *C*, *M*, *pc'*) # *frs*) ∈ *exec-instr* (*ins* ! *pc*) *P t h* (*stk* @ *stk''*) *loc C M pc frs*
using *assms*
proof(*cases ins* ! *pc*)
case (*Invoke M n*)
thus *?thesis* **using** *exec check*
by(*auto split*: *if-split-asm extCallRet.splits split del*: *if-split simp add*: *split-beta nth-append min-def extRet2JVM-def*)
qed(*force simp add*: *nth-append is-Ref-def has-method-def nth-Cons split-beta hd-append tl-append neq-Nil-conv split*: *list.split if-split-asm nat.splits sum.split-asm*)+

lemma *exec-meth-stk-offer*:
assumes *exec*: *exec-meth ci P ins xt t h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
shows *exec-meth ci P ins* (*stack-xlift* (*length stk''*) *xt*) *t h* (*stk* @ *stk''*, *loc*, *pc*, *xcp*) *ta h'* (*stk'* @ *stk''*, *loc'*, *pc'*, *xcp'*)
using *exec*
proof(*cases*)
case (*exec-catch xcp d*)
from ‹*match-ex-table P* (*cname-of h xcp*) *pc xt* = ⌊(*pc'*, *d*)⌋›
have *match-ex-table P* (*cname-of h xcp*) *pc* (*stack-xlift* (*length stk''*) *xt*) = ⌊(*pc'*, *length stk''* + *d*)⌋
by(*simp add*: *match-ex-table-stack-xlift*)
moreover have *length stk''* + *d* ≤ *length* (*stk* @ *stk''*) **using** ‹*d* ≤ *length stk*› **by** *simp*
ultimately have *exec-meth ci P ins* (*stack-xlift* (*length stk''*) *xt*) *t h* ((*stk* @ *stk''*), *loc*, *pc*, ⌊*xcp*⌋) ε *h* ((*Addr xcp* # *drop* (*length* (*stk* @ *stk''*) − (*length stk''* + *d*)) (*stk* @ *stk''*)), *loc*, *pc'*, *None*)
by(*rule exec-meth.exec-catch*)
with *exec-catch* **show** *?thesis* **by**(*simp*)
next

case *exec-instr*
note *ciins* = ‹*ci-app ci* (*ins* ! *pc*) *P h stk loc undefined undefined pc* []›
hence *ci-app ci* (*ins* ! *pc*) *P h* (*stk* @ *stk''*) *loc undefined undefined pc* []
by(*rule wf-ci-stk-offerD*)
moreover from *ciins*
have *check-instr'* (*ins* ! *pc*) *P h stk loc undefined undefined pc* []
by(*rule wf-ciD2-ci-app*)
hence (*ta*, *xcp'*, *h'*, [(*stk'* @ *stk''*, *loc'*, *undefined*, *undefined*, *pc'*)]) ∈ *exec-instr* (*ins* ! *pc*) *P t h* (*stk* @ *stk''*) *loc undefined undefined pc* []
using ‹(*ta*, *xcp'*, *h'*, [(*stk'*, *loc'*, *undefined*,*undefined* , *pc'*)]) ∈ *exec-instr* (*ins* ! *pc*) *P t h stk loc undefined undefined pc* []›
by(*rule exec-instr-stk-offer*)
ultimately show *?thesis* **using** *exec-instr* **by**(*auto intro*: *exec-meth.exec-instr*)
qed

lemma *exec-meth-append-xt* [*intro*]:
exec-meth ci P ins xt t h s ta h' s'
⟹ *exec-meth ci P* (*ins* @ *ins'*) (*xt* @ *xt'*) *t h s ta h' s'*
apply(*erule exec-meth.cases*)
apply(*auto*)
apply(*rule exec-instr*)
apply(*clarsimp simp add*: *nth-append*)
apply(*simp*)
apply(*simp add*: *nth-append*)
apply(*rule exec-catch*)
by(*simp*)

lemma *exec-meth-append* [*intro*]:
exec-meth ci P ins xt t h s ta h' s' ⟹ *exec-meth ci P* (*ins* @ *ins'*) *xt t h s ta h' s'*
by(*rule exec-meth-append-xt*[**where** *xt'*=[], *simplified*])

lemma *append-exec-meth-xt*:
assumes *exec*: *exec-meth ci P ins xt t h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
and *jump*: *jump-ok ins 0 n*
and *pcs*: *pcs xt'* ⊆ {*0*..<*length ins'*}
shows *exec-meth ci P* (*ins'* @ *ins*) (*xt'* @ *shift* (*length ins'*) *xt*) *t h* (*stk*, *loc*, (*length ins'* + *pc*), *xcp*) *ta h'* (*stk'*, *loc'*, (*length ins'* + *pc'*), *xcp'*)
using *exec*
proof(*cases*)
case (*exec-catch xcp d*)
from ‹*match-ex-table P* (*cname-of h xcp*) *pc xt* = ⌊(*pc'*, *d*)⌋›
have *match-ex-table P* (*cname-of h xcp*) (*length ins'* + *pc*) (*shift* (*length ins'*) *xt*) = ⌊(*length ins'* + *pc'*, *d*)⌋
by(*simp add*: *match-ex-table-shift*)
moreover from *pcs* **have** *length ins'* + *pc* ∉ *pcs xt'* **by**(*auto*)
ultimately have *match-ex-table P* (*cname-of h xcp*) (*length ins'* + *pc*) (*xt'* @ *shift* (*length ins'*) *xt*) = ⌊(*length ins'* + *pc'*, *d*)⌋
by(*simp add*: *match-ex-table-append-not-pcs*)
with *exec-catch* **show** *?thesis* **by**(*auto dest*: *exec-meth.exec-catch*)
next
case *exec-instr*
note *exec* = ‹(*ta*, *xcp'*, *h'*, [(*stk'*, *loc'*, *undefined*, *undefined*, *pc'*)]) ∈ *exec-instr* (*ins* ! *pc*) *P t h stk loc undefined undefined pc* []›
hence (*ta*, *xcp'*, *h'*, [(*stk'*, *loc'*, *undefined*, *undefined*, *length ins'* + *pc'*)]) ∈ *exec-instr* (*ins* ! *pc*) *P t*

h stk loc undefined undefined (*length ins′* + *pc*) []
proof(*cases ins* ! *pc*)
case (*Goto i*)
with *jump* ‹*pc* < *length ins*› **have** − *int pc* ≤ *i i* < *int* (*length ins* − *pc* + *n*)
by(*auto dest*: *jump-ok-GotoD*)
with *exec Goto* **show** *?thesis* **by**(*auto*)
next
case (*IfFalse i*)
with *jump* ‹*pc* < *length ins*› **have** − *int pc* ≤ *i i* < *int* (*length ins* − *pc* + *n*)
by(*auto dest*: *jump-ok-IfFalseD*)
with *exec IfFalse* **show** *?thesis* **by**(*auto*)
next
case (*Invoke M n*)
with *exec* **show** *?thesis*
by(*auto split*: *if-split-asm extCallRet.splits split del*: *if-split simp add*: *split-beta nth-append min-def extRet2JVM-def*)
qed(*auto simp add*: *split-beta split*: *if-split-asm sum.split-asm*)
moreover from ‹*ci-app ci* (*ins* ! *pc*) *P h stk loc undefined undefined pc* []›
have *ci-app ci* (*ins* ! *pc*) *P h stk loc undefined undefined* (*length ins′* + *pc*) []
by(*rule wf-ciD3′-ci-app*) *simp*
ultimately have *exec-meth ci P* (*ins′* @ *ins*) (*xt′* @ *shift* (*length ins′*) *xt*) *t h* (*stk*, *loc*, (*length ins′* + *pc*), *None*) *ta h′* (*stk′*, *loc′*, (*length ins′* + *pc′*), *xcp′*)
using ‹*pc* < *length ins*› **by** −(*rule exec-meth.exec-instr*, *simp-all*)
thus *?thesis* **using** *exec-instr* **by**(*auto*)
qed

lemma *append-exec-meth*:
assumes *exec*: *exec-meth ci P ins xt t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
and *jump*: *jump-ok ins 0 n*
shows *exec-meth ci P* (*ins′* @ *ins*) (*shift* (*length ins′*) *xt*) *t h* (*stk*, *loc*, (*length ins′* + *pc*), *xcp*) *ta h′* (*stk′*, *loc′*, (*length ins′* + *pc′*), *xcp′*)
using *assms* **by**(*rule append-exec-meth-xt* [**where** *xt′*=[], *simplified*])

lemma *exec-meth-take-xt′*:
⟦ *exec-meth ci P* (*ins* @ *ins′*) (*xt′* @ *xt*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′ s′*;
pc < *length ins*; *pc* ∉ *pcs xt* ⟧
⟹ *exec-meth ci P ins xt′ t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′ s′*
apply(*erule exec-meth.cases*)
apply(*auto intro*: *exec-meth.intros simp add*: *match-ex-table-append nth-append dest*: *match-ex-table-pcsD*)
done

lemma *exec-meth-take-xt*:
⟦ *exec-meth ci P* (*ins* @ *ins′*) (*xt′* @ *shift* (*length ins*) *xt*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′ s′*;
pc < *length ins* ⟧
⟹ *exec-meth ci P ins xt′ t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′ s′*
by(*auto intro*: *exec-meth-take-xt′*)

lemma *exec-meth-take*:
⟦ *exec-meth ci P* (*ins* @ *ins′*) *xt t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′ s′*;
pc < *length ins* ⟧
⟹ *exec-meth ci P ins xt t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′ s′*
by(*auto intro*: *exec-meth-take-xt*[**where** *xt* = []])

lemma *exec-meth-drop-xt*:
assumes *exec*: *exec-meth ci P* (*ins* @ *ins'*) (*xt* @ *shift* (*length ins*) *xt'*) *t h* (*stk*, *loc*, (*length ins* + *pc*), *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
and *xt*: *pcs xt* ⊆ {..<*length ins*}
and *jump*: *jump-ok ins' 0 n*
shows *exec-meth ci P ins' xt' t h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, (*pc'* − *length ins*), *xcp'*)
using *exec*
proof(*cases rule*: *exec-meth.cases*)
case *exec-instr*
let *?PC* = *length ins* + *pc*
note [*simp*] = ‹*xcp* = *None*›
from ‹*?PC* < *length* (*ins* @ *ins'*)› **have** *pc*: *pc* < *length ins'* **by** *simp*
moreover with ‹(*ta*, *xcp'*, *h'*, [(*stk'*, *loc'*, *undefined*, *undefined*, *pc'*)]) ∈ *exec-instr* ((*ins* @ *ins'*) ! *?PC*) *P t h stk loc undefined undefined ?PC* []›
have (*ta*, *xcp'*, *h'*, [(*stk'*, *loc'*, *undefined*, *undefined*, *pc'* − *length ins*)]) ∈ *exec-instr* (*ins'* ! *pc*) *P t h stk loc undefined undefined pc* []
apply(*cases ins'* ! *pc*)
apply(*simp-all add*: *split-beta split*: *if-split-asm sum.split-asm split del*: *if-split*)
apply(*force split*: *extCallRet.splits simp add*: *min-def extRet2JVM-def*)+
done
moreover from ‹*ci-app ci* ((*ins* @ *ins'*) ! *?PC*) *P h stk loc undefined undefined ?PC* []› *jump pc*
have *ci-app ci* (*ins'* ! *pc*) *P h stk loc undefined undefined pc* []
by(*fastforce elim*: *wf-ciD3-ci-app dest*: *jump-ok-ins-jump-ok*)
ultimately show *?thesis* **by**(*auto intro*: *exec-meth.intros*)
next
case (*exec-catch XCP D*)
let *?PC* = *length ins* + *pc*
note [*simp*] = ‹*xcp* = ⌊*XCP*⌋›
‹*ta* = ε› ‹*h'* = *h*› ‹*stk'* = *Addr XCP* # *drop* (*length stk* − *D*) *stk*› ‹*loc'* = *loc*› ‹*xcp'* = *None*›
from ‹*match-ex-table P* (*cname-of h XCP*) *?PC* (*xt* @ *shift* (*length ins*) *xt'*) = ⌊(*pc'*, *D*)⌋› *xt*
have *match-ex-table P* (*cname-of h XCP*) *pc xt'* = ⌊(*pc'* − *length ins*, *D*)⌋
by(*auto simp add*: *match-ex-table-append dest*: *match-ex-table-shift-pcD match-ex-table-pcsD*)
with ‹*D* ≤ *length stk*› **show** *?thesis* **by**(*auto intro*: *exec-meth.intros*)
qed

lemma *exec-meth-drop*:
⟦ *exec-meth ci P* (*ins* @ *ins'*) (*shift* (*length ins*) *xt*) *t h* (*stk*, *loc*, (*length ins* + *pc*), *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*);
jump-ok ins' 0 b ⟧
⟹ *exec-meth ci P ins' xt t h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, (*pc'* − *length ins*), *xcp'*)
by(*auto intro*: *exec-meth-drop-xt*[**where** *xt* = []])

lemma *exec-meth-drop-xt-pc*:
assumes *exec*: *exec-meth ci P* (*ins* @ *ins'*) (*xt* @ *shift* (*length ins*) *xt'*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
and *pc*: *pc* ≥ *length ins*
and *pcs*: *pcs xt* ⊆ {..<*length ins*}
and *jump*: *jump-ok ins' 0 n'*
shows *pc'* ≥ *length ins*
using *exec*
proof(*cases rule*: *exec-meth.cases*)
case *exec-instr* **thus** *?thesis* **using** *jump pc*
apply(*cases ins'* ! (*pc* − *length ins*))
apply(*simp-all add*: *split-beta nth-append split*: *if-split-asm sum.split-asm*)

apply(*force split*: *extCallRet.splits simp add*: *min-def extRet2JVM-def dest*: *jump-ok-GotoD jump-ok-IfFalseD*)+
done
next
case *exec-catch* **thus** *?thesis* **using** *pcs pc*
by(*auto dest*: *match-ex-table-pcsD match-ex-table-shift-pcD simp add*: *match-ex-table-append*)
qed

lemmas *exec-meth-drop-pc* = *exec-meth-drop-xt-pc*[**where** *xt*=[], *simplified*]

definition *exec-move* ::
(*'addr*, *'heap*) *check-instr* ⇒ *'addr J1-prog* ⇒ *'thread-id* ⇒ *'addr expr1*
⇒ *'heap* ⇒ (*'addr val list* × *'addr val list* × *pc* × *'addr option*)
⇒ (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action*
⇒ *'heap* ⇒ (*'addr val list* × *'addr val list* × *pc* × *'addr option*) ⇒ *bool*
where *exec-move ci P t e* ≡ *exec-meth ci* (*compP2 P*) (*compE2 e*) (*compxE2 e 0 0*) *t*

definition *exec-moves* ::
(*'addr*, *'heap*) *check-instr* ⇒ *'addr J1-prog* ⇒ *'thread-id* ⇒ *'addr expr1 list*
⇒ *'heap* ⇒ (*'addr val list* × *'addr val list* × *pc* × *'addr option*)
⇒ (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action*
⇒ *'heap* ⇒ (*'addr val list* × *'addr val list* × *pc* × *'addr option*) ⇒ *bool*
where *exec-moves ci P t es* ≡ *exec-meth ci* (*compP2 P*) (*compEs2 es*) (*compxEs2 es 0 0*) *t*

abbreviation *exec-move-a*
where *exec-move-a* ≡ *exec-move* (*Abs-check-instr check-instr'*)

abbreviation *exec-move-d*
where *exec-move-d* ≡ *exec-move* (*Abs-check-instr check-instr*)

abbreviation *exec-moves-a*
where *exec-moves-a* ≡ *exec-moves* (*Abs-check-instr check-instr'*)

abbreviation *exec-moves-d*
where *exec-moves-d* ≡ *exec-moves* (*Abs-check-instr check-instr*)

lemma *exec-move-newArrayI*:
exec-move ci P t e h s ta h' s' ⟹ *exec-move ci P t* (*newA T*⌊*e*⌉) *h s ta h' s'*
unfolding *exec-move-def* **by** *auto*

lemma *exec-move-newArray*:
pc < *length* (*compE2 e*) ⟹ *exec-move ci P t* (*newA T*⌊*e*⌉) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
unfolding *exec-move-def* **by**(*auto intro*!: *ext intro*: *exec-meth-take*)

lemma *exec-move-CastI*:
exec-move ci P t e h s ta h' s' ⟹ *exec-move ci P t* (*Cast T e*) *h s ta h' s'*
unfolding *exec-move-def* **by** *auto*

lemma *exec-move-Cast*:
pc < *length* (*compE2 e*) ⟹ *exec-move ci P t* (*Cast T e*) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
unfolding *exec-move-def* **by**(*auto intro*!: *ext intro*: *exec-meth-take*)

lemma *exec-move-InstanceOfI*:

exec-move ci P t e h s ta h′ s′ ⟹ *exec-move ci P t* (*e instanceof T*) *h s ta h′ s′*
unfolding *exec-move-def* **by** *auto*

lemma *exec-move-InstanceOf*:
pc < *length* (*compE2 e*) ⟹ *exec-move ci P t* (*e instanceof T*) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
unfolding *exec-move-def* **by**(*auto intro*!: *ext intro*: *exec-meth-take*)

lemma *exec-move-BinOpI1*:
exec-move ci P t e h s ta h′ s′ ⟹ *exec-move ci P t* (*e «bop» e′*) *h s ta h′ s′*
unfolding *exec-move-def* **by** *auto*

lemma *exec-move-BinOp1*:
pc < *length* (*compE2 e*) ⟹ *exec-move ci P t* (*e «bop» e′*) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
unfolding *exec-move-def*
by(*auto intro*!: *ext intro*: *exec-meth-take-xt simp add*: *compxE2-size-convs*)

lemma *exec-move-BinOpI2*:
assumes *exec*: *exec-move ci P t e2 h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
shows *exec-move ci P t* (*e1 «bop» e2*) *h* (*stk* @ [*v*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) *ta h′* (*stk′* @ [*v*], *loc′*, *length* (*compE2 e1*) + *pc′*, *xcp′*)
proof −
from *exec* **have** *exec-meth ci* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
unfolding *exec-move-def* **.**
from *exec-meth-stk-offer*[*OF this*, **where** *stk′′*=[*v*]] **show** *?thesis*
by(*fastforce split*: *bop.splits intro*: *append-exec-meth-xt simp add*: *exec-move-def compxE2-size-convs compxE2-stack-xlift-convs*)
qed

lemma *exec-move-LAssI*:
exec-move ci P t e h s ta h′ s′ ⟹ *exec-move ci P t* (*V* := *e*) *h s ta h′ s′*
unfolding *exec-move-def* **by** *auto*

lemma *exec-move-LAss*:
pc < *length* (*compE2 e*) ⟹ *exec-move ci P t* (*V* := *e*) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
unfolding *exec-move-def* **by**(*auto intro*!: *ext intro*: *exec-meth-take*)

lemma *exec-move-AAccI1*:
exec-move ci P t e h s ta h′ s′ ⟹ *exec-move ci P t* (*e*⌊*e′*⌉) *h s ta h′ s′*
unfolding *exec-move-def* **by** *auto*

lemma *exec-move-AAcc1*:
pc < *length* (*compE2 e*) ⟹ *exec-move ci P t* (*e*⌊*e′*⌉) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
unfolding *exec-move-def*
by(*auto intro*!: *ext intro*: *exec-meth-take-xt simp add*: *compxE2-size-convs*)

lemma *exec-move-AAccI2*:
assumes *exec*: *exec-move ci P t e2 h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
shows *exec-move ci P t* (*e1*⌊*e2*⌉) *h* (*stk* @ [*v*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) *ta h′* (*stk′* @ [*v*], *loc′*, *length* (*compE2 e1*) + *pc′*, *xcp′*)

proof −
from *exec* **have** *exec-meth ci* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
unfolding *exec-move-def* **.**
from *exec-meth-stk-offer*[*OF this*, **where** *stk′′*=[*v*]] **show** *?thesis*
by(*fastforce intro*: *append-exec-meth-xt simp add*: *exec-move-def compxE2-size-convs compxE2-stack-xlift-convs*)
qed

lemma *exec-move-AAssI1*:
exec-move ci P t e h s ta h′ s′ ⟹ *exec-move ci P t* (*e*⌊*e*⌉ := *e′′*) *h s ta h′ s′*
unfolding *exec-move-def* **by** *auto*

lemma *exec-move-AAss1*:
assumes *pc*: *pc* < *length* (*compE2 e*)
shows *exec-move ci P t* (*e*⌊*e*⌉ := *e′′*) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
(**is** *?lhs* = *?rhs*)
proof(*rule ext iffI*)+
fix *ta h′ s′* **assume** *?rhs ta h′ s′*
thus *?lhs ta h′ s′* **by**(*rule exec-move-AAssI1*)
next
fix *ta h′ s′* **assume** *?lhs ta h′ s′*
hence *exec-meth ci* (*compP2 P*) (*compE2 e* @ *compE2 e′* @ *compE2 e′′* @ [*AStore*, *Push Unit*])
(*compxE2 e 0 0* @ *shift* (*length* (*compE2 e*)) (*compxE2 e′ 0* (*Suc 0*) @ *compxE2 e′′* (*length* (*compE2 e′*)) (*Suc* (*Suc 0*)))) *t*
h (*stk*, *loc*, *pc*, *xcp*) *ta h′ s′* **by**(*simp add*: *exec-move-def shift-compxE2 ac-simps*)
thus *?rhs ta h′ s′* **unfolding** *exec-move-def* **using** *pc* **by**(*rule exec-meth-take-xt*)
qed

lemma *exec-move-AAssI2*:
assumes *exec*: *exec-move ci P t e2 h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
shows *exec-move ci P t* (*e1*⌊*e2*⌉ := *e3*) *h* (*stk* @ [*v*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) *ta h′* (*stk′* @ [*v*], *loc′*, *length* (*compE2 e1*) + *pc′*, *xcp′*)
proof −
from *exec* **have** *exec-meth ci* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
unfolding *exec-move-def* **.**
from *exec-meth-stk-offer*[*OF this*, **where** *stk′′*=[*v*], *simplified stack-xlift-compxE2*, *simplified*]
have *exec-meth ci* (*compP2 P*) (*compE2 e2* @ *compE2 e3* @ [*AStore*, *Push Unit*]) (*compxE2 e2 0* (*Suc 0*) @ *shift* (*length* (*compE2 e2*)) (*compxE2 e3 0* (*Suc* (*Suc 0*)))) *t h* (*stk* @ [*v*], *loc*, *pc*, *xcp*) *ta h′* (*stk′* @ [*v*], *loc′*, *pc′*, *xcp′*)
by(*rule exec-meth-append-xt*)
hence *exec-meth ci* (*compP2 P*) (*compE2 e1* @ *compE2 e2* @ *compE2 e3* @ [*AStore*, *Push Unit*]) (*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*compxE2 e2 0* (*Suc 0*) @ *shift* (*length* (*compE2 e2*)) (*compxE2 e3 0* (*Suc* (*Suc 0*))))) *t h* (*stk* @ [*v*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) *ta h′* (*stk′* @ [*v*], *loc′*, *length* (*compE2 e1*) + *pc′*, *xcp′*)
by(*rule append-exec-meth-xt*) *auto*
thus *?thesis* **by**(*auto simp add*: *exec-move-def shift-compxE2 ac-simps*)
qed

lemma *exec-move-AAssI3*:
assumes *exec*: *exec-move ci P t e3 h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
shows *exec-move ci P t* (*e1*⌊*e2*⌉ := *e3*) *h* (*stk* @ [*v′*, *v*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*, *xcp*) *ta h′* (*stk′* @ [*v′*, *v*], *loc′*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc′*, *xcp′*)
proof −

from *exec* **have** *exec-meth ci* (*compP2 P*) (*compE2 e3*) (*compxE2 e3 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
unfolding *exec-move-def* **.**
from *exec-meth-stk-offer*[*OF this*, **where** *stk′′*=[*v′*, *v*], *simplified stack-xlift-compxE2*, *simplified*]
have *exec-meth ci* (*compP2 P*) (*compE2 e3* @ [*AStore*, *Push Unit*]) (*compxE2 e3 0* (*Suc* (*Suc 0*))) *t h* (*stk* @ [*v′*, *v*], *loc*, *pc*, *xcp*) *ta h′* (*stk′* @ [*v′*, *v*], *loc′*, *pc′*, *xcp′*)
by(*rule exec-meth-append*)
hence *exec-meth ci* (*compP2 P*) ((*compE2 e1* @ *compE2 e2*) @ *compE2 e3* @ [*AStore*, *Push Unit*])

((*compxE2 e1 0 0* @ *compxE2 e2* (*length* (*compE2 e1*)) (*Suc 0*)) @ *shift* (*length* (*compE2 e1* @ *compE2 e2*)) (*compxE2 e3 0* (*Suc* (*Suc 0*)))) *t h* (*stk* @ [*v′*, *v*], *loc*, *length* (*compE2 e1* @ *compE2 e2*) + *pc*, *xcp*) *ta h′* (*stk′* @ [*v′*, *v*], *loc′*, *length* (*compE2 e1* @ *compE2 e2*) + *pc′*, *xcp′*)
by(*rule append-exec-meth-xt*) *auto*
thus *?thesis* **by**(*auto simp add*: *exec-move-def shift-compxE2 ac-simps*)
qed

lemma *exec-move-ALengthI*:
exec-move ci P t e h s ta h′ s′ ⟹ *exec-move ci P t* (*e*•*length*) *h s ta h′ s′*
unfolding *exec-move-def* **by** *auto*

lemma *exec-move-ALength*:
pc < *length* (*compE2 e*) ⟹ *exec-move ci P t* (*e*•*length*) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
unfolding *exec-move-def* **by**(*auto intro*!: *ext intro*: *exec-meth-take*)

lemma *exec-move-FAccI*:
exec-move ci P t e h s ta h′ s′ ⟹ *exec-move ci P t* (*e*•*F*{*D*}) *h s ta h′ s′*
unfolding *exec-move-def* **by** *auto*

lemma *exec-move-FAcc*:
pc < *length* (*compE2 e*) ⟹ *exec-move ci P t* (*e*•*F*{*D*}) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
unfolding *exec-move-def* **by**(*auto intro*!: *ext intro*: *exec-meth-take*)

lemma *exec-move-FAssI1*:
exec-move ci P t e h s ta h′ s′ ⟹ *exec-move ci P t* (*e*•*F*{*D*} := *e′*) *h s ta h′ s′*
unfolding *exec-move-def* **by** *auto*

lemma *exec-move-FAss1*:
pc < *length* (*compE2 e*) ⟹ *exec-move ci P t* (*e*•*F*{*D*} := *e′*) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
unfolding *exec-move-def*
by(*auto intro*!: *ext intro*: *exec-meth-take-xt simp add*: *compxE2-size-convs*)

lemma *exec-move-FAssI2*:
assumes *exec*: *exec-move ci P t e2 h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
shows *exec-move ci P t* (*e1*•*F*{*D*} := *e2*) *h* (*stk* @ [*v*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) *ta h′* (*stk′* @ [*v*], *loc′*, *length* (*compE2 e1*) + *pc′*, *xcp′*)
proof −
from *exec* **have** *exec-meth ci* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
unfolding *exec-move-def* **.**
from *exec-meth-stk-offer*[*OF this*, **where** *stk′′*=[*v*]] **show** *?thesis*
by(*fastforce intro*: *append-exec-meth-xt simp add*: *exec-move-def compxE2-size-convs compxE2-stack-xlift-convs*)

qed

lemma *exec-move-CASI1*:
exec-move ci P t e h s ta h′ s′ ⟹ *exec-move ci P t* (*e*·*compareAndSwap*(*D*·*F*, *e′*, *e″*)) *h s ta h′ s′*
unfolding *exec-move-def* **by** *auto*

lemma *exec-move-CAS1*:
assumes *pc*: *pc* < *length* (*compE2 e*)
shows *exec-move ci P t* (*e*·*compareAndSwap*(*D*·*F*, *e′*, *e″*)) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
(**is** *?lhs* = *?rhs*)
proof(*rule ext iffI*)+
fix *ta h′ s′* **assume** *?rhs ta h′ s′*
thus *?lhs ta h′ s′* **by**(*rule exec-move-CASI1*)
next
fix *ta h′ s′* **assume** *?lhs ta h′ s′*
hence *exec-meth ci* (*compP2 P*) (*compE2 e* @ *compE2 e′* @ *compE2 e″* @ [*CAS F D*])
(*compxE2 e 0 0* @ *shift* (*length* (*compE2 e*)) (*compxE2 e′ 0* (*Suc 0*) @ *compxE2 e″* (*length* (*compE2 e′*)) (*Suc* (*Suc 0*)))) *t*
h (*stk*, *loc*, *pc*, *xcp*) *ta h′ s′* **by**(*simp add*: *exec-move-def shift-compxE2 ac-simps*)
thus *?rhs ta h′ s′* **unfolding** *exec-move-def* **using** *pc* **by**(*rule exec-meth-take-xt*)
qed

lemma *exec-move-CASI2*:
assumes *exec*: *exec-move ci P t e2 h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
shows *exec-move ci P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk* @ [*v*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) *ta h′* (*stk′* @ [*v*], *loc′*, *length* (*compE2 e1*) + *pc′*, *xcp′*)
proof −
from *exec* **have** *exec-meth ci* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
unfolding *exec-move-def* **.**
from *exec-meth-stk-offer*[*OF this*, **where** *stk″*=[*v*], *simplified stack-xlift-compxE2*, *simplified*]
have *exec-meth ci* (*compP2 P*) (*compE2 e2* @ *compE2 e3* @ [*CAS F D*]) (*compxE2 e2 0* (*Suc 0*) @ *shift* (*length* (*compE2 e2*)) (*compxE2 e3 0* (*Suc* (*Suc 0*)))) *t h* (*stk* @ [*v*], *loc*, *pc*, *xcp*) *ta h′* (*stk′* @ [*v*], *loc′*, *pc′*, *xcp′*)
by(*rule exec-meth-append-xt*)
hence *exec-meth ci* (*compP2 P*) (*compE2 e1* @ *compE2 e2* @ *compE2 e3* @ [*CAS F D*]) (*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*compxE2 e2 0* (*Suc 0*) @ *shift* (*length* (*compE2 e2*)) (*compxE2 e3 0* (*Suc* (*Suc 0*))))) *t h* (*stk* @ [*v*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) *ta h′* (*stk′* @ [*v*], *loc′*, *length* (*compE2 e1*) + *pc′*, *xcp′*)
by(*rule append-exec-meth-xt*) *auto*
thus *?thesis* **by**(*auto simp add*: *exec-move-def shift-compxE2 ac-simps*)
qed

lemma *exec-move-CASI3*:
assumes *exec*: *exec-move ci P t e3 h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
shows *exec-move ci P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk* @ [*v′*, *v*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*, *xcp*) *ta h′* (*stk′* @ [*v′*, *v*], *loc′*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc′*, *xcp′*)
proof −
from *exec* **have** *exec-meth ci* (*compP2 P*) (*compE2 e3*) (*compxE2 e3 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
unfolding *exec-move-def* **.**
from *exec-meth-stk-offer*[*OF this*, **where** *stk″*=[*v′*, *v*], *simplified stack-xlift-compxE2*, *simplified*]

have *exec-meth ci* (*compP2 P*) (*compE2 e3* @ [*CAS F D*]) (*compxE2 e3 0* (*Suc* (*Suc 0*))) *t h* (*stk* @ [*v′*, *v*], *loc*, *pc*, *xcp*) *ta h′* (*stk′* @ [*v′*, *v*], *loc′*, *pc′*, *xcp′*)
by(*rule exec-meth-append*)
hence *exec-meth ci* (*compP2 P*) ((*compE2 e1* @ *compE2 e2*) @ *compE2 e3* @ [*CAS F D*])
((*compxE2 e1 0 0* @ *compxE2 e2* (*length* (*compE2 e1*)) (*Suc 0*)) @ *shift* (*length* (*compE2 e1* @ *compE2 e2*)) (*compxE2 e3 0* (*Suc* (*Suc 0*)))) *t h* (*stk* @ [*v′*, *v*], *loc*, *length* (*compE2 e1* @ *compE2 e2*) + *pc*, *xcp*) *ta h′* (*stk′* @ [*v′*, *v*], *loc′*, *length* (*compE2 e1* @ *compE2 e2*) + *pc′*, *xcp′*)
by(*rule append-exec-meth-xt*) *auto*
thus *?thesis* **by**(*auto simp add*: *exec-move-def shift-compxE2 ac-simps*)
qed

lemma *exec-move-CallI1*:
exec-move ci P t e h s ta h′ s′ ⟹ *exec-move ci P t* (*e*•*M*(*es*)) *h s ta h′ s′*
unfolding *exec-move-def* **by** *auto*

lemma *exec-move-Call1*:
pc < *length* (*compE2 e*) ⟹ *exec-move ci P t* (*e*•*M*(*es*)) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
unfolding *exec-move-def*
by(*auto intro*!: *ext intro*: *exec-meth-take-xt simp add*: *compxEs2-size-convs*)

lemma *exec-move-CallI2*:
assumes *exec*: *exec-moves ci P t es h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
shows *exec-move ci P t* (*e*•*M*(*es*)) *h* (*stk* @ [*v*], *loc*, *length* (*compE2 e*) + *pc*, *xcp*) *ta h′* (*stk′* @ [*v*], *loc′*, *length* (*compE2 e*) + *pc′*, *xcp′*)
proof −
from *exec* **have** *exec-meth ci* (*compP2 P*) (*compEs2 es*) (*compxEs2 es 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
unfolding *exec-moves-def* **.**
from *exec-meth-stk-offer*[*OF this*, **where** *stk″*=[*v*]] **show** *?thesis*
by(*fastforce intro*: *append-exec-meth-xt simp add*: *exec-move-def compxEs2-size-convs compxEs2-stack-xlift-convs*)
qed

lemma *exec-move-BlockNoneI*:
exec-move ci P t e h s ta h′ s′ ⟹ *exec-move ci P t* {*V*:*T*=*None*; *e*} *h s ta h′ s′*
unfolding *exec-move-def* **by** *simp*

lemma *exec-move-BlockNone*:
exec-move ci P t {*V*:*T*=*None*; *e*} = *exec-move ci P t e*
unfolding *exec-move-def* **by**(*simp*)

lemma *exec-move-BlockSomeI*:
assumes *exec*: *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
shows *exec-move ci P t* {*V*:*T*=⌊*v*⌋; *e*} *h* (*stk*, *loc*, *Suc* (*Suc pc*), *xcp*) *ta h′* (*stk′*, *loc′*, *Suc* (*Suc pc′*), *xcp′*)
proof −
let *?ins* = [*Push v*, *Store V*]
from *exec* **have** *exec-meth ci* (*compP2 P*) (*compE2 e*) (*compxE2 e 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *exec-move-def*)
hence *exec-meth ci* (*compP2 P*) (*?ins* @ *compE2 e*) (*shift* (*length ?ins*) (*compxE2 e 0 0*)) *t h* (*stk*, *loc*, *length ?ins* + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *length ?ins* + *pc′*, *xcp′*)
by(*rule append-exec-meth*) *auto*
thus *?thesis* **by**(*simp add*: *exec-move-def shift-compxE2*)

qed

lemma *exec-move-BlockSome*:
exec-move ci P t {*V*:*T*=⌊*v*⌋; *e*} *h* (*stk*, *loc*, *Suc* (*Suc pc*), *xcp*) *ta h′* (*stk′*, *loc′*, *Suc* (*Suc pc′*), *xcp′*)
=
exec-move ci P t e h (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*) (**is** *?lhs* = *?rhs*)
proof
assume *?rhs* **thus** *?lhs* **by**(*rule exec-move-BlockSomeI*)
next
let *?ins* = [*Push v*, *Store V*]
assume *?lhs*
hence *exec-meth ci* (*compP2 P*) (*?ins* @ *compE2 e*) (*shift* (*length ?ins*) (*compxE2 e 0 0*)) *t h* (*stk*, *loc*, *length ?ins* + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *length ?ins* + *pc′*, *xcp′*)
by(*simp add*: *exec-move-def shift-compxE2*)
hence *exec-meth ci* (*compP2 P*) (*compE2 e*) (*compxE2 e 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *length ?ins* + *pc′* − *length ?ins*, *xcp′*)
by(*rule exec-meth-drop*) *auto*
thus *?rhs* **by**(*simp add*: *exec-move-def*)
qed

lemma *exec-move-SyncI1*:
exec-move ci P t e h s ta h′ s′ ⟹ *exec-move ci P t* (*sync*$_V$ (*e*) *e′*) *h s ta h′ s′*
unfolding *exec-move-def* **by** *auto*

lemma *exec-move-Sync1*:
assumes *pc*: *pc* < *length* (*compE2 e*)
shows *exec-move ci P t* (*sync*$_V$ (*e*) *e′*) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
(**is** *?lhs* = *?rhs*)
proof(*rule ext iffI*)+
fix *ta h′ s′*
assume *?lhs ta h′ s′*
hence *exec-meth ci* (*compP2 P*) (*compE2 e* @ *Dup* # *Store V* # *MEnter* # *compE2 e′* @ [*Load V*, *MExit*, *Goto 4*, *Load V*, *MExit*, *ThrowExc*])
(*compxE2 e 0 0* @ *shift* (*length* (*compE2 e*)) (*compxE2 e′ 3 0* @ [(*3*, *3* + *length* (*compE2 e′*), *None*, *6* + *length* (*compE2 e′*), *0*)]))
t h (*stk*, *loc*, *pc*, *xcp*) *ta h′ s′*
by(*simp add*: *shift-compxE2 ac-simps exec-move-def*)
thus *?rhs ta h′ s′* **unfolding** *exec-move-def* **using** *pc* **by**(*rule exec-meth-take-xt*)
qed(*rule exec-move-SyncI1*)

lemma *exec-move-SyncI2*:
assumes *exec*: *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
shows *exec-move ci P t* (*sync*$_V$ (*o′*) *e*) *h* (*stk*, *loc*, (*Suc* (*Suc* (*Suc* (*length* (*compE2 o′*) + *pc*)))), *xcp*) *ta h′* (*stk′*, *loc′*, (*Suc* (*Suc* (*Suc* (*length* (*compE2 o′*) + *pc′*)))), *xcp′*)
proof −
let *?e* = *compE2 o′* @ [*Dup*, *Store V*, *MEnter*]
let *?e′* = [*Load V*, *MExit*, *Goto 4*, *Load V*, *MExit*, *ThrowExc*]
from *exec* **have** *exec-meth ci* (*compP2 P*) (*compE2 e*) (*compxE2 e 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *exec-move-def*)
hence *exec-meth ci* (*compP2 P*) ((*?e* @ *compE2 e*) @ *?e′*) ((*compxE2 o′ 0 0* @ *shift* (*length ?e*) (*compxE2 e 0 0*)) @ [(*length ?e*, *length ?e* + *length* (*compE2 e*), *None*, *length ?e* + *length* (*compE2 e*) + *3*, *0*)]) *t h* (*stk*, *loc*, (*length ?e* + *pc*), *xcp*) *ta h′* (*stk′*, *loc′*, (*length ?e* + *pc′*), *xcp′*)
by(*rule exec-meth-append-xt*[*OF append-exec-meth-xt*]) *auto*

thus *?thesis* **by**(*simp add*: *eval-nat-numeral shift-compxE2 exec-move-def*)
qed

lemma *exec-move-SeqI1*:
exec-move ci P t e h s ta h′ s′ ⟹ *exec-move ci P t* (*e*;;*e′*) *h s ta h′ s′*
unfolding *exec-move-def* **by** *auto*

lemma *exec-move-Seq1*:
assumes *pc*: *pc* < *length* (*compE2 e*)
shows *exec-move ci P t* (*e*;;*e′*) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
(**is** *?lhs* = *?rhs*)
proof(*rule ext iffI*)+
fix *ta h′ s′*
assume *?lhs ta h′ s′*
hence *exec-meth ci* (*compP2 P*) (*compE2 e* @ *Pop* # *compE2 e′*) (*compxE2 e 0 0* @ *shift* (*length* (*compE2 e*)) (*compxE2 e′* (*Suc 0*) *0*)) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′ s′*
by(*simp add*: *exec-move-def shift-compxE2*)
thus *?rhs ta h′ s′* **unfolding** *exec-move-def* **using** *pc* **by**(*rule exec-meth-take-xt*)
qed(*rule exec-move-SeqI1*)

lemma *exec-move-SeqI2*:
assumes *exec*: *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′* ,*xcp′*)
shows *exec-move ci P t* (*e′*;;*e*) *h* (*stk*, *loc*, (*Suc* (*length* (*compE2 e′*) + *pc*)), *xcp*) *ta h′* (*stk′*, *loc′*, (*Suc* (*length* (*compE2 e′*) + *pc′*)), *xcp′*)
proof −
from *exec* **have** *exec-meth ci* (*compP2 P*) (*compE2 e*) (*compxE2 e 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *exec-move-def*)
hence *exec-meth ci* (*compP2 P*) ((*compE2 e′* @ [*Pop*]) @ *compE2 e*) (*compxE2 e′ 0 0* @ *shift* (*length* (*compE2 e′* @ [*Pop*])) (*compxE2 e 0 0*)) *t h* (*stk*, *loc*, (*length* ((*compE2 e′*) @ [*Pop*]) + *pc*), *xcp*) *ta h′* (*stk′*, *loc′*, (*length* ((*compE2 e′*) @ [*Pop*]) + *pc′*), *xcp′*)
by(*rule append-exec-meth-xt*) *auto*
thus *?thesis* **by**(*simp add*: *shift-compxE2 exec-move-def*)
qed

lemma *exec-move-Seq2*:
assumes *pc*: *pc* < *length* (*compE2 e*)
shows *exec-move ci P t* (*e′*;;*e*) *h* (*stk*, *loc*, *Suc* (*length* (*compE2 e′*) + *pc*), *xcp*) *ta*
h′ (*stk′*, *loc′*, *Suc* (*length* (*compE2 e′*) + *pc′*), *xcp′*) =
exec-move ci P t e h (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
(**is** *?lhs* = *?rhs*)
proof
let *?E* = *compE2 e′* @ [*Pop*]
assume *?lhs*
hence *exec-meth ci* (*compP2 P*) (*?E* @ *compE2 e*) (*compxE2 e′ 0 0* @ *shift* (*length ?E*) (*compxE2 e 0 0*)) *t h* (*stk*, *loc*, *length ?E* + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *length ?E* + *pc′*, *xcp′*)
by(*simp add*: *exec-move-def shift-compxE2*)
from *exec-meth-drop-xt*[*OF this*] **show** *?rhs* **unfolding** *exec-move-def* **by** *fastforce*
qed(*rule exec-move-SeqI2*)

lemma *exec-move-CondI1*:
exec-move ci P t e h s ta h′ s′ ⟹ *exec-move ci P t* (*if* (*e*) *e1 else e2*) *h s ta h′ s′*
unfolding *exec-move-def* **by** *auto*

lemma *exec-move-Cond1*:
assumes *pc*: *pc* < *length* (*compE2 e*)
shows *exec-move ci P t* (*if* (*e*) *e1 else e2*) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
(**is** *?lhs* = *?rhs*)
proof(*rule ext iffI*)+
let *?E* = *IfFalse* (2 + *int* (*length* (*compE2 e1*))) # *compE2 e1* @ *Goto* (1 + *int* (*length* (*compE2 e2*))) # *compE2 e2*
let *?xt* = *compxE2 e1* (*Suc 0*) *0* @ *compxE2 e2* (*Suc* (*Suc* (*length* (*compE2 e1*)))) *0*
fix *ta h' s'*
assume *?lhs ta h' s'*
hence *exec-meth ci* (*compP2 P*) (*compE2 e* @ *?E*) (*compxE2 e 0 0* @ *shift* (*length* (*compE2 e*)) *?xt*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h' s'*
by(*simp add*: *exec-move-def shift-compxE2 ac-simps*)
thus *?rhs ta h' s'* **unfolding** *exec-move-def* **using** *pc* **by**(*rule exec-meth-take-xt*)
qed(*rule exec-move-CondI1*)

lemma *exec-move-CondI2*:
assumes *exec*: *exec-move ci P t e1 h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
shows *exec-move ci P t* (*if* (*e*) *e1 else e2*) *h* (*stk*, *loc*, (*Suc* (*length* (*compE2 e*) + *pc*)), *xcp*) *ta h'* (*stk'*, *loc'*, (*Suc* (*length* (*compE2 e*) + *pc'*)), *xcp'*)
proof −
from *exec* **have** *exec-meth ci* (*compP2 P*) (*compE2 e1*) (*compxE2 e1 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *exec-move-def*)
hence *exec-meth ci* (*compP2 P*) (((*compE2 e* @ [*IfFalse* (2 + *int* (*length* (*compE2 e1*)))]) @ *compE2 e1*) @ *Goto* (1 + *int* (*length* (*compE2 e2*))) # *compE2 e2*) ((*compxE2 e 0 0* @ *shift* (*length* (*compE2 e* @ [*IfFalse* (2 + *int* (*length* (*compE2 e1*)))])) (*compxE2 e1 0 0*)) @ (*compxE2 e2* (*Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*)))) *0*)) *t h* (*stk*, *loc*, (*length* (*compE2 e* @ [*IfFalse* (2 + *int* (*length* (*compE2 e1*)))]) + *pc*), *xcp*) *ta h'* (*stk'*, *loc'*, (*length* (*compE2 e* @ [*IfFalse* (2 + *int* (*length* (*compE2 e1*)))]) + *pc'*), *xcp'*)
by −(*rule exec-meth-append-xt*, *rule append-exec-meth-xt*, *auto*)
thus *?thesis* **by**(*simp add*: *shift-compxE2 exec-move-def*)
qed

lemma *exec-move-Cond2*:
assumes *pc*: *pc* < *length* (*compE2 e1*)
shows *exec-move ci P t* (*if* (*e*) *e1 else e2*) *h* (*stk*, *loc*, (*Suc* (*length* (*compE2 e*) + *pc*)), *xcp*) *ta h'* (*stk'*, *loc'*, (*Suc* (*length* (*compE2 e*) + *pc'*)), *xcp'*) = *exec-move ci P t e1 h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
(**is** *?lhs* = *?rhs*)
proof
let *?E1* = *compE2 e* @ [*IfFalse* (2 + *int* (*length* (*compE2 e1*)))]
let *?E2* = *Goto* (1 + *int* (*length* (*compE2 e2*))) # *compE2 e2*
assume *?lhs*
hence *exec-meth ci* (*compP2 P*) (*?E1* @ *compE2 e1* @ *?E2*) (*compxE2 e 0 0* @ *shift* (*length ?E1*) (*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*compxE2 e2* (*Suc 0*) *0*))) *t h* (*stk*, *loc*, *length ?E1* + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *length ?E1* + *pc'*, *xcp'*)
by(*simp add*: *exec-move-def shift-compxE2 ac-simps*)
thus *?rhs* **unfolding** *exec-move-def*
by −(*rule exec-meth-take-xt*,*drule exec-meth-drop-xt*,*auto simp add*: *pc*)
qed(*rule exec-move-CondI2*)

lemma *exec-move-CondI3*:

assumes *exec*: *exec-move ci P t e2 h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
shows *exec-move ci P t* (*if* (*e*) *e1 else e2*) *h* (*stk*, *loc*, *Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + *pc*)), *xcp*) *ta h′* (*stk′*, *loc′*, *Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + *pc′*)), *xcp′*)
proof −
let *?E* = *compE2 e* @ *IfFalse* (2 + *int* (*length* (*compE2 e1*))) # *compE2 e1* @ [*Goto* (1 + *int* (*length* (*compE2 e2*)))]
let *?xt* = *compxE2 e 0 0* @ *compxE2 e1* (*Suc* (*length* (*compE2 e*))) *0*
from *exec* **have** *exec-meth ci* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *exec-move-def*)
hence *exec-meth ci* (*compP2 P*) (*?E* @ *compE2 e2*) (*?xt* @ *shift* (*length ?E*) (*compxE2 e2 0 0*)) *t h* (*stk*, *loc*, *length ?E* + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *length ?E* + *pc′*, *xcp′*)
by(*rule append-exec-meth-xt*) *auto*
thus *?thesis* **by**(*simp add*: *shift-compxE2 exec-move-def*)
qed

lemma *exec-move-Cond3*:
exec-move ci P t (*if* (*e*) *e1 else e2*) *h* (*stk*, *loc*, *Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + *pc*)), *xcp*) *ta*
h′ (*stk′*, *loc′*, *Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + *pc′*)), *xcp′*) =
exec-move ci P t e2 h (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
(**is** *?lhs* = *?rhs*)
proof
let *?E* = *compE2 e* @ *IfFalse* (2 + *int* (*length* (*compE2 e1*))) # *compE2 e1* @ [*Goto* (1 + *int* (*length* (*compE2 e2*)))]
let *?xt* = *compxE2 e 0 0* @ *compxE2 e1* (*Suc* (*length* (*compE2 e*))) *0*
assume *?lhs*
hence *exec-meth ci* (*compP2 P*) (*?E* @ *compE2 e2*) (*?xt* @ *shift* (*length ?E*) (*compxE2 e2 0 0*)) *t h* (*stk*, *loc*, *length ?E* + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *length ?E* + *pc′*, *xcp′*)
by(*simp add*: *shift-compxE2 exec-move-def*)
thus *?rhs* **unfolding** *exec-move-def* **by** −(*drule exec-meth-drop-xt*, *auto*)
qed(*rule exec-move-CondI3*)

lemma *exec-move-WhileI1*:
exec-move ci P t e h s ta h′ s′ ⟹ *exec-move ci P t* (*while* (*e*) *e′*) *h s ta h′ s′*
unfolding *exec-move-def* **by** *auto*

lemma (**in** *ab-group-add*) *uminus-minus-left-commute*:
− *a* − (*b* + *c*) = − *b* − (*a* + *c*)
by (*simp add*: *algebra-simps*)

lemma *exec-move-While1*:
assumes *pc*: *pc* < *length* (*compE2 e*)
shows *exec-move ci P t* (*while* (*e*) *e′*) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
(**is** *?lhs* = *?rhs*)
proof(*rule ext iffI*)+
let *?E* = *IfFalse* (3 + *int* (*length* (*compE2 e′*))) # *compE2 e′* @ [*Pop*, *Goto* (− *int* (*length* (*compE2 e*)) + (−2 − *int* (*length* (*compE2 e′*)))), *Push Unit*]
let *?xt* = *compxE2 e′* (*Suc 0*) *0*
fix *ta h′ s′*
assume *?lhs ta h′ s′*
then have *exec-meth ci* (*compP2 P*) (*compE2 e* @ *?E*) (*compxE2 e 0 0* @ *shift* (*length* (*compE2 e*)) *?xt*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′ s′*

by (*simp add*: *exec-move-def shift-compxE2 algebra-simps uminus-minus-left-commute*)
thus *?rhs ta h′ s′* **unfolding** *exec-move-def* **using** *pc* **by**(*rule exec-meth-take-xt*)
qed(*rule exec-move-WhileI1*)

lemma *exec-move-WhileI2*:
assumes *exec*: *exec-move ci P t e1 h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
shows *exec-move ci P t* (*while* (*e*) *e1*) *h* (*stk*, *loc*, (*Suc* (*length* (*compE2 e*) + *pc*)), *xcp*) *ta h′* (*stk′*, *loc′*, (*Suc* (*length* (*compE2 e*) + *pc′*)), *xcp′*)
proof −
let *?E* = *compE2 e* @ [*IfFalse* (3 + *int* (*length* (*compE2 e1*)))]
let *?E′* = [*Pop*, *Goto* (− *int* (*length* (*compE2 e*)) + (−2 − *int* (*length* (*compE2 e1*)))), *Push Unit*]
from *exec* **have** *exec-meth ci* (*compP2 P*) (*compE2 e1*) (*compxE2 e1 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *exec-move-def*)
hence *exec-meth ci* (*compP2 P*) ((*?E* @ *compE2 e1*) @ *?E′*) (*compxE2 e 0 0* @ *shift* (*length ?E*) (*compxE2 e1 0 0*)) *t h* (*stk*, *loc*, *length ?E* + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *length ?E* + *pc′*, *xcp′*)
by −(*rule exec-meth-append*, *rule append-exec-meth-xt*, *auto*)
thus *?thesis* **by** (*simp add*: *shift-compxE2 exec-move-def algebra-simps uminus-minus-left-commute*)
qed

lemma *exec-move-While2*:
assumes *pc*: *pc* < *length* (*compE2 e′*)
shows *exec-move ci P t* (*while* (*e*) *e′*) *h* (*stk*, *loc*, (*Suc* (*length* (*compE2 e*) + *pc*)), *xcp*) *ta*
h′ (*stk′*, *loc′*, (*Suc* (*length* (*compE2 e*) + *pc′*)), *xcp′*) =
exec-move ci P t e′ h (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
(**is** *?lhs* = *?rhs*)
proof
let *?E* = *compE2 e* @ [*IfFalse* (3 + *int* (*length* (*compE2 e′*)))]
let *?E′* = [*Pop*, *Goto* (− *int* (*length* (*compE2 e*)) + (−2 − *int* (*length* (*compE2 e′*)))), *Push Unit*]
assume *?lhs*
hence *exec-meth ci* (*compP2 P*) ((*?E* @ *compE2 e′*) @ *?E′*) (*compxE2 e 0 0* @ *shift* (*length ?E*) (*compxE2 e′ 0 0*)) *t h* (*stk*, *loc*, *length ?E* + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *length ?E* + *pc′*, *xcp′*)
by(*simp add*: *exec-move-def shift-compxE2 algebra-simps uminus-minus-left-commute*)
thus *?rhs* **unfolding** *exec-move-def* **using** *pc*
by −(*drule exec-meth-take*, *simp*, *drule exec-meth-drop-xt*, *auto*)
qed(*rule exec-move-WhileI2*)

lemma *exec-move-ThrowI*:
exec-move ci P t e h s ta h′ s′ ⟹ *exec-move ci P t* (*throw e*) *h s ta h′ s′*
unfolding *exec-move-def* **by** *auto*

lemma *exec-move-Throw*:
pc < *length* (*compE2 e*) ⟹ *exec-move ci P t* (*throw e*) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
unfolding *exec-move-def* **by**(*auto intro*!: *ext intro*: *exec-meth-take*)

lemma *exec-move-TryI1*:
exec-move ci P t e h s ta h′ s′ ⟹ *exec-move ci P t* (*try e catch*(*C V*) *e′*) *h s ta h′ s′*
unfolding *exec-move-def* **by** *auto*

lemma *exec-move-TryI2*:
assumes *exec*: *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
shows *exec-move ci P t* (*try e′ catch*(*C V*) *e*) *h* (*stk*, *loc*, *Suc* (*Suc* (*length* (*compE2 e′*) + *pc*)), *xcp*) *ta h′* (*stk′*, *loc′*, *Suc* (*Suc* (*length* (*compE2 e′*) + *pc′*)), *xcp′*)

proof −
let *?e* = *compE2 e′* @ [*Goto* (*int*(*size* (*compE2 e*))+*2*), *Store V*]
from *exec* **have** *exec-meth ci* (*compP2 P*) (*compE2 e*) (*compxE2 e 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *exec-move-def*)
hence *exec-meth ci* (*compP2 P*) ((*?e* @ *compE2 e*) @ []) ((*compxE2 e′ 0 0* @ *shift* (*length ?e*) (*compxE2 e 0 0*)) @ [(*0*, *length* (*compE2 e′*), ⌊*C*⌋, *Suc* (*length* (*compE2 e′*)), *0*)]) *t h* (*stk*, *loc*, (*length ?e* + *pc*), *xcp*) *ta h′* (*stk′*, *loc′*, (*length ?e* + *pc′*), *xcp′*)
by(*rule exec-meth-append-xt*[*OF append-exec-meth-xt*]) *auto*
thus *?thesis* **by**(*simp add*: *eval-nat-numeral shift-compxE2 exec-move-def*)
qed

lemma *exec-move-Try2*:
exec-move ci P t (*try e catch*(*C V*) *e′*) *h* (*stk*, *loc*, *Suc* (*Suc* (*length* (*compE2 e*) + *pc*)), *xcp*) *ta*
h′ (*stk′*, *loc′*, *Suc* (*Suc* (*length* (*compE2 e*) + *pc′*)), *xcp′*) =
exec-move ci P t e′ h (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
(**is** *?lhs* = *?rhs*)
proof
let *?E* = *compE2 e* @ [*Goto* (*int*(*size* (*compE2 e′*))+*2*), *Store V*]
let *?xt* = [(*0*, *length* (*compE2 e*), ⌊*C*⌋, *Suc* (*length* (*compE2 e*)), *0*)]
assume *lhs*: *?lhs*
hence *pc*: *pc* < *length* (*compE2 e′*)
by(*fastforce elim*!: *exec-meth.cases simp add*: *exec-move-def match-ex-table-append match-ex-entry dest*: *match-ex-table-pcsD*)
from *lhs* **have** *exec-meth ci* (*compP2 P*) ((*?E* @ *compE2 e′*) @ []) ((*compxE2 e 0 0* @ *shift* (*length ?E*) (*compxE2 e′ 0 0*)) @ *?xt*) *t h* (*stk*, *loc*, *length ?E* + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *length ?E* + *pc′*, *xcp′*)
by(*simp add*: *exec-move-def shift-compxE2 ac-simps*)
thus *?rhs* **unfolding** *exec-move-def* **using** *pc*
by−(*drule exec-meth-drop-xt*[*OF exec-meth-take-xt′*], *auto*)
qed(*rule exec-move-TryI2*)

lemma *exec-move-raise-xcp-pcD*:
exec-move ci P t E h (*stk*, *loc*, *pc*, *None*) *ta h′* (*stk′*, *loc′*, *pc′*, *Some a*) ⟹ *pc′* = *pc*
apply(*cases compE2 E* ! *pc*)
apply(*auto simp add*: *exec-move-def elim*!: *exec-meth.cases split*: *if-split-asm sum.split-asm*)
apply(*auto split*: *extCallRet.split-asm simp add*: *split-beta*)
done

definition *τexec-meth* ::
('*addr*, '*heap*) *check-instr* ⇒ '*addr jvm-prog* ⇒ '*addr instr list* ⇒ *ex-table* ⇒ '*thread-id* ⇒ '*heap*
⇒ ('*addr val list* × '*addr val list* × *pc* × '*addr option*)
⇒ ('*addr val list* × '*addr val list* × *pc* × '*addr option*) ⇒ *bool*
where
τexec-meth ci P ins xt t h s s′ ⟷
exec-meth ci P ins xt t h s ε h s′ ∧ (*snd* (*snd* (*snd s*)) = *None* ⟶ *τinstr P h* (*fst s*) (*ins* ! *fst* (*snd* (*snd s*))))

abbreviation *τexec-meth-a*
where *τexec-meth-a* ≡ *τexec-meth* (*Abs-check-instr check-instr′*)

abbreviation *τexec-meth-d*
where *τexec-meth-d* ≡ *τexec-meth* (*Abs-check-instr check-instr*)

lemma *τexec-methI* [*intro*]:
⟦ *exec-meth ci P ins xt t h* (*stk*, *loc*, *pc*, *xcp*) *ε h s′*; *xcp* = *None* ⟹ *τinstr P h stk* (*ins* ! *pc*) ⟧
⟹ *τexec-meth ci P ins xt t h* (*stk*, *loc*, *pc*, *xcp*) *s′*
by(*simp add*: *τexec-meth-def*)

lemma *τexec-methE* [*elim*]:
assumes *τexec-meth ci P ins xt t h s s′*
obtains *stk loc pc xcp*
where *s* = (*stk*, *loc*, *pc*, *xcp*)
and *exec-meth ci P ins xt t h* (*stk*, *loc*, *pc*, *xcp*) *ε h s′*
and *xcp* = *None* ⟹ *τinstr P h stk* (*ins* ! *pc*)
using *assms*
by(*cases s*)(*auto simp add*: *τexec-meth-def*)

abbreviation *τExec-methr* ::
('*addr*, '*heap*) *check-instr* ⇒ '*addr jvm-prog* ⇒ '*addr instr list* ⇒ *ex-table* ⇒ '*thread-id* ⇒ '*heap*
⇒ ('*addr val list* × '*addr val list* × *pc* × '*addr option*)
⇒ ('*addr val list* × '*addr val list* × *pc* × '*addr option*) ⇒ *bool*
where
τExec-methr ci P ins xt t h == (*τexec-meth ci P ins xt t h*)^**

abbreviation *τExec-metht* ::
('*addr*, '*heap*) *check-instr* ⇒ '*addr jvm-prog* ⇒ '*addr instr list* ⇒ *ex-table* ⇒ '*thread-id* ⇒ '*heap*
⇒ ('*addr val list* × '*addr val list* × *pc* × '*addr option*)
⇒ ('*addr val list* × '*addr val list* × *pc* × '*addr option*) ⇒ *bool*
where
τExec-metht ci P ins xt t h == (*τexec-meth ci P ins xt t h*)^++

abbreviation *τExec-methr-a*
where *τExec-methr-a* ≡ *τExec-methr* (*Abs-check-instr check-instr′*)

abbreviation *τExec-methr-d*
where *τExec-methr-d* ≡ *τExec-methr* (*Abs-check-instr check-instr*)

abbreviation *τExec-metht-a*
where *τExec-metht-a* ≡ *τExec-metht* (*Abs-check-instr check-instr′*)

abbreviation *τExec-metht-d*
where *τExec-metht-d* ≡ *τExec-metht* (*Abs-check-instr check-instr*)

lemma *τExec-methr-refl*: *τExec-methr ci P ins xt t h s s* **..**

lemma *τExec-methr-step′*:
⟦ *τExec-methr ci P ins xt t h s* (*stk′*, *loc′*, *pc′*, *xcp′*);
τexec-meth ci P ins xt t h (*stk′*, *loc′*, *pc′*, *xcp′*) *s′* ⟧
⟹ *τExec-methr ci P ins xt t h s s′*
by(*rule rtranclp.rtrancl-into-rtrancl*)

lemma *τExec-methr-step*:
⟦ *τExec-methr ci P ins xt t h s* (*stk′*, *loc′*, *pc′*, *xcp′*);
exec-meth ci P ins xt t h (*stk′*, *loc′*, *pc′*, *xcp′*) *ε h s′*;
xcp′ = *None* ⟹ *τinstr P h stk′* (*ins* ! *pc′*) ⟧
⟹ *τExec-methr ci P ins xt t h s s′*

by(*erule* τ*Exec-methr-step'*)(*rule* τ*exec-methI*)

lemmas τ*Exec-methr-intros* = τ*Exec-methr-refl* τ*Exec-methr-step*
lemmas τ*Exec-methr1step* = τ*Exec-methr-step*[*OF* τ*Exec-methr-refl*]
lemmas τ*Exec-methr2step* = τ*Exec-methr-step*[*OF* τ*Exec-methr-step*, *OF* τ*Exec-methr-refl*]
lemmas τ*Exec-methr3step* = τ*Exec-methr-step*[*OF* τ*Exec-methr-step*, *OF* τ*Exec-methr-step*, *OF* τ*Exec-methr-refl*]

lemma τ*Exec-methr-cases* [*consumes 1*, *case-names refl step*]:
assumes τ*Exec-methr ci P ins xt t h s s'*
obtains $s = s'$
| *stk' loc' pc' xcp'*
where τ*Exec-methr ci P ins xt t h s* (*stk'*, *loc'*, *pc'*, *xcp'*)
exec-meth ci P ins xt t h (*stk'*, *loc'*, *pc'*, *xcp'*) ε *h s'*
$xcp' = None \Longrightarrow \tau$*instr P h stk'* (*ins* ! *pc'*)
using *assms*
by(*rule rtranclp.cases*)(*auto elim*!: τ*exec-methE*)

lemma τ*Exec-methr-induct* [*consumes 1*, *case-names refl step*]:
⟦ τ*Exec-methr ci P ins xt t h s s'*;
Q s;
$\bigwedge$*stk loc pc xcp s'*. ⟦ τ*Exec-methr ci P ins xt t h s* (*stk*, *loc*, *pc*, *xcp*); *exec-meth ci P ins xt t h* (*stk*, *loc*, *pc*, *xcp*) ε *h s'*;
$xcp = None \Longrightarrow \tau$*instr P h stk* (*ins* ! *pc*); *Q* (*stk*, *loc*, *pc*, *xcp*) ⟧ $\Longrightarrow$ *Q s'* ⟧
$\Longrightarrow$ *Q s'*
by(*erule* (*1*) *rtranclp-induct*)(*blast elim*: τ*exec-methE*)

lemma τ*Exec-methr-trans*:
⟦ τ*Exec-methr ci P ins xt t h s s'*; τ*Exec-methr ci P ins xt t h s' s''* ⟧ $\Longrightarrow \tau$*Exec-methr ci P ins xt t h s s''*
by(*rule rtranclp-trans*)

lemmas τ*Exec-meth-induct-split* = τ*Exec-methr-induct*[*split-format* (*complete*), *consumes 1*, *case-names* τ*Exec-refl* τ*Exec-step*]

lemma τ*Exec-methr-converse-cases* [*consumes 1*, *case-names refl step*]:
assumes τ*Exec-methr ci P ins xt t h s s'*
obtains $s = s'$
| *stk loc pc xcp s''*
where $s = (stk, loc, pc, xcp)$
exec-meth ci P ins xt t h (*stk*, *loc*, *pc*, *xcp*) ε *h s''*
$xcp = None \Longrightarrow \tau$*instr P h stk* (*ins* ! *pc*)
τ*Exec-methr ci P ins xt t h s'' s'*
using *assms*
by(*erule converse-rtranclpE*)(*blast elim*: τ*exec-methE*)

definition τ*exec-move* ::
('*addr*, '*heap*) *check-instr* ⇒ '*addr J1-prog* ⇒ '*thread-id* ⇒ '*addr expr1* ⇒ '*heap*
⇒ ('*addr val list* × '*addr val list* × *pc* × '*addr option*)
⇒ ('*addr val list* × '*addr val list* × *pc* × '*addr option*) ⇒ *bool*
where
τ*exec-move ci P t e h* =
(λ(*stk*, *loc*, *pc*, *xcp*) *s'*. *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*) ε *h s'* ∧ τ*move2 P h stk e pc xcp*)

definition τ*exec-moves* ::

('addr, 'heap) check-instr ⇒ 'addr J1-prog ⇒ 'thread-id ⇒ 'addr expr1 list ⇒ 'heap
⇒ ('addr val list × 'addr val list × pc × 'addr option)
⇒ ('addr val list × 'addr val list × pc × 'addr option) ⇒ bool
where
τexec-moves ci P t es h =
(λ(stk, loc, pc, xcp) s'. exec-moves ci P t es h (stk, loc, pc, xcp) ε h s' ∧ τmoves2 P h stk es pc xcp)

lemma τexec-moveI:
⟦ exec-move ci P t e h (stk, loc, pc, xcp) ε h s'; τmove2 P h stk e pc xcp ⟧
⟹ τexec-move ci P t e h (stk, loc, pc, xcp) s'
by(simp add: τexec-move-def)

lemma τexec-moveE:
assumes τexec-move ci P t e h (stk, loc, pc, xcp) s'
obtains exec-move ci P t e h (stk, loc, pc, xcp) ε h s' τmove2 P h stk e pc xcp
using assms **by**(simp add: τexec-move-def)

lemma τexec-movesI:
⟦ exec-moves ci P t es h (stk, loc, pc, xcp) ε h s'; τmoves2 P h stk es pc xcp ⟧
⟹ τexec-moves ci P t es h (stk, loc, pc, xcp) s'
by(simp add: τexec-moves-def)

lemma τexec-movesE:
assumes τexec-moves ci P t es h (stk, loc, pc, xcp) s'
obtains exec-moves ci P t es h (stk, loc, pc, xcp) ε h s' τmoves2 P h stk es pc xcp
using assms **by**(simp add: τexec-moves-def)

lemma τexec-move-conv-τexec-meth:
τexec-move ci P t e = τexec-meth ci (compP2 P) (compE2 e) (compxE2 e 0 0) t
by(auto simp add: τexec-move-def exec-move-def τmove2-iff compP2-def intro!: ext τexec-methI elim!: τexec-methE)

lemma τexec-moves-conv-τexec-meth:
τexec-moves ci P t es = τexec-meth ci (compP2 P) (compEs2 es) (compxEs2 es 0 0) t
by(auto simp add: τexec-moves-def exec-moves-def τmoves2-iff compP2-def intro!: ext τexec-methI elim!: τexec-methE)

abbreviation τExec-mover
where τExec-mover ci P t e h == (τexec-move ci P t e h)^**

abbreviation τExec-movet
where τExec-movet ci P t e h == (τexec-move ci P t e h)^++

abbreviation τExec-mover-a
where τExec-mover-a ≡ τExec-mover (Abs-check-instr check-instr')

abbreviation τExec-mover-d
where τExec-mover-d ≡ τExec-mover (Abs-check-instr check-instr)

abbreviation τExec-movet-a
where τExec-movet-a ≡ τExec-movet (Abs-check-instr check-instr')

abbreviation τExec-movet-d
where τExec-movet-d ≡ τExec-movet (Abs-check-instr check-instr)

abbreviation τ*Exec-movesr*
where τ*Exec-movesr ci P t e h* == (τ*exec-moves ci P t e h*)^**

abbreviation τ*Exec-movest*
where τ*Exec-movest ci P t e h* == (τ*exec-moves ci P t e h*)^++

abbreviation τ*Exec-movesr-a*
where τ*Exec-movesr-a* ≡ τ*Exec-movesr* (*Abs-check-instr check-instr'*)

abbreviation τ*Exec-movesr-d*
where τ*Exec-movesr-d* ≡ τ*Exec-movesr* (*Abs-check-instr check-instr*)

abbreviation τ*Exec-movest-a*
where τ*Exec-movest-a* ≡ τ*Exec-movest* (*Abs-check-instr check-instr'*)

abbreviation τ*Exec-movest-d*
where τ*Exec-movest-d* ≡ τ*Exec-movest* (*Abs-check-instr check-instr*)

lemma τ*Execr-refl*: τ*Exec-mover ci P t e h s s*
by(*rule rtranclp.rtrancl-refl*)

lemma τ*Execsr-refl*: τ*Exec-movesr ci P t e h s s*
by(*rule rtranclp.rtrancl-refl*)

lemma τ*Execr-step*:
⟦ τ*Exec-mover ci P t e h s* (*stk'*, *loc'*, *pc'*, *xcp'*);
exec-move ci P t e h (*stk'*, *loc'*, *pc'*, *xcp'*) ε *h s'*;
τ*move2 P h stk' e pc' xcp'* ⟧
⟹ τ*Exec-mover ci P t e h s s'*
by(*rule rtranclp.rtrancl-into-rtrancl*)(*auto elim*: τ*exec-moveI*)

lemma τ*Execsr-step*:
⟦ τ*Exec-movesr ci P t es h s* (*stk'*, *loc'*, *pc'*, *xcp'*);
exec-moves ci P t es h (*stk'*, *loc'*, *pc'*, *xcp'*) ε *h s'*;
τ*moves2 P h stk' es pc' xcp'* ⟧
⟹ τ*Exec-movesr ci P t es h s s'*
by(*rule rtranclp.rtrancl-into-rtrancl*)(*auto elim*: τ*exec-movesI*)

lemma τ*Exect-step*:
⟦ τ*Exec-movet ci P t e h s* (*stk'*, *loc'*, *pc'*, *xcp'*);
exec-move ci P t e h (*stk'*, *loc'*, *pc'*, *xcp'*) ε *h s'*;
τ*move2 P h stk' e pc' xcp'* ⟧
⟹ τ*Exec-movet ci P t e h s s'*
by(*rule tranclp.trancl-into-trancl*)(*auto intro*: τ*exec-moveI*)

lemma τ*Execst-step*:
⟦ τ*Exec-movest ci P t es h s* (*stk'*, *loc'*, *pc'*, *xcp'*);
exec-moves ci P t es h (*stk'*, *loc'*, *pc'*, *xcp'*) ε *h s'*;
τ*moves2 P h stk' es pc' xcp'* ⟧
⟹ τ*Exec-movest ci P t es h s s'*
by(*rule tranclp.trancl-into-trancl*)(*auto intro*: τ*exec-movesI*)

lemmas τ*Execr1step* = τ*Execr-step*[*OF* τ*Execr-refl*]

lemmas *τExecr2step* = *τExecr-step*[*OF τExecr-step*, *OF τExecr-refl*]
lemmas *τExecr3step* = *τExecr-step*[*OF τExecr-step*, *OF τExecr-step*, *OF τExecr-refl*]

lemmas *τExecsr1step* = *τExecsr-step*[*OF τExecsr-refl*]
lemmas *τExecsr2step* = *τExecsr-step*[*OF τExecsr-step*, *OF τExecsr-refl*]
lemmas *τExecsr3step* = *τExecsr-step*[*OF τExecsr-step*, *OF τExecsr-step*, *OF τExecsr-refl*]

lemma *τExect1step*:
⟦ *exec-move ci P t e h s ε h s′*;
τmove2 P h (*fst s*) *e* (*fst* (*snd* (*snd s*))) (*snd* (*snd* (*snd s*))) ⟧
⟹ *τExec-movet ci P t e h s s′*
by(*rule tranclp.r-into-trancl*)(*cases s*, *auto intro*: *τexec-moveI*)

lemmas *τExect2step* = *τExect-step*[*OF τExect1step*]
lemmas *τExect3step* = *τExect-step*[*OF τExect-step*, *OF τExect1step*]

lemma *τExecst1step*:
⟦ *exec-moves ci P t es h s ε h s′*;
τmoves2 P h (*fst s*) *es* (*fst* (*snd* (*snd s*))) (*snd* (*snd* (*snd s*))) ⟧
⟹ *τExec-movest ci P t es h s s′*
by(*rule tranclp.r-into-trancl*)(*cases s*, *auto intro*: *τexec-movesI*)

lemmas *τExecst2step* = *τExecst-step*[*OF τExecst1step*]
lemmas *τExecst3step* = *τExecst-step*[*OF τExecst-step*, *OF τExecst1step*]

lemma *τExecr-induct* [*consumes 1*, *case-names refl step*]:
assumes *major*: *τExec-mover ci P t e h* (*stk*, *loc*, *pc*, *xcp*) (*stk″*, *loc″*, *pc″*, *xcp″*)
and *refl*: *Q stk loc pc xcp*
and *step*: ⋀*stk′ loc′ pc′ xcp′ stk″ loc″ pc″ xcp″*.
⟦ *τExec-mover ci P t e h* (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*);
τexec-move ci P t e h (*stk′*, *loc′*, *pc′*, *xcp′*) (*stk″*, *loc″*, *pc″*, *xcp″*); *Q stk′ loc′ pc′ xcp′* ⟧
⟹ *Q stk″ loc″ pc″ xcp″*
shows *Q stk″ loc″ pc″ xcp″*
using *major refl*
by(*rule rtranclp-induct4*)(*rule step*)

lemma *τExecsr-induct* [*consumes 1*, *case-names refl step*]:
assumes *major*: *τExec-movesr ci P t es h* (*stk*, *loc*, *pc*, *xcp*) (*stk″*, *loc″*, *pc″*, *xcp″*)
and *refl*: *Q stk loc pc xcp*
and *step*: ⋀*stk′ loc′ pc′ xcp′ stk″ loc″ pc″ xcp″*.
⟦ *τExec-movesr ci P t es h* (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*);
τexec-moves ci P t es h (*stk′*, *loc′*, *pc′*, *xcp′*) (*stk″*, *loc″*, *pc″*, *xcp″*); *Q stk′ loc′ pc′ xcp′* ⟧
⟹ *Q stk″ loc″ pc″ xcp″*
shows *Q stk″ loc″ pc″ xcp″*
using *major refl*
by(*rule rtranclp-induct4*)(*rule step*)

lemma *τExect-induct* [*consumes 1*, *case-names base step*]:
assumes *major*: *τExec-movet ci P t e h* (*stk*, *loc*, *pc*, *xcp*) (*stk″*, *loc″*, *pc″*, *xcp″*)
and *base*: ⋀*stk′ loc′ pc′ xcp′*. *τexec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*) ⟹ *Q stk′ loc′ pc′ xcp′*
and *step*: ⋀*stk′ loc′ pc′ xcp′ stk″ loc″ pc″ xcp″*.
⟦ *τExec-movet ci P t e h* (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*);
τexec-move ci P t e h (*stk′*, *loc′*, *pc′*, *xcp′*) (*stk″*, *loc″*, *pc″*, *xcp″*); *Q stk′ loc′ pc′ xcp′* ⟧

$\Longrightarrow Q\ stk''\ loc''\ pc''\ xcp''$
shows $Q\ stk''\ loc''\ pc''\ xcp''$
using *major*
by(*rule tranclp-induct4*)(*erule base step*)+

lemma τ*Execst-induct* [*consumes 1*, *case-names base step*]:
assumes *major*: τ*Exec-movest ci P t es h* (*stk*, *loc*, *pc*, *xcp*) (*stk''*, *loc''*, *pc''*, *xcp''*)
and *base*: $\bigwedge stk'\ loc'\ pc'\ xcp'.\ \tau$*exec-moves ci P t es h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*) $\Longrightarrow Q\ stk'\ loc'\ pc'\ xcp'$
and *step*: $\bigwedge stk'\ loc'\ pc'\ xcp'\ stk''\ loc''\ pc''\ xcp''.$
$[\![\ \tau$*Exec-movest ci P t es h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*);
τ*exec-moves ci P t es h* (*stk'*, *loc'*, *pc'*, *xcp'*) (*stk''*, *loc''*, *pc''*, *xcp''*); $Q\ stk'\ loc'\ pc'\ xcp'\ ]\!]$
$\Longrightarrow Q\ stk''\ loc''\ pc''\ xcp''$
shows $Q\ stk''\ loc''\ pc''\ xcp''$
using *major*
by(*rule tranclp-induct4*)(*erule base step*)+

lemma τ*Exec-mover-*τ*Exec-methr*:
τ*Exec-mover ci P t e* = τ*Exec-methr ci* (*compP2 P*) (*compE2 e*) (*compxE2 e 0 0*) *t*
by(*simp only*: τ*exec-move-conv-*τ*exec-meth*)

lemma τ*Exec-movesr-*τ*Exec-methr*:
τ*Exec-movesr ci P t es* = τ*Exec-methr ci* (*compP2 P*) (*compEs2 es*) (*compxEs2 es 0 0*) *t*
by(*simp only*: τ*exec-moves-conv-*τ*exec-meth*)

lemma τ*Exec-movet-*τ*Exec-metht*:
τ*Exec-movet ci P t e* = τ*Exec-metht ci* (*compP2 P*) (*compE2 e*) (*compxE2 e 0 0*) *t*
by(*simp only*: τ*exec-move-conv-*τ*exec-meth*)

lemma τ*Exec-movest-*τ*Exec-metht*:
τ*Exec-movest ci P t es* = τ*Exec-metht ci* (*compP2 P*) (*compEs2 es*) (*compxEs2 es 0 0*) *t*
by(*simp only*: τ*exec-moves-conv-*τ*exec-meth*)

lemma τ*Exec-mover-trans*:
$[\![\ \tau$*Exec-mover ci P t e h s s'*; τ*Exec-mover ci P t e h s' s''* $]\!] \Longrightarrow \tau$*Exec-mover ci P t e h s s''*
by(*rule rtranclp-trans*)

lemma τ*Exec-movesr-trans*:
$[\![\ \tau$*Exec-movesr ci P t es h s s'*; τ*Exec-movesr ci P t es h s' s''* $]\!] \Longrightarrow \tau$*Exec-movesr ci P t es h s s''*
by(*rule rtranclp-trans*)

lemma τ*Exec-movet-trans*:
$[\![\ \tau$*Exec-movet ci P t e h s s'*; τ*Exec-movet ci P t e h s' s''* $]\!] \Longrightarrow \tau$*Exec-movet ci P t e h s s''*
by(*rule tranclp-trans*)

lemma τ*Exec-movest-trans*:
$[\![\ \tau$*Exec-movest ci P t es h s s'*; τ*Exec-movest ci P t es h s' s''* $]\!] \Longrightarrow \tau$*Exec-movest ci P t es h s s''*
by(*rule tranclp-trans*)

lemma τ*exec-move-into-*τ*exec-moves*:
τ*exec-move ci P t e h s s'* $\Longrightarrow \tau$*exec-moves ci P t* (*e # es*) *h s s'*
by(*cases s*)(*auto elim*!: τ*exec-moveE intro*!: τ*exec-movesI simp add*: *exec-move-def exec-moves-def intro*: τ*moves2Hd*)

lemma *τExec-mover-τExec-movesr*:
τExec-mover ci P t e h s s′ ⟹ τExec-movesr ci P t (e # es) h s s′
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl τexec-move-into-τexec-moves*)+

lemma *τExec-movet-τExec-movest*:
τExec-movet ci P t e h s s′ ⟹ τExec-movest ci P t (e # es) h s s′
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl τexec-move-into-τexec-moves*)+

lemma *exec-moves-append*: *exec-moves ci P t es h s ta h′ s′ ⟹ exec-moves ci P t (es @ es′) h s ta h′ s′*
by(*auto simp add*: *exec-moves-def*)

lemma *τexec-moves-append*: *τexec-moves ci P t es h s s′ ⟹ τexec-moves ci P t (es @ es′) h s s′*
by(*cases s*)(*auto elim*!: *τexec-movesE intro*!: *τexec-movesI exec-moves-append*)

lemma *τExec-movesr-append* [*intro*]:
τExec-movesr ci P t es h s s′ ⟹ τExec-movesr ci P t (es @ es′) h s s′
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl τexec-moves-append*)+

lemma *τExec-movest-append* [*intro*]:
τExec-movest ci P t es h s s′ ⟹ τExec-movest ci P t (es @ es′) h s s′
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl τexec-moves-append*)+

lemma *append-exec-moves*:
assumes *len*: *length vs = length es′*
and *exec*: *exec-moves ci P t es h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
shows *exec-moves ci P t* (*es′* @ *es*) *h* ((*stk* @ *vs*), *loc*, (*length* (*compEs2 es′*) + *pc*), *xcp*) *ta h′* ((*stk′* @ *vs*), *loc′*, (*length* (*compEs2 es′*) + *pc′*), *xcp′*)
proof −
from *exec* **have** *exec-meth ci* (*compP2 P*) (*compEs2 es*) (*compxEs2 es 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
unfolding *exec-moves-def* **.**
hence *exec-meth ci* (*compP2 P*) (*compEs2 es*) (*stack-xlift* (*length vs*) (*compxEs2 es 0 0*)) *t h* ((*stk* @ *vs*), *loc*, *pc*, *xcp*) *ta h′* ((*stk′* @ *vs*), *loc′*, *pc′*, *xcp′*) **by**(*rule exec-meth-stk-offer*)
hence *exec-meth ci* (*compP2 P*) (*compEs2 es′* @ *compEs2 es*) (*compxEs2 es′ 0 0* @ *shift* (*length* (*compEs2 es′*)) (*stack-xlift* (*length* (*vs*)) (*compxEs2 es 0 0*))) *t h* ((*stk* @ *vs*), *loc*, (*length* (*compEs2 es′*) + *pc*), *xcp*) *ta h′* ((*stk′* @ *vs*), *loc′*, (*length* (*compEs2 es′*) + *pc′*), *xcp′*)
by(*rule append-exec-meth-xt*) *auto*
thus *?thesis* **by**(*simp add*: *exec-moves-def stack-xlift-compxEs2 shift-compxEs2 len*)
qed

lemma *append-τexec-moves*:
⟦ *length vs = length es′*;
τexec-moves ci P t es h (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*) ⟧
⟹ *τexec-moves ci P t* (*es′* @ *es*) *h* ((*stk* @ *vs*), *loc*, (*length* (*compEs2 es′*) + *pc*), *xcp*) ((*stk′* @ *vs*), *loc′*, (*length* (*compEs2 es′*) + *pc′*), *xcp′*)
by(*auto elim*!: *τexec-movesE intro*: *τexec-movesI append-exec-moves τmoves2-stk-append append-τmoves2*)

lemma *append-τExec-movesr*:
assumes *len*: *length vs = length es′*
shows *τExec-movesr ci P t es h* (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*)
⟹ *τExec-movesr ci P t* (*es′* @ *es*) *h* ((*stk* @ *vs*), *loc*, (*length* (*compEs2 es′*) + *pc*), *xcp*) ((*stk′* @ *vs*), *loc′*, (*length* (*compEs2 es′*) + *pc′*), *xcp′*)

by(*induct rule*: *rtranclp-induct4*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl append-*τ*exec-moves*[*OF len*])+

lemma *append-*τ*Exec-movest*:
assumes *len*: *length vs* = *length es'*
shows τ*Exec-movest ci P t es h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*Exec-movest ci P t* (*es'* @ *es*) *h* ((*stk* @ *vs*), *loc*, (*length* (*compEs2 es'*) + *pc*), *xcp*) ((*stk'* @ *vs*), *loc'*, (*length* (*compEs2 es'*) + *pc'*), *xcp'*)
by(*induct rule*: *tranclp-induct4*)(*blast intro*: *tranclp.trancl-into-trancl append-*τ*exec-moves*[*OF len*])+

lemma *NewArray-*τ*execI*:
τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t* (*newA T*⌊*e*⌉) *h s s'*
by(*cases s*)(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-newArrayI*)

lemma *Cast-*τ*execI*:
τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t* (*Cast T e*) *h s s'*
by(*cases s*)(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-CastI*)

lemma *InstanceOf-*τ*execI*:
τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t* (*e instanceof T*) *h s s'*
by(*cases s*)(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-InstanceOfI*)

lemma *BinOp-*τ*execI1*:
τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t* (*e «bop» e'*) *h s s'*
by(*cases s*)(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-BinOpI1*)

lemma *BinOp-*τ*execI2*:
τ*exec-move ci P t e' h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*exec-move ci P t* (*e «bop» e'*) *h* ((*stk* @ [*v*]), *loc*, (*length* (*compE2 e*) + *pc*), *xcp*) ((*stk'* @ [*v*]), *loc'*, (*length* (*compE2 e*) + *pc'*), *xcp'*)
by(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-BinOpI2* τ*move2-stk-append*)

lemma *LAss-*τ*execI*:
τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t* (*V* := *e*) *h s s'*
by(*cases s*)(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-LAssI*)

lemma *AAcc-*τ*execI1*:
τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t* (*e*⌊*i*⌉) *h s s'*
by(*cases s*)(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-AAccI1*)

lemma *AAcc-*τ*execI2*:
τ*exec-move ci P t e' h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*exec-move ci P t* (*e*⌊*e'*⌉) *h* ((*stk* @ [*v*]), *loc*, (*length* (*compE2 e*) + *pc*), *xcp*) ((*stk'* @ [*v*]), *loc'*, (*length* (*compE2 e*) + *pc'*), *xcp'*)
by(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-AAccI2* τ*move2-stk-append*)

lemma *AAss-*τ*execI1*:
τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t* (*e*⌊*i*⌉ := *e'*) *h s s'*
by(*cases s*)(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-AAssI1*)

lemma *AAss-*τ*execI2*:
τ*exec-move ci P t e' h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*exec-move ci P t* (*e*⌊*e''*⌉ := *e'''*) *h* ((*stk* @ [*v*]), *loc*, (*length* (*compE2 e*) + *pc*), *xcp*) ((*stk'* @ [*v*]), *loc'*, (*length* (*compE2 e*) + *pc'*), *xcp'*)

by(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-AAssI2* τ*move2-stk-append*)

lemma *AAss-*τ*execI3*:
τ*exec-move ci P t e'' h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*exec-move ci P t* (*e*⌊*e'*⌉ := *e''*) *h* ((*stk* @ [*v*, *v'*]), *loc*, (*length* (*compE2 e*) + *length* (*compE2 e'*) + *pc*), *xcp*) ((*stk'* @ [*v*, *v'*]), *loc'*, (*length* (*compE2 e*) + *length* (*compE2 e'*) + *pc'*), *xcp'*)
by(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-AAssI3* τ*move2-stk-append*)

lemma *ALength-*τ*execI*:
τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t* (*e*·*length*) *h s s'*
by(*cases s*)(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-ALengthI*)

lemma *FAcc-*τ*execI*:
τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t* (*e*·*F*{*D*}) *h s s'*
by(*cases s*)(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-FAccI*)

lemma *FAss-*τ*execI1*:
τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t* (*e*·*F*{*D*} := *e'*) *h s s'*
by(*cases s*)(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-FAssI1*)

lemma *FAss-*τ*execI2*:
τ*exec-move ci P t e' h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*exec-move ci P t* (*e*·*F*{*D*} := *e'*) *h* ((*stk* @ [*v*]), *loc*, (*length* (*compE2 e*) + *pc*), *xcp*) ((*stk'* @ [*v*]), *loc'*, (*length* (*compE2 e*) + *pc'*), *xcp'*)
by(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-FAssI2* τ*move2-stk-append*)

lemma *CAS-*τ*execI1*:
τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t* (*e*·*compareAndSwap*(*D*·*F*, *e'*, *e''*)) *h s s'*
by(*cases s*)(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-CASI1*)

lemma *CAS-*τ*execI2*:
τ*exec-move ci P t e' h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*exec-move ci P t* (*e*·*compareAndSwap*(*D*·*F*, *e'*, *e''*)) *h* ((*stk* @ [*v*]), *loc*, (*length* (*compE2 e*) + *pc*), *xcp*) ((*stk'* @ [*v*]), *loc'*, (*length* (*compE2 e*) + *pc'*), *xcp'*)
by(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-CASI2* τ*move2-stk-append*)

lemma *CAS-*τ*execI3*:
τ*exec-move ci P t e'' h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*exec-move ci P t* (*e*·*compareAndSwap*(*D*·*F*, *e'*, *e''*)) *h* ((*stk* @ [*v*, *v'*]), *loc*, (*length* (*compE2 e*) + *length* (*compE2 e'*) + *pc*), *xcp*) ((*stk'* @ [*v*, *v'*]), *loc'*, (*length* (*compE2 e*) + *length* (*compE2 e'*) + *pc'*), *xcp'*)
by(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-CASI3* τ*move2-stk-append*)

lemma *Call-*τ*execI1*:
τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t* (*e*·*M*(*es*)) *h s s'*
by(*cases s*)(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-CallI1*)

lemma *Call-*τ*execI2*:
τ*exec-moves ci P t es h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*exec-move ci P t* (*e*·*M*(*es*)) *h* ((*stk* @ [*v*]), *loc*, (*length* (*compE2 e*) + *pc*), *xcp*) ((*stk'* @ [*v*]), *loc'*, (*length* (*compE2 e*) + *pc'*), *xcp'*)
by(*blast elim*: τ*exec-movesE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-CallI2* τ*moves2-stk-append*)

lemma *Block-*τ*execI-Some*:

τ*exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*exec-move ci P t* {*V*:*T*=⌊*v*⌋; *e*} *h* (*stk*, *loc*, *Suc* (*Suc pc*), *xcp*) (*stk'*, *loc'*, *Suc* (*Suc pc'*), *xcp'*)
by(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-BlockSomeI*)

lemma *Block-*τ*execI-None*:
τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t* {*V*:*T*=*None*; *e*} *h s s'*
by(*cases s*)(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-BlockNoneI*)

lemma *Sync-*τ*execI*:
τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t* (*sync*$_V$ (*e*) *e'*) *h s s'*
by(*cases s*)(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-SyncI1*)

lemma *Insync-*τ*execI*:
τ*exec-move ci P t e' h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*exec-move ci P t* (*sync*$_V$ (*e*) *e'*) *h* (*stk*, *loc*, *Suc* (*Suc* (*Suc* (*length* (*compE2 e*) + *pc*))), *xcp*) (*stk'*, *loc'*, *Suc* (*Suc* (*Suc* (*length* (*compE2 e*) + *pc'*))), *xcp'*)
by(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-SyncI2*)

lemma *Seq-*τ*execI1*:
τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t* (*e*;; *e'*) *h s s'*
by(*cases s*)(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-SeqI1*)

lemma *Seq-*τ*execI2*:
τ*exec-move ci P t e' h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*exec-move ci P t* (*e*;; *e'*) *h* (*stk*, *loc*, *Suc* (*length* (*compE2 e*) + *pc*), *xcp*) (*stk'*, *loc'*, *Suc* (*length* (*compE2 e*) + *pc'*), *xcp'*)
by(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-SeqI2*)

lemma *Cond-*τ*execI1*:
τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t* (*if* (*e*) *e'* *else e''*) *h s s'*
by(*cases s*)(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-CondI1*)

lemma *Cond-*τ*execI2*:
τ*exec-move ci P t e' h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*exec-move ci P t* (*if* (*e*) *e'* *else e''*) *h* (*stk*, *loc*, *Suc* (*length* (*compE2 e*) + *pc*), *xcp*) (*stk'*, *loc'*, *Suc* (*length* (*compE2 e*) + *pc'*), *xcp'*)
by(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-CondI2*)

lemma *Cond-*τ*execI3*:
τ*exec-move ci P t e'' h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*exec-move ci P t* (*if* (*e*) *e'* *else e''*) *h* (*stk*, *loc*, *Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e'*) + *pc*)), *xcp*) (*stk'*, *loc'*, *Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e'*) + *pc'*)), *xcp'*)
by(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-CondI3*)

lemma *While-*τ*execI1*:
τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t* (*while* (*e*) *e'*) *h s s'*
by(*cases s*)(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-WhileI1*)

lemma *While-*τ*execI2*:
τ*exec-move ci P t e' h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*exec-move ci P t* (*while* (*e*) *e'*) *h* (*stk*, *loc*, *Suc* (*length* (*compE2 e*) + *pc*), *xcp*) (*stk'*, *loc'*, *Suc* (*length* (*compE2 e*) + *pc'*), *xcp'*)
by(*blast elim*: τ*exec-moveE intro*: τ*exec-moveI* τ*move2-*τ*moves2.intros exec-move-WhileI2*)

lemma *Throw-τexecI*:
τexec-move ci P t e h s s′ ⟹ τexec-move ci P t (throw e) h s s′
by(*cases s*)(*blast elim*: *τexec-moveE intro*: *τexec-moveI τmove2-τmoves2.intros exec-move-ThrowI*)

lemma *Try-τexecI1*:
τexec-move ci P t e h s s′ ⟹ τexec-move ci P t (try e catch(C V) e′) h s s′
by(*cases s*)(*blast elim*: *τexec-moveE intro*: *τexec-moveI τmove2-τmoves2.intros exec-move-TryI1*)

lemma *Try-τexecI2*:
τexec-move ci P t e′ h (stk, loc, pc, xcp) (stk′, loc′, pc′, xcp′)
⟹ τexec-move ci P t (try e catch(C V) e′) h (stk, loc, Suc (Suc (length (compE2 e) + pc)), xcp) (stk′, loc′, Suc (Suc (length (compE2 e) + pc′)), xcp′)
by(*blast elim*: *τexec-moveE intro*: *τexec-moveI τmove2-τmoves2.intros exec-move-TryI2*)

lemma *NewArray-τExecrI*:
τExec-mover ci P t e h s s′ ⟹ τExec-mover ci P t (newA T⌊e⌉) h s s′
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl NewArray-τexecI*)+

lemma *Cast-τExecrI*:
τExec-mover ci P t e h s s′ ⟹ τExec-mover ci P t (Cast T e) h s s′
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl Cast-τexecI*)+

lemma *InstanceOf-τExecrI*:
τExec-mover ci P t e h s s′ ⟹ τExec-mover ci P t (e instanceof T) h s s′
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl InstanceOf-τexecI*)+

lemma *BinOp-τExecrI1*:
τExec-mover ci P t e1 h s s′ ⟹ τExec-mover ci P t (e1 «bop» e2) h s s′
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl BinOp-τexecI1*)+

lemma *BinOp-τExecrI2*:
τExec-mover ci P t e2 h (stk, loc, pc, xcp) (stk′, loc′, pc′, xcp′)
⟹ τExec-mover ci P t (e «bop» e2) h ((stk @ [v]), loc, (length (compE2 e) + pc), xcp) ((stk′ @ [v]), loc′, (length (compE2 e) + pc′), xcp′)
by(*induct rule*: *τExecr-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl BinOp-τexecI2*)+

lemma *LAss-τExecrI*:
τExec-mover ci P t e h s s′ ⟹ τExec-mover ci P t (V := e) h s s′
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl LAss-τexecI*)+

lemma *AAcc-τExecrI1*:
τExec-mover ci P t e h s s′ ⟹ τExec-mover ci P t (e⌊i⌉) h s s′
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl AAcc-τexecI1*)+

lemma *AAcc-τExecrI2*:
τExec-mover ci P t i h (stk, loc, pc, xcp) (stk′, loc′, pc′, xcp′)
⟹ τExec-mover ci P t (a⌊i⌉) h ((stk @ [v]), loc, (length (compE2 a) + pc), xcp) ((stk′ @ [v]), loc′, (length (compE2 a) + pc′), xcp′)
by(*induct rule*: *τExecr-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl AAcc-τexecI2*)+

lemma *AAss-τExecrI1*:
τExec-mover ci P t e h s s′ ⟹ τExec-mover ci P t (e⌊i⌉ := e′) h s s′

by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl AAss-τexecI1*)+

lemma *AAss-τExecrI2*:
τExec-mover ci P t i h (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*)
⟹ *τExec-mover ci P t* (*a*⌊*i*⌉ := *e*) *h* ((*stk* @ [*v*]), *loc*, (*length* (*compE2 a*) + *pc*), *xcp*) ((*stk′* @ [*v*]), *loc′*, (*length* (*compE2 a*) + *pc′*), *xcp′*)
by(*induct rule*: *τExecr-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl AAss-τexecI2*)+

lemma *AAss-τExecrI3*:
τExec-mover ci P t e h (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*)
⟹ *τExec-mover ci P t* (*a*⌊*i*⌉ := *e*) *h* ((*stk* @ [*v*, *v′*]), *loc*, (*length* (*compE2 a*) + *length* (*compE2 i*) + *pc*), *xcp*) ((*stk′* @ [*v*, *v′*]), *loc′*, (*length* (*compE2 a*) + *length* (*compE2 i*) + *pc′*), *xcp′*)
by(*induct rule*: *τExecr-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl AAss-τexecI3*)+

lemma *ALength-τExecrI*:
τExec-mover ci P t e h s s′ ⟹ *τExec-mover ci P t* (*e*·*length*) *h s s′*
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl ALength-τexecI*)+

lemma *FAcc-τExecrI*:
τExec-mover ci P t e h s s′ ⟹ *τExec-mover ci P t* (*e*·*F*{*D*}) *h s s′*
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl FAcc-τexecI*)+

lemma *FAss-τExecrI1*:
τExec-mover ci P t e h s s′ ⟹ *τExec-mover ci P t* (*e*·*F*{*D*} := *e′*) *h s s′*
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl FAss-τexecI1*)+

lemma *FAss-τExecrI2*:
τExec-mover ci P t e′ h (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*)
⟹ *τExec-mover ci P t* (*e*·*F*{*D*} := *e′*) *h* ((*stk* @ [*v*]), *loc*, (*length* (*compE2 e*) + *pc*), *xcp*) ((*stk′* @ [*v*]), *loc′*, (*length* (*compE2 e*) + *pc′*), *xcp′*)
by(*induct rule*: *τExecr-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl FAss-τexecI2*)+

lemma *CAS-τExecrI1*:
τExec-mover ci P t e h s s′ ⟹ *τExec-mover ci P t* (*e*·*compareAndSwap*(*D*·*F*, *e′*, *e″*)) *h s s′*
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl CAS-τexecI1*)+

lemma *CAS-τExecrI2*:
τExec-mover ci P t e′ h (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*)
⟹ *τExec-mover ci P t* (*e*·*compareAndSwap*(*D*·*F*, *e′*, *e″*)) *h* ((*stk* @ [*v*]), *loc*, (*length* (*compE2 e*) + *pc*), *xcp*) ((*stk′* @ [*v*]), *loc′*, (*length* (*compE2 e*) + *pc′*), *xcp′*)
by(*induct rule*: *τExecr-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl CAS-τexecI2*)+

lemma *CAS-τExecrI3*:
τExec-mover ci P t e″ h (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*)
⟹ *τExec-mover ci P t* (*e*·*compareAndSwap*(*D*·*F*, *e′*, *e″*)) *h* ((*stk* @ [*v*, *v′*]), *loc*, (*length* (*compE2 e*) + *length* (*compE2 e′*) + *pc*), *xcp*) ((*stk′* @ [*v*, *v′*]), *loc′*, (*length* (*compE2 e*) + *length* (*compE2 e′*) + *pc′*), *xcp′*)
by(*induct rule*: *τExecr-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl CAS-τexecI3*)+

lemma *Call-τExecrI1*:
τExec-mover ci P t obj h s s′ ⟹ *τExec-mover ci P t* (*obj*·*M′*(*es*)) *h s s′*
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl Call-τexecI1*)+

lemma *Call-τExecrI2*:

τ*Exec-movesr ci P t es h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*Exec-mover ci P t* (*obj*·*M'*(*es*)) *h* ((*stk* @ [*v*]), *loc*, (*length* (*compE2 obj*) + *pc*), *xcp*) ((*stk'* @ [*v*]), *loc'*, (*length* (*compE2 obj*) + *pc'*), *xcp'*)
by(*induct rule*: τ*Execsr-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl Call-*τ*execI2*)+

lemma *Block-*τ*ExecrI-Some*:
τ*Exec-mover ci P t e h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*Exec-mover ci P t* {*V*:*T*=⌊*v*⌋; *e*} *h* (*stk*, *loc*, (*Suc* (*Suc pc*)), *xcp*) (*stk'*, *loc'*, (*Suc* (*Suc pc'*)), *xcp'*)
by(*induct rule*: τ*Execr-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl Block-*τ*execI-Some*)+

lemma *Block-*τ*ExecrI-None*:
τ*Exec-mover ci P t e h s s'* $\Longrightarrow$ τ*Exec-mover ci P t* {*V*:*T*=*None*; *e*} *h s s'*
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl Block-*τ*execI-None*)+

lemma *Sync-*τ*ExecrI*:
τ*Exec-mover ci P t e h s s'* $\Longrightarrow$ τ*Exec-mover ci P t* (*sync*$_V$ (*e*) *e'*) *h s s'*
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl Sync-*τ*execI*)+

lemma *Insync-*τ*ExecrI*:
τ*Exec-mover ci P t e' h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*Exec-mover ci P t* (*sync*$_V$ (*e*) *e'*) *h* (*stk*, *loc*, (*Suc* (*Suc* (*Suc* (*length* (*compE2 e*) + *pc*)))), *xcp*) (*stk'*, *loc'*, (*Suc* (*Suc* (*Suc* (*length* (*compE2 e*) + *pc'*)))), *xcp'*)
by(*induct rule*: τ*Execr-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl Insync-*τ*execI*)+

lemma *Seq-*τ*ExecrI1*:
τ*Exec-mover ci P t e h s s'* $\Longrightarrow$ τ*Exec-mover ci P t* (*e*;;*e'*) *h s s'*
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl Seq-*τ*execI1*)+

lemma *Seq-*τ*ExecrI2*:
τ*Exec-mover ci P t e h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'* ,*xcp'*) $\Longrightarrow$
τ*Exec-mover ci P t* (*e'*;;*e*) *h* (*stk*, *loc*, (*Suc* (*length* (*compE2 e'*) + *pc*)), *xcp*) (*stk'*, *loc'*, (*Suc* (*length* (*compE2 e'*) + *pc'*)), *xcp'*)
by(*induct rule*: τ*Execr-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl Seq-*τ*execI2*)+

lemma *Cond-*τ*ExecrI1*:
τ*Exec-mover ci P t e h s s'* $\Longrightarrow$ τ*Exec-mover ci P t* (*if* (*e*) *e1 else e2*) *h s s'*
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl Cond-*τ*execI1*)+

lemma *Cond-*τ*ExecrI2*:
τ*Exec-mover ci P t e1 h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*) $\Longrightarrow$
τ*Exec-mover ci P t* (*if* (*e*) *e1 else e2*) *h* (*stk*, *loc*, (*Suc* (*length* (*compE2 e*) + *pc*)), *xcp*) (*stk'*, *loc'*, (*Suc* (*length* (*compE2 e*) + *pc'*)), *xcp'*)
by(*induct rule*: τ*Execr-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl Cond-*τ*execI2*)+

lemma *Cond-*τ*ExecrI3*:
τ*Exec-mover ci P t e2 h* (*stk*, *loc* ,*pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*) $\Longrightarrow$
τ*Exec-mover ci P t* (*if* (*e*) *e1 else e2*) *h* (*stk*, *loc*, (*Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + *pc*))), *xcp*) (*stk'*, *loc'*, (*Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + *pc'*))), *xcp'*)
by(*induct rule*: τ*Execr-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl Cond-*τ*execI3*)+

lemma *While-*τ*ExecrI1*:
τ*Exec-mover ci P t c h s s'* $\Longrightarrow$ τ*Exec-mover ci P t* (*while* (*c*) *e*) *h s s'*
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl While-*τ*execI1*)+

lemma *While-τExecrI2*:
τExec-mover ci P t E h (stk, loc, pc, xcp) (stk′, loc′, pc′, xcp′)
⟹ τExec-mover ci P t (while (c) E) h (stk, loc ,(Suc (length (compE2 c) + pc)), xcp) (stk′, loc′, (Suc (length (compE2 c) + pc′)), xcp′)
by(*induct rule*: *τExecr-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl While-τexecI2*)+

lemma *Throw-τExecrI*:
τExec-mover ci P t e h s s′ ⟹ τExec-mover ci P t (throw e) h s s′
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl Throw-τexecI*)+

lemma *Try-τExecrI1*:
τExec-mover ci P t E h s s′ ⟹ τExec-mover ci P t (try E catch(C′ V) e) h s s′
by(*induct rule*: *rtranclp-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl Try-τexecI1*)+

lemma *Try-τExecrI2*:
τExec-mover ci P t e h (stk, loc, pc, xcp) (stk′, loc′, pc′, xcp′)
⟹ τExec-mover ci P t (try E catch(C′ V) e) h (stk, loc, (Suc (Suc (length (compE2 E) + pc))), xcp) (stk′, loc′, (Suc (Suc (length (compE2 E) + pc′))), xcp′)
by(*induct rule*: *τExecr-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl Try-τexecI2*)+

lemma *NewArray-τExectI*:
τExec-movet ci P t e h s s′ ⟹ τExec-movet ci P t (newA T⌊e⌉) h s s′
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl NewArray-τexecI*)+

lemma *Cast-τExectI*:
τExec-movet ci P t e h s s′ ⟹ τExec-movet ci P t (Cast T e) h s s′
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl Cast-τexecI*)+

lemma *InstanceOf-τExectI*:
τExec-movet ci P t e h s s′ ⟹ τExec-movet ci P t (e instanceof T) h s s′
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl InstanceOf-τexecI*)+

lemma *BinOp-τExectI1*:
τExec-movet ci P t e1 h s s′ ⟹ τExec-movet ci P t (e1 «bop» e2) h s s′
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl BinOp-τexecI1*)+

lemma *BinOp-τExectI2*:
τExec-movet ci P t e2 h (stk, loc, pc, xcp) (stk′, loc′, pc′, xcp′)
⟹ τExec-movet ci P t (e «bop» e2) h ((stk @ [v]), loc, (length (compE2 e) + pc), xcp) ((stk′ @ [v]), loc′, (length (compE2 e) + pc′), xcp′)
by(*induct rule*: *τExect-induct*)(*blast intro*: *tranclp.trancl-into-trancl BinOp-τexecI2*)+

lemma *LAss-τExectI*:
τExec-movet ci P t e h s s′ ⟹ τExec-movet ci P t (V := e) h s s′
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl LAss-τexecI*)+

lemma *AAcc-τExectI1*:
τExec-movet ci P t e h s s′ ⟹ τExec-movet ci P t (e⌊i⌉) h s s′
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl AAcc-τexecI1*)+

lemma *AAcc-τExectI2*:
τExec-movet ci P t i h (stk, loc, pc, xcp) (stk′, loc′, pc′, xcp′)

$\Longrightarrow \tau Exec\text{-}movet\ ci\ P\ t\ (a\lfloor i\rceil)\ h\ ((stk\ @\ [v]),\ loc,\ (length\ (compE2\ a)\ +\ pc),\ xcp)\ ((stk'\ @\ [v]),\ loc',\ (length\ (compE2\ a)\ +\ pc'),\ xcp')$
by(*induct rule*: $\tau Exect\text{-}induct$)(*blast intro*: *tranclp.trancl-into-trancl AAcc-*τ*execI2*)+

lemma *AAss-*τ*ExectI1*:
$\tau Exec\text{-}movet\ ci\ P\ t\ e\ h\ s\ s' \Longrightarrow \tau Exec\text{-}movet\ ci\ P\ t\ (e\lfloor i\rceil := e')\ h\ s\ s'$
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl AAss-*τ*execI1*)+

lemma *AAss-*τ*ExectI2*:
$\tau Exec\text{-}movet\ ci\ P\ t\ i\ h\ (stk,\ loc,\ pc,\ xcp)\ (stk',\ loc',\ pc',\ xcp')$
$\Longrightarrow \tau Exec\text{-}movet\ ci\ P\ t\ (a\lfloor i\rceil := e)\ h\ ((stk\ @\ [v]),\ loc,\ (length\ (compE2\ a)\ +\ pc),\ xcp)\ ((stk'\ @\ [v]),\ loc',\ (length\ (compE2\ a)\ +\ pc'),\ xcp')$
by(*induct rule*: $\tau Exect\text{-}induct$)(*blast intro*: *tranclp.trancl-into-trancl AAss-*τ*execI2*)+

lemma *AAss-*τ*ExectI3*:
$\tau Exec\text{-}movet\ ci\ P\ t\ e\ h\ (stk,\ loc,\ pc,\ xcp)\ (stk',\ loc',\ pc',\ xcp')$
$\Longrightarrow \tau Exec\text{-}movet\ ci\ P\ t\ (a\lfloor i\rceil := e)\ h\ ((stk\ @\ [v,\ v']),\ loc,\ (length\ (compE2\ a)\ +\ length\ (compE2\ i)\ +\ pc),\ xcp)\ ((stk'\ @\ [v,\ v']),\ loc',\ (length\ (compE2\ a)\ +\ length\ (compE2\ i)\ +\ pc'),\ xcp')$
by(*induct rule*: $\tau Exect\text{-}induct$)(*blast intro*: *tranclp.trancl-into-trancl AAss-*τ*execI3*)+

lemma *ALength-*τ*ExectI*:
$\tau Exec\text{-}movet\ ci\ P\ t\ e\ h\ s\ s' \Longrightarrow \tau Exec\text{-}movet\ ci\ P\ t\ (e{\cdot}length)\ h\ s\ s'$
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl ALength-*τ*execI*)+

lemma *FAcc-*τ*ExectI*:
$\tau Exec\text{-}movet\ ci\ P\ t\ e\ h\ s\ s' \Longrightarrow \tau Exec\text{-}movet\ ci\ P\ t\ (e{\cdot}F\{D\})\ h\ s\ s'$
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl FAcc-*τ*execI*)+

lemma *FAss-*τ*ExectI1*:
$\tau Exec\text{-}movet\ ci\ P\ t\ e\ h\ s\ s' \Longrightarrow \tau Exec\text{-}movet\ ci\ P\ t\ (e{\cdot}F\{D\} := e')\ h\ s\ s'$
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl FAss-*τ*execI1*)+

lemma *FAss-*τ*ExectI2*:
$\tau Exec\text{-}movet\ ci\ P\ t\ e'\ h\ (stk,\ loc,\ pc,\ xcp)\ (stk',\ loc',\ pc',\ xcp')$
$\Longrightarrow \tau Exec\text{-}movet\ ci\ P\ t\ (e{\cdot}F\{D\} := e')\ h\ ((stk\ @\ [v]),\ loc,\ (length\ (compE2\ e)\ +\ pc),\ xcp)\ ((stk'\ @\ [v]),\ loc',\ (length\ (compE2\ e)\ +\ pc'),\ xcp')$
by(*induct rule*: $\tau Exect\text{-}induct$)(*blast intro*: *tranclp.trancl-into-trancl FAss-*τ*execI2*)+

lemma *CAS-*τ*ExectI1*:
$\tau Exec\text{-}movet\ ci\ P\ t\ e\ h\ s\ s' \Longrightarrow \tau Exec\text{-}movet\ ci\ P\ t\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e''))\ h\ s\ s'$
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl CAS-*τ*execI1*)+

lemma *CAS-*τ*ExectI2*:
$\tau Exec\text{-}movet\ ci\ P\ t\ e'\ h\ (stk,\ loc,\ pc,\ xcp)\ (stk',\ loc',\ pc',\ xcp')$
$\Longrightarrow \tau Exec\text{-}movet\ ci\ P\ t\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e''))\ h\ ((stk\ @\ [v]),\ loc,\ (length\ (compE2\ e)\ +\ pc),\ xcp)\ ((stk'\ @\ [v]),\ loc',\ (length\ (compE2\ e)\ +\ pc'),\ xcp')$
by(*induct rule*: $\tau Exect\text{-}induct$)(*blast intro*: *tranclp.trancl-into-trancl CAS-*τ*execI2*)+

lemma *CAS-*τ*ExectI3*:
$\tau Exec\text{-}movet\ ci\ P\ t\ e''\ h\ (stk,\ loc,\ pc,\ xcp)\ (stk',\ loc',\ pc',\ xcp')$
$\Longrightarrow \tau Exec\text{-}movet\ ci\ P\ t\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e''))\ h\ ((stk\ @\ [v,\ v']),\ loc,\ (length\ (compE2\ e)\ +\ length\ (compE2\ e')\ +\ pc),\ xcp)\ ((stk'\ @\ [v,\ v']),\ loc',\ (length\ (compE2\ e)\ +\ length\ (compE2\ e')\ +\ pc'),\ xcp')$
by(*induct rule*: $\tau Exect\text{-}induct$)(*blast intro*: *tranclp.trancl-into-trancl CAS-*τ*execI3*)+

lemma *Call-τExectI1*:
τ*Exec-movet ci P t obj h s s'* $\Longrightarrow$ τ*Exec-movet ci P t* (*obj*·*M'*(*es*)) *h s s'*
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl Call-*τ*execI1*)+

lemma *Call-τExectI2*:
τ*Exec-movest ci P t es h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*Exec-movet ci P t* (*obj*·*M'*(*es*)) *h* ((*stk* @ [*v*]), *loc*, (*length* (*compE2 obj*) + *pc*), *xcp*) ((*stk'* @ [*v*]), *loc'*, (*length* (*compE2 obj*) + *pc'*), *xcp'*)
by(*induct rule*: τ*Execst-induct*)(*blast intro*: *tranclp.trancl-into-trancl Call-*τ*execI2*)+

lemma *Block-τExectI-Some*:
τ*Exec-movet ci P t e h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*Exec-movet ci P t* {*V*:*T*=⌊*v*⌋; *e*} *h* (*stk*, *loc*, (*Suc* (*Suc pc*)), *xcp*) (*stk'*, *loc'*, (*Suc* (*Suc pc'*)), *xcp'*)
by(*induct rule*: τ*Exect-induct*)(*blast intro*: *tranclp.trancl-into-trancl Block-*τ*execI-Some*)+

lemma *Block-τExectI-None*:
τ*Exec-movet ci P t e h s s'* $\Longrightarrow$ τ*Exec-movet ci P t* {*V*:*T*=*None*; *e*} *h s s'*
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl Block-*τ*execI-None*)+

lemma *Sync-τExectI*:
τ*Exec-movet ci P t e h s s'* $\Longrightarrow$ τ*Exec-movet ci P t* (*sync*$_V$ (*e*) *e'*) *h s s'*
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl Sync-*τ*execI*)+

lemma *Insync-τExectI*:
τ*Exec-movet ci P t e' h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*Exec-movet ci P t* (*sync*$_V$ (*e*) *e'*) *h* (*stk*, *loc*, (*Suc* (*Suc* (*Suc* (*length* (*compE2 e*) + *pc*)))), *xcp*) (*stk'*, *loc'*, (*Suc* (*Suc* (*Suc* (*length* (*compE2 e*) + *pc'*)))), *xcp'*)
by(*induct rule*: τ*Exect-induct*)(*blast intro*: *tranclp.trancl-into-trancl Insync-*τ*execI*)+

lemma *Seq-τExectI1*:
τ*Exec-movet ci P t e h s s'* $\Longrightarrow$ τ*Exec-movet ci P t* (*e*;;*e'*) *h s s'*
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl Seq-*τ*execI1*)+

lemma *Seq-τExectI2*:
τ*Exec-movet ci P t e h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'* ,*xcp'*) $\Longrightarrow$
τ*Exec-movet ci P t* (*e'*;;*e*) *h* (*stk*, *loc*, (*Suc* (*length* (*compE2 e'*) + *pc*)), *xcp*) (*stk'*, *loc'*, (*Suc* (*length* (*compE2 e'*) + *pc'*)), *xcp'*)
by(*induct rule*: τ*Exect-induct*)(*blast intro*: *tranclp.trancl-into-trancl Seq-*τ*execI2*)+

lemma *Cond-τExectI1*:
τ*Exec-movet ci P t e h s s'* $\Longrightarrow$ τ*Exec-movet ci P t* (*if* (*e*) *e1 else e2*) *h s s'*
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl Cond-*τ*execI1*)+

lemma *Cond-τExectI2*:
τ*Exec-movet ci P t e1 h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*) $\Longrightarrow$
τ*Exec-movet ci P t* (*if* (*e*) *e1 else e2*) *h* (*stk*, *loc*, (*Suc* (*length* (*compE2 e*) + *pc*)), *xcp*) (*stk'*, *loc'*, (*Suc* (*length* (*compE2 e*) + *pc'*)), *xcp'*)
by(*induct rule*: τ*Exect-induct*)(*blast intro*: *tranclp.trancl-into-trancl Cond-*τ*execI2*)+

lemma *Cond-τExectI3*:
τ*Exec-movet ci P t e2 h* (*stk*, *loc* ,*pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*) $\Longrightarrow$
τ*Exec-movet ci P t* (*if* (*e*) *e1 else e2*) *h* (*stk*, *loc*, (*Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2*

e1) + *pc*))), *xcp*) (*stk'*, *loc'*, (*Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + *pc'*))), *xcp'*)
by(*induct rule*: *τExect-induct*)(*blast intro*: *tranclp.trancl-into-trancl Cond-τexecI3*)+

lemma *While-τExectI1*:
τExec-movet ci P t c h s s' ⟹ *τExec-movet ci P t* (*while* (*c*) *e*) *h s s'*
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl While-τexecI1*)+

lemma *While-τExectI2*:
τExec-movet ci P t E h (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
⟹ *τExec-movet ci P t* (*while* (*c*) *E*) *h* (*stk*, *loc* ,(*Suc* (*length* (*compE2 c*) + *pc*)), *xcp*) (*stk'*, *loc'*, (*Suc* (*length* (*compE2 c*) + *pc'*)), *xcp'*)
by(*induct rule*: *τExect-induct*)(*blast intro*: *tranclp.trancl-into-trancl While-τexecI2*)+

lemma *Throw-τExectI*:
τExec-movet ci P t e h s s' ⟹ *τExec-movet ci P t* (*throw e*) *h s s'*
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl Throw-τexecI*)+

lemma *Try-τExectI1*:
τExec-movet ci P t E h s s' ⟹ *τExec-movet ci P t* (*try E catch*(*C' V*) *e*) *h s s'*
by(*induct rule*: *tranclp-induct*)(*blast intro*: *tranclp.trancl-into-trancl Try-τexecI1*)+

lemma *Try-τExectI2*:
τExec-movet ci P t e h (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
⟹ *τExec-movet ci P t* (*try E catch*(*C' V*) *e*) *h* (*stk*, *loc*, (*Suc* (*Suc* (*length* (*compE2 E*) + *pc*))), *xcp*) (*stk'*, *loc'*, (*Suc* (*Suc* (*length* (*compE2 E*) + *pc'*))), *xcp'*)
by(*induct rule*: *τExect-induct*)(*blast intro*: *tranclp.trancl-into-trancl Try-τexecI2*)+

lemma *τExec-movesr-map-Val*:
τExec-movesr-a P t (*map Val vs*) *h* ([], *xs*, *0*, *None*) ((*rev vs*), *xs*, (*length* (*compEs2* (*map Val vs*))), *None*)
proof(*induct vs arbitrary*: *pc stk Ts rule*: *rev-induct*)
case *Nil* **thus** *?case* **by**(*auto*)
next
case (*snoc v vs'*)
let *?E* = *compEs2* (*map Val vs'*)
from *snoc* **have** *τExec-movesr-a P t* (*map Val* (*vs'* @ [*v*])) *h* ([], *xs*, *0*, *None*) ((*rev vs'*), *xs*, (*length ?E*), *None*)
by *auto*
also **{**
have *exec-meth-a* (*compP2 P*) (*?E* @ [*Push v*]) (*compxEs2* (*map Val vs'*) *0 0* @ *shift* (*length ?E*) []) *t h* ((*rev vs'*), *xs*, (*length ?E* + *0*), *None*) *ε h* ((*v* # *rev vs'*), *xs*, (*length ?E* + *Suc 0*), *None*)
by −(*rule append-exec-meth-xt*, *auto simp add*: *exec-meth-instr*)
moreover have *τmoves2* (*compP2 P*) *h* (*rev vs'*) (*map Val vs'* @ [*Val v*]) (*length* (*compEs2* (*map Val vs'*)) + *0*) *None*
by(*rule append-τmoves2 τmoves2Hd τmove2Val*)+
ultimately have *τExec-movesr-a P t* (*map Val* (*vs'* @ [*v*])) *h* ((*rev vs'*), *xs*, (*length ?E*), *None*) ((*rev* (*vs'* @ [*v*])), *xs*, (*length* (*compEs2* (*map Val* (*vs'* @ [*v*])))), *None*)
by −(*rule τExecsr1step*, *auto simp add*: *exec-moves-def compP2-def*) **}**
finally show *?case* **.**
qed

lemma *τExec-mover-blocks1* [*simp*]:
τExec-mover ci P t (*blocks1 n Ts body*) *h s s'* = *τExec-mover ci P t body h s s'*
by(*simp add*: *τexec-move-conv-τexec-meth*)

lemma *τExec-movet-blocks1* [*simp*]:
τExec-movet ci P t (*blocks1 n Ts body*) *h s s'* = *τExec-movet ci P t body h s s'*
by(*simp add*: *τexec-move-conv-τexec-meth*)

definition *τexec-1* :: *'addr jvm-prog* ⇒ *'thread-id* ⇒ (*'addr*, *'heap*) *jvm-state* ⇒ (*'addr*, *'heap*) *jvm-state* ⇒ *bool*
where *τexec-1 P t σ σ'* ⟷ *exec-1 P t σ ε σ'* ∧ *τMove2 P σ*

lemma *τexec-1I* [*intro*]:
⟦ *exec-1 P t σ ε σ'*; *τMove2 P σ* ⟧ ⟹ *τexec-1 P t σ σ'*
by(*simp add*: *τexec-1-def*)

lemma *τexec-1E* [*elim*]:
assumes *τexec-1 P t σ σ'*
obtains *exec-1 P t σ ε σ' τMove2 P σ*
using *assms* **by**(*auto simp add*: *τexec-1-def*)

abbreviation *τExec-1r* :: *'addr jvm-prog* ⇒ *'thread-id* ⇒ (*'addr*, *'heap*) *jvm-state* ⇒ (*'addr*, *'heap*) *jvm-state* ⇒ *bool*
where *τExec-1r P t* == (*τexec-1 P t*)^**

abbreviation *τExec-1t* :: *'addr jvm-prog* ⇒ *'thread-id* ⇒ (*'addr*, *'heap*) *jvm-state* ⇒ (*'addr*, *'heap*) *jvm-state* ⇒ *bool*
where *τExec-1t P t* == (*τexec-1 P t*)^++

definition *τexec-1-d* :: *'addr jvm-prog* ⇒ *'thread-id* ⇒ (*'addr*, *'heap*) *jvm-state* ⇒ (*'addr*, *'heap*) *jvm-state* ⇒ *bool*
where *τexec-1-d P t σ σ'* ⟷ *exec-1 P t σ ε σ'* ∧ *τMove2 P σ* ∧ *check P σ*

lemma *τexec-1-dI* [*intro*]:
⟦ *exec-1 P t σ ε σ'*; *check P σ*; *τMove2 P σ* ⟧ ⟹ *τexec-1-d P t σ σ'*
by(*simp add*: *τexec-1-d-def*)

lemma *τexec-1-dE* [*elim*]:
assumes *τexec-1-d P t σ σ'*
obtains *exec-1 P t σ ε σ' check P σ τMove2 P σ*
using *assms* **by**(*auto simp add*: *τexec-1-d-def*)

abbreviation *τExec-1-dr* :: *'addr jvm-prog* ⇒ *'thread-id* ⇒ (*'addr*, *'heap*) *jvm-state* ⇒ (*'addr*, *'heap*) *jvm-state* ⇒ *bool*
where *τExec-1-dr P t* == (*τexec-1-d P t*)^**

abbreviation *τExec-1-dt* :: *'addr jvm-prog* ⇒ *'thread-id* ⇒ (*'addr*, *'heap*) *jvm-state* ⇒ (*'addr*, *'heap*) *jvm-state* ⇒ *bool*
where *τExec-1-dt P t* == (*τexec-1-d P t*)^++

declare *compxE2-size-convs*[*simp del*] *compxEs2-size-convs*[*simp del*]
declare *compxE2-stack-xlift-convs*[*simp del*] *compxEs2-stack-xlift-convs*[*simp del*]

lemma *exec-instr-frs-offer*:
(*ta*, *xcp'*, *h'*, (*stk'*, *loc'*, *C*, *M*, *pc'*) # *frs*) ∈ *exec-instr ins P t h stk loc C M pc frs*
⟹ (*ta*, *xcp'*, *h'*, (*stk'*, *loc'*, *C*, *M*, *pc'*) # *frs* @ *frs'*) ∈ *exec-instr ins P t h stk loc C M pc* (*frs* @

frs')
apply(*cases ins*)
apply(*simp-all add*: *nth-append split-beta split*: *if-split-asm sum.split-asm*)
apply(*force split*: *extCallRet.split-asm simp add*: *extRet2JVM-def*)+
done

lemma *check-instr-frs-offer*:
⟦ *check-instr ins P h stk loc C M pc frs*; *ins* ≠ *Return* ⟧
⟹ *check-instr ins P h stk loc C M pc* (*frs* @ *frs'*)
by(*cases ins*)(*simp-all split*: *if-split-asm*)

lemma *exec-instr-CM-change*:
(*ta*, *xcp'*, *h'*, (*stk'*, *loc'*, *C*, *M*, *pc'*) # *frs*) ∈ *exec-instr ins P t h stk loc C M pc frs*
⟹ (*ta*, *xcp'*, *h'*, (*stk'*, *loc'*, *C'*, *M'*, *pc'*) # *frs*) ∈ *exec-instr ins P t h stk loc C' M' pc frs*
apply(*cases ins*)
apply(*simp-all add*: *nth-append split-beta neq-Nil-conv split*: *if-split-asm sum.split-asm*)
apply(*force split*: *extCallRet.split-asm simp add*: *extRet2JVM-def*)+
done

lemma *check-instr-CM-change*:
⟦ *check-instr ins P h stk loc C M pc frs*; *ins* ≠ *Return* ⟧
⟹ *check-instr ins P h stk loc C' M' pc frs*
by(*cases ins*)(*simp-all split*: *if-split-asm*)

lemma *exec-move-exec-1*:
assumes *exec*: *exec-move ci P t body h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
and *sees*: *P* ⊢ *C sees M* : *Ts*→*T* = ⌊*body*⌋ *in D*
shows *exec-1* (*compP2 P*) *t* (*xcp*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs*) *ta* (*xcp'*, *h'*, (*stk'*, *loc'*, *C*, *M*, *pc'*) # *frs*)
using *exec* **unfolding** *exec-move-def*
proof(*cases*)
case *exec-instr*
note [*simp*] = ‹*xcp* = *None*›
and *exec* = ‹(*ta*, *xcp'*, *h'*, [(*stk'*, *loc'*, *undefined*, *undefined*, *pc'*)])
∈ *exec-instr* (*compE2 body* ! *pc*) (*compP2 P*) *t h stk loc undefined undefined pc* []›
from *exec* **have** (*ta*, *xcp'*, *h'*, [(*stk'*, *loc'*, *C*, *M*, *pc'*)])
∈ *exec-instr* (*compE2 body* ! *pc*) (*compP2 P*) *t h stk loc C M pc* []
by(*rule exec-instr-CM-change*)
from *exec-instr-frs-offer*[*OF this*, *of frs*]
have (*ta*, *xcp'*, *h'*, (*stk'*, *loc'*, *C*, *M*, *pc'*) # *frs*)
∈ *exec-instr* (*compE2 body* ! *pc*) (*compP2 P*) *t h stk loc C M pc frs* **by** *simp*
with *sees* ‹*pc* < *length* (*compE2 body*)› **show** *?thesis*
by(*simp add*: *exec-1-iff compP2-def compMb2-def nth-append*)
next
case *exec-catch*
thus *?thesis* **using** *sees-method-compP*[*OF sees*, *of* λ*C M Ts T*. *compMb2*]
by(*simp add*: *exec-1-iff compMb2-def compP2-def*)
qed

lemma τ*exec-move-*τ*exec-1*:
assumes *exec*: τ*exec-move ci P t body h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
and *sees*: *P* ⊢ *C sees M* : *Ts*→*T* = ⌊*body*⌋ *in D*
shows τ*exec-1* (*compP2 P*) *t* (*xcp*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs*) (*xcp'*, *h*, (*stk'*, *loc'*, *C*, *M*, *pc'*) # *frs*)

proof(*rule τexec-1I*)
from *exec* **obtain** *exec′*: *exec-move ci P t body h* (*stk*, *loc*, *pc*, *xcp*) *ε h* (*stk′*, *loc′*, *pc′*, *xcp′*)
and *τ*: *τmove2 P h stk body pc xcp* **by**(*rule τexec-moveE*)
have *exec-1* (*compP2 P*) *t* (*xcp*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs*) *ε* (*xcp′*, *h*, (*stk′*, *loc′*, *C*, *M*, *pc′*) # *frs*)
using *exec′ sees* **by**(*rule exec-move-exec-1*)
thus *compP2 P*,*t* ⊢ (*xcp*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs*) −*ε*−*jvm*→ (*xcp′*, *h*, (*stk′*, *loc′*, *C*, *M*, *pc′*) # *frs*) **by** *auto*
{ fix *a*
assume [*simp*]: *xcp* = ⌊*a*⌋
from *sees-method-compP*[*OF sees*, *of* λ*C M Ts T*. *compMb2*]
have *ex-table-of* (*compP2 P*) *C M* = *compxE2 body 0 0* **by**(*simp add*: *compP2-def compMb2-def*)
hence *match-ex-table* (*compP2 P*) (*cname-of h a*) *pc* (*ex-table-of* (*compP2 P*) *C M*) ≠ *None pc* < *length* (*compE2 body*)
using *exec′ sees* **by**(*auto simp add*: *exec-move-def elim*: *exec-meth.cases*) **}**
with *τ sees sees-method-compP*[*OF sees*, *of* λ*C M Ts T*. *compMb2*]
show *τMove2* (*compP2 P*) (*xcp*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs*)
unfolding *τMove2-compP2*[*OF sees*] **by**(*fastforce simp add*: *compP2-def compMb2-def*)
qed

lemma *τExec-mover-τExec-1r*:
assumes *move*: *τExec-mover ci P t body h* (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*)
and *sees*: *P* ⊢ *C sees M* : *Ts*→*T* = ⌊*body*⌋ *in D*
shows *τExec-1r* (*compP2 P*) *t* (*xcp*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs′*) (*xcp′*, *h*, (*stk′*, *loc′*, *C*, *M*, *pc′*) # *frs′*)
using *move*
by(*induct rule*: *τExecr-induct*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl τexec-move-τexec-1*[*OF* - *sees*])+

lemma *τExec-movet-τExec-1t*:
assumes *move*: *τExec-movet ci P t body h* (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*)
and *sees*: *P* ⊢ *C sees M* : *Ts*→*T* = ⌊*body*⌋ *in D*
shows *τExec-1t* (*compP2 P*) *t* (*xcp*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs′*) (*xcp′*, *h*, (*stk′*, *loc′*, *C*, *M*, *pc′*) # *frs′*)
using *move*
by(*induct rule*: *τExect-induct*)(*blast intro*: *tranclp.trancl-into-trancl τexec-move-τexec-1*[*OF* - *sees*])+

lemma *τExec-1r-rtranclpD*:
τExec-1r P t (*xcp*, *h*, *frs*) (*xcp′*, *h′*, *frs′*)
⟹ (λ((*xcp*, *frs*), *h*) ((*xcp′*, *frs′*), *h′*). *exec-1 P t* (*xcp*, *h*, *frs*) *ε* (*xcp′*, *h′*, *frs′*) ∧ *τMove2 P* (*xcp*, *h*, *frs*))^** ((*xcp*, *frs*), *h*) ((*xcp′*, *frs′*), *h′*)
by(*induct rule*: *rtranclp-induct3*)(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl*)+

lemma *τExec-1t-rtranclpD*:
τExec-1t P t (*xcp*, *h*, *frs*) (*xcp′*, *h′*, *frs′*)
⟹ (λ((*xcp*, *frs*), *h*) ((*xcp′*, *frs′*), *h′*). *exec-1 P t* (*xcp*, *h*, *frs*) *ε* (*xcp′*, *h′*, *frs′*) ∧ *τMove2 P* (*xcp*, *h*, *frs*))^++ ((*xcp*, *frs*), *h*) ((*xcp′*, *frs′*), *h′*)
by(*induct rule*: *tranclp-induct3*)(*fastforce intro*: *tranclp.trancl-into-trancl*)+

lemma *exec-meth-length-compE2-stack-xliftD*:
exec-meth ci P (*compE2 e*) (*stack-xlift d* (*compxE2 e 0 0*)) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′ s′*
⟹ *pc* < *length* (*compE2 e*)
by(*cases s′*)(*auto simp add*: *stack-xlift-compxE2*)

lemma *exec-meth-length-pc-xt-Nil*:

exec-meth ci P ins [] *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′ s′* ⟹ *pc* < *length ins*
apply(*erule exec-meth.cases*)
apply(*auto dest*: *match-ex-table-pc-length-compE2*)
done

lemma *BinOp-exec2D*:
assumes *exec*: *exec-meth ci* (*compP2 P*) (*compE2* (*e1* «*bop*» *e2*)) (*compxE2* (*e1* «*bop*» *e2*) *0 0*) *t h* (*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
and *pc*: *pc* < *length* (*compE2 e2*)
shows *exec-meth ci* (*compP2 P*) (*compE2 e2*) (*stack-xlift* (*length* [*v1*]) (*compxE2 e2 0 0*)) *t h* (*stk* @ [*v1*], *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′* − *length* (*compE2 e1*), *xcp′*) ∧ *pc′* ≥ *length* (*compE2 e1*)
proof
from *exec* **have** *exec-meth ci* (*compP2 P*) ((*compE2 e1* @ *compE2 e2*) @ [*BinOpInstr bop*])
(*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length* [*v1*]) (*compxE2 e2 0 0*))) *t h*
(*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *compxE2-size-convs compxE2-stack-xlift-convs*)
hence *exec′*: *exec-meth ci* (*compP2 P*) (*compE2 e1* @ *compE2 e2*) (*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length* [*v1*]) (*compxE2 e2 0 0*))) *t h*
(*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*rule exec-meth-take*) (*simp add*: *pc*)
thus *exec-meth ci* (*compP2 P*) (*compE2 e2*) (*stack-xlift* (*length* [*v1*]) (*compxE2 e2 0 0*)) *t h*
(*stk* @ [*v1*], *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′* − *length* (*compE2 e1*), *xcp′*)
by(*rule exec-meth-drop-xt*) *auto*
from *exec′* **show** *pc′* ≥ *length* (*compE2 e1*)
by(*rule exec-meth-drop-xt-pc*)(*auto*)
qed

lemma *Call-execParamD*:
assumes *exec*: *exec-meth ci* (*compP2 P*) (*compE2* (*obj*·*M′*(*ps*))) (*compxE2* (*obj*·*M′*(*ps*)) *0 0*) *t h* (*stk* @ [*v*], *loc*, *length* (*compE2 obj*) + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
and *pc*: *pc* < *length* (*compEs2 ps*)
shows *exec-meth ci* (*compP2 P*) (*compEs2 ps*) (*stack-xlift* (*length* [*v*]) (*compxEs2 ps 0 0*)) *t h* (*stk* @ [*v*], *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′* − *length* (*compE2 obj*), *xcp′*) ∧ *pc′* ≥ *length* (*compE2 obj*)
proof
from *exec* **have** *exec-meth ci* (*compP2 P*) ((*compE2 obj* @ *compEs2 ps*) @ [*Invoke M′* (*length ps*)])
(*compxE2 obj 0 0* @ *shift* (*length* (*compE2 obj*)) (*stack-xlift* (*length* [*v*]) (*compxEs2 ps 0 0*))) *t h*
(*stk* @ [*v*], *loc*, *length* (*compE2 obj*) + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *compxEs2-size-convs compxEs2-stack-xlift-convs*)
hence *exec′*: *exec-meth ci* (*compP2 P*) (*compE2 obj* @ *compEs2 ps*) (*compxE2 obj 0 0* @ *shift* (*length* (*compE2 obj*)) (*stack-xlift* (*length* [*v*]) (*compxEs2 ps 0 0*))) *t h*
(*stk* @ [*v*], *loc*, *length* (*compE2 obj*) + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*rule exec-meth-take*)(*simp add*: *pc*)
thus *exec-meth ci* (*compP2 P*) (*compEs2 ps*) (*stack-xlift* (*length* [*v*]) (*compxEs2 ps 0 0*)) *t h* (*stk* @ [*v*], *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′* − *length* (*compE2 obj*), *xcp′*)
by(*rule exec-meth-drop-xt*) *auto*
from *exec′* **show** *pc′* ≥ *length* (*compE2 obj*)
by(*rule exec-meth-drop-xt-pc*)(*auto*)
qed

lemma *exec-move-length-compE2D* [*dest*]:
exec-move ci P t e h (*stk*, *loc*, *pc*, *xcp*) *ta h′ s′* ⟹ *pc* < *length* (*compE2 e*)
by(*cases s′*)(*auto simp add*: *exec-move-def*)

lemma *exec-moves-length-compEs2D* [*dest*]:

exec-moves ci P t es h (stk, loc, pc, xcp) ta h′ s′ ⟹ pc < length (compEs2 es)
by(*cases s′*)(*auto simp add*: *exec-moves-def*)

lemma *exec-meth-ci-appD*:
⟦ exec-meth ci P ins xt t h (stk, loc, pc, None) ta h′ fr′ ⟧
⟹ ci-app ci (ins ! pc) P h stk loc undefined undefined pc []
by(*cases fr′*)(*simp add*: *exec-meth-instr*)

lemma *exec-move-ci-appD*:
exec-move ci P t E h (stk, loc, pc, None) ta h′ fr′
⟹ ci-app ci (compE2 E ! pc) (compP2 P) h stk loc undefined undefined pc []
unfolding *exec-move-def* **by**(*rule exec-meth-ci-appD*)

lemma *exec-moves-ci-appD*:
exec-moves ci P t Es h (stk, loc, pc, None) ta h′ fr′
⟹ ci-app ci (compEs2 Es ! pc) (compP2 P) h stk loc undefined undefined pc []
unfolding *exec-moves-def* **by**(*rule exec-meth-ci-appD*)

lemma *τinstr-stk-append-check*:
check-instr′ i P h stk loc C M pc frs ⟹ τinstr P h (stk @ vs) i = τinstr P h stk i
by(*cases i*)(*simp-all add*: *nth-append*)

lemma *τinstr-stk-drop-exec-move*:
exec-move ci P t e h (stk, loc, pc, None) ta h′ fr′
⟹ τinstr (compP2 P) h (stk @ vs) (compE2 e ! pc) = τinstr (compP2 P) h stk (compE2 e ! pc)
apply(*drule exec-move-ci-appD*)
apply(*drule wf-ciD2-ci-app*)
apply(*erule τinstr-stk-append-check*)
done

lemma *τinstr-stk-drop-exec-moves*:
exec-moves ci P t es h (stk, loc, pc, None) ta h′ fr′
⟹ τinstr (compP2 P) h (stk @ vs) (compEs2 es ! pc) = τinstr (compP2 P) h stk (compEs2 es ! pc)
apply(*drule exec-moves-ci-appD*)
apply(*drule wf-ciD2-ci-app*)
apply(*erule τinstr-stk-append-check*)
done

end

end

7.16 The delay bisimulation between intermediate language and JVM

theory *J1JVMBisim* **imports**
Execs
../BV/BVNoTypeError
J1
begin

declare *Listn.lesub-list-impl-same-size*[*simp del*]

lemma (**in** *JVM-heap-conf-base'*) *τexec-1-τexec-1-d*:
  ⟦ *wf-jvm-prog*$_\Phi$ *P*; *τexec-1 P t σ σ'*; Φ |− *t*:*σ* [*ok*] ⟧ ⟹ *τexec-1-d P t σ σ'*
by(*auto simp add*: *τexec-1-def τexec-1-d-def welltyped-commute*[*symmetric*] *elim*: *jvmd-NormalE*)

context *JVM-conf-read* **begin**

lemma *τExec-1r-preserves-correct-state*:
  assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
  and *exec*: *τExec-1r P t σ σ'*
  shows Φ |− *t*:*σ* [*ok*] ⟹ Φ |− *t*:*σ'* [*ok*]
using *exec*
by(*induct*)(*blast intro*: *BV-correct-1*[*OF wf*])+

lemma *τExec-1t-preserves-correct-state*:
  assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
  and *exec*: *τExec-1t P t σ σ'*
  shows Φ |− *t*:*σ* [*ok*] ⟹ Φ |− *t*:*σ'* [*ok*]
using *exec*
by(*induct*)(*blast intro*: *BV-correct-1*[*OF wf*])+

lemma *τExec-1r-τExec-1-dr*:
  assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
  shows ⟦ *τExec-1r P t σ σ'*; Φ |− *t*:*σ* [*ok*] ⟧ ⟹ *τExec-1-dr P t σ σ'*
apply(*induct rule*: *rtranclp-induct*)
apply(*blast intro*: *rtranclp.rtrancl-into-rtrancl τexec-1-τexec-1-d*[*OF wf*] *τExec-1r-preserves-correct-state*[*OF wf*])+
done

lemma *τExec-1t-τExec-1-dt*:
  assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
  shows ⟦ *τExec-1t P t σ σ'*; Φ |− *t*:*σ* [*ok*] ⟧ ⟹ *τExec-1-dt P t σ σ'*
apply(*induct rule*: *tranclp-induct*)
apply(*blast intro*: *tranclp.trancl-into-trancl τexec-1-τexec-1-d*[*OF wf*] *τExec-1t-preserves-correct-state*[*OF wf*])+
done

lemma *τExec-1-dr-preserves-correct-state*:
  assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
  and *exec*: *τExec-1-dr P t σ σ'*
  shows Φ |− *t*: *σ* [*ok*] ⟹ Φ |− *t*: *σ'* [*ok*]
using *exec*
by(*induct*)(*blast intro*: *BV-correct-1*[*OF wf*])+

lemma *τExec-1-dt-preserves-correct-state*:
  assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
  and *exec*: *τExec-1-dt P t σ σ'*
  shows Φ |− *t*:*σ* [*ok*] ⟹ Φ |− *t*:*σ'* [*ok*]
using *exec*
by(*induct*)(*blast intro*: *BV-correct-1*[*OF wf*])+

end

locale *J1-JVM-heap-base* =

J1-heap-base +
JVM-heap-base +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
begin

inductive *bisim1* ::
'm prog ⇒ *'heap* ⇒ *'addr expr1* ⇒ (*'addr expr1* × *'addr locals1*)
⇒ (*'addr val list* × *'addr val list* × *pc* × *'addr option*) ⇒ *bool*

and *bisims1* ::
'm prog ⇒ *'heap* ⇒ *'addr expr1 list* ⇒ (*'addr expr1 list* × *'addr locals1*)
⇒ (*'addr val list* × *'addr val list* × *pc* × *'addr option*) ⇒ *bool*

and *bisim1-syntax* ::
'm prog ⇒ *'addr expr1* ⇒ *'heap* ⇒ (*'addr expr1* × *'addr locals1*)
⇒ (*'addr val list* × *'addr val list* × *pc* × *'addr option*) ⇒ *bool*
(‹-,-,- ⊢ - ↔ -› [*50*, *0*, *0*, *0*, *50*] *100*)

and *bisims1-syntax* ::
'm prog ⇒ *'addr expr1 list* ⇒ *'heap* ⇒ (*'addr expr1 list* × *'addr locals1*)
⇒ (*'addr val list* × *'addr val list* × *pc* × *'addr option*) ⇒ *bool*
(‹-,-,- ⊢ - [↔] -› [*50*, *0*, *0*, *0*, *50*] *100*)
for *P* :: *'m prog* **and** *h* :: *'heap*
where
P, *e*, *h* ⊢ *exs* ↔ *s* ≡ *bisim1 P h e exs s*
| *P*, *es*, *h* ⊢ *esxs* [↔] *s* ≡ *bisims1 P h es esxs s*

| *bisim1Val2*:
pc = *length* (*compE2 e*) ⟹ *P*, *e*, *h* ⊢ (*Val v*, *xs*) ↔ (*v* # [], *xs*, *pc*, *None*)

| *bisim1New*:
P, *new C*, *h* ⊢ (*new C*, *xs*) ↔ ([], *xs*, *0*, *None*)

| *bisim1NewThrow*:
P, *new C*, *h* ⊢ (*THROW OutOfMemory*, *xs*) ↔ ([], *xs*, *0*, ⌊*addr-of-sys-xcpt OutOfMemory*⌋)

| *bisim1NewArray*:
P, *e*, *h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*) ⟹ *P*, *newA T*⌊*e*⌉, *h* ⊢ (*newA T*⌊*e'*⌉, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1NewArrayThrow*:
P, *e*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋) ⟹ *P*, *newA T*⌊*e*⌉, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)

| *bisim1NewArrayFail*:
P, *newA T*⌊*e*⌉, *h* ⊢ (*Throw a*, *xs*) ↔ ([*v*], *xs*, *length* (*compE2 e*), ⌊*a*⌋)

| *bisim1Cast*:
$P,\ e,\ h \vdash (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp) \Longrightarrow P,\ Cast\ T\ e,\ h \vdash (Cast\ T\ e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1CastThrow*:
$P,\ e,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor) \Longrightarrow P,\ Cast\ T\ e,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$

| *bisim1CastFail*:
$P,\ Cast\ T\ e,\ h \vdash (THROW\ ClassCast,\ xs) \leftrightarrow ([v],\ xs,\ length\ (compE2\ e),\ \lfloor addr\text{-}of\text{-}sys\text{-}xcpt\ Class\text{-}Cast \rfloor)$

| *bisim1InstanceOf*:
$P,\ e,\ h \vdash (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp) \Longrightarrow P,\ e\ instanceof\ T,\ h \vdash (e'\ instanceof\ T,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1InstanceOfThrow*:
$P,\ e,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor) \Longrightarrow P,\ e\ instanceof\ T,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$

| *bisim1Val*: $P,\ Val\ v,\ h \vdash (Val\ v,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)$

| *bisim1Var*: $P,\ Var\ V,\ h \vdash (Var\ V,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)$

| *bisim1BinOp1*:
$P,\ e1,\ h \vdash (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp) \Longrightarrow P,\ e1\ «bop»\ e2,\ h \vdash (e'«bop»\ e2,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1BinOp2*:
$P,\ e2,\ h \vdash (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ e1\ «bop»\ e2,\ h \vdash (Val\ v1\ «bop»\ e',\ xs) \leftrightarrow (stk\ @\ [v1],\ loc,\ length\ (compE2\ e1) + pc,\ xcp)$

| *bisim1BinOpThrow1*:
$P,\ e1,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$
$\Longrightarrow P,\ e1\ «bop»\ e2,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$

| *bisim1BinOpThrow2*:
$P,\ e2,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$
$\Longrightarrow P,\ e1\ «bop»\ e2,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk\ @\ [v1],\ loc,\ length\ (compE2\ e1) + pc,\ \lfloor a \rfloor)$

| *bisim1BinOpThrow*:
$P,\ e1\ «bop»\ e2,\ h \vdash (Throw\ a,\ xs) \leftrightarrow ([v1,\ v2],\ xs,\ length\ (compE2\ e1) + length\ (compE2\ e2),\ \lfloor a \rfloor)$

| *bisim1LAss1*:
$P,\ e,\ h \vdash (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp) \Longrightarrow P,\ V{:=}e,\ h \vdash (V{:=}e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1LAss2*:
$P,\ V{:=}e,\ h \vdash (unit,\ xs) \leftrightarrow ([],\ xs,\ Suc\ (length\ (compE2\ e)),\ None)$

| *bisim1LAssThrow*:
$P,\ e,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor) \Longrightarrow P,\ V{:=}e,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$

| *bisim1AAcc1*:
$P,\ a,\ h \vdash (a',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ a\lfloor i\rceil,\ h \vdash (a'\lfloor i\rceil,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1AAcc2*:
$P,\ i,\ h \vdash (i',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ a\lfloor i\rceil,\ h \vdash (Val\ v\lfloor i'\rceil,\ xs) \leftrightarrow (stk\ @\ [v],\ loc,\ length\ (compE2\ a) + pc,\ xcp)$

| *bisim1AAccThrow1*:
$P,\ a,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor ad\rfloor)$
$\Longrightarrow P,\ a\lfloor i\rceil,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor ad\rfloor)$

| *bisim1AAccThrow2*:
$P,\ i,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor ad\rfloor)$
$\Longrightarrow P,\ a\lfloor i\rceil,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk\ @\ [v],\ loc,\ length\ (compE2\ a) + pc,\ \lfloor ad\rfloor)$

| *bisim1AAccFail*:
$P,\ a\lfloor i\rceil,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow ([v,\ v'],\ xs,\ length\ (compE2\ a) + length\ (compE2\ i),\ \lfloor ad\rfloor)$

| *bisim1AAss1*:
$P,\ a,\ h \vdash (a',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ a\lfloor i\rceil := e,\ h \vdash (a'\lfloor i\rceil := e,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1AAss2*:
$P,\ i,\ h \vdash (i',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ a\lfloor i\rceil := e,\ h \vdash (Val\ v\lfloor i'\rceil := e,\ xs) \leftrightarrow (stk\ @\ [v],\ loc,\ length\ (compE2\ a) + pc,\ xcp)$

| *bisim1AAss3*:
$P,\ e,\ h \vdash (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ a\lfloor i\rceil := e,\ h \vdash (Val\ v\lfloor Val\ v'\rceil := e',\ xs) \leftrightarrow (stk\ @\ [v',\ v],\ loc,\ length\ (compE2\ a) + length\ (compE2\ i) + pc,\ xcp)$

| *bisim1AAssThrow1*:
$P,\ a,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor ad\rfloor)$
$\Longrightarrow P,\ a\lfloor i\rceil := e,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor ad\rfloor)$

| *bisim1AAssThrow2*:
$P,\ i,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor ad\rfloor)$
$\Longrightarrow P,\ a\lfloor i\rceil := e,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk\ @\ [v],\ loc,\ length\ (compE2\ a) + pc,\ \lfloor ad\rfloor)$

| *bisim1AAssThrow3*:
$P,\ e,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor ad\rfloor)$
$\Longrightarrow P,\ a\lfloor i\rceil := e,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk\ @\ [v',\ v],\ loc,\ length\ (compE2\ a) + length\ (compE2\ i) + pc,\ \lfloor ad\rfloor)$

| *bisim1AAssFail*:
$P,\ a\lfloor i\rceil := e,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow ([v',\ v,\ v''],\ xs,\ length\ (compE2\ a) + length\ (compE2\ i) + length\ (compE2\ e),\ \lfloor ad\rfloor)$

| *bisim1AAss4*:
$P,\ a\lfloor i\rceil := e,\ h \vdash (unit,\ xs) \leftrightarrow ([],\ xs,\ Suc\ (length\ (compE2\ a) + length\ (compE2\ i) + length\ (compE2\ e)),\ None)$

| *bisim1ALength*:
$P,\ a,\ h \vdash (a',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ a{\cdot}length,\ h \vdash (a'{\cdot}length,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1ALengthThrow*:
$P,\ a,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor ad \rfloor)$
$\Longrightarrow P,\ a{\cdot}length,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor ad \rfloor)$

| *bisim1ALengthNull*:
$P,\ a{\cdot}length,\ h \vdash (THROW\ NullPointer,\ xs) \leftrightarrow ([Null],\ xs,\ length\ (compE2\ a),\ \lfloor addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer \rfloor)$

| *bisim1FAcc*:
$P,\ e,\ h \vdash (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ e{\cdot}F\{D\},\ h \vdash (e'{\cdot}F\{D\},\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1FAccThrow*:
$P,\ e,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor ad \rfloor)$
$\Longrightarrow P,\ e{\cdot}F\{D\},\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor ad \rfloor)$

| *bisim1FAccNull*:
$P,\ e{\cdot}F\{D\},\ h \vdash (THROW\ NullPointer,\ xs) \leftrightarrow ([Null],\ xs,\ length\ (compE2\ e),\ \lfloor addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer \rfloor)$

| *bisim1FAss1*:
$P,\ e,\ h \vdash (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ e{\cdot}F\{D\} := e2,\ h \vdash (e'{\cdot}F\{D\} := e2,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1FAss2*:
$P,\ e2,\ h \vdash (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ e{\cdot}F\{D\} := e2,\ h \vdash (Val\ v{\cdot}F\{D\} := e',\ xs) \leftrightarrow (stk\ @\ [v],\ loc,\ length\ (compE2\ e) + pc,\ xcp)$

| *bisim1FAssThrow1*:
$P,\ e,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor ad \rfloor)$
$\Longrightarrow P,\ e{\cdot}F\{D\} := e2,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor ad \rfloor)$

| *bisim1FAssThrow2*:
$P,\ e2,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor ad \rfloor)$
$\Longrightarrow P,\ e{\cdot}F\{D\} := e2,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk\ @\ [v],\ loc,\ length\ (compE2\ e) + pc,\ \lfloor ad \rfloor)$

| *bisim1FAssNull*:
$P,\ e{\cdot}F\{D\} := e2,\ h \vdash (THROW\ NullPointer,\ xs) \leftrightarrow ([v,\ Null],\ xs,\ length\ (compE2\ e) + length\ (compE2\ e2),\ \lfloor addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer \rfloor)$

| *bisim1FAss3*:
$P,\ e{\cdot}F\{D\} := e2,\ h \vdash (unit,\ xs) \leftrightarrow ([],\ xs,\ Suc\ (length\ (compE2\ e) + length\ (compE2\ e2)),\ None)$

| *bisim1CAS1*:

P, $e1$, $h \vdash (e1', xs) \leftrightarrow (stk, loc, pc, xcp)$
$\Longrightarrow P$, $e1{\cdot}compareAndSwap(D{\cdot}F, e2, e3)$, $h \vdash (e1'{\cdot}compareAndSwap(D{\cdot}F, e2, e3), xs) \leftrightarrow (stk, loc, pc, xcp)$

| *bisim1CAS2*:
P, $e2$, $h \vdash (e2', xs) \leftrightarrow (stk, loc, pc, xcp)$
$\Longrightarrow P$, $e1{\cdot}compareAndSwap(D{\cdot}F, e2, e3)$, $h \vdash (Val\ v{\cdot}compareAndSwap(D{\cdot}F, e2', e3), xs) \leftrightarrow (stk @ [v], loc, length\ (compE2\ e1) + pc, xcp)$

| *bisim1CAS3*:
P, $e3$, $h \vdash (e3', xs) \leftrightarrow (stk, loc, pc, xcp)$
$\Longrightarrow P$, $e1{\cdot}compareAndSwap(D{\cdot}F, e2, e3)$, $h \vdash (Val\ v{\cdot}compareAndSwap(D{\cdot}F, Val\ v', e3'), xs) \leftrightarrow (stk @ [v', v], loc, length\ (compE2\ e1) + length\ (compE2\ e2) + pc, xcp)$

| *bisim1CASThrow1*:
P, $e1$, $h \vdash (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor)$
$\Longrightarrow P$, $e1{\cdot}compareAndSwap(D{\cdot}F, e2, e3)$, $h \vdash (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor)$

| *bisim1CASThrow2*:
P, $e2$, $h \vdash (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor)$
$\Longrightarrow P$, $e1{\cdot}compareAndSwap(D{\cdot}F, e2, e3)$, $h \vdash (Throw\ ad, xs) \leftrightarrow (stk @ [v], loc, length\ (compE2\ e1) + pc, \lfloor ad \rfloor)$

| *bisim1CASThrow3*:
P, $e3$, $h \vdash (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor)$
$\Longrightarrow P$, $e1{\cdot}compareAndSwap(D{\cdot}F, e2, e3)$, $h \vdash (Throw\ ad, xs) \leftrightarrow (stk @ [v', v], loc, length\ (compE2\ e1) + length\ (compE2\ e2) + pc, \lfloor ad \rfloor)$

| *bisim1CASFail*:
P, $e1{\cdot}compareAndSwap(D{\cdot}F, e2, e3)$, $h \vdash (Throw\ ad, xs) \leftrightarrow ([v', v, v''], xs, length\ (compE2\ e1) + length\ (compE2\ e2) + length\ (compE2\ e3), \lfloor ad \rfloor)$

| *bisim1Call1*:
P, obj, $h \vdash (obj', xs) \leftrightarrow (stk, loc, pc, xcp)$
$\Longrightarrow P$, $obj{\cdot}M(ps)$, $h \vdash (obj'{\cdot}M(ps), xs) \leftrightarrow (stk, loc, pc, xcp)$

| *bisim1CallParams*:
P, ps, $h \vdash (ps', xs)\ [\leftrightarrow]\ (stk, loc, pc, xcp)$
$\Longrightarrow P$, $obj{\cdot}M(ps)$, $h \vdash (Val\ v{\cdot}M(ps'), xs) \leftrightarrow (stk @ [v], loc, length\ (compE2\ obj) +\ \ pc, xcp)$

| *bisim1CallThrowObj*:
P, obj, $h \vdash (Throw\ a, xs) \leftrightarrow (stk, loc, pc, \lfloor a \rfloor)$
$\Longrightarrow P$, $obj{\cdot}M(ps)$, $h \vdash (Throw\ a, xs) \leftrightarrow (stk, loc, pc, \lfloor a \rfloor)$

| *bisim1CallThrowParams*:
P, ps, $h \vdash (map\ Val\ vs @ Throw\ a \#\ ps', xs)\ [\leftrightarrow]\ (stk, loc, pc, \lfloor a \rfloor)$
$\Longrightarrow P$, $obj{\cdot}M(ps)$, $h \vdash (Throw\ a, xs) \leftrightarrow (stk @ [v], loc, length\ (compE2\ obj) + pc, \lfloor a \rfloor)$

| *bisim1CallThrow*:
$length\ ps = length\ vs$
$\Longrightarrow P$, $obj{\cdot}M(ps)$, $h \vdash (Throw\ a, xs) \leftrightarrow (vs @ [v], xs, length\ (compE2\ obj) + length\ (compEs2\ ps), \lfloor a \rfloor)$

| *bisim1BlockSome1*:
$P, \{V{:}T{=}\lfloor v\rfloor;\ e\}, h \vdash (\{V{:}T{=}\lfloor v\rfloor;\ e\}, xs) \leftrightarrow ([], xs, 0, None)$

| *bisim1BlockSome2*:
$P, \{V{:}T{=}\lfloor v\rfloor;\ e\}, h \vdash (\{V{:}T{=}\lfloor v\rfloor;\ e\}, xs) \leftrightarrow ([v], xs, Suc\ 0, None)$

| *bisim1BlockSome4*:
$P, e, h \vdash (e', xs) \leftrightarrow (stk, loc, pc, xcp)$
$\Longrightarrow P, \{V{:}T{=}\lfloor v\rfloor;\ e\}, h \vdash (\{V{:}T{=}None;\ e'\}, xs) \leftrightarrow (stk, loc, Suc\ (Suc\ pc), xcp)$

| *bisim1BlockThrowSome*:
$P, e, h \vdash (Throw\ a, xs) \leftrightarrow (stk, loc, pc, \lfloor a\rfloor)$
$\Longrightarrow P, \{V{:}T{=}\lfloor v\rfloor;\ e\}, h \vdash (Throw\ a, xs) \leftrightarrow (stk, loc, Suc\ (Suc\ pc), \lfloor a\rfloor)$

| *bisim1BlockNone*:
$P, e, h \vdash (e', xs) \leftrightarrow (stk, loc, pc, xcp)$
$\Longrightarrow P, \{V{:}T{=}None;\ e\}, h \vdash (\{V{:}T{=}None;\ e'\}, xs) \leftrightarrow (stk, loc, pc, xcp)$

| *bisim1BlockThrowNone*:
$P, e, h \vdash (Throw\ a, xs) \leftrightarrow (stk, loc, pc, \lfloor a\rfloor)$
$\Longrightarrow P, \{V{:}T{=}None;\ e\}, h \vdash (Throw\ a, xs) \leftrightarrow (stk, loc, pc, \lfloor a\rfloor)$

| *bisim1Sync1*:
$P, e1, h \vdash (e', xs) \leftrightarrow (stk, loc, pc, xcp)$
$\Longrightarrow P, sync_V\ (e1)\ e2, h \vdash (sync_V\ (e')\ e2, xs) \leftrightarrow (stk, loc, pc, xcp)$

| *bisim1Sync2*:
$P, sync_V\ (e1)\ e2, h \vdash (sync_V\ (Val\ v)\ e2, xs) \leftrightarrow ([v, v], xs, Suc\ (length\ (compE2\ e1)), None)$

| *bisim1Sync3*:
$P, sync_V\ (e1)\ e2, h \vdash (sync_V\ (Val\ v)\ e2, xs) \leftrightarrow ([v], xs[V := v], Suc\ (Suc\ (length\ (compE2\ e1))), None)$

| *bisim1Sync4*:
$P, e2, h \vdash (e', xs) \leftrightarrow (stk, loc, pc, xcp)$
$\Longrightarrow P, sync_V\ (e1)\ e2, h \vdash (insync_V\ (a)\ e', xs) \leftrightarrow (stk, loc, Suc\ (Suc\ (Suc\ (length\ (compE2\ e1) + pc))), xcp)$

| *bisim1Sync5*:
$P, sync_V\ (e1)\ e2, h \vdash (insync_V\ (a)\ Val\ v, xs) \leftrightarrow ([xs\ !\ V, v], xs, 4 + length\ (compE2\ e1) + length\ (compE2\ e2), None)$

| *bisim1Sync6*:
$P, sync_V\ (e1)\ e2, h \vdash (Val\ v, xs) \leftrightarrow ([v], xs, 5 + length\ (compE2\ e1) + length\ (compE2\ e2), None)$

| *bisim1Sync7*:
$P, sync_V\ (e1)\ e2, h \vdash (insync_V\ (a)\ Throw\ a', xs) \leftrightarrow ([Addr\ a'], xs, 6 + length\ (compE2\ e1) + length\ (compE2\ e2), None)$

| *bisim1Sync8*:
$P, sync_V\ (e1)\ e2, h \vdash (insync_V\ (a)\ Throw\ a', xs) \leftrightarrow$
$([xs\ !\ V, Addr\ a'], xs, 7 + length\ (compE2\ e1) + length\ (compE2\ e2), None)$

| *bisim1Sync9*:
P, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *xs*, *8* + *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)

| *bisim1Sync10*:
P, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *xs*, *8* + *length* (*compE2 e1*) + *length* (*compE2 e2*), ⌊*a*⌋)

| *bisim1Sync11*:
P, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*THROW NullPointer*, *xs*) ↔ ([*Null*], *xs*, *Suc* (*Suc* (*length* (*compE2 e1*))), ⌊*addr-of-sys-xcpt NullPointer*⌋)

| *bisim1Sync12*:
P, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*v*, *v'*], *xs*, *4* + *length* (*compE2 e1*) + *length* (*compE2 e2*), ⌊*a*⌋)

| *bisim1Sync14*:
P, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*Throw a*, *xs*) ↔
([*v*, *Addr a'*], *xs*, *7* + *length* (*compE2 e1*) + *length* (*compE2 e2*), ⌊*a*⌋)

| *bisim1SyncThrow*:
P, *e1*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
⟹ *P*, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)

| *bisim1InSync*: — This rule only exists such that *P*,*e*,*h* ⊢ (*e*, *xs*) ↔ ([], *xs*, *0*, *None*) holds for all *e*
P, *insync*$_V$ (*a*) *e*, *h* ⊢ (*insync*$_V$ (*a*) *e*, *xs*) ↔ ([], *xs*, *0*, *None*)

| *bisim1Seq1*:
P, *e1*, *h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*) ⟹ *P*, *e1*;;*e2*, *h* ⊢ (*e'*;;*e2*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1SeqThrow1*:
P, *e1*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋) ⟹ *P*, *e1*;;*e2*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)

| *bisim1Seq2*:
P, *e2*, *h* ⊢ *exs* ↔ (*stk*, *loc*, *pc*, *xcp*)
⟹ *P*, *e1*;;*e2*, *h* ⊢ *exs* ↔ (*stk*, *loc*, *Suc* (*length* (*compE2 e1*) + *pc*), *xcp*)

| *bisim1Cond1*:
P, *e*, *h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
⟹ *P*, *if* (*e*) *e1 else e2*, *h* ⊢ (*if* (*e'*) *e1 else e2*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1CondThen*:
P, *e1*, *h* ⊢ *exs* ↔ (*stk*, *loc*, *pc*, *xcp*)
⟹ *P*, *if* (*e*) *e1 else e2*, *h* ⊢ *exs* ↔ (*stk*, *loc*, *Suc* (*length* (*compE2 e*) + *pc*), *xcp*)

| *bisim1CondElse*:
P, *e2*, *h* ⊢ *exs* ↔ (*stk*, *loc*, *pc*, *xcp*)
⟹ *P*, *if* (*e*) *e1 else e2*, *h* ⊢ *exs* ↔ (*stk*, *loc*, *Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + *pc*)), *xcp*)

| *bisim1CondThrow*:

$P,\ e,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$
$\Longrightarrow P,\ if\ (e)\ e1\ else\ e2,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$

| *bisim1While1*:
$P,\ while\ (c)\ e,\ h \vdash (while\ (c)\ e,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)$

| *bisim1While3*:
$P,\ c,\ h \vdash (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ while\ (c)\ e,\ h \vdash (if\ (e')\ (e;;\ while\ (c)\ e)\ else\ unit,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1While4*:
$P,\ e,\ h \vdash (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ while\ (c)\ e,\ h\ \vdash (e';;\ while\ (c)\ e,\ xs) \leftrightarrow (stk,\ loc,\ Suc\ (length\ (compE2\ c) + pc),\ xcp)$

| *bisim1While6*:
$P,\ while\ (c)\ e,\ h \vdash (while\ (c)\ e,\ xs) \leftrightarrow ([],\ xs,\ Suc\ (Suc\ (length\ (compE2\ c) + length\ (compE2\ e))),$
$None)$

| *bisim1While7*:
$P,\ while\ (c)\ e,\ h \vdash (unit,\ xs) \leftrightarrow ([],\ xs,\ Suc\ (Suc\ (Suc\ (length\ (compE2\ c) + length\ (compE2\ e)))),$
$None)$

| *bisim1WhileThrow1*:
$P,\ c,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$
$\Longrightarrow P,\ while\ (c)\ e,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$

| *bisim1WhileThrow2*:
$P,\ e,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$
$\Longrightarrow P,\ while\ (c)\ e,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ Suc\ (length\ (compE2\ c) + pc),\ \lfloor a \rfloor)$

| *bisim1Throw1*:
$P,\ e,\ h \vdash (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp) \Longrightarrow P,\ throw\ e,\ h \vdash (throw\ e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1Throw2*:
$P,\ throw\ e,\ h \vdash (Throw\ a,\ xs) \leftrightarrow ([Addr\ a],\ xs,\ length\ (compE2\ e),\ \lfloor a \rfloor)$

| *bisim1ThrowNull*:
$P,\ throw\ e,\ h \vdash (THROW\ NullPointer,\ xs) \leftrightarrow ([Null],\ xs,\ length\ (compE2\ e),\ \lfloor addr\text{-}of\text{-}sys\text{-}xcpt$
$NullPointer \rfloor)$

| *bisim1ThrowThrow*:
$P,\ e,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor) \Longrightarrow P,\ throw\ e,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$

| *bisim1Try*:
$P,\ e,\ h \vdash (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ try\ e\ catch(C\ V)\ e2,\ h \vdash (try\ e'\ catch(C\ V)\ e2,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1TryCatch1*:
$[\![\ P,\ e,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor);\ typeof\text{-}addr\ h\ a = \lfloor Class\text{-}type\ C' \rfloor;\ P \vdash C' \preceq^*\ C\ ]\!]$
$\Longrightarrow P,\ try\ e\ catch(C\ V)\ e2,\ h \vdash (\{V{:}Class\ C{=}None;\ e2\},\ xs[V := Addr\ a]) \leftrightarrow ([Addr\ a],\ loc,\ Suc$
$(length\ (compE2\ e)),\ None)$

| *bisim1TryCatch2*:
P, *e2*, *h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
⟹ *P*, *try e catch*(*C V*) *e2*, *h* ⊢ ({*V*:*Class C*=*None*; *e'*}, *xs*) ↔ (*stk*, *loc*, *Suc* (*Suc* (*length* (*compE2 e*) + *pc*)), *xcp*)

| *bisim1TryFail*:
⟦ *P*, *e*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋); *typeof-addr h a* = ⌊*Class-type C'*⌋; ¬ *P* ⊢ *C'* ≼* *C* ⟧
⟹ *P*, *try e catch*(*C V*) *e2*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)

| *bisim1TryCatchThrow*:
P, *e2*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
⟹ *P*, *try e catch*(*C V*) *e2*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *Suc* (*Suc* (*length* (*compE2 e*) + *pc*)), ⌊*a*⌋)

| *bisims1Nil*: *P*, [], *h* ⊢ ([], *xs*) [↔] ([], *xs*, *0*, *None*)

| *bisims1List1*:
P, *e*, *h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*) ⟹ *P*, *e*#*es*, *h* ⊢ (*e'*#*es*, *xs*) [↔] (*stk*, *loc*, *pc*, *xcp*)

| *bisims1List2*:
P, *es*, *h* ⊢ (*es'*, *xs*) [↔] (*stk*, *loc*, *pc*, *xcp*)
⟹ *P*, *e*#*es*, *h* ⊢ (*Val v* # *es'*, *xs*) [↔] (*stk* @ [*v*], *loc*, *length* (*compE2 e*) + *pc*, *xcp*)

inductive-cases *bisim1-cases*:
P,*e*,*h* ⊢ (*Val v*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

lemma *bisim1-refl*: *P*,*e*,*h* ⊢ (*e*, *xs*) ↔ ([], *xs*, *0*, *None*)
and *bisims1-refl*: *P*,*es*,*h* ⊢ (*es*, *xs*) [↔] ([], *xs*, *0*, *None*)
apply(*induct e* **and** *es rule*: *call.induct calls.induct*)
apply(*auto intro*: *bisim1-bisims1.intros simp add*: *nat-fun-sum-eq-conv*)
apply(*rename-tac option a*)
apply(*case-tac option*)
apply(*auto intro*: *bisim1-bisims1.intros split*: *if-split-asm*)
done

lemma *bisims1-lengthD*: *P*, *es*, *h* ⊢ (*es'*, *xs*) [↔] *s* ⟹ *length es* = *length es'*
apply(*induct es arbitrary*: *es' s*)
apply(*auto elim*: *bisims1.cases*)
done

Derive an alternative induction rule for *bisim1* such that (i) induction hypothesis are generated for all subexpressions and (ii) the number of surrounding blocks is passed through.

inductive *bisim1'* ::
'm prog ⇒ *'heap* ⇒ *'addr expr1* ⇒ *nat* ⇒ (*'addr expr1* × *'addr locals1*)
⇒ (*'addr val list* × *'addr val list* × *pc* × *'addr option*) ⇒ *bool*

and *bisims1'* ::
'm prog ⇒ *'heap* ⇒ *'addr expr1 list* ⇒ *nat* ⇒ (*'addr expr1 list* × *'addr locals1*)
⇒ (*'addr val list* × *'addr val list* × *pc* × *'addr option*) ⇒ *bool*

and *bisim1'-syntax* ::

'm prog ⇒ 'addr expr1 ⇒ nat ⇒ 'heap ⇒ ('addr expr1 × 'addr locals1)
⇒ ('addr val list × 'addr val list × pc × 'addr option) ⇒ bool
(‹-,-,-,- ⊢'' - ↔ -› [50, 0, 0, 0, 0, 50] 100)

and *bisims1'-syntax* ::
'm prog ⇒ 'addr expr1 list ⇒ nat ⇒ 'heap ⇒ ('addr expr1 list × 'addr val list)
⇒ ('addr val list × 'addr val list × pc × 'addr option) ⇒ bool
(‹-,-,-,- ⊢'' - [↔] -› [50, 0, 0, 0, 0, 50] 100)
for *P* :: *'m prog* **and** *h* :: *'heap*
where
P, e, n, h ⊢' exs ↔ s ≡ bisim1' P h e n exs s
| *P, es, n, h ⊢' esxs [↔] s ≡ bisims1' P h es n esxs s*

| *bisim1Val2'*:
P, e, n, h ⊢' (Val v, xs) ↔ (v # [], xs, length (compE2 e), None)

| *bisim1New'*:
P, new C, n, h ⊢' (new C, xs) ↔ ([], xs, 0, None)

| *bisim1NewThrow'*:
P, new C, n, h ⊢' (THROW OutOfMemory, xs) ↔ ([], xs, 0, ⌊addr-of-sys-xcpt OutOfMemory⌋)

| *bisim1NewArray'*:
P, e, n, h ⊢' (e', xs) ↔ (stk, loc, pc, xcp)
⟹ P, newA T⌊e⌉, n, h ⊢' (newA T⌊e'⌉, xs) ↔ (stk, loc, pc, xcp)

| *bisim1NewArrayThrow'*:
P, e, n, h ⊢' (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)
⟹ P, newA T⌊e⌉, n, h ⊢' (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)

| *bisim1NewArrayFail'*:
(⋀xs. P, e, n, h ⊢' (e, xs) ↔ ([], xs, 0, None))
⟹ P, newA T⌊e⌉, n, h ⊢' (Throw a, xs) ↔ ([v], xs, length (compE2 e), ⌊a⌋)

| *bisim1Cast'*:
P, e, n, h ⊢' (e', xs) ↔ (stk, loc, pc, xcp)
⟹ P, Cast T e, n, h ⊢' (Cast T e', xs) ↔ (stk, loc, pc, xcp)

| *bisim1CastThrow'*:
P, e, n, h ⊢' (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)
⟹ P, Cast T e, n, h ⊢' (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)

| *bisim1CastFail'*:
(⋀xs. P, e, n, h ⊢' (e, xs) ↔ ([], xs, 0, None))
⟹ P, Cast T e, n, h ⊢' (THROW ClassCast, xs) ↔ ([v], xs, length (compE2 e), ⌊addr-of-sys-xcpt ClassCast⌋)

| *bisim1InstanceOf'*:
P, e, n, h ⊢' (e', xs) ↔ (stk, loc, pc, xcp)
⟹ P, e instanceof T, n, h ⊢' (e' instanceof T, xs) ↔ (stk, loc, pc, xcp)

| *bisim1InstanceOfThrow'*:
$P,\ e,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$
$\Longrightarrow P,\ e\ instanceof\ T,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$

| *bisim1Val'*: $P,\ Val\ v,\ n,\ h \vdash' (Val\ v,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)$

| *bisim1Var'*: $P,\ Var\ V,\ n,\ h \vdash' (Var\ V,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)$

| *bisim1BinOp1'*:
$[\![P,\ e1,\ n,\ h \vdash' (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp);$
$\bigwedge xs.\ P,\ e2,\ n,\ h \vdash' (e2,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)]\!]$
$\Longrightarrow P,\ e1 \mathbin{«bop»} e2,\ n,\ h \vdash' (e' \mathbin{«bop»} e2,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1BinOp2'*:
$[\![P,\ e2,\ n,\ h \vdash' (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp);$
$\bigwedge xs.\ P,\ e1,\ n,\ h \vdash' (e1,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)]\!]$
$\Longrightarrow P,\ e1 \mathbin{«bop»} e2,\ n,\ h \vdash' (Val\ v1 \mathbin{«bop»} e',\ xs) \leftrightarrow (stk\ @\ [v1],\ loc,\ length\ (compE2\ e1) + pc,\ xcp)$

| *bisim1BinOpThrow1'*:
$[\![P,\ e1,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor);$
$\bigwedge xs.\ P,\ e2,\ n,\ h \vdash' (e2,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)]\!]$
$\Longrightarrow P,\ e1 \mathbin{«bop»} e2,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$

| *bisim1BinOpThrow2'*:
$[\![P,\ e2,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor);$
$\bigwedge xs.\ P,\ e1,\ n,\ h \vdash' (e1,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)]\!]$
$\Longrightarrow P,\ e1 \mathbin{«bop»} e2,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk\ @\ [v1],\ loc,\ length\ (compE2\ e1) + pc,\ \lfloor a \rfloor)$

| *bisim1BinOpThrow'*:
$[\![\bigwedge xs.\ P,\ e1,\ n,\ h \vdash' (e1,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None);$
$\bigwedge xs.\ P,\ e2,\ n,\ h \vdash' (e2,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)]\!]$
$\Longrightarrow P,\ e1 \mathbin{«bop»} e2,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow ([v1,\ v2],\ xs,\ length\ (compE2\ e1) + length\ (compE2\ e2),\ \lfloor a \rfloor)$

| *bisim1LAss1'*:
$P,\ e,\ n,\ h \vdash' (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ V{:=}e,\ n,\ h \vdash' (V{:=}e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1LAss2'*:
$(\bigwedge xs.\ P,\ e,\ n,\ h \vdash' (e,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None))$
$\Longrightarrow P,\ V{:=}e,\ n,\ h \vdash' (unit,\ xs) \leftrightarrow ([],\ xs,\ Suc\ (length\ (compE2\ e)),\ None)$

| *bisim1LAssThrow'*:
$P,\ e,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$
$\Longrightarrow P,\ V{:=}e,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$

| *bisim1AAcc1'*:
$[\![P,\ a,\ n,\ h \vdash' (a',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp);\ \bigwedge xs.\ P,\ i,\ n,\ h \vdash' (i,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)]\!]$
$\Longrightarrow P,\ a\lfloor i \rceil,\ n,\ h \vdash' (a'\lfloor i \rceil,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1AAcc2'*:

$[\![P, i, n, h \vdash' (i', xs) \leftrightarrow (stk, loc, pc, xcp); \bigwedge xs.\ P, a, n, h \vdash' (a, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, a\lfloor i \rceil, n, h \vdash' (Val\ v\lfloor i' \rceil, xs) \leftrightarrow (stk\ @\ [v], loc, length\ (compE2\ a) + pc, xcp)$

| *bisim1AAccThrow1'*:
$[\![P, a, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor);$
$\bigwedge xs.\ P, i, n, h \vdash' (i, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, a\lfloor i \rceil, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor)$

| *bisim1AAccThrow2'*:
$[\![P, i, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor);$
$\bigwedge xs.\ P, a, n, h \vdash' (a, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, a\lfloor i \rceil, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk\ @\ [v], loc, length\ (compE2\ a) + pc, \lfloor ad \rfloor)$

| *bisim1AAccFail'*:
$[\![\bigwedge xs.\ P, a, n, h \vdash' (a, xs) \leftrightarrow ([], xs, 0, None); \bigwedge xs.\ P, i, n, h \vdash' (i, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, a\lfloor i \rceil, n, h \vdash' (Throw\ ad, xs) \leftrightarrow ([v, v'], xs, length\ (compE2\ a) + length\ (compE2\ i), \lfloor ad \rfloor)$

| *bisim1AAss1'*:
$[\![P, a, n, h \vdash' (a', xs) \leftrightarrow (stk, loc, pc, xcp);$
$\bigwedge xs.\ P, i, n, h \vdash' (i, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e, n, h \vdash' (e, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, a\lfloor i \rceil := e, n, h \vdash' (a'\lfloor i \rceil := e, xs) \leftrightarrow (stk, loc, pc, xcp)$

| *bisim1AAss2'*:
$[\![P, i, n, h \vdash' (i', xs) \leftrightarrow (stk, loc, pc, xcp);$
$\bigwedge xs.\ P, a, n, h \vdash' (a, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e, n, h \vdash' (e, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, a\lfloor i \rceil := e, n, h \vdash' (Val\ v\lfloor i' \rceil := e, xs) \leftrightarrow (stk\ @\ [v], loc, length\ (compE2\ a) + pc, xcp)$

| *bisim1AAss3'*:
$[\![P, e, n, h \vdash' (e', xs) \leftrightarrow (stk, loc, pc, xcp);$
$\bigwedge xs.\ P, a, n, h \vdash' (a, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, i, n, h \vdash' (i, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, a\lfloor i \rceil := e, n, h \vdash' (Val\ v\lfloor Val\ v' \rceil := e', xs) \leftrightarrow (stk\ @\ [v', v], loc, length\ (compE2\ a) + length\ (compE2\ i) + pc, xcp)$

| *bisim1AAssThrow1'*:
$[\![P, a, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor);$
$\bigwedge xs.\ P, i, n, h \vdash' (i, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e, n, h \vdash' (e, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, a\lfloor i \rceil := e, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor)$

| *bisim1AAssThrow2'*:
$[\![P, i, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor);$
$\bigwedge xs.\ P, a, n, h \vdash' (a, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e, n, h \vdash' (e, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, a\lfloor i \rceil := e, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk\ @\ [v], loc, length\ (compE2\ a) + pc, \lfloor ad \rfloor)$

| *bisim1AAssThrow3'*:
$[\![P, e, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor);$
$\bigwedge xs.\ P, a, n, h \vdash' (a, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, i, n, h \vdash' (i, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, a\lfloor i \rceil := e, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk\ @\ [v', v], loc, length\ (compE2\ a) + length\ (compE2$

$i) + pc, \lfloor ad \rfloor)$

| *bisim1AAssFail'*:
$[\![\bigwedge xs.\ P, a, n, h \vdash' (a, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, i, n, h \vdash' (i, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e, n, h \vdash' (e, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, a\lfloor i \rceil := e, n, h \vdash' (Throw\ ad, xs) \leftrightarrow ([v', v, v''], xs, length\ (compE2\ a) + length\ (compE2\ i) + length\ (compE2\ e), \lfloor ad \rfloor)$

| *bisim1AAss4'*:
$[\![\bigwedge xs.\ P, a, n, h \vdash' (a, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, i, n, h \vdash' (i, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e, n, h \vdash' (e, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, a\lfloor i \rceil := e, n, h \vdash' (unit, xs) \leftrightarrow ([], xs, Suc\ (length\ (compE2\ a) + length\ (compE2\ i) + length\ (compE2\ e)), None)$

| *bisim1ALength'*:
$P, a, n, h \vdash' (a', xs) \leftrightarrow (stk, loc, pc, xcp)$
$\Longrightarrow P, a{\cdot}length, n, h \vdash' (a'{\cdot}length, xs) \leftrightarrow (stk, loc, pc, xcp)$

| *bisim1ALengthThrow'*:
$P, a, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor)$
$\Longrightarrow P, a{\cdot}length, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor)$

| *bisim1ALengthNull'*:
$(\bigwedge xs.\ P, a, n, h \vdash' (a, xs) \leftrightarrow ([], xs, 0, None))$
$\Longrightarrow P, a{\cdot}length, n, h \vdash' (THROW\ NullPointer, xs) \leftrightarrow ([Null], xs, length\ (compE2\ a), \lfloor addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer \rfloor)$

| *bisim1FAcc'*:
$P, e, n, h \vdash' (e', xs) \leftrightarrow (stk, loc, pc, xcp)$
$\Longrightarrow P, e{\cdot}F\{D\}, n, h \vdash' (e'{\cdot}F\{D\}, xs) \leftrightarrow (stk, loc, pc, xcp)$

| *bisim1FAccThrow'*:
$P, e, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor)$
$\Longrightarrow P, e{\cdot}F\{D\}, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor)$

| *bisim1FAccNull'*:
$(\bigwedge xs.\ P, e, n, h \vdash' (e, xs) \leftrightarrow ([], xs, 0, None))$
$\Longrightarrow P, e{\cdot}F\{D\}, n, h \vdash' (THROW\ NullPointer, xs) \leftrightarrow ([Null], xs, length\ (compE2\ e), \lfloor addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer \rfloor)$

| *bisim1FAss1'*:
$[\![P, e, n, h \vdash' (e', xs) \leftrightarrow (stk, loc, pc, xcp);$
$\bigwedge xs.\ P, e2, n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, e{\cdot}F\{D\} := e2, n, h \vdash' (e'{\cdot}F\{D\} := e2, xs) \leftrightarrow (stk, loc, pc, xcp)$

| *bisim1FAss2'*:
$[\![P, e2, n, h \vdash' (e', xs) \leftrightarrow (stk, loc, pc, xcp);$
$\bigwedge xs.\ P, e, n, h \vdash' (e, xs) \leftrightarrow ([], xs, 0, None)]\!]$

$\Longrightarrow P, e{\cdot}F\{D\} := e2, n, h \vdash' (Val\ v{\cdot}F\{D\} := e', xs) \leftrightarrow (stk\ @\ [v], loc, length\ (compE2\ e) + pc, xcp)$

| *bisim1FAssThrow1'*:
$[\![P, e, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor);$
$\bigwedge xs.\ P, e2, n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, e{\cdot}F\{D\} := e2, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor)$

| *bisim1FAssThrow2'*:
$[\![P, e2, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor);$
$\bigwedge xs.\ P, e, n, h \vdash' (e, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, e{\cdot}F\{D\} := e2, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk\ @\ [v], loc, length\ (compE2\ e) + pc, \lfloor ad \rfloor)$

| *bisim1FAssNull'*:
$[\![\bigwedge xs.\ P, e, n, h \vdash' (e, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e2, n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, e{\cdot}F\{D\} := e2, n, h \vdash' (THROW\ NullPointer, xs) \leftrightarrow ([v, Null], xs, length\ (compE2\ e) + length\ (compE2\ e2), \lfloor addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer \rfloor)$

| *bisim1FAss3'*:
$[\![\bigwedge xs.\ P, e, n, h \vdash' (e, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e2, n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, e{\cdot}F\{D\} := e2, n, h \vdash' (unit, xs) \leftrightarrow ([], xs, Suc\ (length\ (compE2\ e) + length\ (compE2\ e2)), None)$

| *bisim1CAS1'*:
$[\![P, e1, n, h \vdash' (e1', xs) \leftrightarrow (stk, loc, pc, xcp);$
$\bigwedge xs.\ P, e2, n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e3, n, h \vdash' (e3, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, e1{\cdot}compareAndSwap(D{\cdot}F, e2, e3), n, h \vdash' (e1'{\cdot}compareAndSwap(D{\cdot}F, e2, e3), xs) \leftrightarrow (stk, loc, pc, xcp)$

| *bisim1CAS2'*:
$[\![P, e2, n, h \vdash' (e2', xs) \leftrightarrow (stk, loc, pc, xcp);$
$\bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e3, n, h \vdash' (e3, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, e1{\cdot}compareAndSwap(D{\cdot}F, e2, e3), n, h \vdash' (Val\ v{\cdot}compareAndSwap(D{\cdot}F, e2', e3), xs) \leftrightarrow (stk\ @\ [v], loc, length\ (compE2\ e1) + pc, xcp)$

| *bisim1CAS3'*:
$[\![P, e3, n, h \vdash' (e3', xs) \leftrightarrow (stk, loc, pc, xcp);$
$\bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e2, n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, e1{\cdot}compareAndSwap(D{\cdot}F, e2, e3), n, h \vdash' (Val\ v{\cdot}compareAndSwap(D{\cdot}F, Val\ v', e3'), xs) \leftrightarrow (stk\ @\ [v', v], loc, length\ (compE2\ e1) + length\ (compE2\ e2) + pc, xcp)$

| *bisim1CASThrow1'*:
$[\![P, e1, n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor);$
$\bigwedge xs.\ P, e2, n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e3, n, h \vdash' (e3, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, e1{\cdot}compareAndSwap(D{\cdot}F, e2, e3), n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor)$

| *bisim1CASThrow2'*:

⟦ P, $e2$, n, $h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor)$;
  $\bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None)$;
  $\bigwedge xs.\ P, e3, n, h \vdash' (e3, xs) \leftrightarrow ([], xs, 0, None)$ ⟧
⟹ P, $e1 \cdot compareAndSwap(D \cdot F, e2, e3), n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk\ @\ [v], loc, length\ (compE2\ e1) + pc, \lfloor ad \rfloor)$

| *bisim1CASThrow3'*:
⟦ P, $e3$, n, $h \vdash' (Throw\ ad, xs) \leftrightarrow (stk, loc, pc, \lfloor ad \rfloor)$;
  $\bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None)$;
  $\bigwedge xs.\ P, e2, n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None)$ ⟧
⟹ P, $e1 \cdot compareAndSwap(D \cdot F, e2, e3), n, h \vdash' (Throw\ ad, xs) \leftrightarrow (stk\ @\ [v', v], loc, length\ (compE2\ e1) + length\ (compE2\ e2) + pc, \lfloor ad \rfloor)$

| *bisim1CASFail'*:
⟦ $\bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None)$;
  $\bigwedge xs.\ P, e2, n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None)$;
  $\bigwedge xs.\ P, e3, n, h \vdash' (e3, xs) \leftrightarrow ([], xs, 0, None)$ ⟧
⟹ P, $e1 \cdot compareAndSwap(D \cdot F, e2, e3), n, h \vdash' (Throw\ ad, xs) \leftrightarrow ([v', v, v''], xs, length\ (compE2\ e1) + length\ (compE2\ e2) + length\ (compE2\ e3), \lfloor ad \rfloor)$

| *bisim1Call1'*:
⟦ P, obj, n, $h \vdash' (obj', xs) \leftrightarrow (stk, loc, pc, xcp)$;
  $\bigwedge xs.\ P, ps, n, h \vdash' (ps, xs)\ [\leftrightarrow]\ ([], xs, 0, None)$ ⟧
⟹ P, $obj \cdot M(ps), n, h \vdash' (obj' \cdot M(ps), xs) \leftrightarrow (stk, loc, pc, xcp)$

| *bisim1CallParams'*:
⟦ P, ps, n, $h \vdash' (ps', xs)\ [\leftrightarrow]\ (stk, loc, pc, xcp)$; $ps \neq []$;
  $\bigwedge xs.\ P, obj, n, h \vdash' (obj, xs) \leftrightarrow ([], xs, 0, None)$ ⟧
⟹ P, $obj \cdot M(ps), n, h \vdash' (Val\ v \cdot M(ps'), xs) \leftrightarrow (stk\ @\ [v], loc, length\ (compE2\ obj) + pc, xcp)$

| *bisim1CallThrowObj'*:
⟦ P, obj, n, $h \vdash' (Throw\ a, xs) \leftrightarrow (stk, loc, pc, \lfloor a \rfloor)$;
  $\bigwedge xs.\ P, ps, n, h \vdash' (ps, xs)\ [\leftrightarrow]\ ([], xs, 0, None)$⟧
⟹ P, $obj \cdot M(ps), n, h \vdash' (Throw\ a, xs) \leftrightarrow (stk, loc, pc, \lfloor a \rfloor)$

| *bisim1CallThrowParams'*:
⟦ P, ps, n, $h \vdash' (map\ Val\ vs\ @\ Throw\ a\ \#\ ps', xs)\ [\leftrightarrow]\ (stk, loc, pc, \lfloor a \rfloor)$;
  $\bigwedge xs.\ P, obj, n, h \vdash' (obj, xs) \leftrightarrow ([], xs, 0, None)$ ⟧
⟹ P, $obj \cdot M(ps), n, h \vdash' (Throw\ a, xs) \leftrightarrow (stk\ @\ [v], loc, length\ (compE2\ obj) + pc, \lfloor a \rfloor)$

| *bisim1CallThrow'*:
⟦ $length\ ps = length\ vs$;
  $\bigwedge xs.\ P, obj, n, h \vdash' (obj, xs) \leftrightarrow ([], xs, 0, None)$; $\bigwedge xs.\ P, ps, n, h \vdash' (ps, xs)\ [\leftrightarrow]\ ([], xs, 0, None)$ ⟧
⟹ P, $obj \cdot M(ps), n, h \vdash' (Throw\ a, xs) \leftrightarrow (vs\ @\ [v], xs, length\ (compE2\ obj) + length\ (compEs2\ ps), \lfloor a \rfloor)$

| *bisim1BlockSome1'*:
$(\bigwedge xs.\ P, e, Suc\ n, h \vdash' (e, xs) \leftrightarrow ([], xs, 0, None))$
⟹ P, $\{V{:}T{=}\lfloor v \rfloor;\ e\}, n, h \vdash' (\{V{:}T{=}\lfloor v \rfloor;\ e\}, xs) \leftrightarrow ([], xs, 0, None)$

| *bisim1BlockSome2'*:
$(\bigwedge xs.\ P, e, Suc\ n, h \vdash' (e, xs) \leftrightarrow ([], xs, 0, None))$

$\Longrightarrow P, \{V{:}T{=}\lfloor v\rfloor;\ e\}, n, h \vdash' (\{V{:}T{=}\lfloor v\rfloor;\ e\}, xs) \leftrightarrow ([v], xs, Suc\ 0, None)$

| *bisim1BlockSome4'*:
$P, e, Suc\ n, h \vdash' (e', xs) \leftrightarrow (stk, loc, pc, xcp)$
$\Longrightarrow P, \{V{:}T{=}\lfloor v\rfloor;\ e\}, n, h \vdash' (\{V{:}T{=}None;\ e'\}, xs) \leftrightarrow (stk, loc, Suc\ (Suc\ pc), xcp)$

| *bisim1BlockThrowSome'*:
$P, e, Suc\ n, h \vdash' (Throw\ a, xs) \leftrightarrow (stk, loc, pc, \lfloor a\rfloor)$
$\Longrightarrow P, \{V{:}T{=}\lfloor v\rfloor;\ e\}, n, h \vdash' (Throw\ a, xs) \leftrightarrow (stk, loc, Suc\ (Suc\ pc), \lfloor a\rfloor)$

| *bisim1BlockNone'*:
$P, e, Suc\ n, h \vdash' (e', xs) \leftrightarrow (stk, loc, pc, xcp)$
$\Longrightarrow P, \{V{:}T{=}None;\ e\}, n, h \vdash' (\{V{:}T{=}None;\ e'\}, xs) \leftrightarrow (stk, loc, pc, xcp)$

| *bisim1BlockThrowNone'*:
$P, e, Suc\ n, h \vdash' (Throw\ a, xs) \leftrightarrow (stk, loc, pc, \lfloor a\rfloor)$
$\Longrightarrow P, \{V{:}T{=}None;\ e\}, n, h \vdash' (Throw\ a, xs) \leftrightarrow (stk, loc, pc, \lfloor a\rfloor)$

| *bisim1Sync1'*:
$[\![P, e1, n, h \vdash' (e', xs) \leftrightarrow (stk, loc, pc, xcp);$
$\quad \bigwedge xs.\ P, e2, Suc\ n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, sync_V\ (e1)\ e2, n, h \vdash' (sync_V\ (e')\ e2, xs) \leftrightarrow (stk, loc, pc, xcp)$

| *bisim1Sync2'*:
$[\![\bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None);$
$\quad \bigwedge xs.\ P, e2, Suc\ n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, sync_V\ (e1)\ e2, n, h \vdash' (sync_V\ (Val\ v)\ e2, xs) \leftrightarrow ([v, v], xs, Suc\ (length\ (compE2\ e1)), None)$

| *bisim1Sync3'*:
$[\![\bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None);$
$\quad \bigwedge xs.\ P, e2, Suc\ n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, sync_V\ (e1)\ e2, n, h \vdash' (sync_V\ (Val\ v)\ e2, xs) \leftrightarrow ([v], xs[V := v], Suc\ (Suc\ (length\ (compE2\ e1))), None)$

| *bisim1Sync4'*:
$[\![P, e2, Suc\ n, h \vdash' (e', xs) \leftrightarrow (stk, loc, pc, xcp);$
$\quad \bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, sync_V\ (e1)\ e2, n, h \vdash' (insync_V\ (a)\ e', xs) \leftrightarrow (stk, loc, Suc\ (Suc\ (Suc\ (length\ (compE2\ e1) + pc))), xcp)$

| *bisim1Sync5'*:
$[\![\bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None);$
$\quad \bigwedge xs.\ P, e2, Suc\ n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, sync_V\ (e1)\ e2, n, h \vdash' (insync_V\ (a)\ Val\ v, xs) \leftrightarrow ([xs\ !\ V, v], xs, 4 + length\ (compE2\ e1) + length\ (compE2\ e2), None)$

| *bisim1Sync6'*:
$[\![\bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None);$
$\quad \bigwedge xs.\ P, e2, Suc\ n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None)]\!]$
$\Longrightarrow P, sync_V\ (e1)\ e2, n, h \vdash' (Val\ v, xs) \leftrightarrow ([v], xs, 5 + length\ (compE2\ e1) + length\ (compE2\ e2), None)$

| *bisim1Sync7'*:

⟦ ⋀xs. P, e1, n, h ⊢′ (e1, xs) ↔ ([], xs, 0, None);
⋀xs. P, e2, Suc n, h ⊢′ (e2, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, sync_V (e1) e2, n, h ⊢′ (insync_V (a) Throw a′, xs) ↔ ([Addr a′], xs, 6 + length (compE2 e1) + length (compE2 e2), None)

| bisim1Sync8′:
⟦ ⋀xs. P, e1, n, h ⊢′ (e1, xs) ↔ ([], xs, 0, None);
⋀xs. P, e2, Suc n, h ⊢′ (e2, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, sync_V (e1) e2, n, h ⊢′ (insync_V (a) Throw a′, xs) ↔
([xs ! V, Addr a′], xs, 7 + length (compE2 e1) + length (compE2 e2), None)

| bisim1Sync9′:
⟦ ⋀xs. P, e1, n, h ⊢′ (e1, xs) ↔ ([], xs, 0, None);
⋀xs. P, e2, Suc n, h ⊢′ (e2, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, sync_V (e1) e2, n, h ⊢′ (Throw a, xs) ↔ ([Addr a], xs, 8 + length (compE2 e1) + length (compE2 e2), None)

| bisim1Sync10′:
⟦ ⋀xs. P, e1, n, h ⊢′ (e1, xs) ↔ ([], xs, 0, None);
⋀xs. P, e2, Suc n, h ⊢′ (e2, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, sync_V (e1) e2, n, h ⊢′ (Throw a, xs) ↔ ([Addr a], xs, 8 + length (compE2 e1) + length (compE2 e2), ⌊a⌋)

| bisim1Sync11′:
⟦ ⋀xs. P, e1, n, h ⊢′ (e1, xs) ↔ ([], xs, 0, None);
⋀xs. P, e2, Suc n, h ⊢′ (e2, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, sync_V (e1) e2, n, h ⊢′ (THROW NullPointer, xs) ↔ ([Null], xs, Suc (Suc (length (compE2 e1))), ⌊addr-of-sys-xcpt NullPointer⌋)

| bisim1Sync12′:
⟦ ⋀xs. P, e1, n, h ⊢′ (e1, xs) ↔ ([], xs, 0, None);
⋀xs. P, e2, Suc n, h ⊢′ (e2, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, sync_V (e1) e2, n, h ⊢′ (Throw a, xs) ↔ ([v, v′], xs, 4 + length (compE2 e1) + length (compE2 e2), ⌊a⌋)

| bisim1Sync14′:
⟦ ⋀xs. P, e1, n, h ⊢′ (e1, xs) ↔ ([], xs, 0, None);
⋀xs. P, e2, Suc n, h ⊢′ (e2, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, sync_V (e1) e2, n, h ⊢′ (Throw a, xs) ↔
([v, Addr a′], xs, 7 + length (compE2 e1) + length (compE2 e2), ⌊a⌋)

| bisim1SyncThrow′:
⟦ P, e1, n, h ⊢′ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋);
⋀xs. P, e2, Suc n, h ⊢′ (e2, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, sync_V (e1) e2, n, h ⊢′ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)

| bisim1InSync′:
P, insync_V (a) e, n, h ⊢′ (insync_V (a) e, xs) ↔ ([], xs, 0, None)

| bisim1Seq1′:
⟦ P, e1, n, h ⊢′ (e′, xs) ↔ (stk, loc, pc, xcp);
⋀xs. P, e2, n, h ⊢′ (e2, xs) ↔ ([], xs, 0, None) ⟧

$\Longrightarrow P,\ e1;;e2,\ n,\ h \vdash' (e';;e2,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1SeqThrow1′*:
$[\![\ P,\ e1,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor);$
$\bigwedge xs.\ P,\ e2,\ n,\ h \vdash' (e2,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ e1;;e2,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$

| *bisim1Seq2′*:
$[\![\ P,\ e2,\ n,\ h \vdash' (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp);$
$\bigwedge xs.\ P,\ e1,\ n,\ h \vdash' (e1,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ e1;;e2,\ n,\ h \vdash' (e',\ xs) \leftrightarrow (stk,\ loc,\ Suc\ (length\ (compE2\ e1) + pc),\ xcp)$

| *bisim1Cond1′*:
$[\![\ P,\ e,\ n,\ h \vdash' (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp);$
$\bigwedge xs.\ P,\ e1,\ n,\ h \vdash' (e1,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None);$
$\bigwedge xs.\ P,\ e2,\ n,\ h \vdash' (e2,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ if\ (e)\ e1\ else\ e2,\ n,\ h \vdash' (if\ (e')\ e1\ else\ e2,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1CondThen′*:
$[\![\ P,\ e1,\ n,\ h \vdash' (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp);$
$\bigwedge xs.\ P,\ e,\ n,\ h \vdash' (e,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None);$
$\bigwedge xs.\ P,\ e2,\ n,\ h \vdash' (e2,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ if\ (e)\ e1\ else\ e2,\ n,\ h \vdash' (e',\ xs) \leftrightarrow (stk,\ loc,\ Suc\ (length\ (compE2\ e) + pc),\ xcp)$

| *bisim1CondElse′*:
$[\![\ P,\ e2,\ n,\ h \vdash' (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp);$
$\bigwedge xs.\ P,\ e,\ n,\ h \vdash' (e,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None);$
$\bigwedge xs.\ P,\ e1,\ n,\ h \vdash' (e1,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ if\ (e)\ e1\ else\ e2,\ n,\ h \vdash' (e',\ xs) \leftrightarrow (stk,\ loc,\ Suc\ (Suc\ (length\ (compE2\ e) + length\ (compE2\ e1) +\ pc)),\ xcp)$

| *bisim1CondThrow′*:
$[\![\ P,\ e,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor);$
$\bigwedge xs.\ P,\ e1,\ n,\ h \vdash' (e1,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None);$
$\bigwedge xs.\ P,\ e2,\ n,\ h \vdash' (e2,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ if\ (e)\ e1\ else\ e2,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$

| *bisim1While1′*:
$[\![\ \bigwedge xs.\ P,\ c,\ n,\ h \vdash' (c,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None);$
$\bigwedge xs.\ P,\ e,\ n,\ h \vdash' (e,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ while\ (c)\ e,\ n,\ h \vdash' (while\ (c)\ e,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)$

| *bisim1While3′*:
$[\![\ P,\ c,\ n,\ h \vdash' (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp);$
$\bigwedge xs.\ P,\ e,\ n,\ h \vdash' (e,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ while\ (c)\ e,\ n,\ h \vdash' (if\ (e')\ (e;;\ while\ (c)\ e)\ else\ unit,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1While4′*:
$[\![\ P,\ e,\ n,\ h \vdash' (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp);$
$\bigwedge xs.\ P,\ c,\ n,\ h \vdash' (c,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ while\ (c)\ e,\ n,\ h\ \vdash' (e';;\ while\ (c)\ e,\ xs) \leftrightarrow (stk,\ loc,\ Suc\ (length\ (compE2\ c) + pc),\ xcp)$

| *bisim1While6'*:
⟦ ⋀*xs*. *P*, *c*, *n*, *h* ⊢′ (*c*, *xs*) ↔ ([], *xs*, *0*, *None*);
⋀*xs*. *P*, *e*, *n*, *h* ⊢′ (*e*, *xs*) ↔ ([], *xs*, *0*, *None*) ⟧ ⟹
P, *while* (*c*) *e*, *n*, *h* ⊢′ (*while* (*c*) *e*, *xs*) ↔ ([], *xs*, *Suc* (*Suc* (*length* (*compE2 c*) + *length* (*compE2 e*))), *None*)

| *bisim1While7'*:
⟦ ⋀*xs*. *P*, *c*, *n*, *h* ⊢′ (*c*, *xs*) ↔ ([], *xs*, *0*, *None*);
⋀*xs*. *P*, *e*, *n*, *h* ⊢′ (*e*, *xs*) ↔ ([], *xs*, *0*, *None*) ⟧ ⟹
P, *while* (*c*) *e*, *n*, *h* ⊢′ (*unit*, *xs*) ↔ ([], *xs*, *Suc* (*Suc* (*Suc* (*length* (*compE2 c*) + *length* (*compE2 e*)))), *None*)

| *bisim1WhileThrow1'*:
⟦ *P*, *c*, *n*, *h* ⊢′ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋);
⋀*xs*. *P*, *e*, *n*, *h* ⊢′ (*e*, *xs*) ↔ ([], *xs*, *0*, *None*) ⟧
⟹ *P*, *while* (*c*) *e*, *n*, *h* ⊢′ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)

| *bisim1WhileThrow2'*:
⟦ *P*, *e*, *n*, *h* ⊢′ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋);
⋀*xs*. *P*, *c*, *n*, *h* ⊢′ (*c*, *xs*) ↔ ([], *xs*, *0*, *None*) ⟧
⟹ *P*, *while* (*c*) *e*, *n*, *h* ⊢′ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *Suc* (*length* (*compE2 c*) + *pc*), ⌊*a*⌋)

| *bisim1Throw1'*:
P, *e*, *n*, *h* ⊢′ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
⟹ *P*, *throw e*, *n*, *h* ⊢′ (*throw e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1Throw2'*:
(⋀*xs*. *P*, *e*, *n*, *h* ⊢′ (*e*, *xs*) ↔ ([], *xs*, *0*, *None*))
⟹ *P*, *throw e*, *n*, *h* ⊢′ (*Throw a*, *xs*) ↔ ([*Addr a*], *xs*, *length* (*compE2 e*), ⌊*a*⌋)

| *bisim1ThrowNull'*:
(⋀*xs*. *P*, *e*, *n*, *h* ⊢′ (*e*, *xs*) ↔ ([], *xs*, *0*, *None*))
⟹ *P*, *throw e*, *n*, *h* ⊢′ (*THROW NullPointer*, *xs*) ↔ ([*Null*], *xs*, *length* (*compE2 e*), ⌊*addr-of-sys-xcpt NullPointer*⌋)

| *bisim1ThrowThrow'*:
P, *e*, *n*, *h* ⊢′ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
⟹ *P*, *throw e*, *n*, *h* ⊢′ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)

| *bisim1Try'*:
⟦ *P*, *e*, *n*, *h* ⊢′ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*);
⋀*xs*. *P*, *e2*, *Suc n*, *h* ⊢′ (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*) ⟧
⟹ *P*, *try e catch*(*C V*) *e2*, *n*, *h* ⊢′ (*try e' catch*(*C V*) *e2*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1TryCatch1'*:
⟦ *P*, *e*, *n*, *h* ⊢′ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋); *typeof-addr h a* = ⌊*Class-type C'*⌋; *P* ⊢ *C'* ≼* *C*;
⋀*xs*. *P*, *e2*, *Suc n*, *h* ⊢′ (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*) ⟧
⟹ *P*, *try e catch*(*C V*) *e2*, *n*, *h* ⊢′ ({*V*:*Class C*=*None*; *e2*}, *xs*[*V* := *Addr a*]) ↔ ([*Addr a*], *loc*, *Suc* (*length* (*compE2 e*)), *None*)

| *bisim1TryCatch2'*:
⟦ *P*, *e2*, *Suc n*, *h* ⊢′ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*);

$\bigwedge xs.\ P,\ e,\ n,\ h \vdash' (e,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ try\ e\ catch(C\ V)\ e2,\ n,\ h \vdash' (\{V{:}Class\ C{=}None;\ e'\},\ xs) \leftrightarrow (stk,\ loc,\ Suc\ (Suc\ (length\ (compE2\ e) + pc)),\ xcp)$

| *bisim1TryFail'*:
$[\![P,\ e,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor);\ typeof\text{-}addr\ h\ a = \lfloor Class\text{-}type\ C' \rfloor;\ \neg\ P \vdash C' \preceq^* C;$
$\bigwedge xs.\ P,\ e2,\ Suc\ n,\ h \vdash' (e2,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ try\ e\ catch(C\ V)\ e2,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$

| *bisim1TryCatchThrow'*:
$[\![P,\ e2,\ Suc\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor);$
$\bigwedge xs.\ P,\ e,\ n,\ h \vdash' (e,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ try\ e\ catch(C\ V)\ e2,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ Suc\ (Suc\ (length\ (compE2\ e) + pc)),\ \lfloor a \rfloor)$

| *bisims1Nil'*: $P,\ [],\ n,\ h \vdash' ([],\ xs)\ [\leftrightarrow]\ ([],\ xs,\ 0,\ None)$

| *bisims1List1'*:
$[\![P,\ e,\ n,\ h \vdash' (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp);$
$\bigwedge xs.\ P,\ es,\ n,\ h \vdash' (es,\ xs)\ [\leftrightarrow]\ ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ e\#es,\ n,\ h \vdash' (e'\#es,\ xs)\ [\leftrightarrow]\ (stk,\ loc,\ pc,\ xcp)$

| *bisims1List2'*:
$[\![P,\ es,\ n,\ h \vdash' (es',\ xs)\ [\leftrightarrow]\ (stk,\ loc,\ pc,\ xcp);$
$\bigwedge xs.\ P,\ e,\ n,\ h \vdash' (e,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ e\#es,\ n,\ h \vdash' (Val\ v\ \#\ es',\ xs)\ [\leftrightarrow]\ (stk\ @\ [v],\ loc,\ length\ (compE2\ e) + pc,\ xcp)$

lemma *bisim1'-refl*: $P,e,n,h \vdash' (e,xs) \leftrightarrow ([],xs,0,None)$
and *bisims1'-refl*: $P,es,n,h \vdash' (es,xs)\ [\leftrightarrow]\ ([],xs,0,None)$
apply(*induct e* **and** *es arbitrary*: *n xs* **and** *n xs rule*: *call.induct calls.induct*)
apply(*auto intro*: *bisim1'-bisims1'.intros simp add*: *nat-fun-sum-eq-conv*)
apply(*rename-tac option a b c*)
apply(*case-tac option*)
apply(*auto intro*: *bisim1'-bisims1'.intros simp add*: *fun-eq-iff split*: *if-split-asm*)
done

lemma *bisim1-imp-bisim1'*: $P,\ e,\ h \vdash exs \leftrightarrow s \Longrightarrow P,\ e,\ n,\ h \vdash' exs \leftrightarrow s$
and *bisims1-imp-bisims1'*: $P,\ es,\ h \vdash esxs\ [\leftrightarrow]\ s \Longrightarrow P,\ es,\ n,\ h \vdash' esxs\ [\leftrightarrow]\ s$
proof(*induct arbitrary*: *n* **and** *n rule*: *bisim1-bisims1.inducts*)
case (*bisim1CallParams ps ps' xs stk loc pc xcp obj M v*)
show *?case*
proof(*cases ps = []*)
case *True*
with ‹$P,ps,h \vdash (ps',\ xs)\ [\leftrightarrow]\ (stk,\ loc,\ pc,\ xcp)$› **have** $ps' = []\ pc = 0\ stk = []\ loc = xs\ xcp = None$
by(*auto elim*: *bisims1.cases*)
moreover have $P,obj,n,h \vdash' (Val\ v,xs) \leftrightarrow ([v],xs,length\ (compE2\ obj),None)$
by(*blast intro*: *bisim1Val2' bisim1'-refl*)
hence $P,obj{\cdot}M([]),n,h \vdash' (Val\ v{\cdot}M([]),xs) \leftrightarrow ([v],xs,length\ (compE2\ obj),None)$
by−(*rule bisim1Call1', auto intro*!: *bisims1Nil' simp add*: *bsoks-def*)
ultimately show *?thesis* **using** *True* **by** *simp*
next
case *False* **with** *bisim1CallParams* **show** *?thesis*
by(*auto intro*: *bisim1CallParams' bisims1'-refl bisim1'-refl*)

qed
qed(*auto intro*: *bisim1'-bisims1'.intros bisim1'-refl bisims1'-refl*)

lemma *bisim1'-imp-bisim1*: P, e, n, $h \vdash' exs \leftrightarrow s \Longrightarrow P$, e, $h \vdash exs \leftrightarrow s$
and *bisims1'-imp-bisims1*: P, es, n, $h \vdash' esxs\ [\leftrightarrow]\ s \Longrightarrow P$, es, $h \vdash esxs\ [\leftrightarrow]\ s$
apply(*induct rule*: *bisim1'-bisims1'.inducts*)
apply(*blast intro*: *bisim1-bisims1.intros*)+
done

lemma *bisim1'-eq-bisim1*: *bisim1' P h e n* = *bisim1 P h e*
and *bisims1'-eq-bisims1*: *bisims1' P h es n* = *bisims1 P h es*
by(*blast intro*!: *ext bisim1-imp-bisim1' bisims1-imp-bisims1' bisim1'-imp-bisim1 bisims1'-imp-bisims1*)+

end

lemmas *bisim1-bisims1-inducts* =
J1-JVM-heap-base.bisim1'-bisims1'.inducts
[*simplified J1-JVM-heap-base.bisim1'-eq-bisim1 J1-JVM-heap-base.bisims1'-eq-bisims1*,
consumes 1,
case-names bisim1Val2 bisim1New bisim1NewThrow
bisim1NewArray bisim1NewArrayThrow bisim1NewArrayFail bisim1Cast bisim1CastThrow bisim1Cast-Fail
bisim1InstanceOf bisim1InstanceOfThrow
bisim1Val bisim1Var bisim1BinOp1 bisim1BinOp2 bisim1BinOpThrow1 bisim1BinOpThrow2 bisim1BinOpThrow
bisim1LAss1 bisim1LAss2 bisim1LAssThrow
bisim1AAcc1 bisim1AAcc2 bisim1AAccThrow1 bisim1AAccThrow2 bisim1AAccFail
bisim1AAss1 bisim1AAss2 bisim1AAss3 bisim1AAssThrow1 bisim1AAssThrow2
bisim1AAssThrow3 bisim1AAssFail bisim1AAss4
bisim1ALength bisim1ALengthThrow bisim1ALengthNull
bisim1FAcc bisim1FAccThrow bisim1FAccNull
bisim1FAss1 bisim1FAss2 bisim1FAssThrow1 bisim1FAssThrow2 bisim1FAssNull bisim1FAss3
bisim1CAS1 bisim1CAS2 bisim1CAS3 bisim1CASThrow1 bisim1CASThrow2
bisim1CASThrow3 bisim1CASFail
bisim1Call1 bisim1CallParams bisim1CallThrowObj bisim1CallThrowParams
bisim1CallThrow
bisim1BlockSome1 bisim1BlockSome2 bisim1BlockSome4 bisim1BlockThrowSome
bisim1BlockNone bisim1BlockThrowNone
bisim1Sync1 bisim1Sync2 bisim1Sync3 bisim1Sync4 bisim1Sync5 bisim1Sync6
bisim1Sync7 bisim1Sync8 bisim1Sync9 bisim1Sync10 bisim1Sync11 bisim1Sync12
bisim1Sync14 bisim1SyncThrow bisim1InSync
bisim1Seq1 bisim1SeqThrow1 bisim1Seq2
bisim1Cond1 bisim1CondThen bisim1CondElse bisim1CondThrow
bisim1While1 bisim1While3 bisim1While4
bisim1While6 bisim1While7 bisim1WhileThrow1 bisim1WhileThrow2
bisim1Throw1 bisim1Throw2 bisim1ThrowNull bisim1ThrowThrow
bisim1Try bisim1TryCatch1 bisim1TryCatch2 bisim1TryFail bisim1TryCatchThrow
bisims1Nil bisims1List1 bisims1List2]

lemmas *bisim1-bisims1-inducts-split* = *bisim1-bisims1-inducts*[*split-format* (*complete*)]

context *J1-JVM-heap-base* **begin**

lemma *bisim1-pc-length-compE2*: $P,E,h \vdash (e, xs) \leftrightarrow (stk, loc, pc, xcp) \Longrightarrow pc \leq length\ (compE2\ E)$
and *bisims1-pc-length-compEs2*: $P,Es,h \vdash (es, xs)\ [\leftrightarrow]\ (stk, loc, pc, xcp) \Longrightarrow pc \leq length\ (compEs2\ Es)$
apply(*induct* (*stk*, *loc*, *pc*, *xcp*) **and** (*stk*, *loc*, *pc*, *xcp*)
arbitrary: *stk loc pc xcp* **and** *stk loc pc xcp rule*: *bisim1-bisims1.inducts*)
apply(*auto*)
done

lemma *bisim1-pc-length-compE2D*:
$P,e,h \vdash (e', xs) \leftrightarrow (stk,loc,length\ (compE2\ e),xcp)$
$\Longrightarrow xcp = None \wedge call1\ e' = None \wedge (\exists v.\ stk = [v] \wedge (is\text{-}val\ e' \longrightarrow e' = Val\ v \wedge xs = loc))$

and *bisims1-pc-length-compEs2D*:
$P,es,h \vdash (es', xs)\ [\leftrightarrow]\ (stk,loc,length\ (compEs2\ es),xcp)$
$\Longrightarrow xcp = None \wedge calls1\ es' = None \wedge (\exists vs.\ stk = rev\ vs \wedge length\ vs = length\ es \wedge (is\text{-}vals\ es' \longrightarrow es' = map\ Val\ vs \wedge xs = loc))$
proof(*induct* (*e'*, *xs*) (*stk*, *loc*, *length* (*compE2 e*), *xcp*)
and (*es'*, *xs*) (*stk*, *loc*, *length* (*compEs2 es*), *xcp*)
arbitrary: *e' xs stk loc xcp* **and** *es' xs stk loc xcp rule*: *bisim1-bisims1.inducts*)
case (*bisims1List2 es es' xs stk loc pc xcp e v*)
then obtain *vs* **where** *xcp = None calls1 es' = None*
stk = rev vs length vs = length es is-vals es' ⟶ es' = map Val vs ∧ xs = loc **by** *auto*
thus *?case*
by(*clarsimp*)(*rule-tac x=v#vs* **in** *exI*, *auto*)
qed(*simp-all* (*no-asm-use*), (*fastforce dest*: *bisim1-pc-length-compE2 bisims1-pc-length-compEs2 split*: *bop.split-asm if-split-asm*)+)

corollary *bisim1-call-pcD*: $[\![\ P,e,h \vdash (e', xs) \leftrightarrow (stk, loc, pc, xcp);\ call1\ e' = \lfloor aMvs \rfloor\ ]\!] \Longrightarrow pc < length\ (compE2\ e)$
and *bisims1-calls-pcD*: $[\![\ P,es,h \vdash (es', xs)\ [\leftrightarrow]\ (stk, loc, pc, xcp);\ calls1\ es' = \lfloor aMvs \rfloor\ ]\!] \Longrightarrow pc < length\ (compEs2\ es)$
proof −
assume *bisim*: $P,e,h \vdash (e', xs) \leftrightarrow (stk, loc, pc, xcp)$
and *call*: $call1\ e' = \lfloor aMvs \rfloor$

{ assume $pc = length\ (compE2\ e)$
with *bisim call* **have** *False*
by(*auto dest*: *bisim1-pc-length-compE2D*) **}**
moreover from *bisim* **have** $pc \leq length\ (compE2\ e)$
by(*rule bisim1-pc-length-compE2*)
ultimately show $pc < length\ (compE2\ e)$
by(*cases* $pc < length\ (compE2\ e)$)(*auto*)
next
assume *bisim*: $P,es,h \vdash (es', xs)\ [\leftrightarrow]\ (stk, loc, pc, xcp)$
and *call*: $calls1\ es' = \lfloor aMvs \rfloor$
{ assume $pc = length\ (compEs2\ es)$
with *bisim call* **have** *False*
by(*auto dest*: *bisims1-pc-length-compEs2D*) **}**
moreover from *bisim* **have** $pc \leq length\ (compEs2\ es)$
by(*rule bisims1-pc-length-compEs2*)
ultimately show $pc < length\ (compEs2\ es)$
by(*cases* $pc < length\ (compEs2\ es)$)(*auto*)
qed

lemma *bisim1-length-xs*: $P,e,h \vdash (e',xs) \leftrightarrow (stk, loc, pc, xcp) \Longrightarrow length\ xs = length\ loc$
and *bisims1-length-xs*: $P,es,h \vdash (es',xs)\ [\leftrightarrow]\ (stk, loc, pc, xcp) \Longrightarrow length\ xs = length\ loc$
by(*induct* $(e',xs)\ (stk, loc, pc, xcp)$ **and** $(es',xs)\ (stk, loc, pc, xcp)$
arbitrary: $e'\ xs\ stk\ loc\ pc\ xcp$ **and** $es'\ xs\ stk\ loc\ pc\ xcp$ *rule*: *bisim1-bisims1.inducts*)
auto

lemma *bisim1-Val-length-compE2D*:
$P,e,h \vdash (Val\ v,xs) \leftrightarrow (stk, loc, length\ (compE2\ e), xcp) \Longrightarrow stk = [v] \wedge xs = loc \wedge xcp = None$

and *bisims1-Val-length-compEs2D*:
$P,es,h \vdash (map\ Val\ vs,xs)\ [\leftrightarrow]\ (stk, loc, length\ (compEs2\ es), xcp) \Longrightarrow stk = rev\ vs \wedge xs = loc \wedge xcp = None$
by(*auto dest*: *bisim1-pc-length-compE2D bisims1-pc-length-compEs2D*)

lemma *bisim-Val-loc-eq-xcp-None*:
$P,\ e,\ h \vdash (Val\ v,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp) \Longrightarrow xs = loc \wedge xcp = None$

and *bisims-Val-loc-eq-xcp-None*:
$P,\ es,\ h \vdash (map\ Val\ vs,\ xs)\ [\leftrightarrow]\ (stk,\ loc,\ pc,\ xcp) \Longrightarrow xs = loc \wedge xcp = None$
apply(*induct* $(Val\ v :: 'addr\ expr1,\ xs)\ (stk,\ loc,\ pc,\ xcp)$
and $(map\ Val\ vs :: 'addr\ expr1\ list,\ xs)\ (stk,\ loc,\ pc,\ xcp)$
arbitrary: $v\ xs\ stk\ loc\ pc\ xcp$ **and** $vs\ xs\ stk\ loc\ pc\ xcp$ *rule*: *bisim1-bisims1.inducts*)
apply(*auto*)
done

lemma *bisim-Val-pc-not-Invoke*:
$[\![\ P,e,h \vdash (Val\ v,xs) \leftrightarrow (stk,loc,pc,xcp);\ pc < length\ (compE2\ e)\ ]\!] \Longrightarrow compE2\ e\ !\ pc \neq Invoke\ M\ n'$

and *bisims-Val-pc-not-Invoke*:
$[\![\ P,es,h \vdash (map\ Val\ vs,xs)\ [\leftrightarrow]\ (stk,loc,pc,xcp);\ pc < length\ (compEs2\ es)\ ]\!] \Longrightarrow compEs2\ es\ !\ pc \neq Invoke\ M\ n'$
apply(*induct* $(Val\ v :: 'addr\ expr1,\ xs)\ (stk,\ loc,\ pc,\ xcp)$
and $(map\ Val\ vs :: 'addr\ expr1\ list,\ xs)\ (stk,\ loc,\ pc,\ xcp)$
arbitrary: $v\ xs\ stk\ loc\ pc\ xcp$ **and** $vs\ xs\ stk\ loc\ pc\ xcp$ *rule*: *bisim1-bisims1.inducts*)
apply(*auto simp add*: *nth-append compEs2-map-Val dest*: *bisim1-pc-length-compE2*)
done

lemma *bisim1-VarD*: $P,\ E,\ h \vdash (Var\ V,xs) \leftrightarrow (stk,loc,pc,xcp) \Longrightarrow xs = loc$
and $P,\ es,\ h \vdash (es',\ xs)\ [\leftrightarrow]\ (stk,\ loc,\ pc,\ xcp) \Longrightarrow True$
by(*induct* $(Var\ V :: 'addr\ expr1,\ xs)\ (stk,\ loc,\ pc,\ xcp)$ **and** *arbitrary*: $V\ xs\ stk\ loc\ pc\ xcp$ **and** *rule*: *bisim1-bisims1.inducts*) *auto*

lemma *bisim1-ThrowD*:
$P,\ e,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow pc < length\ (compE2\ e) \wedge (xcp = \lfloor a \rfloor \vee xcp = None) \wedge xs = loc$

and *bisims1-ThrowD*:
$P,\ es,\ h \vdash (map\ Val\ vs\ @\ Throw\ a\ \#\ es',\ xs)\ [\leftrightarrow]\ (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow pc < length\ (compEs2\ es) \wedge (xcp = \lfloor a \rfloor \vee xcp = None) \wedge xs = loc$
apply(*induct* $(Throw\ a :: 'addr\ expr1,\ xs)\ (stk,\ loc,\ pc,\ xcp)$
and $(map\ Val\ vs\ @\ Throw\ a\ \#\ es',\ xs)\ (stk,\ loc,\ pc,\ xcp)$
arbitrary: $xs\ stk\ loc\ pc\ xcp$ **and** $vs\ es'\ xs\ stk\ loc\ pc\ xcp$ *rule*: *bisim1-bisims1.inducts*)
apply(*fastforce dest*: *bisim1-pc-length-compE2 bisim-Val-loc-eq-xcp-None simp add*: *Cons-eq-append-conv*)+
done

lemma fixes P :: *'addr J1-prog*
shows *bisim1-Invoke-stkD*:
$[\![P,e,h \vdash exs \leftrightarrow (stk,loc,pc,None);\ pc < length\ (compE2\ e);\ compE2\ e\ !\ pc = Invoke\ M\ n']\!]$
$\Longrightarrow \exists vs\ v\ stk'.\ stk = vs\ @\ v\ \#\ stk' \wedge length\ vs = n'$

and *bisims1-Invoke-stkD*:
$[\![P,es,h \vdash esxs\ [\leftrightarrow]\ (stk,loc,pc,None);\ pc < length\ (compEs2\ es);\ compEs2\ es\ !\ pc = Invoke\ M\ n']\!]$
$\Longrightarrow \exists vs\ v\ stk'.\ stk = vs\ @\ v\ \#\ stk' \wedge length\ vs = n'$
proof(*induct* (*stk*, *loc*, *pc*, *None* :: *'addr option*) **and** (*stk*, *loc*, *pc*, *None* :: *'addr option*)
arbitrary: *stk loc pc* **and** *stk loc pc rule*: *bisim1-bisims1.inducts*)
case *bisim1Call1*
thus *?case*
apply(*clarsimp simp add*: *nth-append append-eq-append-conv2 neq-Nil-conv split*: *if-split-asm*)
apply(*drule bisim1-pc-length-compE2*, *clarsimp simp add*: *neq-Nil-conv nth-append*)
apply(*frule bisim1-pc-length-compE2*, *clarsimp*)
apply(*drule bisim1-pc-length-compE2D*, *fastforce*)
done
next
case *bisim1CallParams* **thus** *?case*
apply(*clarsimp simp add*: *nth-append append-eq-Cons-conv split*: *if-split-asm*)
apply(*fastforce simp add*: *append-eq-append-conv2 Cons-eq-append-conv*)
apply(*frule bisims1-pc-length-compEs2*, *clarsimp*)
apply(*drule bisims1-pc-length-compEs2D*, *fastforce simp add*: *append-eq-append-conv2*)
done
qed(*fastforce simp add*: *nth-append append-eq-append-conv2 neq-Nil-conv split*: *if-split-asm bop.split-asm dest*: *bisim1-pc-length-compE2 bisims1-pc-length-compEs2*)+

lemma fixes P :: *'addr J1-prog*
shows *bisim1-call-xcpNone*: $P,e,h \vdash (e',xs) \leftrightarrow (stk,loc,pc,\lfloor a \rfloor) \Longrightarrow call1\ e' = None$
and *bisims1-calls-xcpNone*: $P,es,h \vdash (es',xs)\ [\leftrightarrow]\ (stk,loc,pc,\lfloor a \rfloor) \Longrightarrow calls1\ es' = None$
apply(*induct* (e', *xs*) (*stk*, *loc*, *pc*, $\lfloor a$ :: *'addr*$\rfloor$) **and** (*es'*,*xs*) (*stk*, *loc*, *pc*, $\lfloor a$ :: *'addr*$\rfloor$)
arbitrary: e' *xs stk loc pc* **and** es' *xs stk loc pc rule*: *bisim1-bisims1.inducts*)
apply(*auto dest*: *bisim-Val-loc-eq-xcp-None bisims-Val-loc-eq-xcp-None simp add*: *is-vals-conv*)
done

lemma *bisims1-map-Val-append*:
assumes *bisim*: P, es', $h \vdash (es''$, *xs*$)$ $[\leftrightarrow]$ (*stk*, *loc*, *pc*, *xcp*)
shows *length es* = *length vs*
$\Longrightarrow P$, *es* @ es', $h \vdash$ (*map Val vs* @ es'', *xs*) $[\leftrightarrow]$ (*stk* @ *rev vs*, *loc*, *length* (*compEs2 es*) + *pc*, *xcp*)
proof(*induction vs arbitrary*: *es*)
case *Nil* **thus** *?case* **using** *bisim* **by** *simp*
next
case (*Cons v vs*)
from ‹*length es* = *length* (*v* # *vs*)› **obtain** *e es'''* **where** [*simp*]: *es* = *e* # *es'''* **by**(*cases es*, *auto*)
with ‹*length es* = *length* (*v* # *vs*)› **have** *len*: *length es'''* = *length vs* **by** *simp*
from *Cons.IH*[*OF len*]
show *?case* **by**(*simp add*: *add.assoc append-assoc*[*symmetric*] *del*: *append-assoc*)(*rule bisims1List2*, *auto*)
qed

lemma *bisim1-hext-mono*: $[\![P,e,h \vdash exs \leftrightarrow s;\ hext\ h\ h']\!] \Longrightarrow P,e,h' \vdash exs \leftrightarrow s$ (**is** *PROP ?thesis1*)
and *bisims1-hext-mono*: $[\![P,es,h \vdash esxs\ [\leftrightarrow]\ s;\ hext\ h\ h']\!] \Longrightarrow P,es,h' \vdash esxs\ [\leftrightarrow]\ s$ (**is** *PROP*

?thesis2)
proof −
assume *hext*: *hext h h′*
have *P*,*e*,*h* ⊢ *exs* [↔] *s* ⟹ *P*,*e*,*h′* ⊢ *exs* [↔] *s*
and *P*,*es*,*h* ⊢ *esxs* [↔] *s* ⟹ *P*,*es*,*h′* ⊢ *esxs* [↔] *s*
apply(*induct rule*: *bisim1-bisims1.inducts*)
apply(*insert hext*)
apply(*auto intro*: *bisim1-bisims1.intros dest*: *hext-objD*)
done
thus *PROP ?thesis1* **and** *PROP ?thesis2* **by** *auto*
qed

declare *match-ex-table-append-not-pcs* [*simp*]
match-ex-table-eq-NoneI[*simp*]
outside-pcs-compxE2-not-matches-entry[*simp*]
outside-pcs-compxEs2-not-matches-entry[*simp*]

lemma *bisim1-xcp-Some-not-caught*:
P, *e*, *h* ⊢ (*Throw a*, *xs*) [↔] (*stk*, *loc*, *pc*, ⌊*a*⌋)
⟹ *match-ex-table* (*compP f P*) (*cname-of h a*) (*pc′* + *pc*) (*compxE2 e pc′ d*) = *None*

and *bisims1-xcp-Some-not-caught*:
P, *es*, *h* ⊢ (*map Val vs* @ *Throw a* # *es′*, *xs*) [↔] (*stk*, *loc*, *pc*, ⌊*a*⌋)
⟹ *match-ex-table* (*compP f P*) (*cname-of h a*) (*pc′* + *pc*) (*compxEs2 es pc′ d*) = *None*
proof(*induct* (*Throw a* :: *′addr expr1*, *xs*) (*stk*, *loc*, *pc*, ⌊*a* :: *′addr*⌋)
and (*map Val vs* @ *Throw a* # *es′* :: *′addr expr1 list*, *xs*) (*stk*, *loc*, *pc*, ⌊*a* :: *′addr*⌋)
arbitrary: *xs stk loc pc pc′ d* **and** *xs stk loc pc vs es′ pc′ d rule*: *bisim1-bisims1.inducts*)
case *bisim1Sync10*
thus *?case* **by**(*simp add*: *matches-ex-entry-def*)
next
case *bisim1Sync11*
thus *?case* **by**(*simp add*: *matches-ex-entry-def*)
next
case (*bisim1SyncThrow e1 xs stk loc pc e2*)
note *IH* = ‹*match-ex-table* (*compP f P*) (*cname-of h a*) (*pc′* + *pc*) (*compxE2 e1 pc′ d*) = *None*›
from ‹*P*,*e1*,*h* ⊢ (*Throw a*,*xs*) [↔] (*stk*,*loc*,*pc*,⌊*a*⌋)› **have** *pc* < *length* (*compE2 e1*) **by**(*auto dest*: *bisim1-ThrowD*)
with *IH* **show** *?case*
by(*auto simp add*: *match-ex-table-append matches-ex-entry-def dest*: *match-ex-table-pc-length-compE2 intro*: *match-ex-table-not-pcs-None*)
next
case *bisims1List1* **thus** *?case*
by(*auto simp add*: *Cons-eq-append-conv dest*: *bisim1-ThrowD bisim-Val-loc-eq-xcp-None*)
next
case (*bisims1List2 es es′′ xs stk loc pc e v*)
hence ⋀*pc′ d*. *match-ex-table* (*compP f P*) (*cname-of h a*) (*pc′* + *pc*) (*compxEs2 es pc′ d*) = *None*
by(*auto simp add*: *Cons-eq-append-conv*)
from *this*[*of pc′* + *length* (*compE2 e*) *Suc d*] **show** *?case* **by**(*auto simp add*: *add.assoc*)
next
case (*bisim1BlockThrowSome e xs stk loc pc T v*)
hence ⋀*pc′*. *match-ex-table* (*compP f P*) (*cname-of h a*) (*pc′* + *pc*) (*compxE2 e pc′ d*) = *None* **by** *auto*
from *this*[*of* 2+*pc′*] **show** *?case* **by**(*auto*)
next

case (*bisim1Seq2 e2 stk loc pc e1 xs*)
hence $\bigwedge pc'$. *match-ex-table* (*compP f P*) (*cname-of h a*) (*pc'* + *pc*) (*compxE2 e2 pc' d*) = *None* **by** *auto*
from *this*[*of Suc* (*pc'* + *length* (*compE2 e1*))] **show** *?case* **by**(*simp add*: *add.assoc*)
next
case (*bisim1CondThen e1 stk loc pc e e2 xs*)
hence $\bigwedge pc'$. *match-ex-table* (*compP f P*) (*cname-of h a*) (*pc'* + *pc*) (*compxE2 e1 pc' d*) = *None* **by** *auto*
note *this*[*of Suc* (*pc'* + *length* (*compE2 e*))]
moreover from ‹*P*,*e1*,*h* ⊢ (*Throw a*,*xs*) ↔ (*stk*,*loc*,*pc*,⌊*a*⌋)›
have *pc* < *length* (*compE2 e1*) **by**(*auto dest*: *bisim1-ThrowD*)
ultimately show *?case* **by**(*simp add*: *add.assoc match-ex-table-eq-NoneI outside-pcs-compxE2-not-matches-entry*
next
case (*bisim1CondElse e2 stk loc pc e e1 xs*)
hence $\bigwedge pc'$. *match-ex-table* (*compP f P*) (*cname-of h a*) (*pc'* + *pc*) (*compxE2 e2 pc' d*) = *None* **by** *auto*
note *this*[*of Suc* (*Suc* (*pc'* + (*length* (*compE2 e*) + *length* (*compE2 e1*))))]
thus *?case* **by**(*simp add*: *add.assoc*)
next
case (*bisim1WhileThrow2 e xs stk loc pc c*)
hence $\bigwedge pc'$. *match-ex-table* (*compP f P*) (*cname-of h a*) (*pc'* + *pc*) (*compxE2 e pc' d*) = *None* **by** *auto*
from *this*[*of Suc* (*pc'* + (*length* (*compE2 c*)))]
show *?case* **by**(*simp add*: *add.assoc*)
next
case (*bisim1Throw1 e xs stk loc pc*)
thus *?case* **by**(*auto dest*: *bisim-Val-loc-eq-xcp-None*)
next
case (*bisim1TryFail e xs stk loc pc C' C e2*)
hence *match-ex-table* (*compP f P*) (*cname-of h a*) (*pc'* + *pc*) (*compxE2 e pc' d*) = *None* **by** *auto*
moreover from ‹*P*,*e*,*h* ⊢ (*Throw a*,*xs*) ↔ (*stk*,*loc*,*pc*,⌊*a*⌋)› **have** *pc* < *length* (*compE2 e*)
by(*auto dest*: *bisim1-ThrowD*)
ultimately show *?case* **using** ‹*typeof-addr h a* = ⌊*Class-type C'*⌋› ‹¬ *P* ⊢ *C'* ⪯* *C*›
by(*simp add*: *matches-ex-entry-def cname-of-def*)
next
case (*bisim1TryCatchThrow e2 xs stk loc pc e C*)
hence $\bigwedge pc'$. *match-ex-table* (*compP f P*) (*cname-of h a*) (*pc'* + *pc*) (*compxE2 e2 pc' d*) = *None* **by** *auto*
from *this*[*of Suc* (*Suc* (*pc'* + (*length* (*compE2 e*))))]
show *?case* **by**(*simp add*: *add.assoc matches-ex-entry-def*)
next
case *bisim1Sync12* **thus** *?case*
by(*auto dest*: *bisim1-ThrowD simp add*: *match-ex-table-append eval-nat-numeral*, *simp add*: *matches-ex-entry-def*
next
case *bisim1Sync14* **thus** *?case*
by(*auto dest*: *bisim1-ThrowD simp add*: *match-ex-table-append eval-nat-numeral*, *simp add*: *matches-ex-entry-def*
qed(*fastforce dest*: *bisim1-ThrowD simp add*: *add.assoc*[*symmetric*])+

declare *match-ex-table-append-not-pcs* [*simp del*]
match-ex-table-eq-NoneI[*simp del*]
outside-pcs-compxE2-not-matches-entry[*simp del*]
outside-pcs-compxEs2-not-matches-entry[*simp del*]

lemma *bisim1-xcp-pcD*: *P*,*e*,*h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋) ⟹ *pc* < *length* (*compE2 e*)

and *bisims1-xcp-pcD*: *P*,*es*,*h* ⊢ (*es′*, *xs*) [↔] (*stk*, *loc*, *pc*, ⌊*a*⌋) ⟹ *pc* < *length* (*compEs2 es*)
by(*induct* (*e′*, *xs*) (*stk*, *loc*, *pc*, ⌊*a* :: *′addr*⌋) **and** (*es′*, *xs*) (*stk*, *loc*, *pc*, ⌊*a* :: *′addr*⌋)
arbitrary: *e′ xs stk loc pc* **and** *es′ xs stk loc pc rule*: *bisim1-bisims1.inducts*)
auto

declare *nth-append* [*simp*]

lemma *bisim1-Val-τExec-move*:
⟦ *P*, *E*, *h* ⊢ (*Val v*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*); *pc* < *length* (*compE2 E*) ⟧
⟹ *xs* = *loc* ∧ *xcp* = *None* ∧
τExec-mover-a P t E h (*stk*, *xs*, *pc*, *None*) ([*v*], *xs*, *length* (*compE2 E*), *None*)

and *bisims1-Val-τExec-moves*:
⟦ *P*, *Es*, *h* ⊢ (*map Val vs*, *xs*) [↔] (*stk*, *loc*, *pc*, *xcp*); *pc* < *length* (*compEs2 Es*) ⟧
⟹ *xs* = *loc* ∧ *xcp* = *None* ∧
τExec-movesr-a P t Es h (*stk*, *xs*, *pc*, *None*) (*rev vs*, *xs*, *length* (*compEs2 Es*), *None*)
proof(*induct* (*Val v* :: *′addr expr1*, *xs*) (*stk*, *loc*, *pc*, *xcp*)
and (*map Val vs* :: *′addr expr1 list*, *xs*) (*stk*, *loc*, *pc*, *xcp*)
arbitrary: *v xs stk loc pc xcp* **and** *vs xs stk loc pc xcp rule*: *bisim1-bisims1.inducts*)
case *bisim1Val* **thus** *?case* **by**(*auto intro*!: *τExecr1step exec-instr τmove2Val simp add*: *exec-move-def*)
next
case (*bisim1LAss2 V e xs*)
have *τExec-mover-a P t* (*V*:=*e*) *h* ([], *xs*, *Suc* (*length* (*compE2 e*)), *None*) ([*Unit*], *xs*, *Suc* (*Suc* (*length* (*compE2 e*))), *None*)
by(*auto intro*!: *τExecr1step exec-instr τmove2LAssRed2 simp add*: *exec-move-def*)
with *bisim1LAss2* **show** *?case* **by** *simp*
next
case (*bisim1AAss4 a i e xs*)
have *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* ([], *xs*, *Suc* (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)), *None*) ([*Unit*], *xs*, *Suc* (*Suc* (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*))), *None*)
by(*auto intro*!: *τExecr1step exec-instr τmove2AAssRed simp add*: *exec-move-def*)
with *bisim1AAss4* **show** *?case* **by**(*simp add*: *ac-simps*)
next
case (*bisim1FAss3 e F D e2 xs*)
have *τExec-mover-a P t* (*e*•*F*{*D*} := *e2*) *h* ([], *xs*, *Suc* (*length* (*compE2 e*) + *length* (*compE2 e2*)), *None*) ([*Unit*], *xs*, *Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e2*))), *None*)
by(*auto intro*!: *τExecr1step exec-instr τmove2FAssRed simp add*: *exec-move-def*)
with *bisim1FAss3* **show** *?case* **by** *simp*
next
case (*bisim1Sync6 V e1 e2 v xs*)
have *τExec-mover-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*v*], *xs*, *5* + *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
([*v*], *xs*, *9* + *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
by(*rule τExecr1step*)(*auto intro*: *exec-instr τmove2Sync6 simp add*: *exec-move-def*)
with *bisim1Sync6* **show** *?case* **by**(*auto simp add*: *eval-nat-numeral*)
next
case (*bisim1Seq2 e2 stk loc pc xcp e1 v xs*)
from ‹*Suc* (*length* (*compE2 e1*) + *pc*) < *length* (*compE2* (*e1*;; *e2*))› **have** *pc*: *pc* < *length* (*compE2 e2*) **by** *simp*
with ‹*pc* < *length* (*compE2 e2*) ⟹ *xs* = *loc* ∧ *xcp* = *None* ∧ *τExec-mover-a P t e2 h* (*stk*, *xs*, *pc*, *None*) ([*v*], *xs*, *length* (*compE2 e2*), *None*)›
have *xs* = *loc xcp* = *None*
τExec-mover-a P t e2 h (*stk*, *xs*, *pc*, *None*) ([*v*], *xs*, *length* (*compE2 e2*), *None*) **by** *auto*

moreover
hence *τExec-mover-a P t (e1;;e2) h (stk, xs, Suc (length (compE2 e1) + pc), None) ([v], xs, Suc (length (compE2 e1) + length (compE2 e2)), None)*
by −(*rule Seq-τExecrI2*)
ultimately show *?case* **by**(*simp*)
next
case (*bisim1CondThen e1 stk loc pc xcp e e2 v xs*)
from ‹*P, e1, h* ⊢ (*Val v,xs*) ↔ (*stk,loc,pc,xcp*)›
have *pc* ≤ *length* (*compE2 e1*) **by**(*rule bisim1-pc-length-compE2*)

have *e*: *τExec-mover-a P t* (*if* (*e*) *e1 else e2*) *h*
([*v*], *xs*, *Suc* (*length* (*compE2 e*) + (*length* (*compE2 e1*))), *None*)
([*v*], *xs*, *length* (*compE2* (*if* (*e*) *e1 else e2*)), *None*)
by(*rule τExecr1step*)(*auto simp add*: *exec-move-def intro*!: *exec-instr τmove2CondThenExit*)
show *?case*
proof(*cases pc* < *length* (*compE2 e1*))
case *True*
with ‹*pc* < *length* (*compE2 e1*)
⟹ *xs* = *loc* ∧ *xcp* = *None* ∧ *τExec-mover-a P t e1 h* (*stk*, *xs*, *pc*, *None*) ([*v*], *xs*, *length* (*compE2 e1*), *None*)›
have *s*: *xs* = *loc xcp* = *None*
and *τExec-mover-a P t e1 h* (*stk*, *xs*, *pc*, *None*) ([*v*], *xs*, *length* (*compE2 e1*), *None*) **by** *auto*
hence *τExec-mover-a P t* (*if* (*e*) *e1 else e2*) *h*
(*stk*, *xs*, *Suc* (*length* (*compE2 e*) + *pc*), *None*)
([*v*], *xs*, *Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*)), *None*)
by −(*rule Cond-τExecrI2*)
with *e True s* **show** *?thesis* **by**(*simp*)
next
case *False*
with ‹*pc* ≤ *length* (*compE2 e1*)› **have** *pc*: *pc* = *length* (*compE2 e1*) **by** *auto*
with ‹*P, e1, h* ⊢ (*Val v,xs*) ↔ (*stk,loc,pc,xcp*)›
have *stk* = [*v*] *xs* = *loc xcp* = *None* **by**(*auto dest*: *bisim1-Val-length-compE2D*)
with *pc e* **show** *?thesis* **by**(*simp*)
qed
next
case (*bisim1CondElse e2 stk loc pc xcp e e1 v xs*)
from ‹*P, e2, h* ⊢ (*Val v,xs*) ↔ (*stk,loc,pc,xcp*)›
have *pc* ≤ *length* (*compE2 e2*) **by**(*rule bisim1-pc-length-compE2*)

show *?case*
proof(*cases pc* < *length* (*compE2 e2*))
case *True*
with ‹*pc* < *length* (*compE2 e2*)
⟹ *xs* = *loc* ∧ *xcp* = *None* ∧ *τExec-mover-a P t e2 h* (*stk*, *xs*, *pc*, *None*) ([*v*], *xs*, *length* (*compE2 e2*), *None*)›
have *s*: *xs* = *loc xcp* = *None*
and *e*: *τExec-mover-a P t e2 h* (*stk*, *xs*, *pc*, *None*) ([*v*], *xs*, *length* (*compE2 e2*), *None*) **by** *auto*
from *e* **have** *τExec-mover-a P t* (*if* (*e*) *e1 else e2*) *h* (*stk*, *xs*, *Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + *pc*)), *None*) ([*v*], *xs*, *Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + *length* (*compE2 e2*))), *None*)
by(*rule Cond-τExecrI3*)
with *True s* **show** *?thesis* **by**(*simp add*: *add.assoc*)
next
case *False*

```
    with ‹pc ≤ length (compE2 e2)› have pc: pc = length (compE2 e2) by auto
    with ‹P, e2, h ⊢ (Val v,xs) ↔ (stk,loc,pc,xcp)›
    have stk = [v] xs = loc xcp = None by(auto dest: bisim1-Val-length-compE2D)
    with pc show ?thesis by(simp add: add.assoc)
  qed
next
  case (bisim1While7 c e xs)
  have τExec-mover-a P t (while (c) e) h
                 ([], xs, Suc (Suc (Suc (length (compE2 c) + length (compE2 e)))), None)
                 ([Unit], xs, length (compE2 (while (c) e)), None)
    by(auto intro!: τExecr1step exec-instr τmove2While4 simp add: exec-move-def)
  thus ?case by(simp)
next
  case (bisims1List1 e e' xs stk loc pc xcp es)
  from ‹e' # es = map Val vs› obtain v vs' where [simp]: vs = v # vs' e' = Val v es = map Val vs'
by auto
  from ‹P,e,h ⊢ (e',xs) ↔ (stk,loc,pc,xcp)›
  have length: pc ≤ length (compE2 e) by(auto dest: bisim1-pc-length-compE2)
  hence xs = loc ∧ xcp = None ∧ τExec-mover-a P t e h (stk, xs, pc, None) ([v], xs, length (compE2
e), None)
  proof(cases pc < length (compE2 e))
    case True
    with ‹⟦e' = Val v; pc < length (compE2 e)⟧ ⟹ xs = loc ∧ xcp = None ∧ τExec-mover-a P t e h
(stk, xs, pc, None) ([v], xs, length (compE2 e), None)›
    show ?thesis by auto
  next
    case False
    with length have pc: pc = length (compE2 e) by auto
    with ‹P,e,h ⊢ (e',xs) ↔ (stk,loc,pc,xcp)› have stk = [v] xs = loc xcp = None
      by(auto dest: bisim1-Val-length-compE2D)
    with pc show ?thesis by(auto)
  qed
  hence s: xs = loc xcp = None
    and exec1: τExec-mover-a P t e h (stk, xs, pc, None) ([v], xs, length (compE2 e), None) by auto
  from exec1 have τExec-movesr-a P t (e # es) h (stk, xs, pc, None) ([v], xs, length (compE2 e),
None)
    by(auto intro: τExec-mover-τExec-movesr)
  moreover have τExec-movesr-a P t (map Val vs') h ([], xs, 0, None) (rev vs', xs, length (compEs2
(map Val vs')), None)
    by(rule τExec-movesr-map-Val)
  hence τExec-movesr-a P t ([e] @ map Val vs') h ([] @ [v], xs, length (compEs2 [e]) + 0, None) (rev
vs' @ [v], xs, length (compEs2 [e]) + length (compEs2 (map Val vs')), None)
    by −(rule append-τExec-movesr, auto)
  ultimately show ?case using s by(auto)
next
  case (bisims1List2 es es' xs stk loc pc xcp e v)
  from ‹Val v # es' = map Val vs› obtain vs' where [simp]: vs = v # vs' es' = map Val vs' by auto
  from ‹P,es,h ⊢ (es',xs) [↔] (stk,loc,pc,xcp)›
  have length: pc ≤ length (compEs2 es) by(auto dest: bisims1-pc-length-compEs2)
  hence xs = loc ∧ xcp = None ∧ τExec-movesr-a P t es h (stk, xs, pc, None) (rev vs', xs, length
(compEs2 es), None)
  proof(cases pc < length (compEs2 es))
    case True
    with ‹⟦es' = map Val vs'; pc < length (compEs2 es)⟧ ⟹ xs = loc ∧ xcp = None ∧ τExec-movesr-a
```

```
P t es h (stk, xs, pc, None)
      (rev vs', xs, length (compEs2 es), None)›
    show ?thesis by auto
  next
    case False
    with length have pc: pc = length (compEs2 es) by auto
    with ‹P,es,h ⊢ (es',xs) [↔] (stk,loc,pc,xcp)› have stk = rev vs' xs = loc xcp = None
      by(auto dest: bisims1-Val-length-compEs2D)
    with pc show ?thesis by(auto)
  qed
  hence s: xs = loc xcp = None
    and exec1: τExec-movesr-a P t es h (stk, xs, pc, None) (rev vs', xs, length (compEs2 es), None)
by auto
  from exec1 have τExec-movesr-a P t ([e] @ es) h (stk @ [v], xs, length (compEs2 [e]) + pc, None)
(rev vs' @ [v], xs, length (compEs2 [e]) + length (compEs2 es), None)
    by −(rule append-τExec-movesr, auto)
  thus ?case using s by(auto)
qed(auto)

lemma bisim1Val2D1:
  assumes bisim: P, e, h ⊢ (Val v,xs) ↔ (stk,loc,pc,xcp)
  shows xcp = None ∧ xs = loc ∧ τExec-mover-a P t e h (stk, loc, pc, xcp) ([v], loc, length (compE2
e), None)
proof −
  from bisim have xcp = None xs = loc by(auto dest: bisim-Val-loc-eq-xcp-None)
  moreover
  have τExec-mover-a P t e h (stk, loc, pc, xcp) ([v], loc, length (compE2 e), None)
  proof(cases pc < length (compE2 e))
    case True
    from bisim1-Val-τExec-move[OF bisim True] show ?thesis by auto
  next
    case False
    from bisim have pc ≤ length (compE2 e) by(auto dest: bisim1-pc-length-compE2)
    with False have pc = length (compE2 e) by auto
    with bisim have stk = [v] loc = xs xcp=None by(auto dest: bisim1-Val-length-compE2D)
    with ‹pc = length (compE2 e)› show ?thesis by(auto)
  qed
  ultimately show ?thesis by simp
qed

lemma bisim1-Throw-τExec-movet:
  ⟦ P, e, h ⊢ (Throw a,xs) ↔ (stk,loc,pc,None) ⟧
  ⟹ ∃pc'. τExec-movet-a P t e h (stk, loc, pc, None) ([Addr a], loc, pc', ⌊a⌋) ∧
      P, e, h ⊢ (Throw a,xs) ↔ ([Addr a], loc, pc', ⌊a⌋) ∧ xs = loc

  and bisims1-Throw-τExec-movest:
  ⟦ P, es, h ⊢  (map Val vs @ Throw a # es',xs) [↔] (stk,loc,pc,None) ⟧
  ⟹ ∃pc'. τExec-movest-a P t es h (stk, loc, pc, None) (Addr a # rev vs, loc, pc', ⌊a⌋) ∧
      P, es, h ⊢ (map Val vs @ Throw a # es',xs) [↔] (Addr a # rev vs, loc, pc', ⌊a⌋) ∧ xs = loc
proof(induct e n :: nat Throw a :: 'addr expr1 xs stk loc pc None :: 'addr option
    and es n :: nat map Val vs @ Throw a # es' :: 'addr expr1 list xs stk loc pc None :: 'addr option
    arbitrary: and vs rule: bisim1-bisims1-inducts-split)
  case (bisim1Sync9 e1 n e2 V xs)
  let ?pc = 8 + length (compE2 e1) + length (compE2 e2)
```

have *τExec-movet-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr a*], *xs*, *?pc*, *None*) ([*Addr a*], *xs*, *?pc*, ⌊*a*⌋)
by(*rule τExect1step*)(*auto intro*: *exec-instr τmove2-τmoves2.intros simp add*: *is-Ref-def exec-move-def*)
moreover
have *P*,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*Throw a*,*xs*) ↔ ([*Addr a*],*xs*,*?pc*,⌊*a*⌋) **by**(*rule bisim1Sync10*)
ultimately show *?case* **by** *auto*
next
case (*bisim1Seq2 e2 n xs stk loc pc e1*)
then obtain *pc′* **where** *τExec-movet-a P t e2 h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
P, *e2*, *h* ⊢ (*Throw a*,*xs*) ↔ ([*Addr a*],*loc*,*pc′*,⌊*a*⌋) *xs* = *loc* **by** *auto*
thus *?case* **by**(*auto intro*: *Seq-τExectI2 bisim1-bisims1.bisim1Seq2*)
next
case (*bisim1CondThen e1 n xs stk loc pc e e2*)
then obtain *pc′* **where** *exec*: *τExec-movet-a P t e1 h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
and *bisim*: *P*, *e1*, *h* ⊢ (*Throw a*,*xs*) ↔ ([*Addr a*],*loc*,*pc′*,⌊*a*⌋) **and** *s*: *xs* = *loc* **by** *auto*
from *exec* **have** *τExec-movet-a P t* (*if* (*e*) *e1 else e2*) *h* (*stk*, *loc*, *Suc* (*length* (*compE2 e*) + *pc*), *None*) ([*Addr a*], *loc*, *Suc* (*length* (*compE2 e*) + *pc′*), ⌊*a*⌋)
by(*rule Cond-τExectI2*)
moreover from *bisim*
have *P*, *if* (*e*) *e1 else e2*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *Suc* (*length* (*compE2 e*) + *pc′*), ⌊*a*⌋)
by(*rule bisim1-bisims1.bisim1CondThen*)
ultimately show *?case* **using** *s* **by**(*auto*)
next
case (*bisim1CondElse e2 n xs stk loc pc e e1*)
then obtain *pc′* **where** *exec*: *τExec-movet-a P t e2 h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
and *bisim*: *P*, *e2*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋) **and** *s*: *xs* = *loc* **by** *auto*
let *?pc pc* = *Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + *pc*))
from *exec* **have** *τExec-movet-a P t* (*if* (*e*) *e1 else e2*) *h* (*stk*, *loc*, (*?pc pc*), *None*) ([*Addr a*], *loc*, *?pc pc′*, ⌊*a*⌋)
by(*rule Cond-τExectI3*)
moreover from *bisim*
have *P*, *if* (*e*) *e1 else e2*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *?pc pc′*, ⌊*a*⌋)
by(*rule bisim1-bisims1.bisim1CondElse*)
ultimately show *?case* **using** *s* **by** *auto*
next
case (*bisim1Throw1 e n xs stk loc pc*)
note *bisim* = ‹*P*, *e*, *h* ⊢ (*addr a*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
hence *s*: *xs* = *loc*
and *exec*: *τExec-mover-a P t e h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *length* (*compE2 e*), *None*)
by(*auto dest*: *bisim1Val2D1*)
from *exec* **have** *τExec-mover-a P t* (*throw e*) *h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *length* (*compE2 e*), *None*)
by(*rule Throw-τExecrI*)
also have *τExec-movet-a P t* (*throw e*) *h* ([*Addr a*], *loc*, *length* (*compE2 e*), *None*) ([*Addr a*], *loc*, *length* (*compE2 e*), ⌊*a*⌋)
by(*rule τExect1step*, *auto intro*: *exec-instr τmove2Throw2 simp add*: *is-Ref-def exec-move-def*)
also have *P*, *throw e*, *h* ⊢ (*Throw a*, *loc*) ↔ ([*Addr a*], *loc*, *length* (*compE2 e*), ⌊*a*⌋)
by(*rule bisim1Throw2*)
ultimately show *?case* **using** *s* **by** *auto*
next
case (*bisims1List1 e n e′ xs stk loc pc es vs*)
note *bisim* = ‹*P*,*e*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
show *?case*
proof(*cases is-val e′*)
case *True*

with ‹$e' \# es = map\ Val\ vs\ @\ Throw\ a \# es'$› **obtain** $v\ vs'$ **where** $vs = v \# vs'\ e' = Val\ v$
and *es*: $es = map\ Val\ vs' @ Throw\ a \# es'$ **by**(*auto simp add*: *Cons-eq-append-conv*)
with *bisim* **have** $P,e,h \vdash (Val\ v, xs) \leftrightarrow (stk, loc, pc, None)$ **by** *simp*
from *bisim1Val2D1*[*OF this*] **have** [*simp*]: $xs = loc$
and *exec*: $\tau Exec\text{-}mover\text{-}a\ P\ t\ e\ h\ (stk, loc, pc, None)\ ([v], loc, length\ (compE2\ e), None)$
by *auto*
from *exec* **have** $\tau Exec\text{-}movesr\text{-}a\ P\ t\ (e \# es)\ h\ (stk, loc, pc, None)\ ([v], loc, length\ (compE2\ e), None)$
by(*rule* $\tau Exec\text{-}mover\text{-}\tau Exec\text{-}movesr$)
also from *es* ‹$es = map\ Val\ vs' @ Throw\ a \# es' \Longrightarrow \exists pc'.\ \tau Exec\text{-}movest\text{-}a\ P\ t\ es\ h\ ([], loc, 0, None)\ (Addr\ a \# rev\ vs', loc, pc', \lfloor a \rfloor) \wedge P,es,h \vdash (map\ Val\ vs' @ Throw\ a \# es', loc)\ [\leftrightarrow]\ (Addr\ a \# rev\ vs', loc, pc', \lfloor a \rfloor) \wedge loc = loc$›
obtain pc' **where** *execes*: $\tau Exec\text{-}movest\text{-}a\ P\ t\ es\ h\ ([], loc, 0, None)\ (Addr\ a \# rev\ vs', loc, pc', \lfloor a \rfloor)$
and *bisim'*: $P,es,h \vdash (map\ Val\ vs' @ Throw\ a \# es', loc)\ [\leftrightarrow]\ (Addr\ a \# rev\ vs', loc, pc', \lfloor a \rfloor)$ **by** *auto*
from $append\text{-}\tau Exec\text{-}movest$[*OF - execes*, *of* [*v*] [*e*]]
have $\tau Exec\text{-}movest\text{-}a\ P\ t\ (e \# es)\ h\ ([v], loc, length\ (compE2\ e), None)\ (Addr\ a \# rev\ vs' @ [v], loc, length\ (compE2\ e) + pc', \lfloor a \rfloor)$ **by** *simp*
also from *bisims1List2*[*OF bisim'*, *of e v*] *es* ‹$e' = Val\ v$› ‹$vs = v \# vs'$›
have $P,e \# es,h \vdash (e' \# es, xs)\ [\leftrightarrow]\ ((Addr\ a \# rev\ vs), loc, length\ (compE2\ e) + pc', \lfloor a \rfloor)$ **by** *simp*
ultimately show *?thesis* **using** ‹$vs = v \# vs'$› *es* ‹$e' = Val\ v$› **by** *auto*
next
case *False*
with ‹$e' \# es = map\ Val\ vs @ Throw\ a \# es'$› **have** [*simp*]: $e' = Throw\ a\ es = es'\ vs = []$
by(*auto simp add*: *Cons-eq-append-conv*)
from ‹$e' = Throw\ a \Longrightarrow \exists pc'.\ \tau Exec\text{-}movet\text{-}a\ P\ t\ e\ h\ (stk, loc, pc, None)\ ([Addr\ a], loc, pc', \lfloor a \rfloor) \wedge P,e,h \vdash (Throw\ a, xs) \leftrightarrow ([Addr\ a], loc, pc', \lfloor a \rfloor) \wedge xs = loc$›
obtain pc' **where** $\tau Exec\text{-}movet\text{-}a\ P\ t\ e\ h\ (stk, loc, pc, None)\ ([Addr\ a], loc, pc', \lfloor a \rfloor)$
and *bisim*: $P,e,h \vdash (Throw\ a, xs) \leftrightarrow ([Addr\ a], loc, pc', \lfloor a \rfloor)$ **and** *s*: $xs = loc$ **by** *auto*
hence $\tau Exec\text{-}movest\text{-}a\ P\ t\ (e \# es)\ h\ (stk, loc, pc, None)\ ([Addr\ a], loc, pc', \lfloor a \rfloor)$
by−(*rule* $\tau Exec\text{-}movet\text{-}\tau Exec\text{-}movest$)
moreover from *bisim*
have $P,e\#es,h \vdash (Throw\ a\#es,xs)\ [\leftrightarrow]\ ([Addr\ a],loc,pc',\lfloor a \rfloor)$ **by**(*rule bisim1-bisims1.bisims1List1*)
ultimately show *?thesis* **using** *s* **by** *auto*
qed
next
case (*bisims1List2 es n es'' xs stk loc pc e v*)
note *IH* = ‹$\bigwedge vs.\ es'' = map\ Val\ vs @ Throw\ a \# es' \Longrightarrow \exists pc'.\ \tau Exec\text{-}movest\text{-}a\ P\ t\ es\ h\ (stk, loc, pc, None)\ (Addr\ a \# rev\ vs, loc, pc', \lfloor a \rfloor) \wedge P,es,h \vdash (map\ Val\ vs @ Throw\ a \# es',xs)\ [\leftrightarrow]\ (Addr\ a \# rev\ vs,loc,pc',\lfloor a \rfloor) \wedge xs = loc$›
from ‹$Val\ v \# es'' = map\ Val\ vs @ Throw\ a \# es'$›
obtain vs' **where** [*simp*]: $vs = v \# vs'\ es'' = map\ Val\ vs' @ Throw\ a \# es'$ **by**(*auto simp add*: *Cons-eq-append-conv*)
from *IH*[*OF* ‹$es'' = map\ Val\ vs' @ Throw\ a \# es'$›]
obtain pc' **where** *exec*: $\tau Exec\text{-}movest\text{-}a\ P\ t\ es\ h\ (stk, loc, pc, None)\ (Addr\ a \# rev\ vs', loc, pc', \lfloor a \rfloor)$
and *bisim*: $P,es,h \vdash (map\ Val\ vs' @ Throw\ a \# es',xs)\ [\leftrightarrow]\ (Addr\ a \# rev\ vs',loc,pc',\lfloor a \rfloor)$
and [*simp*]: $xs = loc$ **by** *auto*
from $append\text{-}\tau Exec\text{-}movest$[*OF - exec*, *of* [*v*] [*e*]]
have $\tau Exec\text{-}movest\text{-}a\ P\ t\ (e \# es)\ h\ (stk @ [v], loc, length\ (compE2\ e) + pc, None)\ (Addr\ a \# rev\ vs, loc, length\ (compE2\ e) + pc', \lfloor a \rfloor)$ **by** *simp*
moreover from *bisim*
have $P,e\#es,h \vdash (Val\ v \# map\ Val\ vs' @ Throw\ a \# es',xs)\ [\leftrightarrow]\ ((Addr\ a \# rev\ vs')@[v],loc,length$

```
(compE2 e) + pc',⌊a⌋)
    by(rule bisim1-bisims1.bisims1List2)
  ultimately show ?case by(auto)
qed(auto)

lemma bisim1-Throw-τExec-mover:
  ⟦ P, e, h ⊢ (Throw a,xs) ↔ (stk,loc,pc,None) ⟧
  ⟹ ∃ pc'. τExec-mover-a P t e h (stk, loc, pc, None) ([Addr a], loc, pc', ⌊a⌋) ∧
      P, e, h ⊢ (Throw a,xs) ↔ ([Addr a], loc, pc', ⌊a⌋) ∧ xs = loc
by(drule bisim1-Throw-τExec-movet)(blast intro: tranclp-into-rtranclp)

lemma bisims1-Throw-τExec-movesr:
  ⟦ P, es, h ⊢  (map Val vs @ Throw a # es',xs) [↔] (stk,loc,pc,None) ⟧
  ⟹ ∃ pc'. τExec-movesr-a P t es h (stk, loc, pc, None) (Addr a # rev vs, loc, pc', ⌊a⌋) ∧
      P, es, h ⊢ (map Val vs @ Throw a # es',xs) [↔] (Addr a # rev vs, loc, pc', ⌊a⌋) ∧ xs = loc
by(drule bisims1-Throw-τExec-movest)(blast intro: tranclp-into-rtranclp)

declare split-beta [simp]

lemma bisim1-inline-call-Throw:
  ⟦ P,e,h ⊢ (e', xs) ↔ (stk, loc, pc, None); call1 e' = ⌊(a, M, vs)⌋;
    compE2 e ! pc = Invoke M n0; pc < length (compE2 e) ⟧
  ⟹ n0 = length vs ∧ P,e,h ⊢ (inline-call (Throw A) e', xs) ↔ (stk, loc, pc, ⌊A⌋)
  (is ⟦ -; -; -; - ⟧ ⟹ ?concl e n e' xs pc stk loc)

  and bisims1-inline-calls-Throw:
  ⟦ P,es,h ⊢ (es', xs) [↔] (stk, loc, pc, None); calls1 es' = ⌊(a, M, vs)⌋;
    compEs2 es ! pc = Invoke M n0; pc < length (compEs2 es) ⟧
  ⟹ n0 = length vs ∧ P,es,h ⊢ (inline-calls (Throw A) es', xs) [↔] (stk, loc, pc, ⌊A⌋)
  (is ⟦ -; -; -; - ⟧ ⟹ ?concls es n es' xs pc stk loc)
proof(induct e n :: nat e' xs stk loc pc None :: 'addr option
       and es n :: nat es' xs stk loc pc None :: 'addr option
      rule: bisim1-bisims1-inducts-split)
  case (bisim1BinOp1 e1 n e' xs stk loc pc e2 bop)
  note IH1 = ‹⟦call1 e' = ⌊(a, M, vs)⌋; compE2 e1 ! pc = Invoke M n0; pc < length (compE2 e1) ⟧
            ⟹ ?concl e1 n e' xs pc stk loc›
  note bisim1 = ‹P,e1,h ⊢ (e', xs) ↔ (stk, loc, pc, None)›
  note ins = ‹compE2 (e1 «bop» e2) ! pc = Invoke M n0›
  note call = ‹call1 (e' «bop» e2) = ⌊(a, M, vs)⌋›
  show ?case
  proof(cases is-val e')
    case False
    with bisim1 call have pc < length (compE2 e1)
      by(auto intro: bisim1-call-pcD)
    with call ins IH1 False show ?thesis
      by(auto intro: bisim1-bisims1.bisim1BinOp1)
  next
    case True
    then obtain v where [simp]: e' = Val v by auto
    from bisim1 have pc ≤ length (compE2 e1) by(auto dest: bisim1-pc-length-compE2)
    moreover {
      assume pc: pc < length (compE2 e1)
      with bisim1 ins have False
        by(auto dest: bisim-Val-pc-not-Invoke) }
```

```
    ultimately have [simp]: pc = length (compE2 e1) by(cases pc < length (compE2 e1)) auto
    from call ins show ?thesis by simp
  qed
next
  case bisim1BinOp2 thus ?case
    by(auto split: if-split-asm bop.split-asm dest: bisim1-bisims1.bisim1BinOp2)
next
  case (bisim1AAcc1 A n a' xs stk loc pc i)
  note IH1 = ‹⟦call1 a' = ⌊(a, M, vs)⌋; compE2 A ! pc = Invoke M n0; pc < length (compE2 A)⟧
            ⟹ ?concl A n a' xs pc stk loc›
  note bisim1 = ‹P,A,h ⊢ (a', xs) ↔ (stk, loc, pc, None)›
  note ins = ‹compE2 (A⌊i⌉) ! pc = Invoke M n0›
  note call = ‹call1 (a'⌊i⌉) = ⌊(a, M, vs)⌋›
  show ?case
  proof(cases is-val a')
    case False
    with bisim1 call have pc < length (compE2 A)
      by(auto intro: bisim1-call-pcD)
    with call ins IH1 False show ?thesis
      by(auto intro: bisim1-bisims1.bisim1AAcc1)
  next
    case True
    then obtain v where [simp]: a' = Val v by auto
    from bisim1 have pc ≤ length (compE2 A) by(auto dest: bisim1-pc-length-compE2)
    moreover {
      assume pc: pc < length (compE2 A)
      with bisim1 ins have False
        by(auto dest: bisim-Val-pc-not-Invoke) }
    ultimately have [simp]: pc = length (compE2 A) by(cases pc < length (compE2 A)) auto
    from call ins show ?thesis by simp
  qed
next
  case bisim1AAcc2 thus ?case
    by(auto split: if-split-asm dest: bisim1-bisims1.bisim1AAcc2)
next
  case (bisim1AAss1 A n a' xs stk loc pc i e)
  note IH1 = ‹⟦call1 a' = ⌊(a, M, vs)⌋; compE2 A ! pc = Invoke M n0; pc < length (compE2 A)⟧
            ⟹ ?concl A n a' xs pc stk loc›
  note bisim1 = ‹P,A,h ⊢ (a', xs) ↔ (stk, loc, pc, None)›
  note ins = ‹compE2 (A⌊i⌉ := e) ! pc = Invoke M n0›
  note call = ‹call1 (a'⌊i⌉ := e) = ⌊(a, M, vs)⌋›
  show ?case
  proof(cases is-val a')
    case False
    with bisim1 call have pc < length (compE2 A)
      by(auto intro: bisim1-call-pcD)
    with call ins IH1 False show ?thesis
      by(auto intro: bisim1-bisims1.bisim1AAss1)
  next
    case True
    then obtain v where [simp]: a' = Val v by auto
    from bisim1 have pc ≤ length (compE2 A) by(auto dest: bisim1-pc-length-compE2)
    moreover {
      assume pc: pc < length (compE2 A)
```

```
    with bisim1 ins have False
      by(auto dest: bisim-Val-pc-not-Invoke) }
   ultimately have [simp]: pc = length (compE2 A) by(cases pc < length (compE2 A)) auto
   from call ins show ?thesis by simp
 qed
next
 case (bisim1AAss2 i n i' xs stk loc pc A e v)
 note IH1 = ‹⟦call1 i' = ⌊(a, M, vs)⌋; compE2 i ! pc = Invoke M n0; pc < length (compE2 i)⟧
          ⟹ ?concl i n i' xs pc stk loc›
 note bisim1 = ‹P,i,h ⊢ (i', xs) ↔ (stk, loc, pc, None)›
 note ins = ‹compE2 (A⌊i⌉ := e) ! (length (compE2 A) + pc) = Invoke M n0›
 note call = ‹call1 (Val v⌊i'⌉ := e) = ⌊(a, M, vs)⌋›
 show ?case
 proof(cases is-val i')
   case False
   with bisim1 call have pc < length (compE2 i)
     by(auto intro: bisim1-call-pcD)
   with call ins IH1 False show ?thesis
     by(auto intro: bisim1-bisims1.bisim1AAss2)
 next
   case True
   then obtain v where [simp]: i' = Val v by auto
   from bisim1 have pc ≤ length (compE2 i) by(auto dest: bisim1-pc-length-compE2)
   moreover {
     assume pc: pc < length (compE2 i)
     with bisim1 ins have False
       by(auto dest: bisim-Val-pc-not-Invoke) }
   ultimately have [simp]: pc = length (compE2 i) by(cases pc < length (compE2 i)) auto
   from call ins show ?thesis by simp
 qed
next
 case bisim1AAss3 thus ?case
   by(auto split: if-split-asm nat.split-asm simp add: nth-Cons dest: bisim1-bisims1.bisim1AAss3)
next
 case (bisim1FAss1 e n e' xs stk loc pc e2 F D)
 note IH1 = ‹⟦call1 e' = ⌊(a, M, vs)⌋; compE2 e ! pc = Invoke M n0; pc < length (compE2 e)⟧
          ⟹ ?concl e n e' xs pc stk loc›
 note bisim1 = ‹P,e,h ⊢ (e', xs) ↔ (stk, loc, pc, None)›
 note ins = ‹compE2 (e•F{D} := e2) ! pc = Invoke M n0›
 note call = ‹call1 (e'•F{D} := e2) = ⌊(a, M, vs)⌋›
 show ?case
 proof(cases is-val e')
   case False
   with bisim1 call have pc < length (compE2 e)
     by(auto intro: bisim1-call-pcD)
   with call ins IH1 False show ?thesis
     by(auto intro: bisim1-bisims1.bisim1FAss1)
 next
   case True
   then obtain v where [simp]: e' = Val v by auto
   from bisim1 have pc ≤ length (compE2 e) by(auto dest: bisim1-pc-length-compE2)
   moreover {
     assume pc: pc < length (compE2 e)
     with bisim1 ins have False
```

```
      by(auto dest: bisim-Val-pc-not-Invoke) }
    ultimately have [simp]: pc = length (compE2 e) by(cases pc < length (compE2 e)) auto
    from call ins show ?thesis by simp
  qed
next
  case bisim1FAss2 thus ?case
    by(auto split: if-split-asm nat.split-asm simp add: nth-Cons dest: bisim1-bisims1.bisim1FAss2)
next
  case (bisim1CAS1 E n e' xs stk loc pc e2 e3 D F)
  note IH1 = ‹⟦call1 e' = ⌊(a, M, vs)⌋; compE2 E ! pc = Invoke M n0; pc < length (compE2 E)⟧
            ⟹ ?concl E n e' xs pc stk loc›
  note bisim1 = ‹P,E,h ⊢ (e', xs) ↔ (stk, loc, pc, None)›
  note ins = ‹compE2 - ! pc = Invoke M n0›
  note call = ‹call1 - = ⌊(a, M, vs)⌋›
  show ?case
  proof(cases is-val e')
    case False
    with bisim1 call have pc < length (compE2 E)
      by(auto intro: bisim1-call-pcD)
    with call ins IH1 False show ?thesis
      by(auto intro: bisim1-bisims1.bisim1CAS1)
  next
    case True
    then obtain v where [simp]: e' = Val v by auto
    from bisim1 have pc ≤ length (compE2 E) by(auto dest: bisim1-pc-length-compE2)
    moreover {
      assume pc: pc < length (compE2 E)
      with bisim1 ins have False
        by(auto dest: bisim-Val-pc-not-Invoke) }
    ultimately have [simp]: pc = length (compE2 E) by(cases pc < length (compE2 E)) auto
    from call ins show ?thesis by simp
  qed
next
  case (bisim1CAS2 e2 n e2' xs stk loc pc e1 e3 D F v)
  note IH1 = ‹⟦call1 e2' = ⌊(a, M, vs)⌋; compE2 e2 ! pc = Invoke M n0; pc < length (compE2 e2)⟧
            ⟹ ?concl e2 n e2' xs pc stk loc›
  note bisim1 = ‹P,e2,h ⊢ (e2', xs) ↔ (stk, loc, pc, None)›
  note ins = ‹compE2 - ! (length (compE2 e1) + pc) = Invoke M n0›
  note call = ‹call1 - = ⌊(a, M, vs)⌋›
  show ?case
  proof(cases is-val e2')
    case False
    with bisim1 call have pc < length (compE2 e2)
      by(auto intro: bisim1-call-pcD)
    with call ins IH1 False show ?thesis
      by(auto intro: bisim1-bisims1.bisim1CAS2)
  next
    case True
    then obtain v where [simp]: e2' = Val v by auto
    from bisim1 have pc ≤ length (compE2 e2) by(auto dest: bisim1-pc-length-compE2)
    moreover {
      assume pc: pc < length (compE2 e2)
      with bisim1 ins have False
        by(auto dest: bisim-Val-pc-not-Invoke) }
```

```
    ultimately have [simp]: pc = length (compE2 e2) by(cases pc < length (compE2 e2)) auto
    from call ins show ?thesis by simp
  qed
next
  case (bisim1Call1 obj n obj' xs stk loc pc ps M')
  note IH1 = ‹⟦call1 obj' = ⌊(a, M, vs)⌋; compE2 obj ! pc = Invoke M n0;
              pc < length (compE2 obj) ⟧
              ⟹ ?concl obj n obj' xs pc stk loc›
  note IH2 = ‹⋀xs. ⟦calls1 ps = ⌊(a, M, vs)⌋; compEs2 ps ! 0 = Invoke M n0; 0 < length (compEs2
ps) ⟧
            ⟹ ?concls ps n ps xs 0 [] xs›
  note ins = ‹compE2 (obj·M'(ps)) ! pc = Invoke M n0›
  note bisim1 = ‹P,obj,h ⊢ (obj', xs) ↔ (stk, loc, pc, None)›
  note call = ‹call1 (obj'·M'(ps)) = ⌊(a, M, vs)⌋›
  thus ?case
  proof(cases rule: call1-callE)
    case CallObj
    with bisim1 call have pc < length (compE2 obj) by(auto intro: bisim1-call-pcD)
    with call ins CallObj IH1 show ?thesis
      by(auto intro: bisim1-bisims1.bisim1Call1)
  next
    case (CallParams v)
    from bisim1 have pc ≤ length (compE2 obj) by(auto dest: bisim1-pc-length-compE2)
    moreover {
      assume pc: pc < length (compE2 obj)
      with bisim1 ins CallParams have False by(auto dest: bisim-Val-pc-not-Invoke) }
    ultimately have [simp]: pc = length (compE2 obj) by(cases pc < length (compE2 obj)) auto
   with bisim1 CallParams have [simp]: stk = [v] loc = xs by(auto dest: bisim1-Val-length-compE2D)
    from IH2[of loc] CallParams ins
    show ?thesis
      apply(clarsimp simp add: compEs2-map-Val is-vals-conv split: if-split-asm)
      apply(drule bisim1-bisims1.bisim1CallParams)
      apply(auto simp add: neq-Nil-conv)
      done
  next
    case [simp]: Call
    from bisim1 have pc ≤ length (compE2 obj) by(auto dest: bisim1-pc-length-compE2)
    moreover {
      assume pc: pc < length (compE2 obj)
      with bisim1 ins have False by(auto dest: bisim-Val-pc-not-Invoke) }
    ultimately have [simp]: pc = length (compE2 obj) by(cases pc < length (compE2 obj)) auto
    with ins have [simp]: vs = [] by(auto simp add: compEs2-map-Val split: if-split-asm)
    from bisim1 have [simp]: stk = [Addr a] xs = loc by(auto dest: bisim1-Val-length-compE2D)
    from ins show ?thesis by(auto intro: bisim1CallThrow[of [] [], simplified])
  qed
next
  case (bisim1CallParams ps n ps' xs stk loc pc obj M' v)
  note IH2 = ‹⟦calls1 ps' = ⌊(a, M, vs)⌋; compEs2 ps ! pc = Invoke M n0; pc < length (compEs2 ps)
⟧
            ⟹ ?concls ps n ps' xs pc stk loc›
  note ins = ‹compE2 (obj·M'(ps)) ! (length (compE2 obj) + pc) = Invoke M n0›
  note bisim2 = ‹P,ps,h ⊢ (ps', xs) [↔] (stk, loc, pc, None)›
  note call = ‹call1 (Val v·M'(ps')) = ⌊(a, M, vs)⌋›
  thus ?case
```

```
  proof(cases rule: call1-callE)
    case CallObj thus ?thesis by simp
  next
    case (CallParams v')
    hence [simp]: v' = v and call': calls1 ps' = ⌊(a, M, vs)⌋ by auto
    from bisim2 call' have pc < length (compEs2 ps) by(auto intro: bisims1-calls-pcD)
    with IH2 CallParams ins show ?thesis
      by(auto simp add: is-vals-conv split: if-split-asm intro: bisim1-bisims1.bisim1CallParams)
  next
    case Call
    hence [simp]: v = Addr a M' = M ps' = map Val vs by auto
    from bisim2 have pc ≤ length (compEs2 ps) by(auto dest: bisims1-pc-length-compEs2)
    moreover {
      assume pc: pc < length (compEs2 ps)
      with bisim2 ins have False by(auto dest: bisims-Val-pc-not-Invoke) }
    ultimately have [simp]: pc = length (compEs2 ps) by(cases pc < length (compEs2 ps)) auto
    from bisim2 have [simp]: stk = rev vs xs = loc by(auto dest: bisims1-Val-length-compEs2D)
    from bisim2 have length ps = length vs by(auto dest: bisims1-lengthD)
    with ins show ?thesis by(auto intro: bisim1CallThrow)
  qed
next
  case (bisims1List1 e n e' xs stk loc pc es)
  note IH1 = ‹⟦call1 e' = ⌊(a, M, vs)⌋; compE2 e ! pc = Invoke M n0; pc < length (compE2 e)⟧
          ⟹ ?concl e n e' xs pc stk loc›
  note IH2 = ‹⋀xs. ⟦calls1 es = ⌊(a, M, vs)⌋; compEs2 es ! 0 = Invoke M n0; 0 < length (compEs2
es)⟧
          ⟹ ?concls es n es xs 0 [] xs›
  note bisim1 = ‹P,e,h ⊢ (e', xs) ↔ (stk, loc, pc, None)›
  note call = ‹calls1 (e' # es) = ⌊(a, M, vs)⌋›
  note ins = ‹compEs2 (e # es) ! pc = Invoke M n0›
  show ?case
  proof(cases is-val e')
    case False
    with bisim1 call have pc < length (compE2 e) by(auto intro: bisim1-call-pcD)
    with call ins False IH1 show ?thesis
      by(auto intro: bisim1-bisims1.bisims1List1)
  next
    case True
    then obtain v where [simp]: e' = Val v by auto
    from bisim1 have pc ≤ length (compE2 e) by(auto dest: bisim1-pc-length-compE2)
    moreover {
      assume pc: pc < length (compE2 e)
      with bisim1 ins have False by(auto dest: bisim-Val-pc-not-Invoke) }
    ultimately have [simp]: pc = length (compE2 e) by(cases pc < length (compE2 e)) auto
    with bisim1 have [simp]: stk = [v] loc = xs by(auto dest: bisim1-Val-length-compE2D)
    from call have es ≠ [] by(cases es) simp-all
    with IH2[of loc] call ins
    show ?thesis by(auto split: if-split-asm dest: bisims1List2)
  qed
qed(auto split: if-split-asm bop.split-asm intro: bisim1-bisims1.intros dest: bisim1-pc-length-compE2)

lemma bisim1-max-stack: P,e,h ⊢ (e', xs) ↔ (stk, loc, pc, xcp) ⟹ length stk ≤ max-stack e
  and bisims1-max-stacks: P,es,h ⊢ (es', xs) [↔] (stk, loc, pc, xcp) ⟹ length stk ≤ max-stacks es
apply(induct (e', xs) (stk, loc, pc, xcp) and (es', xs) (stk, loc, pc, xcp)
```

arbitrary: *e' xs stk loc pc xcp* **and** *es' xs stk loc pc xcp rule*: *bisim1-bisims1.inducts*)
apply(*auto simp add*: *max-stack1*[*simplified*] *max-def max-stacks-ge-length*)
apply(*drule sym*, *simp add*: *max-stacks-ge-length*, *drule sym*, *simp*, *rule le-trans*[*OF max-stacks-ge-length*], *simp*)
done

inductive *bisim1-fr* :: *'addr J1-prog* ⇒ *'heap* ⇒ *'addr expr1* × *'addr locals1* ⇒ *'addr frame* ⇒ *bool*
for *P* :: *'addr J1-prog* **and** *h* :: *'heap*
where
⟦ *P* ⊢ *C sees M*:*Ts*→*T* = ⌊*body*⌋ *in D*;
P,*blocks1 0* (*Class D*#*Ts*) *body*, *h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*);
call1 e = ⌊(*a*, *M'*, *vs*)⌋;
max-vars e ≤ *length xs* ⟧
⟹ *bisim1-fr P h* (*e*, *xs*) (*stk*, *loc*, *C*, *M*, *pc*)

declare *bisim1-fr.intros* [*intro*]
declare *bisim1-fr.cases* [*elim*]

lemma *bisim1-fr-hext-mono*:
⟦ *bisim1-fr P h exs fr*; *hext h h'* ⟧ ⟹ *bisim1-fr P h' exs fr*
by(*auto intro*: *bisim1-hext-mono*)

lemma *bisim1-max-vars*: *P*,*E*,*h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*) ⟹ *max-vars E* ≥ *max-vars e*
and *bisims1-max-varss*: *P*,*Es*,*h* ⊢ (*es*,*xs*) [↔] (*stk*,*loc*,*pc*,*xcp*) ⟹ *max-varss Es* ≥ *max-varss es*
apply(*induct E* (*e*, *xs*) (*stk*, *loc*, *pc*, *xcp*) **and** *Es* (*es*, *xs*) (*stk*, *loc*, *pc*, *xcp*)
arbitrary: *e xs stk loc pc xcp* **and** *es xs stk loc pc xcp rule*: *bisim1-bisims1.inducts*)
apply(*auto*)
done

lemma *bisim1-call-τExec-move*:
⟦ *P*,*e*,*h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*); *call1 e'* = ⌊(*a*, *M'*, *vs*)⌋; *n* + *max-vars e'* ≤ *length xs*; ¬ *contains-insync e* ⟧
⟹ ∃ *pc' loc' stk'*. *τExec-mover-a P t e h* (*stk*, *loc*, *pc*, *None*) (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*) ∧
pc' < *length* (*compE2 e*) ∧ *compE2 e* ! *pc'* = *Invoke M'* (*length vs*) ∧
P,*e*,*h* ⊢ (*e'*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*)
(**is** ⟦ -; -; -; - ⟧ ⟹ *?concl e n e' xs pc stk loc*)

and *bisims1-calls-τExec-moves*:
⟦ *P*,*es*,*h* ⊢ (*es'*,*xs*) [↔] (*stk*, *loc*, *pc*, *None*); *calls1 es'* = ⌊(*a*, *M'*, *vs*)⌋;
n + *max-varss es'* ≤ *length xs*; ¬ *contains-insyncs es* ⟧
⟹ ∃ *pc' stk' loc'*. *τExec-movesr-a P t es h* (*stk*, *loc*, *pc*, *None*) (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*) ∧
pc' < *length* (*compEs2 es*) ∧ *compEs2 es* ! *pc'* = *Invoke M'* (*length vs*) ∧
P,*es*,*h* ⊢ (*es'*, *xs*) [↔] (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*)
(**is** ⟦-; -; -; - ⟧ ⟹ *?concls es n es' xs pc stk loc*)
proof(*induct e n* :: *nat e' xs stk loc pc xcp*≡*None* :: *'addr option*
and *es n* :: *nat es' xs stk loc pc xcp*≡*None* :: *'addr option*
rule: *bisim1-bisims1-inducts-split*)
case *bisim1Val2* **thus** *?case* **by** *auto*
next
case *bisim1New* **thus** *?case* **by** *auto*
next
case *bisim1NewArray* **thus** *?case*

by *auto* (*fastforce intro*: *bisim1-bisims1.bisim1NewArray elim*!: *NewArray-τExecrI intro*!: *exI*)
next
case *bisim1Cast* **thus** *?case*
by(*auto*)(*fastforce intro*: *bisim1-bisims1.bisim1Cast elim*!: *Cast-τExecrI intro*!: *exI*)+
next
case *bisim1InstanceOf* **thus** *?case*
by(*auto*)(*fastforce intro*: *bisim1-bisims1.bisim1InstanceOf elim*!: *InstanceOf-τExecrI intro*!: *exI*)+
next
case *bisim1Val* **thus** *?case* **by** *auto*
next
case *bisim1Var* **thus** *?case* **by** *auto*
next
case (*bisim1BinOp1 e1 n e′ xs stk loc pc e2 bop*)
note *IH1* = ‹⟦*call1 e′* = ⌊(*a*, *M′*, *vs*)⌋; *n* + *max-vars e′* ≤ *length xs*; ¬ *contains-insync e1* ⟧ ⟹ *?concl e1 n e′ xs pc stk loc*›
note *IH2* = ‹⋀*xs*. ⟦*call1 e2* = ⌊(*a*, *M′*, *vs*)⌋; *n* + *max-vars e2* ≤ *length xs*; ¬ *contains-insync e2* ⟧ ⟹ *?concl e2 n e2 xs 0* [] *xs*›
note *call* = ‹*call1* (*e′* «*bop*» *e2*) = ⌊(*a*, *M′*, *vs*)⌋›
note *len* = ‹*n* + *max-vars* (*e′* «*bop*» *e2*) ≤ *length xs*›
note *bisim1* = ‹*P*,*e1*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *cs* = ‹¬ *contains-insync* (*e1* «*bop*» *e2*)›
show *?case*
proof(*cases is-val e′*)
case *True*
then obtain *v* **where** [*simp*]: *e′* = *Val v* **by** *auto*
from *bisim1* **have** *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *None*) ([*v*], *loc*, *length* (*compE2 e1*), *None*)
and [*simp*]: *xs* = *loc* **by**(*auto dest*!: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*«*bop*»*e2*) *h* (*stk*, *loc*, *pc*, *None*) ([*v*], *loc*, *length* (*compE2 e1*), *None*)
by−(*rule BinOp-τExecrI1*)
also from *call IH2*[*of loc*] *len cs* **obtain** *pc′ stk′ loc′*
where *exec*: *τExec-mover-a P t e2 h* ([], *xs*, *0*, *None*) (*rev vs* @ *Addr a* # *stk′*, *loc′*, *pc′*, *None*)
and *ins*: *compE2 e2* ! *pc′* = *Invoke M′* (*length vs*) *pc′* < *length* (*compE2 e2*)
and *bisim′*: *P*,*e2*,*h* ⊢ (*e2*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk′*, *loc′*, *pc′*, *None*) **by** *auto*
from *BinOp-τExecrI2*[*OF exec*, *of e1 bop v*]
have *τExec-mover-a P t* (*e1*«*bop*»*e2*) *h* ([*v*], *loc*, *length* (*compE2 e1*), *None*) (*rev vs* @ *Addr a* # (*stk′* @ [*v*]), *loc′*, *length* (*compE2 e1*) + *pc′*, *None*) **by** *simp*
also (*rtranclp-trans*) **from** *bisim′*
have *P*,*e1*«*bop*»*e2*,*h* ⊢ (*Val v* «*bop*» *e2*, *xs*) ↔ ((*rev vs* @ *Addr a* # *stk′*) @ [*v*], *loc′*, *length* (*compE2 e1*) + *pc′*, *None*)
by(*rule bisim1BinOp2*)
ultimately show *?thesis* **using** *ins* **by** *fastforce*
next
case *False* **with** *IH1 len False call cs* **show** *?thesis*
by(*clarsimp*)(*fastforce intro*: *bisim1-bisims1.bisim1BinOp1 elim*!: *BinOp-τExecrI1 intro*!: *exI*)
qed
next
case (*bisim1BinOp2 e2 n e′ xs stk loc pc e1 bop v1*)
then obtain *pc′ loc′ stk′* **where** *pc′*: *pc′* < *length* (*compE2 e2*) *compE2 e2* ! *pc′* = *Invoke M′* (*length vs*)
and *exec*: *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *None*) (*rev vs* @ *Addr a* # *stk′*, *loc′*, *pc′*, *None*)
and *bisim′*: *P*,*e2*,*h* ⊢ (*e′*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk′*, *loc′*, *pc′*, *None*) **by** *fastforce*
from *exec* **have** *τExec-mover-a P t* (*e1* «*bop*» *e2*) *h* (*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *None*)
((*rev vs* @ *Addr a* # *stk′*) @ [*v1*], *loc′*, *length* (*compE2 e1*) + *pc′*,

None)
by(*rule BinOp-τExecrI2*)
moreover from *bisim′*
have *P*,*e1* «*bop*» *e2*,*h* ⊢ (*Val v1* «*bop*» *e′*, *xs*) ↔ ((*rev vs* @ *Addr a* # *stk′*) @ [*v1*], *loc′*, *length* (*compE2 e1*) + *pc′*, *None*)
by(*rule bisim1-bisims1.bisim1BinOp2*)
ultimately show *?case* **using** *pc′* **by**(*fastforce*)
next
case *bisim1LAss1* **thus** *?case*
by(*auto*)(*fastforce intro*: *bisim1-bisims1.bisim1LAss1 elim*!: *LAss-τExecrI intro*!: *exI*)
next
case *bisim1LAss2* **thus** *?case* **by** *simp*
next
case (*bisim1AAcc1 A n a′ xs stk loc pc i*)
note *IH1* = ‹⟦*call1 a′* = ⌊(*a*, *M′*, *vs*)⌋; *n* + *max-vars a′* ≤ *length xs*; ¬ *contains-insync A*⟧ ⟹ *?concl A n a′ xs pc stk loc*›
note *IH2* = ‹⋀*xs*. ⟦*call1 i* = ⌊(*a*, *M′*, *vs*)⌋; *n* + *max-vars i* ≤ *length xs*; ¬ *contains-insync i*⟧ ⟹ *?concl i n i xs 0* [] *xs*›
note *call* = ‹*call1* (*a′*⌊*i*⌉) = ⌊(*a*, *M′*, *vs*)⌋›
note *len* = ‹*n* + *max-vars* (*a′*⌊*i*⌉) ≤ *length xs*›
note *bisim1* = ‹*P*,*A*,*h* ⊢ (*a′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *cs* = ‹¬ *contains-insync* (*A*⌊*i*⌉)›
show *?case*
proof(*cases is-val a′*)
case *True*
then obtain *v* **where** [*simp*]: *a′* = *Val v* **by** *auto*
from *bisim1* **have** *τExec-mover-a P t A h* (*stk*, *loc*, *pc*, *None*) ([*v*], *loc*, *length* (*compE2 A*), *None*)
and [*simp*]: *xs* = *loc* **by**(*auto dest*!: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*A*⌊*i*⌉) *h* (*stk*, *loc*, *pc*, *None*) ([*v*], *loc*, *length* (*compE2 A*), *None*)
by−(*rule AAcc-τExecrI1*)
also from *call IH2*[*of loc*] *len cs* **obtain** *pc′ stk′ loc′*
where *exec*: *τExec-mover-a P t i h* ([], *xs*, *0*, *None*) (*rev vs* @ *Addr a* # *stk′*, *loc′*, *pc′*, *None*)
and *ins*: *compE2 i* ! *pc′* = *Invoke M′* (*length vs*) *pc′* < *length* (*compE2 i*)
and *bisim′*: *P*,*i*,*h* ⊢ (*i*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk′*, *loc′*, *pc′*, *None*) **by** *auto*
from *AAcc-τExecrI2*[*OF exec*, *of A v*]
have *τExec-mover-a P t* (*A*⌊*i*⌉) *h* ([*v*], *loc*, *length* (*compE2 A*), *None*) (*rev vs* @ *Addr a* # (*stk′* @ [*v*]), *loc′*, *length* (*compE2 A*) + *pc′*, *None*) **by** *simp*
also (*rtranclp-trans*) **from** *bisim′*
have *P*,*A*⌊*i*⌉,*h* ⊢ (*Val v*⌊*i*⌉, *xs*) ↔ ((*rev vs* @ *Addr a* # *stk′*) @ [*v*], *loc′*, *length* (*compE2 A*) + *pc′*, *None*)
by(*rule bisim1AAcc2*)
ultimately show *?thesis* **using** *ins* **by** *fastforce*
next
case *False* **with** *IH1 len False call cs* **show** *?thesis*
by(*clarsimp*)(*fastforce intro*: *bisim1-bisims1.bisim1AAcc1 elim*!: *AAcc-τExecrI1 intro*!: *exI*)
qed
next
case (*bisim1AAcc2 i n i′ xs stk loc pc A v*)
then obtain *pc′ loc′ stk′* **where** *pc′*: *pc′* < *length* (*compE2 i*) *compE2 i* ! *pc′* = *Invoke M′* (*length vs*)
and *exec*: *τExec-mover-a P t i h* (*stk*, *loc*, *pc*, *None*) (*rev vs* @ *Addr a* # *stk′*, *loc′*, *pc′*, *None*)
and *bisim′*: *P*,*i*,*h* ⊢ (*i′*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk′*, *loc′*, *pc′*, *None*) **by** *fastforce*
from *exec* **have** *τExec-mover-a P t* (*A*⌊*i*⌉) *h* (*stk* @ [*v*], *loc*, *length* (*compE2 A*) + *pc*, *None*)
((*rev vs* @ *Addr a* # *stk′*) @ [*v*], *loc′*, *length* (*compE2 A*) + *pc′*, *None*)

by(*rule AAcc-τExecrI2*)
moreover from *bisim′*
have *P*,*A*⌊*i*⌉,*h* ⊢ (*Val v*⌊*i′*⌉, *xs*) ↔ ((*rev vs* @ *Addr a* # *stk′*) @ [*v*], *loc′*, *length* (*compE2 A*) + *pc′*, *None*)
by(*rule bisim1-bisims1.bisim1AAcc2*)
ultimately show *?case* **using** *pc′* **by**(*fastforce*)
next
case (*bisim1AAss1 A n a′ xs stk loc pc i e*)
note *IH1* = ‹⟦*call1 a′* = ⌊(*a*, *M′*, *vs*)⌋; *n* + *max-vars a′* ≤ *length xs*; ¬ *contains-insync A* ⟧ ⟹ *?concl A n a′ xs pc stk loc*›
note *IH2* = ‹⋀*xs*. ⟦*call1 i* = ⌊(*a*, *M′*, *vs*)⌋; *n* + *max-vars i* ≤ *length xs*; ¬ *contains-insync i*⟧ ⟹ *?concl i n i xs 0* [] *xs*›
note *IH3* = ‹⋀*xs*. ⟦*call1 e* = ⌊(*a*, *M′*, *vs*)⌋; *n* + *max-vars e* ≤ *length xs*; ¬ *contains-insync e*⟧ ⟹ *?concl e n e xs 0* [] *xs*›
note *call* = ‹*call1* (*a′*⌊*i*⌉ := *e*) = ⌊(*a*, *M′*, *vs*)⌋›
note *len* = ‹*n* + *max-vars* (*a′*⌊*i*⌉ := *e*) ≤ *length xs*›
note *bisim1* = ‹*P*,*A*,*h* ⊢ (*a′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *bisim2* = ‹*P*,*i*,*h* ⊢ (*i*, *loc*) ↔ ([], *loc*, *0*, *None*)›
note *cs* = ‹¬ *contains-insync* (*A*⌊*i*⌉ := *e*)›
show *?case*
proof(*cases is-val a′*)
case *True*
then obtain *v* **where** [*simp*]: *a′* = *Val v* **by** *auto*
from *bisim1* **have** *τExec-mover-a P t A h* (*stk*, *loc*, *pc*, *None*) ([*v*], *loc*, *length* (*compE2 A*), *None*)
and [*simp*]: *xs* = *loc* **by**(*auto dest*!: *bisim1Val2D1*)
hence *exec*: *τExec-mover-a P t* (*A*⌊*i*⌉ := *e*) *h* (*stk*, *loc*, *pc*, *None*) ([*v*], *loc*, *length* (*compE2 A*), *None*)
by−(*rule AAss-τExecrI1*)
show *?thesis*
proof(*cases is-val i*)
case *True*
then obtain *v′* **where** [*simp*]: *i* = *Val v′* **by** *auto*
note *exec* **also from** *bisim2*
have *τExec-mover-a P t i h* ([], *loc*, *0*, *None*) ([*v′*], *loc*, *length* (*compE2 i*), *None*)
by(*auto dest*!: *bisim1Val2D1*)
from *AAss-τExecrI2*[*OF this*, *of A e v*]
have *τExec-mover-a P t* (*A*⌊*i*⌉ := *e*) *h* ([*v*], *loc*, *length* (*compE2 A*), *None*) ([*v′*, *v*], *loc*, *length* (*compE2 A*) + *length* (*compE2 i*), *None*) **by** *simp*
also (*rtranclp-trans*) **from** *call IH3*[*of loc*] *len cs* **obtain** *pc′ stk′ loc′*
where *exec*: *τExec-mover-a P t e h* ([], *loc*, *0*, *None*) (*rev vs* @ *Addr a* # *stk′*, *loc′*, *pc′*, *None*)
and *ins*: *compE2 e* ! *pc′* = *Invoke M′* (*length vs*) *pc′* < *length* (*compE2 e*)
and *bisim′*: *P*,*e*,*h* ⊢ (*e*, *loc*) ↔ (*rev vs* @ *Addr a* # *stk′*, *loc′*, *pc′*, *None*) **by** *auto*
from *AAss-τExecrI3*[*OF exec*, *of A i v′ v*]
have *τExec-mover-a P t* (*A*⌊*i*⌉ := *e*) *h* ([*v′*, *v*], *loc*, *length* (*compE2 A*) + *length* (*compE2 i*), *None*)
((*rev vs* @ *Addr a* # *stk′*) @ [*v′*, *v*], *loc′*, *length* (*compE2 A*) + *length* (*compE2 i*) + *pc′*, *None*) **by** *simp*
also (*rtranclp-trans*) **from** *bisim′*
have *P*,*A*⌊*i*⌉ := *e*,*h* ⊢ (*Val v*⌊*Val v′*⌉ := *e*, *xs*) ↔ ((*rev vs* @ *Addr a* # *stk′*) @ [*v′*, *v*], *loc′*, *length* (*compE2 A*) + *length* (*compE2 i*) + *pc′*, *None*)
by − (*rule bisim1AAss3*, *simp*)
ultimately show *?thesis* **using** *ins* **by** *fastforce*
next
case *False*

note *exec* **also from** *False call IH2*[*of loc*] *len cs* **obtain** *pc′ stk′ loc′*
where *exec*: *τExec-mover-a P t i h* ([], *xs*, *0*, *None*) (*rev vs* @ *Addr a* # *stk′*, *loc′*, *pc′*, *None*)
and *ins*: *compE2 i* ! *pc′* = *Invoke M′* (*length vs*) *pc′* < *length* (*compE2 i*)
and *bisim′*: *P*,*i*,*h* ⊢ (*i*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk′*, *loc′*, *pc′*, *None*) **by** *auto*
from *AAss-τExecrI2*[*OF exec*, *of A e v*]
have *τExec-mover-a P t* (*A*⌊*i*⌉ := *e*) *h* ([*v*], *loc*, *length* (*compE2 A*), *None*) (*rev vs* @ *Addr a* # (*stk′* @ [*v*]), *loc′*, *length* (*compE2 A*) + *pc′*, *None*) **by** *simp*
also (*rtranclp-trans*) **from** *bisim′*
have *P*,*A*⌊*i*⌉ := *e*,*h* ⊢ (*Val v*⌊*i*⌉ := *e*, *xs*) ↔ ((*rev vs* @ *Addr a* # *stk′*) @ [*v*], *loc′*, *length* (*compE2 A*) + *pc′*, *None*)
by(*rule bisim1AAss2*)
ultimately show *?thesis* **using** *ins False* **by**(*fastforce intro*!: *exI*)
qed
next
case *False* **with** *IH1 len False call cs* **show** *?thesis*
by(*clarsimp*)(*fastforce intro*: *bisim1-bisims1.bisim1AAss1 elim*!: *AAss-τExecrI1 intro*!: *exI*)
qed
next
case (*bisim1AAss2 i n i′ xs stk loc pc A e v*)
note *IH2* = ‹⟦*call1 i′* = ⌊(*a*, *M′*, *vs*)⌋; *n* + *max-vars i′* ≤ *length xs*; ¬ *contains-insync i*⟧ ⟹ *?concl i n i′ xs pc stk loc*›
note *IH3* = ‹⋀*xs*. ⟦*call1 e* = ⌊(*a*, *M′*, *vs*)⌋; *n* + *max-vars e* ≤ *length xs*; ¬ *contains-insync e*⟧ ⟹ *?concl e n e xs 0* [] *xs*›
note *call* = ‹*call1* (*Val v*⌊*i′*⌉ := *e*) = ⌊(*a*, *M′*, *vs*)⌋›
note *len* = ‹*n* + *max-vars* (*Val v*⌊*i′*⌉ := *e*) ≤ *length xs*›
note *bisim2* = ‹*P*,*i*,*h* ⊢ (*i′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *cs* = ‹¬ *contains-insync* (*A*⌊*i*⌉ := *e*)›
show *?case*
proof(*cases is-val i′*)
case *True*
then obtain *v′* **where** [*simp*]: *i′* = *Val v′* **by** *auto*
from *bisim2* **have** *exec*: *τExec-mover-a P t i h* (*stk*, *loc*, *pc*, *None*) ([*v′*], *loc*, *length* (*compE2 i*), *None*)
and [*simp*]: *xs* = *loc* **by**(*auto dest*!: *bisim1Val2D1*)
from *AAss-τExecrI2*[*OF exec*, *of A e v*]
have *τExec-mover-a P t* (*A*⌊*i*⌉ := *e*) *h* (*stk* @ [*v*], *loc*, *length* (*compE2 A*) + *pc*, *None*) ([*v′*, *v*], *loc*, *length* (*compE2 A*) + *length* (*compE2 i*), *None*) **by** *simp*
also from *call IH3*[*of loc*] *len cs* **obtain** *pc′ stk′ loc′*
where *exec*: *τExec-mover-a P t e h* ([], *xs*, *0*, *None*) (*rev vs* @ *Addr a* # *stk′*, *loc′*, *pc′*, *None*)
and *ins*: *compE2 e* ! *pc′* = *Invoke M′* (*length vs*) *pc′* < *length* (*compE2 e*)
and *bisim′*: *P*,*e*,*h* ⊢ (*e*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk′*, *loc′*, *pc′*, *None*) **by** *auto*
from *AAss-τExecrI3*[*OF exec*, *of A i v′ v*]
have *τExec-mover-a P t* (*A*⌊*i*⌉ := *e*) *h* ([*v′*, *v*], *loc*, *length* (*compE2 A*) + *length* (*compE2 i*), *None*) ((*rev vs* @ *Addr a* # *stk′*) @ [*v′*, *v*], *loc′*, *length* (*compE2 A*) + *length* (*compE2 i*) + *pc′*, *None*) **by** *simp*
also (*rtranclp-trans*) **from** *bisim′*
have *P*,*A*⌊*i*⌉ := *e*,*h* ⊢ (*Val v*⌊*Val v′*⌉ := *e*, *xs*) ↔ ((*rev vs* @ *Addr a* # *stk′*) @ [*v′*, *v*], *loc′*, *length* (*compE2 A*) + *length* (*compE2 i*) + *pc′*, *None*)
by(*rule bisim1AAss3*)
ultimately show *?thesis* **using** *ins* **by**(*fastforce intro*!: *exI*)
next
case *False*
with *IH2 len call cs* **obtain** *pc′ loc′ stk′*
where *ins*: *pc′* < *length* (*compE2 i*) *compE2 i* ! *pc′* = *Invoke M′* (*length vs*)

and *exec*: τ*Exec-mover-a P t i h* (*stk*, *loc*, *pc*, *None*) (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*)
and *bisim'*: *P*,*i*,*h* ⊢ (*i'*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*) **by** *fastforce*
from *bisim'* **have** *P*,*A*⌊*i*⌉ := *e*,*h* ⊢ (*Val v*⌊*i'*⌉ := *e*, *xs*) ↔ ((*rev vs* @ *Addr a* # *stk'*) @ [*v*], *loc'*, *length* (*compE2 A*) + *pc'*, *None*)
by(*rule bisim1-bisims1.bisim1AAss2*)
with *AAss-*τ*ExecrI2*[*OF exec*, *of A e v*] *ins False* **show** *?thesis* **by**(*auto intro*!: *exI*)
qed
next
case (*bisim1AAss3 e n e' xs stk loc pc A i v v'*)
then obtain *pc' loc' stk'* **where** *pc'*: *pc'* < *length* (*compE2 e*) *compE2 e* ! *pc'* = *Invoke M'* (*length vs*)
and *exec*: τ*Exec-mover-a P t e h* (*stk*, *loc*, *pc*, *None*) (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*)
and *bisim'*: *P*,*e*,*h* ⊢ (*e'*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*) **by** *fastforce*
from *exec* **have** τ*Exec-mover-a P t* (*A*⌊*i*⌉:=*e*) *h* (*stk* @ [*v'*, *v*], *loc*, *length* (*compE2 A*) + *length* (*compE2 i*) + *pc*, *None*)
((*rev vs* @ *Addr a* # *stk'*) @ [*v'*, *v*], *loc'*, *length* (*compE2 A*) + *length* (*compE2 i*) + *pc'*, *None*)
by(*rule AAss-*τ*ExecrI3*)
moreover from *bisim'*
have *P*,*A*⌊*i*⌉ := *e*,*h* ⊢ (*Val v*⌊*Val v'*⌉ := *e'*, *xs*) ↔ ((*rev vs* @ *Addr a* # *stk'*) @ [*v'*, *v*], *loc'*, *length* (*compE2 A*) + *length* (*compE2 i*) + *pc'*, *None*)
by(*rule bisim1-bisims1.bisim1AAss3*)
ultimately show *?case* **using** *pc'* **by**(*fastforce intro*!: *exI*)
next
case *bisim1AAss4* **thus** *?case* **by** *simp*
next
case *bisim1ALength* **thus** *?case*
by(*auto*)(*fastforce intro*: *bisim1-bisims1.bisim1ALength elim*!: *ALength-*τ*ExecrI intro*!: *exI*)
next
case *bisim1FAcc* **thus** *?case*
by(*auto*)(*fastforce intro*: *bisim1-bisims1.bisim1FAcc elim*!: *FAcc-*τ*ExecrI intro*!: *exI*)
next
case (*bisim1FAss1 e n e' xs stk loc pc e2 F D*)
note *IH1* = ‹⟦*call1 e'* = ⌊(*a*, *M'*, *vs*)⌋; *n* + *max-vars e'* ≤ *length xs*; ¬ *contains-insync e*⟧ ⟹ *?concl e n e' xs pc stk loc*›
note *IH2* = ‹⋀*xs*. ⟦*call1 e2* = ⌊(*a*, *M'*, *vs*)⌋; *n* + *max-vars e2* ≤ *length xs*; ¬ *contains-insync e2*⟧ ⟹ *?concl e2 n e2 xs 0* [] *xs*›
note *call* = ‹*call1* (*e'*·*F*{*D*} := *e2*) = ⌊(*a*, *M'*, *vs*)⌋›
note *len* = ‹*n* + *max-vars* (*e'*·*F*{*D*} := *e2*) ≤ *length xs*›
note *bisim1* = ‹*P*,*e*,*h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *cs* = ‹¬ *contains-insync* (*e*·*F*{*D*} := *e2*)›
show *?case*
proof(*cases is-val e'*)
case *True*
then obtain *v* **where** [*simp*]: *e'* = *Val v* **by** *auto*
from *bisim1* **have** τ*Exec-mover-a P t e h* (*stk*, *loc*, *pc*, *None*) ([*v*], *loc*, *length* (*compE2 e*), *None*)
and [*simp*]: *xs* = *loc* **by**(*auto dest*!: *bisim1Val2D1*)
hence τ*Exec-mover-a P t* (*e*·*F*{*D*} := *e2*) *h* (*stk*, *loc*, *pc*, *None*) ([*v*], *loc*, *length* (*compE2 e*), *None*)
by−(*rule FAss-*τ*ExecrI1*)
also from *call IH2*[*of loc*] *len cs* **obtain** *pc' stk' loc'*
where *exec*: τ*Exec-mover-a P t e2 h* ([], *xs*, *0*, *None*) (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*)
and *ins*: *compE2 e2* ! *pc'* = *Invoke M'* (*length vs*) *pc'* < *length* (*compE2 e2*)
and *bisim'*: *P*,*e2*,*h* ⊢ (*e2*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*) **by** *auto*
from *FAss-*τ*ExecrI2*[*OF exec*, *of e F D v*]

have *τExec-mover-a P t* (*e·F{D}* := *e2*) *h* ([*v*], *loc*, *length* (*compE2 e*), *None*) (*rev vs* @ *Addr a* # (*stk′* @ [*v*]), *loc′*, *length* (*compE2 e*) + *pc′*, *None*) **by** *simp*
also (*rtranclp-trans*) **from** *bisim′*
have *P*,*e·F{D}* := *e2*,*h* ⊢ (*Val v·F{D}* := *e2*, *xs*) ↔ ((*rev vs* @ *Addr a* # *stk′*) @ [*v*], *loc′*, *length* (*compE2 e*) + *pc′*, *None*)
by(*rule bisim1FAss2*)
ultimately show *?thesis* **using** *ins* **by** *fastforce*
next
case *False* **with** *IH1 len False call cs* **show** *?thesis*
by(*clarsimp*)(*fastforce intro*: *bisim1-bisims1.bisim1FAss1 elim*!: *FAss-τExecrI1 intro*!: *exI*)
qed
next
case (*bisim1FAss2 e2 n e′ xs stk loc pc e F D v*)
then obtain *pc′ loc′ stk′* **where** *pc′*: *pc′* < *length* (*compE2 e2*) *compE2 e2* ! *pc′* = *Invoke M′* (*length vs*)
and *exec*: *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *None*) (*rev vs* @ *Addr a* # *stk′*, *loc′*, *pc′*, *None*)
and *bisim′*: *P*,*e2*,*h* ⊢ (*e′*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk′*, *loc′*, *pc′*, *None*) **by** *fastforce*
from *exec* **have** *τExec-mover-a P t* (*e·F{D}* := *e2*) *h* (*stk* @ [*v*], *loc*, *length* (*compE2 e*) + *pc*, *None*) ((*rev vs* @ *Addr a* # *stk′*) @ [*v*], *loc′*, *length* (*compE2 e*) + *pc′*, *None*)
by(*rule FAss-τExecrI2*)
moreover from *bisim′*
have *P*,*e·F{D}* := *e2*,*h* ⊢ (*Val v·F{D}* := *e′*, *xs*) ↔ ((*rev vs* @ *Addr a* # *stk′*) @ [*v*], *loc′*, *length* (*compE2 e*) + *pc′*, *None*)
by(*rule bisim1-bisims1.bisim1FAss2*)
ultimately show *?case* **using** *pc′* **by**(*fastforce*)
next
case *bisim1FAss3* **thus** *?case* **by** *simp*
next
case (*bisim1CAS1 e1 n e′ xs stk loc pc e2 e3 D F*)
note *IH1* = ‹⟦*call1 e′* = ⌊(*a*, *M′*, *vs*)⌋; *n* + *max-vars e′* ≤ *length xs*; ¬ *contains-insync e1* ⟧ ⟹ *?concl e1 n e′ xs pc stk loc*›
note *IH2* = ‹⋀*xs*. ⟦*call1 e2* = ⌊(*a*, *M′*, *vs*)⌋; *n* + *max-vars e2* ≤ *length xs*; ¬ *contains-insync e2*⟧ ⟹ *?concl e2 n e2 xs 0* [] *xs*›
note *IH3* = ‹⋀*xs*. ⟦*call1 e3* = ⌊(*a*, *M′*, *vs*)⌋; *n* + *max-vars e3* ≤ *length xs*; ¬ *contains-insync e3*⟧ ⟹ *?concl e3 n e3 xs 0* [] *xs*›
note *call* = ‹*call1* - = ⌊(*a*, *M′*, *vs*)⌋›
note *len* = ‹*n* + *max-vars* - ≤ *length xs*›
note *bisim1* = ‹*P*,*e1*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e2*, *loc*) ↔ ([], *loc*, *0*, *None*)›
note *cs* = ‹¬ *contains-insync* -›
show *?case*
proof(*cases is-val e′*)
case *True*
then obtain *v* **where** [*simp*]: *e′* = *Val v* **by** *auto*
from *bisim1* **have** *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *None*) ([*v*], *loc*, *length* (*compE2 e1*), *None*)
and [*simp*]: *xs* = *loc* **by**(*auto dest*!: *bisim1Val2D1*)
hence *exec*: *τExec-mover-a P t* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) *h* (*stk*, *loc*, *pc*, *None*) ([*v*], *loc*, *length* (*compE2 e1*), *None*)
by−(*rule CAS-τExecrI1*)
show *?thesis*
proof(*cases is-val e2*)
case *True*
then obtain *v′* **where** [*simp*]: *e2* = *Val v′* **by** *auto*
note *exec* **also from** *bisim2*

have *τExec-mover-a P t e2 h* ([], *loc*, *0*, *None*) ([*v'*], *loc*, *length* (*compE2 e2*), *None*)
by(*auto dest*!: *bisim1Val2D1*)
from *CAS-τExecrI2*[*OF this*, *of e1 D F e3*]
have *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([*v*], *loc*, *length* (*compE2 e1*), *None*) ([*v'*, *v*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*) **by** *simp*
also (*rtranclp-trans*) **from** *call IH3*[*of loc*] *len cs* **obtain** *pc' stk' loc'*
where *exec*: *τExec-mover-a P t e3 h* ([], *loc*, *0*, *None*) (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*)
and *ins*: *compE2 e3* ! *pc'* = *Invoke M'* (*length vs*) *pc'* < *length* (*compE2 e3*)
and *bisim'*: *P*,*e3*,*h* ⊢ (*e3*, *loc*) ↔ (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*) **by** *auto*
from *CAS-τExecrI3*[*OF exec*, *of e1 D F e2 v' v*]
have *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([*v'*, *v*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
((*rev vs* @ *Addr a* # *stk'*) @ [*v'*, *v*], *loc'*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc'*, *None*) **by** *simp*
also (*rtranclp-trans*) **from** *bisim'*
have *P*,*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*),*h* ⊢ (*Val v*·*compareAndSwap*(*D*·*F*, *Val v'*, *e3*), *xs*) ↔ ((*rev vs* @ *Addr a* # *stk'*) @ [*v'*, *v*], *loc'*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc'*, *None*)
by − (*rule bisim1CAS3*, *simp*)
ultimately show *?thesis* **using** *ins* **by** *fastforce*
next
case *False*
note *exec* **also from** *False call IH2*[*of loc*] *len cs* **obtain** *pc' stk' loc'*
where *exec*: *τExec-mover-a P t e2 h* ([], *xs*, *0*, *None*) (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*)
and *ins*: *compE2 e2* ! *pc'* = *Invoke M'* (*length vs*) *pc'* < *length* (*compE2 e2*)
and *bisim'*: *P*,*e2*,*h* ⊢ (*e2*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*) **by** *auto*
from *CAS-τExecrI2*[*OF exec*, *of e1 D F e3 v*]
have *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([*v*], *loc*, *length* (*compE2 e1*), *None*) (*rev vs* @ *Addr a* # (*stk'* @ [*v*]), *loc'*, *length* (*compE2 e1*) + *pc'*, *None*) **by** *simp*
also (*rtranclp-trans*) **from** *bisim'*
have *P*,*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*),*h* ⊢ (*Val v*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *xs*) ↔ ((*rev vs* @ *Addr a* # *stk'*) @ [*v*], *loc'*, *length* (*compE2 e1*) + *pc'*, *None*)
by(*rule bisim1CAS2*)
ultimately show *?thesis* **using** *ins False* **by**(*fastforce intro*!: *exI*)
qed
next
case *False* **with** *IH1 len False call cs* **show** *?thesis*
by(*clarsimp*)(*fastforce intro*: *bisim1-bisims1.bisim1CAS1 elim*!: *CAS-τExecrI1 intro*!: *exI*)
qed
next
case (*bisim1CAS2 e2 n e2' xs stk loc pc e1 e3 D F v*)
note *IH2* = ‹⟦*call1 e2'* = ⌊(*a*, *M'*, *vs*)⌋; *n* + *max-vars e2'* ≤ *length xs*; ¬ *contains-insync e2*⟧ ⟹ *?concl e2 n e2' xs pc stk loc*›
note *IH3* = ‹⋀*xs*. ⟦*call1 e3* = ⌊(*a*, *M'*, *vs*)⌋; *n* + *max-vars e3* ≤ *length xs*; ¬ *contains-insync e3*⟧ ⟹ *?concl e3 n e3 xs 0* [] *xs*›
note *call* = ‹*call1* - = ⌊(*a*, *M'*, *vs*)⌋›
note *len* = ‹*n* + *max-vars* - ≤ *length xs*›
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e2'*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *cs* = ‹¬ *contains-insync* -›
show *?case*
proof(*cases is-val e2'*)
case *True*
then obtain *v'* **where** [*simp*]: *e2'* = *Val v'* **by** *auto*
from *bisim2* **have** *exec*: *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *None*) ([*v'*], *loc*, *length* (*compE2 e2*), *None*)

and [*simp*]: *xs* = *loc* **by**(*auto dest*!: *bisim1Val2D1*)
from *CAS-τExecrI2*[*OF exec*, *of e1 D F e3 v*]
have *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk* @ [*v*], *loc*, *length* (*compE2 e1*) + *pc*, *None*) ([*v*′, *v*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*) **by** *simp*
also from *call IH3*[*of loc*] *len cs* **obtain** *pc*′ *stk*′ *loc*′
where *exec*: *τExec-mover-a P t e3 h* ([], *xs*, *0*, *None*) (*rev vs* @ *Addr a* # *stk*′, *loc*′, *pc*′, *None*)
and *ins*: *compE2 e3* ! *pc*′ = *Invoke M*′ (*length vs*) *pc*′ < *length* (*compE2 e3*)
and *bisim*′: *P*,*e3*,*h* ⊢ (*e3*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk*′, *loc*′, *pc*′, *None*) **by** *auto*
from *CAS-τExecrI3*[*OF exec*, *of e1 D F e2 v*′ *v*]
have *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([*v*′, *v*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
((*rev vs* @ *Addr a* # *stk*′) @ [*v*′, *v*], *loc*′, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*′, *None*) **by** *simp*
also (*rtranclp-trans*) **from** *bisim*′
have *P*,*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*),*h* ⊢ (*Val v*·*compareAndSwap*(*D*·*F*, *Val v*′, *e3*), *xs*) ↔ ((*rev vs* @ *Addr a* # *stk*′) @ [*v*′, *v*], *loc*′, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*′, *None*)
by(*rule bisim1CAS3*)
ultimately show *?thesis* **using** *ins* **by**(*fastforce intro*!: *exI*)
next
case *False*
with *IH2 len call cs* **obtain** *pc*′ *loc*′ *stk*′
where *ins*: *pc*′ < *length* (*compE2 e2*) *compE2 e2* ! *pc*′ = *Invoke M*′ (*length vs*)
and *exec*: *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *None*) (*rev vs* @ *Addr a* # *stk*′, *loc*′, *pc*′, *None*)
and *bisim*′: *P*,*e2*,*h* ⊢ (*e2*′, *xs*) ↔ (*rev vs* @ *Addr a* # *stk*′, *loc*′, *pc*′, *None*) **by** *fastforce*
from *bisim*′ **have** *P*,*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*),*h* ⊢ (*Val v*·*compareAndSwap*(*D*·*F*, *e2*′, *e3*), *xs*) ↔ ((*rev vs* @ *Addr a* # *stk*′) @ [*v*], *loc*′, *length* (*compE2 e1*) + *pc*′, *None*)
by(*rule bisim1-bisims1.bisim1CAS2*)
with *CAS-τExecrI2*[*OF exec*, *of e1 D F e3 v*] *ins False* **show** *?thesis* **by**(*auto intro*!: *exI*)
qed
next
case (*bisim1CAS3 e3 n e3*′ *xs stk loc pc e1 e2 D F v v*′)
then obtain *pc*′ *loc*′ *stk*′ **where** *pc*′: *pc*′ < *length* (*compE2 e3*) *compE2 e3* ! *pc*′ = *Invoke M*′ (*length vs*)
and *exec*: *τExec-mover-a P t e3 h* (*stk*, *loc*, *pc*, *None*) (*rev vs* @ *Addr a* # *stk*′, *loc*′, *pc*′, *None*)
and *bisim*′: *P*,*e3*,*h* ⊢ (*e3*′, *xs*) ↔ (*rev vs* @ *Addr a* # *stk*′, *loc*′, *pc*′, *None*) **by** *fastforce*
from *exec* **have** *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk* @ [*v*′, *v*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*, *None*)
((*rev vs* @ *Addr a* # *stk*′) @ [*v*′, *v*], *loc*′, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*′, *None*)
by(*rule CAS-τExecrI3*)
moreover from *bisim*′
have *P*,*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*),*h* ⊢ (*Val v*·*compareAndSwap*(*D*·*F*, *Val v*′, *e3*′), *xs*) ↔ ((*rev vs* @ *Addr a* # *stk*′) @ [*v*′, *v*], *loc*′, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*′, *None*)
by(*rule bisim1-bisims1.bisim1CAS3*)
ultimately show *?case* **using** *pc*′ **by**(*fastforce intro*!: *exI*)
next
case (*bisim1Call1 obj n obj*′ *xs stk loc pc ps M*)
note *IH1* = ‹⟦*call1 obj*′ = ⌊(*a*, *M*′, *vs*)⌋; *n* + *max-vars obj*′ ≤ *length xs*; ¬ *contains-insync obj*⟧ ⟹ *?concl obj n obj*′ *xs pc stk loc*›
note *IH2* = ‹⋀*xs*. ⟦*calls1 ps* = ⌊(*a*, *M*′, *vs*)⌋; *n* + *max-varss ps* ≤ *length xs*; ¬ *contains-insyncs ps*⟧ ⟹ *?concls ps n ps xs 0* [] *xs*›
note *len* = ‹*n* + *max-vars* (*obj*′·*M*(*ps*)) ≤ *length xs*›
note *bisim1* = ‹*P*,*obj*,*h* ⊢ (*obj*′, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *call* = ‹*call1* (*obj*′·*M*(*ps*)) = ⌊(*a*, *M*′, *vs*)⌋›

note *cs* = ‹¬ *contains-insync* (*obj*·*M*(*ps*))›
from *call* **show** *?case*
proof(*cases rule*: *call1-callE*)
case *CallObj*
hence ¬ *is-val obj'* **by** *auto*
with *CallObj IH1 len cs* **show** *?thesis*
by(*clarsimp*)(*fastforce intro*: *bisim1-bisims1.bisim1Call1 elim*!: *Call-τExecrI1 intro*!: *exI*)
next
case (*CallParams v*)
with *bisim1* **have** *τExec-mover-a P t obj h* (*stk*, *loc*, *pc*, *None*) ([*v*], *loc*, *length* (*compE2 obj*), *None*)
and [*simp*]: *xs* = *loc* **by**(*auto dest*!: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*obj*·*M*(*ps*)) *h* (*stk*, *loc*, *pc*, *None*) ([*v*], *loc*, *length* (*compE2 obj*), *None*)
by−(*rule Call-τExecrI1*)
also from *IH2*[*of loc*] *CallParams len cs* **obtain** *pc' stk' loc'*
where *exec*: *τExec-movesr-a P t ps h* ([], *loc*, *0*, *None*) (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*)
and *ins*: *compEs2 ps* ! *pc'* = *Invoke M'* (*length vs*) *pc'* < *length* (*compEs2 ps*)
and *bisim'*: *P*,*ps*,*h* ⊢ (*ps*, *xs*) [↔] (*rev vs* @ *Addr a* # *stk'*,*loc'*,*pc'*,*None*) **by** *auto*
from *Call-τExecrI2*[*OF exec*, *of obj M v*]
have *τExec-mover-a P t* (*obj*·*M*(*ps*)) *h* ([*v*], *loc*, *length* (*compE2 obj*), *None*) (*rev vs* @ *Addr a* # (*stk'* @ [*v*]), *loc'*, *length* (*compE2 obj*) + *pc'*, *None*) **by** *simp*
also (*rtranclp-trans*)
have *P*,*obj*·*M*(*ps*),*h* ⊢ (*Val v*·*M*(*ps*), *xs*) ↔ ((*rev vs* @ *Addr a* # *stk'*) @ [*v*], *loc'*, *length* (*compE2 obj*) + *pc'*, *None*)
using *bisim'* **by**(*rule bisim1CallParams*)
ultimately show *?thesis* **using** *ins CallParams* **by** *fastforce*
next
case [*simp*]: *Call*
from *bisim1* **have** *τExec-mover-a P t obj h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *length* (*compE2 obj*), *None*)
and [*simp*]: *xs* = *loc* **by**(*auto dest*!: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*obj*·*M*(*ps*)) *h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *length* (*compE2 obj*), *None*)
by−(*rule Call-τExecrI1*)
also have *τExec-movesr-a P t ps h* ([], *xs*, *0*, *None*) (*rev vs*, *xs*, *length* (*compEs2 ps*), *None*)
proof(*cases vs*)
case *Nil* **with** *Call* **show** *?thesis* **by**(*auto*)
next
case *Cons* **with** *Call bisims1-Val-τExec-moves*[*OF bisims1-refl*[*of P h map Val vs loc*]]
show *?thesis* **by**(*auto simp add*: *bsoks-def*)
qed
from *Call-τExecrI2*[*OF this*, *of obj M Addr a*]
have *τExec-mover-a P t* (*obj*·*M*(*ps*)) *h* ([*Addr a*], *loc*, *length* (*compE2 obj*), *None*) (*rev vs* @ [*Addr a*], *xs*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), *None*) **by** *simp*
also (*rtranclp-trans*)
have *P*,*ps*,*h* ⊢ (*map Val vs*,*xs*) [↔] (*rev vs*,*xs*,*length* (*compEs2 ps*),*None*)
by(*rule bisims1-map-Val-append*[*OF bisims1Nil*, *simplified*])(*simp-all add*: *bsoks-def*)
hence *P*,*obj*·*M*(*ps*),*h* ⊢ (*addr a*·*M*(*map Val vs*), *xs*) ↔ (*rev vs* @ [*Addr a*], *xs*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), *None*)
by(*rule bisim1CallParams*)
ultimately show *?thesis* **by** *fastforce*
qed
next
case (*bisim1CallParams ps n ps' xs stk loc pc obj M v*)

note *IH2* = ‹⟦*calls1 ps′* = ⌊(*a*, *M′*, *vs*)⌋; *n* + *max-varss ps′* ≤ *length xs*; ¬ *contains-insyncs ps*⟧ ⟹ *?concls ps n ps′ xs pc stk loc*›
note *bisim2* = ‹*P*,*ps*,*h* ⊢ (*ps′*, *xs*) [↔] (*stk*, *loc*, *pc*, *None*)›
note *call* = ‹*call1* (*Val v*·*M*(*ps′*)) = ⌊(*a*, *M′*, *vs*)⌋›
note *len* = ‹*n* + *max-vars* (*Val v*·*M*(*ps′*)) ≤ *length xs*›
note *cs* = ‹¬ *contains-insync* (*obj*·*M*(*ps*))›
from *call* **show** *?case*
proof(*cases rule*: *call1-callE*)
case *CallObj* **thus** *?thesis* **by** *simp*
next
case (*CallParams v′*)
with *IH2 len cs* **obtain** *pc′ stk′ loc′*
where *exec*: *τExec-movesr-a P t ps h* (*stk*, *loc*, *pc*, *None*) (*rev vs* @ *Addr a* # *stk′*, *loc′*, *pc′*, *None*)
and *ins*: *pc′* < *length* (*compEs2 ps*) *compEs2 ps* ! *pc′* = *Invoke M′* (*length vs*)
and *bisim′*: *P*,*ps*,*h* ⊢ (*ps′*, *xs*) [↔] (*rev vs* @ *Addr a* # *stk′*,*loc′*,*pc′*,*None*) **by** *auto*
from *exec* **have** *τExec-mover-a P t* (*obj*·*M*(*ps*)) *h* (*stk* @ [*v*], *loc*, *length* (*compE2 obj*) + *pc*, *None*)
((*rev vs* @ *Addr a* # *stk′*) @ [*v*], *loc′*, *length* (*compE2 obj*) + *pc′*, *None*)
by(*rule Call-τExecrI2*)
moreover have *P*,*obj*·*M*(*ps*),*h* ⊢ (*Val v*·*M*(*ps′*), *xs*) ↔
((*rev vs* @ *Addr a* # *stk′*) @ [*v*], *loc′*, *length* (*compE2 obj*) + *pc′*, *None*)
using *bisim′* **by**(*rule bisim1-bisims1.bisim1CallParams*)
ultimately show *?thesis* **using** *ins* **by** *fastforce*
next
case *Call*
hence [*simp*]: *v* = *Addr a ps′* = *map Val vs M′* = *M* **by** *simp-all*
have *xs* = *loc* ∧ *τExec-movesr-a P t ps h* (*stk*, *loc*, *pc*, *None*) (*rev vs*, *loc*, *length* (*compEs2 ps*), *None*)
proof(*cases pc* < *length* (*compEs2 ps*))
case *True* **with** *bisim2* **show** *?thesis* **by**(*auto dest*: *bisims1-Val-τExec-moves*)
next
case *False*
from *bisim2* **have** *pc* ≤ *length* (*compEs2 ps*) **by**(*rule bisims1-pc-length-compEs2*)
with *False* **have** *pc* = *length* (*compEs2 ps*) **by** *simp*
with *bisim2* **show** *?thesis* **by**(*auto dest*: *bisims1-Val-length-compEs2D*)
qed
then obtain [*simp*]: *xs* = *loc*
and *exec*: *τExec-movesr-a P t ps h* (*stk*, *loc*, *pc*, *None*) (*rev vs*, *loc*, *length* (*compEs2 ps*), *None*)
..
from *exec* **have** *τExec-mover-a P t* (*obj*·*M*(*ps*)) *h* (*stk* @ [*v*], *loc*, *length* (*compE2 obj*) + *pc*, *None*)
(*rev vs* @ [*v*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), *None*)
by(*rule Call-τExecrI2*)
moreover from *bisim2* **have** *len*: *length ps* = *length ps′* **by**(*auto dest*: *bisims1-lengthD*)
moreover have *P*,*ps*,*h* ⊢ (*map Val vs*,*xs*) [↔] (*rev vs*,*xs*,*length* (*compEs2 ps*),*None*) **using** *len*
by−(*rule bisims1-map-Val-append*[*OF bisims1Nil*, *simplified*], *simp-all*)
hence *P*,*obj*·*M*(*ps*),*h* ⊢ (*addr a*·*M*(*map Val vs*), *xs*) ↔ (*rev vs* @ [*Addr a*], *xs*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), *None*) **by**(*rule bisim1-bisims1.bisim1CallParams*)
ultimately show *?thesis* **by** *fastforce*
qed
next
case *bisim1BlockSome1* **thus** *?case* **by** *simp*
next
case *bisim1BlockSome2* **thus** *?case* **by** *simp*
next

case (*bisim1BlockSome4 e n e' xs stk loc pc V T v*)
then obtain *pc' loc' stk'* **where** *pc'*: *pc' < length* (*compE2 e*) *compE2 e* ! *pc' = Invoke M'* (*length vs*)
and *exec*: *τExec-mover-a P t e h* (*stk*, *loc*, *pc*, *None*) (*rev vs* @ *Addr a # stk'*, *loc'*, *pc'*, *None*)
and *bisim'*: *P*,*e*,*h* ⊢ (*e'*, *xs*) ↔ (*rev vs* @ *Addr a # stk'*, *loc'*, *pc'*, *None*) **by** *auto*
note *Block-τExecrI-Some*[*OF exec*, *of V T v*]
moreover from *bisim'* **have** *P*,{*V*:*T*=⌊*v*⌋; *e*},*h* ⊢ ({*V*:*T*=*None*; *e'*}, *xs*) ↔ (*rev vs* @ *Addr a # stk'*, *loc'*, *Suc* (*Suc pc'*), *None*)
by(*rule bisim1-bisims1.bisim1BlockSome4*)
ultimately show *?case* **using** *pc'* **by** *fastforce*
next
case (*bisim1BlockNone e n e' xs stk loc pc V T*)
then obtain *pc' loc' stk'* **where** *pc'*: *pc' < length* (*compE2 e*) *compE2 e* ! *pc' = Invoke M'* (*length vs*)
and *exec*: *τExec-mover-a P t e h* (*stk*, *loc*, *pc*, *None*) (*rev vs* @ *Addr a # stk'*, *loc'*, *pc'*, *None*)
and *bisim'*: *P*,*e*,*h* ⊢ (*e'*, *xs*) ↔ (*rev vs* @ *Addr a # stk'*, *loc'*, *pc'*, *None*) **by** *auto*
note *Block-τExecrI-None*[*OF exec*, *of V T*]
moreover from *bisim'* **have** *P*,{*V*:*T*=*None*; *e*},*h* ⊢ ({*V*:*T*=*None*; *e'*}, *xs*) ↔ (*rev vs* @ *Addr a # stk'*, *loc'*, *pc'*, *None*)
by(*rule bisim1-bisims1.bisim1BlockNone*)
ultimately show *?case* **using** *pc'* **by** *fastforce*
next
case *bisim1Sync1* **thus** *?case*
by (*auto*)(*fastforce intro*: *bisim1-bisims1.bisim1Sync1 elim*!: *Sync-τExecrI intro*!: *exI*)
next
case *bisim1Sync2* **thus** *?case* **by** *simp*
next
case *bisim1Sync3* **thus** *?case* **by** *simp*
next
case *bisim1Sync4* **thus** *?case*
by (*auto*)(*fastforce intro*: *bisim1-bisims1.bisim1Sync4 elim*!: *Insync-τExecrI intro*!: *exI*)
next
case *bisim1Sync5* **thus** *?case* **by** *simp*
next
case *bisim1Sync6* **thus** *?case* **by** *simp*
next
case *bisim1Sync7* **thus** *?case* **by** *simp*
next
case *bisim1Sync8* **thus** *?case* **by** *simp*
next
case *bisim1Sync9* **thus** *?case* **by** *simp*
next
case *bisim1InSync* **thus** *?case* **by** *simp*
next
case *bisim1Seq1* **thus** *?case*
by (*auto*)(*fastforce intro*: *bisim1-bisims1.bisim1Seq1 elim*!: *Seq-τExecrI1 intro*!: *exI*)
next
case (*bisim1Seq2 e2 n e' xs stk loc pc e1*)
then obtain *pc' loc' stk'* **where** *pc'*: *pc' < length* (*compE2 e2*) *compE2 e2* ! *pc' = Invoke M'* (*length vs*)
and *exec*: *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *None*) (*rev vs* @ *Addr a # stk'*, *loc'*, *pc'*, *None*)
and *bisim'*: *P*,*e2*,*h* ⊢ (*e'*, *xs*) ↔ (*rev vs* @ *Addr a # stk'*, *loc'*, *pc'*, *None*) **by** *auto*
from *Seq-τExecrI2*[*OF exec*, *of e1*] *pc' bisim'*
show *?case* **by**(*fastforce intro*: *bisim1-bisims1.bisim1Seq2 intro*!: *exI*)

next
case *bisim1Cond1* **thus** *?case*
by (*auto*)(*fastforce intro*: *bisim1-bisims1.bisim1Cond1 elim*!: *Cond-τExecrI1 intro*!: *exI*)+
next
case (*bisim1CondThen e1 n e' xs stk loc pc e e2*)
then obtain *pc' loc' stk'* **where** *pc'*: *pc' < length* (*compE2 e1*) *compE2 e1* ! *pc' = Invoke M'* (*length vs*)
and *exec*: *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *None*) (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*)
and *bisim'*: *P*,*e1*,*h* ⊢ (*e'*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*) **by** *auto*
from *Cond-τExecrI2*[*OF exec*] *pc' bisim'* **show** *?case*
by(*fastforce intro*: *bisim1-bisims1.bisim1CondThen intro*!: *exI*)
next
case (*bisim1CondElse e2 n e' xs stk loc pc e e1*)
then obtain *pc' loc' stk'* **where** *pc'*: *pc' < length* (*compE2 e2*) *compE2 e2* ! *pc' = Invoke M'* (*length vs*)
and *exec*: *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *None*) (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*)
and *bisim'*: *P*,*e2*,*h* ⊢ (*e'*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*) **by** *auto*
from *Cond-τExecrI3*[*OF exec*] *pc' bisim'* **show** *?case*
by (*fastforce intro*: *bisim1-bisims1.bisim1CondElse intro*!: *exI*)
next
case *bisim1While1* **thus** *?case* **by** *simp*
next
case *bisim1While3* **thus** *?case*
by (*auto*)(*fastforce intro*: *bisim1-bisims1.bisim1While3 elim*!: *While-τExecrI1 intro*!: *exI*)+
next
case *bisim1While4* **thus** *?case*
by (*auto*)(*fastforce intro*!: *While-τExecrI2 bisim1-bisims1.bisim1While4 exI*)+
next
case *bisim1While6* **thus** *?case* **by** *simp*
next
case *bisim1While7* **thus** *?case* **by** *simp*
next
case *bisim1Throw1* **thus** *?case*
by (*auto*)(*fastforce intro*!: *exI bisim1-bisims1.bisim1Throw1 elim*!: *Throw-τExecrI*)+
next
case *bisim1Try* **thus** *?case*
by (*auto*)(*fastforce intro*: *bisim1-bisims1.bisim1Try elim*!: *Try-τExecrI1 intro*!: *exI*)+
next
case (*bisim1TryCatch1 e n a' xs stk loc pc C' C e2 V*)
note *IH2* = ‹⋀*xs*. ⟦*call1 e2* = ⌊(*a*, *M'*, *vs*)⌋; *Suc n* + *max-vars e2* ≤ *length xs*; ¬ *contains-insync e2* ⟧ ⟹ *?concl e2* (*Suc V*) *e2 xs 0* [] *xs*›
note *bisim1* = ‹*P*,*e*,*h* ⊢ (*Throw a'*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a'*⌋)›
note *bisim2* = ‹⋀*xs*. *P*,*e2*,*h* ⊢ (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *len* = ‹*n* + *max-vars* {*V*:*Class C*=*None*; *e2*} ≤ *length* (*xs*[*V* := *Addr a'*])›
note *cs* = ‹¬ *contains-insync* (*try e catch*(*C V*) *e2*)›
from *bisim1* **have** [*simp*]: *xs* = *loc* **by**(*auto dest*: *bisim1-ThrowD*)
from *len* **have** *τExec-mover-a P t* (*try e catch*(*C V*) *e2*) *h* ([*Addr a'*], *loc*, *Suc* (*length* (*compE2 e*)), *None*) ([], *loc*[*V* := *Addr a'*], *Suc* (*Suc* (*length* (*compE2 e*))), *None*)
by −(*rule τExecr1step*,*auto simp add*: *exec-move-def intro*: *τmove2-τmoves2.intros exec-instr*)
also from *IH2*[*of loc*[*V* := *Addr a'*]] *len* ‹*call1* {*V*:*Class C*=*None*; *e2*} = ⌊(*a*, *M'*, *vs*)⌋› *cs*
obtain *pc' loc' stk'*
where *exec*: *τExec-mover-a P t e2 h* ([], *loc*[*V* := *Addr a'*], *0*, *None*) (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*)
and *ins*: *pc' < length* (*compE2 e2*) *compE2 e2* ! *pc' = Invoke M'* (*length vs*)

and *bisim'*: P,*e2*,$h \vdash$ (*e2*, *loc*[V := *Addr a'*]) $\leftrightarrow$ (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*) **by** *auto*
from *Try-τExecrI2*[*OF exec*, *of e C V*]
have *τExec-mover-a P t* (*try e catch*(*C V*) *e2*) *h* ([], *loc*[*V* := *Addr a'*], *Suc* (*Suc* (*length* (*compE2 e*))), *None*) (*rev vs* @ *Addr a* # *stk'*, *loc'*, *Suc* (*Suc* (*length* (*compE2 e*) + *pc'*)), *None*) **by** *simp*
also from *bisim'*
have P,*try e catch*(*C V*) *e2*,$h \vdash$ ({*V*:*Class C*=*None*; *e2*}, *loc*[*V* := *Addr a'*]) $\leftrightarrow$ (*rev vs* @ *Addr a* # *stk'*, *loc'*, (*Suc* (*Suc* (*length* (*compE2 e*) + *pc'*))), *None*)
by(*rule bisim1TryCatch2*)
ultimately show *?case* **using** *ins* **by** *fastforce*
next
case *bisim1TryCatch2* **thus** *?case*
by (*auto*)(*fastforce intro*!: *Try-τExecrI2 bisim1-bisims1.bisim1TryCatch2 exI*)+
next
case *bisims1Nil* **thus** *?case* **by** *simp*
next
case (*bisims1List1 e n e' xs stk loc pc es*)
note *IH1* = ‹⟦*call1 e'* = ⌊(*a*, *M'*, *vs*)⌋; *n* + *max-vars e'* ≤ *length xs*; ¬ *contains-insync e*⟧ ⟹ *?concl e n e' xs pc stk loc*›
note *IH2* = ‹⋀*xs*. ⟦*calls1 es* = ⌊(*a*, *M'*, *vs*)⌋; *n* + *max-varss es* ≤ *length xs*; ¬ *contains-insyncs es*⟧ ⟹ *?concls es n es xs 0* [] *xs*›
note *bisim1* = ‹P,*e*,$h \vdash$ (*e'*, *xs*) $\leftrightarrow$ (*stk*, *loc*, *pc*, *None*)›
note *call* = ‹*calls1* (*e'* # *es*) = ⌊(*a*, *M'*, *vs*)⌋›
note *len* = ‹*n* + *max-varss* (*e'* # *es*) ≤ *length xs*›
note *cs* = ‹¬ *contains-insyncs* (*e* # *es*)›
show *?case*
proof(*cases is-val e'*)
case *True*
then obtain *v* **where** [*simp*]: *e'* = *Val v* **by** *auto*
with *bisim1* **have** *τExec-mover-a P t e h* (*stk*, *loc*, *pc*, *None*) ([*v*], *loc*, *length* (*compE2 e*), *None*)
and [*simp*]: *xs* = *loc* **by**(*auto dest*!: *bisim1Val2D1*)
hence *τExec-movesr-a P t* (*e* # *es*) *h* (*stk*, *loc*, *pc*, *None*) ([*v*], *loc*, *length* (*compE2 e*), *None*)
by−(*rule τExec-mover-τExec-movesr*)
also from *call IH2*[*of loc*] *len cs* **obtain** *pc' stk' loc'*
where *exec*: *τExec-movesr-a P t es h* ([], *xs*, *0*, *None*) (*rev vs* @ *Addr a* # *stk'*, *loc'*, *pc'*, *None*)
and *ins*: *compEs2 es* ! *pc'* = *Invoke M'* (*length vs*) *pc'* < *length* (*compEs2 es*)
and *bisim'*: P,*es*,$h \vdash$ (*es*, *xs*) [↔] (*rev vs* @ *Addr a* # *stk'*,*loc'*,*pc'*,*None*) **by** *auto*
from *append-τExec-movesr*[*OF - exec*, *of* [*v*] [*e*]]
have *τExec-movesr-a P t* (*e* # *es*) *h* ([*v*], *loc*, *length* (*compE2 e*), *None*) (*rev vs* @ *Addr a* # (*stk'* @ [*v*]), *loc'*, *length* (*compE2 e*) + *pc'*, *None*)
by *simp*
also (*rtranclp-trans*) **from** *bisim'*
have P,*e* # *es*,$h \vdash$ (*Val v* # *es*, *xs*) [↔]
((*rev vs* @ *Addr a* # *stk'*) @ [*v*],*loc'*,*length* (*compE2 e*) + *pc'*,*None*)
by(*rule bisim1-bisims1.bisims1List2*)
ultimately show *?thesis* **using** *ins* **by** *fastforce*
next
case *False*
with *call IH1 len cs* **show** *?thesis*
by (*auto*)(*fastforce intro*!: *τExec-mover-τExec-movesr bisim1-bisims1.bisims1List1 exI*)+
qed
next
case (*bisims1List2 es n es' xs stk loc pc e v*)
then obtain *pc' stk' loc'* **where** *pc'*: *pc'* < *length* (*compEs2 es*) *compEs2 es* ! *pc'* = *Invoke M'* (*length vs*)

```
    and exec: τExec-movesr-a P t es h (stk, loc, pc, None) (rev vs @ Addr a # stk', loc', pc', None)
    and bisim': P,es,h ⊢ (es', xs) [↔] (rev vs @ Addr a # stk', loc', pc', None) by auto
  note append-τExec-movesr[OF - exec, of [v] [e]]
  moreover from bisim'
  have P,e#es,h ⊢ (Val v# es', xs) [↔] ((rev vs @ Addr a # stk') @ [v],loc',length (compE2 e) +
pc',None)
    by(rule bisim1-bisims1.bisims1List2)
  ultimately show ?case using pc' by fastforce
qed

lemma fixes P :: 'addr J1-prog
  shows bisim1-inline-call-Val:
  ⟦ P,e,h ⊢ (e', xs) ↔ (stk, loc, pc, None); call1 e' = ⌊(a, M, vs)⌋;
     compE2 e ! pc = Invoke M n0 ⟧
    ⟹ length stk ≥ Suc (length vs) ∧ n0 = length vs ∧
       P,e,h ⊢ (inline-call (Val v) e', xs) ↔ (v # drop (Suc (length vs)) stk, loc, Suc pc, None)
  (is ⟦ -; -; - ⟧ ⟹ ?concl e n e' xs pc stk loc)

  and bisims1-inline-calls-Val:
  ⟦ P,es,h ⊢ (es',xs) [↔] (stk,loc,pc,None); calls1 es' = ⌊(a, M, vs)⌋;
     compEs2 es ! pc = Invoke M n0 ⟧
    ⟹ length stk ≥ Suc (length vs) ∧ n0 = length vs ∧
       P,es,h ⊢ (inline-calls (Val v) es', xs) [↔] (v # drop (Suc (length vs)) stk,loc,Suc pc,None)
  (is ⟦ -; -; - ⟧ ⟹ ?concls es n es' xs pc stk loc)
proof(induct (e', xs) (stk, loc, pc, None :: 'addr option)
    and (es', xs) (stk, loc, pc, None :: 'addr option)
    arbitrary: e' xs stk loc pc and es' xs stk loc pc rule: bisim1-bisims1.inducts)
  case bisim1Val2 thus ?case by simp
next
  case bisim1New thus ?case by simp
next
  case bisim1NewArray thus ?case
    by(auto split: if-split-asm dest: bisim1-pc-length-compE2 intro: bisim1-bisims1.bisim1NewArray)
next
  case bisim1Cast thus ?case
    by(auto split: if-split-asm dest: bisim1-pc-length-compE2 intro: bisim1-bisims1.bisim1Cast)
next
  case bisim1InstanceOf thus ?case
    by(auto split: if-split-asm dest: bisim1-pc-length-compE2 intro: bisim1-bisims1.bisim1InstanceOf)
next
  case bisim1Val thus ?case by simp
next
  case bisim1Var thus ?case by simp
next
  case (bisim1BinOp1 e1 e' xs stk loc pc bop e2)
  note IH1 = ‹⟦call1 e' = ⌊(a, M, vs)⌋; compE2 e1 ! pc = Invoke M n0 ⟧ ⟹ ?concl e1 n e' xs pc
stk loc›
  note bisim1 = ‹P,e1,h ⊢ (e', xs) ↔ (stk, loc, pc, None)›
  note call = ‹call1 (e' «bop» e2) = ⌊(a, M, vs)⌋›
  note ins = ‹compE2 (e1 «bop» e2) ! pc = Invoke M n0›
  show ?case
  proof(cases is-val e')
    case False
    with bisim1 call have pc < length (compE2 e1) by(auto intro: bisim1-call-pcD)
```

```
    with call ins False IH1 show ?thesis
      by(auto intro: bisim1-bisims1.bisim1BinOp1)
  next
    case True
    then obtain v where [simp]: e' = Val v by auto
    from bisim1 have pc ≤ length (compE2 e1) by(auto dest: bisim1-pc-length-compE2)
    moreover {
      assume pc: pc < length (compE2 e1)
      with bisim1 ins have False by(auto dest: bisim-Val-pc-not-Invoke) }
    ultimately have [simp]: pc = length (compE2 e1) by(cases pc < length (compE2 e1)) auto
    with ins have False by(simp)
    thus ?thesis ..
  qed
next
  case (bisim1BinOp2 e2 e' xs stk loc pc e1 bop v1)
  note IH2 = ‹⟦call1 e' = ⌊(a, M, vs)⌋; compE2 e2 ! pc = Invoke M n0⟧ ⟹ ?concl e2 n e' xs pc stk
loc›
  note bisim2 = ‹P,e2,h ⊢ (e', xs) ↔ (stk, loc, pc, None)›
  note call = ‹call1 (Val v1 «bop» e') = ⌊(a, M, vs)⌋›
  note ins = ‹compE2 (e1 «bop» e2) ! (length (compE2 e1) + pc) = Invoke M n0›
  from call bisim2 have pc: pc < length (compE2 e2) by(auto intro: bisim1-call-pcD)
  with ins have ins': compE2 e2 ! pc = Invoke M n0 by(simp)
  from IH2 ins' pc call show ?case by(auto dest: bisim1-bisims1.bisim1BinOp2)
next
  case bisim1LAss1 thus ?case
    by(auto split: if-split-asm dest: bisim1-pc-length-compE2 intro: bisim1-bisims1.bisim1LAss1)
next
  case bisim1LAss2 thus ?case by simp
next
  case (bisim1AAcc1 A a' xs stk loc pc i)
  note IH1 = ‹⟦call1 a' = ⌊(a, M, vs)⌋; compE2 A ! pc = Invoke M n0⟧ ⟹ ?concl A n a' xs pc stk
loc›
  note bisim1 = ‹P,A,h ⊢ (a', xs) ↔ (stk, loc, pc, None)›
  note call = ‹call1 (a'⌊i⌉) = ⌊(a, M, vs)⌋›
  note ins = ‹compE2 (A⌊i⌉) ! pc = Invoke M n0›
  show ?case
  proof(cases is-val a')
    case False
    with bisim1 call have pc < length (compE2 A) by(auto intro: bisim1-call-pcD)
    with call ins False IH1 show ?thesis
      by(auto intro: bisim1-bisims1.bisim1AAcc1)
  next
    case True
    then obtain v where [simp]: a' = Val v by auto
    from bisim1 have pc ≤ length (compE2 A) by(auto dest: bisim1-pc-length-compE2)
    moreover {
      assume pc: pc < length (compE2 A)
      with bisim1 ins have False by(auto dest: bisim-Val-pc-not-Invoke) }
    ultimately have [simp]: pc = length (compE2 A) by(cases pc < length (compE2 A)) auto
    with ins have False by(simp)
    thus ?thesis ..
  qed
next
  case (bisim1AAcc2 i i' xs stk loc pc A v)
```

note *IH2* = ‹⟦*call1 i′* = ⌊(*a*, *M*, *vs*)⌋; *compE2 i* ! *pc* = *Invoke M n0*⟧ ⟹ *?concl i n i′ xs pc stk loc*›
note *bisim2* = ‹*P*,*i*,*h* ⊢ (*i′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *call* = ‹*call1* (*Val v*⌊*i′*⌉) = ⌊(*a*, *M*, *vs*)⌋›
note *ins* = ‹*compE2* (*A*⌊*i*⌉) ! (*length* (*compE2 A*) + *pc*) = *Invoke M n0*›
from *call bisim2* **have** *pc*: *pc* < *length* (*compE2 i*) **by**(*auto intro*: *bisim1-call-pcD*)
with *ins* **have** *ins′*: *compE2 i* ! *pc* = *Invoke M n0* **by**(*simp*)
from *IH2 ins′ pc call* **show** *?case*
by(*auto dest*: *bisim1-bisims1.bisim1AAcc2*)
next
case (*bisim1AAss1 A a′ xs stk loc pc i e*)
note *IH1* = ‹⟦*call1 a′* = ⌊(*a*, *M*, *vs*)⌋; *compE2 A* ! *pc* = *Invoke M n0*⟧ ⟹ *?concl A n a′ xs pc stk loc*›
note *bisim1* = ‹*P*,*A*,*h* ⊢ (*a′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *call* = ‹*call1* (*a′*⌊*i*⌉ := *e*) = ⌊(*a*, *M*, *vs*)⌋›
note *ins* = ‹*compE2* (*A*⌊*i*⌉ := *e*) ! *pc* = *Invoke M n0*›
show *?case*
proof(*cases is-val a′*)
case *False*
with *bisim1 call* **have** *pc* < *length* (*compE2 A*) **by**(*auto intro*: *bisim1-call-pcD*)
with *call ins False IH1* **show** *?thesis* **by**(*auto intro*: *bisim1-bisims1.bisim1AAss1*)
next
case *True*
then obtain *v* **where** [*simp*]: *a′* = *Val v* **by** *auto*
from *bisim1* **have** *pc* ≤ *length* (*compE2 A*) **by**(*auto dest*: *bisim1-pc-length-compE2*)
moreover {
assume *pc*: *pc* < *length* (*compE2 A*)
with *bisim1 ins* **have** *False* **by**(*auto dest*: *bisim-Val-pc-not-Invoke*) }
ultimately have [*simp*]: *pc* = *length* (*compE2 A*) **by**(*cases pc* < *length* (*compE2 A*)) *auto*
with *ins* **have** *False* **by**(*simp*)
thus *?thesis* **..**
qed
next
case (*bisim1AAss2 i i′ xs stk loc pc A e v*)
note *IH2* = ‹⟦*call1 i′* = ⌊(*a*, *M*, *vs*)⌋; *compE2 i* ! *pc* = *Invoke M n0*⟧ ⟹ *?concl i n i′ xs pc stk loc*›
note *bisim2* = ‹*P*,*i*,*h* ⊢ (*i′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *call* = ‹*call1* (*Val v*⌊*i′*⌉ := *e*) = ⌊(*a*, *M*, *vs*)⌋›
note *ins* = ‹*compE2* (*A*⌊*i*⌉ := *e*) ! (*length* (*compE2 A*) + *pc*) = *Invoke M n0*›
show *?case*
proof(*cases is-val i′*)
case *False*
with *bisim2 call* **have** *pc*: *pc* < *length* (*compE2 i*) **by**(*auto intro*: *bisim1-call-pcD*)
with *ins* **have** *ins′*: *compE2 i* ! *pc* = *Invoke M n0* **by**(*simp*)
from *IH2 ins′ pc False call* **show** *?thesis* **by**(*auto dest*: *bisim1-bisims1.bisim1AAss2*)
next
case *True*
then obtain *v* **where** [*simp*]: *i′* = *Val v* **by** *auto*
from *bisim2* **have** *pc* ≤ *length* (*compE2 i*) **by**(*auto dest*: *bisim1-pc-length-compE2*)
moreover {
assume *pc*: *pc* < *length* (*compE2 i*)
with *bisim2 ins* **have** *False* **by**(*auto dest*: *bisim-Val-pc-not-Invoke*) }
ultimately have [*simp*]: *pc* = *length* (*compE2 i*) **by**(*cases pc* < *length* (*compE2 i*)) *auto*
with *ins* **have** *False* **by**(*simp*)
thus *?thesis* **..**
qed

next
case (*bisim1AAss3 e e′ xs stk loc pc i A v v′*)
note *IH2* = ‹⟦*call1 e′* = ⌊(*a*, *M*, *vs*)⌋; *compE2 e* ! *pc* = *Invoke M n0*⟧ ⟹ *?concl e n e′ xs pc stk loc›
note *bisim3* = ‹*P*,*e*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *call* = ‹*call1* (*Val v*⌊*Val v′*⌉ := *e′*) = ⌊(*a*, *M*, *vs*)⌋›
note *ins* = ‹*compE2* (*i*⌊*A*⌉ := *e*) ! (*length* (*compE2 i*) + *length* (*compE2 A*) + *pc*) = *Invoke M n0*›
from *call bisim3* **have** *pc*: *pc* < *length* (*compE2 e*) **by**(*auto intro*: *bisim1-call-pcD*)
with *ins* **have** *ins′*: *compE2 e* ! *pc* = *Invoke M n0* **by**(*simp*)
from *IH2 ins′ pc call* **show** *?case* **by**(*auto dest*: *bisim1-bisims1.bisim1AAss3*)
next
case *bisim1AAss4* **thus** *?case* **by** *simp*
next
case *bisim1ALength* **thus** *?case*
by(*auto split*: *if-split-asm dest*: *bisim1-pc-length-compE2 intro*: *bisim1-bisims1.bisim1ALength*)
next
case *bisim1FAcc* **thus** *?case*
by(*auto split*: *if-split-asm dest*: *bisim1-pc-length-compE2 intro*: *bisim1-bisims1.bisim1FAcc*)
next
case (*bisim1FAss1 e1 e′ xs stk loc pc F D e2*)
note *IH1* = ‹⟦*call1 e′* = ⌊(*a*, *M*, *vs*)⌋; *compE2 e1* ! *pc* = *Invoke M n0*⟧ ⟹ *?concl e1 n e′ xs pc stk loc›
note *bisim1* = ‹*P*,*e1*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *call* = ‹*call1* (*e′*·*F*{*D*} := *e2*) = ⌊(*a*, *M*, *vs*)⌋›
note *ins* = ‹*compE2* (*e1*·*F*{*D*} := *e2*) ! *pc* = *Invoke M n0*›
show *?case*
proof(*cases is-val e′*)
case *False*
with *bisim1 call* **have** *pc* < *length* (*compE2 e1*) **by**(*auto intro*: *bisim1-call-pcD*)
with *call ins False IH1* **show** *?thesis*
by(*auto intro*: *bisim1-bisims1.bisim1FAss1*)
next
case *True*
then obtain *v* **where** [*simp*]: *e′* = *Val v* **by** *auto*
from *bisim1* **have** *pc* ≤ *length* (*compE2 e1*) **by**(*auto dest*: *bisim1-pc-length-compE2*)
moreover {
assume *pc*: *pc* < *length* (*compE2 e1*)
with *bisim1 ins* **have** *False* **by**(*auto dest*: *bisim-Val-pc-not-Invoke*) **}**
ultimately have [*simp*]: *pc* = *length* (*compE2 e1*) **by**(*cases pc* < *length* (*compE2 e1*)) *auto*
with *ins* **have** *False* **by**(*simp*)
thus *?thesis* **..**
qed
next
case (*bisim1FAss2 e2 e′ xs stk loc pc e1 F D v1*)
note *IH2* = ‹⟦*call1 e′* = ⌊(*a*, *M*, *vs*)⌋; *compE2 e2* ! *pc* = *Invoke M n0*⟧ ⟹ *?concl e2 n e′ xs pc stk loc›
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *call* = ‹*call1* (*Val v1*·*F*{*D*} := *e′*) = ⌊(*a*, *M*, *vs*)⌋›
note *ins* = ‹*compE2* (*e1*·*F*{*D*} := *e2*) ! (*length* (*compE2 e1*) + *pc*) = *Invoke M n0*›
from *call bisim2* **have** *pc*: *pc* < *length* (*compE2 e2*) **by**(*auto intro*: *bisim1-call-pcD*)
with *ins* **have** *ins′*: *compE2 e2* ! *pc* = *Invoke M n0* **by**(*simp*)
from *IH2 ins′ pc call* **show** *?case* **by**(*auto dest*: *bisim1-bisims1.bisim1FAss2*)
next
case *bisim1FAss3* **thus** *?case* **by** *simp*

next
case (*bisim1CAS1 e1 e′ xs stk loc pc D F e2 E3*)
note *IH1* = ‹⟦*call1 e′* = ⌊(*a*, *M*, *vs*)⌋; *compE2 e1* ! *pc* = *Invoke M n0*⟧ ⟹ *?concl e1 n e′ xs pc stk loc›
note *bisim1* = ‹*P*,*e1*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *call* = ‹*call1* - = ⌊(*a*, *M*, *vs*)⌋›
note *ins* = ‹*compE2* - ! *pc* = *Invoke M n0*›
show *?case*
proof(*cases is-val e′*)
case *False*
with *bisim1 call* **have** *pc* < *length* (*compE2 e1*) **by**(*auto intro*: *bisim1-call-pcD*)
with *call ins False IH1* **show** *?thesis* **by**(*auto intro*: *bisim1-bisims1.bisim1CAS1*)
next
case *True*
then obtain *v* **where** [*simp*]: *e′* = *Val v* **by** *auto*
from *bisim1* **have** *pc* ≤ *length* (*compE2 e1*) **by**(*auto dest*: *bisim1-pc-length-compE2*)
moreover {
assume *pc*: *pc* < *length* (*compE2 e1*)
with *bisim1 ins* **have** *False* **by**(*auto dest*: *bisim-Val-pc-not-Invoke*) }
ultimately have [*simp*]: *pc* = *length* (*compE2 e1*) **by**(*cases pc* < *length* (*compE2 e1*)) *auto*
with *ins* **have** *False* **by**(*simp*)
thus *?thesis* **..**
qed
next
case (*bisim1CAS2 e2 e2′ xs stk loc pc e1 D F e3 v*)
note *IH2* = ‹⟦*call1 e2′* = ⌊(*a*, *M*, *vs*)⌋; *compE2 e2* ! *pc* = *Invoke M n0*⟧ ⟹ *?concl e2 n e2′ xs pc stk loc›
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e2′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *call* = ‹*call1* - = ⌊(*a*, *M*, *vs*)⌋›
note *ins* = ‹*compE2* - ! (*length* (*compE2 e1*) + *pc*) = *Invoke M n0*›
show *?case*
proof(*cases is-val e2′*)
case *False*
with *bisim2 call* **have** *pc*: *pc* < *length* (*compE2 e2*) **by**(*auto intro*: *bisim1-call-pcD*)
with *ins* **have** *ins′*: *compE2 e2* ! *pc* = *Invoke M n0* **by**(*simp*)
from *IH2 ins′ pc False call* **show** *?thesis* **by**(*auto dest*: *bisim1-bisims1.bisim1CAS2*)
next
case *True*
then obtain *v* **where** [*simp*]: *e2′* = *Val v* **by** *auto*
from *bisim2* **have** *pc* ≤ *length* (*compE2 e2*) **by**(*auto dest*: *bisim1-pc-length-compE2*)
moreover {
assume *pc*: *pc* < *length* (*compE2 e2*)
with *bisim2 ins* **have** *False* **by**(*auto dest*: *bisim-Val-pc-not-Invoke*) }
ultimately have [*simp*]: *pc* = *length* (*compE2 e2*) **by**(*cases pc* < *length* (*compE2 e2*)) *auto*
with *ins* **have** *False* **by**(*simp*)
thus *?thesis* **..**
qed
next
case (*bisim1CAS3 e3 e3′ xs stk loc pc e1 D F e2 v v′*)
note *IH2* = ‹⟦*call1 e3′* = ⌊(*a*, *M*, *vs*)⌋; *compE2 e3* ! *pc* = *Invoke M n0*⟧ ⟹ *?concl e3 n e3′ xs pc stk loc›
note *bisim3* = ‹*P*,*e3*,*h* ⊢ (*e3′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *call* = ‹*call1* - = ⌊(*a*, *M*, *vs*)⌋›
note *ins* = ‹*compE2* - ! (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*) = *Invoke M n0*›

from *call bisim3* **have** *pc*: *pc* < *length* (*compE2 e3*) **by**(*auto intro*: *bisim1-call-pcD*)
with *ins* **have** *ins'*: *compE2 e3* ! *pc* = *Invoke M n0* **by**(*simp*)
from *IH2 ins' pc call* **show** *?case* **by**(*auto dest*: *bisim1-bisims1.bisim1CAS3*)
next
case (*bisim1Call1 obj obj' xs stk loc pc M' ps*)
note *IH1* = ‹⟦*call1 obj'* = ⌊(*a*, *M*, *vs*)⌋; *compE2 obj* ! *pc* = *Invoke M n0*⟧ ⟹ *?concl obj n obj' xs pc stk loc*›
note *bisim1* = ‹*P*,*obj*,*h* ⊢ (*obj'*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
note *call* = ‹*call1* (*obj'*·*M'*(*ps*)) = ⌊(*a*, *M*, *vs*)⌋›
note *ins* = ‹*compE2* (*obj*·*M'*(*ps*)) ! *pc* = *Invoke M n0*›
show *?case*
proof(*cases is-val obj'*)
case *False*
with *call bisim1* **have** *pc* < *length* (*compE2 obj*) **by**(*auto intro*: *bisim1-call-pcD*)
with *call False ins IH1 False* **show** *?thesis*
by(*auto intro*: *bisim1-bisims1.bisim1Call1*)
next
case *True*
then obtain *v'* **where** [*simp*]: *obj'* = *Val v'* **by** *auto*
from *bisim1* **have** *pc* ≤ *length* (*compE2 obj*) **by**(*auto dest*: *bisim1-pc-length-compE2*)
moreover {
assume *pc*: *pc* < *length* (*compE2 obj*)
with *bisim1 ins* **have** *False* **by**(*auto dest*: *bisim-Val-pc-not-Invoke*) **}**
ultimately have [*simp*]: *pc* = *length* (*compE2 obj*) **by**(*cases pc* < *length* (*compE2 obj*)) *auto*
with *ins* **have** [*simp*]: *ps* = [] *M'* = *M*
by(*auto split*: *if-split-asm*)(*auto simp add*: *neq-Nil-conv*)
from *ins call* **have** [*simp*]: *vs* = [] **by**(*auto split*: *if-split-asm*)
with *bisim1* **have** [*simp*]: *stk* = [*v'*] *xs* = *loc* **by**(*auto dest*: *bisim1-pc-length-compE2D*)
from *bisim1Val2*[*of length* (*compE2* (*obj*·*M*([]))) *obj*·*M*([]) *P h v loc*] *call ins*
show *?thesis* **by**(*auto simp add*: *is-val-iff*)
qed
next
case (*bisim1CallParams ps ps' xs stk loc pc obj M' v'*)
note *IH2* = ‹⟦*calls1 ps'* = ⌊(*a*, *M*, *vs*)⌋; *compEs2 ps* ! *pc* = *Invoke M n0*⟧ ⟹ *?concls ps n ps' xs pc stk loc*›
note *bisim* = ‹*P*,*ps*,*h* ⊢ (*ps'*, *xs*) [↔] (*stk*, *loc*, *pc*, *None*)›
note *call* = ‹*call1* (*Val v'*·*M'*(*ps'*)) = ⌊(*a*, *M*, *vs*)⌋›
note *ins* = ‹*compE2* (*obj*·*M'*(*ps*)) ! (*length* (*compE2 obj*) + *pc*) = *Invoke M n0*›
from *call* **show** *?case*
proof(*cases rule*: *call1-callE*)
case *CallObj* **thus** *?thesis* **by** *simp*
next
case (*CallParams v''*)
hence [*simp*]: *v''* = *v'* **and** *call'*: *calls1 ps'* = ⌊(*a*, *M*, *vs*)⌋ **by** *simp-all*
from *bisim call'* **have** *pc*: *pc* < *length* (*compEs2 ps*) **by**(*rule bisims1-calls-pcD*)
with *ins* **have** *ins'*: *compEs2 ps* ! *pc* = *Invoke M n0* **by**(*simp*)
with *IH2 call' ins pc*
have *P*,*ps*,*h* ⊢ (*inline-calls* (*Val v*) *ps'*, *xs*)
[↔] (*v* # *drop* (*Suc* (*length vs*)) *stk*, *loc*, *Suc pc*, *None*)
and *len*: *Suc* (*length vs*) ≤ *length stk* **and** *n0*: *n0* = *length vs* **by** *auto*
hence *P*,*obj*·*M'*(*ps*),*h* ⊢ (*Val v'*·*M'*(*inline-calls* (*Val v*) *ps'*), *xs*)
↔ ((*v* # *drop* (*Suc* (*length vs*)) *stk*) @ [*v'*], *loc*, *length* (*compE2 obj*) + *Suc pc*, *None*)
by−(*rule bisim1-bisims1.bisim1CallParams*)

thus *?thesis* **using** *call′ len n0* **by**(*auto simp add*: *is-vals-conv*)
next
case *Call*
hence [*simp*]: *v′* = *Addr a M′* = *M ps′* = *map Val vs* **by** *auto*
from *bisim* **have** *pc* ≤ *length* (*compEs2 ps*) **by**(*auto dest*: *bisims1-pc-length-compEs2*)
moreover {
assume *pc*: *pc* < *length* (*compEs2 ps*)
with *bisim ins* **have** *False* **by**(*auto dest*: *bisims-Val-pc-not-Invoke*) }
ultimately have [*simp*]: *pc* = *length* (*compEs2 ps*) **by**(*cases pc* < *length* (*compEs2 ps*)) *auto*
from *bisim* **have** [*simp*]: *stk* = *rev vs xs* = *loc* **by**(*auto dest*: *bisims1-Val-length-compEs2D*)
hence *P*,*obj*·*M*(*ps*),*h* ⊢ (*Val v*, *loc*) ↔ ([*v*], *loc*, *length* (*compE2* (*obj*·*M*(*ps*))), *None*) **by**−(*rule bisim1Val2*, *simp*)
moreover from *bisim* **have** *length ps* = *length ps′* **by**(*rule bisims1-lengthD*)
ultimately show *?thesis* **using** *ins* **by**(*auto*)
qed
next
case *bisim1BlockSome1* **thus** *?case* **by** *simp*
next
case *bisim1BlockSome2* **thus** *?case* **by** *simp*
next
case *bisim1BlockSome4* **thus** *?case*
by(*auto intro*: *bisim1-bisims1.bisim1BlockSome4*)
next
case *bisim1BlockNone* **thus** *?case*
by(*auto intro*: *bisim1-bisims1.bisim1BlockNone*)
next
case *bisim1Sync1* **thus** *?case*
by(*auto split*: *if-split-asm dest*: *bisim1-pc-length-compE2 intro*: *bisim1-bisims1.bisim1Sync1*)
next
case *bisim1Sync2* **thus** *?case* **by** *simp*
next
case *bisim1Sync3* **thus** *?case* **by** *simp*
next
case *bisim1Sync4* **thus** *?case*
by(*auto split*: *if-split-asm dest*: *bisim1-pc-length-compE2 bisim1-bisims1.bisim1Sync4*)
next
case *bisim1Sync5* **thus** *?case* **by** *simp*
next
case *bisim1Sync6* **thus** *?case* **by** *simp*
next
case *bisim1Sync7* **thus** *?case* **by** *simp*
next
case *bisim1Sync8* **thus** *?case* **by** *simp*
next
case *bisim1Sync9* **thus** *?case* **by** *simp*
next
case *bisim1InSync* **thus** *?case* **by**(*simp*)
next
case *bisim1Seq1* **thus** *?case*
by(*auto split*: *if-split-asm dest*: *bisim1-pc-length-compE2 intro*: *bisim1-bisims1.bisim1Seq1*)
next
case *bisim1Seq2* **thus** *?case*
by(*auto split*: *if-split-asm dest*: *bisim1-pc-length-compE2*)(*fastforce dest*: *bisim1-bisims1.bisim1Seq2*)
next

```
  case bisim1Cond1 thus ?case
    by(auto split: if-split-asm dest: bisim1-pc-length-compE2 intro: bisim1-bisims1.bisim1Cond1)
next
  case (bisim1CondThen e1 stk loc pc e e2 e' xs) thus ?case
    by(auto split: if-split-asm dest: bisim1-pc-length-compE2)
      (fastforce dest: bisim1-bisims1.bisim1CondThen[where e=e and ?e2.0=e2])
next
  case (bisim1CondElse e2 stk loc pc e e1 e' xs) thus ?case
    by(auto split: if-split-asm dest: bisim1-pc-length-compE2)
      (fastforce dest: bisim1-bisims1.bisim1CondElse[where e=e and ?e1.0=e1])
next
  case bisim1While1 thus ?case by simp
next
  case bisim1While3 thus ?case
    by(auto split: if-split-asm dest: bisim1-pc-length-compE2 intro: bisim1-bisims1.bisim1While3)
next
  case bisim1While4 thus ?case
   by(auto split: if-split-asm dest: bisim1-pc-length-compE2)(fastforce dest: bisim1-bisims1.bisim1While4)
next
  case bisim1While6 thus ?case by simp
next
  case bisim1While7 thus ?case by simp
next
  case bisim1Throw1 thus ?case
    by(auto split: if-split-asm dest: bisim1-pc-length-compE2 intro: bisim1-bisims1.bisim1Throw1)
next
  case bisim1Try thus ?case
    by(auto split: if-split-asm dest: bisim1-pc-length-compE2 intro: bisim1-bisims1.bisim1Try)
next
  case bisim1TryCatch1 thus ?case by simp
next
  case bisim1TryCatch2 thus ?case
    by(fastforce dest: bisim1-bisims1.bisim1TryCatch2)
next
  case bisims1Nil thus ?case by simp
next
  case (bisims1List1 e e' xs stk loc pc es)
  note IH1 = ‹⟦call1 e' = ⌊(a, M, vs)⌋; compE2 e ! pc = Invoke M n0⟧ ⟹ ?concl e n e' xs pc stk
loc›
  note bisim1 = ‹P,e,h ⊢ (e', xs) ↔ (stk, loc, pc, None)›
  note call = ‹calls1 (e' # es) = ⌊(a, M, vs)⌋›
  note ins = ‹compEs2 (e # es) ! pc = Invoke M n0›
  show ?case
  proof(cases is-val e')
    case False
    with bisim1 call have pc < length (compE2 e) by(auto intro: bisim1-call-pcD)
    with call ins False IH1 show ?thesis
      by(auto intro: bisim1-bisims1.bisims1List1)
  next
    case True
    then obtain v where [simp]: e' = Val v by auto
    from bisim1 have pc ≤ length (compE2 e) by(auto dest: bisim1-pc-length-compE2)
    moreover {
      assume pc: pc < length (compE2 e)
```

```
    with bisim1 ins have False by(auto dest: bisim-Val-pc-not-Invoke) }
   ultimately have [simp]: pc = length (compE2 e) by(cases pc < length (compE2 e)) auto
   with ins call have False by(cases es)(auto)
   thus ?thesis ..
 qed
next
 case (bisims1List2 es es' xs stk loc pc e v')
 note IH = ‹⟦calls1 es' = ⌊(a, M, vs)⌋; compEs2 es ! pc = Invoke M n0⟧ ⟹ ?concls es n es' xs pc
stk loc›
 note call = ‹calls1 (Val v' # es') = ⌊(a, M, vs)⌋›
 note bisim = ‹P,es,h ⊢ (es', xs) [↔] (stk, loc, pc, None)›
 note ins = ‹compEs2 (e # es) ! (length (compE2 e) + pc) = Invoke M n0›
 from call have call': calls1 es' = ⌊(a, M, vs)⌋ by simp
 with bisim have pc: pc < length (compEs2 es) by(rule bisims1-calls-pcD)
 with ins have ins': compEs2 es ! pc = Invoke M n0 by(simp)
 from IH call ins pc show ?case
  by(auto split: if-split-asm dest: bisim1-bisims1.bisims1List2)
qed

lemma bisim1-fv: P,e,h ⊢ (e', xs) ↔ s ⟹ fv e' ⊆ fv e
 and bisims1-fvs: P,es,h ⊢ (es', xs) [↔] s ⟹ fvs es' ⊆ fvs es
apply(induct (e', xs) s and (es', xs) s arbitrary: e' xs and es' xs rule: bisim1-bisims1.inducts)
apply(auto)
done


lemma bisim1-syncvars: ⟦ P,e,h ⊢ (e', xs) ↔ s; syncvars e ⟧ ⟹ syncvars e'
 and bisims1-syncvarss: ⟦ P,es,h ⊢ (es', xs) [↔] s; syncvarss es ⟧ ⟹ syncvarss es'
apply(induct (e', xs) s and (es', xs) s arbitrary: e' xs and es' xs rule: bisim1-bisims1.inducts)
apply(auto dest: bisim1-fv)
done

declare pcs-stack-xlift [simp]

lemma bisim1-Val-τred1r:
 ⟦ P, E, h ⊢ (e, xs) ↔ ([v], loc, length (compE2 E), None); n + max-vars e ≤ length xs; ℬ E n ⟧
 ⟹ τred1r P t h (e, xs) (Val v, loc)

 and bisims1-Val-τReds1r:
 ⟦ P, Es, h ⊢ (es, xs) [↔] (rev vs, loc, length (compEs2 Es), None); n + max-varss es ≤ length xs;
ℬs Es n ⟧
  ⟹ τreds1r P t h (es, xs) (map Val vs, loc)
proof(induct E n e xs stk≡[v] loc pc≡length (compE2 E) xcp≡None::'addr option
       and Es n es xs stk≡rev vs loc pc≡length (compEs2 Es) xcp≡None::'addr option
     arbitrary: v and vs rule: bisim1-bisims1-inducts-split)
 case bisim1BlockSome2 thus ?case by(simp (no-asm-use))
next
 case (bisim1BlockSome4 e n e' xs loc pc V T val)
 from ‹ℬ {V:T=⌊val⌋; e} n› have [simp]: n = V and ℬ e (Suc n) by auto
 note len = ‹n + max-vars {V:T=None; e'} ≤ length xs›
 hence V: V < length xs by simp
 from ‹P,e,h ⊢ (e', xs) ↔ ([v], loc, pc, None)›
 have lenxs: length xs = length loc by(auto dest: bisim1-length-xs)
 note IH = ‹⟦pc = length (compE2 e); Suc n + max-vars e' ≤ length xs; ℬ e (Suc n)⟧
```

```
          ⟹ τred1r P t h (e', xs) (Val v, loc)›
  with len ‹Suc (Suc pc) = length (compE2 {V:T=⌊val⌋; e})› ‹B e (Suc n)›
  have τred1r P t h (e', xs) (Val v, loc) by(simp)
  hence τred1r P t h ({V:T=None; e'}, xs) ({V:T=None; Val v}, loc)
    by(rule Block-None-τred1r-xt)
 thus ?case using V lenxs by(auto elim!: rtranclp.rtrancl-into-rtrancl intro: Red1Block τmove1BlockRed)
next
  case (bisim1BlockNone e n e' xs loc V T)
  from ‹B {V:T=None; e} n› have [simp]: n = V and B e (Suc n) by auto
  note len = ‹n + max-vars {V:T=None; e'} ≤ length xs›
  hence V: V < length xs by simp
  from ‹P,e,h ⊢ (e', xs) ↔ ([v], loc, length (compE2 {V:T=None; e}), None)›
  have lenxs: length xs = length loc by(auto dest: bisim1-length-xs)
  note IH = ‹⟦length (compE2 {V:T=None; e}) = length (compE2 e); Suc n + max-vars e' ≤ length
xs; B e (Suc n) ⟧
          ⟹ τred1r P t h (e', xs) (Val v, loc)›
  with len ‹B e (Suc n)› have τred1r P t h (e', xs) (Val v, loc) by(simp)
  hence τred1r P t h ({V:T=None; e'}, xs) ({V:T=None; Val v}, loc)
    by(rule Block-None-τred1r-xt)
 thus ?case using V lenxs by(auto elim!: rtranclp.rtrancl-into-rtrancl intro: Red1Block τmove1BlockRed)
next
  case (bisim1TryCatch2 e2 n e' xs loc pc e C V)
  from ‹B (try e catch(C V) e2) n› have [simp]: n = V and B e2 (Suc n) by auto
  note len = ‹n + max-vars {V:Class C=None; e'} ≤ length xs›
  hence V: V < length xs by simp
  from ‹P,e2,h ⊢ (e', xs) ↔ ([v], loc, pc, None)›
  have lenxs: length xs = length loc by(auto dest: bisim1-length-xs)
  note IH = ‹⟦pc = length (compE2 e2); Suc n + max-vars e' ≤ length xs; B e2 (Suc n)⟧
          ⟹ τred1r P t h (e', xs) (Val v, loc)›
  with len ‹Suc (Suc (length (compE2 e) + pc)) = length (compE2 (try e catch(C V) e2))› ‹B e2
(Suc n)›
  have τred1r P t h (e', xs) (Val v, loc) by(simp)
  hence τred1r P t h ({V:Class C=None; e'}, xs) ({V:Class C=None; Val v}, loc)
    by(rule Block-None-τred1r-xt)
 thus ?case using V lenxs by(auto elim!: rtranclp.rtrancl-into-rtrancl intro: Red1Block τmove1BlockRed)
next
  case (bisims1List1 e n e' xs loc es)
  note bisim = ‹P,e,h ⊢ (e', xs) ↔ (rev vs, loc, length (compEs2 (e # es)), None)›
  then have es: es = [] and pc: length (compEs2 (e # es)) = length (compE2 e)
    by(auto dest: bisim1-pc-length-compE2)
  with bisim obtain val where stk: rev vs = [val] and e': is-val e' ⟹ e' = Val val
    by(auto dest: bisim1-pc-length-compE2D)
  with es pc bisims1List1 have τred1r P t h (e', xs) (Val val, loc) by simp
  with stk es show ?case by(auto intro: τred1r-inj-τreds1r)
next
  case (bisims1List2 es n es' xs stk loc pc e v)
  from ‹stk @ [v] = rev vs› obtain vs' where vs: vs = v # vs' by(cases vs) auto
  with bisims1List2 show ?case by(auto intro: τreds1r-cons-τreds1r)
qed(fastforce dest: bisim1-pc-length-compE2 bisims1-pc-length-compEs2)+

lemma exec-meth-stk-split:
  ⟦ P,E,h ⊢ (e, xs) ↔ (stk, loc, pc, xcp);
    exec-meth-d (compP2 P) (compE2 E) (stack-xlift (length STK) (compxE2 E 0 0)) t
              h (stk @ STK, loc, pc, xcp) ta h' (stk', loc', pc', xcp') ⟧
```

$\Longrightarrow \exists stk''.\ stk' = stk'' @ STK \wedge exec\text{-}meth\text{-}d\ (compP2\ P)\ (compE2\ E)\ (compxE2\ E\ 0\ 0)\ t$
$h\ (stk, loc, pc, xcp)\ ta\ h'\ (stk'', loc', pc', xcp')$
(**is** $[\![$ -; *?exec E stk STK loc pc xcp stk' loc' pc' xcp'* $]\!] \Longrightarrow$ *?concl E stk STK loc pc xcp stk' loc' pc' xcp'*)

and *exec-meth-stk-splits*:
$[\![\ P,Es,h \vdash (es,xs)\ [\leftrightarrow]\ (stk,loc,pc,xcp);$
exec-meth-d (compP2 P) (compEs2 Es) (stack-xlift (length STK) (compxEs2 Es 0 0)) t
$h\ (stk @ STK, loc, pc, xcp)\ ta\ h'\ (stk', loc', pc', xcp')\ ]\!]$
$\Longrightarrow \exists stk''.\ stk' = stk'' @ STK \wedge exec\text{-}meth\text{-}d\ (compP2\ P)\ (compEs2\ Es)\ (compxEs2\ Es\ 0\ 0)\ t$
$h\ (stk, loc, pc, xcp)\ ta\ h'\ (stk'', loc', pc', xcp')$
(**is** $[\![$ -; *?execs Es stk STK loc pc xcp stk' loc' pc' xcp'* $]\!] \Longrightarrow$ *?concls Es stk STK loc pc xcp stk' loc' pc' xcp'*)
proof(*induct E n* :: *nat e xs stk loc pc xcp* **and** *Es n* :: *nat es xs stk loc pc xcp*
arbitrary: *stk' loc' pc' xcp' STK* **and** *stk' loc' pc' xcp' STK rule*: *bisim1-bisims1-inducts-split*)
case *bisim1InSync* **thus** *?case* **by**(*auto elim*!: *exec-meth.cases intro*!: *exec-meth.intros*)
next
case *bisim1Val2* **thus** *?case* **by**(*auto dest*: *exec-meth-length-compE2-stack-xliftD*)
next
case *bisim1New* **thus** *?case*
by (*fastforce elim*: *exec-meth.cases intro*: *exec-meth.intros split*: *if-split-asm cong del*: *image-cong-simp*)
next
case *bisim1NewThrow* **thus** *?case* **by**(*fastforce elim*: *exec-meth.cases intro*: *exec-meth.intros*)
next
case (*bisim1NewArray e n e' xs stk loc pc xcp T*)
note *bisim* = ‹*P,e,h* $\vdash$ (*e', xs*) $\leftrightarrow$ (*stk, loc, pc, xcp*)›
note *IH* = ‹$\bigwedge$*stk' loc' pc' xcp' STK. ?exec e stk STK loc pc xcp stk' loc' pc' xcp'*
$\Longrightarrow$ *?concl e stk STK loc pc xcp stk' loc' pc' xcp'*›
note *exec* = ‹*?exec* (*newA T*⌊*e*⌉) *stk STK loc pc xcp stk' loc' pc' xcp'*›
from *bisim* **have** *pc*: *pc* $\leq$ *length* (*compE2 e*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* $<$ *length* (*compE2 e*))
case *True*
with *exec* **have** *?exec e stk STK loc pc xcp stk' loc' pc' xcp'*
by(*simp add*: *compxE2-size-convs*)(*erule exec-meth-take*)
from *IH*[*OF this*] **show** *?thesis* **by** *auto*
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compE2 e*) **by** *simp*
with *bisim* **obtain** *v* **where** [*simp*]: *stk* = [*v*] *xcp* = *None*
by(*auto dest*: *dest*: *bisim1-pc-length-compE2D*)
with *exec* **show** *?thesis*
apply *simp*
apply (*erule exec-meth.cases*)
apply (*auto 4 4 intro*: *exec-meth.intros split*: *if-split-asm cong del*: *image-cong-simp*)
done
qed
next
case (*bisim1NewArrayThrow e n a xs stk loc pc T*)
note *bisim* = ‹*P,e,h* $\vdash$ (*Throw a, xs*) $\leftrightarrow$ (*stk, loc, pc,* ⌊*a*⌋)›
note *IH* = ‹$\bigwedge$*stk' loc' pc' xcp' STK. ?exec e stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*
$\Longrightarrow$ *?concl e stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*›
note *exec* = ‹*?exec* (*newA T*⌊*e*⌉) *stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*›
from *bisim* **have** *pc*: *pc* $<$ *length* (*compE2 e*) **and** [*simp*]: *xs* = *loc*

```
    by(auto dest: bisim1-ThrowD)
  from exec have ?exec e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'
    by(simp)(erule exec-meth-take[OF - pc])
  from IH[OF this] show ?case by(auto)
next
  case bisim1NewArrayFail thus ?case
   by(auto elim!: exec-meth.cases dest: match-ex-table-pcsD simp add: stack-xlift-compxEs2 stack-xlift-compxE2)
next
  case (bisim1Cast e n e' xs stk loc pc xcp T)
  note bisim = ‹P,e,h ⊢ (e', xs) ↔ (stk, loc, pc, xcp)›
  note IH = ‹⋀stk' loc' pc' xcp' STK. ?exec e stk STK loc pc xcp stk' loc' pc' xcp'
          ⟹ ?concl e stk STK loc pc xcp stk' loc' pc' xcp'›
  note exec = ‹?exec (Cast T e) stk STK loc pc xcp stk' loc' pc' xcp'›
  from bisim have pc: pc ≤ length (compE2 e) by(rule bisim1-pc-length-compE2)
  show ?case
  proof(cases pc < length (compE2 e))
    case True
    with exec have ?exec e stk STK loc pc xcp stk' loc' pc' xcp'
      by(simp add: compxE2-size-convs)(erule exec-meth-take)
    from IH[OF this] show ?thesis by auto
  next
    case False
    with pc have [simp]: pc = length (compE2 e) by simp
    with bisim obtain v where [simp]: stk = [v] xcp = None
      by(auto dest: dest: bisim1-pc-length-compE2D)
    with exec show ?thesis apply(simp)
      by(erule exec-meth.cases)(auto intro!: exec-meth.intros split: if-split-asm)
  qed
next
  case (bisim1CastThrow e n a xs stk loc pc T)
  note bisim = ‹P,e,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
  note IH = ‹⋀stk' loc' pc' xcp' STK. ?exec e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'
          ⟹ ?concl e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'›
  note exec = ‹?exec (Cast T e) stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'›
  from bisim have pc: pc < length (compE2 e) and [simp]: xs = loc
    by(auto dest: bisim1-ThrowD)
  from exec have ?exec e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'
    by(simp)(erule exec-meth-take[OF - pc])
  from IH[OF this] show ?case by(auto)
next
  case bisim1CastFail thus ?case
   by(auto elim!: exec-meth.cases dest: match-ex-table-pcsD simp add: stack-xlift-compxEs2 stack-xlift-compxE2)
next
  case (bisim1InstanceOf e n e' xs stk loc pc xcp T)
  note bisim = ‹P,e,h ⊢ (e', xs) ↔ (stk, loc, pc, xcp)›
  note IH = ‹⋀stk' loc' pc' xcp' STK. ?exec e stk STK loc pc xcp stk' loc' pc' xcp'
          ⟹ ?concl e stk STK loc pc xcp stk' loc' pc' xcp'›
  note exec = ‹?exec (e instanceof T) stk STK loc pc xcp stk' loc' pc' xcp'›
  from bisim have pc: pc ≤ length (compE2 e) by(rule bisim1-pc-length-compE2)
  show ?case
  proof(cases pc < length (compE2 e))
    case True
    with exec have ?exec e stk STK loc pc xcp stk' loc' pc' xcp'
      by(simp add: compxE2-size-convs)(erule exec-meth-take)
```

```
    from IH[OF this] show ?thesis by auto
  next
    case False
    with pc have [simp]: pc = length (compE2 e) by simp
    with bisim obtain v where [simp]: stk = [v] xcp = None
      by(auto dest: dest: bisim1-pc-length-compE2D)
    with exec show ?thesis apply(simp)
      by(erule exec-meth.cases)(auto intro!: exec-meth.intros split: if-split-asm)
  qed
next
  case (bisim1InstanceOfThrow e n a xs stk loc pc T)
  note bisim = ‹P,e,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
  note IH = ‹⋀stk' loc' pc' xcp' STK. ?exec e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'
          ⟹ ?concl e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'›
  note exec = ‹?exec (e instanceof T) stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'›
  from bisim have pc: pc < length (compE2 e) and [simp]: xs = loc
    by(auto dest: bisim1-ThrowD)
  from exec have ?exec e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'
    by(simp)(erule exec-meth-take[OF - pc])
  from IH[OF this] show ?case by(auto)
next
  case bisim1Val thus ?case by(fastforce elim: exec-meth.cases intro: exec-meth.intros)
next
  case bisim1Var thus ?case by(fastforce elim: exec-meth.cases intro: exec-meth.intros)
next
  case (bisim1BinOp1 e1 n e1' xs stk loc pc xcp e2 bop)
  note IH1 = ‹⋀stk' loc' pc' xcp' STK. ?exec e1 stk STK loc pc xcp stk' loc' pc' xcp'
          ⟹ ?concl e1 stk STK loc pc xcp stk' loc' pc' xcp'›
  note IH2 = ‹⋀xs stk' loc' pc' xcp' STK. ?exec e2 [] STK xs 0 None stk' loc' pc' xcp'
          ⟹ ?concl e2 [] STK xs 0 None stk' loc' pc' xcp'›
  note bisim1 = ‹P,e1,h ⊢ (e1', xs) ↔ (stk, loc, pc, xcp)›
  note bisim2 = ‹P,e2,h ⊢ (e2, loc) ↔ ([], loc, 0, None)›
  note exec = ‹?exec (e1 «bop» e2) stk STK loc pc xcp stk' loc' pc' xcp'›
  from bisim1 have pc: pc ≤ length (compE2 e1) by(rule bisim1-pc-length-compE2)
  show ?case
  proof(cases pc < length (compE2 e1))
    case True
    with exec have ?exec e1 stk STK loc pc xcp stk' loc' pc' xcp'
      by(simp add: compxE2-size-convs)(erule exec-meth-take-xt)
    from IH1[OF this] show ?thesis by auto
  next
    case False
    with pc have pc: pc = length (compE2 e1) by simp
    with exec have pc' ≥ length (compE2 e1)
    by(simp add: compxE2-size-convs stack-xlift-compxE2)(auto split: bop.splits elim!: exec-meth-drop-xt-pc)
    then obtain PC where PC: pc' = PC + length (compE2 e1)
      by −(rule-tac PC34=pc' − length (compE2 e1) in that, simp)
    from pc bisim1 obtain v where stk = [v] xcp = None by(auto dest: bisim1-pc-length-compE2D)
    with exec pc have exec-meth-d (compP2 P) (compE2 e1 @ compE2 e2)
   (stack-xlift (length STK) (compxE2 e1 0 0 @ compxE2 e2 (length (compE2 e1)) (Suc 0))) t h (stk
@ STK, loc, length (compE2 e1) + 0, xcp) ta h' (stk', loc', pc', xcp')
      by−(rule exec-meth-take, auto)
    hence ?exec e2 [] (v # STK) loc 0 None stk' loc' (pc' − length (compE2 e1)) xcp'
      using ‹stk = [v]› ‹xcp = None›
```

by −(*rule exec-meth-drop-xt*, *auto simp add*: *stack-xlift-compxE2 shift-compxE2*)
from *IH2*[*OF this*] *PC* **obtain** *stk''* **where** *stk'*: *stk' = stk'' @ v # STK*
and *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t h* ([], *loc*, *0*, *None*) *ta h'* (*stk''*, *loc'*, *PC*, *xcp'*) **by** *auto*
hence *exec-meth-d* (*compP2 P*) ((*compE2 e1* @ *compE2 e2*) @ [*BinOpInstr bop*])
(*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length* [*v*]) (*compxE2 e2 0 0*))) *t h*
([] @ [*v*], *loc*, *length* (*compE2 e1*) + *0*, *None*) *ta h'* (*stk''* @ [*v*], *loc'*, *length* (*compE2 e1*) + *PC*, *xcp'*)
apply −
apply(*rule exec-meth-append*)
apply(*rule append-exec-meth-xt*)
apply(*erule exec-meth-stk-offer*)
by(*auto*)
thus *?thesis* **using** ‹*stk* = [*v*]› ‹*xcp* = *None*› *stk' pc PC*
by(*clarsimp simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps*)
qed
next
case (*bisim1BinOp2 e2 n e2' xs stk loc pc xcp e1 bop v1*)
note *IH2* = ‹⋀*stk' loc' pc' xcp' STK*. *?exec e2 stk STK loc pc xcp stk' loc' pc' xcp'*
⟹ *?concl e2 stk STK loc pc xcp stk' loc' pc' xcp'*›
note *bisim1* = ‹*P*,*e1*,*h* ⊢ (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e2'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *exec* = ‹*?exec* (*e1* «*bop*» *e2*) (*stk* @ [*v1*]) *STK loc* (*length* (*compE2 e1*) + *pc*) *xcp stk' loc' pc' xcp'*›
from *bisim2* **have** *pc*: *pc* ≤ *length* (*compE2 e2*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e2*))
case *True*
from *exec* **have** *exec-meth-d* (*compP2 P*) ((*compE2 e1* @ *compE2 e2*) @ [*BinOpInstr bop*])
(*stack-xlift* (*length STK*) (*compxE2 e1 0 0*) @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length STK*) (*compxE2 e2 0* (*Suc 0*)))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*) **by**(*simp add*: *compxE2-size-convs*)
hence *exec'*: *exec-meth-d* (*compP2 P*) (*compE2 e1* @ *compE2 e2*) (*stack-xlift* (*length STK*) (*compxE2 e1 0 0*) @
shift (*length* (*compE2 e1*)) (*stack-xlift* (*length STK*) (*compxE2 e2 0* (*Suc 0*)))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*rule exec-meth-take*)(*simp add*: *True*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*stack-xlift* (*length STK*) (*compxE2 e2 0* (*Suc 0*))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 e1*), *xcp'*)
by(*rule exec-meth-drop-xt*)(*auto simp add*: *stack-xlift-compxE2*)
hence *?exec e2 stk* (*v1* # *STK*) *loc pc xcp stk' loc'* (*pc'* − *length* (*compE2 e1*)) *xcp'*
by(*simp add*: *compxE2-stack-xlift-convs*)
from *IH2*[*OF this*] **obtain** *stk''* **where** *stk'*: *stk' = stk'' @ v1 # STK*
and *exec''*: *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 e1*), *xcp'*) **by** *blast*
from *exec''* **have** *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*stack-xlift* (*length* [*v1*]) (*compxE2 e2 0 0*)) *t h* (*stk* @ [*v1*], *loc*, *pc*, *xcp*)
ta h' (*stk''* @ [*v1*], *loc'*, *pc'* − *length* (*compE2 e1*), *xcp'*)
by(*rule exec-meth-stk-offer*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e1* @ *compE2 e2*) (*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length* [*v1*]) (*compxE2 e2 0 0*))) *t h* (*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*)
ta h' (*stk''* @ [*v1*], *loc'*, *length* (*compE2 e1*) + (*pc'* − *length* (*compE2 e1*)), *xcp'*)

by(*rule append-exec-meth-xt*) *auto*
hence *exec-meth-d* (*compP2 P*) ((*compE2 e1* @ *compE2 e2*) @ [*BinOpInstr bop*]) (*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length* [*v1*]) (*compxE2 e2 0 0*))) *t h* (*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*)
ta h′ (*stk″* @ [*v1*], *loc′*, *length* (*compE2 e1*) + (*pc′* − *length* (*compE2 e1*)), *xcp′*)
by(*rule exec-meth-append*)
moreover from *exec′* **have** *pc′* ≥ *length* (*compE2 e1*)
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
ultimately show *?thesis* **using** *stk′* **by**(*simp add*: *stack-xlift-compxE2 shift-compxE2*)
next
case *False*
with *pc* **have** *pc*: *pc* = *length* (*compE2 e2*) **by** *simp*
with *bisim2* **obtain** *v2* **where** [*simp*]: *stk* = [*v2*] *xcp* = *None*
by(*auto dest*: *dest*: *bisim1-pc-length-compE2D*)
with *exec pc* **show** *?thesis*
by(*fastforce elim*: *exec-meth.cases split*: *sum.split-asm intro*!: *exec-meth.intros*)
qed
next
case (*bisim1BinOpThrow1 e1 n a xs stk loc pc e2 bop*)
note *bisim1* = ‹*P*,*e1*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
note *IH1* = ‹⋀*stk′ loc′ pc′ xcp′ STK*. *?exec e1 stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*
⟹ *?concl e1 stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
note *exec* = ‹*?exec* (*e1* «*bop*» *e2*) *stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
from *bisim1* **have** *pc*: *pc* < *length* (*compE2 e1*) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-ThrowD*)
from *exec* **have** *exec-meth-d* (*compP2 P*) (*compE2 e1* @ (*compE2 e2* @ [*BinOpInstr bop*]))
(*stack-xlift* (*length STK*) (*compxE2 e1 0 0*) @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length STK*) (*compxE2 e2 0* (*Suc 0*)))) *t*
h (*stk* @ *STK*, *loc*, *pc*, ⌊*a*⌋) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*) **by**(*simp add*: *compxE2-size-convs*)
hence *?exec e1 stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*
by(*rule exec-meth-take-xt*)(*rule pc*)
from *IH1*[*OF this*] **show** *?case* **by**(*auto*)
next
case (*bisim1BinOpThrow2 e2 n a xs stk loc pc e1 bop v1*)
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
note *IH2* = ‹⋀*stk′ loc′ pc′ xcp′ STK*. *?exec e2 stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*
⟹ *?concl e2 stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
note *exec* = ‹*?exec* (*e1* «*bop*» *e2*) (*stk* @ [*v1*]) *STK loc* (*length* (*compE2 e1*) + *pc*) ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
from *bisim2* **have** *pc*: *pc* < *length* (*compE2 e2*) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-ThrowD*)
from *exec* **have** *exec-meth-d* (*compP2 P*) ((*compE2 e1* @ *compE2 e2*) @ [*BinOpInstr bop*])
(*stack-xlift* (*length STK*) (*compxE2 e1 0 0*) @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length STK*) (*compxE2 e2 0* (*Suc 0*))))
t h (*stk* @ *v1* # *STK*, *loc*, *length* (*compE2 e1*) + *pc*, ⌊*a*⌋) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *compxE2-size-convs*)
hence *exec′*: *exec-meth-d* (*compP2 P*) (*compE2 e1* @ *compE2 e2*)
(*stack-xlift* (*length STK*) (*compxE2 e1 0 0*) @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length STK*) (*compxE2 e2 0* (*Suc 0*)))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *length* (*compE2 e1*) + *pc*, ⌊*a*⌋) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*rule exec-meth-take*)(*simp add*: *pc*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*stack-xlift* (*length STK*) (*compxE2 e2 0* (*Suc 0*))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *pc*, ⌊*a*⌋) *ta h′* (*stk′*, *loc′*, *pc′* − *length* (*compE2 e1*), *xcp′*)
by(*rule exec-meth-drop-xt*)(*auto simp add*: *stack-xlift-compxE2*)

hence *?exec e2 stk (v1 # STK) loc pc ⌊a⌋ stk' loc' (pc' − length (compE2 e1)) xcp'*
by(*simp add*: *compxE2-stack-xlift-convs*)
from *IH2*[*OF this*] **obtain** *stk''* **where** *stk'*: *stk' = stk'' @ v1 # STK* **and**
exec'': *exec-meth-d (compP2 P) (compE2 e2) (compxE2 e2 0 0) t h (stk, loc, pc, ⌊a⌋) ta h' (stk'', loc', pc' − length (compE2 e1), xcp')* **by** *blast*
from *exec''* **have** *exec-meth-d (compP2 P) (compE2 e2) (stack-xlift (length [v1]) (compxE2 e2 0 0)) t h (stk @ [v1], loc, pc, ⌊a⌋)*
ta h' (stk'' @ [v1], loc', pc' − length (compE2 e1), xcp')
by(*rule exec-meth-stk-offer*)
hence *exec-meth-d (compP2 P) (compE2 e1 @ compE2 e2) (compxE2 e1 0 0 @ shift (length (compE2 e1)) (stack-xlift (length [v1]) (compxE2 e2 0 0))) t h (stk @ [v1], loc, length (compE2 e1) + pc, ⌊a⌋)*
ta h' (stk'' @ [v1], loc', length (compE2 e1) + (pc' − length (compE2 e1)), xcp')
by(*rule append-exec-meth-xt*)(*auto*)
hence *exec-meth-d (compP2 P) ((compE2 e1 @ compE2 e2) @ [BinOpInstr bop]) (compxE2 e1 0 0 @ shift (length (compE2 e1)) (stack-xlift (length [v1]) (compxE2 e2 0 0))) t h (stk @ [v1], loc, length (compE2 e1) + pc, ⌊a⌋)*
ta h' (stk'' @ [v1], loc', length (compE2 e1) + (pc' − length (compE2 e1)), xcp')
by(*rule exec-meth-append*)
moreover from *exec'* **have** *pc'*: *pc' ≥ length (compE2 e1)*
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
ultimately show *?case* **using** *stk'* **by**(*auto simp add*: *stack-xlift-compxE2 shift-compxE2*)
next
case *bisim1BinOpThrow* **thus** *?case*
by(*auto elim*!: *exec-meth.cases dest*: *match-ex-table-pcsD simp add*: *stack-xlift-compxEs2 stack-xlift-compxE2*)
next
case (*bisim1LAss1 e n e' xs stk loc pc xcp V*)
note *bisim* = ‹*P*,*e*,*h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *IH* = ‹⋀*stk' loc' pc' xcp' STK*. *?exec e stk STK loc pc xcp stk' loc' pc' xcp'*
⟹ *?concl e stk STK loc pc xcp stk' loc' pc' xcp'*›
note *exec* = ‹*?exec* (*V* := *e*) *stk STK loc pc xcp stk' loc' pc' xcp'*›
from *bisim* **have** *pc*: *pc* ≤ *length* (*compE2 e*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc < length* (*compE2 e*))
case *True*
with *exec* **have** *?exec e stk STK loc pc xcp stk' loc' pc' xcp'*
by(*simp add*: *compxE2-size-convs*)(*erule exec-meth-take*)
from *IH*[*OF this*] **show** *?thesis* **by** *auto*
next
case *False*
with *pc* **have** [*simp*]: *pc = length* (*compE2 e*) **by** *simp*
with *bisim* **obtain** *v* **where** [*simp*]: *stk* = [*v*] *xcp* = *None*
by(*auto dest*: *dest*: *bisim1-pc-length-compE2D*)
with *exec* **show** *?thesis* **apply**(*simp*)
by(*erule exec-meth.cases*)(*auto intro*!: *exec-meth.intros*)
qed
next
case (*bisim1LAss2 e n xs V*)
thus *?case* **by**(*fastforce elim*: *exec-meth.cases intro*: *exec-meth.intros*)
next
case (*bisim1LAssThrow e n a xs stk loc pc V*)
note *bisim* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
note *IH* = ‹⋀*stk' loc' pc' xcp' STK*. *?exec e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'*
⟹ *?concl e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'*›
note *exec* = ‹*?exec* (*V* := *e*) *stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'*›

```
  from bisim have pc: pc < length (compE2 e) and [simp]: xs = loc
    by(auto dest: bisim1-ThrowD)
  from exec have ?exec e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'
    by(simp)(erule exec-meth-take[OF - pc])
  from IH[OF this] show ?case by(auto)
next
  case (bisim1AAcc1 a n a' xs stk loc pc xcp i)
  note IH1 = ‹⋀stk' loc' pc' xcp' STK. ?exec a stk STK loc pc xcp stk' loc' pc' xcp'
          ⟹ ?concl a stk STK loc pc xcp stk' loc' pc' xcp'›
  note IH2 = ‹⋀xs stk' loc' pc' xcp' STK. ?exec i [] STK xs 0 None stk' loc' pc' xcp'
          ⟹ ?concl i [] STK xs 0 None stk' loc' pc' xcp'›
  note bisim1 = ‹P,a,h ⊢ (a', xs) ↔ (stk, loc, pc, xcp)›
  note bisim2 = ‹P,i,h ⊢ (i, loc) ↔ ([], loc, 0, None)›
  note exec = ‹?exec (a⌊i⌋) stk STK loc pc xcp stk' loc' pc' xcp'›
  from bisim1 have pc: pc ≤ length (compE2 a) by(rule bisim1-pc-length-compE2)
  show ?case
  proof(cases pc < length (compE2 a))
    case True
    with exec have ?exec a stk STK loc pc xcp stk' loc' pc' xcp'
      by(simp add: compxE2-size-convs)(erule exec-meth-take-xt)
    from IH1[OF this] show ?thesis by auto
  next
    case False
    with pc have pc: pc = length (compE2 a) by simp
    with exec have pc' ≥ length (compE2 a)
      by(simp add: compxE2-size-convs stack-xlift-compxE2)(auto elim!: exec-meth-drop-xt-pc)
    then obtain PC where PC: pc' = PC + length (compE2 a)
      by -(rule-tac PC34=pc' − length (compE2 a) in that, simp)
    from pc bisim1 obtain v where stk = [v] xcp = None by(auto dest: bisim1-pc-length-compE2D)
    with exec pc have exec-meth-d (compP2 P) (compE2 a @ compE2 i)
   (stack-xlift (length STK) (compxE2 a 0 0 @ compxE2 i (length (compE2 a)) (Suc 0))) t h (stk @
STK, loc, length (compE2 a) + 0, xcp) ta h' (stk', loc', pc', xcp')
      by−(rule exec-meth-take, auto)
    hence ?exec i [] (v # STK) loc 0 None stk' loc' (pc' − length (compE2 a)) xcp'
      using ‹stk = [v]› ‹xcp = None›
      by −(rule exec-meth-drop-xt, auto simp add: stack-xlift-compxE2 shift-compxE2)
    from IH2[OF this] PC obtain stk'' where stk': stk' = stk'' @ v # STK
      and exec-meth-d (compP2 P) (compE2 i) (compxE2 i 0 0) t h ([], loc, 0, None) ta h' (stk'', loc',
PC, xcp') by auto
    hence exec-meth-d (compP2 P) ((compE2 a @ compE2 i) @ [ALoad])
       (compxE2 a 0 0 @ shift (length (compE2 a)) (stack-xlift (length [v]) (compxE2 i 0 0))) t h
       ([] @ [v], loc, length (compE2 a) + 0, None) ta h' (stk'' @ [v], loc', length (compE2 a) + PC,
xcp')
      apply −
      apply(rule exec-meth-append)
      apply(rule append-exec-meth-xt)
      apply(erule exec-meth-stk-offer)
      by(auto)
    thus ?thesis using ‹stk = [v]› ‹xcp = None› stk' pc PC
      by(clarsimp simp add: shift-compxE2 stack-xlift-compxE2 ac-simps)
  qed
next
  case (bisim1AAcc2 i n i' xs stk loc pc xcp a v1)
  note IH2 = ‹⋀stk' loc' pc' xcp' STK. ?exec i stk STK loc pc xcp stk' loc' pc' xcp'
```

$\Longrightarrow$ *?concl i stk STK loc pc xcp stk′ loc′ pc′ xcp′*›
note *bisim2* = ‹*P*,*i*,*h* ⊢ (*i′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *exec* = ‹*?exec* (*a*⌊*i*⌋) (*stk* @ [*v1*]) *STK loc* (*length* (*compE2 a*) + *pc*) *xcp stk′ loc′ pc′ xcp′*›
from *bisim2* **have** *pc*: *pc* ≤ *length* (*compE2 i*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 i*))
case *True*
from *exec* **have** *exec-meth-d* (*compP2 P*) ((*compE2 a* @ *compE2 i*) @ [*ALoad*])
(*stack-xlift* (*length STK*) (*compxE2 a 0 0*) @ *shift* (*length* (*compE2 a*)) (*stack-xlift* (*length STK*) (*compxE2 i 0* (*Suc 0*)))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *length* (*compE2 a*) + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*) **by**(*simp add*: *compxE2-size-convs*)
hence *exec′*: *exec-meth-d* (*compP2 P*) (*compE2 a* @ *compE2 i*) (*stack-xlift* (*length STK*) (*compxE2 a 0 0*) @
shift (*length* (*compE2 a*)) (*stack-xlift* (*length STK*) (*compxE2 i 0* (*Suc 0*)))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *length* (*compE2 a*) + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*rule exec-meth-take*)(*simp add*: *True*)
hence *exec-meth-d* (*compP2 P*) (*compE2 i*) (*stack-xlift* (*length STK*) (*compxE2 i 0* (*Suc 0*))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′* − *length* (*compE2 a*), *xcp′*)
by(*rule exec-meth-drop-xt*)(*auto simp add*: *stack-xlift-compxE2*)
hence *?exec i stk* (*v1* # *STK*) *loc pc xcp stk′ loc′* (*pc′* − *length* (*compE2 a*)) *xcp′*
by(*simp add*: *compxE2-stack-xlift-convs*)
from *IH2*[*OF this*] **obtain** *stk″* **where** *stk′*: *stk′* = *stk″* @ *v1* # *STK*
and *exec″*: *exec-meth-d* (*compP2 P*) (*compE2 i*) (*compxE2 i 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk″*, *loc′*, *pc′* − *length* (*compE2 a*), *xcp′*) **by** *blast*
from *exec″* **have** *exec-meth-d* (*compP2 P*) (*compE2 i*) (*stack-xlift* (*length* [*v1*]) (*compxE2 i 0 0*)) *t h* (*stk* @ [*v1*], *loc*, *pc*, *xcp*)
ta h′ (*stk″* @ [*v1*], *loc′*, *pc′* − *length* (*compE2 a*), *xcp′*)
by(*rule exec-meth-stk-offer*)
hence *exec-meth-d* (*compP2 P*) (*compE2 a* @ *compE2 i*) (*compxE2 a 0 0* @ *shift* (*length* (*compE2 a*)) (*stack-xlift* (*length* [*v1*]) (*compxE2 i 0 0*))) *t h* (*stk* @ [*v1*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*)
ta h′ (*stk″* @ [*v1*], *loc′*, *length* (*compE2 a*) + (*pc′* − *length* (*compE2 a*)), *xcp′*)
by(*rule append-exec-meth-xt*) *auto*
hence *exec-meth-d* (*compP2 P*) ((*compE2 a* @ *compE2 i*) @ [*ALoad*]) (*compxE2 a 0 0* @ *shift* (*length* (*compE2 a*)) (*stack-xlift* (*length* [*v1*]) (*compxE2 i 0 0*))) *t h* (*stk* @ [*v1*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*)
ta h′ (*stk″* @ [*v1*], *loc′*, *length* (*compE2 a*) + (*pc′* − *length* (*compE2 a*)), *xcp′*)
by(*rule exec-meth-append*)
moreover from *exec′* **have** *pc′* ≥ *length* (*compE2 a*)
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
ultimately show *?thesis* **using** *stk′* **by**(*simp add*: *stack-xlift-compxE2 shift-compxE2*)
next
case *False*
with *pc* **have** *pc*: *pc* = *length* (*compE2 i*) **by** *simp*
with *bisim2* **obtain** *v2* **where** [*simp*]: *stk* = [*v2*] *xcp* = *None*
by(*auto dest*: *dest*: *bisim1-pc-length-compE2D*)
with *exec pc* **show** *?thesis*
by(*clarsimp*)(*erule exec-meth.cases*, *auto intro*!: *exec-meth.intros split*: *if-split-asm*)
qed
next
case (*bisim1AAccThrow1 A n a xs stk loc pc i*)
note *bisim1* = ‹*P*,*A*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
note *IH1* = ‹⋀*stk′ loc′ pc′ xcp′ STK*. *?exec A stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*
$\Longrightarrow$ *?concl A stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›

note *exec* = ‹*?exec* (*A*⌊*i*⌉) *stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
from *bisim1* **have** *pc*: *pc* < *length* (*compE2 A*) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-ThrowD*)
from *exec* **have** *exec-meth-d* (*compP2 P*) (*compE2 A* @ (*compE2 i* @ [*ALoad*]))
(*stack-xlift* (*length STK*) (*compxE2 A 0 0*) @ *shift* (*length* (*compE2 A*)) (*stack-xlift* (*length STK*) (*compxE2 i 0* (*Suc 0*)))) *t*
h (*stk* @ *STK*, *loc*, *pc*, ⌊*a*⌋) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*) **by**(*simp add*: *compxE2-size-convs*)
hence *?exec A stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′* **by**(*rule exec-meth-take-xt*)(*rule pc*)
from *IH1*[*OF this*] **show** *?case* **by**(*auto*)
next
case (*bisim1AAccThrow2 i n a xs stk loc pc A v1*)
note *bisim2* = ‹*P*,*i*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
note *IH2* = ‹⋀*stk′ loc′ pc′ xcp′ STK*. *?exec i stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*
⟹ *?concl i stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
note *exec* = ‹*?exec* (*A*⌊*i*⌉) (*stk* @ [*v1*]) *STK loc* (*length* (*compE2 A*) + *pc*) ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
from *bisim2* **have** *pc*: *pc* < *length* (*compE2 i*) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-ThrowD*)
from *exec* **have** *exec-meth-d* (*compP2 P*) ((*compE2 A* @ *compE2 i*) @ [*ALoad*])
(*stack-xlift* (*length STK*) (*compxE2 A 0 0*) @ *shift* (*length* (*compE2 A*)) (*stack-xlift* (*length STK*) (*compxE2 i 0* (*Suc 0*)))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *length* (*compE2 A*) + *pc*, ⌊*a*⌋) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *compxE2-size-convs*)
hence *exec′*: *exec-meth-d* (*compP2 P*) (*compE2 A* @ *compE2 i*)
(*stack-xlift* (*length STK*) (*compxE2 A 0 0*) @ *shift* (*length* (*compE2 A*)) (*stack-xlift* (*length STK*) (*compxE2 i 0* (*Suc 0*)))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *length* (*compE2 A*) + *pc*, ⌊*a*⌋) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*rule exec-meth-take*)(*simp add*: *pc*)
hence *exec-meth-d* (*compP2 P*) (*compE2 i*) (*stack-xlift* (*length STK*) (*compxE2 i 0* (*Suc 0*))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *pc*, ⌊*a*⌋) *ta h′* (*stk′*, *loc′*, *pc′* − *length* (*compE2 A*), *xcp′*)
by(*rule exec-meth-drop-xt*)(*auto simp add*: *stack-xlift-compxE2*)
hence *?exec i stk* (*v1* # *STK*) *loc pc* ⌊*a*⌋ *stk′ loc′* (*pc′* − *length* (*compE2 A*)) *xcp′*
by(*simp add*: *compxE2-stack-xlift-convs*)
from *IH2*[*OF this*] **obtain** *stk″* **where** *stk′*: *stk′* = *stk″* @ *v1* # *STK* **and**
exec″: *exec-meth-d* (*compP2 P*) (*compE2 i*) (*compxE2 i 0 0*) *t h* (*stk*, *loc*, *pc*, ⌊*a*⌋) *ta h′* (*stk″*, *loc′*, *pc′* − *length* (*compE2 A*), *xcp′*) **by** *blast*
from *exec″* **have** *exec-meth-d* (*compP2 P*) (*compE2 i*) (*stack-xlift* (*length* [*v1*]) (*compxE2 i 0 0*)) *t*
h (*stk* @ [*v1*], *loc*, *pc*, ⌊*a*⌋)
ta h′ (*stk″* @ [*v1*], *loc′*, *pc′* − *length* (*compE2 A*), *xcp′*)
by(*rule exec-meth-stk-offer*)
hence *exec-meth-d* (*compP2 P*) (*compE2 A* @ *compE2 i*) (*compxE2 A 0 0* @ *shift* (*length* (*compE2 A*)) (*stack-xlift* (*length* [*v1*]) (*compxE2 i 0 0*))) *t h* (*stk* @ [*v1*], *loc*, *length* (*compE2 A*) + *pc*, ⌊*a*⌋)
ta h′ (*stk″* @ [*v1*], *loc′*, *length* (*compE2 A*) + (*pc′* − *length* (*compE2 A*)), *xcp′*)
by(*rule append-exec-meth-xt*)(*auto*)
hence *exec-meth-d* (*compP2 P*) ((*compE2 A* @ *compE2 i*) @ [*ALoad*]) (*compxE2 A 0 0* @ *shift* (*length* (*compE2 A*)) (*stack-xlift* (*length* [*v1*]) (*compxE2 i 0 0*))) *t h* (*stk* @ [*v1*], *loc*, *length* (*compE2 A*) + *pc*, ⌊*a*⌋)
ta h′ (*stk″* @ [*v1*], *loc′*, *length* (*compE2 A*) + (*pc′* − *length* (*compE2 A*)), *xcp′*)
by(*rule exec-meth-append*)
moreover from *exec′* **have** *pc′*: *pc′* ≥ *length* (*compE2 A*)
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
ultimately show *?case* **using** *stk′* **by**(*auto simp add*: *stack-xlift-compxE2 shift-compxE2*)
next
case *bisim1AAccFail* **thus** *?case*
by(*auto elim*!: *exec-meth.cases dest*: *match-ex-table-pcsD simp add*: *stack-xlift-compxEs2 stack-xlift-compxE2*)

next
case (*bisim1AAss1 a n a′ xs stk loc pc xcp i e*)
note *IH1* = ‹⋀*stk′ loc′ pc′ xcp′ STK*. *?exec a stk STK loc pc xcp stk′ loc′ pc′ xcp′* ⟹ *?concl a stk STK loc pc xcp stk′ loc′ pc′ xcp′*›
note *IH2* = ‹⋀*xs stk′ loc′ pc′ xcp′ STK*. *?exec i* [] *STK xs 0 None stk′ loc′ pc′ xcp′* ⟹ *?concl i* [] *STK xs 0 None stk′ loc′ pc′ xcp′*›
note *bisim1* = ‹*P*,*a*,*h* ⊢ (*a′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim2* = ‹*P*,*i*,*h* ⊢ (*i*, *loc*) ↔ ([], *loc*, *0*, *None*)›
note *exec* = ‹*?exec* (*a*⌊*i*⌉ := *e*) *stk STK loc pc xcp stk′ loc′ pc′ xcp′*›
from *bisim1* **have** *pc*: *pc* ≤ *length* (*compE2 a*) **by**(*rule bisim1-pc-length-compE2*)
from *exec* **have** *exec′*: *exec-meth-d* (*compP2 P*) (*compE2 a* @ *compE2 i* @ *compE2 e* @ [*AStore*, *Push Unit*]) (*stack-xlift* (*length STK*) (*compxE2 a 0 0*) @ *shift* (*length* (*compE2 a*)) (*stack-xlift* (*length STK*) (*compxE2 i 0* (*Suc 0*) @ *compxE2 e* (*length* (*compE2 i*)) (*Suc* (*Suc 0*))))) *t*
h (*stk* @ *STK*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *compxE2-size-convs*)
show *?case*
proof(*cases pc* < *length* (*compE2 a*))
case *True*
with *exec′* **have** *?exec a stk STK loc pc xcp stk′ loc′ pc′ xcp′* **by**(*rule exec-meth-take-xt*)
from *IH1*[*OF this*] **show** *?thesis* **by** *auto*
next
case *False*
with *pc* **have** *pc*: *pc* = *length* (*compE2 a*) **by** *simp*
with *exec′* **have** *pc′* ≥ *length* (*compE2 a*) **by** −(*erule exec-meth-drop-xt-pc*, *auto*)
then obtain *PC* **where** *PC*: *pc′* = *PC* + *length* (*compE2 a*)
by −(*rule-tac PC34=pc′ − length* (*compE2 a*) **in** *that*, *simp*)
from *pc bisim1* **obtain** *v* **where** *stk* = [*v*] *xcp* = *None* **by**(*auto dest*: *bisim1-pc-length-compE2D*)
with *exec PC pc*
have *exec-meth-d* (*compP2 P*) ((*compE2 a* @ *compE2 i*) @ *compE2 e* @ [*AStore*, *Push Unit*]) (*stack-xlift* (*length STK*) (*compxE2 a 0 0* @ *shift* (*length* (*compE2 a*)) (*compxE2 i 0* (*Suc 0*))) @ *shift* (*length* (*compE2 a* @ *compE2 i*)) (*compxE2 e 0* (*length STK* + *Suc* (*Suc 0*)))) *t*
h (*v* # *STK*, *loc*, *length* (*compE2 a*) + *0*, *None*) *ta h′* (*stk′*, *loc′*, *length* (*compE2 a*) + *PC*, *xcp′*)
by(*simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps*)
hence *exec-meth-d* (*compP2 P*) (*compE2 a* @ *compE2 i*)
(*stack-xlift* (*length STK*) (*compxE2 a 0 0* @ *shift* (*length* (*compE2 a*)) (*compxE2 i 0* (*Suc 0*)))) *t h* (*v* # *STK*, *loc*, *length* (*compE2 a*) + *0*, *None*) *ta h′* (*stk′*, *loc′*, *length* (*compE2 a*) + *PC*, *xcp′*)
by(*rule exec-meth-take-xt*) *simp*
hence *?exec i* [] (*v* # *STK*) *loc 0 None stk′ loc′* ((*length* (*compE2 a*) + *PC*) − *length* (*compE2 a*)) *xcp′*
by −(*rule exec-meth-drop-xt*, *auto simp add*: *stack-xlift-compxE2 shift-compxE2*)
from *IH2*[*OF this*] *PC* **obtain** *stk″* **where** *stk′*: *stk′* = *stk″* @ *v* # *STK*
and *exec-meth-d* (*compP2 P*) (*compE2 i*) (*compxE2 i 0 0*) *t h* ([], *loc*, *0*, *None*) *ta h′* (*stk″*, *loc′*, *PC*, *xcp′*) **by** *auto*
hence *exec-meth-d* (*compP2 P*) ((*compE2 a* @ *compE2 i*) @ (*compE2 e* @ [*AStore*, *Push Unit*])) ((*compxE2 a 0 0* @ *shift* (*length* (*compE2 a*)) (*stack-xlift* (*length* [*v*]) (*compxE2 i 0 0*))) @ *shift* (*length* (*compE2 a* @ *compE2 i*)) (*compxE2 e 0* (*Suc* (*Suc 0*)))) *t h* ([] @ [*v*], *loc*, *length* (*compE2 a*) + *0*, *None*) *ta h′* (*stk″* @ [*v*], *loc′*, *length* (*compE2 a*) + *PC*, *xcp′*)
apply −
apply(*rule exec-meth-append-xt*)
apply(*rule append-exec-meth-xt*)
apply(*erule exec-meth-stk-offer*)
by(*auto*)
thus *?thesis* **using** ‹*stk* = [*v*]› ‹*xcp* = *None*› *stk′ pc PC*

by(*clarsimp simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps*)
qed
next
case (*bisim1AAss2 i n i' xs stk loc pc xcp a e v*)
note *IH2* = ‹⋀*stk' loc' pc' xcp' STK*. *?exec i stk STK loc pc xcp stk' loc' pc' xcp'* ⟹ *?concl i stk STK loc pc xcp stk' loc' pc' xcp'›
note *IH3* = ‹⋀*xs stk' loc' pc' xcp' STK*. *?exec e [] STK xs 0 None stk' loc' pc' xcp'* ⟹ *?concl e [] STK xs 0 None stk' loc' pc' xcp'›
note *bisim2* = ‹*P*,*i*,*h* ⊢ (*i'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim3* = ‹*P*,*e*,*h* ⊢ (*e*, *loc*) ↔ ([], *loc*, *0*, *None*)›
note *exec* = ‹*?exec* (*a*⌊*i*⌋ := *e*) (*stk* @ [*v*]) *STK loc* (*length* (*compE2 a*) + *pc*) *xcp stk' loc' pc' xcp'›
from *bisim2* **have** *pc*: *pc* ≤ *length* (*compE2 i*) **by**(*rule bisim1-pc-length-compE2*)
from *exec* **have** *exec'*: ⋀*v'*. *exec-meth-d* (*compP2 P*) ((*compE2 a* @ *compE2 i*) @ *compE2 e* @ [*AStore*, *Push Unit*]) ((*compxE2 a 0* (*length STK*) @ *shift* (*length* (*compE2 a*)) (*stack-xlift* (*length* (*v* # *STK*)) (*compxE2 i 0 0*))) @ *shift* (*length* (*compE2 a* @ *compE2 i*)) (*stack-xlift* (*length* (*v'* # *v* # *STK*)) (*compxE2 e 0 0*))) *t*
h (*stk* @ *v* # *STK*, *loc*, *length* (*compE2 a*) + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *shift-compxE2 stack-xlift-compxE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 i*))
case *True* **with** *exec'*[*of arbitrary*]
have *exec''*: *exec-meth-d* (*compP2 P*) (*compE2 a* @ *compE2 i*) (*compxE2 a 0* (*length STK*) @ *shift* (*length* (*compE2 a*)) (*stack-xlift* (*length* (*v* # *STK*)) (*compxE2 i 0 0*))) *t h* (*stk* @ *v* # *STK*, *loc*, *length* (*compE2 a*) + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by−(*erule exec-meth-take-xt*, *simp*)
hence *?exec i stk* (*v* # *STK*) *loc pc xcp stk' loc'* (*pc'* − *length* (*compE2 a*)) *xcp'*
by(*rule exec-meth-drop-xt*) *auto*
from *IH2*[*OF this*] **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ *v* # *STK*
and *exec'''*: *exec-meth-d* (*compP2 P*) (*compE2 i*) (*compxE2 i 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 a*), *xcp'*) **by** *blast*
from *exec'''* **have** *exec-meth-d* (*compP2 P*) ((*compE2 a* @ *compE2 i*) @ *compE2 e* @ [*AStore*, *Push Unit*]) ((*compxE2 a 0 0* @ *shift* (*length* (*compE2 a*)) (*stack-xlift* (*length* [*v*]) (*compxE2 i 0 0*))) @ *shift* (*length* (*compE2 a* @ *compE2 i*)) (*compxE2 e 0* (*Suc* (*Suc 0*)))) *t h* (*stk* @ [*v*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) *ta h'* (*stk''* @ [*v*], *loc'*, *length* (*compE2 a*) + (*pc'* − *length* (*compE2 a*)), *xcp'*)
apply −
apply(*rule exec-meth-append-xt*)
apply(*rule append-exec-meth-xt*)
apply(*erule exec-meth-stk-offer*)
by *auto*
moreover from *exec''* **have** *pc'* ≥ *length* (*compE2 a*)
by(*rule exec-meth-drop-xt-pc*) *auto*
ultimately show *?thesis* **using** *stk'* **by**(*auto simp add*: *shift-compxE2 stack-xlift-compxE2*)
next
case *False*
with *pc* **have** *pc*: *pc* = *length* (*compE2 i*) **by** *simp*
with *exec'*[*of arbitrary*] **have** *pc'* ≥ *length* (*compE2 a* @ *compE2 i*)
by−(*erule exec-meth-drop-xt-pc*, *auto simp add*: *shift-compxE2 stack-xlift-compxE2*)
then obtain *PC* **where** *PC*: *pc'* = *PC* + *length* (*compE2 a*) + *length* (*compE2 i*)
by −(*rule-tac PC34*=*pc'* − *length* (*compE2 a* @ *compE2 i*) **in** *that*, *simp*)
from *pc bisim2* **obtain** *v'* **where** *stk*: *stk* = [*v'*] **and** *xcp*: *xcp* = *None* **by**(*auto dest*: *bisim1-pc-length-compE2D*)
from *exec'*[*of v'*]
have *exec-meth-d* (*compP2 P*) (*compE2 e* @ [*AStore*, *Push Unit*]) (*stack-xlift* (*length* (*v'* # *v* # *STK*)) (*compxE2 e 0 0*)) *t*
h (*v'* # *v* # *STK*, *loc*, *0*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 a* @ *compE2 i*),

xcp′)
unfolding *stk pc append-Cons append-Nil*
by −(*rule exec-meth-drop-xt*, *simp only*: *add-0-right length-append*, *auto simp add*: *shift-compxE2 stack-xlift-compxE2*)
with *PC xcp* **have** *?exec e* [] (*v′* # *v* # *STK*) *loc 0 None stk′ loc′ PC xcp′*
by −(*rule exec-meth-take*,*auto*)
from *IH3*[*OF this*] **obtain** *stk″* **where** *stk′*: *stk′* = *stk″* @ *v′* # *v* # *STK*
and *exec-meth-d* (*compP2 P*) (*compE2 e*) (*compxE2 e 0 0*) *t h* ([], *loc*, *0*, *None*) *ta h′* (*stk″*, *loc′*, *PC*, *xcp′*) **by** *auto*
hence *exec-meth-d* (*compP2 P*) (((*compE2 a* @ *compE2 i*) @ *compE2 e*) @ [*AStore*, *Push Unit*]) ((*compxE2 a 0 0* @ *compxE2 i* (*length* (*compE2 a*)) (*Suc 0*)) @ *shift* (*length* (*compE2 a* @ *compE2 i*)) (*stack-xlift* (*length* [*v′*, *v*]) (*compxE2 e 0 0*))) *t h* ([] @ [*v′*, *v*], *loc*, *length* (*compE2 a* @ *compE2 i*) + *0*, *None*) *ta h′* (*stk″* @ [*v′*, *v*], *loc′*, *length* (*compE2 a* @ *compE2 i*) + *PC*, *xcp′*)
apply −
apply(*rule exec-meth-append*)
apply(*rule append-exec-meth-xt*)
apply(*erule exec-meth-stk-offer*)
by *auto*
thus *?thesis* **using** *stk xcp stk′ pc PC*
by(*clarsimp simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps*)
qed
next
case (*bisim1AAss3 e n e′ xs stk loc pc xcp a i v1 v2*)
note *IH3* = ‹⋀*stk′ loc′ pc′ xcp′ STK*. *?exec e stk STK loc pc xcp stk′ loc′ pc′ xcp′* ⟹ *?concl e stk STK loc pc xcp stk′ loc′ pc′ xcp′*›
note *bisim3* = ‹*P*,*e*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *exec* = ‹*?exec* (*a*⌊*i*⌉ := *e*) (*stk* @ [*v2*, *v1*]) *STK loc* (*length* (*compE2 a*) + *length* (*compE2 i*) + *pc*) *xcp stk′ loc′ pc′ xcp′*›
from *bisim3* **have** *pc*: *pc* ≤ *length* (*compE2 e*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e*))
case *True*
from *exec* **have** *exec-meth-d* (*compP2 P*) (((*compE2 a* @ *compE2 i*) @ *compE2 e*) @ [*AStore*, *Push Unit*]) ((*compxE2 a 0* (*length STK*) @ *compxE2 i* (*length* (*compE2 a*)) (*Suc* (*length STK*))) @ *shift* (*length* (*compE2 a* @ *compE2 i*)) (*stack-xlift* (*length* (*v2* # *v1* # *STK*)) (*compxE2 e 0 0*))) *t h* (*stk* @ *v2* # *v1* # *STK*, *loc*, *length* (*compE2 a* @ *compE2 i*) + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *shift-compxE2 stack-xlift-compxE2*)
hence *exec′*: *exec-meth-d* (*compP2 P*) ((*compE2 a* @ *compE2 i*) @ *compE2 e*) ((*compxE2 a 0* (*length STK*) @ *compxE2 i* (*length* (*compE2 a*)) (*Suc* (*length STK*))) @ *shift* (*length* (*compE2 a* @ *compE2 i*)) (*stack-xlift* (*length* (*v2* # *v1* # *STK*)) (*compxE2 e 0 0*))) *t h* (*stk* @ *v2* # *v1* # *STK*, *loc*, *length* (*compE2 a* @ *compE2 i*) + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*rule exec-meth-take*)(*simp add*: *True*)
hence *?exec e stk* (*v2* # *v1* # *STK*) *loc pc xcp stk′ loc′* (*pc′* − *length* (*compE2 a* @ *compE2 i*)) *xcp′*
by(*rule exec-meth-drop-xt*) *auto*
from *IH3*[*OF this*] **obtain** *stk″* **where** *stk′*: *stk′* = *stk″* @ *v2* # *v1* # *STK*
and *exec″*: *exec-meth-d* (*compP2 P*) (*compE2 e*) (*compxE2 e 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk″*, *loc′*, *pc′* − *length* (*compE2 a* @ *compE2 i*), *xcp′*) **by** *blast*
from *exec″* **have** *exec-meth-d* (*compP2 P*) (*compE2 e*) (*stack-xlift* (*length* [*v2*, *v1*]) (*compxE2 e 0 0*)) *t h* (*stk* @ [*v2*, *v1*], *loc*, *pc*, *xcp*) *ta h′* (*stk″* @ [*v2*, *v1*], *loc′*, *pc′* − *length* (*compE2 a* @ *compE2 i*), *xcp′*)

by(*rule exec-meth-stk-offer*)
hence *exec-meth-d* (*compP2 P*) ((*compE2 a* @ *compE2 i*) @ *compE2 e*) ((*compxE2 a 0 0* @ *compxE2 i* (*length* (*compE2 a*)) (*Suc 0*)) @ *shift* (*length* (*compE2 a* @ *compE2 i*)) (*stack-xlift* (*length* [*v2*, *v1*]) (*compxE2 e 0 0*))) *t*
h (*stk* @ [*v2*, *v1*], *loc*, *length* (*compE2 a* @ *compE2 i*) + *pc*, *xcp*)
ta h′ (*stk″* @ [*v2*, *v1*], *loc′*, *length* (*compE2 a* @ *compE2 i*) + (*pc′* − *length* (*compE2 a* @ *compE2 i*)), *xcp′*)
by(*rule append-exec-meth-xt*) *auto*
hence *exec-meth-d* (*compP2 P*) (((*compE2 a* @ *compE2 i*) @ *compE2 e*) @ [*AStore*, *Push Unit*]) ((*compxE2 a 0 0* @ *compxE2 i* (*length* (*compE2 a*)) (*Suc 0*)) @ *shift* (*length* (*compE2 a* @ *compE2 i*)) (*stack-xlift* (*length* [*v2*, *v1*]) (*compxE2 e 0 0*))) *t*
h (*stk* @ [*v2*, *v1*], *loc*, *length* (*compE2 a* @ *compE2 i*) + *pc*, *xcp*)
ta h′ (*stk″* @ [*v2*, *v1*], *loc′*, *length* (*compE2 a* @ *compE2 i*) + (*pc′* − *length* (*compE2 a* @ *compE2 i*)), *xcp′*)
by(*rule exec-meth-append*)
moreover from *exec′* **have** *pc′* ≥ *length* (*compE2 a* @ *compE2 i*)
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
ultimately show *?thesis* **using** *stk′* **by**(*simp add*: *stack-xlift-compxE2 shift-compxE2*)
next
case *False*
with *pc* **have** *pc*: *pc* = *length* (*compE2 e*) **by** *simp*
with *bisim3* **obtain** *v3* **where** [*simp*]: *stk* = [*v3*] *xcp* = *None*
by(*auto dest*: *dest*: *bisim1-pc-length-compE2D*)
with *exec pc* **show** *?thesis* **apply**(*simp*)
by(*erule exec-meth.cases*)(*auto intro*!: *exec-meth.intros split*: *if-split-asm*)
qed
next
case (*bisim1AAssThrow1 A n a xs stk loc pc i e*)
note *bisim1* = ‹*P*,*A*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
note *IH1* = ‹⋀*stk′ loc′ pc′ xcp′ STK*. *?exec A stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′* ⟹ *?concl A stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
note *exec* = ‹*?exec* (*A*⌊*i*⌉ := *e*) *stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
from *bisim1* **have** *pc*: *pc* < *length* (*compE2 A*) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-ThrowD*)
from *exec* **have** *exec-meth-d* (*compP2 P*) (*compE2 A* @ (*compE2 i* @ *compE2 e* @ [*AStore*, *Push Unit*]))
(*stack-xlift* (*length STK*) (*compxE2 A 0 0*) @ *shift* (*length* (*compE2 A*)) (*stack-xlift* (*length STK*) (*compxE2 i 0* (*Suc 0*) @ *compxE2 e* (*length* (*compE2 i*)) (*Suc* (*Suc 0*))))) *t*
h (*stk* @ *STK*, *loc*, *pc*, ⌊*a*⌋) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*) **by**(*simp add*: *compxE2-size-convs*)
hence *?exec A stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′* **by**(*rule exec-meth-take-xt*)(*rule pc*)
from *IH1*[*OF this*] **show** *?case* **by**(*auto*)
next
case (*bisim1AAssThrow2 i n a xs stk loc pc A e v1*)
note *bisim2* = ‹*P*,*i*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
note *IH2* = ‹⋀*stk′ loc′ pc′ xcp′ STK*. *?exec i stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′* ⟹ *?concl i stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
note *exec* = ‹*?exec* (*A*⌊*i*⌉ := *e*) (*stk* @ [*v1*]) *STK loc* (*length* (*compE2 A*) + *pc*) ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
from *bisim2* **have** *pc*: *pc* < *length* (*compE2 i*) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-ThrowD*)
from *exec* **have** *exec-meth-d* (*compP2 P*) ((*compE2 A* @ *compE2 i*) @ *compE2 e* @ [*AStore*, *Push Unit*])
((*stack-xlift* (*length STK*) (*compxE2 A 0 0*) @ *shift* (*length* (*compE2 A*)) (*stack-xlift* (*length STK*) (*compxE2 i 0* (*Suc 0*)))) @ (*shift* (*length* (*compE2 A* @ *compE2 i*)) (*compxE2 e 0* (*Suc* (*Suc* (*length*

STK)))))) *t*
    h (*stk* @ *v1* # *STK*, *loc*, *length* (*compE2 A*) + *pc*, ⌊*a*⌋) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
   by(*simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps*)
 hence *exec'*: *exec-meth-d* (*compP2 P*) (*compE2 A* @ *compE2 i*)
    (*stack-xlift* (*length STK*) (*compxE2 A 0 0*) @ *shift* (*length* (*compE2 A*)) (*stack-xlift* (*length STK*) (*compxE2 i 0* (*Suc 0*)))) *t*
    h (*stk* @ *v1* # *STK*, *loc*, *length* (*compE2 A*) + *pc*, ⌊*a*⌋) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
   by(*rule exec-meth-take-xt*)(*simp add*: *pc*)
 hence *exec-meth-d* (*compP2 P*) (*compE2 i*) (*stack-xlift* (*length STK*) (*compxE2 i 0* (*Suc 0*))) *t*
    h (*stk* @ *v1* # *STK*, *loc*, *pc*, ⌊*a*⌋) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 A*), *xcp'*)
   by(*rule exec-meth-drop-xt*)(*auto simp add*: *stack-xlift-compxE2*)
 hence *?exec i stk* (*v1* # *STK*) *loc pc* ⌊*a*⌋ *stk' loc'* (*pc'* − *length* (*compE2 A*)) *xcp'*
   by(*simp add*: *compxE2-stack-xlift-convs*)
 from *IH2*[*OF this*] **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ *v1* # *STK* **and**
   exec'': *exec-meth-d* (*compP2 P*) (*compE2 i*) (*compxE2 i 0 0*) *t h* (*stk*, *loc*, *pc*, ⌊*a*⌋) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 A*), *xcp'*) **by** *blast*
 from *exec''* **have** *exec-meth-d* (*compP2 P*) (*compE2 i*) (*stack-xlift* (*length* [*v1*]) (*compxE2 i 0 0*)) *t*
    h (*stk* @ [*v1*], *loc*, *pc*, ⌊*a*⌋)
    ta h' (*stk''* @ [*v1*], *loc'*, *pc'* − *length* (*compE2 A*), *xcp'*)
   by(*rule exec-meth-stk-offer*)
 hence *exec-meth-d* (*compP2 P*) (*compE2 A* @ *compE2 i*) (*compxE2 A 0 0* @ *shift* (*length* (*compE2 A*)) (*stack-xlift* (*length* [*v1*]) (*compxE2 i 0 0*))) *t*
    h (*stk* @ [*v1*], *loc*, *length* (*compE2 A*) + *pc*, ⌊*a*⌋)
    ta h' (*stk''* @ [*v1*], *loc'*, *length* (*compE2 A*) + (*pc'* − *length* (*compE2 A*)), *xcp'*)
   by(*rule append-exec-meth-xt*)(*auto*)
 hence *exec-meth-d* (*compP2 P*) ((*compE2 A* @ *compE2 i*) @ *compE2 e* @ [*AStore*, *Push Unit*]) ((*compxE2 A 0 0* @ *shift* (*length* (*compE2 A*)) (*stack-xlift* (*length* [*v1*]) (*compxE2 i 0 0*))) @ (*shift* (*length* (*compE2 A* @ *compE2 i*)) (*compxE2 e 0* (*Suc* (*Suc 0*))))) *t*
    h (*stk* @ [*v1*], *loc*, *length* (*compE2 A*) + *pc*, ⌊*a*⌋)
    ta h' (*stk''* @ [*v1*], *loc'*, *length* (*compE2 A*) + (*pc'* − *length* (*compE2 A*)), *xcp'*)
   by(*rule exec-meth-append-xt*)
 moreover from *exec'* **have** *pc'*: *pc'* ≥ *length* (*compE2 A*)
   by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
 ultimately show *?case* **using** *stk'* **by**(*auto simp add*: *stack-xlift-compxE2 shift-compxE2*)
next
 case (*bisim1AAssThrow3 e n a xs stk loc pc A i v2 v1*)
 note *bisim3* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
 note *IH3* = ‹⋀*stk' loc' pc' xcp' STK*. *?exec e stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*
        ⟹ *?concl e stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'›*
 note *exec* = ‹*?exec* (*A*⌊*i*⌉ := *e*) (*stk* @ [*v2*, *v1*]) *STK loc* (*length* (*compE2 A*) + *length* (*compE2 i*) + *pc*) ⌊*a*⌋ *stk' loc' pc' xcp'*›
 from *bisim3* **have** *pc*: *pc* < *length* (*compE2 e*) **and** [*simp*]: *xs* = *loc*
   by(*auto dest*: *bisim1-ThrowD*)
 from *exec* **have** *exec-meth-d* (*compP2 P*) (((*compE2 A* @ *compE2 i*) @ *compE2 e*) @ [*AStore*, *Push Unit*])
    ((*stack-xlift* (*length STK*) (*compxE2 A 0 0* @ *compxE2 i* (*length* (*compE2 A*)) (*Suc 0*))) @ *shift* (*length* (*compE2 A* @ *compE2 i*)) (*stack-xlift* (*length* (*v2* # *v1* # *STK*)) (*compxE2 e 0 0*))) *t*
    h (*stk* @ *v2* # *v1* # *STK*, *loc*, *length* (*compE2 A* @ *compE2 i*) + *pc*, ⌊*a*⌋) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
   by(*simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps*)
 hence *exec'*: *exec-meth-d* (*compP2 P*) ((*compE2 A* @ *compE2 i*) @ *compE2 e*)
    ((*stack-xlift* (*length STK*) (*compxE2 A 0 0* @ *compxE2 i* (*length* (*compE2 A*)) (*Suc 0*))) @ *shift* (*length* (*compE2 A* @ *compE2 i*)) (*stack-xlift* (*length* (*v2* # *v1* # *STK*)) (*compxE2 e 0 0*))) *t*
    h (*stk* @ *v2* # *v1* # *STK*, *loc*, *length* (*compE2 A* @ *compE2 i*) + *pc*, ⌊*a*⌋) *ta h'* (*stk'*, *loc'*, *pc'*,

xcp′)
by(*rule exec-meth-take*)(*simp add*: *pc*)
hence *?exec e stk* (*v2* # *v1* # *STK*) *loc pc* ⌊*a*⌋ *stk′ loc′* (*pc′* − *length* (*compE2 A* @ *compE2 i*)) *xcp′*
by(*rule exec-meth-drop-xt*) *auto*
from *IH3*[*OF this*] **obtain** *stk″* **where** *stk′*: *stk′* = *stk″* @ *v2* # *v1* # *STK* **and**
exec″: *exec-meth-d* (*compP2 P*) (*compE2 e*) (*compxE2 e 0 0*) *t h* (*stk*, *loc*, *pc*, ⌊*a*⌋) *ta h′* (*stk″*, *loc′*, *pc′* − *length* (*compE2 A* @ *compE2 i*), *xcp′*) **by** *blast*
from *exec″* **have** *exec-meth-d* (*compP2 P*) (*compE2 e*) (*stack-xlift* (*length* [*v2*, *v1*]) (*compxE2 e 0 0*)) *t h* (*stk* @ [*v2*, *v1*], *loc*, *pc*, ⌊*a*⌋)
ta h′ (*stk″* @ [*v2*, *v1*], *loc′*, *pc′* − *length* (*compE2 A* @ *compE2 i*), *xcp′*)
by(*rule exec-meth-stk-offer*)
hence *exec-meth-d* (*compP2 P*) ((*compE2 A* @ *compE2 i*) @ *compE2 e*) ((*compxE2 A 0 0* @ *compxE2 i* (*length* (*compE2 A*)) (*Suc 0*)) @ *shift* (*length* (*compE2 A* @ *compE2 i*)) (*stack-xlift* (*length* [*v2*, *v1*]) (*compxE2 e 0 0*))) *t h* (*stk* @ [*v2*, *v1*], *loc*, *length* (*compE2 A* @ *compE2 i*) + *pc*, ⌊*a*⌋)
ta h′ (*stk″* @ [*v2*, *v1*], *loc′*, *length* (*compE2 A* @ *compE2 i*) + (*pc′* − *length* (*compE2 A* @ *compE2 i*)), *xcp′*)
by(*rule append-exec-meth-xt*)(*auto*)
hence *exec-meth-d* (*compP2 P*) (((*compE2 A* @ *compE2 i*) @ *compE2 e*) @ [*AStore*, *Push Unit*]) ((*compxE2 A 0 0* @ *compxE2 i* (*length* (*compE2 A*)) (*Suc 0*)) @ *shift* (*length* (*compE2 A* @ *compE2 i*)) (*stack-xlift* (*length* [*v2*, *v1*]) (*compxE2 e 0 0*))) *t h* (*stk* @ [*v2*, *v1*], *loc*, *length* (*compE2 A* @ *compE2 i*) + *pc*, ⌊*a*⌋)
ta h′ (*stk″* @ [*v2*, *v1*], *loc′*, *length* (*compE2 A* @ *compE2 i*) + (*pc′* − *length* (*compE2 A* @ *compE2 i*)), *xcp′*)
by(*rule exec-meth-append*)
moreover from *exec′* **have** *pc′*: *pc′* ≥ *length* (*compE2 A* @ *compE2 i*)
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
ultimately show *?case* **using** *stk′* **by**(*auto simp add*: *stack-xlift-compxE2 shift-compxE2*)
next
case *bisim1AAssFail* **thus** *?case*
by(*auto elim*!: *exec-meth.cases dest*: *match-ex-table-pcsD simp add*: *stack-xlift-compxEs2 stack-xlift-compxE2*)
next
case *bisim1AAss4* **thus** *?case*
by −(*erule exec-meth.cases*, *auto intro*!: *exec-meth.exec-instr*)
next
case (*bisim1ALength a n a′ xs stk loc pc xcp*)
note *bisim* = ‹*P*,*a*,*h* ⊢ (*a′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *IH* = ‹⋀*stk′ loc′ pc′ xcp′ STK*. *?exec a stk STK loc pc xcp stk′ loc′ pc′ xcp′*
⟹ *?concl a stk STK loc pc xcp stk′ loc′ pc′ xcp′*›
note *exec* = ‹*?exec* (*a*·*length*) *stk STK loc pc xcp stk′ loc′ pc′ xcp′*›
from *bisim* **have** *pc*: *pc* ≤ *length* (*compE2 a*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 a*))
case *True*
with *exec* **have** *?exec a stk STK loc pc xcp stk′ loc′ pc′ xcp′*
by(*simp add*: *compxE2-size-convs*)(*erule exec-meth-take*)
from *IH*[*OF this*] **show** *?thesis* **by** *auto*
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compE2 a*) **by** *simp*
with *bisim* **obtain** *v* **where** [*simp*]: *stk* = [*v*] *xcp* = *None*
by(*auto dest*: *dest*: *bisim1-pc-length-compE2D*)
with *exec* **show** *?thesis* **apply**(*simp*)
by(*erule exec-meth.cases*)(*auto intro*!: *exec-meth.intros split*: *if-split-asm*)

```
  qed
next
  case (bisim1ALengthThrow e n a xs stk loc pc)
  note bisim = ‹P,e,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
  note IH = ‹⋀stk' loc' pc' xcp' STK. ?exec e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'
          ⟹ ?concl e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'›
  note exec = ‹?exec (e•length) stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'›
  from bisim have pc: pc < length (compE2 e) and [simp]: xs = loc
    by(auto dest: bisim1-ThrowD)
  from exec have ?exec e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'
    by(simp)(erule exec-meth-take[OF - pc])
  from IH[OF this] show ?case by(auto)
next
  case bisim1ALengthNull thus ?case
   by(auto elim!: exec-meth.cases dest: match-ex-table-pcsD simp add: stack-xlift-compxEs2 stack-xlift-compxE2)
next
  case (bisim1FAcc e n e' xs stk loc pc xcp F D)
  note bisim = ‹P,e,h ⊢ (e', xs) ↔ (stk, loc, pc, xcp)›
  note IH = ‹⋀stk' loc' pc' xcp' STK. ?exec e stk STK loc pc xcp stk' loc' pc' xcp'
          ⟹ ?concl e stk STK loc pc xcp stk' loc' pc' xcp'›
  note exec = ‹?exec (e•F{D}) stk STK loc pc xcp stk' loc' pc' xcp'›
  from bisim have pc: pc ≤ length (compE2 e) by(rule bisim1-pc-length-compE2)
  show ?case
  proof(cases pc < length (compE2 e))
    case True
    with exec have ?exec e stk STK loc pc xcp stk' loc' pc' xcp'
      by(simp add: compxE2-size-convs)(erule exec-meth-take)
    from IH[OF this] show ?thesis by auto
  next
    case False
    with pc have [simp]: pc = length (compE2 e) by simp
    with bisim obtain v where [simp]: stk = [v] xcp = None
      by(auto dest: dest: bisim1-pc-length-compE2D)
    with exec show ?thesis apply(simp)
    by(erule exec-meth.cases)(fastforce intro!: exec-meth.intros simp add: is-Ref-def split: if-split-asm)+
  qed
next
  case (bisim1FAccThrow e n a xs stk loc pc F D)
  note bisim = ‹P,e,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
  note IH = ‹⋀stk' loc' pc' xcp' STK. ?exec e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'
          ⟹ ?concl e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'›
  note exec = ‹?exec (e•F{D}) stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'›
  from bisim have pc: pc < length (compE2 e) and [simp]: xs = loc
    by(auto dest: bisim1-ThrowD)
  from exec have ?exec e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'
    by(simp)(erule exec-meth-take[OF - pc])
  from IH[OF this] show ?case by(auto)
next
  case bisim1FAccNull thus ?case
   by(auto elim!: exec-meth.cases dest: match-ex-table-pcsD simp add: stack-xlift-compxEs2 stack-xlift-compxE2)
next
  case (bisim1FAss1 e n e' xs stk loc pc xcp e2 F D)
  note IH1 = ‹⋀stk' loc' pc' xcp' STK. ?exec e stk STK loc pc xcp stk' loc' pc' xcp'
          ⟹ ?concl e stk STK loc pc xcp stk' loc' pc' xcp'›
```

note *IH2* = ‹⋀*xs stk′ loc′ pc′ xcp′ STK*. *?exec e2* [] *STK xs 0 None stk′ loc′ pc′ xcp′*
⟹ *?concl e2* [] *STK xs 0 None stk′ loc′ pc′ xcp′*›
note *bisim1* = ‹*P*,*e*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e2*, *loc*) ↔ ([], *loc*, *0*, *None*)›
note *exec* = ‹*?exec* (*e*•*F*{*D*} := *e2*) *stk STK loc pc xcp stk′ loc′ pc′ xcp′*›
from *bisim1* **have** *pc*: *pc* ≤ *length* (*compE2 e*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e*))
case *True*
with *exec* **have** *?exec e stk STK loc pc xcp stk′ loc′ pc′ xcp′*
by(*simp add*: *compxE2-size-convs*)(*erule exec-meth-take-xt*)
from *IH1*[*OF this*] **show** *?thesis* **by** *auto*
next
case *False*
with *pc* **have** *pc*: *pc* = *length* (*compE2 e*) **by** *simp*
with *exec* **have** *pc′* ≥ *length* (*compE2 e*)
by(*simp add*: *compxE2-size-convs stack-xlift-compxE2*)(*auto elim*!: *exec-meth-drop-xt-pc*)
then obtain *PC* **where** *PC*: *pc′* = *PC* + *length* (*compE2 e*)
by −(*rule-tac PC34*=*pc′* − *length* (*compE2 e*) **in** *that*, *simp*)
from *pc bisim1* **obtain** *v* **where** *stk* = [*v*] *xcp* = *None* **by**(*auto dest*: *bisim1-pc-length-compE2D*)
with *exec pc* **have** *exec-meth-d* (*compP2 P*) (*compE2 e* @ *compE2 e2*)
(*stack-xlift* (*length STK*) (*compxE2 e 0 0* @ *compxE2 e2* (*length* (*compE2 e*)) (*Suc 0*))) *t*
h (*stk* @ *STK*, *loc*, *length* (*compE2 e*) + *0*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by−(*rule exec-meth-take*, *auto*)
hence *?exec e2* [] (*v* # *STK*) *loc 0 None stk′ loc′* (*pc′* − *length* (*compE2 e*)) *xcp′*
using ‹*stk* = [*v*]› ‹*xcp* = *None*›
by −(*rule exec-meth-drop-xt*, *auto simp add*: *stack-xlift-compxE2 shift-compxE2*)
from *IH2*[*OF this*] *PC* **obtain** *stk″* **where** *stk′*: *stk′* = *stk″* @ *v* # *STK*
and *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t h* ([], *loc*, *0*, *None*) *ta h′* (*stk″*,
loc′, *PC*, *xcp′*) **by** *auto*
hence *exec-meth-d* (*compP2 P*) ((*compE2 e* @ *compE2 e2*) @ [*Putfield F D*, *Push Unit*])
(*compxE2 e 0 0* @ *shift* (*length* (*compE2 e*)) (*stack-xlift* (*length* [*v*]) (*compxE2 e2 0 0*))) *t h*
([] @ [*v*], *loc*, *length* (*compE2 e*) + *0*, *None*) *ta h′* (*stk″* @ [*v*], *loc′*, *length* (*compE2 e*) + *PC*,
xcp′)
apply −
apply(*rule exec-meth-append*)
apply(*rule append-exec-meth-xt*)
apply(*erule exec-meth-stk-offer*)
by(*auto*)
thus *?thesis* **using** ‹*stk* = [*v*]› ‹*xcp* = *None*› *stk′ pc PC*
by(*clarsimp simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps*)
qed
next
case (*bisim1FAss2 e2 n e′ xs stk loc pc xcp e F D v1*)
note *IH2* = ‹⋀*stk′ loc′ pc′ xcp′ STK*. *?exec e2 stk STK loc pc xcp stk′ loc′ pc′ xcp′*
⟹ *?concl e2 stk STK loc pc xcp stk′ loc′ pc′ xcp′*›
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *exec* = ‹*?exec* (*e*•*F*{*D*} := *e2*) (*stk* @ [*v1*]) *STK loc* (*length* (*compE2 e*) + *pc*) *xcp stk′ loc′ pc′*
xcp′›
from *bisim2* **have** *pc*: *pc* ≤ *length* (*compE2 e2*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e2*))
case *True*
from *exec* **have** *exec-meth-d* (*compP2 P*) ((*compE2 e* @ *compE2 e2*) @ [*Putfield F D*, *Push Unit*])

(*stack-xlift* (*length STK*) (*compxE2 e 0 0*) @ *shift* (*length* (*compE2 e*)) (*stack-xlift* (*length STK*) (*compxE2 e2 0* (*Suc 0*)))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *length* (*compE2 e*) + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*) **by**(*simp add*: *compxE2-size-convs*)
hence *exec'*: *exec-meth-d* (*compP2 P*) (*compE2 e* @ *compE2 e2*) (*stack-xlift* (*length STK*) (*compxE2 e 0 0*) @
shift (*length* (*compE2 e*)) (*stack-xlift* (*length STK*) (*compxE2 e2 0* (*Suc 0*)))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *length* (*compE2 e*) + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*rule exec-meth-take*)(*simp add*: *True*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*stack-xlift* (*length STK*) (*compxE2 e2 0* (*Suc 0*))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 e*), *xcp'*)
by(*rule exec-meth-drop-xt*)(*auto simp add*: *stack-xlift-compxE2*)
hence *?exec e2 stk* (*v1* # *STK*) *loc pc xcp stk' loc'* (*pc'* − *length* (*compE2 e*)) *xcp'*
by(*simp add*: *compxE2-stack-xlift-convs*)
from *IH2*[*OF this*] **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ *v1* # *STK*
and *exec''*: *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 e*), *xcp'*) **by** *blast*
from *exec''* **have** *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*stack-xlift* (*length* [*v1*]) (*compxE2 e2 0 0*)) *t h* (*stk* @ [*v1*], *loc*, *pc*, *xcp*)
ta h' (*stk''* @ [*v1*], *loc'*, *pc'* − *length* (*compE2 e*), *xcp'*)
by(*rule exec-meth-stk-offer*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e* @ *compE2 e2*) (*compxE2 e 0 0* @ *shift* (*length* (*compE2 e*)) (*stack-xlift* (*length* [*v1*]) (*compxE2 e2 0 0*))) *t h* (*stk* @ [*v1*], *loc*, *length* (*compE2 e*) + *pc*, *xcp*)
ta h' (*stk''* @ [*v1*], *loc'*, *length* (*compE2 e*) + (*pc'* − *length* (*compE2 e*)), *xcp'*)
by(*rule append-exec-meth-xt*) *auto*
hence *exec-meth-d* (*compP2 P*) ((*compE2 e* @ *compE2 e2*) @ [*Putfield F D*, *Push Unit*]) (*compxE2 e 0 0* @ *shift* (*length* (*compE2 e*)) (*stack-xlift* (*length* [*v1*]) (*compxE2 e2 0 0*))) *t h* (*stk* @ [*v1*], *loc*, *length* (*compE2 e*) + *pc*, *xcp*)
ta h' (*stk''* @ [*v1*], *loc'*, *length* (*compE2 e*) + (*pc'* − *length* (*compE2 e*)), *xcp'*)
by(*rule exec-meth-append*)
moreover from *exec'* **have** *pc'* ≥ *length* (*compE2 e*)
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
ultimately show *?thesis* **using** *stk'* **by**(*simp add*: *stack-xlift-compxE2 shift-compxE2*)
next
case *False*
with *pc* **have** *pc*: *pc* = *length* (*compE2 e2*) **by** *simp*
with *bisim2* **obtain** *v2* **where** [*simp*]: *stk* = [*v2*] *xcp* = *None*
by(*auto dest*: *dest*: *bisim1-pc-length-compE2D*)
with *exec pc* **show** *?thesis* **apply**(*simp*)
by(*erule exec-meth.cases*)(*fastforce intro*!: *exec-meth.intros split*: *if-split-asm*)+
qed
next
case (*bisim1FAssThrow1 e n a xs stk loc pc e2 F D*)
note *bisim1* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
note *IH1* = ‹⋀*stk' loc' pc' xcp' STK*. *?exec e stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*
⟹ *?concl e stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*›
note *exec* = ‹*?exec* (*e*•*F*{*D*} := *e2*) *stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*›
from *bisim1* **have** *pc*: *pc* < *length* (*compE2 e*) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-ThrowD*)
from *exec* **have** *exec-meth-d* (*compP2 P*) (*compE2 e* @ (*compE2 e2* @ [*Putfield F D*, *Push Unit*]))
(*stack-xlift* (*length STK*) (*compxE2 e 0 0*) @ *shift* (*length* (*compE2 e*)) (*stack-xlift* (*length STK*) (*compxE2 e2 0* (*Suc 0*)))) *t*
h (*stk* @ *STK*, *loc*, *pc*, ⌊*a*⌋) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*) **by**(*simp add*: *compxE2-size-convs*)
hence *?exec e stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'* **by**(*rule exec-meth-take-xt*)(*rule pc*)

from *IH1*[*OF this*] **show** *?case* **by**(*auto*)
next
case (*bisim1FAssThrow2 e2 n a xs stk loc pc e F D v1*)
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
note *IH2* = ‹⋀*stk' loc' pc' xcp' STK*. *?exec e2 stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*
⟹ *?concl e2 stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'›
note *exec* = ‹*?exec* (*e*·*F*{*D*} := *e2*) (*stk* @ [*v1*]) *STK loc* (*length* (*compE2 e*) + *pc*) ⌊*a*⌋ *stk' loc' pc' xcp'›
from *bisim2* **have** *pc*: *pc* < *length* (*compE2 e2*) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-ThrowD*)
from *exec* **have** *exec-meth-d* (*compP2 P*) ((*compE2 e* @ *compE2 e2*) @ [*Putfield F D*, *Push Unit*])
(*stack-xlift* (*length STK*) (*compxE2 e 0 0*) @ *shift* (*length* (*compE2 e*)) (*stack-xlift* (*length STK*) (*compxE2 e2 0* (*Suc 0*)))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *length* (*compE2 e*) + *pc*, ⌊*a*⌋) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *compxE2-size-convs*)
hence *exec'*: *exec-meth-d* (*compP2 P*) (*compE2 e* @ *compE2 e2*)
(*stack-xlift* (*length STK*) (*compxE2 e 0 0*) @ *shift* (*length* (*compE2 e*)) (*stack-xlift* (*length STK*) (*compxE2 e2 0* (*Suc 0*)))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *length* (*compE2 e*) + *pc*, ⌊*a*⌋) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*rule exec-meth-take*)(*simp add*: *pc*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*stack-xlift* (*length STK*) (*compxE2 e2 0* (*Suc 0*))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *pc*, ⌊*a*⌋) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 e*), *xcp'*)
by(*rule exec-meth-drop-xt*)(*auto simp add*: *stack-xlift-compxE2*)
hence *?exec e2 stk* (*v1* # *STK*) *loc pc* ⌊*a*⌋ *stk' loc'* (*pc'* − *length* (*compE2 e*)) *xcp'*
by(*simp add*: *compxE2-stack-xlift-convs*)
from *IH2*[*OF this*] **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ *v1* # *STK* **and**
exec'': *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t h* (*stk*, *loc*, *pc*, ⌊*a*⌋) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 e*), *xcp'*) **by** *blast*
from *exec''* **have** *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*stack-xlift* (*length* [*v1*]) (*compxE2 e2 0 0*))
t h (*stk* @ [*v1*], *loc*, *pc*, ⌊*a*⌋)
ta h' (*stk''* @ [*v1*], *loc'*, *pc'* − *length* (*compE2 e*), *xcp'*)
by(*rule exec-meth-stk-offer*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e* @ *compE2 e2*) (*compxE2 e 0 0* @ *shift* (*length* (*compE2 e*)) (*stack-xlift* (*length* [*v1*]) (*compxE2 e2 0 0*))) *t h* (*stk* @ [*v1*], *loc*, *length* (*compE2 e*) + *pc*, ⌊*a*⌋)
ta h' (*stk''* @ [*v1*], *loc'*, *length* (*compE2 e*) + (*pc'* − *length* (*compE2 e*)), *xcp'*)
by(*rule append-exec-meth-xt*)(*auto*)
hence *exec-meth-d* (*compP2 P*) ((*compE2 e* @ *compE2 e2*) @ [*Putfield F D*, *Push Unit*]) (*compxE2 e 0 0* @ *shift* (*length* (*compE2 e*)) (*stack-xlift* (*length* [*v1*]) (*compxE2 e2 0 0*))) *t h* (*stk* @ [*v1*], *loc*, *length* (*compE2 e*) + *pc*, ⌊*a*⌋)
ta h' (*stk''* @ [*v1*], *loc'*, *length* (*compE2 e*) + (*pc'* − *length* (*compE2 e*)), *xcp'*)
by(*rule exec-meth-append*)
moreover from *exec'* **have** *pc'*: *pc'* ≥ *length* (*compE2 e*)
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
ultimately show *?case* **using** *stk'* **by**(*auto simp add*: *stack-xlift-compxE2 shift-compxE2*)
next
case *bisim1FAssNull* **thus** *?case*
by(*auto elim*!: *exec-meth.cases dest*: *match-ex-table-pcsD simp add*: *stack-xlift-compxEs2 stack-xlift-compxE2*)
next
case *bisim1FAss3* **thus** *?case*
by −(*erule exec-meth.cases*, *auto intro*!: *exec-meth.exec-instr*)
next
case (*bisim1CAS1 e1 n e1' xs stk loc pc xcp e2 e3 D F*)
note *IH1* = ‹⋀*stk' loc' pc' xcp' STK*. *?exec e1 stk STK loc pc xcp stk' loc' pc' xcp'*
⟹ *?concl e1 stk STK loc pc xcp stk' loc' pc' xcp'›

note *IH2* = ‹$\bigwedge$*xs stk′ loc′ pc′ xcp′ STK*. *?exec e2* [] *STK xs 0 None stk′ loc′ pc′ xcp′*
⟹ *?concl e2* [] *STK xs 0 None stk′ loc′ pc′ xcp′*›
note *bisim1* = ‹*P*,*e1*,*h* ⊢ (*e1′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e2*, *loc*) ↔ ([], *loc*, *0*, *None*)›
note *exec* = ‹*?exec - stk STK loc pc xcp stk′ loc′ pc′ xcp′*›
from *bisim1* **have** *pc*: *pc* ≤ *length* (*compE2 e1*) **by**(*rule bisim1-pc-length-compE2*)
from *exec* **have** *exec′*: *exec-meth-d* (*compP2 P*) (*compE2 e1* @ *compE2 e2* @ *compE2 e3* @ [*CAS F D*]) (*stack-xlift* (*length STK*) (*compxE2 e1 0 0*) @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length STK*) (*compxE2 e2 0* (*Suc 0*) @ *compxE2 e3* (*length* (*compE2 e2*)) (*Suc* (*Suc 0*))))) *t*
h (*stk* @ *STK*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *compxE2-size-convs*)
show *?case*
proof(*cases pc* < *length* (*compE2 e1*))
case *True*
with *exec′* **have** *?exec e1 stk STK loc pc xcp stk′ loc′ pc′ xcp′* **by**(*rule exec-meth-take-xt*)
from *IH1*[*OF this*] **show** *?thesis* **by** *auto*
next
case *False*
with *pc* **have** *pc*: *pc* = *length* (*compE2 e1*) **by** *simp*
with *exec′* **have** *pc′* ≥ *length* (*compE2 e1*) **by** −(*erule exec-meth-drop-xt-pc*, *auto*)
then obtain *PC* **where** *PC*: *pc′* = *PC* + *length* (*compE2 e1*)
by −(*rule-tac PC34*=*pc′* − *length* (*compE2 e1*) **in** *that*, *simp*)
from *pc bisim1* **obtain** *v* **where** *stk* = [*v*] *xcp* = *None* **by**(*auto dest*: *bisim1-pc-length-compE2D*)
with *exec PC pc*
have *exec-meth-d* (*compP2 P*) ((*compE2 e1* @ *compE2 e2*) @ *compE2 e3* @ [*CAS F D*]) (*stack-xlift* (*length STK*) (*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*compxE2 e2 0* (*Suc 0*))) @ *shift* (*length* (*compE2 e1* @ *compE2 e2*)) (*compxE2 e3 0* (*length STK* + *Suc* (*Suc 0*)))) *t*
h (*v* # *STK*, *loc*, *length* (*compE2 e1*) + *0*, *None*) *ta h′* (*stk′*, *loc′*, *length* (*compE2 e1*) + *PC*, *xcp′*)

by(*simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e1* @ *compE2 e2*)
(*stack-xlift* (*length STK*) (*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*compxE2 e2 0* (*Suc 0*)))) *t h* (*v* # *STK*, *loc*, *length* (*compE2 e1*) + *0*, *None*) *ta h′* (*stk′*, *loc′*, *length* (*compE2 e1*) + *PC*, *xcp′*)
by(*rule exec-meth-take-xt*) *simp*
hence *?exec e2* [] (*v* # *STK*) *loc 0 None stk′ loc′* ((*length* (*compE2 e1*) + *PC*) − *length* (*compE2 e1*)) *xcp′*
by −(*rule exec-meth-drop-xt*, *auto simp add*: *stack-xlift-compxE2 shift-compxE2*)
from *IH2*[*OF this*] *PC* **obtain** *stk″* **where** *stk′*: *stk′* = *stk″* @ *v* # *STK*
and *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t h* ([], *loc*, *0*, *None*) *ta h′* (*stk″*, *loc′*, *PC*, *xcp′*) **by** *auto*
hence *exec-meth-d* (*compP2 P*) ((*compE2 e1* @ *compE2 e2*) @ (*compE2 e3* @ [*CAS F D*]))
((*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length* [*v*]) (*compxE2 e2 0 0*))) @
shift (*length* (*compE2 e1* @ *compE2 e2*)) (*compxE2 e3 0* (*Suc* (*Suc 0*)))) *t h*
([] @ [*v*], *loc*, *length* (*compE2 e1*) + *0*, *None*) *ta h′* (*stk″* @ [*v*], *loc′*, *length* (*compE2 e1*) + *PC*, *xcp′*)
apply −
apply(*rule exec-meth-append-xt*)
apply(*rule append-exec-meth-xt*)
apply(*erule exec-meth-stk-offer*)
by(*auto*)
thus *?thesis* **using** ‹*stk* = [*v*]› ‹*xcp* = *None*› *stk′ pc PC*
by(*clarsimp simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps*)
qed
next

case (*bisim1CAS2 e2 n e2′ xs stk loc pc xcp e1 e3 D F v*)
note *IH2* = ‹⋀*stk′ loc′ pc′ xcp′ STK*. *?exec e2 stk STK loc pc xcp stk′ loc′ pc′ xcp′*
⟹ *?concl e2 stk STK loc pc xcp stk′ loc′ pc′ xcp′*›
note *IH3* = ‹⋀*xs stk′ loc′ pc′ xcp′ STK*. *?exec e3* [] *STK xs 0 None stk′ loc′ pc′ xcp′*
⟹ *?concl e3* [] *STK xs 0 None stk′ loc′ pc′ xcp′*›
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e2′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim3* = ‹*P*,*e3*,*h* ⊢ (*e3*, *loc*) ↔ ([], *loc*, *0*, *None*)›
note *exec* = ‹*?exec* - (*stk* @ [*v*]) *STK loc* (*length* (*compE2 e1*) + *pc*) *xcp stk′ loc′ pc′ xcp′*›
from *bisim2* **have** *pc*: *pc* ≤ *length* (*compE2 e2*) **by**(*rule bisim1-pc-length-compE2*)
from *exec* **have** *exec′*: ⋀*v′*. *exec-meth-d* (*compP2 P*) ((*compE2 e1* @ *compE2 e2*) @ *compE2 e3* @ [*CAS F D*]) ((*compxE2 e1 0* (*length STK*) @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length* (*v* # *STK*)) (*compxE2 e2 0 0*))) @ *shift* (*length* (*compE2 e1* @ *compE2 e2*)) (*stack-xlift* (*length* (*v′* # *v* # *STK*)) (*compxE2 e3 0 0*))) *t*
h (*stk* @ *v* # *STK*, *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *shift-compxE2 stack-xlift-compxE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e2*))
case *True* **with** *exec′*[*of undefined*]
have *exec″*: *exec-meth-d* (*compP2 P*) (*compE2 e1* @ *compE2 e2*) (*compxE2 e1 0* (*length STK*) @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length* (*v* # *STK*)) (*compxE2 e2 0 0*))) *t h* (*stk* @ *v* # *STK*, *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by−(*erule exec-meth-take-xt*, *simp*)
hence *?exec e2 stk* (*v* # *STK*) *loc pc xcp stk′ loc′* (*pc′* − *length* (*compE2 e1*)) *xcp′*
by(*rule exec-meth-drop-xt*) *auto*
from *IH2*[*OF this*] **obtain** *stk″* **where** *stk′*: *stk′* = *stk″* @ *v* # *STK*
and *exec‴*: *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk″*, *loc′*, *pc′* − *length* (*compE2 e1*), *xcp′*) **by** *blast*
from *exec‴* **have** *exec-meth-d* (*compP2 P*) ((*compE2 e1* @ *compE2 e2*) @ *compE2 e3* @ [*CAS F D*]) ((*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length* [*v*]) (*compxE2 e2 0 0*))) @ *shift* (*length* (*compE2 e1* @ *compE2 e2*)) (*compxE2 e3 0* (*Suc* (*Suc 0*)))) *t h* (*stk* @ [*v*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) *ta h′* (*stk″* @ [*v*], *loc′*, *length* (*compE2 e1*) + (*pc′* − *length* (*compE2 e1*)), *xcp′*)
apply −
apply(*rule exec-meth-append-xt*)
apply(*rule append-exec-meth-xt*)
apply(*erule exec-meth-stk-offer*)
by *auto*
moreover from *exec″* **have** *pc′* ≥ *length* (*compE2 e1*)
by(*rule exec-meth-drop-xt-pc*) *auto*
ultimately show *?thesis* **using** *stk′* **by**(*auto simp add*: *shift-compxE2 stack-xlift-compxE2*)
next
case *False*
with *pc* **have** *pc*: *pc* = *length* (*compE2 e2*) **by** *simp*
with *exec′*[*of undefined*] **have** *pc′* ≥ *length* (*compE2 e1* @ *compE2 e2*)
by−(*erule exec-meth-drop-xt-pc*, *auto simp add*: *shift-compxE2 stack-xlift-compxE2*)
then obtain *PC* **where** *PC*: *pc′* = *PC* + *length* (*compE2 e1*) + *length* (*compE2 e2*)
by −(*rule-tac PC34*=*pc′* − *length* (*compE2 e1* @ *compE2 e2*) **in** *that*, *simp*)
from *pc bisim2* **obtain** *v′* **where** *stk*: *stk* = [*v′*] **and** *xcp*: *xcp* = *None* **by**(*auto dest*: *bisim1-pc-length-compE2D*)
from *exec′*[*of v′*]
have *exec-meth-d* (*compP2 P*) (*compE2 e3* @ [*CAS F D*]) (*stack-xlift* (*length* (*v′* # *v* # *STK*)) (*compxE2 e3 0 0*)) *t*
h (*v′* # *v* # *STK*, *loc*, *0*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′* − *length* (*compE2 e1* @ *compE2 e2*), *xcp′*)
unfolding *stk pc append-Cons append-Nil*

by −(*rule exec-meth-drop-xt*, *simp only*: *add-0-right length-append*, *auto simp add*: *shift-compxE2 stack-xlift-compxE2*)
with *PC xcp* **have** *?exec e3* [] (*v′ # v # STK*) *loc 0 None stk′ loc′ PC xcp′*
by −(*rule exec-meth-take*,*auto*)
from *IH3*[*OF this*] **obtain** *stk″* **where** *stk′*: *stk′ = stk″ @ v′ # v # STK*
and *exec-meth-d* (*compP2 P*) (*compE2 e3*) (*compxE2 e3 0 0*) *t h* ([], *loc*, *0*, *None*) *ta h′* (*stk″*, *loc′*, *PC*, *xcp′*) **by** *auto*
hence *exec-meth-d* (*compP2 P*) (((*compE2 e1* @ *compE2 e2*) @ *compE2 e3*) @ [*CAS F D*]) ((*compxE2 e1 0 0* @ *compxE2 e2* (*length* (*compE2 e1*)) (*Suc 0*)) @ *shift* (*length* (*compE2 e1* @ *compE2 e2*)) (*stack-xlift* (*length* [*v′*, *v*]) (*compxE2 e3 0 0*))) *t h* ([] @ [*v′*, *v*], *loc*, *length* (*compE2 e1* @ *compE2 e2*) + *0*, *None*) *ta h′* (*stk″* @ [*v′*, *v*], *loc′*, *length* (*compE2 e1* @ *compE2 e2*) + *PC*, *xcp′*)
apply −
apply(*rule exec-meth-append*)
apply(*rule append-exec-meth-xt*)
apply(*erule exec-meth-stk-offer*)
by *auto*
thus *?thesis* **using** *stk xcp stk′ pc PC*
by(*clarsimp simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps*)
qed
next
case (*bisim1CAS3 e3 n e3′ xs stk loc pc xcp e1 e2 D F v1 v2*)
note *IH3* = ‹⋀*stk′ loc′ pc′ xcp′ STK*. *?exec e3 stk STK loc pc xcp stk′ loc′ pc′ xcp′* ⟹ *?concl e3 stk STK loc pc xcp stk′ loc′ pc′ xcp′*›
note *bisim3* = ‹*P*,*e3*,*h* ⊢ (*e3′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *exec* = ‹*?exec* - (*stk* @ [*v2*, *v1*]) *STK loc* (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*) *xcp stk′ loc′ pc′ xcp′*›
from *bisim3* **have** *pc*: *pc* ≤ *length* (*compE2 e3*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e3*))
case *True*
from *exec* **have** *exec-meth-d* (*compP2 P*) (((*compE2 e1* @ *compE2 e2*) @ *compE2 e3*) @ [*CAS F D*])
((*compxE2 e1 0* (*length STK*) @ *compxE2 e2* (*length* (*compE2 e1*)) (*Suc* (*length STK*))) @ *shift* (*length* (*compE2 e1* @ *compE2 e2*)) (*stack-xlift* (*length* (*v2 # v1 # STK*)) (*compxE2 e3 0 0*))) *t*
h (*stk* @ *v2 # v1 # STK*, *loc*, *length* (*compE2 e1* @ *compE2 e2*) + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *shift-compxE2 stack-xlift-compxE2*)
hence *exec′*: *exec-meth-d* (*compP2 P*) ((*compE2 e1* @ *compE2 e2*) @ *compE2 e3*)
((*compxE2 e1 0* (*length STK*) @ *compxE2 e2* (*length* (*compE2 e1*)) (*Suc* (*length STK*))) @ *shift* (*length* (*compE2 e1* @ *compE2 e2*)) (*stack-xlift* (*length* (*v2 # v1 # STK*)) (*compxE2 e3 0 0*))) *t*
h (*stk* @ *v2 # v1 # STK*, *loc*, *length* (*compE2 e1* @ *compE2 e2*) + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*rule exec-meth-take*)(*simp add*: *True*)
hence *?exec e3 stk* (*v2 # v1 # STK*) *loc pc xcp stk′ loc′* (*pc′* − *length* (*compE2 e1* @ *compE2 e2*)) *xcp′*
by(*rule exec-meth-drop-xt*) *auto*
from *IH3*[*OF this*] **obtain** *stk″* **where** *stk′*: *stk′ = stk″ @ v2 # v1 # STK*
and *exec″*: *exec-meth-d* (*compP2 P*) (*compE2 e3*) (*compxE2 e3 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk″*, *loc′*, *pc′* − *length* (*compE2 e1* @ *compE2 e2*), *xcp′*) **by** *blast*
from *exec″* **have** *exec-meth-d* (*compP2 P*) (*compE2 e3*) (*stack-xlift* (*length* [*v2*, *v1*]) (*compxE2 e3 0 0*)) *t h* (*stk* @ [*v2*, *v1*], *loc*, *pc*, *xcp*)
ta h′ (*stk″* @ [*v2*, *v1*], *loc′*, *pc′* − *length* (*compE2 e1* @ *compE2 e2*), *xcp′*)
by(*rule exec-meth-stk-offer*)
hence *exec-meth-d* (*compP2 P*) ((*compE2 e1* @ *compE2 e2*) @ *compE2 e3*) ((*compxE2 e1 0 0* @

compxE2 e2 (length (compE2 e1)) (Suc 0)) @ shift (length (compE2 e1 @ compE2 e2)) (stack-xlift (length [v2, v1]) (compxE2 e3 0 0))) t
h (stk @ [v2, v1], loc, length (compE2 e1 @ compE2 e2) + pc, xcp)
ta h' (stk'' @ [v2, v1], loc', length (compE2 e1 @ compE2 e2) + (pc' − length (compE2 e1 @ compE2 e2)), xcp')
by(*rule append-exec-meth-xt*) *auto*
hence *exec-meth-d (compP2 P) (((compE2 e1 @ compE2 e2) @ compE2 e3) @ [CAS F D]) ((compxE2 e1 0 0 @ compxE2 e2 (length (compE2 e1)) (Suc 0)) @ shift (length (compE2 e1 @ compE2 e2)) (stack-xlift (length [v2, v1]) (compxE2 e3 0 0))) t*
h (stk @ [v2, v1], loc, length (compE2 e1 @ compE2 e2) + pc, xcp)
ta h' (stk'' @ [v2, v1], loc', length (compE2 e1 @ compE2 e2) + (pc' − length (compE2 e1 @ compE2 e2)), xcp')
by(*rule exec-meth-append*)
moreover from *exec'* **have** *pc' ≥ length (compE2 e1 @ compE2 e2)*
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
ultimately show *?thesis* **using** *stk'* **by**(*simp add*: *stack-xlift-compxE2 shift-compxE2*)
next
case *False*
with *pc* **have** *pc*: *pc = length (compE2 e3)* **by** *simp*
with *bisim3* **obtain** *v3* **where** [*simp*]: *stk = [v3] xcp = None*
by(*auto dest*: *dest*: *bisim1-pc-length-compE2D*)
with *exec pc* **show** *?thesis* **apply**(*simp*)
by(*erule exec-meth.cases*)(*fastforce intro*!: *exec-meth.intros split*: *if-split-asm*)+
qed
next
case (*bisim1CASThrow1 e1 n a xs stk loc pc e2 e3 D F*)
note *bisim1 = ‹P,e1,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›*
note *IH1 = ‹⋀stk' loc' pc' xcp' STK. ?exec e1 stk STK loc pc ⌊a⌋ stk' loc' pc' xcp' ⟹ ?concl e1 stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'›*
note *exec = ‹?exec - stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'›*
from *bisim1* **have** *pc*: *pc < length (compE2 e1)* **and** [*simp*]: *xs = loc*
by(*auto dest*: *bisim1-ThrowD*)
from *exec* **have** *exec-meth-d (compP2 P) (compE2 e1 @ (compE2 e2 @ compE2 e3 @ [CAS F D])) (stack-xlift (length STK) (compxE2 e1 0 0) @ shift (length (compE2 e1)) (stack-xlift (length STK) (compxE2 e2 0 (Suc 0)) @ compxE2 e3 (length (compE2 e2)) (Suc (Suc 0))))) t*
h (stk @ STK, loc, pc, ⌊a⌋) ta h' (stk', loc', pc', xcp') **by**(*simp add*: *compxE2-size-convs*)
hence *?exec e1 stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'* **by**(*rule exec-meth-take-xt*)(*rule pc*)
from *IH1*[*OF this*] **show** *?case* **by**(*auto*)
next
case (*bisim1CASThrow2 e2 n a xs stk loc pc e1 e3 D F v1*)
note *bisim2 = ‹P,e2,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›*
note *IH2 = ‹⋀stk' loc' pc' xcp' STK. ?exec e2 stk STK loc pc ⌊a⌋ stk' loc' pc' xcp' ⟹ ?concl e2 stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'›*
note *exec = ‹?exec - (stk @ [v1]) STK loc (length (compE2 e1) + pc) ⌊a⌋ stk' loc' pc' xcp'›*
from *bisim2* **have** *pc*: *pc < length (compE2 e2)* **and** [*simp*]: *xs = loc*
by(*auto dest*: *bisim1-ThrowD*)
from *exec* **have** *exec-meth-d (compP2 P) ((compE2 e1 @ compE2 e2) @ compE2 e3 @ [CAS F D]) ((stack-xlift (length STK) (compxE2 e1 0 0) @ shift (length (compE2 e1)) (stack-xlift (length STK) (compxE2 e2 0 (Suc 0)))) @ (shift (length (compE2 e1 @ compE2 e2)) (compxE2 e3 0 (Suc (Suc (length STK)))))) t*
h (stk @ v1 # STK, loc, length (compE2 e1) + pc, ⌊a⌋) ta h' (stk', loc', pc', xcp')
by(*simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps*)
hence *exec'*: *exec-meth-d (compP2 P) (compE2 e1 @ compE2 e2) (stack-xlift (length STK) (compxE2 e1 0 0) @ shift (length (compE2 e1)) (stack-xlift (length STK)*

(*compxE2 e2 0* (*Suc 0*)))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *length* (*compE2 e1*) + *pc*, ⌊*a*⌋) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*rule exec-meth-take-xt*)(*simp add*: *pc*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*stack-xlift* (*length STK*) (*compxE2 e2 0* (*Suc 0*))) *t*
h (*stk* @ *v1* # *STK*, *loc*, *pc*, ⌊*a*⌋) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 e1*), *xcp'*)
by(*rule exec-meth-drop-xt*)(*auto simp add*: *stack-xlift-compxE2*)
hence *?exec e2 stk* (*v1* # *STK*) *loc pc* ⌊*a*⌋ *stk' loc'* (*pc'* − *length* (*compE2 e1*)) *xcp'*
by(*simp add*: *compxE2-stack-xlift-convs*)
from *IH2*[*OF this*] **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ *v1* # *STK* **and**
exec'': *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t h* (*stk*, *loc*, *pc*, ⌊*a*⌋) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 e1*), *xcp'*) **by** *blast*
from *exec''* **have** *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*stack-xlift* (*length* [*v1*]) (*compxE2 e2 0 0*)) *t*
h (*stk* @ [*v1*], *loc*, *pc*, ⌊*a*⌋)
ta h' (*stk''* @ [*v1*], *loc'*, *pc'* − *length* (*compE2 e1*), *xcp'*)
by(*rule exec-meth-stk-offer*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e1* @ *compE2 e2*) (*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length* [*v1*]) (*compxE2 e2 0 0*))) *t*
h (*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, ⌊*a*⌋)
ta h' (*stk''* @ [*v1*], *loc'*, *length* (*compE2 e1*) + (*pc'* − *length* (*compE2 e1*)), *xcp'*)
by(*rule append-exec-meth-xt*)(*auto*)
hence *exec-meth-d* (*compP2 P*) ((*compE2 e1* @ *compE2 e2*) @ *compE2 e3* @ [*CAS F D*]) ((*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length* [*v1*]) (*compxE2 e2 0 0*))) @ (*shift* (*length* (*compE2 e1* @ *compE2 e2*)) (*compxE2 e3 0* (*Suc* (*Suc 0*))))) *t*
h (*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, ⌊*a*⌋)
ta h' (*stk''* @ [*v1*], *loc'*, *length* (*compE2 e1*) + (*pc'* − *length* (*compE2 e1*)), *xcp'*)
by(*rule exec-meth-append-xt*)
moreover from *exec'* **have** *pc'*: *pc'* ≥ *length* (*compE2 e1*)
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
ultimately show *?case* **using** *stk'* **by**(*auto simp add*: *stack-xlift-compxE2 shift-compxE2*)
next
case (*bisim1CASThrow3 e3 n a xs stk loc pc e1 e2 D F v2 v1*)
note *bisim3* = ‹*P*,*e3*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
note *IH3* = ‹⋀*stk' loc' pc' xcp' STK*. *?exec e3 stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*
⟹ *?concl e3 stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'›*
note *exec* = ‹*?exec* - (*stk* @ [*v2*, *v1*]) *STK loc* (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*) ⌊*a*⌋ *stk' loc' pc' xcp'›*
from *bisim3* **have** *pc*: *pc* < *length* (*compE2 e3*) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-ThrowD*)
from *exec* **have** *exec-meth-d* (*compP2 P*) (((*compE2 e1* @ *compE2 e2*) @ *compE2 e3*) @ [*CAS F D*])
((*stack-xlift* (*length STK*) (*compxE2 e1 0 0* @ *compxE2 e2* (*length* (*compE2 e1*)) (*Suc 0*))) @ *shift* (*length* (*compE2 e1* @ *compE2 e2*)) (*stack-xlift* (*length* (*v2* # *v1* # *STK*)) (*compxE2 e3 0 0*))) *t*
h (*stk* @ *v2* # *v1* # *STK*, *loc*, *length* (*compE2 e1* @ *compE2 e2*) + *pc*, ⌊*a*⌋) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps*)
hence *exec'*: *exec-meth-d* (*compP2 P*) ((*compE2 e1* @ *compE2 e2*) @ *compE2 e3*)
((*stack-xlift* (*length STK*) (*compxE2 e1 0 0* @ *compxE2 e2* (*length* (*compE2 e1*)) (*Suc 0*))) @ *shift* (*length* (*compE2 e1* @ *compE2 e2*)) (*stack-xlift* (*length* (*v2* # *v1* # *STK*)) (*compxE2 e3 0 0*))) *t*
h (*stk* @ *v2* # *v1* # *STK*, *loc*, *length* (*compE2 e1* @ *compE2 e2*) + *pc*, ⌊*a*⌋) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*rule exec-meth-take*)(*simp add*: *pc*)
hence *?exec e3 stk* (*v2* # *v1* # *STK*) *loc pc* ⌊*a*⌋ *stk' loc'* (*pc'* − *length* (*compE2 e1* @ *compE2 e2*)) *xcp'*

by(*rule exec-meth-drop-xt*) *auto*
from *IH3*[*OF this*] **obtain** *stk''* **where** *stk'*: *stk' = stk'' @ v2 # v1 # STK* **and**
exec'': *exec-meth-d* (*compP2 P*) (*compE2 e3*) (*compxE2 e3 0 0*) *t h* (*stk*, *loc*, *pc*, ⌊*a*⌋) *ta h'* (*stk''*, *loc'*, *pc' − length* (*compE2 e1 @ compE2 e2*), *xcp'*) **by** *blast*
from *exec''* **have** *exec-meth-d* (*compP2 P*) (*compE2 e3*) (*stack-xlift* (*length* [*v2*, *v1*]) (*compxE2 e3 0 0*)) *t h* (*stk @* [*v2*, *v1*], *loc*, *pc*, ⌊*a*⌋)
ta h' (*stk'' @* [*v2*, *v1*], *loc'*, *pc' − length* (*compE2 e1 @ compE2 e2*), *xcp'*)
by(*rule exec-meth-stk-offer*)
hence *exec-meth-d* (*compP2 P*) ((*compE2 e1 @ compE2 e2*) *@ compE2 e3*) ((*compxE2 e1 0 0 @ compxE2 e2* (*length* (*compE2 e1*)) (*Suc 0*)) *@ shift* (*length* (*compE2 e1 @ compE2 e2*)) (*stack-xlift* (*length* [*v2*, *v1*]) (*compxE2 e3 0 0*))) *t h* (*stk @* [*v2*, *v1*], *loc*, *length* (*compE2 e1 @ compE2 e2*) *+ pc*, ⌊*a*⌋)
ta h' (*stk'' @* [*v2*, *v1*], *loc'*, *length* (*compE2 e1 @ compE2 e2*) *+* (*pc' − length* (*compE2 e1 @ compE2 e2*)), *xcp'*)
by(*rule append-exec-meth-xt*)(*auto*)
hence *exec-meth-d* (*compP2 P*) (((*compE2 e1 @ compE2 e2*) *@ compE2 e3*) *@* [*CAS F D*]) ((*compxE2 e1 0 0 @ compxE2 e2* (*length* (*compE2 e1*)) (*Suc 0*)) *@ shift* (*length* (*compE2 e1 @ compE2 e2*)) (*stack-xlift* (*length* [*v2*, *v1*]) (*compxE2 e3 0 0*))) *t h* (*stk @* [*v2*, *v1*], *loc*, *length* (*compE2 e1 @ compE2 e2*) *+ pc*, ⌊*a*⌋)
ta h' (*stk'' @* [*v2*, *v1*], *loc'*, *length* (*compE2 e1 @ compE2 e2*) *+* (*pc' − length* (*compE2 e1 @ compE2 e2*)), *xcp'*)
by(*rule exec-meth-append*)
moreover from *exec'* **have** *pc'*: *pc' ≥ length* (*compE2 e1 @ compE2 e2*)
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
ultimately show *?case* **using** *stk'* **by**(*auto simp add*: *stack-xlift-compxE2 shift-compxE2*)
next
case *bisim1CASFail* **thus** *?case*
by(*auto elim*!: *exec-meth.cases dest*: *match-ex-table-pcsD simp add*: *stack-xlift-compxEs2 stack-xlift-compxE2*)
next
case (*bisim1Call1 obj n obj' xs stk loc pc xcp ps M'*)
note *bisimObj* = ‹*P*,*obj*,*h* ⊢ (*obj'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *IHobj* = ‹⋀*stk' loc' pc' xcp' STK*. *?exec obj stk STK loc pc xcp stk' loc' pc' xcp'*
⟹ *?concl obj stk STK loc pc xcp stk' loc' pc' xcp'*›
note *IHparams* = ‹⋀*xs stk' loc' pc' xcp' STK*. *?execs ps* [] *STK xs 0 None stk' loc' pc' xcp'*
⟹ *?concls ps* [] *STK xs 0 None stk' loc' pc' xcp'*›
note *exec* = ‹*?exec* (*obj*·*M'*(*ps*)) *stk STK loc pc xcp stk' loc' pc' xcp'*›
from *bisimObj* **have** *pc*: *pc ≤ length* (*compE2 obj*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc < length* (*compE2 obj*))
case *True*
from *exec* **have** *?exec obj stk STK loc pc xcp stk' loc' pc' xcp'*
by(*simp add*: *compxEs2-size-convs*)(*erule exec-meth-take-xt*[*OF - True*])
from *IHobj*[*OF this*] **show** *?thesis* **by** *auto*
next
case *False*
with *pc* **have** [*simp*]: *pc = length* (*compE2 obj*) **by** *simp*
with *exec* **have** *pc' ≥ length* (*compE2 obj*)
by(*simp add*: *compxEs2-size-convs stack-xlift-compxE2*)(*auto elim*!: *exec-meth-drop-xt-pc*)
then obtain *PC* **where** *PC*: *pc' = PC + length* (*compE2 obj*)
by −(*rule-tac PC34*=*pc' − length* (*compE2 obj*) **in** *that*, *simp*)
from *pc bisimObj* **obtain** *v* **where** [*simp*]: *stk* = [*v*] *xcp* = *None* **by**(*auto dest*: *bisim1-pc-length-compE2D*)
show *?thesis*
proof(*cases ps*)
case *Cons*

```
    with exec pc have exec-meth-d (compP2 P) (compE2 obj @ compEs2 ps)
      (stack-xlift (length STK) (compxE2 obj 0 0 @ compxEs2 ps (length (compE2 obj)) (Suc 0))) t
      h (stk @ STK, loc, length (compE2 obj) + 0, xcp) ta h' (stk', loc', pc', xcp')
      by −(rule exec-meth-take, auto)
    hence ?execs ps [] (v # STK) loc 0 None stk' loc' (pc' − length (compE2 obj)) xcp'
      apply −
      apply(rule exec-meth-drop-xt)
      apply(simp add: compxEs2-size-convs compxEs2-stack-xlift-convs)
      apply(auto simp add: stack-xlift-compxE2)
      done
    from IHparams[OF this] PC obtain stk'' where stk': stk' = stk'' @ v # STK
      and exec': exec-meth-d (compP2 P) (compEs2 ps) (compxEs2 ps 0 0) t h ([], loc, 0, None) ta
h' (stk'', loc', PC, xcp')
      by auto
    from exec' have exec-meth-d (compP2 P) ((compE2 obj @ compEs2 ps) @ [Invoke M' (length
ps)]) (compxE2 obj 0 0 @ shift (length (compE2 obj)) (stack-xlift (length [v]) (compxEs2 ps 0 0))) t
h ([] @ [v], loc, length (compE2 obj) + 0, None) ta h' (stk'' @ [v], loc', length (compE2 obj) + PC,
xcp')
      apply −
      apply(rule exec-meth-append)
      apply(rule append-exec-meth-xt)
      apply(erule exec-meth-stk-offer)
      by(auto)
   thus ?thesis using stk' PC by(clarsimp simp add: shift-compxEs2 stack-xlift-compxEs2 ac-simps)
  next
    case Nil
    with exec pc show ?thesis
     apply(auto elim!: exec-meth.cases intro!: exec-meth.intros simp add: split-beta split: if-split-asm)
      apply(auto split: extCallRet.split-asm intro!: exec-meth.intros)
      apply(force intro!: exI)
      apply(force intro!: exI)
      apply(force intro!: exI)
      done
  qed
 qed
next
 case (bisim1CallParams ps n ps' xs stk loc pc xcp obj M' v)
 note bisimParam = ‹P,ps,h ⊢ (ps',xs) [↔] (stk,loc,pc,xcp)›
 note IHparam = ‹⋀stk' loc' pc' xcp' STK. ?execs ps stk STK loc pc xcp stk' loc' pc' xcp'
          ⟹ ?concls ps stk STK loc pc xcp stk' loc' pc' xcp'›
 note exec = ‹?exec (obj•M'(ps)) (stk @ [v]) STK loc (length (compE2 obj) + pc) xcp stk' loc' pc'
xcp'›
 show ?case
 proof(cases ps)
   case Nil
   with bisimParam have pc = 0 xcp = None by(auto elim: bisims1.cases)
   with exec Nil show ?thesis
    apply(auto elim!: exec-meth.cases intro!: exec-meth.intros simp add: split-beta extRet2JVM-def
split: if-split-asm)
     apply(auto split: extCallRet.split-asm simp add: neq-Nil-conv)
     apply(force intro!: exec-meth.intros)+
     done
 next
   case Cons
```

```
    from bisimParam have pc: pc ≤ length (compEs2 ps) by(rule bisims1-pc-length-compEs2)
    show ?thesis
    proof(cases pc < length (compEs2 ps))
      case True
       from exec have exec-meth-d (compP2 P) ((compE2 obj @ compEs2 ps) @ [Invoke M' (length
ps)])
        (stack-xlift (length STK) (compxE2 obj 0 0) @ shift (length (compE2 obj)) (stack-xlift (length
(v # STK)) (compxEs2 ps 0 0))) t
        h (stk @ v # STK, loc, length (compE2 obj) + pc, xcp) ta h' (stk', loc', pc', xcp')
        by(simp add: compxEs2-size-convs compxEs2-stack-xlift-convs)
       hence exec': exec-meth-d (compP2 P) (compE2 obj @ compEs2 ps) (stack-xlift (length STK)
(compxE2 obj 0 0) @
        shift (length (compE2 obj)) (stack-xlift (length (v # STK)) (compxEs2 ps 0 0))) t
        h (stk @ v # STK, loc, length (compE2 obj) + pc, xcp) ta h' (stk', loc', pc', xcp')
        by(rule exec-meth-take)(simp add: True)
      hence ?execs ps stk (v # STK) loc pc xcp stk' loc' (pc' − length (compE2 obj)) xcp'
        by(rule exec-meth-drop-xt)(auto simp add: stack-xlift-compxE2)
      from IHparam[OF this] obtain stk'' where stk': stk' = stk'' @ v # STK
        and exec'': exec-meth-d (compP2 P) (compEs2 ps) (compxEs2 ps 0 0) t h (stk, loc, pc, xcp) ta
h' (stk'', loc', pc' − length (compE2 obj), xcp') by blast
      from exec'' have exec-meth-d (compP2 P) (compEs2 ps) (stack-xlift (length [v]) (compxEs2 ps 0
0)) t h (stk @ [v], loc, pc, xcp) ta h' (stk'' @ [v], loc', pc' − length (compE2 obj), xcp')
        by(rule exec-meth-stk-offer)
       hence exec-meth-d (compP2 P) (compE2 obj @ compEs2 ps) (compxE2 obj 0 0 @ shift (length
(compE2 obj)) (stack-xlift (length [v]) (compxEs2 ps 0 0))) t
 h (stk @ [v], loc, length (compE2 obj) + pc, xcp) ta h' (stk'' @ [v], loc', length (compE2 obj) + (pc'
− length (compE2 obj)), xcp')
        by(rule append-exec-meth-xt) auto
      hence exec-meth-d (compP2 P) ((compE2 obj @ compEs2 ps) @ [Invoke M' (length ps)])
     (compxE2 obj 0 0 @ shift (length (compE2 obj)) (stack-xlift (length [v]) (compxEs2 ps 0 0))) t
 h (stk @ [v], loc, length (compE2 obj) + pc, xcp) ta h' (stk'' @ [v], loc', length (compE2 obj) + (pc'
− length (compE2 obj)), xcp')
        by(rule exec-meth-append)
      moreover from exec' have pc' ≥ length (compE2 obj)
        by(rule exec-meth-drop-xt-pc)(auto simp add: stack-xlift-compxE2)
      ultimately show ?thesis using stk'
        by(auto simp add: shift-compxEs2 stack-xlift-compxEs2)
    next
      case False
      with pc have pc: pc = length (compEs2 ps) by simp
      with bisimParam obtain vs where stk = vs length vs = length ps xcp = None
        by(auto dest: bisims1-pc-length-compEs2D)
      with exec pc Cons show ?thesis
       apply(auto elim!: exec-meth.cases intro!: exec-meth.intros simp add: split-beta extRet2JVM-def
split: if-split-asm)
        apply(auto simp add: neq-Nil-conv split: extCallRet.split-asm)
        apply(force intro!: exec-meth.intros)+
        done
    qed
  qed
next
  case (bisim1CallThrowObj obj n a xs stk loc pc ps M')
  note bisim = ‹P,obj,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
  note IH = ‹⋀stk' loc' pc' xcp' STK. ?exec obj stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'
```

$\Longrightarrow$ *?concl obj stk STK loc pc* $\lfloor a \rfloor$ *stk' loc' pc' xcp'*$\rangle$
note *exec* = ‹*?exec* (*obj*·*M'*(*ps*)) *stk STK loc pc* $\lfloor a \rfloor$ *stk' loc' pc' xcp'*›
from *bisim* **have** *pc* < *length* (*compE2 obj*) **and** [*simp*]: *xs* = *loc* **by**(*auto dest*: *bisim1-ThrowD*)
with *exec* **have** *?exec obj stk STK loc pc* $\lfloor a \rfloor$ *stk' loc' pc' xcp'*
by(*simp add*: *compxEs2-size-convs compxEs2-stack-xlift-convs*)(*erule exec-meth-take-xt*)
from *IH*[*OF this*] **show** *?case* **by** *auto*
next
case (*bisim1CallThrowParams ps n vs a ps' xs stk loc pc obj M' v*)
note *bisim* = ‹*P*,*ps*,*h* ⊢ (*map Val vs* @ *Throw a* # *ps'*,*xs*) [↔] (*stk*,*loc*,*pc*,$\lfloor a \rfloor$)›
note *IH* = ‹$\bigwedge$*stk' loc' pc' xcp' STK*. *?execs ps stk STK loc pc* $\lfloor a \rfloor$ *stk' loc' pc' xcp'*
$\Longrightarrow$ *?concls ps stk STK loc pc* $\lfloor a \rfloor$ *stk' loc' pc' xcp'*›
note *exec* = ‹*?exec* (*obj*·*M'*(*ps*)) (*stk* @ [*v*]) *STK loc* (*length* (*compE2 obj*) + *pc*) $\lfloor a \rfloor$ *stk' loc' pc' xcp'*›
from *bisim* **have** *pc*: *pc* < *length* (*compEs2 ps*) *loc* = *xs* **by**(*auto dest*: *bisims1-ThrowD*)
from *exec* **have** *exec-meth-d* (*compP2 P*) ((*compE2 obj* @ *compEs2 ps*) @ [*Invoke M'* (*length ps*)])
(*stack-xlift* (*length STK*) (*compxE2 obj 0 0*) @ *shift* (*length* (*compE2 obj*)) (*stack-xlift* (*length* (*v* # *STK*)) (*compxEs2 ps 0 0*))) *t*
h (*stk* @ *v* # *STK*, *loc*, *length* (*compE2 obj*) + *pc*, $\lfloor a \rfloor$) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *compxEs2-size-convs compxEs2-stack-xlift-convs*)
hence *exec'*: *exec-meth-d* (*compP2 P*) (*compE2 obj* @ *compEs2 ps*) (*stack-xlift* (*length STK*) (*compxE2 obj 0 0*) @
shift (*length* (*compE2 obj*)) (*stack-xlift* (*length* (*v* # *STK*)) (*compxEs2 ps 0 0*))) *t*
h (*stk* @ *v* # *STK*, *loc*, *length* (*compE2 obj*) + *pc*, $\lfloor a \rfloor$) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*rule exec-meth-take*)(*simp add*: *pc*)
hence *?execs ps stk* (*v* # *STK*) *loc pc* $\lfloor a \rfloor$ *stk' loc'* (*pc'* − *length* (*compE2 obj*)) *xcp'*
by(*rule exec-meth-drop-xt*)(*auto simp add*: *stack-xlift-compxE2*)
from *IH*[*OF this*] **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ *v* # *STK*
and *exec''*: *exec-meth-d* (*compP2 P*) (*compEs2 ps*) (*compxEs2 ps 0 0*) *t h* (*stk*, *loc*, *pc*, $\lfloor a \rfloor$) *ta h'*
(*stk''*, *loc'*, *pc'* − *length* (*compE2 obj*), *xcp'*) **by** *auto*
from *exec''* **have** *exec-meth-d* (*compP2 P*) ((*compE2 obj* @ *compEs2 ps*) @ [*Invoke M'* (*length ps*)])
(*compxE2 obj 0 0* @ *shift* (*length* (*compE2 obj*)) (*stack-xlift* (*length* [*v*]) (*compxEs2 ps 0 0*))) *t*
h (*stk* @ [*v*], *loc*, *length* (*compE2 obj*) + *pc*, $\lfloor a \rfloor$) *ta h'* (*stk''* @ [*v*], *loc'*, *length* (*compE2 obj*) +
(*pc'* − *length* (*compE2 obj*)), *xcp'*)
apply −
apply(*rule exec-meth-append*)
apply(*rule append-exec-meth-xt*)
apply(*erule exec-meth-stk-offer*)
apply *auto*
done
moreover from *exec'* **have** *pc'* ≥ *length* (*compE2 obj*)
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
ultimately show *?case* **using** *stk'* **by**(*auto simp add*: *compxEs2-size-convs compxEs2-stack-xlift-convs*)
next
case *bisim1CallThrow* **thus** *?case*
by(*auto elim*!: *exec-meth.cases dest*: *match-ex-table-pcsD simp add*: *stack-xlift-compxEs2 stack-xlift-compxE2*)
next
case *bisim1BlockSome1* **thus** *?case*
by(*fastforce elim*: *exec-meth.cases intro*: *exec-meth.intros*)
next
case *bisim1BlockSome2* **thus** *?case*
by(*fastforce elim*: *exec-meth.cases intro*: *exec-meth.intros*)
next
case (*bisim1BlockSome4 e n e' xs stk loc pc xcp V T v*)
note *IH* = ‹$\bigwedge$*stk' loc' pc' xcp' STK*. *?exec e stk STK loc pc xcp stk' loc' pc' xcp'*

$\Longrightarrow$ *?concl e stk STK loc pc xcp stk′ loc′ pc′ xcp′*⟩
note *bisim* = ⟨*P*,*e*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)⟩
note *exec* = ⟨*?exec* {*V*:*T*=⌊*v*⌋; *e*} *stk STK loc* (*Suc* (*Suc pc*)) *xcp stk′ loc′ pc′ xcp′*⟩
hence *exec′*: *exec-meth-d* (*compP2 P*) ([*Push v*, *Store V*] @ *compE2 e*)
(*shift* (*length* [*Push v*, *Store V*]) (*stack-xlift* (*length STK*) (*compxE2 e 0 0*))) *t h* (*stk* @ *STK*, *loc*, *length* [*Push v*, *Store V*] + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *compxE2-size-convs*)
hence *?exec e stk STK loc pc xcp stk′ loc′* (*pc′* − *length* [*Push v*, *Store V*]) *xcp′*
by(*rule exec-meth-drop*) *auto*
from *IH*[*OF this*] **obtain** *stk″* **where** *stk′*: *stk′* = *stk″* @ *STK*
and *exec″*: *exec-meth-d* (*compP2 P*) (*compE2 e*) (*compxE2 e 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta*
h′ (*stk″*, *loc′*, *pc′* − *length* [*Push v*, *Store V*], *xcp′*) **by** *auto*
from *exec″* **have** *exec-meth-d* (*compP2 P*) ([*Push v*, *Store V*] @ *compE2 e*)
(*shift* (*length* [*Push v*, *Store V*]) (*compxE2 e 0 0*)) *t h* (*stk*, *loc*, *length* [*Push v*, *Store V*] + *pc*, *xcp*) *ta h′* (*stk″*, *loc′*, *length* [*Push v*, *Store V*] + (*pc′* − *length* [*Push v*, *Store V*]), *xcp′*)
by(*rule append-exec-meth*) *auto*
moreover from *exec′* **have** *pc′* ≥ *length* [*Push v*, *Store V*]
by(*rule exec-meth-drop-pc*)(*auto simp add*: *stack-xlift-compxE2*)
hence *Suc* (*Suc* (*pc′* − *Suc* (*Suc* 0))) = *pc′* **by**(*simp*)
ultimately show *?case* **using** *stk′* **by**(*auto simp add*: *compxE2-size-convs*)
next
case (*bisim1BlockThrowSome e n a xs stk loc pc V T v*)
note *IH* = ⟨⋀*stk′ loc′ pc′ xcp′ STK*. *?exec e stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*
$\Longrightarrow$ *?concl e stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*⟩
note *exec* = ⟨*?exec* {*V*:*T*=⌊*v*⌋; *e*} *stk STK loc* (*Suc* (*Suc pc*)) ⌊*a*⌋ *stk′ loc′ pc′ xcp′*⟩
hence *exec′*: *exec-meth-d* (*compP2 P*) ([*Push v*, *Store V*] @ *compE2 e*)
(*shift* (*length* [*Push v*, *Store V*]) (*stack-xlift* (*length STK*) (*compxE2 e 0 0*))) *t h* (*stk* @ *STK*, *loc*, *length* [*Push v*, *Store V*] + *pc*, ⌊*a*⌋) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *compxE2-size-convs*)
hence *?exec e stk STK loc pc* ⌊*a*⌋ *stk′ loc′* (*pc′* − *length* [*Push v*, *Store V*]) *xcp′*
by(*rule exec-meth-drop*) *auto*
from *IH*[*OF this*] **obtain** *stk″* **where** *stk′*: *stk′* = *stk″* @ *STK*
and *exec″*: *exec-meth-d* (*compP2 P*) (*compE2 e*) (*compxE2 e 0 0*) *t h* (*stk*, *loc*, *pc*, ⌊*a*⌋) *ta*
h′ (*stk″*, *loc′*, *pc′* − *length* [*Push v*, *Store V*], *xcp′*) **by** *auto*
from *exec″* **have** *exec-meth-d* (*compP2 P*) ([*Push v*, *Store V*] @ *compE2 e*)
(*shift* (*length* [*Push v*, *Store V*]) (*compxE2 e 0 0*)) *t h* (*stk*, *loc*, *length* [*Push v*, *Store V*] + *pc*, ⌊*a*⌋) *ta h′* (*stk″*, *loc′*, *length* [*Push v*, *Store V*] + (*pc′* − *length* [*Push v*, *Store V*]), *xcp′*)
by(*rule append-exec-meth*) *auto*
moreover from *exec′* **have** *pc′* ≥ *length* [*Push v*, *Store V*]
by(*rule exec-meth-drop-pc*)(*auto simp add*: *stack-xlift-compxE2*)
hence *Suc* (*Suc* (*pc′* − *Suc* (*Suc* 0))) = *pc′* **by**(*simp*)
ultimately show *?case* **using** *stk′* **by**(*auto simp add*: *compxE2-size-convs*)
next
case *bisim1BlockNone* **thus** *?case*
by(*fastforce elim*: *exec-meth.cases intro*: *exec-meth.intros*)
next
case *bisim1BlockThrowNone* **thus** *?case*
by(*fastforce elim*: *exec-meth.cases intro*: *exec-meth.intros*)
next
case (*bisim1Sync1 e1 n e1′ xs stk loc pc xcp e2 V*)
note *bisim* = ⟨*P*,*e1*,*h* ⊢ (*e1′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)⟩
note *IH* = ⟨⋀*stk′ loc′ pc′ xcp′ STK*. *?exec e1 stk STK loc pc xcp stk′ loc′ pc′ xcp′*
$\Longrightarrow$ *?concl e1 stk STK loc pc xcp stk′ loc′ pc′ xcp′*⟩
note *exec* = ⟨*?exec* (*sync*$_V$ (*e1*) *e2*) *stk STK loc pc xcp stk′ loc′ pc′ xcp′*⟩

from *bisim* **have** *pc*: *pc* ≤ *length* (*compE2 e1*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e1*))
case *True*
from *exec* **have** *exec-meth-d* (*compP2 P*)
(*compE2 e1* @ (*Dup* # *Store V* # *MEnter* # *compE2 e2* @ [*Load V*, *MExit*, *Goto 4*, *Load V*, *MExit*, *ThrowExc*]))
(*stack-xlift* (*length STK*) (*compxE2 e1 0 0*) @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length STK*) (*compxE2 e2 3 0*) @
stack-xlift (*length STK*) [(*3*, *3* + *length* (*compE2 e2*), *None*, *6* + *length* (*compE2 e2*), *0*)])) *t*
h (*stk* @ *STK*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps*)
hence *?exec e1 stk STK loc pc xcp stk' loc' pc' xcp'*
by(*rule exec-meth-take-xt*)(*rule True*)
from *IH*[*OF this*] **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ *STK*
and *exec'*: *exec-meth-d* (*compP2 P*) (*compE2 e1*) (*compxE2 e1 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *pc'*, *xcp'*)
by *blast*
from *exec'* **have** *exec-meth-d* (*compP2 P*) (*compE2 e1* @ (*Dup* # *Store V* # *MEnter* # *compE2 e2* @ [*Load V*, *MExit*, *Goto 4*, *Load V*, *MExit*, *ThrowExc*]))
(*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*compxE2 e2 3 0* @ [(*3*, *3* + *length* (*compE2 e2*), *None*, *6* + *length* (*compE2 e2*), *0*)])) *t*
h (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *pc'*, *xcp'*)
by(*rule exec-meth-append-xt*)
thus *?thesis* **using** *stk'* **by**(*simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps*)
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compE2 e1*) **by** *simp*
with *bisim* **obtain** *v* **where** *stk* = [*v*] *xcp* = *None* **by**(*auto dest*: *bisim1-pc-length-compE2D*)
thus *?thesis* **using** *exec* **by**(*auto elim*!: *exec-meth.cases intro*!: *exec-meth.intros*)
qed
next
case *bisim1Sync2* **thus** *?case*
by(*fastforce elim*!: *exec-meth.cases intro*!: *exec-meth.intros*)
next
case *bisim1Sync3* **thus** *?case*
by(*fastforce elim*!: *exec-meth.cases intro*!: *exec-meth.intros split*: *if-split-asm*)
next
case (*bisim1Sync4 e2 n e2' xs stk loc pc xcp e1 V a*)
note *IH* = ‹⋀*stk' loc' pc' xcp' STK*. *?exec e2 stk STK loc pc xcp stk' loc' pc' xcp'*
⟹ *?concl e2 stk STK loc pc xcp stk' loc' pc' xcp'*›
note *bisim* = ‹*P*,*e2*,*h* ⊢ (*e2'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *exec* = ‹*?exec* (*sync*$_V$ (*e1*) *e2*) *stk STK loc* (*Suc* (*Suc* (*Suc* (*length* (*compE2 e1*) + *pc*)))) *xcp stk' loc' pc' xcp'*›
from *bisim* **have** *pc*: *pc* ≤ *length* (*compE2 e2*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e2*))
case *True*
let *?pre* = *compE2 e1* @ [*Dup*, *Store V*, *MEnter*]
from *exec* **have** *exec'*: *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e2* @ [*Load V*, *MExit*, *Goto 4*, *Load V*, *MExit*, *ThrowExc*])
(*stack-xlift* (*length STK*) (*compxE2 e1 0 0*) @ *shift* (*length ?pre*) (*stack-xlift* (*length STK*) (*compxE2 e2 0 0*) @
[(*0*, *length* (*compE2 e2*), *None*, *3* + *length* (*compE2 e2*), *length STK*)])) *t*

h (*stk* @ *STK*, *loc*, *length ?pre* + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *stack-xlift-compxE2 shift-compxE2 eval-nat-numeral ac-simps*)
hence *exec''*: *exec-meth-d* (*compP2 P*) (*compE2 e2* @ [*Load V*, *MExit*, *Goto 4*, *Load V*, *MExit*, *ThrowExc*])
(*stack-xlift* (*length STK*) (*compxE2 e2 0 0*) @ [(*0*, *length* (*compE2 e2*), *None*, *3* + *length* (*compE2 e2*), *length STK*)]) *t*
h (*stk* @ *STK*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'* − *length ?pre*, *xcp'*)
by(*rule exec-meth-drop-xt*[**where** *n=1*])(*auto simp add*: *stack-xlift-compxE2*)
from *exec'* **have** *pc'*: *pc'* ≥ *length ?pre*
by(*rule exec-meth-drop-xt-pc*[**where** *n'=1*])(*auto simp add*: *stack-xlift-compxE2*)
hence *pc''*: (*Suc* (*Suc* (*Suc* (*pc'* − *Suc* (*Suc* (*Suc 0*)))))) = *pc'* **by** *simp*
show *?thesis*
proof(*cases xcp*)
case *None*
from *exec'' None True*
have *?exec e2 stk STK loc pc xcp stk' loc'* (*pc'* − *length ?pre*) *xcp'*
apply −
apply (*erule exec-meth.cases*)
apply (*cases compE2 e2* ! *pc*)
apply (*fastforce simp add*: *is-Ref-def intro*: *exec-meth.intros split*: *if-split-asm cong del*: *image-cong-simp*)+
done
from *IH*[*OF this*] **obtain** *stk''* **where** *stk*: *stk'* = *stk''* @ *STK*
and *exec'''*: *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*)
ta h' (*stk''*, *loc'*, *pc'* − *length ?pre*, *xcp'*) **by** *blast*
from *exec'''* **have** *exec-meth-d* (*compP2 P*) (*compE2 e2* @ [*Load V*, *MExit*, *Goto 4*, *Load V*, *MExit*, *ThrowExc*])
(*compxE2 e2 0 0* @ [(*0*, *length* (*compE2 e2*), *None*, *3* + *length* (*compE2 e2*), *0*)]) *t*
h (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *pc'* − *length ?pre*, *xcp'*)
by(*rule exec-meth-append-xt*)
hence *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e2* @ [*Load V*, *MExit*, *Goto 4*, *Load V*, *MExit*, *ThrowExc*])
(*compxE2 e1 0 0* @ *shift* (*length ?pre*) (*compxE2 e2 0 0* @ [(*0*, *length* (*compE2 e2*), *None*, *3* + *length* (*compE2 e2*), *0*)])) *t*
h (*stk*, *loc*, *length ?pre* + *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *length ?pre* + (*pc'* − *length ?pre*), *xcp'*)
by(*rule append-exec-meth-xt*[**where** *n=1*]) *auto*
thus *?thesis* **using** *stk pc' pc''* **by**(*simp add*: *eval-nat-numeral shift-compxE2 ac-simps*)
next
case (*Some a*)
with *exec''* **have** [*simp*]: *h'* = *h xcp'* = *None loc'* = *loc ta* = *ε*
by(*auto elim*!: *exec-meth.cases simp add*: *match-ex-table-append*
split: *if-split-asm dest*!: *match-ex-table-stack-xliftD*)
show *?thesis*
proof(*cases match-ex-table* (*compP2 P*) (*cname-of h a*) *pc* (*compxE2 e2 0 0*))
case *None*
with *Some exec'' True* **have** [*simp*]: *stk'* = *Addr a* # *STK*
and *pc'*: *pc'* = *length* (*compE2 e1*) + *length* (*compE2 e2*) + *6*
by(*auto elim*!: *exec-meth.cases simp add*: *match-ex-table-append*
split: *if-split-asm dest*!: *match-ex-table-stack-xliftD*)
with *exec'' Some None*
have *exec-meth-d* (*compP2 P*) (*compE2 e2* @ [*Load V*, *MExit*, *Goto 4*, *Load V*, *MExit*, *ThrowExc*])
(*compxE2 e2 0 0* @ [(*0*, *length* (*compE2 e2*), *None*, *3* + *length* (*compE2 e2*), *0*)]) *t*
h (*stk*, *loc*, *pc*, ⌊*a*⌋) *ε h* (*Addr a* # *drop* (*length stk* − *0*) *stk*, *loc*, *pc'* − *length ?pre*, *None*)

by −(*rule exec-catch*, *auto elim*!: *exec-meth.cases simp add*: *match-ex-table-append matches-ex-entry-def split*: *if-split-asm dest*!: *match-ex-table-stack-xliftD*)
hence *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e2* @ [*Load V*, *MExit*, *Goto 4*, *Load V*, *MExit*, *ThrowExc*])
(*compxE2 e1 0 0* @ *shift* (*length ?pre*) (*compxE2 e2 0 0* @ [(*0*, *length* (*compE2 e2*), *None*, *3* + *length* (*compE2 e2*), *0*)])) *t*
h (*stk*, *loc*, *length ?pre* + *pc*, ⌊*a*⌋) *ε h* (*Addr a* # *drop* (*length stk* − *0*) *stk*, *loc*,
length ?pre + (*pc′* − *length ?pre*), *None*)
by(*rule append-exec-meth-xt*[**where** *n=1*]) *auto*
with *pc′ Some* **show** *?thesis* **by**(*simp add*: *eval-nat-numeral shift-compxE2 ac-simps*)
next
case (*Some pcd*)
with ‹*xcp* = ⌊*a*⌋› *exec″ True*
have *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t*
h (*stk*, *loc*, *pc*, ⌊*a*⌋) *ε h* (*Addr a* # *drop* (*length stk* − *snd pcd*) *stk*, *loc*, *pc′* − *length ?pre*, *None*)
apply −
apply(*rule exec-catch*)
apply(*auto elim*!: *exec-meth.cases simp add*: *match-ex-table-append split*: *if-split-asm dest*!: *match-ex-table-stack-xliftD*)
done
hence *exec-meth-d* (*compP2 P*) (*compE2 e2* @ [*Load V*, *MExit*, *Goto 4*, *Load V*, *MExit*, *ThrowExc*]) (*compxE2 e2 0 0* @ [(*0*, *length* (*compE2 e2*), *None*, *3* + *length* (*compE2 e2*), *0*)]) *t*
h (*stk*, *loc*, *pc*, ⌊*a*⌋) *ε h* (*Addr a* # *drop* (*length stk* − *snd pcd*) *stk*, *loc*, *pc′* − *length ?pre*, *None*)
by(*rule exec-meth-append-xt*)
hence *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e2* @ [*Load V*, *MExit*, *Goto 4*, *Load V*, *MExit*, *ThrowExc*])
(*compxE2 e1 0 0* @ *shift* (*length ?pre*) (*compxE2 e2 0 0* @ [(*0*, *length* (*compE2 e2*), *None*, *3* + *length* (*compE2 e2*), *0*)])) *t*
h (*stk*, *loc*, *length ?pre* + *pc*, ⌊*a*⌋) *ε h* (*Addr a* # *drop* (*length stk* − *snd pcd*) *stk*, *loc*, *length ?pre* + (*pc′* − *length ?pre*), *None*)
by(*rule append-exec-meth-xt*[**where** *n=1*])(*auto*)
moreover from *Some* ‹*xcp* = ⌊*a*⌋› *exec″ True pc′*
have *pc′* = *length* (*compE2 e1*) + *3* + *fst pcd stk′* = *Addr a* # *drop* (*length stk* − *snd pcd*) *stk* @ *STK*
by(*auto elim*!: *exec-meth.cases dest*!: *match-ex-table-stack-xliftD simp*: *match-ex-table-append split*: *if-split-asm*)
ultimately show *?thesis* **using** ‹*xcp* = ⌊*a*⌋› **by**(*auto simp add*: *eval-nat-numeral shift-compxE2 ac-simps*)
qed
qed
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compE2 e2*) **by** *simp*
with *exec* **show** *?thesis*
by(*auto elim*!: *exec-meth.cases intro*!: *exec-meth.intros split*: *if-split-asm simp add*: *match-ex-table-append-not-pc eval-nat-numeral*)(*simp-all add*: *matches-ex-entry-def*)
qed
next
case *bisim1Sync5* **thus** *?case*
by(*fastforce elim*: *exec-meth.cases intro*: *exec-meth.intros split*: *if-split-asm*)
next
case *bisim1Sync6* **thus** *?case*
by(*fastforce elim*: *exec-meth.cases intro*: *exec-meth.intros split*: *if-split-asm*)

next
case *bisim1Sync7* **thus** *?case*
by(*fastforce elim*: *exec-meth.cases intro*: *exec-meth.intros split*: *if-split-asm*)
next
case *bisim1Sync8* **thus** *?case*
by(*fastforce elim*: *exec-meth.cases intro*: *exec-meth.intros split*: *if-split-asm*)
next
case (*bisim1Sync9 e1 n e2 V a xs*)
note *exec* = ‹*?exec* (*sync*$_V$ (*e1*) *e2*) [*Addr a*] *STK xs* (*8* + *length* (*compE2 e1*) + *length* (*compE2 e2*)) *None stk′ loc′ pc′ xcp′*›
let *?pre* = *compE2 e1* @ *Dup* # *Store V* # *MEnter* # *compE2 e2* @ [*Load V*, *MExit*, *Goto 4*, *Load V*, *MExit*]
from *exec* **have** *exec′*: *exec-meth-d* (*compP2 P*) (*?pre* @ [*ThrowExc*]) (*stack-xlift* (*length STK*) (*compxE2* (*sync*$_V$ (*e1*) *e2*) *0 0*) @ *shift* (*length ?pre*) []) *t h* (*Addr a* # *STK*, *xs*, *length ?pre* + *0*, *None*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *eval-nat-numeral*)
hence *exec-meth-d* (*compP2 P*) [*ThrowExc*] [] *t h* (*Addr a* # *STK*, *xs*, *0*, *None*) *ta h′* (*stk′*, *loc′*, *pc′* − *length ?pre*, *xcp′*)
by(*rule exec-meth-drop-xt*)(*auto simp add*: *stack-xlift-compxE2*)
moreover from *exec′* **have** *pc′* = *8* + *length* (*compE2 e1*) + *length* (*compE2 e2*) *stk′* = *Addr a* # *STK*
by(*auto elim*!: *exec-meth.cases*)
ultimately show *?case* **by**(*fastforce elim*!: *exec-meth.cases intro*: *exec-meth.intros*)
next
case (*bisim1Sync10 e1 n e2 V a xs*)
note *exec* = ‹*?exec* (*sync*$_V$ (*e1*) *e2*) [*Addr a*] *STK xs* (*8* + *length* (*compE2 e1*) + *length* (*compE2 e2*)) ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
hence *match-ex-table* (*compP2 P*) (*cname-of h a*) (*8* + *length* (*compE2 e1*) + *length* (*compE2 e2*)) (*stack-xlift* (*length STK*) (*compxE2* (*sync*$_V$ (*e1*) *e2*) *0 0*)) ≠ *None*
by(*rule exec-meth.cases*) *auto*
hence *False* **by**(*auto split*: *if-split-asm simp add*: *match-ex-table-append-not-pcs*)(*simp add*: *matches-ex-entry-def*)
thus *?case* **..**
next
case (*bisim1Sync11 e1 n e2 V xs*)
note *exec* = ‹*?exec* (*sync*$_V$ (*e1*) *e2*) [*Null*] *STK xs* (*Suc* (*Suc* (*length* (*compE2 e1*)))) ⌊*addr-of-sys-xcpt NullPointer*⌋ *stk′ loc′ pc′ xcp′*›
hence *match-ex-table* (*compP2 P*) (*cname-of h* (*addr-of-sys-xcpt NullPointer*)) (*2* + *length* (*compE2 e1*)) (*stack-xlift* (*length STK*) (*compxE2* (*sync*$_V$ (*e1*) *e2*) *0 0*)) ≠ *None*
by(*rule exec-meth.cases*)(*auto split*: *if-split-asm*)
hence *False* **by**(*auto split*: *if-split-asm simp add*: *match-ex-table-append-not-pcs*)(*simp add*: *matches-ex-entry-def*)
thus *?case* **..**
next
case (*bisim1SyncThrow e1 n a xs stk loc pc e2 V*)
note *exec* = ‹*?exec* (*sync*$_V$ (*e1*) *e2*) *stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
note *IH* = ‹⋀*stk′ loc′ pc′ xcp′ STK*. *?exec e1 stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*
⟹ *?concl e1 stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
note *bisim* = ‹*P*,*e1*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
from *bisim* **have** *pc*: *pc* < *length* (*compE2 e1*)
and [*simp*]: *loc* = *xs* **by**(*auto dest*: *bisim1-ThrowD*)
from *exec* **have** *exec-meth-d* (*compP2 P*) (*compE2 e1* @ *Dup* # *Store V* # *MEnter* # *compE2 e2* @ [*Load V*, *MExit*, *Goto 4*, *Load V*, *MExit*, *ThrowExc*])
(*stack-xlift* (*length STK*) (*compxE2 e1 0 0*) @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length STK*) (*compxE2 e2 3 0*) @
[(*3*, *3* + *length* (*compE2 e2*), *None*, *6* + *length* (*compE2 e2*), *length STK*)])) *t*

```
    h (stk @ STK, loc, pc, ⌊a⌋) ta h' (stk', loc', pc', xcp')
   by(simp add: shift-compxE2 stack-xlift-compxE2 ac-simps)
  hence ?exec e1 stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'
   by(rule exec-meth-take-xt)(rule pc)
  from IH[OF this] show ?case by auto
next
  case (bisim1Seq1 e1 n e1' xs stk loc pc xcp e2)
  note bisim1 = ‹P,e1,h ⊢ (e1', xs) ↔ (stk, loc, pc, xcp)›
  note IH = ‹⋀stk' loc' pc' xcp' STK. ?exec e1 stk STK loc pc xcp stk' loc' pc' xcp'
          ⟹ ?concl e1 stk STK loc pc xcp stk' loc' pc' xcp'›
  note exec = ‹?exec (e1;;e2) stk STK loc pc xcp stk' loc' pc' xcp'›
  from bisim1 have pc: pc ≤ length (compE2 e1) by(rule bisim1-pc-length-compE2)
  show ?case
  proof(cases pc < length (compE2 e1))
    case True
    from exec have exec-meth-d (compP2 P) (compE2 e1 @ Pop # compE2 e2) (stack-xlift (length
STK) (compxE2 e1 0 0) @
      shift (length (compE2 e1)) (stack-xlift (length STK) (compxE2 e2 (Suc 0) 0))) t
     h (stk @ STK, loc, pc, xcp) ta h' (stk', loc', pc', xcp')
     by(simp add: shift-compxE2 stack-xlift-compxE2)
    hence ?exec e1 stk STK loc pc xcp stk' loc' pc' xcp'
     by(rule exec-meth-take-xt)(rule True)
    from IH[OF this] show ?thesis by auto
  next
    case False
    with pc have [simp]: pc = length (compE2 e1) by simp
    with bisim1 obtain v where xcp = None stk = [v] by(auto dest: bisim1-pc-length-compE2D)
    with exec show ?thesis by(fastforce elim: exec-meth.cases intro: exec-meth.intros)
  qed
next
  case (bisim1SeqThrow1 e1 n a xs stk loc pc e2)
  note bisim1 = ‹P,e1,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
  note IH = ‹⋀stk' loc' pc' xcp' STK. ?exec e1 stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'
          ⟹ ?concl e1 stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'›
  note exec = ‹?exec (e1;;e2) stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'›
  from bisim1 have pc: pc < length (compE2 e1) by(auto dest: bisim1-ThrowD)
  from exec have exec-meth-d (compP2 P) (compE2 e1 @ Pop # compE2 e2) (stack-xlift (length
STK) (compxE2 e1 0 0) @
      shift (length (compE2 e1)) (stack-xlift (length STK) (compxE2 e2 (Suc 0) 0))) t
     h (stk @ STK, loc, pc, ⌊a⌋) ta h' (stk', loc', pc', xcp')
   by(simp add: shift-compxE2 stack-xlift-compxE2)
  hence ?exec e1 stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'
   by(rule exec-meth-take-xt)(rule pc)
  from IH[OF this] show ?case by(fastforce elim: exec-meth.cases intro: exec-meth.intros)
next
  case (bisim1Seq2 e2 n e2' xs stk loc pc xcp e1)
  note bisim2 = ‹P,e2,h ⊢ (e2', xs) ↔ (stk, loc, pc, xcp)›
  note IH = ‹⋀stk' loc' pc' xcp' STK. ?exec e2 stk STK loc pc xcp stk' loc' pc' xcp'
          ⟹ ?concl e2 stk STK loc pc xcp stk' loc' pc' xcp'›
  note exec = ‹?exec (e1;;e2) stk STK loc (Suc (length (compE2 e1) + pc)) xcp stk' loc' pc' xcp'›
  from bisim2 have pc: pc ≤ length (compE2 e2) by(rule bisim1-pc-length-compE2)
  show ?case
  proof(cases pc < length (compE2 e2))
    case False
```

```
    with pc have [simp]: pc = length (compE2 e2) by simp
    from bisim2 have xcp = None by(auto dest: bisim1-pc-length-compE2D)
    with exec have False by(auto elim: exec-meth.cases)
    thus ?thesis ..
  next
    case True
    from exec have exec':
     exec-meth-d (compP2 P) ((compE2 e1 @ [Pop]) @ compE2 e2) (stack-xlift (length STK) (compxE2
e1 0 0) @
      shift (length (compE2 e1 @ [Pop])) (stack-xlift (length STK) (compxE2 e2 0 0))) t
     h (stk @ STK, loc, length ((compE2 e1) @ [Pop]) + pc, xcp) ta h' (stk', loc', pc', xcp')
      by(simp add: compxE2-size-convs)
    hence ?exec e2 stk STK loc pc xcp stk' loc' (pc' − length ((compE2 e1) @ [Pop])) xcp'
      by(rule exec-meth-drop-xt)(auto simp add: stack-xlift-compxE2)
    from IH[OF this] obtain stk'' where stk': stk' = stk'' @ STK
      and exec'': exec-meth-d (compP2 P) (compE2 e2) (compxE2 e2 0 0) t h (stk, loc, pc, xcp)
      ta h' (stk'', loc', pc' − length (compE2 e1 @ [Pop]), xcp') by auto
    from exec'' have exec-meth-d (compP2 P) ((compE2 e1 @ [Pop]) @ compE2 e2) (compxE2 e1 0
0 @ shift (length (compE2 e1 @ [Pop])) (compxE2 e2 0 0)) t
      h (stk, loc, length ((compE2 e1) @ [Pop]) + pc, xcp) ta h' (stk'', loc', length ((compE2 e1) @
[Pop]) + (pc' − length ((compE2 e1) @ [Pop])), xcp')
      by(rule append-exec-meth-xt) auto
    moreover from exec' have pc' ≥ length ((compE2 e1) @ [Pop])
      by(rule exec-meth-drop-xt-pc)(auto simp add: stack-xlift-compxE2)
    ultimately show ?thesis using stk' by(auto simp add: shift-compxE2 stack-xlift-compxE2)
  qed
next
  case (bisim1Cond1 e n e' xs stk loc pc xcp e1 e2)
  note bisim = ‹P,e,h ⊢ (e', xs) ↔ (stk, loc, pc, xcp)›
  note IH = ‹⋀stk' loc' pc' xcp' STK. ?exec e stk STK loc pc xcp stk' loc' pc' xcp'
          ⟹ ?concl e stk STK loc pc xcp stk' loc' pc' xcp'›
  note exec = ‹?exec (if (e) e1 else e2) stk STK loc pc xcp stk' loc' pc' xcp'›
  from bisim have pc: pc ≤ length (compE2 e) by(rule bisim1-pc-length-compE2)
  show ?case
  proof(cases pc < length (compE2 e))
    case True
    from exec have exec-meth-d (compP2 P) (compE2 e @ IfFalse (2 + int (length (compE2 e1))) #
compE2 e1 @ Goto (1 + int (length (compE2 e2))) # compE2 e2)
     (stack-xlift (length STK) (compxE2 e 0 0) @ shift (length (compE2 e)) (stack-xlift (length STK)
(compxE2 e1 (Suc 0) 0) @
      stack-xlift (length STK) (compxE2 e2 (Suc (Suc (length (compE2 e1)))) 0))) t
     h (stk @ STK, loc, pc, xcp) ta h' (stk', loc', pc', xcp')
      by(simp add: stack-xlift-compxE2 shift-compxE2 ac-simps)
    hence ?exec e stk STK loc pc xcp stk' loc' pc' xcp'
      by(rule exec-meth-take-xt)(rule True)
    from IH[OF this] show ?thesis by auto
  next
    case False
    with pc have [simp]: pc = length (compE2 e) by simp
    from bisim obtain v where stk = [v] xcp = None
      by(auto dest: bisim1-pc-length-compE2D)
    with exec show ?thesis by(auto elim!: exec-meth.cases intro!: exec-meth.intros)
  qed
next
```

case (*bisim1CondThen e1 n e1' xs stk loc pc xcp e e2*)
note *bisim* = ‹*P*,*e1*,*h* ⊢ (*e1'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *IH* = ‹⋀*stk' loc' pc' xcp' STK*. *?exec e1 stk STK loc pc xcp stk' loc' pc' xcp'*
⟹ *?concl e1 stk STK loc pc xcp stk' loc' pc' xcp'›
note *exec* = ‹*?exec* (*if* (*e*) *e1 else e2*) *stk STK loc* (*Suc* (*length* (*compE2 e*) + *pc*)) *xcp stk' loc' pc' xcp'›
from *bisim* **have** *pc*: *pc* ≤ *length* (*compE2 e1*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e1*))
case *True*
let *?pre* = *compE2 e* @ [*IfFalse* (2 + *int* (*length* (*compE2 e1*)))]
from *exec* **have** *exec'*: *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e1* @ *Goto* (1 + *int* (*length* (*compE2 e2*))) # *compE2 e2*)
(*stack-xlift* (*length STK*) (*compxE2 e 0 0*) @ *shift* (*length ?pre*) (*stack-xlift* (*length STK*) (*compxE2 e1 0 0*) @
shift (*length* (*compE2 e1*)) (*stack-xlift* (*length STK*) (*compxE2 e2* (*Suc 0*) *0*)))) *t*
h (*stk* @ *STK*, *loc*, *length ?pre* + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *stack-xlift-compxE2 shift-compxE2 ac-simps*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e1* @ *Goto* (1 + *int* (*length* (*compE2 e2*))) # *compE2 e2*)
(*stack-xlift* (*length STK*) (*compxE2 e1 0 0*) @ *shift* (*length* (*compE2 e1*)) (*stack-xlift* (*length STK*) (*compxE2 e2* (*Suc 0*) *0*))) *t*
h (*stk* @ *STK*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'* − *length ?pre*, *xcp'*)
by(*rule exec-meth-drop-xt*)(*auto simp add*: *stack-xlift-compxE2*)
hence *?exec e1 stk STK loc pc xcp stk' loc'* (*pc'* − *length ?pre*) *xcp'*
by(*rule exec-meth-take-xt*)(*rule True*)
from *IH*[*OF this*] **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ *STK*
and *exec''*: *exec-meth-d* (*compP2 P*) (*compE2 e1*) (*compxE2 e1 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*)
ta h' (*stk''*, *loc'*, *pc'* − *length ?pre*, *xcp'*) **by** *blast*
from *exec''* **have** *exec-meth-d* (*compP2 P*) (*compE2 e1* @ *Goto* (1 + *int* (*length* (*compE2 e2*))) # *compE2 e2*)
(*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*compxE2 e2* (*Suc 0*) *0*)) *t*
h (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *pc'* − *length ?pre*, *xcp'*)
by(*rule exec-meth-append-xt*)
hence *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e1* @ *Goto* (1 + *int* (*length* (*compE2 e2*))) # *compE2 e2*)
(*compxE2 e 0 0* @ *shift* (*length ?pre*) (*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*compxE2 e2* (*Suc 0*) *0*))) *t*
h (*stk*, *loc*, *length ?pre* + *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *length ?pre* + (*pc'* − *length ?pre*), *xcp'*)
by(*rule append-exec-meth-xt*)(*auto*)
moreover from *exec'* **have** *pc'* ≥ *length ?pre*
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
ultimately show *?thesis* **using** *stk'*
by(*auto simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps*)
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compE2 e1*) **by** *simp*
from *bisim* **obtain** *v* **where** *stk* = [*v*] *xcp* = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *exec* **show** *?thesis* **by**(*auto elim*!: *exec-meth.cases intro*!: *exec-meth.intros*)
qed
next
case (*bisim1CondElse e2 n e2' xs stk loc pc xcp e e1*)
note *IH* = ‹⋀*stk' loc' pc' xcp' STK*. *?exec e2 stk STK loc pc xcp stk' loc' pc' xcp'*

$\Longrightarrow$ *?concl e2 stk STK loc pc xcp stk′ loc′ pc′ xcp′*›
note *exec* = ‹*?exec* (*if* (*e*) *e1 else e2*) *stk STK loc* (*Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + *pc*))) *xcp stk′ loc′ pc′ xcp′*›
note *bisim* = ‹*P*,*e2*,*h* ⊢ (*e2′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
from *bisim* **have** *pc*: *pc* ≤ *length* (*compE2 e2*) **by**(*rule bisim1-pc-length-compE2*)

let *?pre* = *compE2 e* @ *IfFalse* (2 + *int* (*length* (*compE2 e1*))) # *compE2 e1* @ [*Goto* (1 + *int* (*length* (*compE2 e2*)))]
from *exec* **have** *exec′*: *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e2*)
(*stack-xlift* (*length STK*) (*compxE2 e 0 0* @ *compxE2 e1* (*Suc* (*length* (*compE2 e*))) *0*) @
shift (*length ?pre*) (*stack-xlift* (*length STK*) (*compxE2 e2 0 0*))) *t*
h (*stk* @ *STK*, *loc*, *length ?pre* + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *stack-xlift-compxE2 shift-compxE2 ac-simps*)
hence *?exec e2 stk STK loc pc xcp stk′ loc′* (*pc′* − *length ?pre*) *xcp′*
by(*rule exec-meth-drop-xt*)(*auto simp add*: *stack-xlift-compxE2 shift-compxEs2 ac-simps*)
from *IH*[*OF this*] **obtain** *stk″* **where** *stk′*: *stk′* = *stk″* @ *STK*
and *exec″*: *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*)
ta h′ (*stk″*, *loc′*, *pc′* − *length ?pre*, *xcp′*) **by** *blast*
from *exec″* **have** *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e2*)
((*compxE2 e 0 0* @ *compxE2 e1* (*Suc* (*length* (*compE2 e*))) *0*) @ *shift* (*length ?pre*) (*compxE2 e2 0 0*)) *t*
h (*stk*, *loc*, *length ?pre* + *pc*, *xcp*) *ta h′* (*stk″*, *loc′*, *length ?pre* + (*pc′* − *length ?pre*), *xcp′*)
by(*rule append-exec-meth-xt*)(*auto*)
moreover from *exec′* **have** *pc′* ≥ *length ?pre*
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
moreover hence (*Suc* (*Suc* (*pc′* − *Suc* (*Suc 0*)))) = *pc′* **by** *simp*
ultimately show *?case* **using** *stk′*
by(*auto simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps eval-nat-numeral*)
next
case (*bisim1CondThrow e n a xs stk loc pc e1 e2*)
note *bisim* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
note *IH* = ‹⋀*stk′ loc′ pc′ xcp′ STK*. *?exec e stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*
$\Longrightarrow$ *?concl e stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
note *exec* = ‹*?exec* (*if* (*e*) *e1 else e2*) *stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
from *bisim* **have** *pc*: *pc* < *length* (*compE2 e*) **by**(*auto dest*: *bisim1-ThrowD*)
from *exec* **have** *exec-meth-d* (*compP2 P*) (*compE2 e* @ *IfFalse* (2 + *int* (*length* (*compE2 e1*))) # *compE2 e1* @ *Goto* (1 + *int* (*length* (*compE2 e2*))) # *compE2 e2*)
(*stack-xlift* (*length STK*) (*compxE2 e 0 0*) @ *shift* (*length* (*compE2 e*)) (*stack-xlift* (*length STK*) (*compxE2 e1* (*Suc 0*) *0*) @
stack-xlift (*length STK*) (*compxE2 e2* (*Suc* (*Suc* (*length* (*compE2 e1*)))) *0*))) *t*
h (*stk* @ *STK*, *loc*, *pc*, ⌊*a*⌋) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *stack-xlift-compxE2 shift-compxE2 ac-simps*)
hence *?exec e stk STK loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*
by(*rule exec-meth-take-xt*)(*rule pc*)
from *IH*[*OF this*] **show** *?case* **by** *auto*
next
case (*bisim1While1 c n e xs*)
note *IH* = ‹⋀*stk′ loc′ pc′ xcp′ STK*. *?exec c* [] *STK xs 0 None stk′ loc′ pc′ xcp′*
$\Longrightarrow$ *?concl c* [] *STK xs 0 None stk′ loc′ pc′ xcp′*›
note *exec* = ‹*?exec* (*while* (*c*) *e*) [] *STK xs 0 None stk′ loc′ pc′ xcp′*›
hence *exec-meth-d* (*compP2 P*) (*compE2 c* @ *IfFalse* (*int* (*length* (*compE2 e*)) + *3*) # *compE2 e* @ [*Pop*, *Goto* (−2 + (− *int* (*length* (*compE2 e*)) − *int* (*length* (*compE2 c*)))), *Push Unit*])
(*stack-xlift* (*length STK*) (*compxE2 c 0 0*) @ *shift* (*length* (*compE2 c*)) (*stack-xlift* (*length STK*) (*compxE2 e* (*Suc 0*) *0*))) *t*

```
    h ([] @ STK, xs, 0, None) ta h' (stk', loc', pc', xcp')
   by(simp add: compxE2-size-convs)
  hence ?exec c [] STK xs 0 None stk' loc' pc' xcp'
   by(rule exec-meth-take-xt) simp
  from IH[OF this] show ?case by auto
next
  case (bisim1While3 c n c' xs stk loc pc xcp e)
  note bisim = ‹P,c,h ⊢ (c', xs) ↔ (stk, loc, pc, xcp)›
  note IH = ‹⋀stk' loc' pc' xcp' STK. ?exec c stk STK loc pc xcp stk' loc' pc' xcp'
          ⟹ ?concl c stk STK loc pc xcp stk' loc' pc' xcp'›
  note exec = ‹?exec (while (c) e) stk STK loc pc xcp stk' loc' pc' xcp'›
  from bisim have pc: pc ≤ length (compE2 c) by(rule bisim1-pc-length-compE2)
  show ?case
  proof(cases pc < length (compE2 c))
   case True
   from exec have exec-meth-d (compP2 P) (compE2 c @ IfFalse (int (length (compE2 e)) + 3) #
compE2 e @ [Pop, Goto (−2 + (− int (length (compE2 e)) − int (length (compE2 c)))), Push Unit])
    (stack-xlift (length STK) (compxE2 c 0 0) @ shift (length (compE2 c)) (stack-xlift (length STK)
(compxE2 e (Suc 0) 0))) t
     h (stk @ STK, loc, pc, xcp) ta h' (stk', loc', pc', xcp')
     by(simp add: compxE2-size-convs)
   hence ?exec c stk STK loc pc xcp stk' loc' pc' xcp'
    by(rule exec-meth-take-xt)(rule True)
   from IH[OF this] show ?thesis by auto
  next
   case False
   with pc have [simp]: pc = length (compE2 c) by simp
   from bisim obtain v where stk = [v] xcp = None by(auto dest: bisim1-pc-length-compE2D)
   with exec show ?thesis by(auto elim!: exec-meth.cases intro!: exec-meth.intros)
  qed
next
  case (bisim1While4 e n e' xs stk loc pc xcp c)
  note bisim = ‹P,e,h ⊢ (e', xs) ↔ (stk, loc, pc, xcp)›
  note IH = ‹⋀stk' loc' pc' xcp' STK. ?exec e stk STK loc pc xcp stk' loc' pc' xcp'
          ⟹ ?concl e stk STK loc pc xcp stk' loc' pc' xcp'›
  note exec = ‹?exec (while (c) e) stk STK loc (Suc (length (compE2 c) + pc)) xcp stk' loc' pc' xcp'›
  from bisim have pc: pc ≤ length (compE2 e) by(rule bisim1-pc-length-compE2)
  show ?case
  proof(cases pc < length (compE2 e))
   case True
   let ?pre = compE2 c @ [IfFalse (int (length (compE2 e)) + 3)]
   from exec have exec-meth-d (compP2 P) ((?pre @ compE2 e) @ [Pop, Goto (−2 + (− int (length
(compE2 e)) − int (length (compE2 c)))), Push Unit])
                  (stack-xlift (length STK) (compxE2 c 0 0) @ shift (length ?pre) (stack-xlift (length
STK) (compxE2 e 0 0))) t
                  h (stk @ STK, loc, length ?pre + pc, xcp) ta h' (stk', loc', pc', xcp')
     by(simp add: compxE2-size-convs)
   hence exec': exec-meth-d (compP2 P) (?pre @ compE2 e) (stack-xlift (length STK) (compxE2 c 0
0) @ shift (length ?pre) (stack-xlift (length STK) (compxE2 e 0 0))) t
                h (stk @ STK, loc, length ?pre + pc, xcp) ta h' (stk', loc', pc', xcp')
     by(rule exec-meth-take)(auto intro: True)
   hence ?exec e stk STK loc pc xcp stk' loc' (pc' − length ?pre) xcp'
    by(rule exec-meth-drop-xt)(auto simp add: stack-xlift-compxE2)
   from IH[OF this] obtain stk'' where stk': stk' = stk'' @ STK
```

and *exec''*: *exec-meth-d* (*compP2 P*) (*compE2 e*) (*compxE2 e 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *pc' − length ?pre*, *xcp'*) **by** *auto*
from *exec''* **have** *exec-meth-d* (*compP2 P*) (*?pre @ compE2 e*) (*compxE2 c 0 0 @ shift* (*length ?pre*) (*compxE2 e 0 0*)) *t*
h (*stk*, *loc*, *length ?pre + pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *length ?pre +* (*pc' − length ?pre*), *xcp'*)
by(*rule append-exec-meth-xt*) *auto*
hence *exec-meth-d* (*compP2 P*) ((*?pre @ compE2 e*) @ [*Pop*, *Goto* (*−2 +* (*− int* (*length* (*compE2 e*)) *− int* (*length* (*compE2 c*)))), *Push Unit*])
(*compxE2 c 0 0 @ shift* (*length ?pre*) (*compxE2 e 0 0*)) *t*
h (*stk*, *loc*, *length ?pre + pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *length ?pre +* (*pc' − length ?pre*), *xcp'*)
by(*rule exec-meth-append*)
moreover from *exec'* **have** *pc' ≥ length ?pre*
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
moreover have *−2 +* (*− int* (*length* (*compE2 e*)) *− int* (*length* (*compE2 c*))) *= − int* (*length* (*compE2 c*)) *+* (*−2 − int* (*length* (*compE2 e*))) **by** *simp*
ultimately show *?thesis* **using** *stk'*
by(*auto simp add*: *shift-compxE2 stack-xlift-compxE2 algebra-simps uminus-minus-left-commute*)
next
case *False*
with *pc* **have** [*simp*]: *pc = length* (*compE2 e*) **by** *simp*
from *bisim* **obtain** *v* **where** *stk* = [*v*] *xcp = None* **by**(*auto dest*: *bisim1-pc-length-compE2D*)
with *exec* **show** *?thesis* **by**(*auto elim*!: *exec-meth.cases intro*!: *exec-meth.intros*)
qed
next
case (*bisim1While6 c n e xs*)
note *exec* = ‹*?exec* (*while* (*c*) *e*) [] *STK xs* (*Suc* (*Suc* (*length* (*compE2 c*) *+ length* (*compE2 e*)))) *None stk' loc' pc' xcp'*›
thus *?case* **by**(*rule exec-meth.cases*)(*simp-all*, *auto intro*!: *exec-meth.intros*)
next
case (*bisim1While7 c n e xs*)
note *exec* = ‹*?exec* (*while* (*c*) *e*) [] *STK xs* (*Suc* (*Suc* (*Suc* (*length* (*compE2 c*) *+ length* (*compE2 e*))))) *None stk' loc' pc' xcp'*›
thus *?case* **by**(*rule exec-meth.cases*)(*simp-all*, *auto intro*!: *exec-meth.intros*)
next
case (*bisim1WhileThrow1 c n a xs stk loc pc e*)
note *bisim* = ‹*P*,*c*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
note *IH* = ‹⋀*stk' loc' pc' xcp' STK*. *?exec c stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*
⟹ *?concl c stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*›
note *exec* = ‹*?exec* (*while* (*c*) *e*) *stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*›
from *bisim* **have** *pc*: *pc < length* (*compE2 c*) **by**(*auto dest*: *bisim1-ThrowD*)
from *exec* **have** *exec-meth-d* (*compP2 P*) (*compE2 c @ IfFalse* (*int* (*length* (*compE2 e*)) *+ 3*) *# compE2 e @* [*Pop*, *Goto* (*−2 +* (*− int* (*length* (*compE2 e*)) *− int* (*length* (*compE2 c*)))), *Push Unit*])
(*stack-xlift* (*length STK*) (*compxE2 c 0 0*) *@ shift* (*length* (*compE2 c*)) (*stack-xlift* (*length STK*) (*compxE2 e* (*Suc 0*) *0*))) *t*
h (*stk @ STK*, *loc*, *pc*, ⌊*a*⌋) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *compxE2-size-convs*)
hence *?exec c stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*
by(*rule exec-meth-take-xt*)(*rule pc*)
from *IH*[*OF this*] **show** *?case* **by** *auto*
next
case (*bisim1WhileThrow2 e n a xs stk loc pc c*)
note *bisim* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
note *IH* = ‹⋀*stk' loc' pc' xcp' STK*. *?exec e stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*

$\Longrightarrow$ *?concl e stk STK loc pc* $\lfloor a \rfloor$ *stk' loc' pc' xcp'*$\rangle$
note *exec* = ‹*?exec* (*while* (*c*) *e*) *stk STK loc* (*Suc* (*length* (*compE2 c*) + *pc*)) $\lfloor a \rfloor$ *stk' loc' pc' xcp'*›
from *bisim* **have** *pc*: *pc* < *length* (*compE2 e*) **by**(*auto dest*: *bisim1-ThrowD*)
let *?pre* = *compE2 c* @ [*IfFalse* (*int* (*length* (*compE2 e*)) + *3*)]
from *exec* **have** *exec-meth-d* (*compP2 P*) ((*?pre* @ *compE2 e*) @ [*Pop*, *Goto* (−*2* + (− *int* (*length* (*compE2 e*)) − *int* (*length* (*compE2 c*)))), *Push Unit*])
(*stack-xlift* (*length STK*) (*compxE2 c 0 0*) @ *shift* (*length ?pre*) (*stack-xlift* (*length STK*) (*compxE2 e 0 0*))) *t*
h (*stk* @ *STK*, *loc*, *length ?pre* + *pc*, $\lfloor a \rfloor$) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *compxE2-size-convs*)
hence *exec'*: *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e*)
(*stack-xlift* (*length STK*) (*compxE2 c 0 0*) @ *shift* (*length ?pre*) (*stack-xlift* (*length STK*) (*compxE2 e 0 0*))) *t*
h (*stk* @ *STK*, *loc*, (*length ?pre* + *pc*), $\lfloor a \rfloor$) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*rule exec-meth-take*)(*auto intro*: *pc*)
hence *?exec e stk STK loc pc* $\lfloor a \rfloor$ *stk' loc'* (*pc'* − *length ?pre*) *xcp'*
by(*rule exec-meth-drop-xt*)(*auto simp add*: *stack-xlift-compxE2*)
from *IH*[*OF this*] **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ *STK*
and *exec''*: *exec-meth-d* (*compP2 P*) (*compE2 e*) (*compxE2 e 0 0*) *t h* (*stk*, *loc*, *pc*, $\lfloor a \rfloor$) *ta h'* (*stk''*, *loc'*, *pc'* − *length ?pre*, *xcp'*) **by** *auto*
from *exec''* **have** *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e*) (*compxE2 c 0 0* @ *shift* (*length ?pre*) (*compxE2 e 0 0*)) *t*
h (*stk*, *loc*, *length ?pre* + *pc*, $\lfloor a \rfloor$) *ta h'* (*stk''*, *loc'*, *length ?pre* + (*pc'* − *length ?pre*), *xcp'*)
by(*rule append-exec-meth-xt*) *auto*
hence *exec-meth-d* (*compP2 P*) ((*?pre* @ *compE2 e*) @ [*Pop*, *Goto* (−*2* + (− *int* (*length* (*compE2 e*)) − *int* (*length* (*compE2 c*)))), *Push Unit*])
(*compxE2 c 0 0* @ *shift* (*length ?pre*) (*compxE2 e 0 0*)) *t*
h (*stk*, *loc*, *length ?pre* + *pc*, $\lfloor a \rfloor$) *ta h'* (*stk''*, *loc'*, *length ?pre* + (*pc'* − *length ?pre*), *xcp'*)
by(*rule exec-meth-append*)
moreover from *exec'* **have** *pc'* ≥ *length ?pre*
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
moreover have −*2* + (− *int* (*length* (*compE2 e*)) − *int* (*length* (*compE2 c*))) = − *int* (*length* (*compE2 c*)) + (−*2* − *int* (*length* (*compE2 e*))) **by** *simp*
ultimately show *?case* **using** *stk'*
by(*auto simp add*: *shift-compxE2 stack-xlift-compxE2 algebra-simps uminus-minus-left-commute*)
next
case (*bisim1Throw1 e n e' xs stk loc pc xcp*)
note *bisim* = ‹*P*,*e*,*h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *IH* = ‹⋀*stk' loc' pc' xcp' STK*. *?exec e stk STK loc pc xcp stk' loc' pc' xcp'*
$\Longrightarrow$ *?concl e stk STK loc pc xcp stk' loc' pc' xcp'*›
note *exec* = ‹*?exec* (*throw e*) *stk STK loc pc xcp stk' loc' pc' xcp'*›
from *bisim* **have** *pc*: *pc* ≤ *length* (*compE2 e*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e*))
case *True*
with *exec* **have** *?exec e stk STK loc pc xcp stk' loc' pc' xcp'* **by**(*auto elim*: *exec-meth-take*)
from *IH*[*OF this*] **show** *?thesis* **by** *auto*
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compE2 e*) **by** *simp*
from *bisim* **obtain** *v* **where** *stk* = [*v*] *xcp* = *None* **by**(*auto dest*: *bisim1-pc-length-compE2D*)
with *exec* **show** *?thesis* **by**(*auto elim*!: *exec-meth.cases intro*!: *exec-meth.intros split*: *if-split-asm*)
qed
next

```
  case bisim1Throw2 thus ?case
    apply(auto elim!:exec-meth.cases intro: exec-meth.intros dest!: match-ex-table-stack-xliftD)
    apply(auto intro: exec-meth.intros dest!: match-ex-table-stack-xliftD intro!: exI)
    apply(auto simp add: le-Suc-eq)
    done
next
  case bisim1ThrowNull thus ?case
    apply(auto elim!:exec-meth.cases intro: exec-meth.intros dest!: match-ex-table-stack-xliftD)
    apply(auto intro: exec-meth.intros dest!: match-ex-table-stack-xliftD intro!: exI)
    apply(auto simp add: le-Suc-eq)
    done
next
  case (bisim1ThrowThrow e n a xs stk loc pc)
  note bisim = ‹P,e,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
  note IH = ‹⋀stk' loc' pc' xcp' STK. ?exec e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'
          ⟹ ?concl e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'›
  note exec = ‹?exec (throw e) stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'›
  from bisim have pc: pc < length (compE2 e) by(auto dest: bisim1-ThrowD)
  with exec have ?exec e stk STK loc pc ⌊a⌋ stk' loc' pc' xcp'
    by(auto elim: exec-meth-take simp add: compxE2-size-convs)
  from IH[OF this] show ?case by auto
next
  case (bisim1Try e n e' xs stk loc pc xcp e2 C' V)
  note bisim = ‹P,e,h ⊢ (e', xs) ↔ (stk, loc, pc, xcp)›
  note IH = ‹⋀stk' loc' pc' xcp' STK. ?exec e stk STK loc pc xcp stk' loc' pc' xcp'
          ⟹ ?concl e stk STK loc pc xcp stk' loc' pc' xcp'›
  note exec = ‹?exec (try e catch(C' V) e2) stk STK loc pc xcp stk' loc' pc' xcp'›
  from bisim have pc: pc ≤ length (compE2 e) by(rule bisim1-pc-length-compE2)
  show ?case
  proof(cases pc < length (compE2 e))
    case True
    from exec have exec': exec-meth-d (compP2 P) (compE2 e @ Goto (int (length (compE2 e2)) +
2) # Store V # compE2 e2)
      (stack-xlift (length STK) (compxE2 e 0 0) @  shift (length (compE2 e)) (stack-xlift (length STK)
(compxE2 e2 (Suc (Suc 0)) 0)) @ [(0, length (compE2 e), ⌊C'⌋, Suc (length (compE2 e)), length
STK)]) t
      h (stk @ STK, loc, pc, xcp) ta h' (stk', loc', pc', xcp')
      by(simp add: compxE2-size-convs)
    show ?thesis
    proof(cases xcp)
      case None
      with exec' True have ?exec e stk STK loc pc xcp stk' loc' pc' xcp'
        apply −
        apply (erule exec-meth.cases)
        apply (cases compE2 e ! pc)
          apply (fastforce simp add: is-Ref-def intro: exec-meth.intros split: if-split-asm cong del: im-
age-cong-simp)+
        done
      from IH[OF this] show ?thesis by auto
    next
      case (Some a)
      with exec' have [simp]: h' = h loc' = loc xcp' = None ta = ε
        by(auto elim: exec-meth.cases)
      show ?thesis
```

```
    proof(cases match-ex-table (compP2 P) (cname-of h a) pc (compxE2 e 0 0))
      case (Some pcd)
      from exec ‹xcp = ⌊a⌋› Some pc
      have stk': stk' = Addr a # (drop (length stk − snd pcd) stk) @ STK
          by(auto elim!: exec-meth.cases simp add: match-ex-table-append split: if-split-asm dest!:
match-ex-table-stack-xliftD)
      from exec' ‹xcp = ⌊a⌋› Some pc have exec-meth-d (compP2 P)
        (compE2 e) (stack-xlift (length STK) (compxE2 e 0 0)) t h (stk @ STK, loc, pc, ⌊a⌋) ε h
(Addr a # (drop (length (stk @ STK) − (snd pcd + length STK)) (stk @ STK)), loc, pc', None)
       apply −
       apply(rule exec-meth.intros)
        apply(auto elim!: exec-meth.cases simp add: match-ex-table-append split: if-split-asm dest!:
match-ex-table-shift-pcD match-ex-table-stack-xliftD)
       done
      from IH[unfolded ‹ta = ε› ‹xcp = ⌊a⌋› ‹h' = h›, OF this]
      have stk: Addr a # drop (length stk − snd pcd) (stk @ STK) = Addr a # drop (length stk −
snd pcd) stk @ STK
       and exec'': exec-meth-d (compP2 P) (compE2 e) (compxE2 e 0 0) t h (stk, loc, pc, ⌊a⌋) ε h
(Addr a # drop (length stk − snd pcd) stk, loc, pc', None) by auto
      thus ?thesis using Some stk' ‹xcp = ⌊a⌋› by(auto)
     next
      case None
      with Some exec pc have stk': stk' = Addr a # STK
       and pc': pc' = Suc (length (compE2 e))
       and subcls: compP2 P ⊢ cname-of h a ⪯* C'
      by(auto elim!: exec-meth.cases split: if-split-asm simp add: match-ex-table-append-not-pcs)(simp
add: matches-ex-entry-def)
      moreover from Some True None pc' subcls
      have exec-meth-d (compP2 P) (compE2 (try e catch(C' V) e2)) (compxE2 (try e catch(C' V)
e2) 0 0) t h
       (stk, loc, pc, ⌊a⌋) ε h (Addr a # drop (length stk − 0) stk, loc, pc', None)
       by −(rule exec-catch,auto simp add: match-ex-table-append-not-pcs matches-ex-entry-def)
      ultimately show ?thesis using Some by auto
     qed
   qed
  next
   case False
   with pc have [simp]: pc = length (compE2 e) by simp
   from bisim obtain v where stk = [v] xcp = None by(auto dest: bisim1-pc-length-compE2D)
   with exec show ?thesis by(auto elim!: exec-meth.cases intro!: exec-meth.intros split: if-split-asm)
  qed
next
  case bisim1TryCatch1 thus ?case
   by(auto elim!: exec-meth.cases intro!: exec-meth.intros split: if-split-asm)
next
  case (bisim1TryCatch2 e2 n e' xs stk loc pc xcp e C' V)
  note bisim = ‹P,e2,h ⊢ (e', xs) ↔ (stk, loc, pc, xcp)›
  note IH = ‹⋀stk' loc' pc' xcp' STK. ?exec e2 stk STK loc pc xcp stk' loc' pc' xcp'
         ⟹ ?concl e2 stk STK loc pc xcp stk' loc' pc' xcp'›
  note exec = ‹?exec (try e catch(C' V) e2) stk STK loc (Suc (Suc (length (compE2 e) + pc))) xcp
stk' loc' pc' xcp'›
  let ?pre = compE2 e @ [Goto (int (length (compE2 e2)) + 2), Store V]
  from exec have exec': exec-meth-d (compP2 P) (?pre @ compE2 e2)
   (stack-xlift (length STK) (compxE2 e 0 0) @ shift (length ?pre) (stack-xlift (length STK) (compxE2
```

e2 0 0))) *t*
h (*stk* @ *STK*, *loc*, *length ?pre* + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
proof(*cases*)
case (*exec-catch xcp'' d*)
let *?stk* = *stk* @ *STK* **and** *?PC* = *Suc* (*Suc* (*length* (*compE2 e*) + *pc*))
note *s* = ‹*stk'* = *Addr xcp''* # *drop* (*length ?stk* − *d*) *?stk*›
‹*ta* = ε› ‹*h'* = *h*› ‹*xcp'* = *None*› ‹*xcp* = ⌊*xcp''*⌋› ‹*loc'* = *loc*›
from ‹*match-ex-table* (*compP2 P*) (*cname-of h xcp''*) *?PC* (*stack-xlift* (*length STK*) (*compxE2* (*try e catch*(*C' V*) *e2*) *0 0*)) = ⌊(*pc'*, *d*)⌋› ‹*d* ≤ *length ?stk*›
show *?thesis* **unfolding** *s*
by −(*rule exec-meth.exec-catch*, *simp-all add*: *shift-compxE2 stack-xlift-compxE2*, *simp add*: *match-ex-table-append add*: *matches-ex-entry-def*)
qed(*auto intro*: *exec-meth.intros simp add*: *shift-compxE2 stack-xlift-compxE2*)
hence *?exec e2 stk STK loc pc xcp stk' loc'* (*pc'* − *length ?pre*) *xcp'*
by(*rule exec-meth-drop-xt*)(*auto simp add*: *stack-xlift-compxE2*)
from *IH*[*OF this*] **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ *STK*
and *exec''*: *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*compxE2 e2 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *pc'* − *length ?pre*, *xcp'*) **by** *auto*
from *exec''* **have** *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e2*) (*compxE2 e 0 0* @ *shift* (*length ?pre*) (*compxE2 e2 0 0*)) *t h* (*stk*, *loc*, *length ?pre* + *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *length ?pre* + (*pc'* − *length ?pre*), *xcp'*)
by(*rule append-exec-meth-xt*) *auto*
hence *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e2*) (*compxE2 e 0 0* @ *shift* (*length ?pre*) (*compxE2 e2 0 0*) @ [(*0*, *length* (*compE2 e*), ⌊*C'*⌋, *Suc* (*length* (*compE2 e*)), *0*)]) *t h* (*stk*, *loc*, *length ?pre* + *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *length ?pre* + (*pc'* − *length ?pre*), *xcp'*)
by(*rule exec-meth.cases*)(*auto intro*: *exec-meth.intros simp add*: *match-ex-table-append-not-pcs*)
moreover from *exec'* **have** *pc'* ≥ *length ?pre*
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
moreover hence (*Suc* (*Suc* (*pc'* − *Suc* (*Suc 0*)))) = *pc'* **by** *simp*
ultimately show *?case* **using** *stk'* **by**(*auto simp add*: *shift-compxE2 eval-nat-numeral*)
next
case (*bisim1TryFail e n a xs stk loc pc C' C'' e2 V*)
note *bisim* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
note *exec* = ‹*?exec* (*try e catch*(*C'' V*) *e2*) *stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*›
note *a* = ‹*typeof-addr h a* = ⌊*Class-type C'*⌋› ‹¬ *P* ⊢ *C'* ⪯* *C''*›
from *bisim* **have** *match-ex-table* (*compP2 P*) (*cname-of h a*) (*0* + *pc*) (*compxE2 e 0 0*) = *None*
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)
moreover from *bisim* **have** *pc* < *length* (*compE2 e*) **by**(*auto dest*: *bisim1-ThrowD*)
ultimately have *False* **using** *exec a*
apply(*auto elim*!: *exec-meth.cases simp add*: *outside-pcs-compxE2-not-matches-entry outside-pcs-not-matches-entry split*: *if-split-asm*)
apply(*auto simp add*: *compP2-def match-ex-entry match-ex-table-append-not-pcs cname-of-def split*: *if-split-asm*)
done
thus *?case* **..**
next
case (*bisim1TryCatchThrow e2 n a xs stk loc pc e C' V*)
note *bisim* = ‹*P*,*e2*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
note *IH* = ‹⋀*stk' loc' pc' xcp' STK*. *?exec e2 stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*
⟹ *?concl e2 stk STK loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*›
note *exec* = ‹*?exec* (*try e catch*(*C' V*) *e2*) *stk STK loc* (*Suc* (*Suc* (*length* (*compE2 e*) + *pc*))) ⌊*a*⌋ *stk' loc' pc' xcp'*›
from *bisim* **have** *pc*: *pc* < *length* (*compE2 e2*) **by**(*auto dest*: *bisim1-ThrowD*)
let *?pre* = *compE2 e* @ [*Goto* (*int* (*length* (*compE2 e2*)) + *2*), *Store V*]

```
  from exec have exec': exec-meth-d (compP2 P) (?pre @ compE2 e2) (stack-xlift (length STK)
(compxE2 e 0 0) @
    shift (length ?pre) (stack-xlift (length STK) (compxE2 e2 0 0))) t
    h (stk @ STK, loc, length ?pre + pc, ⌊a⌋) ta h' (stk', loc', pc', xcp')
  proof(cases)
    case (exec-catch d)
    let ?stk = stk @ STK and ?PC = Suc (Suc (length (compE2 e) + pc))
    note s = ‹stk' = Addr a # drop (length ?stk − d) ?stk› ‹loc' = loc›
      ‹ta = ε› ‹h' = h› ‹xcp' = None›
    from ‹match-ex-table (compP2 P) (cname-of h a) ?PC (stack-xlift (length STK) (compxE2 (try e
catch(C' V) e2) 0 0)) = ⌊(pc', d)⌋› ‹d ≤ length ?stk›
    show ?thesis unfolding s
      by −(rule exec-meth.exec-catch, simp-all add: shift-compxE2 stack-xlift-compxE2, simp add:
match-ex-table-append add: matches-ex-entry-def)
  qed
  hence ?exec e2 stk STK loc pc ⌊a⌋ stk' loc' (pc' − length ?pre) xcp'
    by(rule exec-meth-drop-xt)(auto simp add: stack-xlift-compxE2)
  from IH[OF this] obtain stk'' where stk': stk' = stk'' @ STK
    and exec'': exec-meth-d (compP2 P) (compE2 e2) (compxE2 e2 0 0) t h (stk, loc, pc, ⌊a⌋) ta h'
(stk'', loc', pc' − length ?pre, xcp') by auto
  from exec'' have exec-meth-d (compP2 P) (?pre @ compE2 e2) (compxE2 e 0 0 @ shift (length
?pre) (compxE2 e2 0 0)) t h (stk, loc, length ?pre + pc, ⌊a⌋) ta h' (stk'', loc', length ?pre + (pc' −
length ?pre), xcp')
    by(rule append-exec-meth-xt) auto
  hence exec-meth-d (compP2 P) (?pre @ compE2 e2) (compxE2 e 0 0 @ shift (length ?pre) (compxE2
e2 0 0) @ [(0, length (compE2 e), ⌊C'⌋, Suc (length (compE2 e)), 0)]) t h (stk, loc, length ?pre + pc,
⌊a⌋) ta h' (stk'', loc', length ?pre + (pc' − length ?pre), xcp')
    by(rule exec-meth.cases)(auto intro!: exec-meth.intros simp add: match-ex-table-append-not-pcs)
  moreover from exec' have pc' ≥ length ?pre
    by(rule exec-meth-drop-xt-pc)(auto simp add: stack-xlift-compxE2)
  moreover hence (Suc (Suc (pc' − Suc (Suc 0)))) = pc' by simp
  ultimately show ?case using stk' by(auto simp add: shift-compxE2 eval-nat-numeral)
next
  case bisims1Nil thus ?case by(auto elim: exec-meth.cases)
next
  case (bisims1List1 e n e' xs stk loc pc xcp es)
  note bisim1 = ‹P,e,h ⊢ (e', xs ) ↔ (stk, loc, pc, xcp)›
  note IH1 = ‹⋀stk' loc' pc' xcp' STK. ?exec e stk STK loc pc xcp stk' loc' pc' xcp'
          ⟹ ?concl e stk STK loc pc xcp stk' loc' pc' xcp'›
  note IH2 = ‹⋀xs stk' loc' pc' xcp' STK. ?execs es [] STK xs 0 None stk' loc' pc' xcp'
          ⟹ ?concls es [] STK xs 0 None stk' loc' pc' xcp'›
  note exec = ‹?execs (e # es) stk STK loc pc xcp stk' loc' pc' xcp'›
  from bisim1 have pc: pc ≤ length (compE2 e) by(rule bisim1-pc-length-compE2)
  show ?case
  proof(cases pc < length (compE2 e))
    case True
    with exec have ?exec e stk STK loc pc xcp stk' loc' pc' xcp'
      by(simp add: compxEs2-size-convs)(erule exec-meth-take-xt)
    from IH1[OF this] show ?thesis by auto
  next
    case False
    with pc have pc: pc = length (compE2 e) by simp
    with bisim1 obtain v where s: stk = [v] xcp = None by(auto dest: bisim1-pc-length-compE2D)
    with exec pc have exec': exec-meth-d (compP2 P) (compE2 e @ compEs2 es)
```

(*stack-xlift* (*length STK*) (*compxE2 e 0 0*) @ *shift* (*length* (*compE2 e*)) (*stack-xlift* (*length* (*v* # *STK*)) (*compxEs2 es 0 0*))) *t*
h ([] @ *v* # *STK*, *loc*, *length* (*compE2 e*) + *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *compxEs2-size-convs compxEs2-stack-xlift-convs*)
hence *?execs es* [] (*v* # *STK*) *loc 0 None stk' loc'* (*pc'* − *length* (*compE2 e*)) *xcp'*
by(*rule exec-meth-drop-xt*)(*auto simp add*: *stack-xlift-compxE2*)
from *IH2*[*OF this*] **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ *v* # *STK*
and *exec''*: *exec-meth-d* (*compP2 P*) (*compEs2 es*) (*compxEs2 es 0 0*) *t h* ([], *loc*, *0*, *None*) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 e*), *xcp'*) **by** *auto*
from *exec''* **have** *exec-meth-d* (*compP2 P*) (*compEs2 es*) (*stack-xlift* (*length* [*v*]) (*compxEs2 es 0 0*)) *t h* ([] @ [*v*], *loc*, *0*, *None*) *ta h'* (*stk''* @ [*v*], *loc'*, *pc'* − *length* (*compE2 e*), *xcp'*)
by(*rule exec-meth-stk-offer*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e* @ *compEs2 es*) (*compxE2 e 0 0* @ *shift* (*length* (*compE2 e*)) (*stack-xlift* (*length* [*v*]) (*compxEs2 es 0 0*))) *t h* ([] @ [*v*], *loc*, *length* (*compE2 e*) + *0*, *None*) *ta h'* (*stk''* @ [*v*], *loc'*, *length* (*compE2 e*) + (*pc'* − *length* (*compE2 e*)), *xcp'*)
by(*rule append-exec-meth-xt*) *auto*
moreover from *exec'* **have** *pc'* ≥ *length* (*compE2 e*)
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
ultimately show *?thesis* **using** *s pc stk'* **by**(*auto simp add*: *shift-compxEs2 stack-xlift-compxEs2*)
qed
next
case (*bisims1List2 es n es' xs stk loc pc xcp e v*)
note *bisim* = ‹*P*,*es*,*h* ⊢ (*es'*,*xs*) [↔] (*stk*,*loc*,*pc*,*xcp*)›
note *IH* = ‹⋀*stk' loc' pc' xcp' STK*. *?execs es stk STK loc pc xcp stk' loc' pc' xcp'* ⟹ *?concls es stk STK loc pc xcp stk' loc' pc' xcp'›
note *exec* = ‹*?execs* (*e* # *es*) (*stk* @ [*v*]) *STK loc* (*length* (*compE2 e*) + *pc*) *xcp stk' loc' pc' xcp'›
from *exec* **have** *exec'*: *exec-meth-d* (*compP2 P*) (*compE2 e* @ *compEs2 es*)
(*stack-xlift* (*length STK*) (*compxE2 e 0 0*) @ *shift* (*length* (*compE2 e*)) (*stack-xlift* (*length* (*v* # *STK*)) (*compxEs2 es 0 0*))) *t*
h (*stk* @ *v* # *STK*, *loc*, *length* (*compE2 e*) + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *compxEs2-size-convs compxEs2-stack-xlift-convs*)
hence *?execs es stk* (*v* # *STK*) *loc pc xcp stk' loc'* (*pc'* − *length* (*compE2 e*)) *xcp'*
by(*rule exec-meth-drop-xt*)(*auto simp add*: *stack-xlift-compxE2*)
from *IH*[*OF this*] **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ *v* # *STK*
and *exec''*: *exec-meth-d* (*compP2 P*) (*compEs2 es*) (*compxEs2 es 0 0*) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 e*), *xcp'*) **by** *auto*
from *exec''* **have** *exec-meth-d* (*compP2 P*) (*compEs2 es*) (*stack-xlift* (*length* [*v*]) (*compxEs2 es 0 0*)) *t h* (*stk* @ [*v*], *loc*, *pc*, *xcp*) *ta h'* (*stk''* @ [*v*], *loc'*, *pc'* − *length* (*compE2 e*), *xcp'*)
by(*rule exec-meth-stk-offer*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e* @ *compEs2 es*) (*compxE2 e 0 0* @ *shift* (*length* (*compE2 e*)) (*stack-xlift* (*length* [*v*]) (*compxEs2 es 0 0*))) *t h* (*stk* @ [*v*], *loc*, *length* (*compE2 e*) + *pc*, *xcp*) *ta h'* (*stk''* @ [*v*], *loc'*, *length* (*compE2 e*) + (*pc'* − *length* (*compE2 e*)), *xcp'*)
by(*rule append-exec-meth-xt*) *auto*
moreover from *exec'* **have** *pc'* ≥ *length* (*compE2 e*)
by(*rule exec-meth-drop-xt-pc*)(*auto simp add*: *stack-xlift-compxE2*)
ultimately show *?case* **using** *stk'* **by**(*auto simp add*: *shift-compxEs2 stack-xlift-compxEs2*)
next
case (*bisim1Sync12 e1 n e2 V a xs v v'*)
note *exec* = ‹*?exec* (*sync*$_V$ (*e1*) *e2*) [*v*, *v'*] *STK xs* (*4* + *length* (*compE2 e1*) + *length* (*compE2 e2*)) ⌊*a*⌋ *stk' loc' pc' xcp'›*
thus *?case* **by**(*auto elim*!: *exec-meth.cases split*: *if-split-asm simp add*: *match-ex-table-append-not-pcs*)(*simp add*: *matches-ex-entry-def*)
next
case (*bisim1Sync14 e1 n e2 V a xs v a'*)

note *exec* = ‹*?exec* (*sync*$_V$ (*e1*) *e2*) [*v*, *Addr a'*] *STK xs* (7 + *length* (*compE2 e1*) + *length* (*compE2 e2*)) ⌊*a*⌋ *stk' loc' pc' xcp'*›
thus *?case* **by**(*auto elim*!: *exec-meth.cases split*: *if-split-asm simp add*: *match-ex-table-append-not-pcs*)(*simp add*: *matches-ex-entry-def*)
qed

lemma shows *bisim1-callD*:
⟦ *P*,*e*,*h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*); *call1 e'* = ⌊(*a*, *M*, *vs*)⌋;
compE2 e ! *pc* = *Invoke M' n0* ⟧
⟹ *M* = *M'*

and *bisims1-callD*:
⟦ *P*,*es*,*h* ⊢ (*es'*,*xs*) [↔] (*stk*,*loc*,*pc*, *xcp*); *calls1 es'* = ⌊(*a*, *M*, *vs*)⌋;
compEs2 es ! *pc* = *Invoke M' n0* ⟧
⟹ *M* = *M'*
proof(*induct e n* :: *nat e' xs stk loc pc xcp* **and** *es n* :: *nat es' xs stk loc pc xcp*
rule: *bisim1-bisims1-inducts-split*)
case *bisim1AAss1* **thus** *?case*
apply(*simp* (*no-asm-use*) *split*: *if-split-asm add*: *is-val-iff*)
apply(*fastforce dest*: *bisim-Val-pc-not-Invoke*)
apply(*fastforce dest*: *bisim-Val-pc-not-Invoke*)
apply(*fastforce dest*: *bisim-Val-pc-not-Invoke bisim1-pc-length-compE2*)+
done
next
case *bisim1Call1* **thus** *?case*
apply(*clarsimp split*: *if-split-asm simp add*: *is-vals-conv*)
apply(*drule bisim-Val-pc-not-Invoke*, *simp*, *fastforce*)
apply(*drule bisim-Val-pc-not-Invoke*, *simp*, *fastforce*)
apply(*drule bisim1-pc-length-compE2*, *clarsimp simp add*: *neq-Nil-conv*)
apply(*drule bisim1-pc-length-compE2*, *simp*)
apply(*drule bisim1-pc-length-compE2*, *simp*)
apply(*drule bisim1-pc-length-compE2*, *simp*)
apply(*drule bisim1-call-pcD*, *simp*, *simp*)
apply(*drule bisim1-call-pcD*, *simp*, *simp*)
done
next
case *bisim1CallParams* **thus** *?case*
apply(*clarsimp split*: *if-split-asm simp add*: *is-vals-conv*)
apply(*drule bisims-Val-pc-not-Invoke*, *simp*, *fastforce*)
apply(*drule bisims1-pc-length-compEs2*, *simp*)
apply(*drule bisims1-calls-pcD*, *simp*, *simp*)
done
qed(*fastforce split*: *if-split-asm dest*: *bisim1-pc-length-compE2 bisims1-pc-length-compEs2 bisim1-call-pcD bisims1-calls-pcD bisim1-call-xcpNone bisims1-calls-xcpNone bisim-Val-pc-not-Invoke bisims-Val-pc-not-Invoke*)+

lemma *bisim1-xcpD*: *P*,*e*,*h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋) ⟹ *pc* < *length* (*compE2 e*)
and *bisims1-xcpD*: *P*,*es*,*h* ⊢ (*es'*, *xs*) [↔] (*stk*, *loc*, *pc*, ⌊*a*⌋) ⟹ *pc* < *length* (*compEs2 es*)
by(*induct* (*e'*, *xs*) (*stk*, *loc*, *pc*, ⌊*a* :: *'addr*⌋) **and** (*es'*, *xs*) (*stk*, *loc*, *pc*, ⌊*a* :: *'addr*⌋)
arbitrary: *e' xs stk loc pc* **and** *es' xs stk loc pc rule*: *bisim1-bisims1.inducts*)
simp-all

lemma *bisim1-match-Some-stk-length*:
⟦ *P*,*E*,*h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋);
match-ex-table (*compP2 P*) (*cname-of h a*) *pc* (*compxE2 E 0 0*) = ⌊(*pc'*, *d*)⌋ ⟧

```
  ⟹ d ≤ length stk

  and bisims1-match-Some-stk-length:
  ⟦ P,Es,h ⊢ (es, xs) [↔] (stk, loc, pc, ⌊a⌋);
     match-ex-table (compP2 P) (cname-of h a) pc (compxEs2 Es 0 0) = ⌊(pc', d)⌋ ⟧
  ⟹ d ≤ length stk
proof(induct (e, xs) (stk, loc, pc, ⌊a :: 'addr⌋) and (es, xs) (stk, loc, pc, ⌊a :: 'addr⌋)
  arbitrary: pc' d e xs stk loc pc and pc' d es xs stk loc pc rule: bisim1-bisims1.inducts)
  case bisim1Call1 thus ?case
    by(fastforce dest: bisim1-xcpD simp add: match-ex-table-append match-ex-table-not-pcs-None)
next
  case bisim1CallThrowObj thus ?case
    by(fastforce dest: bisim1-xcpD simp add: match-ex-table-append match-ex-table-not-pcs-None)
next
  case bisim1Sync4 thus ?case
   apply(clarsimp simp add: match-ex-table-not-pcs-None match-ex-table-append matches-ex-entry-def
split: if-split-asm)
    apply(fastforce simp add: match-ex-table-compxE2-shift-conv dest: bisim1-xcpD)
    done
next
  case bisim1Try thus ?case
     by(fastforce simp add: match-ex-table-append matches-ex-entry-def match-ex-table-not-pcs-None
dest: bisim1-xcpD split: if-split-asm)
next
  case bisim1TryCatch2 thus ?case
   apply(clarsimp simp add: match-ex-table-not-pcs-None match-ex-table-append matches-ex-entry-def
split: if-split-asm)
    apply(fastforce simp add: match-ex-table-compxE2-shift-conv dest: bisim1-xcpD)
    done
next
  case bisim1TryFail thus ?case
     by(fastforce simp add: match-ex-table-append matches-ex-entry-def match-ex-table-not-pcs-None
dest: bisim1-xcpD split: if-split-asm)
next
  case bisim1TryCatchThrow thus ?case
   apply(clarsimp simp add: match-ex-table-not-pcs-None match-ex-table-append matches-ex-entry-def
split: if-split-asm)
    apply(fastforce simp add: match-ex-table-compxE2-shift-conv dest: bisim1-xcpD)
    done
next
  case bisims1List1 thus ?case
   by(fastforce simp add: match-ex-table-append split: if-split-asm dest: bisim1-xcpD match-ex-table-pcsD)
qed(fastforce simp add: match-ex-table-not-pcs-None match-ex-table-append match-ex-table-compxE2-shift-conv
match-ex-table-compxEs2-shift-conv match-ex-table-compxE2-stack-conv match-ex-table-compxEs2-stack-conv
matches-ex-entry-def dest: bisim1-xcpD)+

end

locale J1-JVM-heap-conf-base =
  J1-JVM-heap-base
    addr2thread-id thread-id2addr
    spurious-wakeups
    empty-heap allocate typeof-addr heap-read heap-write
  +
```

J1-heap-conf-base
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
hconf P
+
JVM-heap-conf-base
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
hconf compP2 P
for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr J1-prog*
begin

inductive *bisim1-list1* ::
'thread-id ⇒ *'heap* ⇒ *'addr expr1* × *'addr locals1* ⇒ (*'addr expr1* × *'addr locals1*) *list*
⇒ *'addr option* ⇒ *'addr frame list* ⇒ *bool*
for *t* :: *'thread-id* **and** *h* :: *'heap*
where
bl1-Normal:
⟦ *compTP P* ⊢ *t*:(*xcp*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs*) √;
P ⊢ *C sees M* : *Ts*→*T* = ⌊*body*⌋ *in D*;
P,*blocks1 0* (*Class D*#*Ts*) *body*, *h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*); *max-vars e* ≤ *length xs*;
list-all2 (*bisim1-fr P h*) *exs frs* ⟧
⟹ *bisim1-list1 t h* (*e*, *xs*) *exs xcp* ((*stk*, *loc*, *C*, *M*, *pc*) # *frs*)

| *bl1-finalVal*:
⟦ *hconf h*; *preallocated h* ⟧ ⟹ *bisim1-list1 t h* (*Val v*, *xs*) [] *None* []

| *bl1-finalThrow*:
⟦ *hconf h*; *preallocated h* ⟧ ⟹ *bisim1-list1 t h* (*Throw a*, *xs*) [] ⌊*a*⌋ []

fun *wbisim1* ::
'thread-id
⇒ ((((*'addr expr1* × *'addr locals1*) × (*'addr expr1* × *'addr locals1*) *list*) × *'heap*,
(*'addr option* × *'addr frame list*) × *'heap*) *bisim*
where *wbisim1 t* ((*ex*, *exs*), *h*) ((*xcp*, *frs*), *h'*) ⟷ *h* = *h'* ∧ *bisim1-list1 t h ex exs xcp frs*

lemma *new-thread-conf-compTP*:
assumes *hconf*: *hconf h preallocated h*
and *ha*: *typeof-addr h a* = ⌊*Class-type C*⌋
and *sub*: *typeof-addr h* (*thread-id2addr t*) = ⌊*Class-type C'*⌋ *P* ⊢ *C'* ≼* *Thread*
and *sees*: *P* ⊢ *C sees M*: []→*T* = ⌊*meth*⌋ *in D*
shows *compTP P* ⊢ *t*:(*None*, *h*, [([], *Addr a* # *replicate* (*max-vars meth*) *undefined-value*, *D*, *M*, *0*)]) √

proof −
from *ha sees-method-decl-above*[*OF sees*]
have *P*,*h* ⊢ *Addr a* :≤ *Class D* **by**(*simp add*: *conf-def*)
moreover
hence *compP2 P*,*h* ⊢ *Addr a* :≤ *Class D* **by**(*simp add*: *compP2-def*)
hence *compP2 P*,*h* ⊢ *Addr a* # *replicate* (*max-vars meth*) *undefined-value* [:≤⊤] *map* (λ*i*. *if i* = *0 then OK* ([*Class D*] ! *i*) *else Err*) [*0*..<*max-vars meth*] @ [*Err*]
by −(*rule list-all2-all-nthI*, *simp-all*)
hence *conf-f* (*compP2 P*) *h* ([], *map* (λ*i*. *if i* = *0 then OK* ([*Class D*] ! *i*) *else Err*) [*0*..<*max-vars meth*] @ [*Err*])
(*compE2 meth* @ [*Return*]) ([], *Addr a* # *replicate* (*max-vars meth*) *undefined-value*, *D*, *M*, *0*)
unfolding *conf-f-def2* **by**(*simp add*: *compP2-def*)
ultimately have *conf-f* (*compP2 P*) *h* ([], *TC0.ty*$_l$ (*Suc* (*max-vars meth*)) [*Class D*] {*0*}) (*compE2 meth* @ [*Return*])
([], *Addr a* # *replicate* (*max-vars meth*) *undefined-value*, *D*, *M*, *0*)
by(*simp add*: *TC0.ty*$_l$*-def conf-f-def2 compP2-def*)
with *hconf ha sub sees-method-compP*[*OF sees*, **where** *f*=λ*C M Ts T*. *compMb2*] *sees-method-idemp*[*OF sees*]
show *?thesis*
by(*auto simp add*: *TC0.ty*$_i$*'-def correct-state-def compTP-def tconf-def*)(*fastforce simp add*: *compP2-def compMb2-def tconf-def intro*: *sees-method-idemp*)+
qed

lemma *ta-bisim12-extTA2J1-extTA2JVM*:
assumes *nt*: ⋀*n T C M a h*. ⟦ *n* < *length* {|*ta*|}$_t$; {|*ta*|}$_t$! *n* = *NewThread T* (*C*, *M*, *a*) *h* ⟧
⟹ *typeof-addr h a* = ⌊*Class-type C*⌋ ∧ (∃ *C'*. *typeof-addr h* (*thread-id2addr T*) = ⌊*Class-type C'*⌋ ∧ *P* ⊢ *C'* ≼* *Thread*) ∧
(∃ *T meth D*. *P* ⊢ *C sees M*:[]→*T* =⌊*meth*⌋ *in D*) ∧ *hconf h* ∧ *preallocated h*
shows *ta-bisim wbisim1* (*extTA2J1 P ta*) (*extTA2JVM* (*compP2 P*) *ta*)
proof −
{ **fix** *n t C M a m*
assume *n* < *length* {|*ta*|}$_t$ **and** {|*ta*|}$_t$! *n* = *NewThread t* (*C*, *M*, *a*) *m*
from *nt*[*OF this*] **obtain** *T meth D C'*
where *ma*: *typeof-addr m a* = ⌊*Class-type C*⌋
and *sees*: *P* ⊢ *C sees M*: []→*T* = ⌊*meth*⌋ *in D*
and *sub*: *typeof-addr m* (*thread-id2addr t*) = ⌊*Class-type C'*⌋ *P* ⊢ *C'* ≼* *Thread*
and *mconf*: *hconf m preallocated m* **by** *fastforce*
from *sees-method-compP*[*OF sees*, **where** *f*=λ*C M Ts T*. *compMb2*]
have *sees'*: *compP2 P* ⊢ *C sees M*: []→*T* = ⌊(*max-stack meth*, *max-vars meth*, *compE2 meth* @ [*Return*], *compxE2 meth 0 0*)⌋ *in D*
by(*simp add*: *compMb2-def compP2-def*)
have *bisim1-list1 t m* ({*0*:*Class D*=*None*; *meth*}, *Addr a* # *replicate* (*max-vars meth*) *undefined-value*) ([]) *None* [([], *Addr a* # *replicate* (*max-vars meth*) *undefined-value*, *D*, *M*, *0*)]
proof
from *mconf ma sub sees*
show *compTP P* ⊢ *t*:(*None*, *m*, [([], *Addr a* # *replicate* (*max-vars meth*) *undefined-value*, *D*, *M*, *0*)]) √
by(*rule new-thread-conf-compTP*)

from *sees* **show** *P* ⊢ *D sees M*: []→*T* = ⌊*meth*⌋ *in D* **by**(*rule sees-method-idemp*)
show *list-all2* (*bisim1-fr P m*) [] [] **by** *simp*
show *P*,*blocks1 0* [*Class D*] *meth*,*m* ⊢ ({*0*:*Class D*=*None*; *meth*}, *Addr a* # *replicate* (*max-vars meth*) *undefined-value*) ↔

([], *Addr a # replicate* (*max-vars meth*) *undefined-value*, *0*, *None*)
by *simp*(*rule bisim1-refl*)
qed *simp*
with *sees sees'* **have** *bisim1-list1 t m* ({*0*:*Class* (*fst* (*method P C M*))=*None*; *the* (*snd* (*snd* (*snd* (*method P C M*))))}, *Addr a # replicate* (*max-vars* (*the* (*snd* (*snd* (*snd* (*method P C M*)))))) *undefined-value*) [] *None* [([], *Addr a # replicate* (*fst* (*snd* (*the* (*snd* (*snd* (*snd* (*method* (*compP2 P*) *C M*)))))))) *undefined-value*, *fst* (*method* (*compP2 P*) *C M*), *M*, *0*)] **by** *simp* **}**
thus *?thesis*
apply(*auto simp add*: *ta-bisim-def intro*!: *list-all2-all-nthI*)
apply(*case-tac* $\{\!|ta|\!\}_t$! *n*, *auto simp add*: *extNTA2JVM-def*)
done
qed

end

definition *no-call2* :: *'addr expr1* ⇒ *pc* ⇒ *bool*
where *no-call2 e pc* ⟷ (*pc* ≤ *length* (*compE2 e*)) ∧ (*pc* < *length* (*compE2 e*) ⟶ (∀ *M n*. *compE2 e* ! *pc* ≠ *Invoke M n*))

definition *no-calls2* :: *'addr expr1 list* ⇒ *pc* ⇒ *bool*
where *no-calls2 es pc* ⟷ (*pc* ≤ *length* (*compEs2 es*)) ∧ (*pc* < *length* (*compEs2 es*) ⟶ (∀ *M n*. *compEs2 es* ! *pc* ≠ *Invoke M n*))

locale *J1-JVM-conf-read* =
J1-JVM-heap-conf-base
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
hconf P
+
JVM-conf-read
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
hconf compP2 P
for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr J1-prog*

locale *J1-JVM-heap-conf* =
J1-JVM-heap-conf-base
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
hconf P
+
JVM-heap-conf

```
    addr2thread-id thread-id2addr
    spurious-wakeups
    empty-heap allocate typeof-addr heap-read heap-write
    hconf compP2 P
  for addr2thread-id :: ('addr :: addr) ⇒ 'thread-id
  and thread-id2addr :: 'thread-id ⇒ 'addr
  and spurious-wakeups :: bool
  and empty-heap :: 'heap
  and allocate :: 'heap ⇒ htype ⇒ ('heap × 'addr) set
  and typeof-addr :: 'heap ⇒ 'addr ⇀ htype
  and heap-read :: 'heap ⇒ 'addr ⇒ addr-loc ⇒ 'addr val ⇒ bool
  and heap-write :: 'heap ⇒ 'addr ⇒ addr-loc ⇒ 'addr val ⇒ 'heap ⇒ bool
  and hconf :: 'heap ⇒ bool
  and P :: 'addr J1-prog
begin

lemma red-external-ta-bisim21:
  ⟦ wf-prog wf-md P; P,t ⊢ ⟨a•M(vs), h⟩ −ta→ext ⟨va, h'⟩; hconf h'; preallocated h' ⟧
  ⟹ ta-bisim wbisim1 (extTA2J1 P ta) (extTA2JVM (compP2 P) ta)
apply(rule ta-bisim12-extTA2J1-extTA2JVM)
apply(frule (1) red-external-new-thread-sees)
 apply(fastforce simp add: in-set-conv-nth)
apply(frule red-ext-new-thread-heap)
 apply(fastforce simp add: in-set-conv-nth)
apply(frule red-external-new-thread-exists-thread-object[unfolded compP2-def, simplified])
 apply(fastforce simp add: in-set-conv-nth)
apply simp
done

lemma ta-bisim-red-extTA2J1-extTA2JVM:
  assumes wf: wf-prog wf-md P
  and red: uf,P,t' ⊢1 ⟨e, s⟩ −ta→ ⟨e', s'⟩
  and hconf: hconf (hp s') preallocated (hp s')
  shows ta-bisim wbisim1 (extTA2J1 P ta) (extTA2JVM (compP2 P) ta)
proof −
  { fix n t C M a H
    assume len: n < length {|ta|}t and tan: {|ta|}t ! n = NewThread t (C, M, a) H
    hence nt: NewThread t (C, M, a) H ∈ set {|ta|}t unfolding set-conv-nth by(auto intro!: exI)
    from red1-new-threadD[OF red nt] obtain ad M' vs va T C' Ts' Tr' D'
      where rede: P,t' ⊢ ⟨ad•M'(vs),hp s⟩ −ta→ext ⟨va,hp s'⟩
      and ad: typeof-addr (hp s) ad = ⌊T⌋ by blast
    from red-ext-new-thread-heap[OF rede nt] have [simp]: hp s' = H by simp
    from red-external-new-thread-sees[OF wf rede nt]
    obtain T body D where Ha: typeof-addr H a = ⌊Class-type C⌋
      and sees: P ⊢ C sees M:[]→T=⌊body⌋ in D by auto
   have sees': compP2 P ⊢ C sees M:[]→T=⌊(max-stack body, max-vars body, compE2 body @ [Return],
compxE2 body 0 0)⌋ in D
      using sees unfolding compP2-def compMb2-def Let-def by(auto dest: sees-method-compP)
     from red-external-new-thread-exists-thread-object[unfolded compP2-def, simplified, OF rede nt]
hconf Ha sees
    have compTP P ⊢ t:(None, H, [([], Addr a # replicate (max-vars body) undefined-value, D, M,
0)]) √
     by(auto intro: new-thread-conf-compTP)
     hence bisim1-list1 t H ({0:Class D=None; body}, Addr a # replicate (max-vars body) unde-
```

fined-value) [] *None* [([], *Addr a # replicate* (*max-vars body*) *undefined-value*, *D*, *M*, *0*)]
proof
from *sees* **show** *P* ⊢ *D sees M*:[]→*T*=⌊*body*⌋ *in D* **by**(*rule sees-method-idemp*)

show *P*,*blocks1 0* [*Class D*] *body*,*H* ⊢ ({*0*:*Class D*=*None*; *body*}, *Addr a # replicate* (*max-vars body*) *undefined-value*) ↔
([], *Addr a # replicate* (*max-vars body*) *undefined-value*, *0*, *None*)
by(*auto intro*: *bisim1-refl*)
qed *simp-all*
hence *bisim1-list1 t H* ({*0*:*Class* (*fst* (*method P C M*))=*None*; *the* (*snd* (*snd* (*snd* (*method P C M*))))},
Addr a # replicate (*max-vars* (*the* (*snd* (*snd* (*snd* (*method P C M*)))))) *undefined-value*)
[]
None [([], *Addr a # replicate* (*fst* (*snd* (*the* (*snd* (*snd* (*snd* (*method* (*compP2 P*) *C M*)))))))) *undefined-value*,
fst (*method* (*compP2 P*) *C M*), *M*, *0*)]
using *sees sees'* **by** *simp* **}**
thus *?thesis*
apply(*auto simp add*: *ta-bisim-def intro*!: *list-all2-all-nthI*)
apply(*case-tac* {|*ta*|}$_t$! *n*)
apply(*auto simp add*: *extNTA2JVM-def extNTA2J1-def*)
done
qed

end

sublocale *J1-JVM-conf-read* < *heap-conf?*: *J1-JVM-heap-conf*
by(*unfold-locales*)

sublocale *J1-JVM-conf-read* < *heap?*: *J1-heap*
apply(*rule J1-heap.intro*)
apply(*subst compP-heap*[*symmetric*, **where** *f*=λ- - - -. *compMb2*, *folded compP2-def*])
apply(*unfold-locales*)
done

end

7.17 Correctness of Stage: From intermediate language to JVM

theory *J1JVM* **imports** *J1JVMBisim* **begin**

context *J1-JVM-heap-base* **begin**

declare *τmove1.simps* [*simp del*] *τmoves1.simps* [*simp del*]

lemma *bisim1-insync-Throw-exec*:
assumes *bisim2*: *P*,*e2*,*h* ⊢ (*Throw ad*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
shows *τExec-movet-a P t* (*sync*$_V$ (*e1*) *e2*) *h* (*stk*, *loc*, *Suc* (*Suc* (*Suc* (*length* (*compE2 e1*) + *pc*))), *xcp*) ([*Addr ad*], *loc*, *6* + *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
proof −
from *bisim2* **have** *pc*: *pc* < *length* (*compE2 e2*) **and** [*simp*]: *xs* = *loc* **by**(*auto dest*: *bisim1-ThrowD*)
let *?pc* = *6* + *length* (*compE2 e1*) + *length* (*compE2 e2*)

```
  let ?stk = Addr ad # drop (size stk − 0) stk
  from bisim2 have xcp = ⌊ad⌋ ∨ xcp = None by(auto dest: bisim1-ThrowD)
  thus ?thesis
  proof
    assume [simp]: xcp = ⌊ad⌋
    have τExec-movet-a P t (sync_V (e1) e2) h (stk, loc, Suc (Suc (Suc (length (compE2 e1) + pc))),
⌊ad⌋) (?stk, loc, ?pc, None)
    proof(rule τExect1step[unfolded exec-move-def, OF exec-catch])
     from bisim1-xcp-Some-not-caught[OF bisim2[simplified], of λC M Ts T. compMb2 Suc (Suc (Suc
(length (compE2 e1)))) 0]
      have match-ex-table (compP2 P) (cname-of h ad) (Suc (Suc (Suc (length (compE2 e1) + pc))))
(compxE2 e2 (Suc (Suc (Suc (length (compE2 e1)))))) 0) = None
        by(simp add: compP2-def)
      thus match-ex-table (compP2 P) (cname-of h ad) (Suc (Suc (Suc (length (compE2 e1) + pc))))
(compxE2 (sync_V (e1) e2) 0 0) = ⌊(6 + length (compE2 e1) + length (compE2 e2), 0)⌋
        using pc
        by(auto simp add: compP2-def match-ex-table-append matches-ex-entry-def eval-nat-numeral
              dest: match-ex-table-pc-length-compE2)
    qed(insert pc, auto intro: τmove2xcp)
    thus ?thesis by simp
  next
    assume [simp]: xcp = None
    with bisim2 obtain pc'
      where τExec-movet-a P t e2 h (stk, loc, pc, None) ([Addr ad], loc, pc', ⌊ad⌋)
      and bisim': P, e2, h ⊢ (Throw ad, xs) ↔ ([Addr ad], loc, pc', ⌊ad⌋) and [simp]: xs = loc
      by(auto dest: bisim1-Throw-τExec-movet)
    hence τExec-movet-a P t (sync_V (e1) e2) h (stk, loc, Suc (Suc (Suc (length (compE2 e1) + pc))),
None) ([Addr ad], loc, Suc (Suc (Suc (length (compE2 e1) + pc'))), ⌊ad⌋)
      by−(rule Insync-τExectI)
    also let ?stk = Addr ad # drop (size [Addr ad] − 0) [Addr ad]
    from bisim' have pc': pc' < length (compE2 e2) by(auto dest: bisim1-ThrowD)
    have τExec-movet-a P t (sync_V (e1) e2) h ([Addr ad], loc, Suc (Suc (Suc (length (compE2 e1) +
pc'))), ⌊ad⌋) (?stk, loc, ?pc, None)
    proof(rule τExect1step[unfolded exec-move-def, OF exec-catch])
     from bisim1-xcp-Some-not-caught[OF bisim', of λC M Ts T. compMb2 Suc (Suc (Suc (length
(compE2 e1)))) 0]
      have match-ex-table (compP2 P) (cname-of h ad) (Suc (Suc (Suc (length (compE2 e1) + pc'))))
(compxE2 e2 (Suc (Suc (Suc (length (compE2 e1)))))) 0) = None
        by(simp add: compP2-def)
      thus match-ex-table (compP2 P) (cname-of h ad) (Suc (Suc (Suc (length (compE2 e1) + pc'))))
(compxE2 (sync_V (e1) e2) 0 0) = ⌊(6 + length (compE2 e1) + length (compE2 e2), 0)⌋
        using pc'
        by(auto simp add: compP2-def match-ex-table-append matches-ex-entry-def eval-nat-numeral
              dest: match-ex-table-pc-length-compE2)
    qed(insert pc', auto intro: τmove2xcp)
    finally (tranclp-trans) show ?thesis by simp
  qed
qed

end

primrec sim12-size :: ('a, 'b, 'addr) exp ⇒ nat
  and sim12-sizes :: ('a, 'b, 'addr) exp list ⇒ nat
where
```

```
  sim12-size (new C) = 0
| sim12-size (newA T⌊e⌉) = Suc (sim12-size e)
| sim12-size (Cast T e) = Suc (sim12-size e)
| sim12-size (e instanceof T) = Suc (sim12-size e)
| sim12-size (e «bop» e') = Suc (sim12-size e + sim12-size e')
| sim12-size (Val v) = 0
| sim12-size (Var V) = 0
| sim12-size (V := e) = Suc (sim12-size e)
| sim12-size (a⌊i⌉) = Suc (sim12-size a + sim12-size i)
| sim12-size (a⌊i⌉ := e) = Suc (sim12-size a + sim12-size i + sim12-size e)
| sim12-size (a·length) = Suc (sim12-size a)
| sim12-size (e·F{D}) = Suc (sim12-size e)
| sim12-size (e·F{D} := e') = Suc (sim12-size e + sim12-size e')
| sim12-size (e·compareAndSwap(D·F, e', e'')) = Suc (sim12-size e + sim12-size e' + sim12-size e'')
| sim12-size (e·M(es)) = Suc (sim12-size e + sim12-sizes es)
| sim12-size ({V:T=vo; e}) = Suc (sim12-size e)
| sim12-size (sync_V(e) e') = Suc (sim12-size e + sim12-size e')
| sim12-size (insync_V(a) e) = Suc (sim12-size e)
| sim12-size (e;; e') = Suc (sim12-size e + sim12-size e')
| sim12-size (if (e) e1 else e2) = Suc (sim12-size e)
| sim12-size (while(b) c) = Suc (Suc (sim12-size b))
| sim12-size (throw e) = Suc (sim12-size e)
| sim12-size (try e catch(C V) e') = Suc (sim12-size e)

| sim12-sizes [] = 0
| sim12-sizes (e # es) = sim12-size e + sim12-sizes es

lemma sim12-sizes-map-Val [simp]:
  sim12-sizes (map Val vs) = 0
by(induct vs) simp-all

lemma sim12-sizes-append [simp]:
  sim12-sizes (es @ es') = sim12-sizes es + sim12-sizes es'
by(induct es) simp-all

context JVM-heap-base begin

lemma τExec-mover-length-compE2-conv [simp]:
  assumes pc: pc ≥ length (compE2 e)
  shows τExec-mover ci P t e h (stk, loc, pc, xcp) s ⟷ s = (stk, loc, pc, xcp)
proof
  assume τExec-mover ci P t e h (stk, loc, pc, xcp) s
  thus s = (stk, loc, pc, xcp) using pc
    by induct(auto simp add: τexec-move-def)
qed auto

lemma τExec-movesr-length-compE2-conv [simp]:
  assumes pc: pc ≥ length (compEs2 es)
  shows τExec-movesr ci P t es h (stk, loc, pc, xcp) s ⟷ s = (stk, loc, pc, xcp)
proof
  assume τExec-movesr ci P t es h (stk, loc, pc, xcp) s
  thus s = (stk, loc, pc, xcp) using pc
    by induct(auto simp add: τexec-moves-def)
qed auto
```

end

context *J1-JVM-heap-base* **begin**

lemma assumes *wf*: *wf-J1-prog P*
defines [*simp*]: *sim-move* ≡ λ*e e'*. *if sim12-size e'* < *sim12-size e then τExec-mover-a else τExec-movet-a*
and [*simp*]: *sim-moves* ≡ λ*es es'*. *if sim12-sizes es'* < *sim12-sizes es then τExec-movesr-a else τExec-movest-a*
shows *exec-instr-simulates-red1*:
⟦ *P*, *E*, *h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*); *True*,*P*,*t* ⊢1 ⟨*e*, (*h*, *xs*)⟩ −*ta*→ ⟨*e'*, (*h'*, *xs'*)⟩; *bsok E n* ⟧
⟹ ∃ *pc'' stk'' loc'' xcp''*. *P*, *E*, *h'* ⊢ (*e'*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*) ∧
(*if τmove1 P h e*
then h = *h'* ∧ *sim-move e e' P t E h* (*stk*, *loc*, *pc*, *xcp*) (*stk''*, *loc''*, *pc''*, *xcp''*)
else ∃ *pc' stk' loc' xcp'*. *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*) ∧
exec-move-a P t E h (*stk'*, *loc'*, *pc'*, *xcp'*) (*extTA2JVM* (*compP2 P*) *ta*) *h'* (*stk''*, *loc''*, *pc''*, *xcp''*) ∧
¬ *τmove2* (*compP2 P*) *h stk' E pc' xcp'* ∧
(*call1 e* = *None* ∨ *no-call2 E pc* ∨ *pc'* = *pc* ∧ *stk'* = *stk* ∧ *loc'* = *loc* ∧ *xcp'* = *xcp*))
(**is** ⟦ -; -; - ⟧
⟹ ∃ *pc'' stk'' loc'' xcp''*. - ∧ *?exec ta E e e' h stk loc pc xcp h' pc'' stk'' loc'' xcp''*)

and *exec-instr-simulates-reds1*:
⟦ *P*, *Es*, *h* ⊢ (*es*, *xs*) [↔] (*stk*, *loc*, *pc*, *xcp*); *True*,*P*,*t* ⊢1 ⟨*es*, (*h*, *xs*)⟩ [−*ta*→] ⟨*es'*, (*h'*, *xs'*)⟩; *bsoks Es n* ⟧
⟹ ∃ *pc'' stk'' loc'' xcp''*. *P*, *Es*, *h'* ⊢ (*es'*, *xs'*) [↔] (*stk''*, *loc''*, *pc''*, *xcp''*) ∧
(*if τmoves1 P h es*
then h = *h'* ∧ *sim-moves es es' P t Es h* (*stk*, *loc*, *pc*, *xcp*) (*stk''*, *loc''*, *pc''*, *xcp''*)
else ∃ *pc' stk' loc' xcp'*. *τExec-movesr-a P t Es h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*) ∧
exec-moves-a P t Es h (*stk'*, *loc'*, *pc'*, *xcp'*) (*extTA2JVM* (*compP2 P*) *ta*) *h'* (*stk''*, *loc''*, *pc''*, *xcp''*) ∧
¬ *τmoves2* (*compP2 P*) *h stk' Es pc' xcp'* ∧
(*calls1 es* = *None* ∨ *no-calls2 Es pc* ∨ *pc'* = *pc* ∧ *stk'* = *stk* ∧ *loc'* = *loc* ∧ *xcp'* = *xcp*))
(**is** ⟦ -; -; - ⟧
⟹ ∃ *pc'' stk'' loc'' xcp''*. - ∧ *?execs ta Es es es' h stk loc pc xcp h' pc'' stk'' loc'' xcp''*)
proof(*induction E n e xs stk loc pc xcp* **and** *Es n es xs stk loc pc xcp*
arbitrary: *e' h' xs' Env T Env' T'* **and** *es' h' xs' Env Ts Env' Ts' rule*: *bisim1-bisims1-inducts-split*)
case (*bisim1Call1 obj n obj' xs stk loc pc xcp ps M'*)
note *IHobj* = *bisim1Call1.IH*(*2*)
note *IHparam* = *bisim1Call1.IH*(*4*)
note *bisim1* = ‹*P*,*obj*,*h* ⊢ (*obj'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim2* = ‹⋀*xs*. *P*,*ps*,*h* ⊢ (*ps*, *xs*) [↔] ([], *xs*, *0*, *None*)›
note *bsok* = ‹*bsok* (*obj*·*M'*(*ps*)) *n*›
note *red* = ‹*True*,*P*,*t* ⊢1 ⟨*obj'*·*M'*(*ps*),(*h*, *xs*)⟩ −*ta*→ ⟨*e'*,(*h'*, *xs'*)⟩›
from *red* **show** *?case*
proof(*cases*)
case (*Call1Obj E'*)
note [*simp*] = ‹*e'* = *E'*·*M'*(*ps*)›
and *red* = ‹*True*,*P*,*t* ⊢1 ⟨*obj'*,(*h*, *xs*)⟩ −*ta*→ ⟨*E'*,(*h'*, *xs'*)⟩›
from *red* **have** *τ*: *τmove1 P h obj'* = *τmove1 P h* (*obj'*·*M'*(*ps*)) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
moreover from *red* **have** *call1* (*obj'*·*M'*(*ps*)) = *call1 obj'* **by** *auto*

moreover from *IHobj*[*OF red*] *bsok*
obtain *pc''* *stk''* *loc''* *xcp''* **where** *bisim*: *P*,*obj*,*h'* ⊢ (*E'*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
and *redo*: *?exec ta obj obj' E' h stk loc pc xcp h' pc'' stk'' loc'' xcp''* **by** *auto*
from *bisim*
have *P*,*obj*·*M'*(*ps*),*h'* ⊢ (*E'*·*M'*(*ps*), *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
by(*rule bisim1-bisims1.bisim1Call1*)
moreover {
assume *no-call2 obj pc*
hence *no-call2* (*obj*·*M'*(*ps*)) *pc* ∨ *pc* = *length* (*compE2 obj*) **by**(*auto simp add*: *no-call2-def*) }
ultimately show *?thesis* **using** *redo*
by(*auto simp del*: *call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)(*blast intro*: *Call-τExecrI1 Call-τExectI1 exec-move-CallI1*)+
next
case (*Call1Params es v*)
note [*simp*] = ‹*obj'* = *Val v*› ‹*e'* = *Val v*·*M'*(*es*)›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*ps*, (*h*, *xs*)⟩ [−*ta*→] ⟨*es*, (*h'*, *xs'*)⟩›
from *red* **have** *τ*: *τmove1 P h* (*obj'*·*M'*(*ps*)) = *τmoves1 P h ps* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *execo*: *τExec-mover-a P t obj h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 obj*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*obj*·*M'*(*ps*)) *h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 obj*), *None*)
by−(*rule Call-τExecrI1*)
moreover from *IHparam*[*OF red*] *bsok* **obtain** *pc''* *stk''* *loc''* *xcp''*
where *bisim'*: *P*,*ps*,*h'* ⊢ (*es*, *xs'*) [↔] (*stk''*, *loc''*, *pc''*, *xcp''*)
and *exec'*: *?execs ta ps ps es h* [] *xs 0 None h' pc'' stk'' loc'' xcp''* **by** *auto*
have *?exec ta* (*obj*·*M'*(*ps*)) (*obj'*·*M'*(*ps*)) (*obj'*·*M*(*es*)) *h* [*v*] *loc* (*length* (*compE2 obj*)) *None h'* (*length* (*compE2 obj*) + *pc''*) (*stk''* @ [*v*]) *loc'' xcp''*
proof(*cases τmove1 P h* (*obj'*·*M'*(*ps*)))
case *True*
with *exec' τ* **have** [*simp*]: *h* = *h'*
and *e*: *sim-moves ps es P t ps h* ([], *xs*, *0*, *None*) (*stk''*, *loc''*, *pc''*, *xcp''*) **by** *auto*
from *e* **have** *sim-move* (*obj'*·*M'*(*ps*)) (*obj'*·*M'*(*es*)) *P t* (*obj*·*M'*(*ps*)) *h* ([] @ [*v*], *xs*, *length* (*compE2 obj*) + *0*, *None*) (*stk''* @ [*v*], *loc''*, *length* (*compE2 obj*) + *pc''*, *xcp''*)
by(*fastforce dest*: *Call-τExecrI2 Call-τExectI2*)
with *s True* **show** *?thesis* **by** *auto*
next
case *False*
with *exec' τ* **obtain** *pc' stk' loc' xcp'*
where *e*: *τExec-movesr-a P t ps h* ([], *xs*, *0*, *None*) (*stk'*, *loc'*, *pc'*, *xcp'*)
and *e'*: *exec-moves-a P t ps h* (*stk'*, *loc'*, *pc'*, *xcp'*) (*extTA2JVM* (*compP2 P*) *ta*) *h'* (*stk''*, *loc''*, *pc''*, *xcp''*)
and *τ'*: ¬ *τmoves2* (*compP2 P*) *h stk' ps pc' xcp'*
and *call*: *calls1 ps* = *None* ∨ *no-calls2 ps 0* ∨ *pc'* = *0* ∧ *stk'* = [] ∧ *loc'* = *xs* ∧ *xcp'* = *None* **by** *auto*
from *e* **have** *τExec-mover-a P t* (*obj*·*M'*(*ps*)) *h* ([] @ [*v*], *xs*, *length* (*compE2 obj*) + *0*, *None*) (*stk'* @ [*v*], *loc'*, *length* (*compE2 obj*) + *pc'*, *xcp'*) **by**(*rule Call-τExecrI2*)
moreover from *e'* **have** *exec-move-a P t* (*obj*·*M'*(*ps*)) *h* (*stk'* @ [*v*], *loc'*, *length* (*compE2 obj*) + *pc'*, *xcp'*) (*extTA2JVM* (*compP2 P*) *ta*) *h'* (*stk''* @ [*v*], *loc''*, *length* (*compE2 obj*) + *pc''*, *xcp''*)
by(*rule exec-move-CallI2*)
moreover from *τ' e'* **have** *τmove2* (*compP2 P*) *h* (*stk'* @ [*v*]) (*obj*·*M'*(*ps*)) (*length* (*compE2 obj*) + *pc'*) *xcp'* ⟹ *False*
by(*fastforce simp add*: *τmove2-iff τmoves2-iff τinstr-stk-drop-exec-moves split*: *if-split-asm*)
moreover from *red* **have** *call1* (*obj'*·*M'*(*ps*)) = *calls1 ps* **by**(*auto simp add*: *is-vals-conv*)

moreover have *no-calls2 ps 0* $\Longrightarrow$ *no-call2* (*obj*·*M*′(*ps*)) (*length* (*compE2 obj*)) ∨ *ps* = [] *calls1* [] = *None*
by(*auto simp add*: *no-calls2-def no-call2-def*)
ultimately show *?thesis* **using** *False s call*
by(*auto simp del*: *split-paired-Ex call1.simps calls1.simps*) *blast*
qed
moreover from *bisim*′
have *P*,*obj*·*M*′(*ps*),*h*′ ⊢ (*Val v*·*M*′(*es*), *xs*′) ↔ ((*stk*′′ @ [*v*]), *loc*′′, *length* (*compE2 obj*) + *pc*′′, *xcp*′′)
by(*rule bisim1-bisims1.bisim1CallParams*)
moreover from *bisim1* **have** *pc* ≠ *length* (*compE2 obj*) ⟶ *no-call2* (*obj*·*M*′(*ps*)) *pc*
by(*auto simp add*: *no-call2-def dest*: *bisim-Val-pc-not-Invoke bisim1-pc-length-compE2*)
ultimately show *?thesis* **using** *τ execo*
apply(*auto simp del*: *split-paired-Ex call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)
apply(*blast intro*: *τExec-mover-trans*|*fastforce elim*!: *τExec-mover-trans simp del*: *split-paired-Ex call1.simps calls1.simps*)+
done
next
case (*Call1ThrowObj a*)
note [*simp*] = ‹*obj*′ = *Throw a*› ‹*ta* = ε› ‹*e*′ = *Throw a*› ‹*h*′ = *h*› ‹*xs*′ = *xs*›
have *τ*: *τmove1 P h* (*Throw a*·*M*′(*ps*)) **by**(*rule τmove1CallThrowObj*)
from *bisim1* **have** *xcp* = ⌊*a*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*a*⌋
with *bisim1* **have** *P*, *obj*·*M*′(*ps*), *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
by(*auto intro*: *bisim1-bisims1.bisim1CallThrowObj*)
thus *?thesis* **using** *τ* **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim1* **obtain** *pc*′
where *τExec-mover-a P t obj h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc*′, ⌊*a*⌋)
and *bisim*′: *P*, *obj*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc*′, ⌊*a*⌋) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t* (*obj*·*M*′(*ps*)) *h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc*′, ⌊*a*⌋)
by−(*rule Call-τExecrI1*)
moreover from *bisim*′ **have** *P*, *obj*·*M*′(*ps*), *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc*′, ⌊*a*⌋)
by(*rule bisim1CallThrowObj*)
ultimately show *?thesis* **using** *τ* **by** *auto*
qed
next
case (*Call1ThrowParams vs a es*′ *v*)
note [*simp*] = ‹*obj*′ = *Val v*› ‹*ta* = ε› ‹*e*′ = *Throw a*› ‹*h*′ = *h*› ‹*xs*′ = *xs*›
and *ps* = ‹*ps* = *map Val vs* @ *Throw a* # *es*′›
from *bisim1* **have** [*simp*]: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t obj h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 obj*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*obj*·*M*′(*ps*)) *h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 obj*), *None*)
by−(*rule Call-τExecrI1*)
also from *bisims1-Throw-τExec-movest*[*OF bisim2*[*of xs*, *unfolded ps*]]
obtain *pc*′ **where** *exec*′: *τExec-movest-a P t* (*map Val vs* @ *Throw a* # *es*′) *h* ([], *xs*, *0*, *None*) (*Addr a* # *rev vs*, *xs*, *pc*′, ⌊*a*⌋)
and *bisim*′: *P*,*map Val vs* @ *Throw a* # *es*′,*h* ⊢ (*map Val vs* @ *Throw a* # *es*′, *xs*) [↔] (*Addr a* # *rev vs*, *xs*, *pc*′, ⌊*a*⌋)

by *auto*
from *Call-τExectI2*[*OF exec′*, *of obj M′ v*] *ps*
have *τExec-movet-a P t* (*obj·M′*(*ps*)) *h* ([*v*], *loc*, *length* (*compE2 obj*), *None*) (*Addr a # rev vs* @ [*v*], *xs*, *length* (*compE2 obj*) + *pc′*, ⌊*a*⌋) **by** *simp*
also from *bisim1-bisims1.bisim1CallThrowParams*[*OF bisim′*, *of obj M′ v*] *ps*
have *bisim′′*: *P*,*obj·M′*(*ps*),*h* ⊢ (*Throw a*, *xs*) ↔ (*Addr a # rev vs* @ [*v*], *xs*, *length* (*compE2 obj*) + *pc′*, ⌊*a*⌋) **by** *simp*
moreover have *τmove1 P h* (*obj′·M′*(*ps*)) **using** *ps* **by**(*auto intro*: *τmove1CallThrowParams*)
ultimately show *?thesis* **by** *fastforce*
next
case (*Red1CallExternal a Ta Ts Tr D vs va H′*)
hence [*simp*]: *obj′* = *addr a ps* = *map Val vs*
e′ = *extRet2J* (*addr a·M′*(*map Val vs*)) *va H′* = *h′ xs′* = *xs*
and *Ta*: *typeof-addr h a* = ⌊*Ta*⌋
and *iec*: *P* ⊢ *class-type-of Ta sees M′*: *Ts*→*Tr* = *Native in D*
and *redex*: *P*,*t* ⊢ ⟨*a·M′*(*vs*),*h*⟩ −*ta*→*ext* ⟨*va*,*h′*⟩ **by** *auto*
from *bisim1* **have** [*simp*]: *xs* = *loc* **by**(*auto dest*: *bisim-Val-loc-eq-xcp-None*)
have *τ*: *τmove1 P h* (*addr a·M′*(*map Val vs*)) ⟷ *τmove2* (*compP2 P*) *h* (*rev vs* @ [*Addr a*]) (*obj·M′*(*ps*)) (*length* (*compE2 obj*) + *length* (*compEs2 ps*)) *None* **using** *Ta iec*
by(*auto simp add*: *map-eq-append-conv τmove1.simps τmoves1.simps τmove2-iff compP2-def*)
from *bisim1* **have** *s*: *xcp* = *None lcl* (*h*, *xs*) = *loc*
and *τExec-mover-a P t obj h* (*stk*, *loc*, *pc*, *xcp*) ([*Addr a*], *loc*, *length* (*compE2 obj*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*obj·M′*(*ps*)) *h* (*stk*, *loc*, *pc*, *xcp*) ([*Addr a*], *loc*, *length* (*compE2 obj*), *None*)
by−(*rule Call-τExecrI1*)
also have *τExec-movesr-a P t ps h* ([], *loc*, *0*, *None*) (*rev vs*, *loc*, *length* (*compEs2 ps*), *None*)
unfolding ‹*ps* = *map Val vs*› **by**(*rule τExec-movesr-map-Val*)
from *Call-τExecrI2*[*OF this*, *of obj M′ Addr a*]
have *τExec-mover-a P t* (*obj·M′*(*ps*)) *h* ([*Addr a*], *loc*, *length* (*compE2 obj*), *None*) (*rev vs* @ [*Addr a*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), *None*) **by** *simp*
also (*rtranclp-trans*) **from** *bisim1* **have** *pc* ≤ *length* (*compE2 obj*) **by**(*rule bisim1-pc-length-compE2*)
hence *no-call2* (*obj·M′*(*ps*)) *pc* ∨ *pc* = *length* (*compE2 obj*) + *length* (*compEs2 ps*)
using *bisim1* **by**(*fastforce simp add*: *no-call2-def neq-Nil-conv dest*: *bisim-Val-pc-not-Invoke*)
moreover **{**
assume *pc* = *length* (*compE2 obj*) + *length* (*compEs2 ps*)
with ‹*τExec-mover-a P t obj h* (*stk*, *loc*, *pc*, *xcp*) ([*Addr a*], *loc*, *length* (*compE2 obj*), *None*)›
have *stk* = *rev vs* @ [*Addr a*] *xcp* = *None* **by** *auto* **}**
moreover
let *?ret* = *extRet2JVM* (*length ps*) *h′* (*rev vs* @ [*Addr a*]) *loc undefined undefined* (*length* (*compE2 obj*) + *length* (*compEs2 ps*)) [] *va*
let *?stk′* = *fst* (*hd* (*snd* (*snd ?ret*)))
let *?xcp′* = *fst ?ret*
let *?pc′* = *snd* (*snd* (*snd* (*snd* (*hd* (*snd* (*snd ?ret*))))))
from *redex* **have** *redex′*: (*ta*, *va*, *h′*) ∈ *red-external-aggr* (*compP2 P*) *t a M′ vs h*
by −(*rule red-external-imp-red-external-aggr*, *simp add*: *compP2-def*)
with *Ta iec redex′*
have *exec-move-a P t* (*obj·M′*(*ps*)) *h* (*rev vs* @ [*Addr a*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), *None*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*?stk′*, *loc*, *?pc′*, *?xcp′*)
unfolding *exec-move-def*
by−(*rule exec-instr*,*cases va*,(*force simp add*: *compP2-def simp del*: *split-paired-Ex*)+)
moreover have *P*,*obj·M′*(*ps*),*h′* ⊢ (*extRet2J1* (*addr a·M′*(*map Val vs*)) *va*, *loc*) ↔ (*?stk′*, *loc*, *?pc′*, *?xcp′*)
proof(*cases va*)

case (*RetVal v*)
have *P*,*obj*·*M*′(*ps*),*h*′ ⊢ (*Val v*, *loc*) ↔ ([*v*], *loc*, *length* (*compE2* (*obj*·*M*′(*ps*))), *None*)
by(*rule bisim1Val2*) *simp*
thus *?thesis* **unfolding** *RetVal* **by** *simp*
next
case (*RetExc ad*) **thus** *?thesis* **by**(*auto intro*: *bisim1CallThrow*)
next
case *RetStaySame*
from *bisims1-map-Val-append*[*OF bisims1Nil*, *of map Val vs vs P h*′ *loc*]
have *P*,*map Val vs*,*h*′ ⊢ (*map Val vs*, *loc*) [↔] (*rev vs*, *loc*, *length* (*compEs2* (*map Val vs*)), *None*)
by *simp*
hence *P*,*obj*·*M*′(*map Val vs*),*h*′ ⊢ (*addr a*·*M*′(*map Val vs*), *loc*) ↔ (*rev vs* @ [*Addr a*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2* (*map Val vs*)), *None*)
by(*rule bisim1CallParams*)
thus *?thesis* **using** *RetStaySame* **by** *simp*
qed
moreover from *redex Ta iec*
have *τmove1 P h* (*addr a*·*M*′(*map Val vs*)) ⟹ *ta* = *ε* ∧ *h*′ = *h*
by(*fastforce simp add*: *τmove1.simps τmoves1.simps map-eq-append-conv τexternal*′*-def τexternal-def dest*: *τexternal*′*-red-external-heap-unchanged τexternal*′*-red-external-TA-empty sees-method-fun*)
ultimately show *?thesis* **using** *τ*
apply(*cases τmove1 P h* (*addr a*·*M*′(*map Val vs*) :: *'addr expr1*))
apply(*auto simp del*: *split-paired-Ex simp add*: *compP2-def*)
apply(*blast intro*: *rtranclp.rtrancl-into-rtrancl rtranclp-into-tranclp1 τexec-moveI*)+
done
next
case (*Red1CallNull vs*)
note [*simp*] = ‹*h*′ = *h*› ‹*xs*′ = *xs*› ‹*ta* = *ε*› ‹*obj*′ = *null*› ‹*ps* = *map Val vs*› ‹*e*′ = *THROW NullPointer*›
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t obj h* (*stk*, *loc*, *pc*, *xcp*) ([*Null*], *loc*, *length* (*compE2 obj*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*obj*·*M*′(*map Val vs*)) *h* (*stk*, *loc*, *pc*, *xcp*) ([*Null*], *loc*, *length* (*compE2 obj*), *None*)
by−(*rule Call-τExecrI1*)
also have *τExec-movesr-a P t* (*map Val vs*) *h* ([], *loc*, *0*, *None*) (*rev vs*, *loc*, *length* (*compEs2* (*map Val vs*)), *None*)
proof(*cases vs*)
case *Nil* **thus** *?thesis* **by**(*auto*)
next
case *Cons*
with *bisims1-refl*[*of P h map Val vs loc*, *simplified*] **show** *?thesis*
by −(*drule bisims1-Val-τExec-moves*, *auto*)
qed
from *Call-τExecrI2*[*OF this*, *of obj M*′ *Null*]
have *τExec-mover-a P t* (*obj*·*M*′(*map Val vs*)) *h* ([*Null*], *loc*, *length* (*compE2 obj*), *None*) (*rev vs* @ [*Null*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2* (*map Val vs*)), *None*) **by** *simp*
also (*rtranclp-trans*) **{**
have *τmove2* (*compP2 P*) *h* (*rev vs* @ [*Null*]) (*obj*·*M*′(*map Val vs*)) (*length* (*compE2 obj*) + *length* (*compEs2* (*map Val vs*))) *None*
by(*simp add*: *τmove2-iff*)
moreover have *exec-move-a P t* (*obj*·*M*′(*map Val vs*)) *h* (*rev vs* @ [*Null*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2* (*map Val vs*)), *None*) *ε h* (*rev vs* @ [*Null*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2* (*map Val vs*)), ⌊*addr-of-sys-xcpt NullPointer*⌋)

unfolding *exec-move-def* **by**(*cases vs*)(*auto intro*: *exec-instr*)
ultimately have *τExec-movet-a P t* (*obj·M′*(*map Val vs*)) *h* (*rev vs* @ [*Null*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2* (*map Val vs*)), *None*) (*rev vs* @ [*Null*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2* (*map Val vs*)), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*auto intro*: *τexec-moveI simp add*: *compP2-def*) **}**
also have *τmove1 P h* (*null·M′*(*map Val vs*)) **by**(*auto simp add*: *τmove1.simps τmoves1.simps map-eq-append-conv*)
moreover have *P*,*obj·M′*(*map Val vs*),*h* ⊢ (*THROW NullPointer*, *loc*) ↔ ((*rev vs* @ [*Null*]), *loc*, *length* (*compE2 obj*) + *length* (*compEs2* (*map Val vs*)), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule bisim1CallThrow*) *simp*
ultimately show *?thesis* **using** *s* **by**(*auto simp del*: *split-paired-Ex*)
qed
next
case *bisim1Val2* **thus** *?case* **by** *fastforce*
next
case (*bisim1New C′ n xs*)
have *τ*: ¬ *τmove1 P h* (*new C′*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from ‹*True*,*P*,*t* ⊢*1* ⟨*new C′*,(*h*, *xs*)⟩ −*ta*→ ⟨*e′*,(*h′*, *xs′*)⟩› **show** *?case*
proof *cases*
case (*Red1New a*)
hence *exec-meth-a* (*compP2 P*) [*New C′*] [] *t h* ([], *xs*, *0*, *None*) {|*NewHeapElem a* (*Class-type C′*)|} *h′* ([*Addr a*], *xs*, *Suc 0*, *None*)
and [*simp*]: *e′* = *addr a xs′* = *xs ta* = {|*NewHeapElem a* (*Class-type C′*)|}
by (*auto intro*!: *exec-instr simp add*: *compP2-def simp del*: *fun-upd-apply cong cong del*: *image-cong-simp*)
moreover have *P*, *new C′*, *h′* ⊢ (*addr a*, *xs*) ↔ ([*Addr a*], *xs*, *length* (*compE2* (*new C′*)), *None*)
by(*rule bisim1Val2*)(*simp*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [] (*new C′*) *0 None* **by**(*simp add*: *τmove2-iff*)
ultimately show *?thesis* **using** *τ*
by(*fastforce simp add*: *exec-move-def ta-upd-simps*)
next
case *Red1NewFail*
hence *exec-meth-a* (*compP2 P*) [*New C′*] [] *t h* ([], *xs*, *0*, *None*) *ε h′* ([], *xs*, *0*, ⌊*addr-of-sys-xcpt OutOfMemory*⌋)
and [*simp*]: *ta* = *ε xs′* = *xs e′* = *THROW OutOfMemory*
by(*auto intro*!:*exec-instr simp add*: *compP2-def simp del*: *fun-upd-apply*)
moreover have *P*, *new C′*, *h′* ⊢ (*THROW OutOfMemory*, *xs*) ↔ ([], *xs*, *0*, ⌊*addr-of-sys-xcpt OutOfMemory*⌋)
by(*rule bisim1NewThrow*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [] (*new C′*) *0 None* **by**(*simp add*: *τmove2-iff*)
ultimately show *?thesis* **using** *τ* **by**(*fastforce simp add*: *exec-move-def*)
qed
next
case *bisim1NewThrow* **thus** *?case* **by** *fastforce*
next
case (*bisim1NewArray E n e xs stk loc pc xcp U*)
note *IH* = *bisim1NewArray.IH*(*2*)
note *bisim* = ‹*P*,*E*,*h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*newA U*⌊*e*⌉,(*h*, *xs*)⟩ −*ta*→ ⟨*e′*,(*h′*, *xs′*)⟩›
note *bsok* = ‹*bsok* (*newA U*⌊*E*⌉) *n*›
from *red* **show** *?case*
proof *cases*
case (*New1ArrayRed ee′*)
note [*simp*] = ‹*e′* = *newA U*⌊*ee′*⌉›

and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e*,(*h*, *xs*)⟩ −*ta*→ ⟨*ee′*, (*h′*, *xs′*)⟩›
from *red* **have** *τmove1 P h* (*newA U*⌊*e*⌉) = *τmove1 P h e* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
moreover from *red* **have** *call1* (*newA U*⌊*e*⌉) = *call1 e* **by** *auto*
moreover from *IH*[*OF red*] *bsok*
obtain *pc″ stk″ loc″ xcp″* **where** *bisim*: *P*,*E*,*h′* ⊢ (*ee′*, *xs′*) ↔ (*stk″*, *loc″*, *pc″*, *xcp″*)
and *redo*: *?exec ta E e ee′ h stk loc pc xcp h′ pc″ stk″ loc″ xcp″* **by** *auto*
from *bisim*
have *P*,*newA U*⌊*E*⌉,*h′* ⊢ (*newA U*⌊*ee′*⌉, *xs′*) ↔ (*stk″*, *loc″*, *pc″*, *xcp″*)
by(*rule bisim1-bisims1.bisim1NewArray*)
moreover {
assume *no-call2 E pc*
hence *no-call2* (*newA U*⌊*E*⌉) *pc* **by**(*auto simp add*: *no-call2-def*) **}**
ultimately show *?thesis* **using** *redo*
by(*auto simp del*: *call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)(*blast intro*: *NewArray-τExecrI NewArray-τExectI exec-move-newArrayI*)+
next
case (*Red1NewArray i a*)
note [*simp*] = ‹*e* = *Val* (*Intg i*)› ‹*ta* = {|*NewHeapElem a* (*Array-type U* (*nat* (*sint i*)))|}› ‹*e′* = *addr a*› ‹*xs′* = *xs*›
and *new* = ‹(*h′*, *a*) ∈ *allocate h* (*Array-type U* (*nat* (*sint i*)))›
from *bisim* **have** *s*: *xcp* = *None xs* = *loc* **by**(*auto dest*: *bisim-Val-loc-eq-xcp-None*)
from *bisim* **have** *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *xcp*) ([*Intg i*], *loc*, *length* (*compE2 E*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*newA U*⌊*E*⌉) *h* (*stk*, *loc*, *pc*, *xcp*) ([*Intg i*], *loc*, *length* (*compE2 E*), *None*)
by(*rule NewArray-τExecrI*)
moreover from *new* ‹*0* <=*s i*›
have *exec-move-a P t* (*newA U*⌊*E*⌉) *h* ([*Intg i*], *loc*, *length* (*compE2 E*), *None*) {|*NewHeapElem a* (*Array-type U* (*nat* (*sint i*)))|} *h′* ([*Addr a*], *loc*, *Suc* (*length* (*compE2 E*)), *None*)
by (*auto intro*!: *exec-instr simp add*: *compP2-def exec-move-def cong del*: *image-cong-simp*)
moreover have *τmove2* (*compP2 P*) *h* [*Intg i*] (*newA U*⌊*E*⌉) (*length* (*compE2 E*)) *None* ⟹ *False* **by**(*simp add*: *τmove2-iff*)
moreover have ¬ *τmove1 P h* (*newA U*⌊*Val* (*Intg i*)⌉) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
moreover have *P*, *newA U*⌊*E*⌉, *h′* ⊢ (*addr a*, *loc*) ↔ ([*Addr a*], *loc*, *length* (*compE2* (*newA U*⌊*E*⌉)), *None*)
by(*rule bisim1Val2*) *simp*
ultimately show *?thesis* **using** *s* **by**(*auto simp del*: *fun-upd-apply simp add*: *ta-upd-simps*) *blast*
next
case (*Red1NewArrayNegative i*)
note [*simp*] = ‹*e* = *Val* (*Intg i*)› ‹*e′* = *THROW NegativeArraySize*› ‹*h′* = *h*› ‹*xs′* = *xs*› ‹*ta* = ε›
have ¬ *τmove1 P h* (*newA U*⌊*Val* (*Intg i*)⌉) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
moreover from *bisim* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *xcp*) ([*Intg i*], *loc*, *length* (*compE2 E*), *None*)
by(*auto dest*: *bisim1Val2D1*)
moreover from ‹*i* <*s* *0*›
have *exec-meth-a* (*compP2 P*) (*compE2* (*newA U*⌊*E*⌉)) (*compxE2* (*newA U*⌊*E*⌉) *0 0*) *t h* ([*Intg i*], *loc*, *length* (*compE2 E*), *None*) ε *h* ([*Intg i*], *loc*, *length* (*compE2 E*), ⌊*addr-of-sys-xcpt NegativeArraySize*⌋)
by −(*rule exec-instr*, *auto simp add*: *compP2-def*)
moreover have *τmove2* (*compP2 P*) *h* [*Intg i*] (*newA U*⌊*E*⌉) (*length* (*compE2 E*)) *None* ⟹ *False* **by**(*simp add*: *τmove2-iff*)
moreover
have *P*,*newA U*⌊*E*⌉,*h* ⊢ (*THROW NegativeArraySize*, *loc*) ↔ ([*Intg i*], *loc*, *length* (*compE2 E*),

```
⌊addr-of-sys-xcpt NegativeArraySize⌋)
      by(auto intro!: bisim1-bisims1.bisim1NewArrayFail)
    ultimately show ?thesis using s
      by(auto simp add: exec-move-def)(blast intro: NewArray-τExecrI)
  next
    case (Red1NewArrayFail i)
    note [simp] = ‹e = Val (Intg i)› ‹e' = THROW OutOfMemory› ‹xs' = xs› ‹ta = ε› ‹h' = h›
      and new = ‹allocate h (Array-type U (nat (sint i))) = {}›
    have ¬ τmove1 P h (newA U⌊Val (Intg i)⌉) by(auto simp add: τmove1.simps τmoves1.simps)
    moreover from bisim have s: xcp = None xs = loc
      and τExec-mover-a P t E h (stk, loc, pc, xcp) ([Intg i], loc, length (compE2 E), None)
      by(auto dest: bisim1Val2D1)
    moreover from ‹0 <=s i› new
   have exec-meth-a (compP2 P) (compE2 (newA U⌊E⌉)) (compxE2 (newA U⌊E⌉) 0 0) t h ([Intg i],
loc, length (compE2 E), None) ε h' ([Intg i], loc, length (compE2 E), ⌊addr-of-sys-xcpt OutOfMemory⌋)
      by −(rule exec-instr, auto simp add: compP2-def)
     moreover have τmove2 (compP2 P) h [Intg i] (newA U⌊E⌉) (length (compE2 E)) None ⟹
False by(simp add: τmove2-iff)
    moreover
     have P,newA U⌊E⌉,h' ⊢ (THROW OutOfMemory, loc) ↔ ([Intg i], loc, length (compE2 E),
⌊addr-of-sys-xcpt OutOfMemory⌋)
      by(auto intro!: bisim1-bisims1.bisim1NewArrayFail)
   ultimately show ?thesis using s by (auto simp add: exec-move-def)(blast intro: NewArray-τExecrI)
  next
    case (New1ArrayThrow a)
    note [simp] = ‹e = Throw a› ‹h' = h› ‹xs' = xs› ‹ta = ε› ‹e' = Throw a›
    have τ: τmove1 P h (newA U⌊e⌉) by(auto intro: τmove1NewArrayThrow)
    from bisim have xcp = ⌊a⌋ ∨ xcp = None by(auto dest: bisim1-ThrowD)
    thus ?thesis
    proof
      assume [simp]: xcp = ⌊a⌋
      with bisim have P,newA U⌊E⌉, h ⊢ (Throw a, xs) ↔ (stk, loc, pc, xcp)
        by(auto intro: bisim1-bisims1.bisim1NewArrayThrow)
      thus ?thesis using τ by(fastforce)
    next
      assume [simp]: xcp = None
      with bisim obtain pc'
        where τExec-mover-a P t E h (stk, loc, pc, None) ([Addr a], loc, pc', ⌊a⌋)
        and bisim': P, E, h ⊢ (Throw a, xs) ↔ ([Addr a], loc, pc', ⌊a⌋) and [simp]: xs = loc
        by(auto dest: bisim1-Throw-τExec-mover)
      hence τExec-mover-a P t (newA U⌊E⌉) h (stk, loc, pc, None) ([Addr a], loc, pc', ⌊a⌋)
        by−(rule NewArray-τExecrI)
      moreover from bisim' have P, newA U⌊E⌉, h ⊢ (Throw a, xs) ↔ ([Addr a], loc, pc', ⌊a⌋)
        by(rule bisim1-bisims1.bisim1NewArrayThrow)
      ultimately show ?thesis using τ by auto
    qed
  qed
next
  case bisim1NewArrayThrow thus ?case by auto
next
  case bisim1NewArrayFail thus ?case by auto
next
  case (bisim1Cast E n e xs stk loc pc xcp U)
  note IH = bisim1Cast.IH(2)
```

note *bisim* = ‹*P*,*E*,*h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*Cast U e*,(*h*, *xs*)⟩ −*ta*→ ⟨*e*′,(*h*′, *xs*′)⟩›
note *bsok* = ‹*bsok* (*Cast U E*) *n*›
from *red* **show** *?case*
proof *cases*
case (*Cast1Red ee*′)
note [*simp*] = ‹*e*′ = *Cast U ee*′›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e*,(*h*, *xs*)⟩ −*ta*→ ⟨*ee*′, (*h*′, *xs*′)⟩›
from *red* **have** *τmove1 P h* (*Cast U e*) = *τmove1 P h e* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
moreover from *red* **have** *call1* (*Cast U e*) = *call1 e* **by** *auto*
moreover from *IH*[*OF red*] *bsok*
obtain *pc*′′ *stk*′′ *loc*′′ *xcp*′′ **where** *bisim*: *P*,*E*,*h*′ ⊢ (*ee*′, *xs*′) ↔ (*stk*′′, *loc*′′, *pc*′′, *xcp*′′)
and *redo*: *?exec ta E e ee*′ *h stk loc pc xcp h*′ *pc*′′ *stk*′′ *loc*′′ *xcp*′′ **by** *auto*
from *bisim*
have *P*,*Cast U E*,*h*′ ⊢ (*Cast U ee*′, *xs*′) ↔ (*stk*′′, *loc*′′, *pc*′′, *xcp*′′)
by(*rule bisim1-bisims1.bisim1Cast*)
moreover {
assume *no-call2 E pc*
hence *no-call2* (*Cast U E*) *pc* **by**(*auto simp add*: *no-call2-def*) }
ultimately show *?thesis* **using** *redo*
by(*auto simp del*: *call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)(*blast intro*: *Cast-τExecrI Cast-τExectI exec-move-CastI*)+
next
case (*Red1Cast c U*′)
hence [*simp*]: *e* = *Val c ta* = *ε e*′ = *Val c h*′ = *h xs*′ = *xs*
and *type*: *typeof*$_h$ *c* = ⌊*U*′⌋ *P* ⊢ *U*′ ≤ *U* **by** *auto*
from *bisim* **have** *s*: *xcp* = *None xs* = *loc* **by**(*auto dest*: *bisim-Val-loc-eq-xcp-None*)
from *bisim* **have** *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *xcp*) ([*c*], *loc*, *length* (*compE2 E*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*Cast U E*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*c*], *loc*, *length* (*compE2 E*), *None*)
by(*rule Cast-τExecrI*)
moreover from *type*
have *exec-meth-a* (*compP2 P*) (*compE2* (*Cast U E*)) (*compxE2* (*Cast U E*) *0 0*) *t h* ([*c*], *loc*, *length* (*compE2 E*), *None*) *ε h*′ ([*c*], *loc*, *Suc* (*length* (*compE2 E*)), *None*)
by(*auto intro*!: *exec-instr simp add*: *compP2-def*)
moreover have *τmove2* (*compP2 P*) *h* [*c*] (*Cast U E*) (*length* (*compE2 E*)) *None* **by**(*simp add*: *τmove2-iff*)
ultimately have *τExec-mover-a P t* (*Cast U E*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*c*], *loc*, *Suc* (*length* (*compE2 E*)), *None*)
by(*fastforce elim*: *rtranclp.rtrancl-into-rtrancl intro*: *τexec-moveI simp add*: *exec-move-def compP2-def*)
moreover have *τmove1 P h* (*Cast U* (*Val c*)) **by**(*rule τmove1CastRed*)
moreover
have *P*, *Cast U E*, *h*′ ⊢ (*Val c*, *loc*) ↔ ([*c*], *loc*, *length* (*compE2* (*Cast U E*)), *None*)
by(*rule bisim1Val2*) *simp*
ultimately show *?thesis* **using** *s* **by**(*auto simp add*: *exec-move-def*)
next
case (*Red1CastFail v U*′)
note [*simp*] = ‹*e* = *Val v*› ‹*e*′ = *THROW ClassCast*› ‹*h*′ = *h*› ‹*xs*′ = *xs*› ‹*ta* = *ε*›
moreover from *bisim* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 E*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*Cast U E*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 E*), *None*)
by(*auto elim*: *Cast-τExecrI*)

moreover from ‹*typeof*$_{hp\ (h,\ xs)}$ $v = \lfloor U' \rfloor$› ‹$\neg\ P \vdash\ U' \leq\ U$›
have *exec-meth-a* (*compP2 P*) (*compE2* (*Cast U E*)) (*compxE2* (*Cast U E*) *0 0*) *t h* ([*v*], *loc*, *length* (*compE2 E*), *None*) *ε h* ([*v*], *loc*, *length* (*compE2 E*), ⌊*addr-of-sys-xcpt ClassCast*⌋)
by −(*rule exec-instr*, *auto simp add*: *compP2-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v*] (*Cast U E*) (*length* (*compE2 E*)) *None* **by**(*simp add*: *τmove2-iff*)
ultimately have *τExec-movet-a P t* (*Cast U E*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 E*), ⌊*addr-of-sys-xcpt ClassCast*⌋)
by(*fastforce simp add*: *exec-move-def compP2-def intro*: *rtranclp-into-tranclp1 τexec-moveI*)
moreover have *τmove1 P h* (*Cast U* (*Val v*)) **by**(*rule τmove1CastRed*)
moreover
have *P*,*Cast U E*,*h* ⊢ (*THROW ClassCast*, *loc*) ↔ ([*v*], *loc*, *length* (*compE2 E*), ⌊*addr-of-sys-xcpt ClassCast*⌋)
by(*auto intro*!: *bisim1-bisims1.bisim1CastFail*)
ultimately show *?thesis* **using** *s* **by**(*auto simp add*: *exec-move-def*)
next
case [*simp*]: (*Cast1Throw a*)
have *τ*: *τmove1 P h* (*Cast U e*) **by**(*auto intro*: *τmove1CastThrow*)
from *bisim* **have** *xcp* = ⌊*a*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*a*⌋
with *bisim* **have** *P*,*Cast U E*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
by(*auto intro*: *bisim1-bisims1.bisim1CastThrow*)
thus *?thesis* **using** *τ* **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim* **obtain** *pc′*
where *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
and *bisim′*: *P*, *E*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t* (*Cast U E*) *h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
by−(*rule Cast-τExecrI*)
moreover from *bisim′* **have** *P*, *Cast U E*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
by−(*rule bisim1-bisims1.bisim1CastThrow*, *auto*)
ultimately show *?thesis* **using** *τ* **by** *auto*
qed
qed
next
case *bisim1CastThrow* **thus** *?case* **by** *auto*
next
case *bisim1CastFail* **thus** *?case* **by** *auto*
next
case (*bisim1InstanceOf E n e xs stk loc pc xcp U*)
note *IH* = *bisim1InstanceOf*(*2*)
note *bisim* = ‹*P*,*E*,*h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e instanceof U*,(*h*, *xs*)⟩ −*ta*→ ⟨*e′*,(*h′*, *xs′*)⟩›
note *bsok* = ‹*bsok* (*E instanceof U*) *n*›
from *red* **show** *?case*
proof *cases*
case (*InstanceOf1Red ee′*)
note [*simp*] = ‹*e′* = *ee′ instanceof U*›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e*,(*h*, *xs*)⟩ −*ta*→ ⟨*ee′*, (*h′*, *xs′*)⟩›
from *red* **have** *τmove1 P h* (*e instanceof U*) = *τmove1 P h e* **by**(*auto simp add*: *τmove1.simps*

τmoves1.simps)
moreover from *red* **have** *call1* (*e instanceof U*) = *call1 e* **by** *auto*
moreover from *IH*[*OF red*] *bsok*
obtain *pc''* *stk''* *loc''* *xcp''* **where** *bisim*: *P*,*E*,*h'* ⊢ (*ee'*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
and *redo*: *?exec ta E e ee' h stk loc pc xcp h' pc'' stk'' loc'' xcp''* **by** *auto*
from *bisim*
have *P*,*E instanceof U*,*h'* ⊢ (*ee' instanceof U*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
by(*rule bisim1-bisims1.bisim1InstanceOf*)
moreover {
assume *no-call2 E pc*
hence *no-call2* (*E instanceof U*) *pc* **by**(*auto simp add*: *no-call2-def*) **}**
ultimately show *?thesis* **using** *redo*
by(*auto simp del*: *call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)(*blast intro*: *InstanceOf-τExecrI InstanceOf-τExectI exec-move-InstanceOfI*)+
next
case (*Red1InstanceOf c U' b*)
hence [*simp*]: *e* = *Val c ta* = *ε e'* = *Val* (*Bool* (*c* ≠ *Null* ∧ *P* ⊢ *U'* ≤ *U*)) *h'* = *h xs'* = *xs*
b = (*c* ≠ *Null* ∧ *P* ⊢ *U'* ≤ *U*)
and *type*: *typeof*$_h$ *c* = ⌊*U'*⌋ **by** *auto*
from *bisim* **have** *s*: *xcp* = *None xs* = *loc* **by**(*auto dest*: *bisim-Val-loc-eq-xcp-None*)
from *bisim* **have** *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *xcp*) ([*c*], *loc*, *length* (*compE2 E*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*E instanceof U*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*c*], *loc*, *length* (*compE2 E*), *None*)
by(*rule InstanceOf-τExecrI*)
moreover from *type*
have *exec-meth-a* (*compP2 P*) (*compE2* (*E instanceof U*)) (*compxE2* (*E instanceof U*) *0 0*) *t h* ([*c*], *loc*, *length* (*compE2 E*), *None*) *ε h'* ([*Bool b*], *loc*, *Suc* (*length* (*compE2 E*)), *None*)
by(*auto intro*!: *exec-instr simp add*: *compP2-def*)
moreover have *τmove2* (*compP2 P*) *h* [*c*] (*E instanceof U*) (*length* (*compE2 E*)) *None* **by**(*simp add*: *τmove2-iff*)
ultimately have *τExec-mover-a P t* (*E instanceof U*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*Bool b*], *loc*, *Suc* (*length* (*compE2 E*)), *None*)
by(*fastforce elim*: *rtranclp.rtrancl-into-rtrancl intro*: *τexec-moveI simp add*: *exec-move-def compP2-def*)
moreover have *τmove1 P h* ((*Val c*) *instanceof U*) **by**(*rule τmove1InstanceOfRed*)
moreover
have *P*, *E instanceof U*, *h'* ⊢ (*Val* (*Bool b*), *loc*) ↔ ([*Bool b*], *loc*, *length* (*compE2* (*E instanceof U*)), *None*)
by(*rule bisim1Val2*) *simp*
ultimately show *?thesis* **using** *s* **by**(*auto simp add*: *exec-move-def*)
next
case (*InstanceOf1Throw a*)
note [*simp*] = ‹*e* = *Throw a*› ‹*h'* = *h*› ‹*xs'* = *xs*› ‹*ta* = *ε*› ‹*e'* = *Throw a*›
have *τ*: *τmove1 P h* (*e instanceof U*) **by**(*auto intro*: *τmove1InstanceOfThrow*)
from *bisim* **have** *xcp* = ⌊*a*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*a*⌋
with *bisim* **have** *P*,*E instanceof U*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
by(*auto intro*: *bisim1-bisims1.bisim1InstanceOfThrow*)
thus *?thesis* **using** *τ* **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim* **obtain** *pc'*
where *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc'*, ⌊*a*⌋)

```
      and bisim': P, E, h ⊢ (Throw a, xs) ↔ ([Addr a], loc, pc', ⌊a⌋) and [simp]: xs = loc
      by(auto dest: bisim1-Throw-τExec-mover)
    hence τExec-mover-a P t (E instanceof U) h (stk, loc, pc, None) ([Addr a], loc, pc', ⌊a⌋)
      by−(rule InstanceOf-τExecrI)
    moreover from bisim' have P, E instanceof U, h ⊢ (Throw a, xs) ↔ ([Addr a], loc, pc', ⌊a⌋)
      by−(rule bisim1-bisims1.bisim1InstanceOfThrow, auto)
    ultimately show ?thesis using τ by auto
  qed
 qed
next
 case bisim1InstanceOfThrow thus ?case by auto
next
 case bisim1Val thus ?case by fastforce
next
 case (bisim1Var V n xs)
 from ‹True,P,t ⊢1 ⟨Var V,(h, xs)⟩ −ta→ ⟨e',(h', xs')⟩› show ?case
 proof cases
  case (Red1Var v)
  hence exec-meth-a (compP2 P) [Load V] [] t h ([], xs, 0, None) ε h ([v], xs, 1, None)
    and [simp]: ta = ε h' = h xs' = xs e' = Val v
    by(auto intro: exec-instr)
  moreover have τmove2 (compP2 P) h [] (Var V) 0 None by(simp add: τmove2-iff)
  ultimately have τExec-movet-a P t (Var V) h ([], xs, 0, None) ([v], xs, 1, None)
    by(auto intro: τExect1step simp add: exec-move-def compP2-def)
  moreover have P, Var V, h ⊢ (Val v, xs) ↔ ([v], xs, length (compE2 (Var V)), None)
    by(rule bisim1Val2) simp
  moreover have τmove1 P h (Var V) by(rule τmove1Var)
  ultimately show ?thesis by(fastforce)
 qed
next
 case (bisim1BinOp1 e1 n e1' xs stk loc pc xcp e2 bop)
 note IH1 = bisim1BinOp1.IH(2)
 note IH2 = bisim1BinOp1.IH(4)
 note bisim1 = ‹P,e1,h ⊢ (e1', xs) ↔ (stk, loc, pc, xcp)›
 note bisim2 = ‹⋀xs. P,e2,h ⊢ (e2, xs) ↔ ([], xs, 0, None)›
 note bsok = ‹bsok (e1 «bop» e2) n›
 from ‹True,P,t ⊢1 ⟨e1' «bop» e2,(h, xs)⟩ −ta→ ⟨e',(h', xs')⟩› show ?case
 proof cases
  case (Bin1OpRed1 E')
  note [simp] = ‹e' = E' «bop» e2›
    and red = ‹True,P,t ⊢1 ⟨e1',(h, xs)⟩ −ta→ ⟨E',(h', xs')⟩›
  from red have τmove1 P h (e1' «bop» e2) = τmove1 P h e1' by(auto simp add: τmove1.simps
τmoves1.simps)
  moreover from red have call1 (e1' «bop» e2) = call1 e1' by auto
  moreover from IH1[OF red] bsok
  obtain pc'' stk'' loc'' xcp'' where bisim: P,e1,h' ⊢ (E', xs') ↔ (stk'', loc'', pc'', xcp'')
    and redo: ?exec ta e1 e1' E' h stk loc pc xcp h' pc'' stk'' loc'' xcp'' by auto
  from bisim
  have P,e1«bop»e2,h' ⊢ (E'«bop»e2, xs') ↔ (stk'', loc'', pc'', xcp'')
    by(rule bisim1-bisims1.bisim1BinOp1)
  moreover {
    assume no-call2 e1 pc
    hence no-call2 (e1«bop»e2) pc ∨ pc = length (compE2 e1) by(auto simp add: no-call2-def) }
  ultimately show ?thesis using redo
```

by(*auto simp del*: *call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)(*blast intro*: *BinOp-τExecrI1 BinOp-τExectI1 exec-move-BinOpI1*)+
next
case (*Bin1OpRed2 E′ v*)
note [*simp*] = ‹*e1′ = Val v*› ‹*e′ = Val v «bop» E′*›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e2*,(*h*, *xs*)⟩ −*ta*→ ⟨*E′*,(*h′*, *xs′*)⟩›
from *red* **have** *τ*: *τmove1 P h* (*Val v «bop» e2*) = *τmove1 P h e2* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *exec1*: *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *None*) ([*v*], *xs*, *length* (*compE2 e1*), *None*)
by(*auto dest*: *bisim1Val2D1*)
from *exec1* **have** *τExec-mover-a P t* (*e1«bop»e2*) *h* (*stk*, *loc*, *pc*, *None*) ([*v*], *xs*, *length* (*compE2 e1*), *None*)
by(*rule BinOp-τExecrI1*)
moreover
from *IH2*[*OF red*] *bsok* **obtain** *pc″ stk″ loc″ xcp″*
where *bisim′*: *P*,*e2*,*h′* ⊢ (*E′*, *xs′*) ↔ (*stk″*, *loc″*, *pc″*, *xcp″*)
and *exec′*: *?exec ta e2 e2 E′ h* [] *xs 0 None h′ pc″ stk″ loc″ xcp″* **by** *auto*
have *?exec ta* (*e1«bop»e2*) (*Val v«bop»e2*) (*Val v«bop»E′*) *h* ([] @ [*v*]) *xs* (*length* (*compE2 e1*) + *0*) *None h′* (*length* (*compE2 e1*) + *pc″*) (*stk″* @ [*v*]) *loc″ xcp″*
proof(*cases τmove1 P h* (*Val v «bop» e2*))
case *True*
with *exec′ τ* **have** [*simp*]: *h* = *h′* **and** *e*: *sim-move e2 E′ P t e2 h* ([], *xs*, *0*, *None*) (*stk″*, *loc″*, *pc″*, *xcp″*) **by** *auto*
from *e* **have** *sim-move* (*Val v«bop»e2*) (*Val v«bop»E′*) *P t* (*e1 «bop» e2*) *h* ([] @ [*v*], *xs*, *length* (*compE2 e1*) + *0*, *None*) (*stk″* @ [*v*], *loc″*, *length* (*compE2 e1*) + *pc″*, *xcp″*)
by(*fastforce dest*: *BinOp-τExecrI2 BinOp-τExectI2*)
with *True* **show** *?thesis* **by** *auto*
next
case *False*
with *exec′ τ* **obtain** *pc′ stk′ loc′ xcp′*
where *e*: *τExec-mover-a P t e2 h* ([], *xs*, *0*, *None*) (*stk′*, *loc′*, *pc′*, *xcp′*)
and *e′*: *exec-move-a P t e2 h* (*stk′*, *loc′*, *pc′*, *xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk″*, *loc″*, *pc″*, *xcp″*)
and *τ′*: ¬ *τmove2* (*compP2 P*) *h stk′ e2 pc′ xcp′*
and *call*: *call1 e2* = *None* ∨ *no-call2 e2 0* ∨ *pc′* = *0* ∧ *stk′* = [] ∧ *loc′* = *xs* ∧ *xcp′* = *None* **by** *auto*
from *e* **have** *τExec-mover-a P t* (*e1 «bop» e2*) *h* ([] @ [*v*], *xs*, *length* (*compE2 e1*) + *0*, *None*) (*stk′* @ [*v*], *loc′*, *length* (*compE2 e1*) + *pc′*, *xcp′*) **by**(*rule BinOp-τExecrI2*)
moreover from *e′* **have** *exec-move-a P t* (*e1 «bop» e2*) *h* (*stk′* @ [*v*], *loc′*, *length* (*compE2 e1*) + *pc′*, *xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk″* @ [*v*], *loc″*, *length* (*compE2 e1*) + *pc″*, *xcp″*)
by(*rule exec-move-BinOpI2*)
moreover from *e′* **have** *pc′* < *length* (*compE2 e2*) **by** *auto*
with *τ′ e′* **have** ¬ *τmove2* (*compP2 P*) *h* (*stk′* @ [*v*]) (*e1 «bop» e2*) (*length* (*compE2 e1*) + *pc′*) *xcp′*
by(*auto simp add*: *τinstr-stk-drop-exec-move τmove2-iff*)
moreover from *red* **have** *call1* (*e1′«bop»e2*) = *call1 e2* **by**(*auto*)
moreover have *no-call2 e2 0* ⟹ *no-call2* (*e1«bop»e2*) (*length* (*compE2 e1*))
by(*auto simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *False call*
by(*auto simp del*: *split-paired-Ex call1.simps calls1.simps*) *blast*
qed
moreover from *bisim′*
have *P*,*e1«bop»e2*,*h′* ⊢ (*Val v«bop»E′*, *xs′*) ↔ ((*stk″* @ [*v*]), *loc″*, *length* (*compE2 e1*) + *pc″*,

xcp'')
by(*rule bisim1-bisims1.bisim1BinOp2*)
moreover from *bisim1* **have** *pc* ≠ *length* (*compE2 e1*) ⟶ *no-call2* (*e1*«*bop*»*e2*) *pc*
by(*auto simp add*: *no-call2-def dest*: *bisim-Val-pc-not-Invoke bisim1-pc-length-compE2*)
ultimately show *?thesis* **using** *τ exec1 s*
apply(*auto simp del*: *split-paired-Ex call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)
apply(*blast intro*: *τExec-mover-trans*|*fastforce elim*!: *τExec-mover-trans simp del*: *split-paired-Ex call1.simps calls1.simps*)+
done
next
case (*Red1BinOp v1 v2 v*)
note [*simp*] = ‹*e1'* = *Val v1*› ‹*e2* = *Val v2*› ‹*ta* = *ε*› ‹*e'* = *Val v*› ‹*h'* = *h*› ‹*xs'* = *xs*›
and *binop* = ‹*binop bop v1 v2* = ⌊*Inl v*⌋›
have *τ*: *τmove1 P h* (*Val v1* «*bop*» *Val v2*) **by**(*rule τmove1BinOp*)
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *xcp*) ([*v1*], *loc*, *length* (*compE2 e1*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*«*bop*»*Val v2*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*v1*], *loc*, *length* (*compE2 e1*), *None*)
by−(*rule BinOp-τExecrI1*)
also have *τmove2* (*compP2 P*) *h* [*v1*] (*e1* «*bop*» *Val v2*) (*length* (*compE2 e1*) + *0*) *None*
by(*rule τmove2BinOp2*)(*rule τmove2Val*)
with *binop* **have** *τExec-mover-a P t* (*e1*«*bop*»*Val v2*) *h* ([*v1*], *loc*, *length* (*compE2 e1*), *None*) ([*v2*, *v1*], *loc*, *Suc* (*length* (*compE2 e1*)), *None*)
by−(*rule τExecr1step*, *auto intro*!: *exec-instr simp add*: *exec-move-def compP2-def*)
also (*rtranclp-trans*) **from** *binop*
have *exec-meth-a* (*compP2 P*) (*compE2* (*e1*«*bop*»*Val v2*)) (*compxE2* (*e1*«*bop*»*Val v2*) *0 0*) *t*
h ([*v2*, *v1*], *loc*, *Suc* (*length* (*compE2 e1*)), *None*) *ε*
h ([*v*], *loc*, *Suc* (*Suc* (*length* (*compE2 e1*))), *None*)
by−(*rule exec-instr*, *auto*)
moreover have *τmove2* (*compP2 P*) *h* [*v2*, *v1*] (*e1*«*bop*»*Val v2*) (*Suc* (*length* (*compE2 e1*))) *None* **by**(*simp add*: *τmove2-iff*)
ultimately have *τExec-mover-a P t* (*e1* «*bop*» *Val v2*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *Suc* (*Suc* (*length* (*compE2 e1*))), *None*)
by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl τexec-moveI simp add*: *exec-move-def compP2-def*)
moreover
have *P*, *e1* «*bop*» *Val v2*, *h* ⊢ (*Val v*, *loc*) ↔ ([*v*], *loc*, *length* (*compE2* (*e1* «*bop*» *Val v2*)), *None*)
by(*rule bisim1Val2*) *simp*
ultimately show *?thesis* **using** *s τ* **by** *auto*
next
case (*Red1BinOpFail v1 v2 a*)
note [*simp*] = ‹*e1'* = *Val v1*› ‹*e2* = *Val v2*› ‹*ta* = *ε*› ‹*e'* = *Throw a*› ‹*h'* = *h*› ‹*xs'* = *xs*›
and *binop* = ‹*binop bop v1 v2* = ⌊*Inr a*⌋›
have *τ*: *τmove1 P h* (*Val v1* «*bop*» *Val v2*) **by**(*rule τmove1BinOp*)
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *xcp*) ([*v1*], *loc*, *length* (*compE2 e1*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*«*bop*»*Val v2*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*v1*], *loc*, *length* (*compE2 e1*), *None*)
by−(*rule BinOp-τExecrI1*)
also have *τmove2* (*compP2 P*) *h* [*v1*] (*e1* «*bop*» *Val v2*) (*length* (*compE2 e1*) + *0*) *None*
by(*rule τmove2BinOp2*)(*rule τmove2Val*)
with *binop* **have** *τExec-mover-a P t* (*e1*«*bop*»*Val v2*) *h* ([*v1*], *loc*, *length* (*compE2 e1*), *None*) ([*v2*, *v1*], *loc*, *Suc* (*length* (*compE2 e1*)), *None*)

by−(*rule* τ*Execr1step*, *auto intro*!: *exec-instr simp add*: *exec-move-def compP2-def*)
also (*rtranclp-trans*) **from** *binop*
have *exec-meth-a* (*compP2 P*) (*compE2* (*e1* «*bop*» *Val v2*)) (*compxE2* (*e1* «*bop*» *Val v2*) *0 0*) *t*
h ([*v2*, *v1*], *loc*, *Suc* (*length* (*compE2 e1*)), *None*) ε
h ([*v2*, *v1*], *loc*, *Suc* (*length* (*compE2 e1*)), ⌊*a*⌋)
by−(*rule exec-instr*, *auto*)
moreover have τ*move2* (*compP2 P*) *h* [*v2*, *v1*] (*e1* «*bop*» *Val v2*) (*Suc* (*length* (*compE2 e1*))) *None* **by**(*simp add*: τ*move2-iff*)
ultimately have τ*Exec-movet-a P t* (*e1* «*bop*» *Val v2*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*v2*, *v1*], *loc*, *Suc* (*length* (*compE2 e1*)), ⌊*a*⌋)
by(*fastforce intro*: *rtranclp-into-tranclp1* τ*exec-moveI simp add*: *exec-move-def compP2-def*)
moreover
have *P*, *e1* «*bop*» *Val v2*, *h* ⊢ (*Throw a*, *loc*) ↔ ([*v2*, *v1*], *loc*, *length* (*compE2 e1*) + *length* (*compE2* (*Val v2*)), ⌊*a*⌋)
by(*rule bisim1BinOpThrow*)
ultimately show *?thesis* **using** *s* τ **by** *auto*
next
case (*Bin1OpThrow1 a*)
note [*simp*] = ‹*e1* ′ = *Throw a*› ‹*ta* = ε› ‹*e*′ = *Throw a*› ‹*h*′ = *h*› ‹*xs*′ = *xs*›
have τ: τ*move1 P h* (*Throw a* «*bop*» *e2*) **by**(*rule* τ*move1BinOpThrow1*)
from *bisim1* **have** *xcp* = ⌊*a*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*a*⌋
with *bisim1* **have** *P*, *e1* «*bop*» *e2*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
by(*auto intro*: *bisim1-bisims1.intros*)
thus *?thesis* **using** τ **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim1* **obtain** *pc*′ **where** τ*Exec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc*′, ⌊*a*⌋)
and *bisim*′: *P*, *e1*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc*′, ⌊*a*⌋)
and [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-Throw-*τ*Exec-mover*)
hence τ*Exec-mover-a P t* (*e1* «*bop*» *e2*) *h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc*′, ⌊*a*⌋)
by−(*rule BinOp-*τ*ExecrI1*)
moreover from *bisim*′
have *P*, *e1* «*bop*» *e2*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc*′, ⌊*a*⌋)
by(*auto intro*: *bisim1-bisims1.bisim1BinOpThrow1*)
ultimately show *?thesis* **using** τ **by** *auto*
qed
next
case (*Bin1OpThrow2 v a*)
note [*simp*] = ‹*e1* ′ = *Val v*› ‹*e2* = *Throw a*› ‹*ta* = ε› ‹*e*′ = *Throw a*› ‹*h*′ = *h*› ‹*xs*′ = *xs*›
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and τ*Exec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 e1*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence τ*Exec-mover-a P t* (*e1* «*bop*» *Throw a*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 e1*), *None*)
by−(*rule BinOp-*τ*ExecrI1*)
also have τ*Exec-mover-a P t* (*e1* «*bop*» *Throw a*) *h* ([*v*], *loc*, *length* (*compE2 e1*), *None*) ([*Addr a*, *v*], *loc*, *Suc* (*length* (*compE2 e1*)), ⌊*a*⌋)
by(*rule* τ*Execr2step*)(*auto simp add*: *exec-move-def exec-meth-instr* τ*move2-iff* τ*move1.simps* τ*moves1.simps*)

also (*rtranclp-trans*)
have *P*,*e1* «*bop*» *Throw a*,*h* ⊢ (*Throw a*, *loc*) ↔ ([*Addr a*] @ [*v*], *loc*, (*length* (*compE2 e1*) + *length* (*compE2* (*addr a*))), ⌊*a*⌋)
by(*rule bisim1BinOpThrow2*[*OF bisim1Throw2*])
moreover have *τmove1 P h* (*e1'* «*bop*» *e2*) **by**(*auto intro*: *τmove1BinOpThrow2*)
ultimately show *?thesis* **using** *s* **by** *auto*
qed
next
case (*bisim1BinOp2 e2 n e2' xs stk loc pc xcp e1 bop v1*)
note *IH2* = *bisim1BinOp2.IH*(*2*)
note *bisim1* = ‹⋀*xs*. *P*,*e1*,*h* ⊢ (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e2'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*Val v1* «*bop*» *e2'*,(*h*, *xs*)⟩ −*ta*→ ⟨*e'*,(*h'*, *xs'*)⟩›
note *bsok* = ‹*bsok* (*e1* «*bop*» *e2*) *n*›
from *red* **show** *?case*
proof *cases*
case (*Bin1OpRed2 E'*)
note [*simp*] = ‹*e'* = *Val v1* «*bop*» *E'*›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e2'*,(*h*, *xs*)⟩ −*ta*→ ⟨*E'*,(*h'*, *xs'*)⟩›
from *IH2*[*OF red*] *bsok* **obtain** *pc''* *stk''* *loc''* *xcp''*
where *bisim'*: *P*,*e2*,*h'* ⊢ (*E'*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
and *exec'*: *?exec ta e2 e2' E' h stk loc pc xcp h' pc'' stk'' loc'' xcp''* **by** *auto*
from *red* **have** *τ*: *τmove1 P h* (*Val v1* «*bop*» *e2'*) = *τmove1 P h e2'* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
have *no-call2 e2 pc* ⟹ *no-call2* (*e1* «*bop*» *e2*) (*length* (*compE2 e1*) + *pc*) **by**(*auto simp add*: *no-call2-def*)
hence *?exec ta* (*e1*«*bop*»*e2*) (*Val v1*«*bop*»*e2'*) (*Val v1*«*bop*»*E'*) *h* (*stk* @ [*v1*]) *loc* (*length* (*compE2 e1*) + *pc*) *xcp h'* (*length* (*compE2 e1*) + *pc''*) (*stk''* @ [*v1*]) *loc''* *xcp''*
using *exec' τ*
apply(*cases τmove1 P h* (*Val v1* «*bop*» *e2'*))
apply(*auto*)
apply(*blast intro*: *BinOp-τExecrI2 BinOp-τExectI2 exec-move-BinOpI2*)
apply(*blast intro*: *BinOp-τExecrI2 BinOp-τExectI2 exec-move-BinOpI2*)
apply(*rule exI conjI BinOp-τExecrI2 exec-move-BinOpI2*|*assumption*)+
apply(*fastforce simp add*: *τinstr-stk-drop-exec-move τmove2-iff split*: *if-split-asm*)
apply(*rule exI conjI BinOp-τExecrI2 exec-move-BinOpI2*|*assumption*)+
apply(*fastforce simp add*: *τinstr-stk-drop-exec-move τmove2-iff split*: *if-split-asm*)
apply(*rule exI conjI BinOp-τExecrI2 exec-move-BinOpI2 rtranclp.rtrancl-refl*|*assumption*)+
apply(*fastforce simp add*: *τinstr-stk-drop-exec-move τmove2-iff split*: *if-split-asm*)+
done
moreover from *bisim'*
have *P*,*e1*«*bop*»*e2*,*h'* ⊢ (*Val v1*«*bop*»*E'*, *xs'*) ↔ (*stk''*@[*v1*], *loc''*, *length* (*compE2 e1*) + *pc''*, *xcp''*)
by(*rule bisim1-bisims1.bisim1BinOp2*)
ultimately show *?thesis* **using** *τ* **by** *auto blast*+
next
case (*Red1BinOp v2 v*)
note [*simp*] = ‹*e2'* = *Val v2*› ‹*ta* = *ε*› ‹*e'* = *Val v*› ‹*h'* = *h*› ‹*xs'* = *xs*›
and *binop* = ‹*binop bop v1 v2* = ⌊*Inl v*⌋›
have *τ*: *τmove1 P h* (*Val v1* «*bop*» *Val v2*) **by**(*rule τmove1BinOp*)
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *xcp*) ([*v2*], *loc*, *length* (*compE2 e2*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*«*bop*»*e2*) *h* (*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) ([*v2*] @ [*v1*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)

by−(*rule BinOp-τExecrI2*)
moreover from *binop*
have *exec-move-a P t* (*e1*«*bop*»*e2*) *h* ([*v2*, *v1*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*) *ε*
h ([*v*], *loc*, *Suc* (*length* (*compE2 e1*) + *length* (*compE2 e2*)), *None*)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto*)
moreover have *τmove2* (*compP2 P*) *h* [*v2*, *v1*] (*e1*«*bop*»*e2*) (*length* (*compE2 e1*) + *length* (*compE2 e2*)) *None*
by(*simp add*: *τmove2-iff*)
ultimately have *τExec-mover-a P t* (*e1* «*bop*» *e2*) *h* (*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) ([*v*], *loc*, *Suc* (*length* (*compE2 e1*) + *length* (*compE2 e2*)), *None*)
by(*auto intro*: *rtranclp.rtrancl-into-rtrancl τexec-moveI simp add*: *compP2-def*)
moreover
have *P*, *e1* «*bop*» *e2*, *h* ⊢ (*Val v*, *loc*) ↔ ([*v*], *loc*, *length* (*compE2* (*e1* «*bop*» *e2*)), *None*)
by(*rule bisim1Val2*) *simp*
ultimately show *?thesis* **using** *s τ* **by**(*auto*)
next
case (*Red1BinOpFail v2 a*)
note [*simp*] = ‹*e2'* = *Val v2*› ‹*ta* = *ε*› ‹*e'* = *Throw a*› ‹*h'* = *h*› ‹*xs'* = *xs*›
and *binop* = ‹*binop bop v1 v2* = ⌊*Inr a*⌋›
have *τ*: *τmove1 P h* (*Val v1* «*bop*» *Val v2*) **by**(*rule τmove1BinOp*)
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *xcp*) ([*v2*], *loc*, *length* (*compE2 e2*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*«*bop*»*e2*) *h* (*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) ([*v2*] @ [*v1*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
by−(*rule BinOp-τExecrI2*)
moreover from *binop*
have *exec-move-a P t* (*e1*«*bop*»*e2*) *h* ([*v2*, *v1*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*) *ε*
h ([*v2*, *v1*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), ⌊*a*⌋)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto*)
moreover have *τmove2* (*compP2 P*) *h* [*v2*, *v1*] (*e1*«*bop*»*e2*) (*length* (*compE2 e1*) + *length* (*compE2 e2*)) *None*
by(*simp add*: *τmove2-iff*)
ultimately have *τExec-movet-a P t* (*e1* «*bop*» *e2*) *h* (*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) ([*v2*, *v1*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), ⌊*a*⌋)
by(*auto intro*: *rtranclp-into-tranclp1 τexec-moveI simp add*: *compP2-def*)
moreover
have *P*, *e1* «*bop*» *e2*, *h* ⊢ (*Throw a*, *loc*) ↔ ([*v2*, *v1*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), ⌊*a*⌋)
by(*rule bisim1BinOpThrow*)
ultimately show *?thesis* **using** *s τ* **by**(*auto*)
next
case (*Bin1OpThrow2 a*)
note [*simp*] = ‹*e2'* = *Throw a*› ‹*ta* = *ε*› ‹*h'* = *h*› ‹*xs'* = *xs*› ‹*e'* = *Throw a*›
have *τ*: *τmove1 P h* (*Val v1* «*bop*» *Throw a*) **by**(*rule τmove1BinOpThrow2*)
from *bisim2* **have** *xcp* = ⌊*a*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*a*⌋
with *bisim2*
have *P*, *e1*«*bop*»*e2*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*)
by(*auto intro*: *bisim1BinOpThrow2*)

```
    thus ?thesis using τ by(fastforce)
   next
    assume [simp]: xcp = None
    with bisim2 obtain pc'
     where τExec-mover-a P t e2 h (stk, loc, pc, None) ([Addr a], loc, pc', ⌊a⌋)
     and bisim': P, e2, h ⊢ (Throw a, xs) ↔ ([Addr a], loc, pc', ⌊a⌋) and [simp]: xs = loc
     by(auto dest: bisim1-Throw-τExec-mover)
   hence τExec-mover-a P t (e1 «bop» e2) h (stk @ [v1], loc, length (compE2 e1) + pc, None) ([Addr
a] @ [v1], loc, length (compE2 e1) + pc', ⌊a⌋)
     by−(rule BinOp-τExecrI2)
    moreover from bisim'
    have P, e1 «bop» e2, h ⊢ (Throw a, xs) ↔ ([Addr a]@[v1], loc, length (compE2 e1) + pc', ⌊a⌋)
     by−(rule bisim1BinOpThrow2, auto)
    ultimately show ?thesis using τ by auto
   qed
  qed auto
next
  case bisim1BinOpThrow1 thus ?case by fastforce
next
  case bisim1BinOpThrow2 thus ?case by fastforce
next
  case bisim1BinOpThrow thus ?case by fastforce
next
  case (bisim1LAss1 E n e xs stk loc pc xcp V)
  note IH = bisim1LAss1.IH(2)
  note bisim = ‹P,E,h ⊢ (e, xs) ↔ (stk, loc, pc, xcp)›
  note red = ‹True,P,t ⊢1 ⟨V:=e,(h, xs)⟩ −ta→ ⟨e',(h', xs')⟩›
  note bsok = ‹bsok (V:=E) n›
  from red show ?case
  proof cases
   case (LAss1Red ee')
   note [simp] = ‹e' = V := ee'›
    and red = ‹True,P,t ⊢1 ⟨e,(h, xs)⟩ −ta→ ⟨ee', (h', xs')⟩›
  from red have τmove1 P h (V:=e) = τmove1 P h e by(auto simp add: τmove1.simps τmoves1.simps)
   moreover from red have call1 (V:=e) = call1 e by auto
   moreover from IH[OF red] bsok
   obtain pc'' stk'' loc'' xcp'' where bisim: P,E,h' ⊢ (ee', xs') ↔ (stk'', loc'', pc'', xcp'')
    and redo: ?exec ta E e ee' h stk loc pc xcp h' pc'' stk'' loc'' xcp'' by auto
   from bisim
   have P,V:=E,h' ⊢ (V:=ee', xs') ↔ (stk'', loc'', pc'', xcp'')
    by(rule bisim1-bisims1.bisim1LAss1)
   moreover {
    assume no-call2 E pc
    hence no-call2 (V:=E) pc by(auto simp add: no-call2-def) }
   ultimately show ?thesis using redo
      by(auto simp del: call1.simps calls1.simps split: if-split-asm split del: if-split)(blast intro:
LAss-τExecrI LAss-τExectI exec-move-LAssI)+
  next
   case (Red1LAss v)
   note [simp] = ‹e = Val v› ‹ta = ε› ‹e' = unit› ‹h' = h› ‹xs' = xs[V := v]›
    and V = ‹V < length xs›
   from bisim have s: xcp = None xs = loc by(auto dest: bisim-Val-loc-eq-xcp-None)
   from bisim have τExec-mover-a P t E h (stk, loc, pc, xcp) ([v], loc, length (compE2 E), None)
    by(auto dest: bisim1Val2D1)
```

```
    hence τExec-mover-a P t (V:=E) h (stk, loc, pc, xcp) ([v], loc, length (compE2 E), None)
      by(rule LAss-τExecrI)
    moreover have exec-move-a P t (V:=E) h ([v], loc, length (compE2 E), None) ε h ([], loc[V :=
v], Suc (length (compE2 E)), None)
      using V s by(auto intro: exec-instr simp add: exec-move-def)
    moreover have τmove2 (compP2 P) h [v] (V := E) (length (compE2 E)) None by(simp add:
τmove2-iff)
    ultimately have τExec-mover-a P t (V:=E) h (stk, loc, pc, xcp) ([], loc[V := v], Suc (length
(compE2 E)), None)
      by(auto intro: rtranclp.rtrancl-into-rtrancl τexec-moveI simp add: compP2-def)
    moreover have τmove1 P h (V := Val v) by(rule τmove1LAssRed)
    moreover have P, V:=E, h ⊢ (unit, loc[V := v]) ↔ ([], loc[V := v], Suc (length (compE2 E)),
None)
      by(rule bisim1LAss2)
    ultimately show ?thesis using s by auto
  next
    case (LAss1Throw a)
    note [simp] = ‹e = Throw a› ‹h' = h› ‹xs' = xs› ‹ta = ε› ‹e' = Throw a›
    have τ: τmove1 P h (V:=e) by(auto intro: τmove1LAssThrow)
    from bisim have xcp = ⌊a⌋ ∨ xcp = None by(auto dest: bisim1-ThrowD)
    thus ?thesis
    proof
      assume [simp]: xcp = ⌊a⌋
    with bisim have P, V:=E, h ⊢ (Throw a, xs) ↔ (stk, loc, pc, xcp) by(auto intro: bisim1LAssThrow)
      thus ?thesis using τ by(fastforce)
    next
      assume [simp]: xcp = None
      with bisim obtain pc'
        where τExec-mover-a P t E h (stk, loc, pc, None) ([Addr a], loc, pc', ⌊a⌋)
        and bisim': P, E, h ⊢ (Throw a, xs) ↔ ([Addr a], loc, pc', ⌊a⌋) and [simp]: xs = loc
        by(auto dest: bisim1-Throw-τExec-mover)
      hence τExec-mover-a P t (V:=E) h (stk, loc, pc, None) ([Addr a], loc, pc', ⌊a⌋)
        by−(rule LAss-τExecrI)
      moreover from bisim' have P, V:=E, h ⊢ (Throw a, xs) ↔ ([Addr a], loc, pc', ⌊a⌋)
        by−(rule bisim1LAssThrow, auto)
      ultimately show ?thesis using τ by auto
    qed
  qed
next
  case bisim1LAss2 thus ?case by fastforce
next
  case bisim1LAssThrow thus ?case by fastforce
next
  case (bisim1AAcc1 a n a' xs stk loc pc xcp i)
  note IH1 = bisim1AAcc1.IH(2)
  note IH2 = bisim1AAcc1.IH(4)
  note bisim1 = ‹P,a,h ⊢ (a', xs) ↔ (stk, loc, pc, xcp)›
  note bisim2 = ‹⋀xs. P,i,h ⊢ (i, xs) ↔ ([], xs, 0, None)›
  note bsok = ‹bsok (a⌊i⌉) n›
  from ‹True,P,t ⊢1 ⟨a'⌊i⌉,(h, xs)⟩ −ta→ ⟨e',(h', xs')⟩› show ?case
  proof cases
    case (AAcc1Red1 E')
    note [simp] = ‹e' = E'⌊i⌉›
      and red = ‹True,P,t ⊢1 ⟨a',(h, xs)⟩ −ta→ ⟨E',(h', xs')⟩›
```

from *red* **have** $\tau move1\ P\ h\ (a'\lfloor i\rceil) = \tau move1\ P\ h\ a'$ **by**(*auto simp add*: $\tau move1.simps\ \tau moves1.simps$)
moreover from *red* **have** *call1* $(a'\lfloor i\rceil) =$ *call1* a' **by** *auto*
moreover from *IH1*[*OF red*] *bsok*
obtain $pc''\ stk''\ loc''\ xcp''$ **where** *bisim*: $P,a,h' \vdash (E', xs') \leftrightarrow (stk'', loc'', pc'', xcp'')$
and *redo*: *?exec ta a a' E' h stk loc pc xcp h' pc'' stk'' loc'' xcp''* **by** *auto*
from *bisim* **have** $P,a\lfloor i\rceil,h' \vdash (E'\lfloor i\rceil, xs') \leftrightarrow (stk'', loc'', pc'', xcp'')$
by(*rule bisim1-bisims1.bisim1AAcc1*)
moreover {
assume *no-call2 a pc*
hence *no-call2* $(a\lfloor i\rceil)$ *pc* $\vee$ $pc =$ *length* (*compE2 a*) **by**(*auto simp add*: *no-call2-def*) **}**
ultimately show *?thesis* **using** *redo*
by(*auto simp del*: *call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)(*blast intro*: *AAcc-τExecrI1 AAcc-τExectI1 exec-move-AAccI1*)+
next
case (*AAcc1Red2 E' v*)
note [*simp*] = ‹$a' =$ *Val v*› ‹$e' =$ *Val* $v\lfloor E'\rceil$›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*i*,(*h*, *xs*)⟩ −*ta*→ ⟨*E'*,(*h'*, *xs'*)⟩›
from *red* **have** *τ*: $\tau move1\ P\ h\ (Val\ v\lfloor i\rceil) = \tau move1\ P\ h\ i$ **by**(*auto simp add*: $\tau move1.simps\ \tau moves1.simps$)
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *exec1*: $\tau Exec\text{-}mover\text{-}a\ P\ t\ a\ h$ (*stk*, *loc*, *pc*, *None*) ([*v*], *xs*, *length* (*compE2 a*), *None*)
by(*auto dest*: *bisim1Val2D1*)
from *exec1* **have** $\tau Exec\text{-}mover\text{-}a\ P\ t\ (a\lfloor i\rceil)\ h$ (*stk*, *loc*, *pc*, *None*) ([*v*], *xs*, *length* (*compE2 a*), *None*)
by(*rule AAcc-τExecrI1*)
moreover
from *IH2*[*OF red*] *bsok* **obtain** $pc''\ stk''\ loc''\ xcp''$
where *bisim'*: $P,i,h' \vdash (E', xs') \leftrightarrow (stk'', loc'', pc'', xcp'')$
and *exec'*: *?exec ta i i E' h* [] *xs 0 None h' pc'' stk'' loc'' xcp''* **by** *auto*
have *?exec ta* $(a\lfloor i\rceil)$ $(Val\ v\lfloor i\rceil)$ $(Val\ v\lfloor E'\rceil)$ *h* ([] @ [*v*]) *xs* (*length* (*compE2 a*) + *0*) *None h'* (*length* (*compE2 a*) + pc'') $(stk'' @ [v])$ $loc''\ xcp''$
proof(*cases* $\tau move1\ P\ h\ (Val\ v\lfloor i\rceil)$)
case *True*
with *exec'* *τ* **have** [*simp*]: $h = h'$ **and** *e*: *sim-move i E' P t i h* ([], *xs*, *0*, *None*) (*stk''*, *loc''*, *pc''*, *xcp''*) **by** *auto*
from *e* **have** *sim-move* $(a\lfloor i\rceil)$ $(a\lfloor E'\rceil)$ *P t* $(a\lfloor i\rceil)$ *h* ([] @ [*v*], *xs*, *length* (*compE2 a*) + *0*, *None*) (*stk''* @ [*v*], *loc''*, *length* (*compE2 a*) + *pc''*, *xcp''*)
by(*fastforce dest*: *AAcc-τExecrI2 AAcc-τExectI2*)
with *True* **show** *?thesis* **by** *auto*
next
case *False*
with *exec'* *τ* **obtain** *pc' stk' loc' xcp'*
where *e*: $\tau Exec\text{-}mover\text{-}a\ P\ t\ i\ h$ ([], *xs*, *0*, *None*) (*stk'*, *loc'*, *pc'*, *xcp'*)
and *e'*: *exec-move-a P t i h* (*stk'*, *loc'*, *pc'*, *xcp'*) (*extTA2JVM* (*compP2 P*) *ta*) *h'* (*stk''*, *loc''*, *pc''*, *xcp''*)
and *τ'*: ¬ $\tau move2\ (compP2\ P)\ h\ stk'\ i\ pc'\ xcp'$
and *call*: *call1 i* = *None* ∨ *no-call2 i 0* ∨ $pc' = 0 \wedge stk' = [] \wedge loc' = xs \wedge xcp' = None$ **by** *auto*
from *e* **have** $\tau Exec\text{-}mover\text{-}a\ P\ t\ (a\lfloor i\rceil)\ h$ ([] @ [*v*], *xs*, *length* (*compE2 a*) + *0*, *None*) (*stk'* @ [*v*], *loc'*, *length* (*compE2 a*) + *pc'*, *xcp'*)
by(*rule AAcc-τExecrI2*)
moreover from *e'* **have** *exec-move-a P t* $(a\lfloor i\rceil)$ *h* (*stk'* @ [*v*], *loc'*, *length* (*compE2 a*) + *pc'*, *xcp'*) (*extTA2JVM* (*compP2 P*) *ta*) *h'* (*stk''* @ [*v*], *loc''*, *length* (*compE2 a*) + *pc''*, *xcp''*)
by(*rule exec-move-AAccI2*)

moreover from $e'\ \tau'$ **have** $\neg\ \tau move2\ (compP2\ P)\ h\ (stk'\ @\ [v])\ (a\lfloor i\rceil)\ (length\ (compE2\ a)\ +\ pc')\ xcp'$
by(*auto simp add*: $\tau instr\text{-}stk\text{-}drop\text{-}exec\text{-}move\ \tau move2\text{-}iff$)
moreover have $call1\ (a'\lfloor i\rceil) = call1\ i$ **by** *simp*
moreover have $no\text{-}call2\ i\ 0 \Longrightarrow no\text{-}call2\ (a\lfloor i\rceil)\ (length\ (compE2\ a))$
by(*auto simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *False call*
by(*auto simp del*: *split-paired-Ex call1.simps calls1.simps*) *blast*
qed
moreover from $bisim'$
have $P,a\lfloor i\rceil,h' \vdash (Val\ v\lfloor E'\rceil,\ xs') \leftrightarrow ((stk''\ @\ [v]),\ loc'',\ length\ (compE2\ a)\ +\ pc'',\ xcp'')$
by(*rule bisim1-bisims1.bisim1AAcc2*)
moreover from *bisim1* **have** $pc \neq length\ (compE2\ a) \longrightarrow no\text{-}call2\ (a\lfloor i\rceil)\ pc$
by(*auto simp add*: *no-call2-def dest*: *bisim-Val-pc-not-Invoke bisim1-pc-length-compE2*)
ultimately show *?thesis* **using** $\tau\ exec1\ s$
apply(*auto simp del*: *split-paired-Ex call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)
apply(*blast intro*: $\tau Exec\text{-}mover\text{-}trans|fastforce\ elim!$: $\tau Exec\text{-}mover\text{-}trans\ simp\ del$: *split-paired-Ex call1.simps calls1.simps*)+
done
next
case (*Red1AAcc A U len I v*)
hence [*simp*]: $a' = addr\ A\ e' = Val\ v\ i = Val\ (Intg\ I)\ h' = h\ xs' = xs$
$ta = \{\!|ReadMem\ A\ (ACell\ (nat\ (sint\ I)))\ v|\!\}$
and *hA*: *typeof-addr h A* $= \lfloor Array\text{-}type\ U\ len\rfloor$ **and** I: $0 <=s\ I\ sint\ I < int\ len$
and *read*: *heap-read h A* (*ACell* (*nat* (*sint I*))) *v* **by** *auto*
have τ: $\neg\ \tau move1\ P\ h\ (addr\ A\lfloor Val\ (Intg\ I)\rceil)$ **by**(*auto simp add*: $\tau move1.simps\ \tau moves1.simps$)
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and $\tau Exec\text{-}mover\text{-}a\ P\ t\ a\ h\ (stk,\ loc,\ pc,\ xcp)\ ([Addr\ A],\ loc,\ length\ (compE2\ a),\ None)$
by(*auto dest*: *bisim1Val2D1*)
hence $\tau Exec\text{-}mover\text{-}a\ P\ t\ (a\lfloor Val\ (Intg\ I)\rceil)\ h\ (stk,\ loc,\ pc,\ xcp)\ ([Addr\ A],\ loc,\ length\ (compE2\ a),\ None)$
by−(*rule AAcc-*τ*ExecrI1*)
also have $\tau move2\ (compP2\ P)\ h\ [Addr\ A]\ (a\lfloor Val\ (Intg\ I)\rceil)\ (length\ (compE2\ a)\ +\ 0)\ None$
by(*rule* $\tau move2AAcc2$)(*rule* $\tau move2Val$)
hence $\tau Exec\text{-}mover\text{-}a\ P\ t\ (a\lfloor Val\ (Intg\ I)\rceil)\ h\ ([Addr\ A],\ loc,\ length\ (compE2\ a),\ None)\ ([Intg\ I,\ Addr\ A],\ loc,\ Suc\ (length\ (compE2\ a)),\ None)$
by−(*rule* $\tau Execr1step$, *auto intro*!: *exec-instr simp add*: *exec-move-def compP2-def*)
also (*rtranclp-trans*) **from** *hA I read*
have $exec\text{-}move\text{-}a\ P\ t\ (a\lfloor Val\ (Intg\ I)\rceil)\ h\ ([Intg\ I,\ Addr\ A],\ loc,\ Suc\ (length\ (compE2\ a)),\ None)$
$\{\!|ReadMem\ A\ (ACell\ (nat\ (sint\ I)))\ v|\!\}$
$h\ ([v],\ loc,\ Suc\ (Suc\ (length\ (compE2\ a))),\ None)$
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *is-Ref-def*)
moreover have $\tau move2\ (compP2\ P)\ h\ [Intg\ I,\ Addr\ A]\ (a\lfloor Val\ (Intg\ I)\rceil)\ (Suc\ (length\ (compE2\ a)))\ None \Longrightarrow False$
by(*simp add*: $\tau move2\text{-}iff$)
moreover
have $P,\ a\lfloor Val\ (Intg\ I)\rceil,\ h \vdash (Val\ v,\ loc) \leftrightarrow ([v],\ loc,\ length\ (compE2\ (a\lfloor Val\ (Intg\ I)\rceil)),\ None)$
by(*rule bisim1Val2*) *simp*
ultimately show *?thesis* **using** $s\ \tau$
by(*auto simp add*: *ta-upd-simps*) *blast*
next
case (*Red1AAccNull v*)
note [*simp*] = ‹$a' = null$› ‹$i = Val\ v$› ‹$ta = \varepsilon$› ‹$e' = THROW\ NullPointer$› ‹$h' = h$› ‹$xs' = xs$›
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*

and *τExec-mover-a P t a h* (*stk*, *loc*, *pc*, *xcp*) ([*Null*], *loc*, *length* (*compE2 a*), *None*)
by(*auto dest*: *bisim1Val2D1 intro*: *AAcc-τExecrI1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉) *h* (*stk*, *loc*, *pc*, *xcp*) ([*Null*], *loc*, *length* (*compE2 a*), *None*)
by−(*rule AAcc-τExecrI1*)
also from *bisim2*[*of loc*] **have** *τExec-mover-a P t i h* ([], *loc*, *0*, *None*) ([*v*], *loc*, *length* (*compE2 i*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉) *h* ([] @ [*Null*], *loc*, *length* (*compE2 a*) + *0*, *None*) ([*v*] @ [*Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*)
by(*rule AAcc-τExecrI2*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉) *h* ([*Null*], *loc*, *length* (*compE2 a*), *None*) ([*v*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*) **by** *simp*
also (*rtranclp-trans*) **have** *exec-move-a P t* (*a*⌊*i*⌉) *h* ([*v*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*) *ε h* ([*v*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [*v*, *Null*] (*a*⌊*i*⌉) (*length* (*compE2 a*) + *length* (*compE2 i*)) *None*
by(*simp add*: *τmove2-iff*)
moreover have ¬ *τmove1 P h* (*a*′⌊*i*⌉) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
moreover
have *P*,*a*⌊*i*⌉,*h* ⊢ (*THROW NullPointer*, *xs*) ↔ ([*v*, *Null*], *xs*, *length* (*compE2 a*) + *length* (*compE2 i*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule bisim1-bisims1.bisim1AAccFail*)
ultimately show *?thesis* **using** *s* **by** *auto blast*
next
case (*Red1AAccBounds A U len I*)
hence [*simp*]: *a*′ = *addr A e*′ = *THROW ArrayIndexOutOfBounds i* = *Val* (*Intg I*)
ta = *ε h*′ = *h xs*′ = *xs*
and *hA*: *typeof-addr h A* = ⌊*Array-type U len*⌋ **and** *I*: *I* <*s* *0* ∨ *int len* ≤ *sint I* **by** *auto*
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t a h* (*stk*, *loc*, *pc*, *xcp*) ([*Addr A*], *loc*, *length* (*compE2 a*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉) *h* (*stk*, *loc*, *pc*, *xcp*) ([*Addr A*], *loc*, *length* (*compE2 a*), *None*)
by−(*rule AAcc-τExecrI1*)
also from *bisim2*[*of loc*] **have** *τExec-mover-a P t i h* ([], *loc*, *0*, *None*) ([*Intg I*], *loc*, *length* (*compE2 i*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉) *h* ([] @ [*Addr A*], *loc*, *length* (*compE2 a*) + *0*, *None*) ([*Intg I*] @ [*Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*)
by(*rule AAcc-τExecrI2*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉) *h* ([*Addr A*], *loc*, *length* (*compE2 a*), *None*) ([*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*) **by** *simp*
also (*rtranclp-trans*) **from** *I hA*
have *exec-move-a P t* (*a*⌊*i*⌉) *h* ([*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*) *ε h* ([*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), ⌊*addr-of-sys-xcpt ArrayIndexOutOfBounds*⌋)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Intg I*, *Addr A*] (*a*⌊*i*⌉) (*length* (*compE2 a*) + *length* (*compE2 i*)) *None*
by(*simp add*: *τmove2-iff*)
moreover have ¬ *τmove1 P h* (*a*′⌊*i*⌉) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
moreover
have *P*,*a*⌊*i*⌉,*h* ⊢ (*THROW ArrayIndexOutOfBounds*, *xs*) ↔ ([*Intg I*, *Addr A*], *xs*, *length* (*compE2*

```
a) + length (compE2 i), ⌊addr-of-sys-xcpt ArrayIndexOutOfBounds⌋)
    by(rule bisim1-bisims1.bisim1AAccFail)
  ultimately show ?thesis using s by auto blast
 next
  case (AAcc1Throw1 A)
  note [simp] = ‹a' = Throw A› ‹ta = ε› ‹e' = Throw A› ‹h' = h› ‹xs' = xs›
  have τ: τmove1 P h (Throw A⌊i⌉) by(rule τmove1AAccThrow1)
  from bisim1 have xcp = ⌊A⌋ ∨ xcp = None by(auto dest: bisim1-ThrowD)
  thus ?thesis
  proof
   assume [simp]: xcp = ⌊A⌋
   with bisim1 have P, a⌊i⌉, h ⊢ (Throw A, xs) ↔ (stk, loc, pc, xcp)
    by(auto intro: bisim1-bisims1.intros)
   thus ?thesis using τ by(fastforce)
  next
   assume [simp]: xcp = None
   with bisim1 obtain pc' where τExec-mover-a P t a h (stk, loc, pc, None) ([Addr A], loc, pc',
⌊A⌋)
    and bisim': P, a, h ⊢ (Throw A, xs) ↔ ([Addr A], loc, pc', ⌊A⌋)
    and [simp]: xs = loc
    by(auto dest: bisim1-Throw-τExec-mover)
   hence τExec-mover-a P t (a⌊i⌉) h (stk, loc, pc, None) ([Addr A], loc, pc', ⌊A⌋)
    by−(rule AAcc-τExecrI1)
   moreover from bisim'
   have P, a⌊i⌉, h ⊢ (Throw A, xs) ↔ ([Addr A], loc, pc', ⌊A⌋)
    by(auto intro: bisim1-bisims1.bisim1AAccThrow1)
   ultimately show ?thesis using τ by auto
  qed
 next
  case (AAcc1Throw2 v ad)
  note [simp] = ‹a' = Val v› ‹i = Throw ad› ‹ta = ε› ‹e' = Throw ad› ‹h' = h› ‹xs' = xs›
  from bisim1 have s: xcp = None xs = loc
   and τExec-mover-a P t a h (stk, loc, pc, xcp) ([v], loc, length (compE2 a), None)
   by(auto dest: bisim1Val2D1)
  hence τExec-mover-a P t (a⌊Throw ad⌉) h (stk, loc, pc, xcp) ([v], loc, length (compE2 a), None)
   by−(rule AAcc-τExecrI1)
  also have τExec-mover-a P t (a⌊Throw ad⌉) h ([v], loc, length (compE2 a), None) ([Addr ad, v],
loc, Suc (length (compE2 a)), ⌊ad⌋)
   by(rule τExecr2step)(auto simp add: exec-move-def exec-meth-instr τmove2-iff τmove1.simps
τmoves1.simps)
  also (rtranclp-trans)
  have P,a⌊Throw ad⌉,h ⊢ (Throw ad, loc) ↔ ([Addr ad] @ [v], loc, (length (compE2 a) + length
(compE2 (addr ad))), ⌊ad⌋)
   by(rule bisim1AAccThrow2[OF bisim1Throw2])
  moreover have τmove1 P h (a'⌊Throw ad⌉) by(auto intro: τmove1AAccThrow2)
  ultimately show ?thesis using s by auto
 qed
next
 case (bisim1AAcc2 i n i' xs stk loc pc xcp a v1)
 note IH2 = bisim1AAcc2.IH(2)
 note bisim1 = ‹⋀xs. P,a,h ⊢ (a, xs) ↔ ([], xs, 0, None)›
 note bisim2 = ‹P,i,h ⊢ (i', xs) ↔ (stk, loc, pc, xcp)›
 note red = ‹True,P,t ⊢1 ⟨Val v1⌊i'⌉,(h, xs)⟩ −ta→ ⟨e',(h', xs')⟩›
 note bsok = ‹bsok (a⌊i⌉) n›
```

from *red* **show** *?case*
proof *cases*
case (*AAcc1Red2 E'*)
note [*simp*] = ‹*e'* = *Val v1*⌊*E'*⌉›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*i'*,(*h*, *xs*)⟩ −*ta*→ ⟨*E'*,(*h'*, *xs'*)⟩›
from *IH2*[*OF red*] *bsok* **obtain** *pc''* *stk''* *loc''* *xcp''*
where *bisim'*: *P*,*i*,*h'* ⊢ (*E'*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
and *exec'*: *?exec ta i i' E' h stk loc pc xcp h' pc'' stk'' loc'' xcp''* **by** *auto*
from *red* **have** *τ*: *τmove1 P h* (*Val v1*⌊*i'*⌉) = *τmove1 P h i'* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
have *no-call2 i pc* ⟹ *no-call2* (*a*⌊*i*⌉) (*length* (*compE2 a*) + *pc*) **by**(*auto simp add*: *no-call2-def*)
hence *?exec ta* (*a*⌊*i*⌉) (*Val v1*⌊*i'*⌉) (*Val v1*⌊*E'*⌉) *h* (*stk* @ [*v1*]) *loc* (*length* (*compE2 a*) + *pc*) *xcp h'* (*length* (*compE2 a*) + *pc''*) (*stk''* @ [*v1*]) *loc''* *xcp''*
using *exec' τ*
apply(*cases τmove1 P h* (*Val v1*⌊*i'*⌉))
apply(*auto*)
apply(*blast intro*: *AAcc-τExecrI2 AAcc-τExectI2 exec-move-AAccI2*)
apply(*blast intro*: *AAcc-τExecrI2 AAcc-τExectI2 exec-move-AAccI2*)
apply(*rule exI conjI AAcc-τExecrI2 exec-move-AAccI2*|*assumption*)+
apply(*fastforce simp add*: *τinstr-stk-drop-exec-move τmove2-iff split*: *if-split-asm*)
apply(*rule exI conjI AAcc-τExecrI2 exec-move-AAccI2*|*assumption*)+
apply(*fastforce simp add*: *τinstr-stk-drop-exec-move τmove2-iff split*: *if-split-asm*)
apply(*rule exI conjI AAcc-τExecrI2 exec-move-AAccI2 rtranclp.rtrancl-refl*|*assumption*)+
apply(*fastforce simp add*: *τinstr-stk-drop-exec-move τmove2-iff split*: *if-split-asm*)+
done
moreover from *bisim'*
have *P*,*a*⌊*i*⌉,*h'* ⊢ (*Val v1*⌊*E'*⌉, *xs'*) ↔ (*stk''*@[*v1*], *loc''*, *length* (*compE2 a*) + *pc''*, *xcp''*)
by(*rule bisim1-bisims1.bisim1AAcc2*)
ultimately show *?thesis* **using** *τ* **by** *auto blast*+
next
case (*Red1AAcc A U len I v*)
hence [*simp*]: *v1* = *Addr A e'* = *Val v i'* = *Val* (*Intg I*)
ta = {|*ReadMem A* (*ACell* (*nat* (*sint I*))) *v*|} *h'* = *h xs'* = *xs*
and *hA*: *typeof-addr h A* = ⌊*Array-type U len*⌋ **and** *I*: *0* <=*s I sint I* < *int len*
and *read*: *heap-read h A* (*ACell* (*nat* (*sint I*))) *v* **by** *auto*
have *τ*: ¬ *τmove1 P h* (*addr A*⌊*Val* (*Intg I*)⌉) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t i h* (*stk*, *loc*, *pc*, *xcp*) ([*Intg I*], *loc*, *length* (*compE2 i*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉) *h* (*stk* @ [*Addr A*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) ([*Intg I*] @ [*Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*)
by−(*rule AAcc-τExecrI2*)
moreover from *hA I read*
have *exec-move-a P t* (*a*⌊*i*⌉) *h* ([*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*)
{|*ReadMem A* (*ACell* (*nat* (*sint I*))) *v*|}
h ([*v*], *loc*, *Suc* (*length* (*compE2 a*) + *length* (*compE2 i*)), *None*)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *is-Ref-def*)
moreover have *τmove2* (*compP2 P*) *h* [*Intg I*, *Addr A*] (*a*⌊*i*⌉) (*length* (*compE2 a*) + *length* (*compE2 i*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *a*⌊*i*⌉, *h* ⊢ (*Val v*, *loc*) ↔ ([*v*], *loc*, *length* (*compE2* (*a*⌊*i*⌉)), *None*)
by(*rule bisim1Val2*) *simp*

ultimately show *?thesis* **using** $s\ \tau$ **by**(*auto simp add*: *ta-upd-simps*) *blast*
next
case (*Red1AAccNull v*)
note [*simp*] = ‹*v1* = *Null*› ‹*i'* = *Val v*› ‹*ta* = ε› ‹*e'* = *THROW NullPointer*› ‹*h'* = *h*› ‹*xs'* = *xs*›
from *bisim2* **have** [*simp*]: *xcp* = *None xs* = *loc*
and τ*Exec-mover-a P t i h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 i*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence τ*Exec-mover-a P t* (*a*⌊*i*⌉) *h* (*stk* @ [*Null*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) ([*v*] @ [*Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*)
by−(*rule AAcc-*τ*ExecrI2*)
moreover have *exec-move-a P t* (*a*⌊*i*⌉) *h* ([*v*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*) ε *h* ([*v*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto*)
moreover have ¬ τ*move2* (*compP2 P*) *h* [*v*, *Null*] (*a*⌊*i*⌉) (*length* (*compE2 a*) + *length* (*compE2 i*)) *None*
by(*simp add*: τ*move2-iff*)
moreover have ¬ τ*move1 P h* (*null*⌊*i'*⌉) **by**(*auto simp add*: τ*move1.simps* τ*moves1.simps*)
moreover
have *P*,*a*⌊*i*⌉,*h* ⊢ (*THROW NullPointer*, *xs*) ↔ ([*v*, *Null*], *xs*, *length* (*compE2 a*) + *length* (*compE2 i*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule bisim1-bisims1.bisim1AAccFail*)
ultimately show *?thesis* **by** *auto blast*
next
case (*Red1AAccBounds A U len I*)
hence [*simp*]: *v1* = *Addr A e'* = *THROW ArrayIndexOutOfBounds i'* = *Val* (*Intg I*)
ta = ε *h'* = *h xs'* = *xs*
and *hA*: *typeof-addr h A* = ⌊*Array-type U len*⌋ **and** *I*: *I* <*s* 0 ∨ *int len* ≤ *sint I* **by** *auto*
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and τ*Exec-mover-a P t i h* (*stk*, *loc*, *pc*, *xcp*) ([*Intg I*], *loc*, *length* (*compE2 i*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence τ*Exec-mover-a P t* (*a*⌊*i*⌉) *h* (*stk* @ [*Addr A*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) ([*Intg I*] @ [*Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*)
by−(*rule AAcc-*τ*ExecrI2*)
moreover from *I hA*
have *exec-move-a P t* (*a*⌊*i*⌉) *h* ([*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*) ε *h* ([*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), ⌊*addr-of-sys-xcpt ArrayIndexOutOfBounds*⌋)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *is-Ref-def*)
moreover have ¬ τ*move2* (*compP2 P*) *h* [*Intg I*, *Addr A*] (*a*⌊*i*⌉) (*length* (*compE2 a*) + *length* (*compE2 i*)) *None*
by(*simp add*: τ*move2-iff*)
moreover have ¬ τ*move1 P h* (*addr A*⌊*i'*⌉) **by**(*auto simp add*: τ*move1.simps* τ*moves1.simps*)
moreover
have *P*,*a*⌊*i*⌉,*h* ⊢ (*THROW ArrayIndexOutOfBounds*, *xs*) ↔ ([*Intg I*, *Addr A*], *xs*, *length* (*compE2 a*) + *length* (*compE2 i*), ⌊*addr-of-sys-xcpt ArrayIndexOutOfBounds*⌋)
by(*rule bisim1-bisims1.bisim1AAccFail*)
ultimately show *?thesis* **using** *s* **by** *auto blast*
next
case (*AAcc1Throw2 A*)
note [*simp*] = ‹*i'* = *Throw A*› ‹*ta* = ε› ‹*e'* = *Throw A*› ‹*h'* = *h*› ‹*xs'* = *xs*›
have τ: τ*move1 P h* (*Val v1*⌊*Throw A*⌉) **by**(*rule* τ*move1AAccThrow2*)
from *bisim2* **have** *xcp* = ⌊*A*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof

```
    assume [simp]: xcp = ⌊A⌋
    with bisim2
    have P, a⌊i⌋, h ⊢ (Throw A, xs) ↔ (stk @ [v1], loc, length (compE2 a) + pc, xcp)
      by(auto intro: bisim1-bisims1.intros)
    thus ?thesis using τ by(auto)
  next
    assume [simp]: xcp = None
    with bisim2 obtain pc' where τExec-mover-a P t i h (stk, loc, pc, None) ([Addr A], loc, pc', ⌊A⌋)
      and bisim': P, i, h ⊢ (Throw A, xs) ↔ ([Addr A], loc, pc', ⌊A⌋)
      and [simp]: xs = loc
      by(auto dest: bisim1-Throw-τExec-mover)
    hence τExec-mover-a P t (a⌊i⌋) h (stk @ [v1], loc, length (compE2 a) + pc, None) ([Addr A] @ [v1], loc, length (compE2 a) + pc', ⌊A⌋)
      by−(rule AAcc-τExecrI2)
    moreover from bisim'
    have P, a⌊i⌋, h ⊢ (Throw A, xs) ↔ ([Addr A] @ [v1], loc, length (compE2 a) + pc', ⌊A⌋)
      by(rule bisim1-bisims1.bisim1AAccThrow2)
    ultimately show ?thesis using τ by auto
  qed
 qed auto
next
 case bisim1AAccThrow1 thus ?case by auto
next
 case bisim1AAccThrow2 thus ?case by auto
next
 case bisim1AAccFail thus ?case by auto
next
 case (bisim1AAss1 a n a' xs stk loc pc xcp i e)
 note IH1 = bisim1AAss1.IH(2)
 note IH2 = bisim1AAss1.IH(4)
 note IH3 = bisim1AAss1.IH(6)
 note bisim1 = ‹P,a,h ⊢ (a', xs) ↔ (stk, loc, pc, xcp)›
 note bisim2 = ‹⋀xs. P,i,h ⊢ (i, xs) ↔ ([], xs, 0, None)›
 note bisim3 = ‹⋀xs. P,e,h ⊢ (e, xs) ↔ ([], xs, 0, None)›
 note bsok = ‹bsok (a⌊i⌋ := e) n›
 from ‹True,P,t ⊢1 ⟨a'⌊i⌋ := e,(h, xs)⟩ −ta→ ⟨e',(h', xs')⟩› show ?case
 proof cases
  case (AAss1Red1 E')
  note [simp] = ‹e' = E'⌊i⌋ := e›
    and red = ‹True,P,t ⊢1 ⟨a',(h, xs)⟩ −ta→ ⟨E',(h', xs')⟩›
   from red have τmove1 P h (a'⌊i⌋ := e) = τmove1 P h a' by(auto simp add: τmove1.simps τmoves1.simps)
  moreover from red have call1 (a'⌊i⌋ := e) = call1 a' by auto
  moreover from IH1[OF red] bsok
  obtain pc'' stk'' loc'' xcp'' where bisim: P,a,h' ⊢ (E', xs') ↔ (stk'', loc'', pc'', xcp'')
    and redo: ?exec ta a a' E' h stk loc pc xcp h' pc'' stk'' loc'' xcp'' by auto
  from bisim
  have P,a⌊i⌋ := e,h' ⊢ (E'⌊i⌋ := e, xs') ↔ (stk'', loc'', pc'', xcp'')
    by(rule bisim1-bisims1.bisim1AAss1)
  moreover {
    assume no-call2 a pc
    hence no-call2 (a⌊i⌋ := e) pc ∨ pc = length (compE2 a) by(auto simp add: no-call2-def) }
  ultimately show ?thesis using redo
```

by(*auto simp del*: *call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)(*blast intro*: *AAss-τExecrI1 AAss-τExectI1 exec-move-AAssI1*)+
next
case (*AAss1Red2 E′ v*)
note [*simp*] = ‹*a′ = Val v*› ‹*e′ = Val v⌊E⌈ := e*›
and *red* = ‹*True,P,t ⊢1* ⟨*i,*(*h, xs*)⟩ −*ta*→ ⟨*E′,*(*h′, xs′*)⟩›
from *red* **have** *τ*: *τmove1 P h* (*Val v⌊i⌉ := e*) = *τmove1 P h i* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim1* **have** *s*: *xcp = None xs = loc*
and *exec1*: *τExec-mover-a P t a h* (*stk, loc, pc, None*) ([*v*], *xs, length* (*compE2 a*), *None*)
by(*auto dest*: *bisim1Val2D1*)
from *exec1* **have** *τExec-mover-a P t* (*a⌊i⌉ := e*) *h* (*stk, loc, pc, None*) ([*v*], *xs, length* (*compE2 a*), *None*)
by(*rule AAss-τExecrI1*)
moreover
from *IH2*[*OF red*] *bsok* **obtain** *pc′′ stk′′ loc′′ xcp′′*
where *bisim′*: *P,i,h′ ⊢* (*E′, xs′*) ↔ (*stk′′, loc′′, pc′′, xcp′′*)
and *exec′*: *?exec ta i i E′ h* [] *xs 0 None h′ pc′′ stk′′ loc′′ xcp′′* **by** *auto*
have *?exec ta* (*a⌊i⌉ := e*) (*Val v⌊i⌉ := e*) (*Val v⌊E⌈ := e*) *h* ([] @ [*v*]) *xs* (*length* (*compE2 a*) + *0*) *None h′* (*length* (*compE2 a*) + *pc′′*) (*stk′′* @ [*v*]) *loc′′ xcp′′*
proof(*cases τmove1 P h* (*Val v⌊i⌉ := e*))
case *True*
with *exec′ τ* **have** [*simp*]: *h = h′* **and** *e*: *sim-move i E′ P t i h* ([], *xs*, *0*, *None*) (*stk′′, loc′′, pc′′, xcp′′*) **by** *auto*
from *e* **have** *sim-move* (*a⌊i⌉ := e*) (*a⌊E⌈ := e*) *P t* (*a⌊i⌉ := e*) *h* ([] @ [*v*], *xs*, *length* (*compE2 a*) + *0*, *None*) (*stk′′* @ [*v*], *loc′′*, *length* (*compE2 a*) + *pc′′*, *xcp′′*)
by(*fastforce dest*: *AAss-τExecrI2 AAss-τExectI2*)
with *True* **show** *?thesis* **by** *auto*
next
case *False*
with *exec′ τ* **obtain** *pc′ stk′ loc′ xcp′*
where *e*: *τExec-mover-a P t i h* ([], *xs*, *0*, *None*) (*stk′, loc′, pc′, xcp′*)
and *e′*: *exec-move-a P t i h* (*stk′, loc′, pc′, xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk′′, loc′′, pc′′, xcp′′*)
and *τ′*: ¬ *τmove2* (*compP2 P*) *h stk′ i pc′ xcp′*
and *call*: *call1 i = None* ∨ *no-call2 i 0* ∨ *pc′ = 0* ∧ *stk′ =* [] ∧ *loc′ = xs* ∧ *xcp′ = None* **by** *auto*
from *e* **have** *τExec-mover-a P t* (*a⌊i⌉ := e*) *h* ([] @ [*v*], *xs*, *length* (*compE2 a*) + *0*, *None*) (*stk′* @ [*v*], *loc′*, *length* (*compE2 a*) + *pc′*, *xcp′*) **by**(*rule AAss-τExecrI2*)
moreover from *e′* **have** *exec-move-a P t* (*a⌊i⌉ := e*) *h* (*stk′* @ [*v*], *loc′*, *length* (*compE2 a*) + *pc′*, *xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk′′* @ [*v*], *loc′′*, *length* (*compE2 a*) + *pc′′*, *xcp′′*)
by(*rule exec-move-AAssI2*)
moreover from *e′* **have** *pc′ < length* (*compE2 i*) **by**(*auto elim*: *exec-meth.cases*)
with *τ′ e′* **have** ¬ *τmove2* (*compP2 P*) *h* (*stk′* @ [*v*]) (*a⌊i⌉ := e*) (*length* (*compE2 a*) + *pc′*) *xcp′*
by(*auto simp add*: *τinstr-stk-drop-exec-move τmove2-iff*)
moreover from *red* **have** *call1* (*a′⌊i⌉ := e*) = *call1 i* **by** *auto*
moreover have *no-call2 i 0* ⟹ *no-call2* (*a⌊i⌉ := e*) (*length* (*compE2 a*))
by(*auto simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *False call*
by(*auto simp del*: *split-paired-Ex call1.simps calls1.simps*) *blast*
qed
moreover from *bisim′*
have *P,a⌊i⌉ := e,h′ ⊢* (*Val v⌊E⌈ := e, xs′*) ↔ ((*stk′′* @ [*v*]), *loc′′*, *length* (*compE2 a*) + *pc′′*, *xcp′′*)
by(*rule bisim1-bisims1.bisim1AAss2*)

moreover from *bisim1* **have** $pc \neq length\ (compE2\ a) \longrightarrow no\text{-}call2\ (a\lfloor i\rceil := e)\ pc$
by(*auto simp add*: *no-call2-def dest*: *bisim-Val-pc-not-Invoke bisim1-pc-length-compE2*)
ultimately show *?thesis* **using** τ *exec1 s*
apply(*auto simp del*: *split-paired-Ex call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)
apply(*blast intro*: τ*Exec-mover-trans*|*fastforce elim*!: τ*Exec-mover-trans simp del*: *split-paired-Ex call1.simps calls1.simps*)+
done
next
case (*AAss1Red3 E' v v'*)
note [*simp*] = ‹$i = Val\ v'$› ‹$a' = Val\ v$› ‹$e' = Val\ v\lfloor Val\ v'\rceil := E'$›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e*,(*h*, *xs*)⟩ −*ta*→ ⟨*E'*,(*h'*, *xs'*)⟩›
from *red* **have** τ: $\tau move1\ P\ h\ (Val\ v\lfloor Val\ v'\rceil := e) = \tau move1\ P\ h\ e$ **by**(*auto simp add*: τ*move1.simps* τ*moves1.simps*)
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *exec1*: τ*Exec-mover-a P t a h* (*stk*, *loc*, *pc*, *None*) ([] @ [*v*], *xs*, *length* (*compE2 a*) + *0*, *None*)
by(*auto dest*: *bisim1Val2D1*)
from *exec1* **have** τ*Exec-mover-a P t* $(a\lfloor i\rceil := e)$ *h* (*stk*, *loc*, *pc*, *None*) ([] @ [*v*], *xs*, *length* (*compE2 a*) + *0*, *None*)
by(*rule AAss-*τ*ExecrI1*)
also from *bisim2*[*of xs*]
have τ*Exec-mover-a P t i h* ([], *xs*, *0*, *None*) ([*v'*], *xs*, *length* (*compE2 i*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence τ*Exec-mover-a P t* $(a\lfloor i\rceil := e)$ *h* ([] @ [*v*], *xs*, *length* (*compE2 a*) + *0*, *None*) ([*v'*] @ [*v*], *xs*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*)
by(*rule AAss-*τ*ExecrI2*)
also (*rtranclp-trans*) **from** *IH3*[*OF red*] *bsok* **obtain** *pc''* *stk''* *loc''* *xcp''*
where *bisim'*: *P*,*e*,*h'* ⊢ (*E'*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
and *exec'*: *?exec ta e e E' h* [] *xs 0 None h' pc'' stk'' loc'' xcp''* **by** *auto*
have *?exec ta* $(a\lfloor i\rceil := e)$ $(Val\ v\lfloor Val\ v'\rceil := e)$ $(Val\ v\lfloor Val\ v'\rceil := E')$ *h* ([] @ [*v'*, *v*]) *xs* (*length* (*compE2 a*) + *length* (*compE2 i*) + *0*) *None h'* (*length* (*compE2 a*) + *length* (*compE2 i*) + *pc''*) (*stk''* @ [*v'*, *v*]) *loc''* *xcp''*
proof(*cases* τ*move1 P h* $(Val\ v\lfloor Val\ v'\rceil := e)$)
case *True*
with *exec'* τ **have** [*simp*]: *h* = *h'* **and** *e*: *sim-move e E' P t e h* ([], *xs*, *0*, *None*) (*stk''*, *loc''*, *pc''*, *xcp''*) **by** *auto*
from *e* **have** *sim-move* $(Val\ v\lfloor Val\ v'\rceil := e)$ $(Val\ v\lfloor Val\ v'\rceil := E')$ *P t* $(a\lfloor i\rceil := e)$ *h* ([] @ [*v'*, *v*], *xs*, *length* (*compE2 a*) + *length* (*compE2 i*) + *0*, *None*) (*stk''* @ [*v'*, *v*], *loc''*, *length* (*compE2 a*) + *length* (*compE2 i*) + *pc''*, *xcp''*)
by(*fastforce dest*: *AAss-*τ*ExectI3 AAss-*τ*ExecrI3 simp del*: *compE2.simps compEs2.simps*)
with *True* **show** *?thesis* **by** *auto*
next
case *False*
with *exec'* τ **obtain** *pc' stk' loc' xcp'*
where *e*: τ*Exec-mover-a P t e h* ([], *xs*, *0*, *None*) (*stk'*, *loc'*, *pc'*, *xcp'*)
and *e'*: *exec-move-a P t e h* (*stk'*, *loc'*, *pc'*, *xcp'*) (*extTA2JVM* (*compP2 P*) *ta*) *h'* (*stk''*, *loc''*, *pc''*, *xcp''*)
and τ': ¬ τ*move2* (*compP2 P*) *h stk' e pc' xcp'*
and *call*: *call1 e* = *None* ∨ *no-call2 e 0* ∨ *pc'* = *0* ∧ *stk'* = [] ∧ *loc'* = *xs* ∧ *xcp'* = *None* **by** *auto*
from *e* **have** τ*Exec-mover-a P t* $(a\lfloor i\rceil := e)$ *h* ([] @ [*v'*, *v*], *xs*, *length* (*compE2 a*) + *length* (*compE2 i*) + *0*, *None*) (*stk'* @ [*v'*, *v*], *loc'*, *length* (*compE2 a*) + *length* (*compE2 i*) + *pc'*, *xcp'*) **by**(*rule AAss-*τ*ExecrI3*)
moreover from *e'* **have** *exec-move-a P t* $(a\lfloor i\rceil := e)$ *h* (*stk'* @ [*v'*, *v*], *loc'*, *length* (*compE2 a*) + *length* (*compE2 i*) + *pc'*, *xcp'*) (*extTA2JVM* (*compP2 P*) *ta*) *h'* (*stk''* @ [*v'*, *v*], *loc''*, *length* (*compE2*

a) + *length* (*compE2 i*) + *pc''*, *xcp''*)
by(*rule exec-move-AAssI3*)
moreover from *e' τ'*
have ¬ *τmove2* (*compP2 P*) *h* (*stk'* @ [*v'*, *v*]) (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *length* (*compE2 i*) + *pc'*) *xcp'*
by(*auto simp add*: *τinstr-stk-drop-exec-move τmove2-iff*)
moreover have *call1* (*a'*⌊*i*⌉ := *e*) = *call1 e* **by** *simp*
moreover have *no-call2 e 0* ⟹ *no-call2* (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *length* (*compE2 i*))
by(*auto simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *False call*
by(*auto simp del*: *split-paired-Ex call1.simps calls1.simps*) *blast*
qed
moreover from *bisim'*
have *P*,*a*⌊*i*⌉ := *e*,*h'* ⊢ (*Val v*⌊*Val v'*⌉ := *E'*, *xs'*) ↔ ((*stk''* @ [*v'*, *v*]), *loc''*, *length* (*compE2 a*) + *length* (*compE2 i*) + *pc''*, *xcp''*)
by(*rule bisim1-bisims1.bisim1AAss3*)
moreover from *bisim1* **have** *pc* ≠ *length* (*compE2 a*) + *length* (*compE2 i*) ⟶ *no-call2* (*a*⌊*i*⌉ := *e*) *pc*
by(*auto simp add*: *no-call2-def dest*: *bisim-Val-pc-not-Invoke bisim1-pc-length-compE2*)
ultimately show *?thesis* **using** *τ exec1 s*
apply(*auto simp del*: *split-paired-Ex call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)
apply(*blast intro*: *τExec-mover-trans*|*fastforce elim*!: *τExec-mover-trans simp del*: *split-paired-Ex call1.simps calls1.simps*)+
done
next
case (*Red1AAss A U len I v U'*)
hence [*simp*]: *a'* = *addr A e'* = *unit i* = *Val* (*Intg I*)
ta = {⌈*WriteMem A* (*ACell* (*nat* (*sint I*))) *v*⌋} *xs'* = *xs e* = *Val v*
and *hA*: *typeof-addr h A* = ⌊*Array-type U len*⌋ **and** *I*: *0* <=*s I sint I* < *int len*
and *v*: *typeof*$_h$ *v* = ⌊*U'*⌋ *P* ⊢ *U'* ≤ *U*
and *h'*: *heap-write h A* (*ACell* (*nat* (*sint I*))) *v h'* **by** *auto*
have *τ*: ¬ *τmove1 P h* (*AAss* (*addr A*) (*Val* (*Intg I*)) (*Val v*)) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t a h* (*stk*, *loc*, *pc*, *xcp*) ([] @ [*Addr A*], *loc*, *length* (*compE2 a*) + *0*, *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk*, *loc*, *pc*, *xcp*) ([] @ [*Addr A*], *loc*, *length* (*compE2 a*) + *0*, *None*)
by−(*rule AAss-τExecrI1*)
also from *bisim2*[*of loc*]
have *τExec-mover-a P t i h* ([], *loc*, *0*, *None*) ([*Intg I*], *loc*, *length* (*compE2 i*) + *0*, *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* ([] @ [*Addr A*], *loc*, *length* (*compE2 a*) + *0*, *None*) ([*Intg I*] @ [*Addr A*], *loc*, *length* (*compE2 a*) + (*length* (*compE2 i*) + *0*), *None*)
by(*rule AAss-τExecrI2*)
also (*rtranclp-trans*) **have** [*Intg I*] @ [*Addr A*] = [] @ [*Intg I*, *Addr A*] **by** *simp*
also note *add.assoc*[*symmetric*]
also from *bisim3*[*of loc*] **have** *τExec-mover-a P t e h* ([], *loc*, *0*, *None*) ([*v*], *loc*, *length* (*compE2 e*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* ([] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *0*, *None*) ([*v*] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*)
by(*rule AAss-τExecrI3*)

also (*rtranclp-trans*) **from** *hA I v h'*
have *exec-move-a P t* (*a*⌊*i*⌉ := *e*) *h* ([*v*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*)
⦃*WriteMem A* (*ACell* (*nat* (*sint I*))) *v*⦄
h' ([], *loc*, *Suc* (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)), *None*)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *compP2-def is-Ref-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v*, *Intg I*, *Addr A*] (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *a*⌊*i*⌉ := *e*, *h'* ⊢ (*unit*, *loc*) ↔ ([], *loc*, *Suc* (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)), *None*)
by(*rule bisim1-bisims1.bisim1AAss4*)
ultimately show *?thesis* **using** *s τ* **by**(*auto simp add*: *ta-upd-simps*) *blast*
next
case (*Red1AAssNull v v'*)
note [*simp*] = ‹*a'* = *null*› ‹*e'* = *THROW NullPointer*› ‹*i* = *Val v*› ‹*xs'* = *xs*› ‹*ta* = ε› ‹*h'* = *h*› ‹*e* = *Val v'*›
have *τ*: ¬ *τmove1 P h* (*AAss null* (*Val v*) (*Val v'*)) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t a h* (*stk*, *loc*, *pc*, *xcp*) ([] @ [*Null*], *loc*, *length* (*compE2 a*) + *0*, *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk*, *loc*, *pc*, *xcp*) ([] @ [*Null*], *loc*, *length* (*compE2 a*) + *0*, *None*)
by−(*rule AAss-τExecrI1*)
also from *bisim2*[*of loc*] **have** *τExec-mover-a P t i h* ([], *loc*, *0*, *None*) ([*v*], *loc*, *length* (*compE2 i*) + *0*, *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* ([] @ [*Null*], *loc*, *length* (*compE2 a*) + *0*, *None*) ([*v*] @ [*Null*], *loc*, *length* (*compE2 a*) + (*length* (*compE2 i*) + *0*), *None*)
by(*rule AAss-τExecrI2*)
also (*rtranclp-trans*) **have** [*v*] @ [*Null*] = [] @ [*v*, *Null*] **by** *simp*
also note *add.assoc*[*symmetric*]
also from *bisim3*[*of loc*] **have** *τExec-mover-a P t e h* ([], *loc*, *0*, *None*) ([*v'*], *loc*, *length* (*compE2 e*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* ([] @ [*v*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *0*, *None*) ([*v'*] @ [*v*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*)
by(*rule AAss-τExecrI3*)
also (*rtranclp-trans*)
have *exec-move-a P t* (*a*⌊*i*⌉ := *e*) *h* ([*v'*, *v*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*) ε
h ([*v'*, *v*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *is-Ref-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v'*, *v*, *Null*] (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *a*⌊*i*⌉ := *e*, *h'* ⊢ (*THROW NullPointer*, *loc*) ↔ ([*v'*, *v*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule bisim1-bisims1.bisim1AAssFail*)

ultimately show *?thesis* **using** $s\ \tau$ **by** *auto blast*
next
case (*Red1AAssBounds A U len I v*)
hence [*simp*]: $a' = addr\ A\ e' = THROW\ ArrayIndexOutOfBounds\ i = Val\ (Intg\ I)\ xs' = xs\ ta = \varepsilon\ h' = h\ e = Val\ v$
and *hA*: *typeof-addr h A* = ⌊*Array-type U len*⌋ **and** *I*: $I <s 0 \vee int\ len \leq sint\ I$ **by** *auto*
have τ: ¬ τ*move1 P h* (*AAss* (*addr A*) *i e*) **by**(*auto simp add*: τ*move1.simps* τ*moves1.simps*)
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and τ*Exec-mover-a P t a h* (*stk*, *loc*, *pc*, *xcp*) ([] @ [*Addr A*], *loc*, *length* (*compE2 a*) + *0*, *None*)
by(*auto dest*: *bisim1Val2D1*)
hence τ*Exec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk*, *loc*, *pc*, *xcp*) ([] @ [*Addr A*], *loc*, *length* (*compE2 a*) + *0*, *None*)
by−(*rule AAss-*τ*ExecrI1*)
also from *bisim2*[*of loc*]
have τ*Exec-mover-a P t i h* ([], *loc*, *0*, *None*) ([*Intg I*], *loc*, *length* (*compE2 i*) + *0*, *None*)
by(*auto dest*: *bisim1Val2D1*)
hence τ*Exec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* ([] @ [*Addr A*], *loc*, *length* (*compE2 a*) + *0*, *None*) ([*Intg I*] @ [*Addr A*], *loc*, *length* (*compE2 a*) + (*length* (*compE2 i*) + *0*), *None*)
by(*rule AAss-*τ*ExecrI2*)
also (*rtranclp-trans*) **have** [*Intg I*] @ [*Addr A*] = [] @ [*Intg I*, *Addr A*] **by** *simp*
also note *add.assoc*[*symmetric*]
also from *bisim3*[*of loc*]
have τ*Exec-mover-a P t e h* ([], *loc*, *0*, *None*) ([*v*], *loc*, *length* (*compE2 e*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence τ*Exec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* ([] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *0*, *None*) ([*v*] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*)
by(*rule AAss-*τ*ExecrI3*)
also (*rtranclp-trans*) **from** *hA I*
have *exec-move-a P t* (*a*⌊*i*⌉ := *e*) *h* ([*v*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*) *ε*
h ([*v*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), ⌊*addr-of-sys-xcpt ArrayIndexOutOfBounds*⌋)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *is-Ref-def*)
moreover have τ*move2* (*compP2 P*) *h* [*v*, *Intg I*, *Addr A*] (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)) *None* ⟹ *False*
by(*simp add*: τ*move2-iff*)
moreover
have *P*, *a*⌊*i*⌉ := *e*, *h'* ⊢ (*THROW ArrayIndexOutOfBounds*, *loc*) ↔ ([*v*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), ⌊*addr-of-sys-xcpt ArrayIndexOutOfBounds*⌋)
by(*rule bisim1-bisims1.bisim1AAssFail*)
ultimately show *?thesis* **using** $s\ \tau$ **by** *auto blast*
next
case (*Red1AAssStore A U len I v U'*)
hence [*simp*]: $a' = addr\ A\ e' = THROW\ ArrayStore\ i = Val\ (Intg\ I)\ xs' = xs\ ta = \varepsilon\ h' = h\ e = Val\ v$
and *hA*: *typeof-addr h A* = ⌊*Array-type U len*⌋ **and** *I*: $0 <=s\ I\ sint\ I < int\ len$
and *U*: ¬ *P* ⊢ *U'* ≤ *U typeof*$_h$ *v* = ⌊*U'*⌋ **by** *auto*
have τ: ¬ τ*move1 P h* (*AAss* (*addr A*) *i e*) **by**(*auto simp add*: τ*move1.simps* τ*moves1.simps*)
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and τ*Exec-mover-a P t a h* (*stk*, *loc*, *pc*, *xcp*) ([] @ [*Addr A*], *loc*, *length* (*compE2 a*) + *0*, *None*)
by(*auto dest*: *bisim1Val2D1*)
hence τ*Exec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk*, *loc*, *pc*, *xcp*) ([] @ [*Addr A*], *loc*, *length* (*compE2 a*) + *0*, *None*)

by−(*rule AAss-τExecrI1*)
also from *bisim2*[*of loc*]
have *τExec-mover-a P t i h* ([], *loc*, *0*, *None*) ([*Intg I*], *loc*, *length* (*compE2 i*) + *0*, *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* ([] @ [*Addr A*], *loc*, *length* (*compE2 a*) + *0*, *None*) ([*Intg I*] @ [*Addr A*], *loc*, *length* (*compE2 a*) + (*length* (*compE2 i*) + *0*), *None*)
by(*rule AAss-τExecrI2*)
also (*rtranclp-trans*) **have** [*Intg I*] @ [*Addr A*] = [] @ [*Intg I*, *Addr A*] **by** *simp*
also note *add.assoc*[*symmetric*]
also from *bisim3*[*of loc*]
have *τExec-mover-a P t e h* ([], *loc*, *0*, *None*) ([*v*], *loc*, *length* (*compE2 e*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* ([] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *0*, *None*) ([*v*] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*)
by(*rule AAss-τExecrI3*)
also (*rtranclp-trans*) **from** *hA I U*
have *exec-move-a P t* (*a*⌊*i*⌉ := *e*) *h* ([*v*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*) *ε*
h ([*v*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), ⌊*addr-of-sys-xcpt ArrayStore*⌋)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *is-Ref-def compP2-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v*, *Intg I*, *Addr A*] (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *a*⌊*i*⌉ := *e*, *h'* ⊢ (*THROW ArrayStore*, *loc*) ↔ ([*v*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), ⌊*addr-of-sys-xcpt ArrayStore*⌋)
by(*rule bisim1-bisims1.bisim1AAssFail*)
ultimately show *?thesis* **using** *s τ* **by** *auto blast*
next
case (*AAss1Throw1 A*)
hence [*simp*]: *a'* = *Throw A ta* = *ε e'* = *Throw A h'* = *h xs'* = *xs* **by** *auto*
have *τ*: *τmove1 P h* (*Throw A*⌊*i*⌉ := *e*) **by**(*rule τmove1AAssThrow1*)
from *bisim1* **have** *xcp* = ⌊*A*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*A*⌋
with *bisim1* **have** *P*, *a*⌊*i*⌉ := *e*, *h* ⊢ (*Throw A*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
by(*auto intro*: *bisim1-bisims1.intros*)
thus *?thesis* **using** *τ* **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim1* **obtain** *pc'* **where** *τExec-mover-a P t a h* (*stk*, *loc*, *pc*, *None*) ([*Addr A*], *loc*, *pc'*, ⌊*A*⌋)
and *bisim'*: *P*, *a*, *h* ⊢ (*Throw A*, *xs*) ↔ ([*Addr A*], *loc*, *pc'*, ⌊*A*⌋)
and [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk*, *loc*, *pc*, *None*) ([*Addr A*], *loc*, *pc'*, ⌊*A*⌋)
by−(*rule AAss-τExecrI1*)
moreover from *bisim'*
have *P*, *a*⌊*i*⌉ := *e*, *h* ⊢ (*Throw A*, *xs*) ↔ ([*Addr A*], *loc*, *pc'*, ⌊*A*⌋)
by(*auto intro*: *bisim1-bisims1.bisim1AAssThrow1*)
ultimately show *?thesis* **using** *τ* **by** *auto*

qed
next
case (*AAss1Throw2 v ad*)
note [*simp*] = ‹*a′ = Val v*› ‹*i = Throw ad*› ‹*ta = ε*› ‹*e′ = Throw ad*› ‹*h′ = h*› ‹*xs′ = xs*›
from *bisim1* **have** *s*: *xcp = None xs = loc*
and *τExec-mover-a P t a h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 a*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*Throw ad*⌉ := *e*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 a*), *None*)
by−(*rule AAss-τExecrI1*)
also have *τExec-mover-a P t* (*a*⌊*Throw ad*⌉:=*e*) *h* ([*v*], *loc*, *length* (*compE2 a*), *None*) ([*Addr ad*, *v*], *loc*, *Suc* (*length* (*compE2 a*)), ⌊*ad*⌋)
by(*rule τExecr2step*)(*auto simp add*: *exec-move-def exec-meth-instr τmove2-iff τmove1.simps τmoves1.simps*)
also (*rtranclp-trans*)
have *P*,*a*⌊*Throw ad*⌉:=*e*,*h* ⊢ (*Throw ad*, *loc*) ↔ ([*Addr ad*] @ [*v*], *loc*, (*length* (*compE2 a*) + *length* (*compE2* (*addr ad*))), ⌊*ad*⌋)
by(*rule bisim1AAssThrow2*[*OF bisim1Throw2*])
moreover have *τmove1 P h* (*a′*⌊*Throw ad*⌉:=*e*) **by**(*auto intro*: *τmove1AAssThrow2*)
ultimately show *?thesis* **using** *s* **by** *auto*
next
case (*AAss1Throw3 va vi ad*)
note [*simp*] = ‹*a′ = Val va*› ‹*i = Val vi*› ‹*e = Throw ad*› ‹*ta = ε*› ‹*e′ = Throw ad*› ‹*h′ = h*› ‹*xs′ = xs*›
from *bisim1* **have** *s*: *xcp = None xs = loc*
and *τExec-mover-a P t a h* (*stk*, *loc*, *pc*, *xcp*) ([*va*], *loc*, *length* (*compE2 a*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *Throw ad*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*va*], *loc*, *length* (*compE2 a*), *None*)
by−(*rule AAss-τExecrI1*)
also from *bisim2*[*of loc*] **have** *τExec-mover-a P t i h* ([], *loc*, *0*, *None*) ([*vi*], *loc*, *length* (*compE2 i*), *None*)
by(*auto dest*: *bisim1Val2D1*)
from *AAss-τExecrI2*[*OF this*, *of a e va*]
have *τExec-mover-a P t* (*a*⌊*i*⌉ := *Throw ad*) *h* ([*va*], *loc*, *length* (*compE2 a*), *None*) ([*vi*, *va*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*) **by** *simp*
also (*rtranclp-trans*)
have *τExec-mover-a P t* (*a*⌊*i*⌉:=*Throw ad*) *h* ([*vi*, *va*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*) ([*Addr ad*, *vi*, *va*], *loc*, *Suc* (*length* (*compE2 a*) + *length* (*compE2 i*)), ⌊*ad*⌋)
by(*rule τExecr2step*)(*auto simp add*: *exec-move-def exec-meth-instr τmove2-iff τmove1.simps τmoves1.simps*)
also (*rtranclp-trans*)
have *P*,*a*⌊*i*⌉:=*Throw ad*,*h* ⊢ (*Throw ad*, *loc*) ↔ ([*Addr ad*] @ [*vi*, *va*], *loc*, (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2* (*addr ad*))), ⌊*ad*⌋)
by(*rule bisim1AAssThrow3*[*OF bisim1Throw2*])
moreover have *τmove1 P h* (*AAss a′* (*Val vi*) (*Throw ad*)) **by**(*auto intro*: *τmove1AAssThrow3*)
ultimately show *?thesis* **using** *s* **by** *auto*
qed
next
case (*bisim1AAss2 i n i′ xs stk loc pc xcp a e v1*)
note *IH2* = *bisim1AAss2.IH*(*2*)
note *IH3* = *bisim1AAss2.IH*(*6*)
note *bisim2* = ‹*P*,*i*,*h* ⊢ (*i′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim1* = ‹⋀*xs*. *P*,*a*,*h* ⊢ (*a*, *xs*) ↔ ([], *xs*, *0*, *None*)›

note *bisim3* = ‹$\bigwedge$*xs*. *P*,*e*,*h* ⊢ (*e*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bsok* = ‹*bsok* (*a*⌊*i*⌉ := *e*) *n*›
from ‹*True*,*P*,*t* ⊢*1* ⟨*Val v1*⌊*i'*⌉ := *e*,(*h*, *xs*)⟩ −*ta*→ ⟨*e'*,(*h'*, *xs'*)⟩› **show** *?case*
proof *cases*
case (*AAss1Red2 E'*)
note [*simp*] = ‹*e'* = *Val v1*⌊*E'*⌉ := *e*›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*i'*,(*h*, *xs*)⟩ −*ta*→ ⟨*E'*,(*h'*, *xs'*)⟩›
from *red* **have** *τ*: *τmove1 P h* (*Val v1*⌊*i'*⌉ := *e*) = *τmove1 P h i'* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *IH2*[*OF red*] *bsok* **obtain** *pc''* *stk''* *loc''* *xcp''*
where *bisim'*: *P*,*i*,*h'* ⊢ (*E'*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
and *exec'*: *?exec ta i i' E' h stk loc pc xcp h' pc'' stk'' loc'' xcp''* **by** *auto*
have *?exec ta* (*a*⌊*i*⌉ := *e*) (*Val v1*⌊*i'*⌉ := *e*) (*Val v1*⌊*E'*⌉ := *e*) *h* (*stk* @ [*v1*]) *loc* (*length* (*compE2 a*) + *pc*) *xcp h'* (*length* (*compE2 a*) + *pc''*) (*stk''* @ [*v1*]) *loc'' xcp''*
proof(*cases τmove1 P h* (*Val v1*⌊*i'*⌉ := *e*))
case *True*
with *exec' τ* **have** [*simp*]: *h* = *h'* **and** *e*: *sim-move i' E' P t i h* (*stk*, *loc*, *pc*, *xcp*) (*stk''*, *loc''*, *pc''*, *xcp''*) **by** *auto*
from *e* **have** *sim-move* (*Val v1*⌊*i'*⌉ := *e*) (*Val v1*⌊*E'*⌉ := *e*) *P t* (*a*⌊*i*⌉ := *e*) *h* (*stk* @ [*v1*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) (*stk''* @ [*v1*], *loc''*, *length* (*compE2 a*) + *pc''*, *xcp''*)
by(*fastforce dest*: *AAss-τExecrI2 AAss-τExectI2 simp del*: *compE2.simps compEs2.simps*)
with *True* **show** *?thesis* **by** *auto*
next
case *False*
with *exec' τ* **obtain** *pc' stk' loc' xcp'*
where *e*: *τExec-mover-a P t i h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
and *e'*: *exec-move-a P t i h* (*stk'*, *loc'*, *pc'*, *xcp'*) (*extTA2JVM* (*compP2 P*) *ta*) *h'* (*stk''*, *loc''*, *pc''*, *xcp''*)
and *τ'*: ¬ *τmove2* (*compP2 P*) *h stk' i pc' xcp'*
and *call*: *call1 i'* = *None* ∨ *no-call2 i pc* ∨ *pc'* = *pc* ∧ *stk'* = *stk* ∧ *loc'* = *loc* ∧ *xcp'* = *xcp* **by** *auto*
from *e* **have** *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk* @ [*v1*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) (*stk'* @ [*v1*], *loc'*, *length* (*compE2 a*) + *pc'*, *xcp'*) **by**(*rule AAss-τExecrI2*)
moreover from *e'* **have** *exec-move-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk'* @ [*v1*], *loc'*, *length* (*compE2 a*) + *pc'*, *xcp'*) (*extTA2JVM* (*compP2 P*) *ta*) *h'* (*stk''* @ [*v1*], *loc''*, *length* (*compE2 a*) + *pc''*, *xcp''*)
by(*rule exec-move-AAssI2*)
moreover from *e'* **have** *pc'* < *length* (*compE2 i*) **by**(*auto elim*: *exec-meth.cases*)
with *τ' e'* **have** ¬ *τmove2* (*compP2 P*) *h* (*stk'* @ [*v1*]) (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *pc'*) *xcp'*
by(*auto simp add*: *τinstr-stk-drop-exec-move τmove2-iff*)
moreover from *red* **have** *call1* (*Val v1*⌊*i'*⌉ := *e*) = *call1 i'* **by** *auto*
moreover have *no-call2 i pc* ⟹ *no-call2* (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *pc*)
by(*auto simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *False call* **by**(*auto simp del*: *split-paired-Ex call1.simps calls1.simps*)

qed
moreover from *bisim'*
have *P*,*a*⌊*i*⌉ := *e*,*h'* ⊢ (*Val v1*⌊*E'*⌉ := *e*, *xs'*) ↔ ((*stk''* @ [*v1*]), *loc''*, *length* (*compE2 a*) + *pc''*, *xcp''*)
by(*rule bisim1-bisims1.bisim1AAss2*)
ultimately show *?thesis*
apply(*auto simp del*: *split-paired-Ex call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)
apply(*blast intro*: *τExec-mover-trans*)+
done

next
case (*AAss1Red3 E′ v′*)
note [*simp*] = ‹*i′ = Val v′*› ‹*e′ = Val v1⌊Val v′⌉ := E′*›
and *red* = ‹*True,P,t ⊢1 ⟨e,(h, xs)⟩ −ta→ ⟨E′,(h′, xs′)⟩*›
from *red* **have** *τ*: *τmove1 P h* (*Val v1⌊Val v′⌉ := e*) = *τmove1 P h e*
by(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and *exec1*: *τExec-mover-a P t i h* (*stk*, *loc*, *pc*, *xcp*) ([*v′*], *xs*, *length* (*compE2 i*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a⌊i⌉ := e*) *h* (*stk* @ [*v1*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) ([*v′*] @ [*v1*], *xs*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*)
by−(*rule AAss-τExecrI2*)
moreover from *IH3*[*OF red*] *bsok* **obtain** *pc″ stk″ loc″ xcp″*
where *bisim′*: *P,e,h′ ⊢* (*E′*, *xs′*) ↔ (*stk″*, *loc″*, *pc″*, *xcp″*)
and *exec′*: *?exec ta e e E′ h* [] *xs 0 None h′ pc″ stk″ loc″ xcp″* **by** *auto*
have *?exec ta* (*a⌊i⌉ := e*) (*Val v1⌊Val v′⌉ := e*) (*Val v1⌊Val v′⌉ := E′*) *h* ([] @ [*v′*, *v1*]) *xs* (*length* (*compE2 a*) + *length* (*compE2 i*) + *0*) *None h′* (*length* (*compE2 a*) + *length* (*compE2 i*) + *pc″*) (*stk″* @ [*v′*, *v1*]) *loc″ xcp″*
proof(*cases τmove1 P h* (*Val v1⌊Val v′⌉ := e*))
case *True*
with *exec′ τ* **have** [*simp*]: *h* = *h′*
and *e*: *sim-move e E′ P t e h* ([], *xs*, *0*, *None*) (*stk″*, *loc″*, *pc″*, *xcp″*) **by** *auto*
from *e* **have** *sim-move* (*Val v1⌊Val v′⌉ := e*) (*Val v1⌊Val v′⌉ := E′*) *P t* (*a⌊i⌉ := e*) *h* ([] @ [*v′*, *v1*], *xs*, *length* (*compE2 a*) + *length* (*compE2 i*) + *0*, *None*) (*stk″* @ [*v′*, *v1*], *loc″*, *length* (*compE2 a*) + *length* (*compE2 i*) + *pc″*, *xcp″*)
by(*fastforce dest*: *AAss-τExectI3 AAss-τExecrI3 simp del*: *compE2.simps compEs2.simps*)
with *True* **show** *?thesis* **by** *auto*
next
case *False*
with *exec′ τ* **obtain** *pc′ stk′ loc′ xcp′*
where *e*: *τExec-mover-a P t e h* ([], *xs*, *0*, *None*) (*stk′*, *loc′*, *pc′*, *xcp′*)
and *e′*: *exec-move-a P t e h* (*stk′*, *loc′*, *pc′*, *xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk″*, *loc″*, *pc″*, *xcp″*)
and *τ′*: ¬ *τmove2* (*compP2 P*) *h stk′ e pc′ xcp′*
and *call*: *call1 e* = *None* ∨ *no-call2 e 0* ∨ *pc′* = *0* ∧ *stk′* = [] ∧ *loc′* = *xs* ∧ *xcp′* = *None* **by** *auto*
from *e* **have** *τExec-mover-a P t* (*a⌊i⌉ := e*) *h* ([] @ [*v′*, *v1*], *xs*, *length* (*compE2 a*) + *length* (*compE2 i*) + *0*, *None*) (*stk′* @ [*v′*, *v1*], *loc′*, *length* (*compE2 a*) + *length* (*compE2 i*) + *pc′*, *xcp′*) **by**(*rule AAss-τExecrI3*)
moreover from *e′* **have** *exec-move-a P t* (*a⌊i⌉ := e*) *h* (*stk′* @ [*v′*, *v1*], *loc′*, *length* (*compE2 a*) + *length* (*compE2 i*) + *pc′*, *xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk″* @ [*v′*, *v1*], *loc″*, *length* (*compE2 a*) + *length* (*compE2 i*) + *pc″*, *xcp″*)
by(*rule exec-move-AAssI3*)
moreover from *e′ τ′* **have** ¬ *τmove2* (*compP2 P*) *h* (*stk′* @ [*v′*, *v1*]) (*a⌊i⌉ := e*) (*length* (*compE2 a*) + *length* (*compE2 i*) + *pc′*) *xcp′*
by(*auto simp add*: *τinstr-stk-drop-exec-move τmove2-iff*)
moreover from *red* **have** *call1* (*Val v1⌊Val v′⌉ := e*) = *call1 e* **by** *auto*
moreover have *no-call2 e 0* ⟹ *no-call2* (*a⌊i⌉ := e*) (*length* (*compE2 a*) + *length* (*compE2 i*))
by(*auto simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *False call* **by**(*auto simp del*: *split-paired-Ex call1.simps calls1.simps*) *blast*
qed
moreover from *bisim′*
have *P,a⌊i⌉ := e,h′ ⊢* (*Val v1⌊Val v′⌉ := E′*, *xs′*) ↔ ((*stk″* @ [*v′*, *v1*]), *loc″*, *length* (*compE2 a*)

+ *length* (*compE2 i*) + *pc''*, *xcp''*)
by(*rule bisim1-bisims1.bisim1AAss3*)
moreover from *bisim2* **have** *pc* ≠ *length* (*compE2 i*) ⟶ *no-call2* (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *pc*)
by(*auto simp add*: *no-call2-def dest*: *bisim-Val-pc-not-Invoke bisim1-pc-length-compE2*)
ultimately show *?thesis* **using** *τ exec1 s*
apply(*auto simp del*: *split-paired-Ex call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)
apply(*blast intro*: *τExec-mover-trans*|*fastforce elim*!: *τExec-mover-trans simp del*: *split-paired-Ex call1.simps calls1.simps*)+
done
next
case (*Red1AAss A U len I v U'*)
hence [*simp*]: *v1* = *Addr A e'* = *unit i'* = *Val* (*Intg I*)
ta = {⌈*WriteMem A* (*ACell* (*nat* (*sint I*))) *v*⌋} *xs'* = *xs e* = *Val v*
and *hA*: *typeof-addr h A* = ⌊*Array-type U len*⌋ **and** *I*: *0* <=*s I sint I* < *int len*
and *v*: *typeof*$_h$ *v* = ⌊*U'*⌋ *P* ⊢ *U'* ≤ *U*
and *h'*: *heap-write h A* (*ACell* (*nat* (*sint I*))) *v h'* **by** *auto*
have *τ*: ¬ *τmove1 P h* (*AAss* (*addr A*) (*Val* (*Intg I*)) (*Val v*)) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t i h* (*stk*, *loc*, *pc*, *xcp*) ([*Intg I*], *loc*, *length* (*compE2 i*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk* @ [*Addr A*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) ([*Intg I*] @ [*Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*)
by−(*rule AAss-τExecrI2*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk* @ [*Addr A*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) ([] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *0*, *None*) **by** *simp*
also from *bisim3*[*of loc*] **have** *τExec-mover-a P t e h* ([], *loc*, *0*, *None*) ([*v*], *loc*, *length* (*compE2 e*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* ([] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *0*, *None*) ([*v*] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*)
by(*rule AAss-τExecrI3*)
also (*rtranclp-trans*) **from** *hA I v h'*
have *exec-move-a P t* (*a*⌊*i*⌉ := *e*) *h* ([*v*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*)
{⌈*WriteMem A* (*ACell* (*nat* (*sint I*))) *v*⌋}
h' ([], *loc*, *Suc* (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)), *None*)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *compP2-def is-Ref-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v*, *Intg I*, *Addr A*] (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *a*⌊*i*⌉ := *e*, *h'* ⊢ (*unit*, *loc*) ↔ ([], *loc*, *Suc* (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)), *None*)
by(*rule bisim1-bisims1.bisim1AAss4*)
ultimately show *?thesis* **using** *s τ* **by**(*auto simp add*: *ta-upd-simps*) *blast*
next
case (*Red1AAssNull v v'*)
note [*simp*] = ‹*v1* = *Null*› ‹*e'* = *THROW NullPointer*› ‹*i'* = *Val v*› ‹*xs'* = *xs*› ‹*ta* = ε› ‹*h'* = *h*› ‹*e* = *Val v'*›
have *τ*: ¬ *τmove1 P h* (*AAss null* (*Val v*) (*Val v'*)) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)

from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t i h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 i*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk* @ [*Null*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) ([*v*] @ [*Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*)
by−(*rule AAss-τExecrI2*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk* @ [*Null*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) ([] @ [*v*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *0*, *None*) **by** *simp*
also from *bisim3*[*of loc*] **have** *τExec-mover-a P t e h* ([], *loc*, *0*, *None*) ([*v'*], *loc*, *length* (*compE2 e*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* ([] @ [*v*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *0*, *None*) ([*v'*] @ [*v*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*)
by(*rule AAss-τExecrI3*)
also (*rtranclp-trans*)
have *exec-move-a P t* (*a*⌊*i*⌉ := *e*) *h* ([*v'*, *v*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*) *ε*
h ([*v'*, *v*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *is-Ref-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v'*, *v*, *Null*] (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *a*⌊*i*⌉ := *e*, *h'* ⊢ (*THROW NullPointer*, *loc*) ↔ ([*v'*, *v*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule bisim1-bisims1.bisim1AAssFail*)
ultimately show *?thesis* **using** *s τ* **by** *auto blast*
next
case (*Red1AAssBounds A U len I v*)
hence [*simp*]: *v1* = *Addr A e'* = *THROW ArrayIndexOutOfBounds i'* = *Val* (*Intg I*) *xs'* = *xs ta* = *ε h'* = *h e* = *Val v*
and *hA*: *typeof-addr h A* = ⌊*Array-type U len*⌋ **and** *I*: *I* <*s 0* ∨ *int len* ≤ *sint I* **by** *auto*
have *τ*: ¬ *τmove1 P h* (*addr A*⌊*i'*⌉ := *e*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t i h* (*stk*, *loc*, *pc*, *xcp*) ([*Intg I*], *loc*, *length* (*compE2 i*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk* @ [*Addr A*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) ([*Intg I*] @ [*Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*)
by−(*rule AAss-τExecrI2*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk* @ [*Addr A*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) ([] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *0*, *None*) **by** *simp*
also from *bisim3*[*of loc*] **have** *τExec-mover-a P t e h* ([], *loc*, *0*, *None*) ([*v*], *loc*, *length* (*compE2 e*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* ([] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *0*, *None*) ([*v*] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*)
by(*rule AAss-τExecrI3*)
also (*rtranclp-trans*) **from** *hA I*
have *exec-move-a P t* (*a*⌊*i*⌉ := *e*) *h* ([*v*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*) *ε*
h ([*v*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length*

(*compE2 e*), ⌊*addr-of-sys-xcpt ArrayIndexOutOfBounds*⌋)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *is-Ref-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v*, *Intg I*, *Addr A*] (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *a*⌊*i*⌉ := *e*, *h'* ⊢ (*THROW ArrayIndexOutOfBounds*, *loc*) ↔ ([*v*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), ⌊*addr-of-sys-xcpt ArrayIndexOutOfBounds*⌋)
by(*rule bisim1-bisims1.bisim1AAssFail*)
ultimately show *?thesis* **using** *s τ* **by** *auto blast*
next
case (*Red1AAssStore A U len I v U'*)
hence [*simp*]: *v1* = *Addr A e'* = *THROW ArrayStore i'* = *Val* (*Intg I*) *xs'* = *xs ta* = *ε h'* = *h e* = *Val v*
and *hA*: *typeof-addr h A* = ⌊*Array-type U len*⌋ **and** *I*: *0* <=*s I sint I* < *int len*
and *U*: ¬ *P* ⊢ *U'* ≤ *U typeof*$_h$ *v* = ⌊*U'*⌋ **by** *auto*
have *τ*: ¬ *τmove1 P h* (*addr A*⌊*i'*⌉ := *e*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t i h* (*stk*, *loc*, *pc*, *xcp*) ([*Intg I*], *loc*, *length* (*compE2 i*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk* @ [*Addr A*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) ([*Intg I*] @ [*Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*)
by−(*rule AAss-τExecrI2*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk* @ [*Addr A*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) ([] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *0*, *None*) **by** *simp*
also from *bisim3*[*of loc*]
have *τExec-mover-a P t e h* ([], *loc*, *0*, *None*) ([*v*], *loc*, *length* (*compE2 e*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* ([] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *0*, *None*) ([*v*] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*)
by(*rule AAss-τExecrI3*)
also (*rtranclp-trans*) **from** *hA I U*
have *exec-move-a P t* (*a*⌊*i*⌉ := *e*) *h* ([*v*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*) *ε*
h ([*v*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), ⌊*addr-of-sys-xcpt ArrayStore*⌋)
unfolding *exec-move-def* **by**− (*rule exec-instr*, *auto simp add*: *is-Ref-def compP2-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v*, *Intg I*, *Addr A*] (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *a*⌊*i*⌉ := *e*, *h'* ⊢ (*THROW ArrayStore*, *loc*) ↔ ([*v*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), ⌊*addr-of-sys-xcpt ArrayStore*⌋)
by(*rule bisim1-bisims1.bisim1AAssFail*)
ultimately show *?thesis* **using** *s τ* **by** *auto fast*
next
case (*AAss1Throw2 A*)
note [*simp*] = ‹*i'* = *Throw A*› ‹*ta* = *ε*› ‹*e'* = *Throw A*› ‹*h'* = *h*› ‹*xs'* = *xs*›
have *τ*: *τmove1 P h* (*Val v1*⌊*Throw A*⌉ := *e*) **by**(*rule τmove1AAssThrow2*)
from *bisim2* **have** *xcp* = ⌊*A*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*A*⌋

with *bisim2*
have P, $a\lfloor i\rceil := e$, $h \vdash$ (*Throw A*, *xs*) $\leftrightarrow$ (*stk* @ [*v1*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*)
by(*auto intro*: *bisim1-bisims1.intros*)
thus *?thesis* **using** τ **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim2* **obtain** *pc′* **where** *τExec-mover-a P t i h* (*stk*, *loc*, *pc*, *None*) ([*Addr A*], *loc*, *pc′*, $\lfloor A\rfloor$)
and *bisim′*: *P*, *i*, *h* $\vdash$ (*Throw A*, *xs*) $\leftrightarrow$ ([*Addr A*], *loc*, *pc′*, $\lfloor A\rfloor$)
and [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t* ($a\lfloor i\rceil := e$) *h* (*stk* @ [*v1*], *loc*, *length* (*compE2 a*) + *pc*, *None*) ([*Addr A*] @ [*v1*], *loc*, *length* (*compE2 a*) + *pc′*, $\lfloor A\rfloor$)
by−(*rule AAss-τExecrI2*)
moreover from *bisim′*
have P, $a\lfloor i\rceil := e$, $h \vdash$ (*Throw A*, *xs*) $\leftrightarrow$ ([*Addr A*] @ [*v1*], *loc*, *length* (*compE2 a*) + *pc′*, $\lfloor A\rfloor$)
by(*rule bisim1-bisims1.bisim1AAssThrow2*)
ultimately show *?thesis* **using** τ **by** *auto*
qed
next
case (*AAss1Throw3 vi ad*)
note [*simp*] = ‹*i′* = *Val vi*› ‹*e* = *Throw ad*› ‹*ta* = ε› ‹*e′* = *Throw ad*› ‹*h′* = *h*› ‹*xs′* = *xs*›
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t i h* (*stk*, *loc*, *pc*, *xcp*) ([*vi*], *loc*, *length* (*compE2 i*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* ($a\lfloor i\rceil$:= *Throw ad*) *h* (*stk* @ [*v1*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) ([*vi*] @ [*v1*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*)
by−(*rule AAss-τExecrI2*)
also have *τExec-mover-a P t* ($a\lfloor i\rceil$:=*Throw ad*) *h* ([*vi*] @ [*v1*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*), *None*) ([*Addr ad*, *vi*, *v1*], *loc*, *Suc* (*length* (*compE2 a*) + *length* (*compE2 i*)), $\lfloor ad\rfloor$)
by(*rule τExecr2step*)(*auto simp add*: *exec-move-def exec-meth-instr τmove2-iff τmove1.simps τmoves1.simps*)
also (*rtranclp-trans*)
have *P*,$a\lfloor i\rceil$:=*Throw ad*,*h* $\vdash$ (*Throw ad*, *loc*) $\leftrightarrow$ ([*Addr ad*] @ [*vi*, *v1*], *loc*, (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2* (*addr ad*))), $\lfloor ad\rfloor$)
by(*rule bisim1AAssThrow3*[*OF bisim1Throw2*])
moreover have *τmove1 P h* (*AAss* (*Val v1*) (*Val vi*) (*Throw ad*)) **by**(*auto intro*: *τmove1AAssThrow3*)
ultimately show *?thesis* **using** *s* **by** *auto*
qed *auto*
next
case (*bisim1AAss3 e n ee xs stk loc pc xcp a i v v′*)
note *IH3* = *bisim1AAss3.IH*(2)
note *bisim3* = ‹*P*,*e*,*h* $\vdash$ (*ee*, *xs*) $\leftrightarrow$ (*stk*, *loc*, *pc*, *xcp*)›
note *bsok* = ‹*bsok* ($a\lfloor i\rceil := e$) *n*›
from ‹*True*,*P*,*t* $\vdash$1 ⟨*Val v*$\lfloor$*Val v′*$\rceil$:= *ee*,(*h*, *xs*)⟩ −*ta*→ ⟨*e′*,(*h′*, *xs′*)⟩› **show** *?case*
proof *cases*
case (*AAss1Red3 E′*)
note [*simp*] = ‹*e′* = *Val v*$\lfloor$*Val v′*$\rceil$:= *E′*›
and *red* = ‹*True*,*P*,*t* $\vdash$1 ⟨*ee*,(*h*, *xs*)⟩ −*ta*→ ⟨*E′*,(*h′*, *xs′*)⟩›
from *red* **have** τ: *τmove1 P h* (*Val v*$\lfloor$*Val v′*$\rceil$:= *ee*) = *τmove1 P h ee* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *IH3*[*OF red*] *bsok* **obtain** *pc″ stk″ loc″ xcp″*
where *bisim′*: *P*,*e*,*h′* $\vdash$ (*E′*, *xs′*) $\leftrightarrow$ (*stk″*, *loc″*, *pc″*, *xcp″*)
and *exec′*: *?exec ta e ee E′ h stk loc pc xcp h′ pc″ stk″ loc″ xcp″* **by** *auto*

have *no-call2 e pc* $\Longrightarrow$ *no-call2* (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *length* (*compE2 i*) + *pc*)
by(*auto simp add*: *no-call2-def*)
hence *?exec ta* (*a*⌊*i*⌉ := *e*) (*Val v*⌊*Val v'*⌉ := *ee*) (*Val v*⌊*Val v'*⌉ := *E'*) *h* (*stk* @ [*v'*, *v*]) *loc* (*length* (*compE2 a*) + *length* (*compE2 i*) + *pc*) *xcp h'* (*length* (*compE2 a*) + *length* (*compE2 i*) + *pc''*) (*stk''* @ [*v'*, *v*]) *loc''* *xcp''*
using *exec'* τ
apply(*cases* τ*move1 P h* (*Val v*⌊*Val v'*⌉ := *ee*))
apply(*auto*)
apply(*blast intro*: *AAss-*τ*ExecrI3 AAss-*τ*ExectI3 exec-move-AAssI3*)
apply(*blast intro*: *AAss-*τ*ExecrI3 AAss-*τ*ExectI3 exec-move-AAssI3*)
apply(*rule exI conjI AAss-*τ*ExecrI3 exec-move-AAssI3*|*assumption*)+
apply(*fastforce simp add*: τ*instr-stk-drop-exec-move* τ*move2-iff split*: *if-split-asm*)
apply(*rule exI conjI AAss-*τ*ExecrI3 exec-move-AAssI3*|*assumption*)+
apply(*fastforce simp add*: τ*instr-stk-drop-exec-move* τ*move2-iff split*: *if-split-asm*)
apply(*rule exI conjI AAss-*τ*ExecrI3 exec-move-AAssI3 rtranclp.rtrancl-refl*|*assumption*)+
apply(*fastforce simp add*: τ*instr-stk-drop-exec-move* τ*move2-iff split*: *if-split-asm*)+
done
moreover from *bisim'*
have *P*,*a*⌊*i*⌉ := *e*,*h'* ⊢ (*Val v*⌊*Val v'*⌉ := *E'*, *xs'*) ↔ (*stk''*@[*v'*,*v*], *loc''*, *length* (*compE2 a*) + *length* (*compE2 i*) + *pc''*, *xcp''*)
by(*rule bisim1-bisims1.bisim1AAss3*)
ultimately show *?thesis* **using** τ **by** *auto blast*+
next
case (*Red1AAss A U len I V U'*)
hence [*simp*]: *v* = *Addr A e'* = *unit v'* = *Intg I xs'* = *xs ee* = *Val V*
ta = {⌈*WriteMem A* (*ACell* (*nat* (*sint I*))) *V*⌋}
and *hA*: *typeof-addr h A* = ⌊*Array-type U len*⌋ **and** *I*: *0* <=*s I sint I* < *int len*
and *v*: *typeof*$_h$ *V* = ⌊*U'*⌋ *P* ⊢ *U'* ≤ *U*
and *h'*: *heap-write h A* (*ACell* (*nat* (*sint I*))) *V h'* **by** *auto*
have τ: ¬ τ*move1 P h* (*AAss* (*addr A*) (*Val* (*Intg I*)) (*Val V*)) **by**(*auto simp add*: τ*move1.simps* τ*moves1.simps*)
from *bisim3* **have** *s*: *xcp* = *None xs* = *loc*
and *exec1*: τ*Exec-mover-a P t e h* (*stk*, *loc*, *pc*, *xcp*) ([*V*], *loc*, *length* (*compE2 e*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence τ*Exec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk* @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *pc*, *xcp*) ([*V*] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*)
by−(*rule AAss-*τ*ExecrI3*)
moreover from *hA I v h'*
have *exec-move-a P t* (*a*⌊*i*⌉ := *e*) *h* ([*V*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*)
{⌈*WriteMem A* (*ACell* (*nat* (*sint I*))) *V*⌋}
h' ([], *loc*, *Suc* (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)), *None*)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *compP2-def is-Ref-def*)
moreover have τ*move2* (*compP2 P*) *h* [*V*, *Intg I*, *Addr A*] (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)) *None* $\Longrightarrow$ *False*
by(*simp add*: τ*move2-iff*)
moreover
have *P*, *a*⌊*i*⌉ := *e*, *h'* ⊢ (*unit*, *loc*) ↔ ([], *loc*, *Suc* (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)), *None*)
by(*rule bisim1-bisims1.bisim1AAss4*)
ultimately show *?thesis* **using** *s* τ **by**(*auto simp add*: *ta-upd-simps*) *blast*
next

case (*Red1AAssNull V'*)
note [*simp*] = ‹*v* = *Null*› ‹*e'* = *THROW NullPointer*› ‹*xs'* = *xs*› ‹*ta* = *ε*› ‹*h'* = *h*› ‹*ee* = *Val V'*›
have *τ*: ¬ *τmove1 P h* (*AAss null* (*Val v'*) (*Val V'*)) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim3* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e h* (*stk*, *loc*, *pc*, *xcp*) ([*V'*], *loc*, *length* (*compE2 e*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌋ := *e*) *h* (*stk* @ [*v'*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *pc*, *xcp*) ([*V'*] @ [*v'*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*)
by−(*rule AAss-τExecrI3*)
moreover
have *exec-move-a P t* (*a*⌊*i*⌋ := *e*) *h* ([*V'*, *v'*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*) *ε*
h ([*V'*, *v'*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *is-Ref-def*)
moreover have *τmove2* (*compP2 P*) *h* [*V'*, *v'*, *Null*] (*a*⌊*i*⌋ := *e*) (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *a*⌊*i*⌋ := *e*, *h'* ⊢ (*THROW NullPointer*, *loc*) ↔ ([*V'*, *v'*, *Null*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule bisim1-bisims1.bisim1AAssFail*)
ultimately show *?thesis* **using** *s τ* **by** *auto blast*
next
case (*Red1AAssBounds A U len I V*)
hence [*simp*]: *v* = *Addr A e'* = *THROW ArrayIndexOutOfBounds v'* = *Intg I xs'* = *xs ta* = *ε h'* = *h ee* = *Val V*
and *hA*: *typeof-addr h A* = ⌊*Array-type U len*⌋ **and** *I*: *I* <*s* *0* ∨ *int len* ≤ *sint I* **by** *auto*
have *τ*: ¬ *τmove1 P h* (*addr A*⌊*Val* (*Intg I*)⌉ := *ee*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim3* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e h* (*stk*, *loc*, *pc*, *xcp*) ([*V*], *loc*, *length* (*compE2 e*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌋ := *e*) *h* (*stk* @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *pc*, *xcp*) ([*V*] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*)
by−(*rule AAss-τExecrI3*)
moreover from *hA I*
have *exec-move-a P t* (*a*⌊*i*⌋ := *e*) *h* ([*V*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*) *ε*
h ([*V*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), ⌊*addr-of-sys-xcpt ArrayIndexOutOfBounds*⌋)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *is-Ref-def*)
moreover have *τmove2* (*compP2 P*) *h* [*V*, *Intg I*, *Addr A*] (*a*⌊*i*⌋ := *e*) (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *a*⌊*i*⌋ := *e*, *h'* ⊢ (*THROW ArrayIndexOutOfBounds*, *loc*) ↔ ([*V*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), ⌊*addr-of-sys-xcpt ArrayIndexOutOfBounds*⌋)
by(*rule bisim1-bisims1.bisim1AAssFail*)
ultimately show *?thesis* **using** *s τ* **by** *auto blast*
next
case (*Red1AAssStore A U len I V U'*)
hence [*simp*]: *v* = *Addr A e'* = *THROW ArrayStore v'* = *Intg I xs'* = *xs ta* = *ε h'* = *h ee* = *Val V*

and *hA*: *typeof-addr h A* = ⌊*Array-type U len*⌋ **and** *I*: *0* <=*s I sint I* < *int len*
and *U*: ¬ *P* ⊢ *U'* ≤ *U typeof*$_h$ *V* = ⌊*U'*⌋ **by** *auto*
have *τ*: ¬ *τmove1 P h* (*addr A*⌊*Val* (*Intg I*)⌉ := *ee*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim3* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e h* (*stk*, *loc*, *pc*, *xcp*) ([*V*], *loc*, *length* (*compE2 e*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk* @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *pc*, *xcp*) ([*V*] @ [*Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*)
by−(*rule AAss-τExecrI3*)
moreover from *hA I U*
have *exec-move-a P t* (*a*⌊*i*⌉ := *e*) *h* ([*V*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), *None*) *ε*
h ([*V*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), ⌊*addr-of-sys-xcpt ArrayStore*⌋)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *is-Ref-def compP2-def*)
moreover have *τmove2* (*compP2 P*) *h* [*V*, *Intg I*, *Addr A*] (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *a*⌊*i*⌉ := *e*, *h'* ⊢ (*THROW ArrayStore*, *loc*) ↔ ([*V*, *Intg I*, *Addr A*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*), ⌊*addr-of-sys-xcpt ArrayStore*⌋)
by(*rule bisim1-bisims1.bisim1AAssFail*)
ultimately show *?thesis* **using** *s τ* **by** *auto blast*
next
case (*AAss1Throw3 A*)
note [*simp*] = ‹*ee* = *Throw A*› ‹*ta* = *ε*› ‹*e'* = *Throw A*› ‹*h'* = *h*› ‹*xs'* = *xs*›
have *τ*: *τmove1 P h* (*AAss* (*Val v*) (*Val v'*) (*Throw A*)) **by**(*rule τmove1AAssThrow3*)
from *bisim3* **have** *xcp* = ⌊*A*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*A*⌋
with *bisim3*
have *P*, *a*⌊*i*⌉ := *e*, *h* ⊢ (*Throw A*, *xs*) ↔ (*stk* @ [*v'*, *v*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *pc*, *xcp*)
by(*auto intro*: *bisim1-bisims1.intros*)
thus *?thesis* **using** *τ* **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim3* **obtain** *pc'* **where** *τExec-mover-a P t e h* (*stk*, *loc*, *pc*, *None*) ([*Addr A*], *loc*, *pc'*, ⌊*A*⌋)
and *bisim'*: *P*, *e*, *h* ⊢ (*Throw A*, *xs*) ↔ ([*Addr A*], *loc*, *pc'*, ⌊*A*⌋)
and [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t* (*a*⌊*i*⌉ := *e*) *h* (*stk* @ [*v'*, *v*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *pc*, *None*) ([*Addr A*] @ [*v'*, *v*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *pc'*, ⌊*A*⌋)
by−(*rule AAss-τExecrI3*)
moreover from *bisim'*
have *P*, *a*⌊*i*⌉ := *e*, *h* ⊢ (*Throw A*, *xs*) ↔ ([*Addr A*] @ [*v'*, *v*], *loc*, *length* (*compE2 a*) + *length* (*compE2 i*) + *pc'*, ⌊*A*⌋)
by(*rule bisim1-bisims1.bisim1AAssThrow3*)
ultimately show *?thesis* **using** *τ* **by** *auto*
qed
qed *auto*

next
case *bisim1AAssThrow1* **thus** *?case* **by** *auto*
next
case *bisim1AAssThrow2* **thus** *?case* **by** *auto*
next
case *bisim1AAssThrow3* **thus** *?case* **by** *auto*
next
case *bisim1AAssFail* **thus** *?case* **by** *auto*
next
case *bisim1AAss4* **thus** *?case* **by** *auto*
next
case (*bisim1ALength a n a' xs stk loc pc xcp*)
note *IH* = *bisim1ALength.IH*(*2*)
note *bisim* = ‹*P*,*a*,*h* ⊢ (*a'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*a'*·*length*,(*h*, *xs*)⟩ −*ta*→ ⟨*e'*,(*h'*, *xs'*)⟩›
note *bsok* = ‹*bsok* (*a*·*length*) *n*›
from *red* **show** *?case*
proof *cases*
case (*ALength1Red ee'*)
note [*simp*] = ‹*e'* = *ee'*·*length*›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*a'*,(*h*, *xs*)⟩ −*ta*→ ⟨*ee'*, (*h'*, *xs'*)⟩›
from *red* **have** *τmove1 P h* (*a'*·*length*) = *τmove1 P h a'* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
moreover have *call1* (*a'*·*length*) = *call1 a'* **by** *auto*
moreover from *IH*[*OF red*] *bsok*
obtain *pc'' stk'' loc'' xcp''* **where** *bisim*: *P*,*a*,*h'* ⊢ (*ee'*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
and *redo*: *?exec ta a a' ee' h stk loc pc xcp h' pc'' stk'' loc'' xcp''* **by** *auto*
from *bisim* **have** *P*,*a*·*length*,*h'* ⊢ (*ee'*·*length*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
by(*rule bisim1-bisims1.bisim1ALength*)
moreover {
assume *no-call2 a pc*
hence *no-call2* (*a*·*length*) *pc* **by**(*auto simp add*: *no-call2-def*) }
ultimately show *?thesis* **using** *redo*
by(*auto simp del*: *call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)(*blast intro*: *ALength-τExecrI ALength-τExectI exec-move-ALengthI*)+
next
case (*Red1ALength A U len*)
hence [*simp*]: *a'* = *addr A ta* = *ε e'* = *Val* (*Intg* (*word-of-int* (*int len*)))
h' = *h xs'* = *xs*
and *hA*: *typeof-addr h A* = ⌊*Array-type U len*⌋ **by** *auto*
from *bisim* **have** *s*: *xcp* = *None xs* = *loc* **by**(*auto dest*: *bisim-Val-loc-eq-xcp-None*)
from *bisim* **have** *τExec-mover-a P t a h* (*stk*, *loc*, *pc*, *xcp*) ([*Addr A*], *loc*, *length* (*compE2 a*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*a*·*length*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*Addr A*], *loc*, *length* (*compE2 a*), *None*)
by(*rule ALength-τExecrI*)
moreover from *hA*
have *exec-move-a P t* (*a*·*length*) *h* ([*Addr A*], *loc*, *length* (*compE2 a*), *None*) *ε h'* ([*Intg* (*word-of-int* (*int len*))], *loc*, *Suc* (*length* (*compE2 a*)), *None*)
by(*auto intro*!: *exec-instr simp add*: *is-Ref-def exec-move-def*)
moreover have *τmove2* (*compP2 P*) *h* [*Addr A*] (*a*·*length*) (*length* (*compE2 a*)) *None* ⟹ *False* **by**(*simp add*: *τmove2-iff*)
moreover have ¬ *τmove1 P h* (*addr A*·*length*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
moreover

have P, $a\cdot length$, $h' \vdash$ (Val ($Intg$ ($word\text{-}of\text{-}int$ (int len))), loc) $\leftrightarrow$ ([$Intg$ ($word\text{-}of\text{-}int$ (int len))], loc, $length$ ($compE2$ ($a\cdot length$)), $None$)
by($rule$ $bisim1Val2$) $simp$
ultimately show $?thesis$ **using** s **by**($auto$) $blast$
next
case $Red1ALengthNull$
note [$simp$] = ‹$a' = null$› ‹$e' = THROW\ NullPointer$› ‹$h' = h$› ‹$xs' = xs$› ‹$ta = \varepsilon$›
have ¬ $\tau move1\ P\ h$ ($null\cdot length$) **by**($auto$ $simp$ add: $\tau move1.simps$ $\tau moves1.simps$)
moreover from $bisim$ **have** s: $xcp = None$ $xs = loc$
and $\tau Exec\text{-}mover\text{-}a\ P\ t\ a\ h$ (stk, loc, pc, xcp) ([$Null$], loc, $length$ ($compE2$ a), $None$)
by($auto$ $dest$: $bisim1Val2D1$)
hence $\tau Exec\text{-}mover\text{-}a\ P\ t$ ($a\cdot length$) h (stk, loc, pc, xcp) ([$Null$], loc, $length$ ($compE2$ a), $None$)
by−($rule$ $ALength\text{-}\tau ExecrI$)
moreover have $exec\text{-}move\text{-}a\ P\ t$ ($a\cdot length$) h ([$Null$], loc, $length$ ($compE2$ a), $None$) ε h ([$Null$], loc, $length$ ($compE2$ a), $\lfloor addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer \rfloor$)
unfolding $exec\text{-}move\text{-}def$ **by** −($rule$ $exec\text{-}instr$, $auto$ $simp$ add: $is\text{-}Ref\text{-}def$)
moreover have $\tau move2$ ($compP2$ P) h [$Null$] ($a\cdot length$) ($length$ ($compE2$ a)) $None \Longrightarrow False$ **by**($simp$ add: $\tau move2\text{-}iff$)
moreover
have P,$a\cdot length$,$h \vdash$ ($THROW\ NullPointer$, loc) $\leftrightarrow$ ([$Null$], loc, $length$ ($compE2$ a), $\lfloor addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer \rfloor$)
by($auto$ $intro$!: $bisim1\text{-}bisims1.bisim1ALengthNull$)
ultimately show $?thesis$ **using** s **by** $auto$ $blast$
next
case ($ALength1Throw$ A)
note [$simp$] = ‹$a' = Throw\ A$› ‹$h' = h$› ‹$xs' = xs$› ‹$ta = \varepsilon$› ‹$e' = Throw\ A$›
have τ: $\tau move1\ P\ h$ ($Throw\ A\cdot length$) **by**($auto$ $intro$: $\tau move1ALengthThrow$)
from $bisim$ **have** $xcp = \lfloor A \rfloor \vee xcp = None$ **by**($auto$ $dest$: $bisim1\text{-}ThrowD$)
thus $?thesis$
proof
assume [$simp$]: $xcp = \lfloor A \rfloor$
with $bisim$ **have** P,$a\cdot length$, $h \vdash$ ($Throw\ A$, xs) $\leftrightarrow$ (stk, loc, pc, xcp)
by($auto$ $intro$: $bisim1\text{-}bisims1.bisim1ALengthThrow$)
thus $?thesis$ **using** τ **by**($fastforce$)
next
assume [$simp$]: $xcp = None$
with $bisim$ **obtain** pc'
where $\tau Exec\text{-}mover\text{-}a\ P\ t\ a\ h$ (stk, loc, pc, $None$) ([$Addr\ A$], loc, pc', $\lfloor A \rfloor$)
and $bisim'$: P, a, $h \vdash$ ($Throw\ A$, xs) $\leftrightarrow$ ([$Addr\ A$], loc, pc', $\lfloor A \rfloor$) **and** [$simp$]: $xs = loc$
by($auto$ $dest$: $bisim1\text{-}Throw\text{-}\tau Exec\text{-}mover$)
hence $\tau Exec\text{-}mover\text{-}a\ P\ t$ ($a\cdot length$) h (stk, loc, pc, $None$) ([$Addr\ A$], loc, pc', $\lfloor A \rfloor$)
by−($rule$ $ALength\text{-}\tau ExecrI$)
moreover from $bisim'$ **have** P, $a\cdot length$, $h \vdash$ ($Throw\ A$, xs) $\leftrightarrow$ ([$Addr\ A$], loc, pc', $\lfloor A \rfloor$)
by($rule$ $bisim1\text{-}bisims1.bisim1ALengthThrow$)
ultimately show $?thesis$ **using** τ **by** $auto$
qed
qed
next
case $bisim1ALengthThrow$ **thus** $?case$ **by** $auto$
next
case $bisim1ALengthNull$ **thus** $?case$ **by** $auto$
next
case ($bisim1FAcc$ E n e xs stk loc pc xcp F D)
note $IH = bisim1FAcc.IH(2)$

note *bisim* = ‹*P*,*E*,*h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e*·*F*{*D*},(*h*, *xs*)⟩ −*ta*→ ⟨*e*′,(*h*′, *xs*′)⟩›
note *bsok* = ‹*bsok* (*E*·*F*{*D*}) *n*›
from *red* **show** *?case*
proof *cases*
case (*FAcc1Red ee*′)
note [*simp*] = ‹*e*′ = *ee*′·*F*{*D*}›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e*,(*h*, *xs*)⟩ −*ta*→ ⟨*ee*′, (*h*′, *xs*′)⟩›
from *red* **have** *τmove1 P h* (*e*·*F*{*D*}) = *τmove1 P h e* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
moreover have *call1* (*e*·*F*{*D*}) = *call1 e* **by** *auto*
moreover from *IH*[*OF red*] *bsok*
obtain *pc*′′ *stk*′′ *loc*′′ *xcp*′′ **where** *bisim*: *P*,*E*,*h*′ ⊢ (*ee*′, *xs*′) ↔ (*stk*′′, *loc*′′, *pc*′′, *xcp*′′)
and *redo*: *?exec ta E e ee*′ *h stk loc pc xcp h*′ *pc*′′ *stk*′′ *loc*′′ *xcp*′′ **by** *auto*
from *bisim*
have *P*,*E*·*F*{*D*},*h*′ ⊢ (*ee*′·*F*{*D*}, *xs*′) ↔ (*stk*′′, *loc*′′, *pc*′′, *xcp*′′)
by(*rule bisim1-bisims1.bisim1FAcc*)
moreover {
assume *no-call2 E pc*
hence *no-call2* (*E*·*F*{*D*}) *pc* **by**(*auto simp add*: *no-call2-def*) **}**
ultimately show *?thesis* **using** *redo*
by(*auto simp del*: *call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)(*blast intro*: *FAcc-τExecrI FAcc-τExectI exec-move-FAccI*)+
next
case (*Red1FAcc a v*)
hence [*simp*]: *e* = *addr a ta* = {⌊*ReadMem a* (*CField D F*) *v*⌋} *e*′ = *Val v h*′ = *h xs*′ = *xs*
and *read*: *heap-read h a* (*CField D F*) *v* **by** *auto*
from *bisim* **have** *s*: *xcp* = *None xs* = *loc* **by**(*auto dest*: *bisim-Val-loc-eq-xcp-None*)
from *bisim* **have** *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *xcp*) ([*Addr a*], *loc*, *length* (*compE2 E*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*E*·*F*{*D*}) *h* (*stk*, *loc*, *pc*, *xcp*) ([*Addr a*], *loc*, *length* (*compE2 E*), *None*)
by(*rule FAcc-τExecrI*)
moreover from *read*
have *exec-move-a P t* (*E*·*F*{*D*}) *h* ([*Addr a*], *loc*, *length* (*compE2 E*), *None*)
{⌊*ReadMem a* (*CField D F*) *v*⌋} *h*′ ([*v*], *loc*, *Suc* (*length* (*compE2 E*)), *None*)
unfolding *exec-move-def* **by**(*auto intro*!: *exec-instr*)
moreover have *τmove2* (*compP2 P*) *h* [*Addr a*] (*E*·*F*{*D*}) (*length* (*compE2 E*)) *None* ⟹ *False* **by**(*simp add*: *τmove2-iff*)
moreover have ¬ *τmove1 P h* (*addr a*·*F*{*D*}) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
moreover
have *P*, *E*·*F*{*D*}, *h*′ ⊢ (*Val v*, *loc*) ↔ ([*v*], *loc*, *length* (*compE2* (*E*·*F*{*D*})), *None*)
by(*rule bisim1Val2*) *simp*
ultimately show *?thesis* **using** *s* **by**(*auto simp add*: *ta-upd-simps*) *blast*
next
case *Red1FAccNull*
note [*simp*] = ‹*e* = *null*› ‹*e*′ = *THROW NullPointer*› ‹*h*′ = *h*› ‹*xs*′ = *xs*› ‹*ta* = ε›
have ¬ *τmove1 P h* (*null*·*F*{*D*}) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
moreover from *bisim* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *xcp*) ([*Null*], *loc*, *length* (*compE2 E*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*E*·*F*{*D*}) *h* (*stk*, *loc*, *pc*, *xcp*) ([*Null*], *loc*, *length* (*compE2 E*), *None*)
by−(*rule FAcc-τExecrI*)
moreover

have *exec-move-a P t* (*E*·*F*{*D*}) *h* ([*Null*], *loc*, *length* (*compE2 E*), *None*) *ε h* ([*Null*], *loc*, *length* (*compE2 E*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *exec-move-def* **by** −(*rule exec-instr*, *auto simp add*: *compP2-def dest*: *sees-field-idemp*)
moreover have *τmove2* (*compP2 P*) *h* [*Null*] (*E*·*F*{*D*}) (*length* (*compE2 E*)) *None* ⟹ *False* **by**(*simp add*: *τmove2-iff*)
moreover
have *P*,*E*·*F*{*D*},*h* ⊢ (*THROW NullPointer*, *loc*) ↔ ([*Null*], *loc*, *length* (*compE2 E*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule bisim1-bisims1.bisim1FAccNull*)
ultimately show *?thesis* **using** *s* **by** *auto blast*
next
case (*FAcc1Throw a*)
note [*simp*] = ‹*e* = *Throw a*› ‹*h'* = *h*› ‹*xs'* = *xs*› ‹*ta* = *ε*› ‹*e'* = *Throw a*›
have *τ*: *τmove1 P h* (*e*·*F*{*D*}) **by**(*auto intro*: *τmove1FAccThrow*)
from *bisim* **have** *xcp* = ⌊*a*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*a*⌋
with *bisim* **have** *P*,*E*·*F*{*D*}, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
by(*auto intro*: *bisim1-bisims1.bisim1FAccThrow*)
thus *?thesis* **using** *τ* **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim* **obtain** *pc'*
where *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc'*, ⌊*a*⌋)
and *bisim'*: *P*, *E*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc'*, ⌊*a*⌋) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t* (*E*·*F*{*D*}) *h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc'*, ⌊*a*⌋)
by−(*rule FAcc-τExecrI*)
moreover from *bisim'* **have** *P*, *E*·*F*{*D*}, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc'*, ⌊*a*⌋)
by(*rule bisim1-bisims1.bisim1FAccThrow*)
ultimately show *?thesis* **using** *τ* **by** *auto*
qed
qed
next
case *bisim1FAccThrow* **thus** *?case* **by** *auto*
next
case *bisim1FAccNull* **thus** *?case* **by** *auto*
next
case (*bisim1FAss1 e1 n e1' xs stk loc pc xcp e2 F D*)
note *IH1* = *bisim1FAss1.IH*(*2*)
note *IH2* = *bisim1FAss1.IH*(*4*)
note *bisim1* = ‹*P*,*e1*,*h* ⊢ (*e1'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim2* = ‹⋀*xs*. *P*,*e2*,*h* ⊢ (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bsok* = ‹*bsok* (*e1*·*F*{*D*} := *e2*) *n*›
from ‹*True*,*P*,*t* ⊢*1* ⟨*e1'*·*F*{*D*} := *e2*,(*h*, *xs*)⟩ −*ta*→ ⟨*e'*,(*h'*, *xs'*)⟩› **show** *?case*
proof *cases*
case (*FAss1Red1 E'*)
note [*simp*] = ‹*e'* = *E'*·*F*{*D*} := *e2*›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e1'*,(*h*, *xs*)⟩ −*ta*→ ⟨*E'*,(*h'*, *xs'*)⟩›
from *red* **have** *τmove1 P h* (*e1'*·*F*{*D*} := *e2*) = *τmove1 P h e1'* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
moreover from *red* **have** *call1* (*e1'*·*F*{*D*} := *e2*) = *call1 e1'* **by** *auto*
moreover from *IH1*[*OF red*] *bsok*

obtain *pc'' stk'' loc'' xcp''* **where** *bisim*: *P,e1,h' ⊢ (E', xs') ↔ (stk'', loc'', pc'', xcp'')*
and *redo*: *?exec ta e1 e1' E' h stk loc pc xcp h' pc'' stk'' loc'' xcp''* **by** *auto*
from *bisim*
have *P,e1·F{D} := e2,h' ⊢ (E'·F{D} := e2, xs') ↔ (stk'', loc'', pc'', xcp'')*
by(*rule bisim1-bisims1.bisim1FAss1*)
moreover {
assume *no-call2 e1 pc*
hence *no-call2 (e1·F{D} := e2) pc ∨ pc = length (compE2 e1)* **by**(*auto simp add*: *no-call2-def*)
}
ultimately show *?thesis* **using** *redo*
by(*auto simp del*: *call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)(*blast intro*: *FAss-τExecrI1 FAss-τExectI1 exec-move-FAssI1*)+
next
case (*FAss1Red2 E' v*)
note [*simp*] = ‹*e1' = Val v*› ‹*e' = Val v·F{D} := E'*›
and *red* = ‹*True,P,t ⊢1 ⟨e2,(h, xs)⟩ −ta→ ⟨E',(h', xs')⟩*›
from *red* **have** *τ*: *τmove1 P h (Val v·F{D} := e2) = τmove1 P h e2* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim1* **have** *s*: *xcp = None xs = loc*
and *exec1*: *τExec-mover-a P t e1 h (stk, loc, pc, None) ([v], xs, length (compE2 e1), None)*
by(*auto dest*: *bisim1Val2D1*)
from *exec1* **have** *τExec-mover-a P t (e1·F{D} := e2) h (stk, loc, pc, None) ([v], xs, length (compE2 e1), None)*
by(*rule FAss-τExecrI1*)
moreover
from *IH2*[*OF red*] *bsok* **obtain** *pc'' stk'' loc'' xcp''*
where *bisim'*: *P,e2,h' ⊢ (E', xs') ↔ (stk'', loc'', pc'', xcp'')*
and *exec'*: *?exec ta e2 e2 E' h [] xs 0 None h' pc'' stk'' loc'' xcp''* **by** *auto*
have *?exec ta (e1·F{D} := e2) (Val v·F{D} := e2) (Val v·F{D} := E') h ([] @ [v]) xs (length (compE2 e1) + 0) None h' (length (compE2 e1) + pc'') (stk'' @ [v]) loc'' xcp''*
proof(*cases τmove1 P h (Val v·F{D} := e2)*)
case *True*
with *exec' τ* **have** [*simp*]: *h = h'* **and** *e*: *sim-move e2 E' P t e2 h ([], xs, 0, None) (stk'', loc'', pc'', xcp'')* **by** *auto*
from *e* **have** *sim-move (Val v·F{D} := e2) (Val v·F{D} := E') P t (e1·F{D} := e2) h ([] @ [v], xs, length (compE2 e1) + 0, None) (stk'' @ [v], loc'', length (compE2 e1) + pc'', xcp'')*
by(*fastforce dest*: *FAss-τExecrI2 FAss-τExectI2 simp del*: *compE2.simps compEs2.simps*)
with *True* **show** *?thesis* **by** *auto*
next
case *False*
with *exec' τ* **obtain** *pc' stk' loc' xcp'*
where *e*: *τExec-mover-a P t e2 h ([], xs, 0, None) (stk', loc', pc', xcp')*
and *e'*: *exec-move-a P t e2 h (stk', loc', pc', xcp') (extTA2JVM (compP2 P) ta) h' (stk'', loc'', pc'', xcp'')*
and *τ'*: *¬ τmove2 (compP2 P) h stk' e2 pc' xcp'*
and *call*: *call1 e2 = None ∨ no-call2 e2 0 ∨ pc' = 0 ∧ stk' = [] ∧ loc' = xs ∧ xcp' = None* **by** *auto*
from *e* **have** *τExec-mover-a P t (e1·F{D} := e2) h ([] @ [v], xs, length (compE2 e1) + 0, None) (stk' @ [v], loc', length (compE2 e1) + pc', xcp')*
by(*rule FAss-τExecrI2*)
moreover from *e'* **have** *exec-move-a P t (e1·F{D} := e2) h (stk' @ [v], loc', length (compE2 e1) + pc', xcp') (extTA2JVM (compP2 P) ta) h' (stk'' @ [v], loc'', length (compE2 e1) + pc'', xcp'')*
by(*rule exec-move-FAssI2*)
moreover from *e'* **have** *pc' < length (compE2 e2)* **by**(*auto elim*: *exec-meth.cases*)

with τ' e' **have** $\neg$ $\tau move2$ ($compP2$ P) h (stk' @ [v]) ($e1 \cdot F\{D\} := e2$) ($length$ ($compE2$ $e1$) + pc') xcp'
by(*auto simp add*: $\tau move2\text{-}iff$ $\tau instr\text{-}stk\text{-}drop\text{-}exec\text{-}move$)
moreover have $call1$ ($e1' \cdot F\{D\} := e2$) = $call1$ $e2$ **by** *simp*
moreover have $no\text{-}call2$ $e2$ 0 $\Longrightarrow$ $no\text{-}call2$ ($e1 \cdot F\{D\} := e2$) ($length$ ($compE2$ $e1$))
by(*auto simp add*: $no\text{-}call2\text{-}def$)
ultimately show *?thesis* **using** *False call*
by(*auto simp del*: *split-paired-Ex call1.simps calls1.simps*) *blast*
qed
moreover from $bisim'$
have P,$e1 \cdot F\{D\} := e2$,$h' \vdash$ (Val $v \cdot F\{D\} := E'$, xs') $\leftrightarrow$ ((stk'' @ [v]), loc'', $length$ ($compE2$ $e1$) + pc'', xcp'')
by(*rule bisim1-bisims1.bisim1FAss2*)
moreover from $bisim1$ **have** $pc \neq length$ ($compE2$ $e1$) $\longrightarrow$ $no\text{-}call2$ ($e1 \cdot F\{D\} := e2$) pc
by(*auto simp add*: *no-call2-def dest*: *bisim-Val-pc-not-Invoke bisim1-pc-length-compE2*)
ultimately show *?thesis* **using** τ $exec1$ s
apply(*auto simp del*: *split-paired-Ex call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)
apply(*blast intro*: $\tau Exec\text{-}mover\text{-}trans$|*fastforce elim*!: $\tau Exec\text{-}mover\text{-}trans$ *simp del*: *split-paired-Ex call1.simps calls1.simps*)+
done
next
case ($Red1FAss$ a v)
note [*simp*] = ‹$e1' = addr\ a$› ‹$e2 = Val\ v$› ‹$ta = \{\!|WriteMem\ a\ (CField\ D\ F)\ v|\!\}$› ‹$e' = unit$› ‹$xs' = xs$›
and $write$ = ‹$heap\text{-}write\ h\ a\ (CField\ D\ F)\ v\ h'$›
have τ: $\neg$ $\tau move1$ P h ($e1' \cdot F\{D\} := e2$) **by**(*auto simp add*: $\tau move1.simps$ $\tau moves1.simps$)
from $bisim1$ **have** s: $xcp = None$ $xs = loc$
and $\tau Exec\text{-}mover\text{-}a$ P t $e1$ h (stk, loc, pc, xcp) ([$Addr$ a], loc, $length$ ($compE2$ $e1$), $None$)
by(*auto dest*: *bisim1Val2D1*)
hence $\tau Exec\text{-}mover\text{-}a$ P t ($e1 \cdot F\{D\} := e2$) h (stk, loc, pc, xcp) ([$Addr$ a], loc, $length$ ($compE2$ $e1$), $None$)
by$-$(*rule FAss-*$\tau ExecrI1$)
also have $\tau move2$ ($compP2$ P) h [$Addr$ a] ($e1 \cdot F\{D\} := Val\ v$) ($length$ ($compE2$ $e1$)) $None$ **by**(*simp add*: $\tau move2\text{-}iff$)
hence $\tau Exec\text{-}mover\text{-}a$ P t ($e1 \cdot F\{D\} := e2$) h ([$Addr$ a], loc, $length$ ($compE2$ $e1$), $None$) ([v, $Addr$ a], loc, Suc ($length$ ($compE2$ $e1$)), $None$)
by$-$(*rule* $\tau Execr1step$, *auto intro*!: *exec-instr simp add*: *exec-move-def compP2-def*)
also ($rtranclp\text{-}trans$) **from** $write$
have $exec\text{-}move\text{-}a$ P t ($e1 \cdot F\{D\} := e2$) h ([v, $Addr$ a], loc, Suc ($length$ ($compE2$ $e1$)), $None$) $\{\!|WriteMem\ a\ (CField\ D\ F)\ v|\!\}$
h' ([], loc, Suc (Suc ($length$ ($compE2$ $e1$))), $None$)
unfolding *exec-move-def* **by**(*auto intro*!: *exec-instr*)
moreover have $\tau move2$ ($compP2$ P) h [v, $Addr$ a] ($e1 \cdot F\{D\} := e2$) (Suc ($length$ ($compE2$ $e1$))) $None \Longrightarrow False$
by(*simp add*: $\tau move2\text{-}iff$)
moreover
have P, $e1 \cdot F\{D\} := e2$, $h' \vdash$ ($unit$, loc) $\leftrightarrow$ ([], loc, Suc ($length$ ($compE2$ $e1$) + $length$ ($compE2$ $e2$)), $None$)
by(*rule bisim1-bisims1.bisim1FAss3*)
ultimately show *?thesis* **using** s τ **by**(*auto simp del*: *fun-upd-apply simp add*: *ta-upd-simps*) *blast*
next
case ($Red1FAssNull$ v)
note [*simp*] = ‹$e1' = null$› ‹$e2 = Val\ v$› ‹$xs' = xs$› ‹$ta = \varepsilon$› ‹$e' = THROW\ NullPointer$› ‹$h' = h$›
have τ: $\neg$ $\tau move1$ P h ($e1' \cdot F\{D\} := e2$) **by**(*auto simp add*: $\tau move1.simps$ $\tau moves1.simps$)

from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *xcp*) ([*Null*], *loc*, *length* (*compE2 e1*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1·F*{*D*} := *e2*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*Null*], *loc*, *length* (*compE2 e1*), *None*)
by−(*rule FAss-τExecrI1*)
also have *τmove2* (*compP2 P*) *h* [*Null*] (*e1·F*{*D*} := *Val v*) (*length* (*compE2 e1*)) *None* **by**(*simp add*: *τmove2-iff*)
hence *τExec-mover-a P t* (*e1·F*{*D*} := *e2*) *h* ([*Null*], *loc*, *length* (*compE2 e1*), *None*) ([*v*, *Null*], *loc*, *Suc* (*length* (*compE2 e1*)), *None*)
by−(*rule τExecr1step*, *auto intro*!: *exec-instr simp add*: *exec-move-def compP2-def*)
also (*rtranclp-trans*)
have *exec-move-a P t* (*e1·F*{*D*} := *e2*) *h* ([*v*, *Null*], *loc*, *Suc* (*length* (*compE2 e1*)), *None*) *ε* *h′* ([*v*, *Null*], *loc*, *Suc* (*length* (*compE2 e1*)), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*auto intro*!: *exec-instr simp add*: *exec-move-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v*, *Null*] (*e1·F*{*D*} := *e2*) (*Suc* (*length* (*compE2 e1*))) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *e1·F*{*D*} := *e2*, *h* ⊢ (*THROW NullPointer*, *loc*) ↔ ([*v*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule bisim1-bisims1.bisim1FAssNull*)
ultimately show *?thesis* **using** *s τ* **by**(*auto simp del*: *fun-upd-apply*) *blast*
next
case (*FAss1Throw1 a*)
note [*simp*] = ‹*e1′* = *Throw a*› ‹*ta* = *ε*› ‹*e′* = *Throw a*› ‹*h′* = *h*› ‹*xs′* = *xs*›
have *τ*: *τmove1 P h* (*Throw a·F*{*D*} := *e2*) **by**(*rule τmove1FAssThrow1*)
from *bisim1* **have** *xcp* = ⌊*a*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*a*⌋
with *bisim1*
have *P*, *e1·F*{*D*} := *e2*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
by(*auto intro*: *bisim1-bisims1.intros*)
thus *?thesis* **using** *τ* **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim1* **obtain** *pc′* **where** *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
and *bisim′*: *P*, *e1*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
and [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t* (*e1·F*{*D*} := *e2*) *h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
by−(*rule FAss-τExecrI1*)
moreover from *bisim′*
have *P*, *e1·F*{*D*} := *e2*, *h*⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
by(*rule bisim1-bisims1.bisim1FAssThrow1*)
ultimately show *?thesis* **using** *τ* **by** *auto*
qed
next
case (*FAss1Throw2 v ad*)
note [*simp*] = ‹*e1′* = *Val v*› ‹*e2* = *Throw ad*› ‹*e′* = *Throw ad*› ‹*h′* = *h*› ‹*xs′* = *xs*›
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 e1*), *None*)

by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1·F*{*D*} := *Throw ad*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 e1*), *None*)
by−(*rule FAss-τExecrI1*)
also have *τExec-mover-a P t* (*e1·F*{*D*} := *Throw ad*) *h* ([*v*], *loc*, *length* (*compE2 e1*), *None*) ([*Addr ad*, *v*], *loc*, *Suc* (*length* (*compE2 e1*)), ⌊*ad*⌋)
by(*rule τExecr2step*)(*auto simp add*: *exec-move-def exec-meth-instr τmove2-iff τmove1.simps τmoves1.simps*)
also (*rtranclp-trans*)
have *P*,*e1·F*{*D*}:=*Throw ad*,*h* ⊢ (*Throw ad*, *loc*) ↔ ([*Addr ad*] @ [*v*], *loc*, (*length* (*compE2 e1*) + *length* (*compE2* (*addr ad*))), ⌊*ad*⌋)
by(*rule bisim1FAssThrow2*[*OF bisim1Throw2*])
moreover have *τmove1 P h* (*FAss e1′ F D* (*Throw ad*)) **by**(*auto intro*: *τmove1FAssThrow2*)
ultimately show *?thesis* **using** *s* **by** *auto*
qed
next
case (*bisim1FAss2 e2 n e2′ xs stk loc pc xcp e1 F D v1*)
note *IH2* = *bisim1FAss2.IH*(*2*)
note *bisim1* = ‹⋀*xs*. *P*,*e1*,*h* ⊢ (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e2′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bsok* = ‹*bsok* (*e1·F*{*D*} := *e2*) *n*›
note *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*Val v1·F*{*D*} := *e2′*,(*h*, *xs*)⟩ −*ta*→ ⟨*e′*,(*h′*, *xs′*)⟩›
from *red* **show** *?case*
proof *cases*
case (*FAss1Red2 E′*)
note [*simp*] = ‹*e′* = *Val v1·F*{*D*} := *E′*›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e2′*,(*h*, *xs*)⟩ −*ta*→ ⟨*E′*,(*h′*, *xs′*)⟩›
from *IH2*[*OF red*] *bsok* **obtain** *pc″ stk″ loc″ xcp″*
where *bisim′*: *P*,*e2*,*h′* ⊢ (*E′*, *xs′*) ↔ (*stk″*, *loc″*, *pc″*, *xcp″*)
and *exec′*: *?exec ta e2 e2′ E′ h stk loc pc xcp h′ pc″ stk″ loc″ xcp″* **by** *auto*
from *red* **have** *τ*: *τmove1 P h* (*Val v1·F*{*D*} := *e2′*) = *τmove1 P h e2′* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
have *no-call2 e2 pc* ⟹ *no-call2* (*e1·F*{*D*} := *e2*) (*length* (*compE2 e1*) + *pc*) **by**(*auto simp add*: *no-call2-def*)
hence *?exec ta* (*e1·F*{*D*} := *e2*) (*Val v1·F*{*D*} := *e2′*) (*Val v1·F*{*D*} := *E′*) *h* (*stk* @ [*v1*]) *loc* (*length* (*compE2 e1*) + *pc*) *xcp h′* (*length* (*compE2 e1*) + *pc″*) (*stk″* @ [*v1*]) *loc″ xcp″*
using *exec′ τ*
apply(*cases τmove1 P h* (*Val v1·F*{*D*} := *e2′*))
apply(*auto*)
apply(*blast intro*: *FAss-τExecrI2 FAss-τExectI2 exec-move-FAssI2*)
apply(*blast intro*: *FAss-τExecrI2 FAss-τExectI2 exec-move-FAssI2*)
apply(*rule exI conjI FAss-τExecrI2 exec-move-FAssI2*|*assumption*)+
apply(*fastforce simp add*: *τinstr-stk-drop-exec-move τmove2-iff split*: *if-split-asm*)
apply(*rule exI conjI FAss-τExecrI2 exec-move-FAssI2*|*assumption*)+
apply(*fastforce simp add*: *τinstr-stk-drop-exec-move τmove2-iff split*: *if-split-asm*)
apply(*rule exI conjI FAss-τExecrI2 exec-move-FAssI2 rtranclp.rtrancl-refl*|*assumption*)+
apply(*fastforce simp add*: *τinstr-stk-drop-exec-move τmove2-iff split*: *if-split-asm*)+
done
moreover from *bisim′*
have *P*,*e1·F*{*D*} := *e2*,*h′* ⊢ (*Val v1·F*{*D*} := *E′*, *xs′*) ↔ (*stk″*@[*v1*], *loc″*, *length* (*compE2 e1*) + *pc″*, *xcp″*)
by(*rule bisim1-bisims1.bisim1FAss2*)
ultimately show *?thesis* **using** *τ* **by** *auto blast*+
next

case (*Red1FAss a v*)
note [*simp*] = ‹*v1* = *Addr a*› ‹*e2'* = *Val v*› ‹*ta* = {|*WriteMem a* (*CField D F*) *v*|}› ‹*e'* = *unit*› ‹*xs'* = *xs*›
and *ha* = ‹*heap-write h a* (*CField D F*) *v h'*›
have *τ*: ¬ *τmove1 P h* (*addr a*·*F*{*D*} := *e2'*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 e2*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*·*F*{*D*} := *e2*) *h* (*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) ([*v*] @ [*v1*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
by−(*rule FAss-τExecrI2*)
moreover from *ha*
have *exec-move-a P t* (*e1*·*F*{*D*} := *e2*) *h* ([*v*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*) {|*WriteMem a* (*CField D F*) *v*|}
h' ([], *loc*, *Suc* (*length* (*compE2 e1*) + *length* (*compE2 e2*)), *None*)
by(*auto intro*!: *exec-instr simp add*: *exec-move-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v*, *Addr a*] (*e1*·*F*{*D*} := *e2*) (*length* (*compE2 e1*) + *length* (*compE2 e2*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *e1*·*F*{*D*} := *e2*, *h'* ⊢ (*unit*, *loc*) ↔ ([], *loc*, *Suc* (*length* (*compE2 e1*) + *length* (*compE2 e2*)), *None*)
by(*rule bisim1-bisims1.bisim1FAss3*)
ultimately show *?thesis* **using** *s τ* **by**(*auto simp del*: *fun-upd-apply simp add*: *ta-upd-simps*) *blast*
next
case (*Red1FAssNull v*)
note [*simp*] = ‹*v1* = *Null*› ‹*e2'* = *Val v*› ‹*xs'* = *xs*› ‹*ta* = *ε*› ‹*e'* = *THROW NullPointer*› ‹*h'* = *h*›
have *τ*: ¬ *τmove1 P h* (*null*·*F*{*D*} := *e2'*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 e2*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*·*F*{*D*} := *e2*) *h* (*stk* @ [*Null*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) ([*v*] @ [*Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
by−(*rule FAss-τExecrI2*)
moreover have *exec-move-a P t* (*e1*·*F*{*D*} := *e2*) *h* ([*v*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*) *ε*
h' ([*v*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*auto intro*!: *exec-instr simp add*: *exec-move-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v*, *Null*] (*e1*·*F*{*D*} := *e2*) (*length* (*compE2 e1*) + *length* (*compE2 e2*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *e1*·*F*{*D*} := *e2*, *h* ⊢ (*THROW NullPointer*, *loc*) ↔ ([*v*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule bisim1-bisims1.bisim1FAssNull*)
ultimately show *?thesis* **using** *s τ* **by**(*auto simp del*: *fun-upd-apply*) *blast*
next
case (*FAss1Throw2 a*)
note [*simp*] = ‹*e2'* = *Throw a*› ‹*ta* = *ε*› ‹*h'* = *h*› ‹*xs'* = *xs*› ‹*e'* = *Throw a*›
have *τ*: *τmove1 P h* (*FAss* (*Val v1*) *F D* (*Throw a*)) **by**(*rule τmove1FAssThrow2*)
from *bisim2* **have** *xcp* = ⌊*a*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof

```
      assume [simp]: xcp = ⌊a⌋
      with bisim2
      have P, e1·F{D} := e2, h ⊢ (Throw a, xs) ↔ (stk @ [v1], loc, length (compE2 e1) + pc, xcp)
        by(auto intro: bisim1FAssThrow2)
      thus ?thesis using τ by(fastforce)
    next
      assume [simp]: xcp = None
      with bisim2 obtain pc'
        where τExec-mover-a P t e2 h (stk, loc, pc, None) ([Addr a], loc, pc', ⌊a⌋)
        and bisim': P, e2, h ⊢ (Throw a, xs) ↔ ([Addr a], loc, pc', ⌊a⌋) and [simp]: xs = loc
        by(auto dest: bisim1-Throw-τExec-mover)
      hence τExec-mover-a P t (e1·F{D} := e2) h (stk @ [v1], loc, length (compE2 e1) + pc, None)
([Addr a] @ [v1], loc, length (compE2 e1) + pc', ⌊a⌋)
        by−(rule FAss-τExecrI2)
      moreover from bisim'
      have P, e1·F{D} := e2, h ⊢ (Throw a, xs) ↔ ([Addr a]@[v1], loc, length (compE2 e1) + pc',
⌊a⌋)
        by−(rule bisim1FAssThrow2, auto)
      ultimately show ?thesis using τ by auto
    qed
  qed auto
next
  case bisim1FAssThrow1 thus ?case by fastforce
next
  case bisim1FAssThrow2 thus ?case by fastforce
next
  case bisim1FAssNull thus ?case by fastforce
next
  case bisim1FAss3 thus ?case by fastforce
next
  case (bisim1CAS1 e1 n e1' xs stk loc pc xcp e2 e3 D F)
  note IH1 = bisim1CAS1.IH(2)
  note IH2 = bisim1CAS1.IH(4)
  note IH3 = bisim1CAS1.IH(6)
  note bisim1 = ‹P,e1,h ⊢ (e1', xs) ↔ (stk, loc, pc, xcp)›
  note bisim2 = ‹⋀xs. P,e2,h ⊢ (e2, xs) ↔ ([], xs, 0, None)›
  note bisim3 = ‹⋀xs. P,e3,h ⊢ (e3, xs) ↔ ([], xs, 0, None)›
  note bsok = ‹bsok - n›
  from ‹True,P,t ⊢1 ⟨-,(h, xs)⟩ −ta→ ⟨e',(h', xs')⟩› show ?case
  proof cases
    case (CAS1Red1 E')
    note [simp] = ‹e' = E'·compareAndSwap(D·F, e2, e3)›
      and red = ‹True,P,t ⊢1 ⟨e1',(h, xs)⟩ −ta→ ⟨E',(h', xs')⟩›
    from red have τmove1 P h (e1'·compareAndSwap(D·F, e2, e3)) = τmove1 P h e1' by(auto simp
add: τmove1.simps τmoves1.simps)
    moreover from red have call1 (e1'·compareAndSwap(D·F, e2, e3)) = call1 e1' by auto
    moreover from IH1[OF red] bsok
    obtain pc'' stk'' loc'' xcp'' where bisim: P,e1,h' ⊢ (E', xs') ↔ (stk'', loc'', pc'', xcp'')
      and redo: ?exec ta e1 e1' E' h stk loc pc xcp h' pc'' stk'' loc'' xcp'' by auto
    from bisim
    have P,e1·compareAndSwap(D·F, e2, e3),h' ⊢ (E'·compareAndSwap(D·F, e2, e3), xs') ↔ (stk'',
loc'', pc'', xcp'')
      by(rule bisim1-bisims1.bisim1CAS1)
    moreover {
```

assume *no-call2 e1 pc*
hence *no-call2* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *pc* ∨ *pc* = *length* (*compE2 e1*) **by**(*auto simp add*: *no-call2-def*) **}**
ultimately show *?thesis* **using** *redo*
by(*auto simp del*: *call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)(*blast intro*: *CAS-τExecrI1 CAS-τExectI1 exec-move-CASI1*)+
next
case (*CAS1Red2 E′ v*)
note [*simp*] = ‹*e1′* = *Val v*› ‹*e′* = *Val v*·*compareAndSwap*(*D*·*F*, *E′*, *e3*)›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e2*,(*h*, *xs*)⟩ −*ta*→ ⟨*E′*,(*h′*, *xs′*)⟩›
from *red* **have** *τ*: *τmove1 P h* (*Val v*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) = *τmove1 P h e2* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *exec1*: *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *None*) ([*v*], *xs*, *length* (*compE2 e1*), *None*)
by(*auto dest*: *bisim1Val2D1*)
from *exec1* **have** *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk*, *loc*, *pc*, *None*) ([*v*], *xs*, *length* (*compE2 e1*), *None*)
by(*rule CAS-τExecrI1*)
moreover
from *IH2*[*OF red*] *bsok* **obtain** *pc″ stk″ loc″ xcp″*
where *bisim′*: *P*,*e2*,*h′* ⊢ (*E′*, *xs′*) ↔ (*stk″*, *loc″*, *pc″*, *xcp″*)
and *exec′*: *?exec ta e2 e2 E′ h* [] *xs 0 None h′ pc″ stk″ loc″ xcp″* **by** *auto*
have *?exec ta* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*Val v*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*Val v*·*compareAndSwap*(*D*·*F*, *E′*, *e3*)) *h* ([] @ [*v*]) *xs* (*length* (*compE2 e1*) + *0*) *None h′* (*length* (*compE2 e1*) + *pc″*) (*stk″* @ [*v*]) *loc″ xcp″*
proof(*cases τmove1 P h* (*Val v*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)))
case *True*
with *exec′ τ* **have** [*simp*]: *h* = *h′* **and** *e*: *sim-move e2 E′ P t e2 h* ([], *xs*, *0*, *None*) (*stk″*, *loc″*, *pc″*, *xcp″*) **by** *auto*
from *e* **have** *sim-move* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*e1*·*compareAndSwap*(*D*·*F*, *E′*, *e3*)) *P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([] @ [*v*], *xs*, *length* (*compE2 e1*) + *0*, *None*) (*stk″* @ [*v*], *loc″*, *length* (*compE2 e1*) + *pc″*, *xcp″*)
by(*fastforce dest*: *CAS-τExecrI2 CAS-τExectI2*)
with *True* **show** *?thesis* **by** *auto*
next
case *False*
with *exec′ τ* **obtain** *pc′ stk′ loc′ xcp′*
where *e*: *τExec-mover-a P t e2 h* ([], *xs*, *0*, *None*) (*stk′*, *loc′*, *pc′*, *xcp′*)
and *e′*: *exec-move-a P t e2 h* (*stk′*, *loc′*, *pc′*, *xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk″*, *loc″*, *pc″*, *xcp″*)
and *τ′*: ¬ *τmove2* (*compP2 P*) *h stk′ e2 pc′ xcp′*
and *call*: *call1 e2* = *None* ∨ *no-call2 e2 0* ∨ *pc′* = *0* ∧ *stk′* = [] ∧ *loc′* = *xs* ∧ *xcp′* = *None* **by** *auto*
from *e* **have** *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([] @ [*v*], *xs*, *length* (*compE2 e1*) + *0*, *None*) (*stk′* @ [*v*], *loc′*, *length* (*compE2 e1*) + *pc′*, *xcp′*) **by**(*rule CAS-τExecrI2*)
moreover from *e′* **have** *exec-move-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk′* @ [*v*], *loc′*, *length* (*compE2 e1*) + *pc′*, *xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk″* @ [*v*], *loc″*, *length* (*compE2 e1*) + *pc″*, *xcp″*)
by(*rule exec-move-CASI2*)
moreover from *e′* **have** *pc′* < *length* (*compE2 e2*) **by**(*auto elim*: *exec-meth.cases*)
with *τ′ e′* **have** ¬ *τmove2* (*compP2 P*) *h* (*stk′* @ [*v*]) (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *pc′*) *xcp′*
by(*auto simp add*: *τinstr-stk-drop-exec-move τmove2-iff*)
moreover from *red* **have** *call1* (*e1′*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) = *call1 e2* **by** *auto*

moreover have *no-call2 e2 0* $\Longrightarrow$ *no-call2* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) (*length* (*compE2 e1*))
by(*auto simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *False call*
by(*auto simp del*: *split-paired-Ex call1.simps calls1.simps*) *blast*
qed
moreover from *bisim'*
have *P*,*e1·compareAndSwap*(*D·F*, *e2*, *e3*),*h'* ⊢ (*Val v·compareAndSwap*(*D·F*, *E'*, *e3*), *xs'*) ↔ ((*stk''* @ [*v*]), *loc''*, *length* (*compE2 e1*) + *pc''*, *xcp''*)
by(*rule bisim1-bisims1.bisim1CAS2*)
moreover from *bisim1* **have** *pc* ≠ *length* (*compE2 e1*) ⟶ *no-call2* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) *pc*
by(*auto simp add*: *no-call2-def dest*: *bisim-Val-pc-not-Invoke bisim1-pc-length-compE2*)
ultimately show *?thesis* **using** *τ exec1 s*
apply(*auto simp del*: *split-paired-Ex call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)
apply(*blast intro*: *τExec-mover-trans*|*fastforce elim*!: *τExec-mover-trans simp del*: *split-paired-Ex call1.simps calls1.simps*)+
done
next
case (*CAS1Red3 E' v v'*)
note [*simp*] = ‹*e2* = *Val v'*› ‹*e1'* = *Val v*› ‹*e'* = *Val v·compareAndSwap*(*D·F*, *Val v'*, *E'*)›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e3*,(*h*, *xs*)⟩ −*ta*→ ⟨*E'*,(*h'*, *xs'*)⟩›
from *red* **have** *τ*: *τmove1 P h* (*Val v·compareAndSwap*(*D·F*, *Val v'*, *e3*)) = *τmove1 P h e3* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *exec1*: *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *None*) ([] @ [*v*], *xs*, *length* (*compE2 e1*) + *0*, *None*)
by(*auto dest*: *bisim1Val2D1*)
from *exec1* **have** *τExec-mover-a P t* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) *h* (*stk*, *loc*, *pc*, *None*) ([] @ [*v*], *xs*, *length* (*compE2 e1*) + *0*, *None*)
by(*rule CAS-τExecrI1*)
also from *bisim2*[*of xs*]
have *τExec-mover-a P t e2 h* ([], *xs*, *0*, *None*) ([*v'*], *xs*, *length* (*compE2 e2*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) *h* ([] @ [*v*], *xs*, *length* (*compE2 e1*) + *0*, *None*) ([*v'*] @ [*v*], *xs*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
by(*rule CAS-τExecrI2*)
also (*rtranclp-trans*) **from** *IH3*[*OF red*] *bsok* **obtain** *pc''* *stk''* *loc''* *xcp''*
where *bisim'*: *P*,*e3*,*h'* ⊢ (*E'*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
and *exec'*: *?exec ta e3 e3 E' h* [] *xs 0 None h' pc'' stk'' loc'' xcp''* **by** *auto*
have *?exec ta* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) (*Val v·compareAndSwap*(*D·F*, *Val v'*, *e3*)) (*Val v·compareAndSwap*(*D·F*, *Val v'*, *E'*)) *h* ([] @ [*v'*, *v*]) *xs* (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *0*) *None h'* (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc''*) (*stk''* @ [*v'*, *v*]) *loc'' xcp''*
proof(*cases τmove1 P h* (*Val v·compareAndSwap*(*D·F*, *Val v'*, *e3*)))
case *True*
with *exec' τ* **have** [*simp*]: *h* = *h'* **and** *e*: *sim-move e3 E' P t e3 h* ([], *xs*, *0*, *None*) (*stk''*, *loc''*, *pc''*, *xcp''*) **by** *auto*
from *e* **have** *sim-move* (*Val v·compareAndSwap*(*D·F*, *Val v'*, *e3*)) (*Val v·compareAndSwap*(*D·F*, *Val v'*, *E'*)) *P t* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) *h* ([] @ [*v'*, *v*], *xs*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *0*, *None*) (*stk''* @ [*v'*, *v*], *loc''*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc''*, *xcp''*)
by(*fastforce dest*: *CAS-τExectI3 CAS-τExecrI3 simp del*: *compE2.simps compEs2.simps*)
with *True* **show** *?thesis* **by** *auto*
next
case *False*

with *exec′ τ* **obtain** *pc′ stk′ loc′ xcp′*
where *e*: *τExec-mover-a P t e3 h* ([], *xs*, *0*, *None*) (*stk′*, *loc′*, *pc′*, *xcp′*)
and *e′*: *exec-move-a P t e3 h* (*stk′*, *loc′*, *pc′*, *xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk″*, *loc″*, *pc″*, *xcp″*)
and *τ′*: ¬ *τmove2* (*compP2 P*) *h stk′ e3 pc′ xcp′*
and *call*: *call1 e3* = *None* ∨ *no-call2 e3 0* ∨ *pc′* = *0* ∧ *stk′* = [] ∧ *loc′* = *xs* ∧ *xcp′* = *None* **by** *auto*
from *e* **have** *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([] @ [*v′*, *v*], *xs*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *0*, *None*) (*stk′* @ [*v′*, *v*], *loc′*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc′*, *xcp′*) **by**(*rule CAS-τExecrI3*)
moreover from *e′* **have** *exec-move-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk′* @ [*v′*, *v*], *loc′*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc′*, *xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk″* @ [*v′*, *v*], *loc″*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc″*, *xcp″*)
by(*rule exec-move-CASI3*)
moreover from *e′ τ′*
have ¬ *τmove2* (*compP2 P*) *h* (*stk′* @ [*v′*, *v*]) (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc′*) *xcp′*
by(*auto simp add*: *τinstr-stk-drop-exec-move τmove2-iff*)
moreover have *call1* (*e1′*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) = *call1 e3* **by** *simp*
moreover have *no-call2 e3 0* ⟹ *no-call2* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *length* (*compE2 e2*))
by(*auto simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *False call*
by(*auto simp del*: *split-paired-Ex call1.simps calls1.simps*) *blast*
qed
moreover from *bisim′*
have *P*,*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*),*h′* ⊢ (*Val v*·*compareAndSwap*(*D*·*F*, *Val v′*, *E′*), *xs′*) ↔ ((*stk″* @ [*v′*, *v*]), *loc″*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc″*, *xcp″*)
by(*rule bisim1-bisims1.bisim1CAS3*)
moreover from *bisim1* **have** *pc* ≠ *length* (*compE2 e1*) + *length* (*compE2 e2*) ⟶ *no-call2* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *pc*
by(*auto simp add*: *no-call2-def dest*: *bisim-Val-pc-not-Invoke bisim1-pc-length-compE2*)
ultimately show *?thesis* **using** *τ exec1 s*
apply(*auto simp del*: *split-paired-Ex call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)
apply(*blast intro*: *τExec-mover-trans*|*fastforce elim*!: *τExec-mover-trans simp del*: *split-paired-Ex call1.simps calls1.simps*)+
done
next
case (*CAS1Null v v′*)
note [*simp*] = ‹*e1′* = *null*› ‹*e′* = *THROW NullPointer*› ‹*e2* = *Val v*› ‹*xs′* = *xs*› ‹*ta* = *ε*› ‹*h′* = *h*› ‹*e3* = *Val v′*›
have *τ*: ¬ *τmove1 P h* (*AAss null* (*Val v*) (*Val v′*)) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *xcp*) ([] @ [*Null*], *loc*, *length* (*compE2 e1*) + *0*, *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk*, *loc*, *pc*, *xcp*) ([] @ [*Null*], *loc*, *length* (*compE2 e1*) + *0*, *None*)
by−(*rule CAS-τExecrI1*)
also from *bisim2*[*of loc*] **have** *τExec-mover-a P t e2 h* ([], *loc*, *0*, *None*) ([*v*], *loc*, *length* (*compE2 e2*) + *0*, *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([] @ [*Null*], *loc*, *length* (*compE2 e1*) + *0*, *None*) ([*v*] @ [*Null*], *loc*, *length* (*compE2 e1*) + (*length* (*compE2 e2*) + *0*), *None*)
by(*rule CAS-τExecrI2*)

also (*rtranclp-trans*) **have** [*v*] @ [*Null*] = [] @ [*v*, *Null*] **by** *simp*
also note *add.assoc*[*symmetric*]
also from *bisim3*[*of loc*] **have** *τExec-mover-a P t e3 h* ([], *loc*, *0*, *None*) ([*v'*], *loc*, *length* (*compE2 e3*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([] @ [*v*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *0*, *None*) ([*v'*] @ [*v*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*)
by(*rule CAS-τExecrI3*)
also (*rtranclp-trans*)
have *exec-move-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([*v'*, *v*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*) *ε*
h ([*v'*, *v*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *is-Ref-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v'*, *v*, *Null*] (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *h'* ⊢ (*THROW NullPointer*, *loc*) ↔ ([*v'*, *v*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule bisim1-bisims1.bisim1CASFail*)
ultimately show *?thesis* **using** *s τ* **by**(*auto simp add*: *τmove1.simps*) *blast*
next
case (*Red1CASSucceed a v v'*)
hence [*simp*]: *e1' = addr a e' = true e2 = Val v*
ta = {⌈*ReadMem a* (*CField D F*) *v*, *WriteMem a* (*CField D F*) *v'*⌉} *xs' = xs e3 = Val v'*
and *read*: *heap-read h a* (*CField D F*) *v*
and *write*: *heap-write h a* (*CField D F*) *v' h'* **by** *auto*
have *τ*: ¬ *τmove1 P h* (*CompareAndSwap* (*addr a*) *D F* (*Val v*) (*Val v'*)) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim1* **have** *s*: *xcp = None xs = loc*
and *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *xcp*) ([] @ [*Addr a*], *loc*, *length* (*compE2 e1*) + *0*, *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk*, *loc*, *pc*, *xcp*) ([] @ [*Addr a*], *loc*, *length* (*compE2 e1*) + *0*, *None*)
by−(*rule CAS-τExecrI1*)
also from *bisim2*[*of loc*]
have *τExec-mover-a P t e2 h* ([], *loc*, *0*, *None*) ([*v*], *loc*, *length* (*compE2 e2*) + *0*, *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([] @ [*Addr a*], *loc*, *length* (*compE2 e1*) + *0*, *None*) ([*v*] @ [*Addr a*], *loc*, *length* (*compE2 e1*) + (*length* (*compE2 e2*) + *0*), *None*)
by(*rule CAS-τExecrI2*)
also (*rtranclp-trans*) **have** [*v*] @ [*Addr a*] = [] @ [*v*, *Addr a*] **by** *simp*
also note *add.assoc*[*symmetric*]
also from *bisim3*[*of loc*] **have** *τExec-mover-a P t e3 h* ([], *loc*, *0*, *None*) ([*v'*], *loc*, *length* (*compE2 e3*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([] @ [*v*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *0*, *None*) ([*v'*] @ [*v*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*)
by(*rule CAS-τExecrI3*)
also (*rtranclp-trans*) **from** *read write*
have *exec-move-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([*v'*, *v*, *Addr a*], *loc*, *length* (*compE2*

e1) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*)
{⟦ *ReadMem a* (*CField D F*) *v*, *WriteMem a* (*CField D F*) *v'* ⟧}
h' ([*Bool True*], *loc*, *Suc* (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*)), *None*)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *compP2-def is-Ref-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v'*, *v*, *Addr a*] (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *h'* ⊢ (*true*, *loc*) ↔ ([*Bool True*], *loc*, *length* (*compE2* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*))), *None*)
by(*rule bisim1Val2*) *simp*
ultimately show *?thesis* **using** *s τ* **by**(*auto simp add*: *ta-upd-simps*) *blast*
next
case (*Red1CASFail a v'' v v'*)
hence [*simp*]: *e1'* = *addr a e'* = *false e2* = *Val v h'* = *h*
ta = {⟦*ReadMem a* (*CField D F*) *v''*⟧} *xs'* = *xs e3* = *Val v'*
and *read*: *heap-read h a* (*CField D F*) *v''* *v* ≠ *v''* **by** *auto*
have *τ*: ¬ *τmove1 P h* (*CompareAndSwap* (*addr a*) *D F* (*Val v*) (*Val v'*)) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *xcp*) ([] @ [*Addr a*], *loc*, *length* (*compE2 e1*) + *0*, *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk*, *loc*, *pc*, *xcp*) ([] @ [*Addr a*], *loc*, *length* (*compE2 e1*) + *0*, *None*)
by−(*rule CAS-τExecrI1*)
also from *bisim2*[*of loc*]
have *τExec-mover-a P t e2 h* ([], *loc*, *0*, *None*) ([*v*], *loc*, *length* (*compE2 e2*) + *0*, *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([] @ [*Addr a*], *loc*, *length* (*compE2 e1*) + *0*, *None*) ([*v*] @ [*Addr a*], *loc*, *length* (*compE2 e1*) + (*length* (*compE2 e2*) + *0*), *None*)
by(*rule CAS-τExecrI2*)
also (*rtranclp-trans*) **have** [*v*] @ [*Addr a*] = [] @ [*v*, *Addr a*] **by** *simp*
also note *add.assoc*[*symmetric*]
also from *bisim3*[*of loc*] **have** *τExec-mover-a P t e3 h* ([], *loc*, *0*, *None*) ([*v'*], *loc*, *length* (*compE2 e3*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([] @ [*v*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *0*, *None*) ([*v'*] @ [*v*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*)
by(*rule CAS-τExecrI3*)
also (*rtranclp-trans*) **from** *read*
have *exec-move-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([*v'*, *v*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*)
{⟦ *ReadMem a* (*CField D F*) *v''* ⟧}
h ([*Bool False*], *loc*, *Suc* (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*)), *None*)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *compP2-def is-Ref-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v'*, *v*, *Addr a*] (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *h* ⊢ (*false*, *loc*) ↔ ([*Bool False*], *loc*, *length* (*compE2* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*))), *None*)

by(*rule bisim1Val2*) *simp*
ultimately show *?thesis* **using** *s τ* **by**(*auto simp add*: *ta-upd-simps*)*blast*
next
case (*CAS1Throw a*)
hence [*simp*]: *e1'* = *Throw a ta* = *ε e'* = *Throw a h'* = *h xs'* = *xs* **by** *auto*
have *τ*: *τmove1 P h* (*Throw a*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) **by**(*rule τmove1CASThrow1*)
from *bisim1* **have** *xcp* = ⌊*a*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*a*⌋
with *bisim1* **have** *P*, *e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
by(*auto intro*: *bisim1-bisims1.intros*)
thus *?thesis* **using** *τ* **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim1* **obtain** *pc'* **where** *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc'*, ⌊*a*⌋)
and *bisim'*: *P*, *e1*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc'*, ⌊*a*⌋)
and [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc'*, ⌊*a*⌋)
by−(*rule CAS-τExecrI1*)
moreover from *bisim'*
have *P*, *e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc'*, ⌊*a*⌋)
by(*auto intro*: *bisim1-bisims1.bisim1CASThrow1*)
ultimately show *?thesis* **using** *τ* **by** *auto*
qed
next
case (*CAS1Throw2 v ad*)
note [*simp*] = ‹*e1'* = *Val v*› ‹*e2* = *Throw ad*› ‹*ta* = *ε*› ‹*e'* = *Throw ad*› ‹*h'* = *h*› ‹*xs'* = *xs*›
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 e1*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *Throw ad*, *e3*)) *h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 e1*), *None*)
by−(*rule CAS-τExecrI1*)
also have *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *Throw ad*, *e3*)) *h* ([*v*], *loc*, *length* (*compE2 e1*), *None*) ([*Addr ad*, *v*], *loc*, *Suc* (*length* (*compE2 e1*)), ⌊*ad*⌋)
by(*rule τExecr2step*)(*auto simp add*: *exec-move-def exec-meth-instr τmove2-iff τmove1.simps τmoves1.simps*)
also (*rtranclp-trans*)
have *P*,*e1*·*compareAndSwap*(*D*·*F*, *Throw ad*, *e3*),*h* ⊢ (*Throw ad*, *loc*) ↔ ([*Addr ad*] @ [*v*], *loc*, (*length* (*compE2 e1*) + *length* (*compE2* (*addr ad*))), ⌊*ad*⌋)
by(*rule bisim1CASThrow2*[*OF bisim1Throw2*])
moreover have *τmove1 P h* (*e1'*·*compareAndSwap*(*D*·*F*, *Throw ad*, *e3*)) **by**(*auto intro*: *τmove1CASThrow2*)
ultimately show *?thesis* **using** *s* **by** *auto*
next
case (*CAS1Throw3 v v' ad*)
note [*simp*] = ‹*e1'* = *Val v*› ‹*e2* = *Val v'*› ‹*e3* = *Throw ad*› ‹*ta* = *ε*› ‹*e'* = *Throw ad*› ‹*h'* = *h*› ‹*xs'* = *xs*›
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 e1*), *None*)
by(*auto dest*: *bisim1Val2D1*)

hence *τExec-mover-a P t (e1·compareAndSwap(D·F, e2, Throw ad)) h (stk, loc, pc, xcp) ([v], loc, length (compE2 e1), None)*
by−(*rule CAS-τExecrI1*)
also from *bisim2*[*of loc*] **have** *τExec-mover-a P t e2 h ([], loc, 0, None) ([v′], loc, length (compE2 e2), None)*
by(*auto dest*: *bisim1Val2D1*)
from *CAS-τExecrI2*[*OF this, of e1 D F e3 v*]
have *τExec-mover-a P t (e1·compareAndSwap(D·F, e2, Throw ad)) h ([v], loc, length (compE2 e1), None) ([v′, v], loc, length (compE2 e1) + length (compE2 e2), None)* **by** *simp*
also (*rtranclp-trans*)
have *τExec-mover-a P t (e1·compareAndSwap(D·F, e2, Throw ad)) h ([v′, v], loc, length (compE2 e1) + length (compE2 e2), None) ([Addr ad, v′, v], loc, Suc (length (compE2 e1) + length (compE2 e2)), ⌊ad⌋)*
by(*rule τExecr2step*)(*auto simp add*: *exec-move-def exec-meth-instr τmove2-iff τmove1.simps τmoves1.simps*)
also (*rtranclp-trans*)
have *P,e1·compareAndSwap(D·F, e2, Throw ad),h ⊢ (Throw ad, loc) ↔ ([Addr ad] @ [v′, v], loc, (length (compE2 e1) + length (compE2 e2) + length (compE2 (addr ad))), ⌊ad⌋)*
by(*rule bisim1CASThrow3*[*OF bisim1Throw2*])
moreover have *τmove1 P h (Val v·compareAndSwap(D·F, Val v′, Throw ad))* **by**(*auto intro*: *τmove1CASThrow3*)
ultimately show *?thesis* **using** *s* **by** *auto*
qed
next
case (*bisim1CAS2 e2 n e2′ xs stk loc pc xcp e1 e3 D F v1*)
note *IH2* = *bisim1CAS2.IH*(*2*)
note *IH3* = *bisim1CAS2.IH*(*6*)
note *bisim2* = ‹*P,e2,h ⊢ (e2′, xs) ↔ (stk, loc, pc, xcp)*›
note *bisim1* = ‹⋀*xs. P,e1,h ⊢ (e1, xs) ↔ ([], xs, 0, None)*›
note *bisim3* = ‹⋀*xs. P,e3,h ⊢ (e3, xs) ↔ ([], xs, 0, None)*›
note *bsok* = ‹*bsok (e1·compareAndSwap(D·F, e2, e3)) n*›
from ‹*True,P,t ⊢1 ⟨Val v1·compareAndSwap(D·F, e2′, e3),(h, xs)⟩ −ta→ ⟨e′,(h′, xs′)⟩*› **show** *?case*
proof *cases*
case (*CAS1Red2 E′*)
note [*simp*] = ‹*e′ = Val v1·compareAndSwap(D·F, E′, e3)*›
and *red* = ‹*True,P,t ⊢1 ⟨e2′,(h, xs)⟩ −ta→ ⟨E′,(h′, xs′)⟩*›
from *red* **have** *τ*: *τmove1 P h (Val v1·compareAndSwap(D·F, e2′, e3)) = τmove1 P h e2′* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *IH2*[*OF red*] *bsok* **obtain** *pc″ stk″ loc″ xcp″*
where *bisim′*: *P,e2,h′ ⊢ (E′, xs′) ↔ (stk″, loc″, pc″, xcp″)*
and *exec′*: *?exec ta e2 e2′ E′ h stk loc pc xcp h′ pc″ stk″ loc″ xcp″* **by** *auto*
have *?exec ta (e1·compareAndSwap(D·F, e2, e3)) (Val v1·compareAndSwap(D·F, e2′, e3)) (Val v1·compareAndSwap(D·F, E′, e3)) h (stk @ [v1]) loc (length (compE2 e1) + pc) xcp h′ (length (compE2 e1) + pc″) (stk″ @ [v1]) loc″ xcp″*
proof(*cases τmove1 P h (Val v1·compareAndSwap(D·F, e2′, e3))*)
case *True*
with *exec′ τ* **have** [*simp*]: *h = h′* **and** *e*: *sim-move e2′ E′ P t e2 h (stk, loc, pc, xcp) (stk″, loc″, pc″, xcp″)* **by** *auto*
from *e* **have** *sim-move (Val v1·compareAndSwap(D·F, e2′, e3)) (Val v1·compareAndSwap(D·F, E′, e3)) P t (e1·compareAndSwap(D·F, e2, e3)) h (stk @ [v1], loc, length (compE2 e1) + pc, xcp) (stk″ @ [v1], loc″, length (compE2 e1) + pc″, xcp″)*
by(*fastforce dest*: *CAS-τExecrI2 CAS-τExectI2 simp del*: *compE2.simps compEs2.simps*)
with *True* **show** *?thesis* **by** *auto*
next

case *False*
with *exec′ τ* **obtain** *pc′ stk′ loc′ xcp′*
where *e*: *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*)
and *e′*: *exec-move-a P t e2 h* (*stk′*, *loc′*, *pc′*, *xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk″*, *loc″*, *pc″*, *xcp″*)
and *τ′*: ¬ *τmove2* (*compP2 P*) *h stk′ e2 pc′ xcp′*
and *call*: *call1 e2′* = *None* ∨ *no-call2 e2 pc* ∨ *pc′* = *pc* ∧ *stk′* = *stk* ∧ *loc′* = *loc* ∧ *xcp′* = *xcp*
by *auto*
from *e* **have** *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) (*stk′* @ [*v1*], *loc′*, *length* (*compE2 e1*) + *pc′*, *xcp′*) **by**(*rule CAS-τExecrI2*)
moreover from *e′* **have** *exec-move-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk′* @ [*v1*], *loc′*, *length* (*compE2 e1*) + *pc′*, *xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk″* @ [*v1*], *loc″*, *length* (*compE2 e1*) + *pc″*, *xcp″*)
by(*rule exec-move-CASI2*)
moreover from *e′* **have** *pc′* < *length* (*compE2 e2*) **by**(*auto elim*: *exec-meth.cases*)
with *τ′ e′* **have** ¬ *τmove2* (*compP2 P*) *h* (*stk′* @ [*v1*]) (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *pc′*) *xcp′*
by(*auto simp add*: *τinstr-stk-drop-exec-move τmove2-iff*)
moreover from *red* **have** *call1* (*Val v1*·*compareAndSwap*(*D*·*F*, *e2′*, *e3*)) = *call1 e2′* **by** *auto*
moreover have *no-call2 e2 pc* ⟹ *no-call2* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *pc*)
by(*auto simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *False call* **by**(*auto simp del*: *split-paired-Ex call1.simps calls1.simps*)

qed
moreover from *bisim′*
have *P*,*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*),*h′* ⊢ (*Val v1*·*compareAndSwap*(*D*·*F*, *E′*, *e3*), *xs′*) ↔ ((*stk″* @ [*v1*]), *loc″*, *length* (*compE2 e1*) + *pc″*, *xcp″*)
by(*rule bisim1-bisims1.bisim1CAS2*)
ultimately show *?thesis*
apply(*auto simp del*: *split-paired-Ex call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)
apply(*blast intro*: *τExec-mover-trans*)+
done
next
case (*CAS1Red3 E′ v′*)
note [*simp*] = ‹*e2′* = *Val v′*› ‹*e′* = *Val v1*·*compareAndSwap*(*D*·*F*, *Val v′*, *E′*)›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e3*,(*h*, *xs*)⟩ −*ta*→ ⟨*E′*,(*h′*, *xs′*)⟩›
from *red* **have** *τ*: *τmove1 P h* (*Val v1*·*compareAndSwap*(*D*·*F*, *Val v′*, *e3*)) = *τmove1 P h e3*
by(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and *exec1*: *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *xcp*) ([*v′*], *xs*, *length* (*compE2 e2*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) ([*v′*] @ [*v1*], *xs*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
by−(*rule CAS-τExecrI2*)
moreover from *IH3*[*OF red*] *bsok* **obtain** *pc″ stk″ loc″ xcp″*
where *bisim′*: *P*,*e3*,*h′* ⊢ (*E′*, *xs′*) ↔ (*stk″*, *loc″*, *pc″*, *xcp″*)
and *exec′*: *?exec ta e3 e3 E′ h* [] *xs 0 None h′ pc″ stk″ loc″ xcp″* **by** *auto*
have *?exec ta* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*Val v1*·*compareAndSwap*(*D*·*F*, *Val v′*, *e3*)) (*Val v1*⌊*Val v′*⌉ := *E′*) *h* ([] @ [*v′*, *v1*]) *xs* (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *0*) *None h′* (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc″*) (*stk″* @ [*v′*, *v1*]) *loc″ xcp″*
proof(*cases τmove1 P h* (*Val v1*·*compareAndSwap*(*D*·*F*, *Val v′*, *e3*)))
case *True*
with *exec′ τ* **have** [*simp*]: *h* = *h′*

and *e*: *sim-move e3 E′ P t e3 h* ([], *xs*, *0*, *None*) (*stk″*, *loc″*, *pc″*, *xcp″*) **by** *auto*
from *e* **have** *sim-move* (*Val v1·compareAndSwap*(*D·F*, *Val v′*, *e3*)) (*Val v1·compareAndSwap*(*D·F*, *Val v′*, *E′*)) *P t* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) *h* ([] @ [*v′*, *v1*], *xs*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *0*, *None*) (*stk″* @ [*v′*, *v1*], *loc″*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc″*, *xcp″*)
by(*fastforce dest*: *CAS-τExectI3 CAS-τExecrI3 simp del*: *compE2.simps compEs2.simps*)
with *True* **show** *?thesis* **by** *auto*
next
case *False*
with *exec′ τ* **obtain** *pc′ stk′ loc′ xcp′*
where *e*: *τExec-mover-a P t e3 h* ([], *xs*, *0*, *None*) (*stk′*, *loc′*, *pc′*, *xcp′*)
and *e′*: *exec-move-a P t e3 h* (*stk′*, *loc′*, *pc′*, *xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk″*, *loc″*, *pc″*, *xcp″*)
and *τ′*: ¬ *τmove2* (*compP2 P*) *h stk′ e3 pc′ xcp′*
and *call*: *call1 e3* = *None* ∨ *no-call2 e3 0* ∨ *pc′* = *0* ∧ *stk′* = [] ∧ *loc′* = *xs* ∧ *xcp′* = *None* **by** *auto*
from *e* **have** *τExec-mover-a P t* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) *h* ([] @ [*v′*, *v1*], *xs*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *0*, *None*) (*stk′* @ [*v′*, *v1*], *loc′*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc′*, *xcp′*) **by**(*rule CAS-τExecrI3*)
moreover from *e′* **have** *exec-move-a P t* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) *h* (*stk′* @ [*v′*, *v1*], *loc′*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc′*, *xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk″* @ [*v′*, *v1*], *loc″*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc″*, *xcp″*)
by(*rule exec-move-CASI3*)
moreover from *e′ τ′* **have** ¬ *τmove2* (*compP2 P*) *h* (*stk′* @ [*v′*, *v1*]) (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc′*) *xcp′*
by(*auto simp add*: *τinstr-stk-drop-exec-move τmove2-iff*)
moreover from *red* **have** *call1* (*Val v1·compareAndSwap*(*D·F*, *Val v′*, *e3*)) = *call1 e3* **by** *auto*
moreover have *no-call2 e3 0* ⟹ *no-call2* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *length* (*compE2 e2*))
by(*auto simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *False call* **by**(*auto simp del*: *split-paired-Ex call1.simps calls1.simps*) *blast*
qed
moreover from *bisim′*
have *P*,*e1·compareAndSwap*(*D·F*, *e2*, *e3*),*h′* ⊢ (*Val v1·compareAndSwap*(*D·F*, *Val v′*, *E′*), *xs′*) ↔ ((*stk″* @ [*v′*, *v1*]), *loc″*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc″*, *xcp″*)
by(*rule bisim1-bisims1.bisim1CAS3*)
moreover from *bisim2* **have** *pc* ≠ *length* (*compE2 e2*) ⟶ *no-call2* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *pc*)
by(*auto simp add*: *no-call2-def dest*: *bisim-Val-pc-not-Invoke bisim1-pc-length-compE2*)
ultimately show *?thesis* **using** *τ exec1 s*
apply(*auto simp del*: *split-paired-Ex call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)
apply(*blast intro*: *τExec-mover-trans*|*fastforce elim*!: *τExec-mover-trans simp del*: *split-paired-Ex call1.simps calls1.simps*)+
done
next
case (*CAS1Null v v′*)
note [*simp*] = ‹*v1* = *Null*› ‹*e′* = *THROW NullPointer*› ‹*e2′* = *Val v*› ‹*xs′* = *xs*› ‹*ta* = *ε*› ‹*h′* = *h*› ‹*e3* = *Val v′*›
have *τ*: ¬ *τmove1 P h* (*CompareAndSwap null D F* (*Val v*) (*Val v′*)) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 e2*), *None*)
by(*auto dest*: *bisim1Val2D1*)

hence *τExec-mover-a P t* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) *h* (*stk* @ [*Null*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) ([*v*] @ [*Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
by−(*rule CAS-τExecrI2*)
hence *τExec-mover-a P t* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) *h* (*stk* @ [*Null*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) ([] @ [*v*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *0*, *None*) **by** *simp*
also from *bisim3*[*of loc*] **have** *τExec-mover-a P t e3 h* ([], *loc*, *0*, *None*) ([*v'*], *loc*, *length* (*compE2 e3*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) *h* ([] @ [*v*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *0*, *None*) ([*v'*] @ [*v*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*)
by(*rule CAS-τExecrI3*)
also (*rtranclp-trans*)
have *exec-move-a P t* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) *h* ([*v'*, *v*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*) *ε*
h ([*v'*, *v*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *is-Ref-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v'*, *v*, *Null*] (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *e1·compareAndSwap*(*D·F*, *e2*, *e3*), *h'* ⊢ (*THROW NullPointer*, *loc*) ↔ ([*v'*, *v*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule bisim1-bisims1.bisim1CASFail*)
ultimately show *?thesis* **using** *s τ* **by** *auto blast*
next
case (*Red1CASSucceed a v v'*)
hence [*simp*]: *v1* = *Addr a e'* = *true e2'* = *Val v*
ta = {⟦*ReadMem a* (*CField D F*) *v*, *WriteMem a* (*CField D F*) *v'*⟧} *xs'* = *xs e3* = *Val v'*
and *read*: *heap-read h a* (*CField D F*) *v*
and *write*: *heap-write h a* (*CField D F*) *v' h'* **by** *auto*
have *τ*: ¬ *τmove1 P h* (*CompareAndSwap* (*addr a*) *D F* (*Val v*) (*Val v'*)) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 e2*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) *h* (*stk* @ [*Addr a*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) ([*v*] @ [*Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
by−(*rule CAS-τExecrI2*)
hence *τExec-mover-a P t* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) *h* (*stk* @ [*Addr a*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) ([] @ [*v*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *0*, *None*) **by** *simp*
also from *bisim3*[*of loc*] **have** *τExec-mover-a P t e3 h* ([], *loc*, *0*, *None*) ([*v'*], *loc*, *length* (*compE2 e3*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) *h* ([] @ [*v*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *0*, *None*) ([*v'*] @ [*v*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*)
by(*rule CAS-τExecrI3*)
also (*rtranclp-trans*) **from** *read write*
have *exec-move-a P t* (*e1·compareAndSwap*(*D·F*, *e2*, *e3*)) *h* ([*v'*, *v*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*)
{⟦ *ReadMem a* (*CField D F*) *v*, *WriteMem a* (*CField D F*) *v'* ⟧}

h' ([*Bool True*], *loc*, *Suc* (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*)), *None*)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *compP2-def is-Ref-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v'*, *v*, *Addr a*] (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *h'* ⊢ (*true*, *loc*) ↔ ([*Bool True*], *loc*, *length* (*compE2* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*))), *None*)
by(*rule bisim1Val2*) *simp*
ultimately show *?thesis* **using** *s τ* **by**(*auto simp add*: *ta-upd-simps*) *blast*
next
case (*Red1CASFail a v'' v v'*)
hence [*simp*]: *v1* = *Addr a e'* = *false e2'* = *Val v h'* = *h*
ta = {⌈*ReadMem a* (*CField D F*) *v''*⌉} *xs'* = *xs e3* = *Val v'*
and *read*: *heap-read h a* (*CField D F*) *v''* *v* ≠ *v''* **by** *auto*
have *τ*: ¬ *τmove1 P h* (*CompareAndSwap* (*addr a*) *D F* (*Val v*) (*Val v'*)) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 e2*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk* @ [*Addr a*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) ([*v*] @ [*Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
by−(*rule CAS-τExecrI2*)
hence *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk* @ [*Addr a*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) ([] @ [*v*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *0*, *None*) **by** *simp*
also from *bisim3*[*of loc*] **have** *τExec-mover-a P t e3 h* ([], *loc*, *0*, *None*) ([*v'*], *loc*, *length* (*compE2 e3*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([] @ [*v*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *0*, *None*) ([*v'*] @ [*v*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*)
by(*rule CAS-τExecrI3*)
also (*rtranclp-trans*) **from** *read*
have *exec-move-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([*v'*, *v*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*)
{⌈ *ReadMem a* (*CField D F*) *v''* ⌉}
h' ([*Bool False*], *loc*, *Suc* (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*)), *None*)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *compP2-def is-Ref-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v'*, *v*, *Addr a*] (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *h'* ⊢ (*false*, *loc*) ↔ ([*Bool False*], *loc*, *length* (*compE2* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*))), *None*)
by(*rule bisim1Val2*) *simp*
ultimately show *?thesis* **using** *s τ* **by**(*auto simp add*: *ta-upd-simps*) *blast*
next
case (*CAS1Throw2 ad*)
note [*simp*] = ‹*e2'* = *Throw ad*› ‹*ta* = *ε*› ‹*e'* = *Throw ad*› ‹*h'* = *h*› ‹*xs'* = *xs*›
have *τ*: *τmove1 P h* (*Val v1*·*compareAndSwap*(*D*·*F*, *Throw ad*, *e3*)) **by**(*rule τmove1CASThrow2*)
from *bisim2* **have** *xcp* = ⌊*ad*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)

thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*ad*⌋
with *bisim2*
have *P*, *e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *h* ⊢ (*Throw ad*, *xs*) ↔ (*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*)
by(*auto intro*: *bisim1-bisims1.intros*)
thus *?thesis* **using** τ **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim2* **obtain** *pc*′ **where** τ*Exec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *None*) ([*Addr ad*], *loc*, *pc*′, ⌊*ad*⌋)
and *bisim*′: *P*, *e2*, *h* ⊢ (*Throw ad*, *xs*) ↔ ([*Addr ad*], *loc*, *pc*′, ⌊*ad*⌋)
and [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-Throw-*τ*Exec-mover*)
hence τ*Exec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *None*) ([*Addr ad*] @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*′, ⌊*ad*⌋)
by−(*rule CAS-*τ*ExecrI2*)
moreover from *bisim*′
have *P*, *e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *h* ⊢ (*Throw ad*, *xs*) ↔ ([*Addr ad*] @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*′, ⌊*ad*⌋)
by(*rule bisim1-bisims1.bisim1CASThrow2*)
ultimately show *?thesis* **using** τ **by** *auto*
qed
next
case (*CAS1Throw3 v*′ *ad*)
note [*simp*] = ‹*e2*′ = *Val v*′› ‹*e3* = *Throw ad*› ‹*ta* = ε› ‹*e*′ = *Throw ad*› ‹*h*′ = *h*› ‹*xs*′ = *xs*›
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and τ*Exec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *xcp*) ([*v*′], *loc*, *length* (*compE2 e2*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence τ*Exec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *Throw ad*)) *h* (*stk* @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*) ([*v*′] @ [*v1*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
by−(*rule CAS-*τ*ExecrI2*)
also have τ*Exec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *Throw ad*)) *h* ([*v*′] @ [*v1*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*) ([*Addr ad*, *v*′, *v1*], *loc*, *Suc* (*length* (*compE2 e1*) + *length* (*compE2 e2*)), ⌊*ad*⌋)
by(*rule* τ*Execr2step*)(*auto simp add*: *exec-move-def exec-meth-instr* τ*move2-iff* τ*move1.simps* τ*moves1.simps*)
also (*rtranclp-trans*)
have *P*,*e1*·*compareAndSwap*(*D*·*F*, *e2*, *Throw ad*),*h* ⊢ (*Throw ad*, *loc*) ↔ ([*Addr ad*] @ [*v*′, *v1*], *loc*, (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2* (*addr ad*))), ⌊*ad*⌋)
by(*rule bisim1CASThrow3*[*OF bisim1Throw2*])
moreover have τ*move1 P h* (*CompareAndSwap* (*Val v1*) *D F* (*Val v*′) (*Throw ad*)) **by**(*auto intro*: τ*move1CASThrow3*)
ultimately show *?thesis* **using** *s* **by** *auto*
qed *auto*
next
case (*bisim1CAS3 e3 n e3*′ *xs stk loc pc xcp e1 e2 D F v v*′)
note *IH3* = *bisim1CAS3.IH*(*2*)
note *bisim3* = ‹*P*,*e3*,*h* ⊢ (*e3*′, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bsok* = ‹*bsok* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *n*›
from ‹*True*,*P*,*t* ⊢1 ⟨*Val v*·*compareAndSwap*(*D*·*F*, *Val v*′, *e3*′),(*h*, *xs*)⟩ −*ta*→ ⟨*e*′,(*h*′, *xs*′)⟩› **show** *?case*
proof *cases*

case (*CAS1Red3 E'*)
note [*simp*] = ‹*e'* = *Val v*·*compareAndSwap*(*D*·*F*, *Val v'*, *E'*)›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e3'*,(*h*, *xs*)⟩ −*ta*→ ⟨*E'*,(*h'*, *xs'*)⟩›
from *red* **have** τ: τ*move1 P h* (*Val v*·*compareAndSwap*(*D*·*F*, *Val v'*, *e3'*)) = τ*move1 P h e3'*
by(*auto simp add*: τ*move1.simps* τ*moves1.simps*)
from *IH3*[*OF red*] *bsok* **obtain** *pc''* *stk''* *loc''* *xcp''*
where *bisim'*: *P*,*e3*,*h'* ⊢ (*E'*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
and *exec'*: *?exec ta e3 e3' E' h stk loc pc xcp h' pc'' stk'' loc'' xcp''* **by** *auto*
have *no-call2 e3 pc* ⟹ *no-call2* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*)
by(*auto simp add*: *no-call2-def*)
hence *?exec ta* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*Val v*·*compareAndSwap*(*D*·*F*, *Val v'*, *e3'*)) (*Val v*·*compareAndSwap*(*D*·*F*, *Val v'*, *E'*)) *h* (*stk* @ [*v'*, *v*]) *loc* (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*) *xcp h'* (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc''*) (*stk''* @ [*v'*, *v*]) *loc''* *xcp''*
using *exec'* τ
apply(*cases* τ*move1 P h* (*Val v*·*compareAndSwap*(*D*·*F*, *Val v'*, *e3'*)))
apply(*auto*)
apply(*blast intro*: *CAS-*τ*ExecrI3 CAS-*τ*ExectI3 exec-move-CASI3*)
apply(*blast intro*: *CAS-*τ*ExecrI3 CAS-*τ*ExectI3 exec-move-CASI3*)
apply(*rule exI conjI CAS-*τ*ExecrI3 exec-move-CASI3*|*assumption*)+
apply(*fastforce simp add*: τ*instr-stk-drop-exec-move* τ*move2-iff split*: *if-split-asm*)
apply(*rule exI conjI CAS-*τ*ExecrI3 exec-move-CASI3*|*assumption*)+
apply(*fastforce simp add*: τ*instr-stk-drop-exec-move* τ*move2-iff split*: *if-split-asm*)
apply(*rule exI conjI CAS-*τ*ExecrI3 exec-move-CASI3 rtranclp.rtrancl-refl*|*assumption*)+
apply(*fastforce simp add*: τ*instr-stk-drop-exec-move* τ*move2-iff split*: *if-split-asm*)+
done
moreover from *bisim'*
have *P*,*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*),*h'* ⊢ (*Val v*·*compareAndSwap*(*D*·*F*, *Val v'*, *E'*), *xs'*) ↔ (*stk''*@[*v'*,*v*], *loc''*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc''*, *xcp''*)
by(*rule bisim1-bisims1.bisim1CAS3*)
ultimately show *?thesis* **using** τ **by** *auto blast*+
next
case (*CAS1Null v''*)
note [*simp*] = ‹*v* = *Null*› ‹*e'* = *THROW NullPointer*› ‹*xs'* = *xs*› ‹*ta* = ε› ‹*h'* = *h*› ‹*e3'* = *Val v''*›
have τ: ¬ τ*move1 P h* (*CompareAndSwap null D F* (*Val v'*) (*Val v''*)) **by**(*auto simp add*: τ*move1.simps* τ*moves1.simps*)
from *bisim3* **have** *s*: *xcp* = *None xs* = *loc*
and τ*Exec-mover-a P t e3 h* (*stk*, *loc*, *pc*, *xcp*) ([*v''*], *loc*, *length* (*compE2 e3*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence τ*Exec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk* @ [*v'*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*, *xcp*) ([*v''*] @ [*v'*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*)
by−(*rule CAS-*τ*ExecrI3*)
moreover
have *exec-move-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([*v''*, *v'*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*) ε
h ([*v''*, *v'*, *Null*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *is-Ref-def*)
moreover have τ*move2* (*compP2 P*) *h* [*v''*, *v'*, *Null*] (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*)) *None* ⟹ *False*
by(*simp add*: τ*move2-iff*)
moreover
have *P*, *e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *h'* ⊢ (*THROW NullPointer*, *loc*) ↔ ([*v''*, *v'*, *Null*], *loc*,

length (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule bisim1-bisims1.bisim1CASFail*)
ultimately show *?thesis* **using** *s τ* **by** *auto blast*
next
case (*Red1CASSucceed a v''*)
hence [*simp*]: *v* = *Addr a e'* = *true e3'* = *Val v''*
ta = {⌈*ReadMem a* (*CField D F*) *v'*, *WriteMem a* (*CField D F*) *v''*⌉} *xs'* = *xs*
and *read*: *heap-read h a* (*CField D F*) *v'*
and *write*: *heap-write h a* (*CField D F*) *v''* *h'* **by** *auto*
have *τ*: ¬ *τmove1 P h* (*CompareAndSwap* (*addr a*) *D F* (*Val v'*) (*Val v''*)) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim3* **have** *s*: *xcp* = *None xs* = *loc*
and *exec1*: *τExec-mover-a P t e3 h* (*stk*, *loc*, *pc*, *xcp*) ([*v'*], *loc*, *length* (*compE2 e3*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk* @ [*v'*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*, *xcp*) ([*v'*] @ [*v'*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*)
by−(*rule CAS-τExecrI3*)
moreover from *read write*
have *exec-move-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([*v''*, *v'*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*)
{⌈*ReadMem a* (*CField D F*) *v'*, *WriteMem a* (*CField D F*) *v''*⌉}
h' ([*Bool True*], *loc*, *Suc* (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*)), *None*)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *compP2-def is-Ref-def*)
moreover have *τmove2* (*compP2 P*) *h* [*v''*, *v'*, *Addr a*] (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*)) *None* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover
have *P*, *e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *h'* ⊢ (*true*, *loc*) ↔ ([*Bool True*], *loc*, *length* (*compE2* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*))), *None*)
by(*rule bisim1Val2*) *simp*
ultimately show *?thesis* **using** *s τ* **by**(*auto simp add*: *ta-upd-simps ac-simps*) *blast*
next
case (*Red1CASFail a v'' v'''*)
hence [*simp*]: *v* = *Addr a e'* = *false e3'* = *Val v'''* *h'* = *h*
ta = {⌈*ReadMem a* (*CField D F*) *v''*⌉} *xs'* = *xs*
and *read*: *heap-read h a* (*CField D F*) *v''* *v'* ≠ *v''* **by** *auto*
have *τ*: ¬ *τmove1 P h* (*CompareAndSwap* (*addr a*) *D F* (*Val v'*) (*Val v'''*)) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim3* **have** *s*: *xcp* = *None xs* = *loc*
and *exec1*: *τExec-mover-a P t e3 h* (*stk*, *loc*, *pc*, *xcp*) ([*v'''*], *loc*, *length* (*compE2 e3*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* (*stk* @ [*v'*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*, *xcp*) ([*v'''*] @ [*v'*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*)
by−(*rule CAS-τExecrI3*)
moreover from *read*
have *exec-move-a P t* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *h* ([*v'''*, *v'*, *Addr a*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), *None*)
{⌈*ReadMem a* (*CField D F*) *v''*⌉}
h' ([*Bool False*], *loc*, *Suc* (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*)), *None*)
unfolding *exec-move-def* **by**−(*rule exec-instr*, *auto simp add*: *compP2-def is-Ref-def*)

moreover have *τmove2 (compP2 P) h [v‴, v′, Addr a] (e1·compareAndSwap(D·F, e2, e3)) (length (compE2 e1) + length (compE2 e2) + length (compE2 e3)) None ⟹ False*
by(*simp add*: *τmove2-iff*)
moreover
have *P, e1·compareAndSwap(D·F, e2, e3), h′ ⊢ (false, loc) ↔ ([Bool False], loc, length (compE2 (e1·compareAndSwap(D·F, e2, e3))), None)*
by(*rule bisim1Val2*) *simp*
ultimately show *?thesis* **using** *s τ* **by**(*auto simp add*: *ta-upd-simps ac-simps*) *blast*
next
case (*CAS1Throw3 A*)
note [*simp*] = ‹*e3′ = Throw A*› ‹*ta = ε*› ‹*e′ = Throw A*› ‹*h′ = h*› ‹*xs′ = xs*›
have *τ*: *τmove1 P h (CompareAndSwap (Val v) D F (Val v′) (Throw A))* **by**(*rule τmove1CASThrow3*)
from *bisim3* **have** *xcp = ⌊A⌋ ∨ xcp = None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp = ⌊A⌋*
with *bisim3*
have *P, e1·compareAndSwap(D·F, e2, e3), h ⊢ (Throw A, xs) ↔ (stk @ [v′, v], loc, length (compE2 e1) + length (compE2 e2) + pc, xcp)*
by(*auto intro*: *bisim1-bisims1.intros*)
thus *?thesis* **using** *τ* **by**(*fastforce*)
next
assume [*simp*]: *xcp = None*
with *bisim3* **obtain** *pc′* **where** *τExec-mover-a P t e3 h (stk, loc, pc, None) ([Addr A], loc, pc′, ⌊A⌋)*
and *bisim′*: *P, e3, h ⊢ (Throw A, xs) ↔ ([Addr A], loc, pc′, ⌊A⌋)*
and [*simp*]: *xs = loc*
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t (e1·compareAndSwap(D·F, e2, e3)) h (stk @ [v′, v], loc, length (compE2 e1) + length (compE2 e2) + pc, None) ([Addr A] @ [v′, v], loc, length (compE2 e1) + length (compE2 e2) + pc′, ⌊A⌋)*
by−(*rule CAS-τExecrI3*)
moreover from *bisim′*
have *P, e1·compareAndSwap(D·F, e2, e3), h ⊢ (Throw A, xs) ↔ ([Addr A] @ [v′, v], loc, length (compE2 e1) + length (compE2 e2) + pc′, ⌊A⌋)*
by(*rule bisim1-bisims1.bisim1CASThrow3*)
ultimately show *?thesis* **using** *τ* **by** *auto*
qed
qed *auto*
next
case *bisim1CASThrow1* **thus** *?case* **by** *auto*
next
case *bisim1CASThrow2* **thus** *?case* **by** *auto*
next
case *bisim1CASThrow3* **thus** *?case* **by** *auto*
next
case *bisim1CASFail* **thus** *?case* **by** *auto*
next
case (*bisim1CallParams ps n ps′ xs stk loc pc xcp obj M′ v*)
note *IHparam = bisim1CallParams.IH*(*2*)
note *bisim1 = ‹⋀xs. P,obj,h ⊢ (obj, xs) ↔ ([], xs, 0, None)›*
note *bisim2 = ‹P,ps,h ⊢ (ps′, xs) [↔] (stk, loc, pc, xcp)›*
note *red = ‹True,P,t ⊢1 ⟨Val v·M′(ps′),(h, xs)⟩ −ta→ ⟨e′,(h′, xs′)⟩›*
note *bsok = ‹bsok (obj·M′(ps)) n›*

from *bisim2* ‹*ps* ≠ []› **have** *ps'*: *ps'* ≠ [] **by**(*auto dest*: *bisims1-lengthD*)
from *red* **show** *?case*
proof *cases*
case (*Call1Params es'*)
note [*simp*] = ‹*e'* = *Val v*·*M'*(*es'*)›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*ps'*, (*h*, *xs*)⟩ [−*ta*→] ⟨*es'*, (*h'*, *xs'*)⟩›
from *red* **have** *τ*: *τmove1 P h* (*Val v*·*M'*(*ps'*)) = *τmoves1 P h ps'* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *IHparam*[*OF red*] *bsok* **obtain** *pc''* *stk''* *loc''* *xcp''*
where *bisim'*: *P*,*ps*,*h'* ⊢ (*es'*, *xs'*) [↔] (*stk''*, *loc''*, *pc''*, *xcp''*)
and *exec'*: *?execs ta ps ps' es' h stk loc pc xcp h' pc'' stk'' loc'' xcp''* **by** *auto*
have *?exec ta* (*obj*·*M'*(*ps*)) (*Val v*·*M'*(*ps'*)) (*Val v*·*M'*(*es'*)) *h* (*stk* @ [*v*]) *loc* (*length* (*compE2 obj*) + *pc*) *xcp h'* (*length* (*compE2 obj*) + *pc''*) (*stk''* @ [*v*]) *loc''* *xcp''*
proof(*cases τmove1 P h* (*Val v*·*M'*(*ps'*)))
case *True*
with *exec' τ* **show** *?thesis* **by** (*auto intro*: *Call-τExecrI2 Call-τExectI2*)
next
case *False*
with *exec' τ* **obtain** *pc'* *stk'* *loc'* *xcp'*
where *e*: *τExec-movesr-a P t ps h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
and *e'*: *exec-moves-a P t ps h* (*stk'*, *loc'*, *pc'*, *xcp'*) (*extTA2JVM* (*compP2 P*) *ta*) *h'* (*stk''*, *loc''*, *pc''*, *xcp''*)
and *τ'*: ¬ *τmoves2* (*compP2 P*) *h stk' ps pc' xcp'*
and *call*: (*calls1 ps'* = *None* ∨ *no-calls2 ps pc* ∨ *pc'* = *pc* ∧ *stk'* = *stk* ∧ *loc'* = *loc* ∧ *xcp'* = *xcp*) **by** *auto*
from *e* **have** *τExec-mover-a P t* (*obj*·*M'*(*ps*)) *h* (*stk* @ [*v*], *loc*, *length* (*compE2 obj*) + *pc*, *xcp*) (*stk'* @ [*v*], *loc'*, *length* (*compE2 obj*) + *pc'*, *xcp'*) **by**(*rule Call-τExecrI2*)
moreover from *e'* **have** *exec-move-a P t* (*obj*·*M'*(*ps*)) *h* (*stk'* @ [*v*], *loc'*, *length* (*compE2 obj*) + *pc'*, *xcp'*) (*extTA2JVM* (*compP2 P*) *ta*) *h'* (*stk''* @ [*v*], *loc''*, *length* (*compE2 obj*) + *pc''*, *xcp''*)
by(*rule exec-move-CallI2*)
moreover from *τ' e'* **have** *τmove2* (*compP2 P*) *h* (*stk'* @ [*v*]) (*obj*·*M'*(*ps*)) (*length* (*compE2 obj*) + *pc'*) *xcp'* ⟹ *False*
by(*auto simp add*: *τmove2-iff τmoves2-iff τinstr-stk-drop-exec-moves split*: *if-split-asm*)
moreover from *red* **have** *call1* (*Val v*·*M'*(*ps'*)) = *calls1 ps'* **by**(*auto simp add*: *is-vals-conv*)
moreover from *red* **have** *no-calls2 ps pc* ⟹ *no-call2* (*obj*·*M'*(*ps*)) (*length* (*compE2 obj*) + *pc*) ∨ *pc* = *length* (*compEs2 ps*)
by(*auto simp add*: *no-call2-def no-calls2-def*)
ultimately show *?thesis* **using** *False call e*
by(*auto simp del*: *split-paired-Ex call1.simps calls1.simps*) *blast*+
qed
moreover from *bisim'*
have *P*,*obj*·*M'*(*ps*),*h'* ⊢ (*Val v*·*M'*(*es'*), *xs'*) ↔ ((*stk''* @ [*v*]), *loc''*, *length* (*compE2 obj*) + *pc''*, *xcp''*)
by(*rule bisim1-bisims1.bisim1CallParams*)
ultimately show *?thesis* **using** *τ*
by(*auto simp del*: *call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*) *blast*+
next
case (*Red1CallNull vs*)
note [*simp*] = ‹*h'* = *h*› ‹*xs'* = *xs*› ‹*ta* = *ε*› ‹*v* = *Null*› ‹*ps'* = *map Val vs*› ‹*e'* = *THROW NullPointer*›
from *bisim2* **have** *length*: *length ps* = *length vs* **by**(*auto dest*: *bisims1-lengthD*)
have *xs* = *loc* ∧ *xcp* = *None* ∧ *τExec-movesr-a P t ps h* (*stk*, *loc*, *pc*, *xcp*) (*rev vs*, *loc*, *length* (*compEs2 ps*), *None*)
proof(*cases pc* < *length* (*compEs2 ps*))

case *True*
with *bisim2* **show** *?thesis* **by**(*auto dest*: *bisims1-Val-τExec-moves*)
next
case *False*
with *bisim2* **have** *pc* = *length* (*compEs2 ps*)
by(*auto dest*: *bisims1-pc-length-compEs2*)
with *bisim2* **show** *?thesis* **by**(*auto dest*: *bisims1-Val-length-compEs2D*)
qed
hence *s*: *xs* = *loc xcp* = *None*
and *τExec-movesr-a P t ps h* (*stk*, *loc*, *pc*, *xcp*) (*rev vs*, *loc*, *length* (*compEs2 ps*), *None*) **by** *auto*
hence *τExec-mover-a P t* (*obj*·*M′*(*ps*)) *h* (*stk* @ [*Null*], *loc*, *length* (*compE2 obj*) + *pc*, *xcp*) (*rev vs* @ [*Null*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), *None*)
by −(*rule Call-τExecrI2*)
also **{**
from *length* **have** *exec-move-a P t* (*obj*·*M′*(*ps*)) *h* (*rev vs* @ [*Null*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), *None*) *ε h* (*rev vs* @ [*Null*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *exec-move-def* **by**(*cases ps*)(*auto intro*: *exec-instr*)
moreover have *τmove2 P h* (*rev vs* @ [*Null*]) (*obj*·*M′*(*ps*)) (*length* (*compE2 obj*) + *length* (*compEs2 ps*)) *None*
using *length* **by**(*simp add*: *τmove2-iff*)
ultimately have *τExec-movet-a P t* (*obj*·*M′*(*ps*)) *h* (*rev vs* @ [*Null*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), *None*) (*rev vs* @ [*Null*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*auto intro*: *τexec-moveI*) **}**
also have *τmove1 P h* (*null*·*M′*(*map Val vs*))
by(*auto simp add*: *τmove1.simps τmoves1.simps map-eq-append-conv*)
moreover
from *length* **have** *P*,*obj*·*M′*(*ps*),*h* ⊢ (*THROW NullPointer*, *loc*) ↔ ((*rev vs* @ [*Null*]), *loc*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by−(*rule bisim1CallThrow*, *simp*)
ultimately show *?thesis* **using** *s* **by** *auto*
next
case (*Call1ThrowParams vs a es′*)
note [*simp*] = ‹*ta* = *ε*› ‹*e′* = *Throw a*› ‹*ps′* = *map Val vs* @ *Throw a* # *es′*› ‹*h′* = *h*› ‹*xs′* = *xs*›
have *τ*: *τmove1 P h* (*Val v*·*M′*(*map Val vs* @ *Throw a* # *es′*)) **by**(*rule τmove1CallThrowParams*)
from *bisim2* **have** [*simp*]: *xs* = *loc* **and** *xcp* = ⌊*a*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisims1-ThrowD*)
from ‹*xcp* = ⌊*a*⌋ ∨ *xcp* = *None*› **show** *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*a*⌋
with *bisim2*
have *P*,*obj*·*M′*(*ps*),*h* ⊢ (*Throw a*, *loc*) ↔ (*stk* @ [*v*], *loc*, *length* (*compE2 obj*) + *pc*, ⌊*a*⌋)
by −(*rule bisim1CallThrowParams*, *auto*)
thus *?thesis* **using** *τ* **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim2* **obtain** *pc′*
where *exec*: *τExec-movesr-a P t ps h* (*stk*, *loc*, *pc*, *None*) (*Addr a* # *rev vs*, *loc*, *pc′*, ⌊*a*⌋)
and *bisim′*: *P*, *ps*, *h* ⊢ (*map Val vs* @ *Throw a* # *es′*, *loc*) [↔] (*Addr a* # *rev vs*, *loc*, *pc′*, ⌊*a*⌋)
by(*auto dest*: *bisims1-Throw-τExec-movesr*)
from *bisim′*
have *P*,*obj*·*M′*(*ps*),*h* ⊢ (*Throw a*, *loc*) ↔ ((*Addr a* # *rev vs*) @ [*v*], *loc*, *length* (*compE2 obj*) + *pc′*, ⌊*a*⌋)
by(*rule bisim1CallThrowParams*)

with *Call-τExecrI2*[*OF exec*, *of obj M′ v*] *τ*
show *?thesis* **by** *auto*
qed
next
case (*Red1CallExternal a Ta Ts Tr D vs va H′*)
hence [*simp*]: *v* = *Addr a ps′* = *map Val vs e′* = *extRet2J* (*addr a*·*M′*(*map Val vs*)) *va H′* = *h′ xs′* = *xs*
and *Ta*: *typeof-addr h a* = ⌊*Ta*⌋
and *iec*: *P* ⊢ *class-type-of Ta sees M′*: *Ts*→*Tr* = *Native in D*
and *redex*: *P*,*t* ⊢ ⟨*a*·*M′*(*vs*),*h*⟩ −*ta*→*ext* ⟨*va*,*h′*⟩ **by** *auto*
from *bisim2* **have** [*simp*]: *xs* = *loc* **by**(*auto dest*: *bisims-Val-loc-eq-xcp-None*)
moreover from *bisim2* **have** *length ps* = *length ps′*
by(*rule bisims1-lengthD*)
hence *τ*: *τmove1 P h* (*addr a*·*M′*(*map Val vs*) :: *′addr expr1*) = *τmove2* (*compP2 P*) *h* (*rev vs* @ [*Addr a*]) (*obj*·*M′*(*ps*)) (*length* (*compE2 obj*) + *length* (*compEs2 ps*)) *None*
using *Ta iec* **by**(*auto simp add*: *τmove1.simps τmoves1.simps map-eq-append-conv τmove2-iff compP2-def*)
obtain *s*: *xcp* = *None xs* = *loc*
and *τExec-movesr-a P t ps h* (*stk*, *loc*, *pc*, *xcp*) (*rev vs*, *loc*, *length* (*compEs2 ps*), *None*)
proof(*cases pc* < *length* (*compEs2 ps*))
case *True*
with *bisim2* **show** *?thesis* **by**(*auto dest*: *bisims1-Val-τExec-moves intro*: *that*)
next
case *False*
with *bisim2* **have** *pc* = *length* (*compEs2 ps*) **by**(*auto dest*: *bisims1-pc-length-compEs2*)
with *bisim2* **show** *?thesis* **by** −(*rule that*, *auto dest*!: *bisims1-pc-length-compEs2D*)
qed
from *Call-τExecrI2*[*OF this*(*3*), *of obj M′ v*]
have *τExec-mover-a P t* (*obj*·*M′*(*ps*)) *h* (*stk* @ [*Addr a*], *loc*, *length* (*compE2 obj*) + *pc*, *xcp*) (*rev vs* @ [*Addr a*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), *None*) **by** *simp*
moreover from *bisim2* **have** *pc* ≤ *length* (*compEs2 ps*) **by**(*rule bisims1-pc-length-compEs2*)
hence *no-call2* (*obj*·*M′*(*ps*)) (*length* (*compE2 obj*) + *pc*) ∨ *pc* = *length* (*compEs2 ps*)
using *bisim2* **by**(*auto simp add*: *no-call2-def neq-Nil-conv dest*: *bisims-Val-pc-not-Invoke*)
moreover {
assume *pc* = *length* (*compEs2 ps*)
with ‹*τExec-movesr-a P t ps h* (*stk*, *loc*, *pc*, *xcp*) (*rev vs*, *loc*, *length* (*compEs2 ps*), *None*)›
have *stk* = *rev vs xcp* = *None* **by** *auto* }
moreover
let *?ret* = *extRet2JVM* (*length ps*) *h′* (*rev vs* @ [*Addr a*]) *loc undefined undefined* (*length* (*compE2 obj*) + *length* (*compEs2 ps*)) [] *va*
let *?stk′* = *fst* (*hd* (*snd* (*snd ?ret*)))
let *?xcp′* = *fst ?ret*
let *?pc′* = *snd* (*snd* (*snd* (*snd* (*hd* (*snd* (*snd ?ret*))))))
from *bisim2* **have** [*simp*]: *length ps* = *length vs* **by**(*auto dest*: *bisims1-lengthD*)
from *redex* **have** *redex′*: (*ta*, *va*, *h′*) ∈ *red-external-aggr* (*compP2 P*) *t a M′ vs h*
by −(*rule red-external-imp-red-external-aggr*, *simp add*: *compP2-def*)
with *Ta iec*
have *exec-move-a P t* (*obj*·*M′*(*ps*)) *h* (*rev vs* @ [*Addr a*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), *None*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*?stk′*, *loc*, *?pc′*, *?xcp′*)
unfolding *exec-move-def*
by −(*rule exec-instr*,*cases va*,(*force simp add*: *compP2-def is-Ref-def simp del*: *split-paired-Ex intro*: *external-WT′.intros*)+)
moreover have *P*,*obj*·*M′*(*ps*),*h′* ⊢ (*extRet2J1* (*addr a*·*M′*(*map Val vs*)) *va*, *loc*) ↔ (*?stk′*, *loc*, *?pc′*, *?xcp′*)

```
    proof(cases va)
      case (RetVal v)
      have P,obj·M'(ps),h' ⊢ (Val v, loc) ↔ ([v], loc, length (compE2 (obj·M'(ps))), None)
        by(rule bisim1Val2) simp
      thus ?thesis unfolding RetVal by simp
    next
      case (RetExc ad) thus ?thesis by(auto intro: bisim1CallThrow)
    next
      case RetStaySame
      from bisims1-map-Val-append[OF bisims1Nil, of ps vs P h' loc]
      have P,ps,h' ⊢ (map Val vs, loc) [↔] (rev vs, loc, length (compEs2 ps), None) by simp
     hence P,obj·M'(ps),h' ⊢ (addr a·M'(map Val vs), loc) ↔ (rev vs @ [Addr a], loc, length (compE2
obj) + length (compEs2 ps), None)
        by(rule bisim1-bisims1.bisim1CallParams)
      thus ?thesis using RetStaySame by simp
    qed
    moreover from redex Ta iec
    have τmove1 P h (addr a·M'(map Val vs) :: 'addr expr1) ⟹ ta = ε ∧ h' = h
    by(fastforce simp add: τmove1.simps τmoves1.simps map-eq-append-conv τexternal'-def τexternal-def
dest: τexternal'-red-external-TA-empty τexternal'-red-external-heap-unchanged sees-method-fun)
    ultimately show ?thesis using τ
      apply(cases τmove1 P h (addr a·M'(map Val vs) :: 'addr expr1))
      apply(auto simp del: split-paired-Ex simp add: compP2-def)
      apply(blast intro: rtranclp.rtrancl-into-rtrancl rtranclp-into-tranclp1 τexec-moveI)+
      done
  qed(insert ps', auto)
next
  case bisim1CallThrowObj thus ?case by fastforce
next
  case bisim1CallThrowParams thus ?case by auto
next
  case bisim1CallThrow thus ?case by fastforce
next
  case (bisim1BlockSome1 e n V Ty v xs e')
  from ‹True,P,t ⊢1 ⟨{V:Ty=⌊v⌋; e},(h, xs)⟩ −ta→ ⟨e',(h', xs')⟩› show ?case
  proof(cases)
    case Block1Some
    note [simp] = ‹ta = ε› ‹e' = {V:Ty=None; e}› ‹h' = h› ‹xs' = xs[V := v]›
      and len = ‹V < length xs›
    from len have exec: τExec-movet-a P t {V:Ty=⌊v⌋; e} h ([], xs, 0, None) ([], xs[V := v], Suc
(Suc 0), None)
      by−(rule τExect2step, auto intro: exec-instr simp add: exec-move-def τmove2-iff)
   moreover have P,{V:Ty=⌊v⌋; e},h ⊢ ({V:Ty=None; e}, xs[V := v]) ↔ ([], xs[V := v], Suc (Suc
0), None)
      by(rule bisim1BlockSome4)(rule bisim1-refl)
    moreover have τmove1 P h {V:Ty=⌊v⌋; e} by(auto intro: τmove1BlockSome)
    ultimately show ?thesis by auto
  qed
next
  case (bisim1BlockSome2 e n V Ty v xs)
  from ‹True,P,t ⊢1 ⟨{V:Ty=⌊v⌋; e},(h, xs)⟩ −ta→ ⟨e',(h', xs')⟩› show ?case
  proof(cases)
    case Block1Some
    note [simp] = ‹ta = ε› ‹e' = {V:Ty=None; e}› ‹h' = h› ‹xs' = xs[V := v]›
```

and *len* = ‹*V* < *length xs*›
from *len* **have** *exec*: *τExec-movet-a P t* {*V*:*Ty*=⌊*v*⌋; *e*} *h* ([*v*], *xs*, *Suc 0*, *None*) ([], *xs*[*V* := *v*], *Suc* (*Suc 0*), *None*)
by−(*rule τExect1step*, *auto intro*: *exec-instr τmove2BlockSome2 simp*: *exec-move-def*)
moreover have *P*,{*V*:*Ty*=⌊*v*⌋; *e*},*h* ⊢ ({*V*:*Ty*=*None*; *e*}, *xs*[*V* := *v*]) ↔ ([], *xs*[*V* := *v*], *Suc* (*Suc 0*), *None*)
by(*rule bisim1BlockSome4*)(*rule bisim1-refl*)
moreover have *τmove1 P h* {*V*:*Ty*=⌊*v*⌋; *e*} **by**(*auto intro*: *τmove1BlockSome*)
ultimately show *?thesis* **by** *auto*
qed
next
case (*bisim1BlockSome4 E n e xs stk loc pc xcp V Ty v*)
note *IH* = *bisim1BlockSome4.IH*(*2*)
note *bisim* = ‹*P*,*E*,*h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bsok* = ‹*bsok* {*V*:*Ty*=⌊*v*⌋; *E*} *n*›
hence [*simp*]: *n* = *V* **by** *simp*
from ‹*True*,*P*,*t* ⊢*1* ⟨{*V*:*Ty*=*None*; *e*},(*h*, *xs*)⟩ −*ta*→ ⟨*e*′,(*h*′, *xs*′)⟩› **show** *?case*
proof *cases*
case (*Block1Red E*′)
note [*simp*] = ‹*e*′ = {*V*:*Ty*=*None*; *E*′}›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e*,(*h*, *xs*)⟩ −*ta*→ ⟨*E*′,(*h*′, *xs*′)⟩›
from *red* **have** *τ*: *τmove1 P h* {*V*:*Ty*=*None*; *e*} = *τmove1 P h e* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *IH*[*OF red*] *bsok* **obtain** *pc*′′ *stk*′′ *loc*′′ *xcp*′′
where *bisim*′: *P*,*E*,*h*′ ⊢ (*E*′, *xs*′) ↔ (*stk*′′, *loc*′′, *pc*′′, *xcp*′′)
and *exec*′: *?exec ta E e E*′ *h stk loc pc xcp h*′ *pc*′′ *stk*′′ *loc*′′ *xcp*′′ **by** *auto*
have *no-call2 E pc* ⟹ *no-call2* ({*V*:*Ty*=⌊*v*⌋; *E*}) (*Suc* (*Suc pc*)) **by**(*auto simp add*: *no-call2-def*)
hence *?exec ta* {*V*:*Ty*=⌊*v*⌋; *E*} {*V*:*Ty*=*None*; *e*} {*V*:*Ty*=*None*; *E*′} *h stk loc* (*Suc* (*Suc pc*)) *xcp h*′ (*Suc* (*Suc pc*′′)) *stk*′′ *loc*′′ *xcp*′′
using *exec*′ *τ*
by(*cases τmove1 P h* {*V*:*Ty*=*None*; *e*})(*auto*, (*blast intro*: *exec-move-BlockSomeI Block-τExecrI-Some Block-τExectI-Some*)+)
with *bisim*′ *τ* **show** *?thesis* **by** *auto*(*blast intro*: *bisim1-bisims1.bisim1BlockSome4*)+
next
case (*Red1Block u*)
note [*simp*] = ‹*e* = *Val u*› ‹*ta* = *ε*› ‹*e*′ = *Val u*› ‹*h*′ = *h*› ‹*xs*′ = *xs*›
have *τmove1 P h* {*V*:*Ty*=*None*; *Val u*} **by**(*rule τmove1BlockRed*)
moreover from *bisim* **have** [*simp*]: *xcp* = *None loc* = *xs*
and *exec*: *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *xcp*) ([*u*], *loc*, *length* (*compE2 E*), *None*) **by**(*auto dest*: *bisim1Val2D1*)
moreover
have *P*,{*V*:*Ty*=⌊*v*⌋; *E*},*h* ⊢ (*Val u*, *xs*) ↔ ([*u*], *xs*, *length* (*compE2* {*V*:*Ty*=⌊*v*⌋; *E*}), *None*)
by(*rule bisim1Val2*) *simp*
ultimately show *?thesis* **by**(*fastforce elim*!: *Block-τExecrI-Some*)
next
case (*Block1Throw a*)
note [*simp*] = ‹*e* = *Throw a*› ‹*ta* = *ε*› ‹*e*′ = *Throw a*› ‹*h*′ = *h*› ‹*xs*′ = *xs*›
have *τ*: *τmove1 P h* {*V*:*Ty*=*None*; *e*} **by**(*auto intro*: *τmove1BlockThrow*)
from *bisim* **have** *xcp* = ⌊*a*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*a*⌋
with *bisim* **have** *P*, {*V*:*Ty*=⌊*v*⌋; *E*}, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *Suc* (*Suc pc*), *xcp*)
by(*auto intro*: *bisim1BlockThrowSome*)

thus *?thesis* **using** τ **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim* **obtain** *pc′*
where *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
and *bisim′*: *P*, *E*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t* {*V*:*Ty*=⌊*v*⌋; *E*} *h* (*stk*, *loc*, *Suc* (*Suc pc*), *None*) ([*Addr a*], *loc*, *Suc* (*Suc pc′*), ⌊*a*⌋)
by(*auto intro*: *Block-τExecrI-Some*)
moreover from *bisim′* **have** *P*, {*V*:*Ty*=⌊*v*⌋; *E*}, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *Suc* (*Suc pc′*), ⌊*a*⌋)
by(*auto intro*: *bisim1-bisims1.bisim1BlockThrowSome*)
ultimately show *?thesis* **using** τ **by** *auto*
qed
qed
next
case *bisim1BlockThrowSome* **thus** *?case* **by** *auto*
next
case (*bisim1BlockNone E n e xs stk loc pc xcp V Ty*)
note *IH* = *bisim1BlockNone.IH*(*2*)
note *bisim* = ‹*P*,*E*,*h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bsok* = ‹*bsok* {*V*:*Ty*=*None*; *E*} *n*›
hence [*simp*]: *n* = *V* **by** *simp*
from ‹*True*,*P*,*t* ⊢*1* ⟨{*V*:*Ty*=*None*; *e*},(*h*, *xs*)⟩ −*ta*→ ⟨*e′*,(*h′*, *xs′*)⟩› **show** *?case*
proof *cases*
case (*Block1Red E′*)
note [*simp*] = ‹*e′* = {*V*:*Ty*=*None*; *E′*}›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e*,(*h*, *xs*)⟩ −*ta*→ ⟨*E′*,(*h′*, *xs′*)⟩›
from *red* **have** τ: *τmove1 P h* {*V*:*Ty*=*None*; *e*} = *τmove1 P h e* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
moreover have *call1* ({*V*:*Ty*=*None*; *e*}) = *call1 e* **by** *auto*
moreover from *IH*[*OF red*] *bsok*
obtain *pc″ stk″ loc″ xcp″* **where** *bisim*: *P*,*E*,*h′* ⊢ (*E′*, *xs′*) ↔ (*stk″*, *loc″*, *pc″*, *xcp″*)
and *redo*: *?exec ta E e E′ h stk loc pc xcp h′ pc″ stk″ loc″ xcp″* **by** *auto*
from *bisim*
have *P*,{*V*:*Ty*=*None*; *E*},*h′* ⊢ ({*V*:*Ty*=*None*; *E′*}, *xs′*) ↔ (*stk″*, *loc″*, *pc″*, *xcp″*)
by(*rule bisim1-bisims1.bisim1BlockNone*)
moreover {
assume *no-call2 E pc*
hence *no-call2* {*V*:*Ty*=*None*; *E*} *pc* **by**(*auto simp add*: *no-call2-def*) }
ultimately show *?thesis* **using** *redo*
by(*auto simp add*: *exec-move-BlockNone simp del*: *call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)(*blast intro*: *Block-τExecrI-None Block-τExectI-None*)+
next
case (*Red1Block u*)
note [*simp*] = ‹*e* = *Val u*› ‹*ta* = *ε*› ‹*e′* = *Val u*› ‹*h′* = *h*› ‹*xs′* = *xs*›
have *τmove1 P h* {*V*:*Ty*=*None*; *Val u*} **by**(*rule τmove1BlockRed*)
moreover from *bisim* **have** [*simp*]: *xcp* = *None loc* = *xs*
and *exec*: *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *xcp*) ([*u*], *loc*, *length* (*compE2 E*), *None*) **by**(*auto dest*: *bisim1Val2D1*)
moreover
have *P*,{*V*:*Ty*=*None*; *E*},*h* ⊢ (*Val u*, *xs*) ↔ ([*u*], *xs*, *length* (*compE2* {*V*:*Ty*=*None*; *E*}), *None*)
by(*rule bisim1Val2*) *simp*

```
    ultimately show ?thesis by(fastforce intro: Block-τExecrI-None)
  next
    case (Block1Throw a)
    note [simp] = ‹e = Throw a› ‹ta = ε› ‹e' = Throw a› ‹h' = h› ‹xs' = xs›
    have τ: τmove1 P h {V:Ty=None; e} by(auto intro: τmove1BlockThrow)
    from bisim have xcp = ⌊a⌋ ∨ xcp = None by(auto dest: bisim1-ThrowD)
    thus ?thesis
    proof
      assume [simp]: xcp = ⌊a⌋
      with bisim have P, {V:Ty=None; E}, h ⊢ (Throw a, xs) ↔ (stk, loc, pc, xcp)
        by(auto intro: bisim1BlockThrowNone)
      thus ?thesis using τ by(fastforce)
    next
      assume [simp]: xcp = None
      with bisim obtain pc'
        where τExec-mover-a P t E h (stk, loc, pc, None) ([Addr a], loc, pc', ⌊a⌋)
        and bisim': P, E, h ⊢ (Throw a, xs) ↔ ([Addr a], loc, pc', ⌊a⌋) and [simp]: xs = loc
        by(auto dest: bisim1-Throw-τExec-mover)
      hence τExec-mover-a P t {V:Ty=None; E} h (stk, loc, pc, None) ([Addr a], loc, pc', ⌊a⌋)
        by(auto intro: Block-τExecrI-None)
      moreover from bisim' have P, {V:Ty=None; E}, h ⊢ (Throw a, xs) ↔ ([Addr a], loc, pc', ⌊a⌋)
        by(auto intro: bisim1-bisims1.bisim1BlockThrowNone)
      ultimately show ?thesis using τ by auto
    qed
  qed
next
  case bisim1BlockThrowNone thus ?case by auto
next
  case (bisim1Sync1 e1 n e1' xs stk loc pc xcp e2 V)
  note IH = bisim1Sync1.IH(2)
  note bisim1 = ‹P,e1,h ⊢ (e1', xs) ↔ (stk, loc, pc, xcp)›
  note bisim2 = ‹⋀xs. P,e2,h ⊢ (e2, xs) ↔ ([], xs, 0, None)›
  note red = ‹True,P,t ⊢1 ⟨sync_V (e1') e2,(h, xs)⟩ −ta→ ⟨e',(h', xs')⟩›
  note bsok = ‹bsok (sync_V (e1) e2) n›
  hence [simp]: n = V by simp
  from red show ?case
  proof cases
    case (Synchronized1Red1 E1')
    note [simp] = ‹e' = sync_V (E1') e2›
      and red = ‹True,P,t ⊢1 ⟨e1', (h, xs)⟩ −ta→ ⟨E1', (h', xs')⟩›
   from red have τ: τmove1 P h (sync_V (e1') e2) = τmove1 P h e1' by(auto simp add: τmove1.simps
τmoves1.simps)
    moreover have call1 (sync_V (e1') e2) = call1 e1' by auto
    moreover from IH[OF red] bsok
    obtain pc'' stk'' loc'' xcp'' where bisim: P,e1,h' ⊢ (E1', xs') ↔ (stk'', loc'', pc'', xcp'')
      and redo: ?exec ta e1 e1' E1' h stk loc pc xcp h' pc'' stk'' loc'' xcp'' by auto
    from bisim
    have P,sync_V (e1) e2,h' ⊢ (sync_V (E1') e2, xs') ↔ (stk'', loc'', pc'', xcp'')
      by(rule bisim1-bisims1.bisim1Sync1)
    moreover {
      assume no-call2 e1 pc
      hence no-call2 (sync_V (e1) e2) pc by(auto simp add: no-call2-def) }
    ultimately show ?thesis using redo
        by(auto simp del: call1.simps calls1.simps split: if-split-asm split del: if-split)(blast intro:
```

Sync-τExecrI Sync-τExectI exec-move-SyncI1)+
next
case *Synchronized1Null*
note [*simp*] = ‹*e1′ = null*› ‹*e′ = THROW NullPointer*› ‹*ta = ε*› ‹*h′ = h*› ‹*xs′ = xs*[*V := Null*]›
and *V* = ‹*V < length xs*›
from *bisim1* **have** [*simp*]: *xcp = None xs = loc*
and *exec*: *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *xcp*) ([*Null*], *loc*, *length* (*compE2 e1*), *None*)
by(*auto dest*: *bisim1Val2D1*)
from *Sync-τExecrI*[*OF exec*]
have *τExec-mover-a P t* (*sync*$_V$ (*e1*) *e2*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*Null*], *loc*, *length* (*compE2 e1*), *None*) **by** *simp*
also from *V*
have *τExec-mover-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Null*], *loc*, *length* (*compE2 e1*), *None*) ([*Null*], *loc*[*V := Null*], *Suc* (*Suc* (*length* (*compE2 e1*))), *None*)
by −(*rule τExecr2step*,*auto intro*: *exec-instr τmove2-τmoves2.intros simp add*: *exec-move-def*)
also (*rtranclp-trans*)
have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Null*], *loc*[*V := Null*], *Suc* (*Suc* (*length* (*compE2 e1*))), *None*) *ε*
h ([*Null*], *loc*[*V := Null*], *Suc* (*Suc* (*length* (*compE2 e1*))), ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *exec-move-def* **by**(*rule exec-instr*) *auto*
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Null*] (*sync*$_V$ (*e1*) *e2*) (*Suc* (*Suc* (*length* (*compE2 e1*)))) *None*
by(*simp add*: *τmove2-iff*)
moreover
have *P*,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*THROW NullPointer*, *loc*[*V := Null*]) ↔ ([*Null*], (*loc*[*V := Null*]), *Suc* (*Suc* (*length* (*compE2 e1*))), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule bisim1Sync11*)
moreover have ¬ *τmove1 P h* (*sync*$_V$ (*null*) *e2*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **by** *auto blast*
next
case (*Lock1Synchronized a*)
note [*simp*] = ‹*e1′ = addr a*› ‹*ta =* {⌈*Lock→a*, *SyncLock a*⌉}› ‹*e′ = insync*$_V$ (*a*) *e2*› ‹*h′ = h*› ‹*xs′ = xs*[*V := Addr a*]›
and *V* = ‹*V < length xs*›
from *bisim1* **have** [*simp*]: *xcp = None xs = loc*
and *exec*: *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *xcp*) ([*Addr a*], *loc*, *length* (*compE2 e1*), *None*)
by(*auto dest*: *bisim1Val2D1*)
from *Sync-τExecrI*[*OF exec*]
have *τExec-mover-a P t* (*sync*$_V$ (*e1*) *e2*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*Addr a*], *loc*, *length* (*compE2 e1*), *None*) **by** *simp*
also from *V*
have *τExec-mover-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr a*], *loc*, *length* (*compE2 e1*), *None*) ([*Addr a*], *loc*[*V := Addr a*], *Suc* (*Suc* (*length* (*compE2 e1*))), *None*)
by −(*rule τExecr2step*,*auto intro*: *exec-instr τmove2-τmoves2.intros simp add*: *exec-move-def*)
also (*rtranclp-trans*)
have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr a*], *loc*[*V := Addr a*], *Suc* (*Suc* (*length* (*compE2 e1*))), *None*)
({⌈*Lock→a*, *SyncLock a*⌉})
h ([], *loc*[*V := Addr a*], *Suc* (*Suc* (*Suc* (*length* (*compE2 e1*)))), *None*)
unfolding *exec-move-def* **by** −(*rule exec-instr*, *auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Addr a*] (*sync*$_V$ (*e1*) *e2*) (*Suc* (*Suc* (*length* (*compE2 e1*)))) *None*
by(*simp add*: *τmove2-iff*)

moreover
from *bisim1Sync4*[*OF bisim1-refl, of P h V e1 e2 a loc*[*V* := *Addr a*]]
have *P*,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*insync*$_V$ (*a*) *e2*, *loc*[*V* := *Addr a*]) ↔ ([], *loc*[*V* := *Addr a*], *Suc* (*Suc* (*Suc* (*length* (*compE2 e1*)))), *None*) **by** *simp*
moreover have ¬ *τmove1 P h* (*sync*$_V$ (*addr a*) *e2*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **by**(*auto simp add*: *eval-nat-numeral ta-upd-simps*) *blast*
next
case (*Synchronized1Throw1 a*)
note [*simp*] = ‹*e1′* = *Throw a*› ‹*ta* = *ε*› ‹*e′* = *Throw a*› ‹*h′* = *h*› ‹*xs′* = *xs*›
have *τ*: *τmove1 P h* (*sync*$_V$ (*Throw a*) *e2*) **by**(*rule τmove1SyncThrow*)
from *bisim1* **have** *xcp* = ⌊*a*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*a*⌋
with *bisim1*
have *P*, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
by(*auto intro*: *bisim1SyncThrow*)
thus *?thesis* **using** *τ* **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim1* **obtain** *pc′*
where *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
and *bisim′*: *P*, *e1*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t* (*sync*$_V$ (*e1*) *e2*) *h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
by−(*rule Sync-τExecrI*)
moreover from *bisim′*
have *P*, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
by −(*rule bisim1-bisims1.bisim1SyncThrow*, *auto*)
ultimately show *?thesis* **using** *τ* **by** *fastforce*
qed
qed
next
case (*bisim1Sync2 e1 n e2 V v xs*)
note *bisim1* = ‹⋀*xs*. *P*,*e1*,*h* ⊢ (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim2* = ‹⋀*xs*. *P*,*e2*,*h* ⊢ (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*)›
from ‹*True*,*P*,*t* ⊢*1* ⟨*sync*$_V$ (*Val v*) *e2*,(*h*, *xs*)⟩ −*ta*→ ⟨*e′*,(*h′*, *xs′*)⟩› **show** *?case*
proof *cases*
case (*Lock1Synchronized a*)
note [*simp*] = ‹*v* = *Addr a*› ‹*ta* = {⌈*Lock*→*a*, *SyncLock a*⌉}› ‹*e′* = *insync*$_V$ (*a*) *e2*›
‹*h′* = *h*› ‹*xs′* = *xs*[*V* := *Addr a*]›
and *V* = ‹*V* < *length xs*›
from *V*
have *τExec-mover-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr a*, *Addr a*], *xs*, *Suc* (*length* (*compE2 e1*)), *None*) ([*Addr a*], *xs*[*V* := *Addr a*], *Suc* (*Suc* (*length* (*compE2 e1*))), *None*)
by −(*rule τExecr1step*,*auto intro*: *exec-instr simp add*: *τmove2-iff exec-move-def*)
moreover
have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr a*], *xs*[*V* := *Addr a*], *Suc* (*Suc* (*length* (*compE2 e1*))), *None*)
({⌈*Lock*→*a*, *SyncLock a*⌉})
h ([], *xs*[*V* := *Addr a*], *Suc* (*Suc* (*Suc* (*length* (*compE2 e1*)))), *None*)
unfolding *exec-move-def* **by** −(*rule exec-instr*, *auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Addr a*] (*sync*$_V$ (*e1*) *e2*) (*Suc* (*Suc* (*length* (*compE2 e1*)))) *None*

by(*simp add*: *τmove2-iff*)
moreover
from *bisim1Sync4*[*OF bisim1-refl*, *of P h V e1 e2 a xs*[*V* := *Addr a*]]
have *P*,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*insync*$_V$ (*a*) *e2*, *xs*[*V* := *Addr a*]) ↔ ([], *xs*[*V* := *Addr a*], *Suc* (*Suc* (*Suc* (*length* (*compE2 e1*)))), *None*) **by** *simp*
moreover have ¬ *τmove1 P h* (*sync*$_V$ (*addr a*) *e2*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **by**(*auto simp add*: *eval-nat-numeral ta-upd-simps*) *blast*
next
case *Synchronized1Null*
note [*simp*] = ‹*v* = *Null*› ‹*e'* = *THROW NullPointer*› ‹*ta* = *ε*› ‹*h'* = *h*› ‹*xs'* = *xs*[*V* := *Null*]›
and *V* = ‹*V* < *length xs*›
from *V*
have *τExec-mover-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Null*, *Null*], *xs*, *Suc* (*length* (*compE2 e1*)), *None*) ([*Null*], *xs*[*V* := *Null*], *Suc* (*Suc* (*length* (*compE2 e1*))), *None*)
by −(*rule τExecr1step*,*auto intro*: *exec-instr simp add*: *exec-move-def τmove2-iff*)
also have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Null*], *xs*[*V* := *Null*], *Suc* (*Suc* (*length* (*compE2 e1*))), *None*) *ε*
h ([*Null*], *xs*[*V* := *Null*], *Suc* (*Suc* (*length* (*compE2 e1*))), ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *exec-move-def* **by**(*rule exec-instr*) *auto*
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Null*] (*sync*$_V$ (*e1*) *e2*) (*Suc* (*Suc* (*length* (*compE2 e1*)))) *None*
by(*simp add*: *τmove2-iff*)
moreover
have *P*,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*THROW NullPointer*, *xs*[*V* := *Null*]) ↔ ([*Null*], (*xs*[*V* := *Null*]), *Suc* (*Suc* (*length* (*compE2 e1*))), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule bisim1Sync11*)
moreover have ¬ *τmove1 P h* (*sync*$_V$ (*null*) *e2*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **by**(*auto simp add*: *eval-nat-numeral*) *blast*
qed *auto*
next
case (*bisim1Sync3 e1 n e2 V v xs*)
note *bisim1* = ‹⋀*xs*. *P*,*e1*,*h* ⊢ (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim2* = ‹⋀*xs*. *P*,*e2*,*h* ⊢ (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*)›
from ‹*True*,*P*,*t* ⊢*1* ⟨*sync*$_V$ (*Val v*) *e2*,(*h*, *xs*)⟩ −*ta*→ ⟨*e'*,(*h'*, *xs'*)⟩› **show** *?case*
proof *cases*
case (*Lock1Synchronized a*)
note [*simp*] = ‹*v* = *Addr a*› ‹*ta* = {⌈*Lock*→*a*, *SyncLock a*⌉}› ‹*e'* = *insync*$_V$ (*a*) *e2*› ‹*h'* = *h*› ‹*xs'* = *xs*[*V* := *Addr a*]›
and *V* = ‹*V* < *length xs*›
have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr a*], *xs*[*V* := *Addr a*], *Suc* (*Suc* (*length* (*compE2 e1*))), *None*)
({⌈*Lock*→*a*, *SyncLock a*⌉})
h ([], *xs*[*V* := *Addr a*], *Suc* (*Suc* (*Suc* (*length* (*compE2 e1*)))), *None*)
unfolding *exec-move-def* **by** −(*rule exec-instr*, *auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Addr a*] (*sync*$_V$ (*e1*) *e2*) (*Suc* (*Suc* (*length* (*compE2 e1*)))) *None*
by(*simp add*: *τmove2-iff*)
moreover
from *bisim1Sync4*[*OF bisim1-refl*, *of P h V e1 e2 a xs*[*V* := *Addr a*]]
have *P*,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*insync*$_V$ (*a*) *e2*, *xs*[*V* := *Addr a*]) ↔ ([], *xs*[*V* := *Addr a*], *Suc* (*Suc* (*Suc* (*length* (*compE2 e1*)))), *None*) **by** *simp*
moreover have ¬ *τmove1 P h* (*sync*$_V$ (*addr a*) *e2*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **by**(*auto simp add*: *eval-nat-numeral ta-upd-simps*) *blast*

next
case *Synchronized1Null*
note [*simp*] = ‹*v* = *Null*› ‹*e'* = *THROW NullPointer*› ‹*ta* = *ε*› ‹*h'* = *h*› ‹*xs'* = *xs*[*V* := *Null*]›
and *V* = ‹*V* < *length xs*›
have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Null*], *xs*[*V* := *Null*], *Suc* (*Suc* (*length* (*compE2 e1*))), *None*) *ε*
h ([*Null*], *xs*[*V* := *Null*], *Suc* (*Suc* (*length* (*compE2 e1*))), ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *exec-move-def* **by**(*rule exec-instr*) *auto*
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Null*] (*sync*$_V$ (*e1*) *e2*) (*Suc* (*Suc* (*length* (*compE2 e1*)))) *None*
by(*simp add*: *τmove2-iff*)
moreover
have *P*,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*THROW NullPointer*, *xs*[*V* := *Null*]) ↔ ([*Null*], (*xs*[*V* := *Null*]), *Suc* (*Suc* (*length* (*compE2 e1*))), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule bisim1Sync11*)
moreover have ¬ *τmove1 P h* (*sync*$_V$ (*null*) *e2*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **by**(*auto simp add*: *eval-nat-numeral*) *blast*
qed *auto*
next
case (*bisim1Sync4 e2 n e2' xs stk loc pc xcp e1 V a*)
note *IH* = *bisim1Sync4.IH*(*2*)
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e2'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim1* = ‹⋀*xs*. *P*,*e1*,*h* ⊢ (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bsok* = ‹*bsok* (*sync*$_V$ (*e1*) *e2*) *n*›
note *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*insync*$_V$ (*a*) *e2'*,(*h*, *xs*)⟩ −*ta*→ ⟨*e'*,(*h'*, *xs'*)⟩›
from *red* **show** *?case*
proof *cases*
case (*Synchronized1Red2 E'*)
note [*simp*] = ‹*e'* = *insync*$_V$ (*a*) *E'*›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e2'*, (*h*, *xs*)⟩ −*ta*→ ⟨*E'*, (*h'*, *xs'*)⟩›
from *red* **have** *τ*: *τmove1 P h* (*insync*$_V$ (*a*) *e2'*) = *τmove1 P h e2'* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *IH*[*OF red*] *bsok* **obtain** *pc''* *stk''* *loc''* *xcp''*
where *bisim'*: *P*,*e2*,*h'* ⊢ (*E'*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
and *exec'*: *?exec ta e2 e2' E' h stk loc pc xcp h' pc'' stk'' loc'' xcp''* **by** *auto*
have *no-call2 e2 pc* ⟹ *no-call2* (*sync*$_V$(*e1*) *e2*) (*Suc* (*Suc* (*Suc* (*length* (*compE2 e1*) + *pc*))))
by(*auto simp add*: *no-call2-def*)
hence *?exec ta* (*sync*$_V$ (*e1*) *e2*) (*insync*$_V$ (*a*) *e2'*) (*insync*$_V$ (*a*) *E'*) *h stk loc* (*Suc* (*Suc* (*Suc* (*length* (*compE2 e1*) + *pc*)))) *xcp h'* (*Suc* (*Suc* (*Suc* (*length* (*compE2 e1*) + *pc''*)))) *stk'' loc'' xcp''*
using *exec' τ*
by(*cases τmove1 P h* (*insync*$_V$ (*a*) *e2'*))(*auto*,(*blast intro*: *exec-move-SyncI2 Insync-τExecrI Insync-τExectI*)+)
moreover from *bisim'*
have *P*,*sync*$_V$ (*e1*) *e2*,*h'* ⊢ (*insync*$_V$ (*a*) *E'*, *xs'*) ↔ (*stk''*, *loc''*, (*Suc* (*Suc* (*Suc* (*length* (*compE2 e1*) + *pc''*)))), *xcp''*)
by(*rule bisim1-bisims1.bisim1Sync4*)
ultimately show *?thesis* **using** *τ* **by** *auto blast*+
next
case (*Unlock1Synchronized a' v*)
note [*simp*] = ‹*e2'* = *Val v*› ‹*e'* = *Val v*› ‹*ta* = {⌈*Unlock*→*a'*, *SyncUnlock a'*⌉}› ‹*h'* = *h*› ‹*xs'* = *xs*›
and *V* = ‹*V* < *length xs*› **and** *xsV* = ‹*xs* ! *V* = *Addr a'*›
from *bisim2* **have** [*simp*]: *xcp* = *None xs* = *loc*
and *exec*: *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 e2*), *None*)

by(*auto dest*: *bisim1Val2D1*)
let *?pc1* = (*Suc* (*Suc* (*Suc* (*length* (*compE2 e1*) + *length* (*compE2 e2*)))))
note *Insync-τExecrI*[*OF exec*, *of V e1*]
also from *V xsV* **have** *τExec-mover-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*v*], *loc*, *?pc1*, *None*) ([*Addr a′*, *v*], *loc*, *Suc ?pc1*, *None*)
by −(*rule τExecr1step*,*auto simp add*: *exec-move-def intro*: *exec-instr τmove2-τmoves2.intros*)
also (*rtranclp-trans*)
have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr a′*, *v*], *loc*, *Suc ?pc1*, *None*) ({|*Unlock→a′*, *SyncUnlock a′*|}) *h* ([*v*], *loc*, *Suc* (*Suc ?pc1*), *None*)
unfolding *exec-move-def* **by**(*rule exec-instr*)(*auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Addr a′*, *v*] (*sync*$_V$ (*e1*) *e2*) (*Suc ?pc1*) *None* **by**(*simp add*: *τmove2-iff*)
moreover
from *bisim1Sync6*[*of P h V e1 e2 v xs*]
have *P*,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*Val v*, *xs*) ↔ ([*v*], *xs*, *Suc* (*Suc ?pc1*), *None*)
by(*auto simp add*: *eval-nat-numeral*)
moreover have ¬ *τmove1 P h* (*insync*$_V$ (*a*) *e2′*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **by**(*auto simp add*: *ta-upd-simps*) *blast*
next
case (*Unlock1SynchronizedNull v*)
note [*simp*] = ‹*e2′* = *Val v*› ‹*e′* = *THROW NullPointer*› ‹*ta* = *ε*› ‹*h′* = *h*› ‹*xs′* = *xs*›
and *V* = ‹*V* < *length xs*› **and** *xsV* = ‹*xs* ! *V* = *Null*›
from *bisim2* **have** [*simp*]: *xcp* = *None xs* = *loc*
and *exec*: *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 e2*), *None*)
by(*auto dest*: *bisim1Val2D1*)
let *?pc1* = (*Suc* (*Suc* (*Suc* (*length* (*compE2 e1*) + *length* (*compE2 e2*)))))
note *Insync-τExecrI*[*OF exec*, *of V e1*]
also from *V xsV* **have** *τExec-mover-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*v*], *loc*, *?pc1*, *None*) ([*Null*, *v*], *loc*, *Suc ?pc1*, *None*)
by −(*rule τExecr1step*,*auto intro*: *exec-instr τmove2-τmoves2.intros simp add*: *exec-move-def*)
also (*rtranclp-trans*)
have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Null*, *v*], *loc*, *Suc ?pc1*, *None*) *ε h* ([*Null*, *v*], *loc*, *Suc ?pc1*, ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *exec-move-def* **by**(*rule exec-instr*)(*auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Null*, *v*] (*sync*$_V$ (*e1*) *e2*) (*Suc ?pc1*) *None* **by**(*simp add*: *τmove2-iff*)
moreover
from *bisim1Sync12*[*of P h V e1 e2 addr-of-sys-xcpt NullPointer xs*]
have *P*,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*THROW NullPointer*,*xs*) ↔ ([*Null*, *v*],*xs*,*Suc ?pc1*,⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*auto simp add*: *eval-nat-numeral*)
moreover have ¬ *τmove1 P h* (*insync*$_V$ (*a*) *e2′*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **by** *auto blast*
next
case (*Unlock1SynchronizedFail a′ v*)
note [*simp*] = ‹*e2′* = *Val v*› ‹*e′* = *THROW IllegalMonitorState*› ‹*ta* = {|*UnlockFail→a′*|}› ‹*xs′* = *xs*› ‹*h′* = *h*›
and *V* = ‹*V* < *length xs*› **and** *xsV* = ‹*xs* ! *V* = *Addr a′*›
from *bisim2* **have** [*simp*]: *xcp* = *None xs* = *loc*
and *exec*: *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 e2*), *None*)
by(*auto dest*: *bisim1Val2D1*)
let *?pc1* = (*Suc* (*Suc* (*Suc* (*length* (*compE2 e1*) + *length* (*compE2 e2*)))))
note *Insync-τExecrI*[*OF exec*, *of V e1*]
also from *V xsV* **have** *τExec-mover-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*v*], *loc*, *?pc1*, *None*) ([*Addr a′*, *v*],

loc, *Suc ?pc1*, *None*)
by −(*rule τExecr1step*,*auto intro*: *exec-instr τmove2-τmoves2.intros simp add*: *exec-move-def*)
also (*rtranclp-trans*)
have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr a'*, *v*], *loc*, *Suc ?pc1*, *None*) ⦃*UnlockFail→a'*⦄ *h* ([*Addr a'*, *v*], *loc*, *Suc ?pc1*, ⌊*addr-of-sys-xcpt IllegalMonitorState*⌋)
unfolding *exec-move-def* **by**(*rule exec-instr*)(*auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Addr a'*, *v*] (*sync*$_V$ (*e1*) *e2*) (*Suc ?pc1*) *None* **by**(*simp add*: *τmove2-iff*)
moreover
from *bisim1Sync12*[*of P h V e1 e2 addr-of-sys-xcpt IllegalMonitorState xs Addr a' v*]
have *P*,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*THROW IllegalMonitorState*,*xs*) ↔ ([*Addr a'*, *v*],*xs*,*Suc ?pc1*,⌊*addr-of-sys-xcpt IllegalMonitorState*⌋)
by(*auto simp add*: *eval-nat-numeral*)
moreover have ¬ *τmove1 P h* (*insync*$_V$ (*a*) *Val v*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **by**(*auto simp add*: *ta-upd-simps*) *blast*
next
case (*Synchronized1Throw2 a' ad*)
note [*simp*] = ‹*e2'* = *Throw ad*› ‹*ta* = ⦃*Unlock→a'*, *SyncUnlock a'*⦄› ‹*e'* = *Throw ad*›
‹*h'* = *h*› ‹*xs'* = *xs*› **and** *xsV* = ‹*xs* ! *V* = *Addr a'*› **and** *V* = ‹*V* < *length xs*›
let *?pc* = *6* + *length* (*compE2 e1*) + *length* (*compE2 e2*)
let *?stk* = *Addr ad* # *drop* (*size stk* − *0*) *stk*
from *bisim2* **have** [*simp*]: *xs* = *loc* **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim2*
have *τExec-movet-a P t* (*sync*$_V$ (*e1*) *e2*) *h* (*stk*, *loc*, *Suc* (*Suc* (*Suc* (*length* (*compE2 e1*) + *pc*))), *xcp*) ([*Addr ad*], *loc*, *?pc*, *None*)
by(*auto intro*: *bisim1-insync-Throw-exec*)
also from *xsV V*
have *τExec-movet-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr ad*], *loc*, *?pc*, *None*) ([*Addr a'*, *Addr ad*], *loc*, *Suc ?pc*, *None*)
by −(*rule τExect1step*,*auto intro*: *exec-instr τmove2Sync7 simp add*: *exec-move-def*)
also (*tranclp-trans*)
have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr a'*, *Addr ad*], *loc*, *Suc ?pc*, *None*) (⦃*Unlock→a'*, *SyncUnlock a'*⦄) *h* ([*Addr ad*], *loc*, *Suc* (*Suc ?pc*), *None*)
unfolding *exec-move-def* **by**(*rule exec-instr*)(*auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Addr a'*, *Addr ad*] (*sync*$_V$ (*e1*) *e2*) (*Suc ?pc*) *None* **by**(*simp add*: *τmove2-iff*)
moreover
hence *P*, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*Throw ad*, *loc*) ↔ ([*Addr ad*], *loc*, *8* + *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
by(*auto intro*: *bisim1Sync9*)
moreover have ¬ *τmove1 P h* (*insync*$_V$ (*a*) *Throw ad*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **by**(*auto simp add*: *add.assoc ta-upd-simps*)(*blast intro*: *tranclp-into-rtranclp*)
next
case (*Synchronized1Throw2Fail a' ad*)
note [*simp*] = ‹*e2'* = *Throw ad*› ‹*ta* = ⦃*UnlockFail→a'*⦄› ‹*e'* = *THROW IllegalMonitorState*› ‹*h'* = *h*› ‹*xs'* = *xs*›
and *xsV* = ‹*xs* ! *V* = *Addr a'*› **and** *V* = ‹*V* < *length xs*›
let *?pc* = *6* + *length* (*compE2 e1*) + *length* (*compE2 e2*)
let *?stk* = *Addr ad* # *drop* (*size stk* − *0*) *stk*
from *bisim2* **have** [*simp*]: *xs* = *loc* **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim2*
have *τExec-movet-a P t* (*sync*$_V$ (*e1*) *e2*) *h* (*stk*, *loc*, *Suc* (*Suc* (*Suc* (*length* (*compE2 e1*) + *pc*))), *xcp*) ([*Addr ad*], *loc*, *?pc*, *None*)

by(*auto intro*: *bisim1-insync-Throw-exec*)
also from *xsV V*
have *τExec-movet-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr ad*], *loc*, *?pc*, *None*) ([*Addr a′*, *Addr ad*], *loc*, *Suc ?pc*, *None*)
by −(*rule τExect1step*,*auto intro*: *exec-instr τmove2Sync7 simp add*: *exec-move-def*)
also (*tranclp-trans*)
have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr a′*, *Addr ad*], *loc*, *Suc ?pc*, *None*) {|*UnlockFail→a′*|} *h* ([*Addr a′*, *Addr ad*], *loc*, *Suc ?pc*, ⌊*addr-of-sys-xcpt IllegalMonitorState*⌋)
unfolding *exec-move-def* **by**(*rule exec-instr*)(*auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Addr a′*, *Addr ad*] (*sync*$_V$ (*e1*) *e2*) (*Suc ?pc*) *None* **by**(*simp add*: *τmove2-iff*)
moreover
hence *P*, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*THROW IllegalMonitorState*, *loc*) ↔ ([*Addr a′*, *Addr ad*], *loc*, *7* + *length* (*compE2 e1*) + *length* (*compE2 e2*), ⌊*addr-of-sys-xcpt IllegalMonitorState*⌋)
by(*auto intro*: *bisim1Sync14*)
moreover have ¬ *τmove1 P h* (*insync*$_V$ (*a*) *e2′*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **by**(*auto simp add*: *add.assoc ta-upd-simps*)(*blast intro*: *tranclp-into-rtranclp*)
next
case (*Synchronized1Throw2Null ad*)
note [*simp*] = ‹*e2′* = *Throw ad*› ‹*ta* = *ε*› ‹*e′* = *THROW NullPointer*› ‹*h′* = *h*› ‹*xs′* = *xs*›
and *xsV* = ‹*xs* ! *V* = *Null*› **and** *V* = ‹*V* < *length xs*›
let *?pc* = *6* + *length* (*compE2 e1*) + *length* (*compE2 e2*)
let *?stk* = *Addr ad* # *drop* (*size stk* − *0*) *stk*
from *bisim2* **have** [*simp*]: *xs* = *loc* **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim2*
have *τExec-movet-a P t* (*sync*$_V$ (*e1*) *e2*) *h* (*stk*, *loc*, *Suc* (*Suc* (*Suc* (*length* (*compE2 e1*) + *pc*))), *xcp*) ([*Addr ad*], *loc*, *?pc*, *None*)
by(*auto intro*: *bisim1-insync-Throw-exec*)
also from *xsV V*
have *τExec-movet-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr ad*], *loc*, *?pc*, *None*) ([*Null*, *Addr ad*], *loc*, *Suc ?pc*, *None*)
by −(*rule τExect1step*,*auto intro*: *exec-instr simp add*: *exec-move-def τmove2-iff*)
also (*tranclp-trans*)
have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Null*, *Addr ad*], *loc*, *Suc ?pc*, *None*) *ε h* ([*Null*, *Addr ad*], *loc*, *Suc ?pc*, ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *exec-move-def* **by**(*rule exec-instr*)(*auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Null*, *Addr ad*] (*sync*$_V$ (*e1*) *e2*) (*Suc ?pc*) *None* **by**(*simp add*: *τmove2-iff*)
moreover
hence *P*, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*THROW NullPointer*, *loc*) ↔ ([*Null*, *Addr ad*], *loc*, *7* + *length* (*compE2 e1*) + *length* (*compE2 e2*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*auto intro*: *bisim1Sync14*)
moreover have ¬ *τmove1 P h* (*insync*$_V$ (*a*) *e2′*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **by**(*auto simp add*: *add.assoc*)(*blast intro*: *tranclp-into-rtranclp*)
qed
next
case (*bisim1Sync5 e1 n e2 V a v xs*)
note *bisim2* = ‹⋀*xs*. *P*,*e2*,*h* ⊢ (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim1* = ‹⋀*xs*. *P*,*e1*,*h* ⊢ (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*)›
from ‹*True*,*P*,*t* ⊢*1* ⟨*insync*$_V$ (*a*) *Val v*,(*h*, *xs*)⟩ −*ta*→ ⟨*e′*,(*h′*, *xs′*)⟩› **show** *?case*
proof *cases*
case (*Unlock1Synchronized a′*)
note [*simp*] = ‹*e′* = *Val v*› ‹*ta* = {|*Unlock→a′*, *SyncUnlock a′*|}› ‹*h′* = *h*› ‹*xs′* = *xs*›
and *V* = ‹*V* < *length xs*› **and** *xsV* = ‹*xs* ! *V* = *Addr a′*›

let *?pc1 = 4 + length (compE2 e1) + length (compE2 e2)*
have *exec-move-a P t (sync$_V$ (e1) e2) h ([Addr a', v], xs, ?pc1, None) ⦃Unlock→a', SyncUnlock a'⦄ h ([v], xs, Suc ?pc1, None)*
unfolding *exec-move-def* **by**(*rule exec-instr*)(*auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2 (compP2 P) h [Addr a', v] (sync$_V$ (e1) e2) ?pc1 None* **by**(*simp add*: *τmove2-iff*)
moreover
from *bisim1Sync6*[*of P h V e1 e2 v xs*]
have *P,sync$_V$ (e1) e2,h ⊢ (Val v, xs) ↔ ([v], xs, Suc ?pc1, None)*
by(*auto simp add*: *eval-nat-numeral*)
moreover have ¬ *τmove1 P h (insync$_V$ (a) Val v)* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **using** *xsV* **by**(*auto simp add*: *eval-nat-numeral ta-upd-simps*) *blast*
next
case *Unlock1SynchronizedNull*
note [*simp*] = ‹*e' = THROW NullPointer*› ‹*ta = ε*› ‹*h' = h*› ‹*xs' = xs*›
and *V = ‹V < length xs›* **and** *xsV = ‹xs ! V = Null›*
let *?pc1 = 4 + length (compE2 e1) + length (compE2 e2)*
have *exec-move-a P t (sync$_V$ (e1) e2) h ([Null, v], xs, ?pc1, None) ε h ([Null, v], xs, ?pc1, ⌊addr-of-sys-xcpt NullPointer⌋)*
unfolding *exec-move-def* **by**(*rule exec-instr*)(*auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2 (compP2 P) h [Null, v] (sync$_V$ (e1) e2) ?pc1 None* **by**(*simp add*: *τmove2-iff*)
moreover
from *bisim1Sync12*[*of P h V e1 e2 addr-of-sys-xcpt NullPointer xs Null v*]
have *P,sync$_V$ (e1) e2,h ⊢ (THROW NullPointer,xs) ↔ ([Null, v],xs,?pc1,⌊addr-of-sys-xcpt NullPointer⌋)*
by(*auto simp add*: *eval-nat-numeral*)
moreover have ¬ *τmove1 P h (insync$_V$ (a) Val v)* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **using** *xsV* **by**(*auto simp add*: *eval-nat-numeral*) *blast*
next
case (*Unlock1SynchronizedFail a'*)
note [*simp*] = ‹*e' = THROW IllegalMonitorState*› ‹*ta = ⦃UnlockFail→a'⦄*› ‹*xs' = xs*› ‹*h' = h*›
and *V = ‹V < length xs›* **and** *xsV = ‹xs ! V = Addr a'›*
let *?pc1 = 4 + length (compE2 e1) + length (compE2 e2)*
have *exec-move-a P t (sync$_V$ (e1) e2) h ([Addr a', v], xs, ?pc1, None) ⦃UnlockFail→a'⦄ h ([Addr a', v], xs, ?pc1, ⌊addr-of-sys-xcpt IllegalMonitorState⌋)*
unfolding *exec-move-def* **by**(*rule exec-instr*)(*auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2 (compP2 P) h [Addr a', v] (sync$_V$ (e1) e2) ?pc1 None* **by**(*simp add*: *τmove2-iff*)
moreover
from *bisim1Sync12*[*of P h V e1 e2 addr-of-sys-xcpt IllegalMonitorState xs Addr a' v*]
have *P,sync$_V$ (e1) e2,h ⊢ (THROW IllegalMonitorState,xs) ↔ ([Addr a', v],xs,?pc1,⌊addr-of-sys-xcpt IllegalMonitorState⌋)*
by(*auto simp add*: *eval-nat-numeral*)
moreover have ¬ *τmove1 P h (insync$_V$ (a) Val v)* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **using** *xsV* **by**(*auto simp add*: *eval-nat-numeral ta-upd-simps*) *blast*
qed *auto*
next
case *bisim1Sync6* **thus** *?case* **by** *auto*
next
case (*bisim1Sync7 e1 n e2 V a ad xs*)
note *bisim2* = ‹⋀*xs. P,e2,h ⊢ (e2, xs) ↔ ([], xs, 0, None)*›
note *bisim1* = ‹⋀*xs. P,e1,h ⊢ (e1, xs) ↔ ([], xs, 0, None)*›
from ‹*True,P,t ⊢1 ⟨insync$_V$ (a) Throw ad,(h, xs)⟩ −ta→ ⟨e',(h', xs')⟩*› **show** *?case*

proof *cases*
case (*Synchronized1Throw2 a'*)
note [*simp*] = ‹*ta* = {⌊*Unlock*→*a'*, *SyncUnlock a'*⌋} › ‹*e'* = *Throw ad*› ‹*h'* = *h*› ‹*xs'* = *xs*›
and *xsV* = ‹*xs* ! *V* = *Addr a'*› **and** *V* = ‹*V* < *length xs*›
let *?pc* = *6* + *length* (*compE2 e1*) + *length* (*compE2 e2*)
from *xsV V*
have *τExec-mover-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr ad*], *xs*, *?pc*, *None*) ([*Addr a'*, *Addr ad*], *xs*, *Suc ?pc*, *None*)
by −(*rule τExecr1step,auto intro*: *exec-instr simp add*: *exec-move-def τmove2-iff*)
moreover have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr a'*, *Addr ad*], *xs*, *Suc ?pc*, *None*) {⌊*Unlock*→*a'*, *SyncUnlock a'*⌋} *h* ([*Addr ad*], *xs*, *Suc* (*Suc ?pc*), *None*)
unfolding *exec-move-def* **by**(*rule exec-instr*)(*auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Addr a'*, *Addr ad*] (*sync*$_V$ (*e1*) *e2*) (*Suc ?pc*) *None* **by**(*simp add*: *τmove2-iff*)
moreover
have *P*, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*Throw ad*, *xs*) ↔ ([*Addr ad*], *xs*, *8* + *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
by(*auto intro*: *bisim1Sync9*)
moreover have ¬ *τmove1 P h* (*insync*$_V$ (*a*) *Throw ad*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **by**(*auto simp add*: *add.assoc eval-nat-numeral ta-upd-simps*) *blast*
next
case (*Synchronized1Throw2Fail a'*)
note [*simp*] = ‹*ta* = {⌊*UnlockFail*→*a'*⌋} › ‹*e'* = *THROW IllegalMonitorState*› ‹*h'* = *h*› ‹*xs'* = *xs*›
and *xsV* = ‹*xs* ! *V* = *Addr a'*› **and** *V* = ‹*V* < *length xs*›
let *?pc* = *6* + *length* (*compE2 e1*) + *length* (*compE2 e2*)
from *xsV V*
have *τExec-mover-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr ad*], *xs*, *?pc*, *None*) ([*Addr a'*, *Addr ad*], *xs*, *Suc ?pc*, *None*)
by −(*rule τExecr1step,auto intro*: *exec-instr simp add*: *exec-move-def τmove2-iff*)
moreover have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr a'*, *Addr ad*], *xs*, *Suc ?pc*, *None*) {⌊*UnlockFail*→*a'*⌋} *h* ([*Addr a'*, *Addr ad*], *xs*, *Suc ?pc*, ⌊*addr-of-sys-xcpt IllegalMonitorState*⌋)
unfolding *exec-move-def* **by**(*rule exec-instr*)(*auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Addr a'*, *Addr ad*] (*sync*$_V$ (*e1*) *e2*) (*Suc ?pc*) *None* **by**(*simp add*: *τmove2-iff*)
moreover
have *P*, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*THROW IllegalMonitorState*, *xs*) ↔ ([*Addr a'*, *Addr ad*], *xs*, *7* + *length* (*compE2 e1*) + *length* (*compE2 e2*), ⌊*addr-of-sys-xcpt IllegalMonitorState*⌋)
by(*auto intro*: *bisim1Sync14*)
moreover have ¬ *τmove1 P h* (*insync*$_V$ (*a*) *Throw ad*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **by**(*auto simp add*: *add.assoc ta-upd-simps*) *blast*
next
case *Synchronized1Throw2Null*
note [*simp*] = ‹*ta* = *ε*› ‹*e'* = *THROW NullPointer*› ‹*h'* = *h*› ‹*xs'* = *xs*›
and *xsV* = ‹*xs* ! *V* = *Null*› **and** *V* = ‹*V* < *length xs*›
let *?pc* = *6* + *length* (*compE2 e1*) + *length* (*compE2 e2*)
from *xsV V*
have *τExec-mover-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr ad*], *xs*, *?pc*, *None*) ([*Null*, *Addr ad*], *xs*, *Suc ?pc*, *None*)
by −(*rule τExecr1step,auto intro*: *exec-instr simp add*: *exec-move-def τmove2-iff*)
moreover have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Null*, *Addr ad*], *xs*, *Suc ?pc*, *None*) *ε h* ([*Null*, *Addr ad*], *xs*, *Suc ?pc*, ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *exec-move-def* **by**(*rule exec-instr*)(*auto simp add*: *is-Ref-def*)

moreover have ¬ *τmove2* (*compP2 P*) *h* [*Null*, *Addr ad*] (*sync*$_V$ (*e1*) *e2*) (*Suc ?pc*) *None* **by**(*simp add*: *τmove2-iff*)
moreover
have *P*, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*THROW NullPointer*, *xs*) ↔ ([*Null*, *Addr ad*], *xs*, *7* + *length* (*compE2 e1*) + *length* (*compE2 e2*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*auto intro*: *bisim1Sync14*)
moreover have ¬ *τmove1 P h* (*insync*$_V$ (*a*) *Throw ad*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **by**(*auto simp add*: *add.assoc*) *blast*
qed *auto*
next
case (*bisim1Sync8 e1 n e2 V a ad xs*)
note *bisim2* = ‹⋀*xs*. *P*,*e2*,*h* ⊢ (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim1* = ‹⋀*xs*. *P*,*e1*,*h* ⊢ (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*)›
from ‹*True*,*P*,*t* ⊢*1* ⟨*insync*$_V$ (*a*) *Throw ad*,(*h*, *xs*)⟩ −*ta*→ ⟨*e'*,(*h'*, *xs'*)⟩› **show** *?case*
proof *cases*
case (*Synchronized1Throw2 a'*)
note [*simp*] = ‹*ta* = {⌈*Unlock*→*a'*, *SyncUnlock a'*⌉}› ‹*e'* = *Throw ad*› ‹*h'* = *h*› ‹*xs'* = *xs*›
and *xsV* = ‹*xs* ! *V* = *Addr a'*› **and** *V* = ‹*V* < *length xs*›
let *?pc* = *7* + *length* (*compE2 e1*) + *length* (*compE2 e2*)
have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr a'*, *Addr ad*], *xs*, *?pc*, *None*) {⌈*Unlock*→*a'*, *SyncUnlock a'*⌉} *h* ([*Addr ad*], *xs*, *Suc ?pc*, *None*)
unfolding *exec-move-def* **by**(*rule exec-instr*)(*auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Addr a'*, *Addr ad*] (*sync*$_V$ (*e1*) *e2*) *?pc None* **by**(*simp add*: *τmove2-iff*)
moreover
have *P*, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*Throw ad*, *xs*) ↔ ([*Addr ad*], *xs*, *8* + *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
by(*auto intro*: *bisim1Sync9*)
moreover have ¬ *τmove1 P h* (*insync*$_V$ (*a*) *Throw ad*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **using** *xsV* **by**(*auto simp add*: *add.assoc eval-nat-numeral ta-upd-simps*) *blast*
next
case (*Synchronized1Throw2Fail a'*)
note [*simp*] = ‹*ta* = {⌈*UnlockFail*→*a'*⌉}› ‹*e'* = *THROW IllegalMonitorState*› ‹*h'* = *h*› ‹*xs'* = *xs*›
and *xsV* = ‹*xs* ! *V* = *Addr a'*› **and** *V* = ‹*V* < *length xs*›
let *?pc* = *7* + *length* (*compE2 e1*) + *length* (*compE2 e2*)
have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Addr a'*, *Addr ad*], *xs*, *?pc*, *None*) {⌈*UnlockFail*→*a'*⌉} *h* ([*Addr a'*, *Addr ad*], *xs*, *?pc*, ⌊*addr-of-sys-xcpt IllegalMonitorState*⌋)
unfolding *exec-move-def* **by**(*rule exec-instr*)(*auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Addr a'*, *Addr ad*] (*sync*$_V$ (*e1*) *e2*) *?pc None* **by**(*simp add*: *τmove2-iff*)
moreover
have *P*, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*THROW IllegalMonitorState*, *xs*) ↔ ([*Addr a'*, *Addr ad*], *xs*, *?pc*, ⌊*addr-of-sys-xcpt IllegalMonitorState*⌋)
by(*auto intro*: *bisim1Sync14*)
moreover have ¬ *τmove1 P h* (*insync*$_V$ (*a*) *Throw ad*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **using** *xsV* **by**(*auto simp add*: *add.assoc ta-upd-simps*) *blast*
next
case *Synchronized1Throw2Null*
note [*simp*] = ‹*ta* = ε› ‹*e'* = *THROW NullPointer*› ‹*h'* = *h*› ‹*xs'* = *xs*›
and *xsV* = ‹*xs* ! *V* = *Null*› **and** *V* = ‹*V* < *length xs*›

let *?pc* = *7* + *length* (*compE2 e1*) + *length* (*compE2 e2*)
have *exec-move-a P t* (*sync*$_V$ (*e1*) *e2*) *h* ([*Null*, *Addr ad*], *xs*, *?pc*, *None*) *ε h* ([*Null*, *Addr ad*], *xs*, *?pc*, ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *exec-move-def* **by**(*rule exec-instr*)(*auto simp add*: *is-Ref-def*)
moreover have ¬ *τmove2* (*compP2 P*) *h* [*Null*, *Addr ad*] (*sync*$_V$ (*e1*) *e2*) *?pc None* **by**(*simp add*: *τmove2-iff*)
moreover
have *P*, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*THROW NullPointer*, *xs*) ↔ ([*Null*, *Addr ad*], *xs*, *?pc*, ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*auto intro*: *bisim1Sync14*)
moreover have ¬ *τmove1 P h* (*insync*$_V$ (*a*) *Throw ad*) **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
ultimately show *?thesis* **using** *xsV* **by**(*auto simp add*: *add.assoc*) *blast*
qed *auto*
next
case *bisim1Sync9* **thus** *?case* **by** *auto*
next
case *bisim1Sync10* **thus** *?case* **by** *auto*
next
case *bisim1Sync11* **thus** *?case* **by** *auto*
next
case *bisim1Sync12* **thus** *?case* **by** *auto*
next
case *bisim1Sync14* **thus** *?case* **by** *auto*
next
case *bisim1SyncThrow* **thus** *?case* **by** *auto*
next
case *bisim1InSync* **thus** *?case* **by** *simp*
next
case (*bisim1Seq1 e1 n e1′ xs stk loc pc xcp e2*)
note *IH* = *bisim1Seq1.IH*(*2*)
note *bisim1* = ‹*P*,*e1*,*h* ⊢ (*e1′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim2* = ‹⋀*xs*. *P*,*e2*,*h* ⊢ (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e1′*;; *e2*,(*h*, *xs*)⟩ −*ta*→ ⟨*e′*,(*h′*, *xs′*)⟩›
note *bsok* = ‹*bsok* (*e1*;; *e2*) *n*›
from *red* **show** *?case*
proof *cases*
case (*Seq1Red E′*)
note [*simp*] = ‹*e′* = *E′*;;*e2*›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e1′*, (*h*, *xs*)⟩ −*ta*→ ⟨*E′*, (*h′*, *xs′*)⟩›
from *red* **have** *τ*: *τmove1 P h* (*e1′*;; *e2*) = *τmove1 P h e1′* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
moreover have *call1* (*e1′*;; *e2*) = *call1 e1′* **by** *auto*
moreover from *IH*[*OF red*] *bsok*
obtain *pc″ stk″ loc″ xcp″* **where** *bisim*: *P*,*e1*,*h′* ⊢ (*E′*, *xs′*) ↔ (*stk″*, *loc″*, *pc″*, *xcp″*)
and *redo*: *?exec ta e1 e1′ E′ h stk loc pc xcp h′ pc″ stk″ loc″ xcp″* **by** *auto*
from *bisim* **have** *P*,*e1*;; *e2*,*h′* ⊢ (*E′*;; *e2*, *xs′*) ↔ (*stk″*, *loc″*, *pc″*, *xcp″*)
by(*rule bisim1-bisims1.bisim1Seq1*)
moreover {
assume *no-call2 e1 pc*
hence *no-call2* (*e1*;; *e2*) *pc* **by**(*auto simp add*: *no-call2-def*) }
ultimately show *?thesis* **using** *redo*
by(*auto simp del*: *call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)(*blast intro*: *Seq-τExecrI1 Seq-τExectI1 exec-move-SeqI1*)+

next
case (*Red1Seq v*)
note [*simp*] = ‹*e1'* = *Val v*› ‹*ta* = *ε*› ‹*h'* = *h*› ‹*xs'* = *xs*› ‹*e'* = *e2*›
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 e1*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*e1*;; *e2*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 e1*), *None*)
by−(*rule Seq-τExecrI1*)
moreover have *exec-move-a P t* (*e1*;; *e2*) *h* ([*v*], *loc*, *length* (*compE2 e1*), *None*) *ε h* ([], *loc*, *Suc* (*length* (*compE2 e1*)), *None*)
unfolding *exec-move-def* **by**(*rule exec-instr*, *auto*)
moreover have *τmove2* (*compP2 P*) *h* [*v*] (*e1*;;*e2*) (*length* (*compE2 e1*)) *None* **by**(*simp add*: *τmove2-iff*)
ultimately have *τExec-mover-a P t* (*e1*;; *e2*) *h* (*stk*, *loc*, *pc*, *xcp*) ([], *loc*, *Suc* (*length* (*compE2 e1*)), *None*)
by(*auto intro*: *rtranclp.rtrancl-into-rtrancl τexec-moveI simp add*: *compP2-def*)
moreover from *bisim1-refl*
have *P*, *e1*;; *e2*, *h* ⊢ (*e2*, *xs*) ↔ ([], *loc*, *Suc* (*length* (*compE2 e1*) + *0*), *None*)
unfolding *s* **by**(*rule bisim1Seq2*)
moreover have *τmove1 P h* (*Val v*;; *e2*) **by**(*rule τmove1SeqRed*)
ultimately show *?thesis* **by**(*auto*)
next
case (*Seq1Throw a*)
note [*simp*] = ‹*e1'* = *Throw a*› ‹*ta* = *ε*› ‹*e'* = *Throw a*› ‹*h'* = *h*› ‹*xs'* = *xs*›
have *τ*: *τmove1 P h* (*Throw a*;; *e2*) **by**(*rule τmove1SeqThrow*)
from *bisim1* **have** *xcp* = ⌊*a*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*a*⌋
with *bisim1* **have** *P*, *e1*;; *e2*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
by(*auto intro*: *bisim1SeqThrow1*)
thus *?thesis* **using** *τ* **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim1* **obtain** *pc'*
where *τExec-mover-a P t e1 h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc'*, ⌊*a*⌋)
and *bisim'*: *P*, *e1*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc'*, ⌊*a*⌋) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t* (*e1*;;*e2*) *h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc'*, ⌊*a*⌋)
by−(*rule Seq-τExecrI1*)
moreover from *bisim'*
have *P*, *e1*;;*e2*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc'*, ⌊*a*⌋)
by(*rule bisim1SeqThrow1*)
ultimately show *?thesis* **using** *τ* **by** *auto*
qed
qed
next
case *bisim1SeqThrow1* **thus** *?case* **by** *fastforce*
next
case (*bisim1Seq2 e2 n e2' xs stk loc pc xcp e1*)
note *IH* = *bisim1Seq2.IH*(*2*)
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e2'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim1* = ‹⋀*xs*. *P*,*e1*,*h* ⊢ (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e2'*,(*h*, *xs*)⟩ −*ta*→ ⟨*e'*,(*h'*, *xs'*)⟩›

note *bsok* = ‹*bsok* (*e1*;; *e2*) *n*›
from *IH*[*OF red*] *bsok* **obtain** *pc''* *stk''* *loc''* *xcp''*
where *bisim'*: *P*,*e2*,*h'* ⊢ (*e'*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
and *exec'*: *?exec ta e2 e2' e' h stk loc pc xcp h' pc'' stk'' loc'' xcp''* **by** *auto*
have *no-call2 e2 pc* ⟹ *no-call2* (*e1*;; *e2*) (*Suc* (*length* (*compE2 e1*) + *pc*))
by(*auto simp add*: *no-call2-def*)
hence *?exec ta* (*e1*;; *e2*) *e2' e' h stk loc* (*Suc* (*length* (*compE2 e1*) + *pc*)) *xcp h'* (*Suc* (*length* (*compE2 e1*) + *pc''*)) *stk'' loc'' xcp''*
using *exec'* **by**(*cases* τ*move1 P h e2'*)(*auto*, (*blast intro*: *Seq-*τ*ExecrI2 Seq-*τ*ExectI2 exec-move-SeqI2*)+)
moreover from *bisim'*
have *P*,*e1*;;*e2*,*h'* ⊢ (*e'*, *xs'*) ↔ (*stk''*, *loc''*, *Suc* (*length* (*compE2 e1*) + *pc''*), *xcp''*)
by(*rule bisim1-bisims1.bisim1Seq2*)
ultimately show *?case* **by**(*auto split*: *if-split-asm*) *blast*+
next
case (*bisim1Cond1 E n e xs stk loc pc xcp e1 e2*)
note *IH* = *bisim1Cond1.IH*(*2*)
note *bisim* = ‹*P*,*E*,*h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim1* = ‹⋀*xs*. *P*,*e1*,*h* ⊢ (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim2* = ‹⋀*xs*. *P*,*e2*,*h* ⊢ (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bsok* = ‹*bsok* (*if* (*E*) *e1 else e2*) *n*›
from ‹*True*,*P*,*t* ⊢*1* ⟨*if* (*e*) *e1 else e2*,(*h*, *xs*)⟩ −*ta*→ ⟨*e'*,(*h'*, *xs'*)⟩› **show** *?case*
proof *cases*
case (*Cond1Red b'*)
note [*simp*] = ‹*e'* = *if* (*b'*) *e1 else e2*›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e*,(*h*, *xs*)⟩ −*ta*→ ⟨*b'*,(*h'*, *xs'*)⟩›
from *red* **have** τ*move1 P h* (*if* (*e*) *e1 else e2*) = τ*move1 P h e* **by**(*auto simp add*: τ*move1.simps* τ*moves1.simps*)
moreover have *call1* (*if* (*e*) *e1 else e2*) = *call1 e* **by** *auto*
moreover from *IH*[*OF red*] *bsok*
obtain *pc''* *stk''* *loc''* *xcp''* **where** *bisim*: *P*,*E*,*h'* ⊢ (*b'*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
and *redo*: *?exec ta E e b' h stk loc pc xcp h' pc'' stk'' loc'' xcp''* **by** *auto*
from *bisim*
have *P*,*if* (*E*) *e1 else e2*,*h'* ⊢ (*if* (*b'*) *e1 else e2*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
by(*rule bisim1-bisims1.bisim1Cond1*)
moreover {
assume *no-call2 E pc*
hence *no-call2* (*if* (*E*) *e1 else e2*) *pc* **by**(*auto simp add*: *no-call2-def*) }
ultimately show *?thesis* **using** *redo*
by(*auto simp del*: *call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)(*blast intro*: *Cond-*τ*ExecrI1 Cond-*τ*ExectI1 exec-move-CondI1*)+
next
case *Red1CondT*
note [*simp*] = ‹*e* = *true*› ‹*e'* = *e1*› ‹*ta* = ε› ‹*h'* = *h*› ‹*xs'* = *xs*›
from *bisim* **have** *s*: *xcp* = *None xs* = *loc*
and τ*Exec-mover-a P t E h* (*stk*, *loc*, *pc*, *xcp*) ([*Bool True*], *loc*, *length* (*compE2 E*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence τ*Exec-mover-a P t* (*if* (*E*) *e1 else e2*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*Bool True*], *loc*, *length* (*compE2 E*), *None*)
by−(*rule Cond-*τ*ExecrI1*)
moreover have *exec-move-a P t* (*if* (*E*) *e1 else e2*) *h* ([*Bool True*], *loc*, *length* (*compE2 E*), *None*) ε *h* ([], *loc*, *Suc* (*length* (*compE2 E*)), *None*)
unfolding *exec-move-def* **by**(*rule exec-instr*, *auto*)
moreover have τ*move2* (*compP2 P*) *h* [*Bool True*] (*if* (*E*) *e1 else e2*) (*length* (*compE2 E*)) *None* **by**(*simp add*: τ*move2-iff*)

ultimately have *τExec-movet-a P t (if (E) e1 else e2) h (stk, loc, pc, xcp) ([], loc, Suc (length (compE2 E)), None)*
by(*auto intro*: *rtranclp-into-tranclp1 τexec-moveI simp add*: *compP2-def*)
moreover have *τmove1 P h (if (true) e1 else e2)* **by**(*rule τmove1CondRed*)
moreover
from *bisim1-refl*
have *P, if (E) e1 else e2, h ⊢ (e1, xs) ↔ ([], loc, Suc (length (compE2 E) + 0), None)*
unfolding *s* **by**(*rule bisim1CondThen*)
ultimately show *?thesis* **by** (*fastforce*)
next
case *Red1CondF*
note [*simp*] = ‹*e = false*› ‹*e′ = e2*› ‹*ta = ε*› ‹*h′ = h*› ‹*xs′ = xs*›
from *bisim* **have** *s*: *xcp = None xs = loc*
and *τExec-mover-a P t E h (stk, loc, pc, xcp) ([Bool False], loc, length (compE2 E), None)*
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t (if (E) e1 else e2) h (stk, loc, pc, xcp) ([Bool False], loc, length (compE2 E), None)*
by−(*rule Cond-τExecrI1*)
moreover have *exec-move-a P t (if (E) e1 else e2) h ([Bool False], loc, length (compE2 E), None) ε h ([], loc, Suc (Suc (length (compE2 E) + length (compE2 e1))), None)*
unfolding *exec-move-def* **by**(*rule exec-instr*)(*auto*)
moreover have *τmove2 (compP2 P) h [Bool False] (if (E) e1 else e2) (length (compE2 E)) None* **by**(*rule τmove2CondRed*)
ultimately have *τExec-movet-a P t (if (E) e1 else e2) h (stk, loc, pc, xcp) ([], loc, Suc (Suc (length (compE2 E) + length (compE2 e1))), None)*
by(*auto intro*: *rtranclp-into-tranclp1 τexec-moveI simp add*: *compP2-def*)
moreover have *τmove1 P h (if (false) e1 else e2)* **by**(*rule τmove1CondRed*)
moreover
from *bisim1-refl*
have *P, if (E) e1 else e2, h ⊢ (e2, loc) ↔ ([], loc, (Suc (Suc (length (compE2 E) + length (compE2 e1) + 0))), None)*
unfolding *s* **by**(*rule bisim1CondElse*)
ultimately show *?thesis* **using** *s* **by** *auto*(*blast intro*: *tranclp-into-rtranclp*)
next
case (*Cond1Throw a*)
note [*simp*] = ‹*e = Throw a*› ‹*ta = ε*› ‹*e′ = Throw a*› ‹*h′ = h*› ‹*xs′ = xs*›
have *τ*: *τmove1 P h (if (Throw a) e1 else e2)* **by**(*rule τmove1CondThrow*)
from *bisim* **have** *xcp = ⌊a⌋ ∨ xcp = None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp = ⌊a⌋*
with *bisim*
have *P, if (E) e1 else e2, h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)*
by(*auto intro*: *bisim1-bisims1.bisim1CondThrow*)
thus *?thesis* **using** *τ* **by**(*fastforce*)
next
assume [*simp*]: *xcp = None*
with *bisim* **obtain** *pc′*
where *τExec-mover-a P t E h (stk, loc, pc, None) ([Addr a], loc, pc′, ⌊a⌋)*
and *bisim′*: *P, E, h ⊢ (Throw a, xs) ↔ ([Addr a], loc, pc′, ⌊a⌋)* **and** [*simp*]: *xs = loc*
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t (if (E) e1 else e2) h (stk, loc, pc, None) ([Addr a], loc, pc′, ⌊a⌋)*
by−(*rule Cond-τExecrI1*)
moreover from *bisim′*

```
    have P, if (E) e1 else e2, h ⊢ (Throw a, xs) ↔ ([Addr a], loc, pc', ⌊a⌋)
      by−(rule bisim1CondThrow, auto)
    ultimately show ?thesis using τ by auto
   qed
  qed
next
  case (bisim1CondThen e1 n e1' xs stk loc pc xcp e e2)
  note IH = bisim1CondThen.IH(2)
  note bisim1 = ‹P,e1,h ⊢ (e1', xs) ↔ (stk, loc, pc, xcp)›
  note bisim = ‹⋀xs. P,e,h ⊢ (e, xs) ↔ ([], xs, 0, None)›
  note bisim2 = ‹⋀xs. P,e2,h ⊢ (e2, xs) ↔ ([], xs, 0, None)›
  note bsok = ‹bsok (if (e) e1 else e2) n›
  from IH[OF ‹True,P,t ⊢1 ⟨e1',(h, xs)⟩ −ta→ ⟨e',(h', xs')⟩›] bsok obtain pc'' stk'' loc'' xcp''
    where bisim': P,e1,h' ⊢ (e', xs') ↔ (stk'', loc'', pc'', xcp'')
    and exec': ?exec ta e1 e1' e' h stk loc pc xcp h' pc'' stk'' loc'' xcp'' by auto
  have no-call2 e1 pc ⟹ no-call2 (if (e) e1 else e2) (Suc (length (compE2 e) + pc))
     by(auto simp add: no-call2-def)
    hence ?exec ta (if (e) e1 else e2) e1' e' h stk loc (Suc (length (compE2 e) + pc)) xcp h' (Suc
(length (compE2 e) + pc'')) stk'' loc'' xcp''
   using exec' by(cases τmove1 P h e1')(auto, (blast intro: Cond-τExecrI2 Cond-τExectI2 exec-move-CondI2)+)
  moreover from bisim'
  have P,if (e) e1 else e2,h' ⊢ (e', xs') ↔ (stk'', loc'', Suc (length (compE2 e) + pc''), xcp'')
    by(rule bisim1-bisims1.bisim1CondThen)
  ultimately show ?case
    by(auto split: if-split-asm) blast+
next
  case (bisim1CondElse e2 n e2' xs stk loc pc xcp e e1)
  note IH = bisim1CondElse.IH(2)
  note bisim2 = ‹P,e2,h ⊢ (e2', xs) ↔ (stk, loc, pc, xcp)›
  note bisim = ‹⋀xs. P,e,h ⊢ (e, xs) ↔ ([], xs, 0, None)›
  note bisim1 = ‹⋀xs. P,e1,h ⊢ (e1, xs) ↔ ([], xs, 0, None)›
  from IH[OF ‹True,P,t ⊢1 ⟨e2',(h, xs)⟩ −ta→ ⟨e',(h', xs')⟩›] ‹bsok (if (e) e1 else e2) n›
  obtain pc'' stk'' loc'' xcp''
    where bisim': P,e2,h' ⊢ (e', xs') ↔ (stk'', loc'', pc'', xcp'')
    and exec': ?exec ta e2 e2' e' h stk loc pc xcp h' pc'' stk'' loc'' xcp'' by auto
  have no-call2 e2 pc ⟹ no-call2 (if (e) e1 else e2) (Suc (Suc (length (compE2 e) + length (compE2
e1) + pc)))
     by(auto simp add: no-call2-def)
  hence ?exec ta (if (e) e1 else e2) e2' e' h stk loc (Suc (Suc (length (compE2 e) + length (compE2
e1) + pc))) xcp h' (Suc (Suc (length (compE2 e) + length (compE2 e1) + pc''))) stk'' loc'' xcp''
   using exec' by(cases τmove1 P h e2')(auto, (blast intro: Cond-τExecrI3 Cond-τExectI3 exec-move-CondI3)+)
  moreover from bisim'
  have P,if (e) e1 else e2,h' ⊢ (e', xs') ↔ (stk'', loc'', Suc (Suc (length (compE2 e) + length (compE2
e1) + pc'')), xcp'')
    by(rule bisim1-bisims1.bisim1CondElse)
  ultimately show ?case
    by(auto split: if-split-asm) blast+
next
  case bisim1CondThrow thus ?case by auto
next
  case (bisim1While1 c n e xs)
  note bisim1 = ‹⋀xs. P,c,h ⊢ (c, xs) ↔ ([], xs, 0, None)›
  note bisim2 = ‹⋀xs. P,e,h ⊢ (e, xs) ↔ ([], xs, 0, None)›
  from ‹True,P,t ⊢1 ⟨while (c) e,(h, xs)⟩ −ta→ ⟨e',(h', xs')⟩› show ?case
```

proof *cases*
case *Red1While*
note [*simp*] = ‹*ta* = *ε*› ‹*e′* = *if* (*c*) (*e*;; *while* (*c*) *e*) *else unit*› ‹*h′* = *h*› ‹*xs′* = *xs*›
have *τmove1 P h* (*while* (*c*) *e*) **by**(*rule τmove1WhileRed*)
moreover
have *P*,*while* (*c*) *e*,*h* ⊢ (*if* (*c*) (*e*;; *while* (*c*) *e*) *else unit*, *xs*) ↔ ([], *xs*, *0*, *None*)
by(*rule bisim1-bisims1.bisim1While3*[*OF bisim1-refl*])
moreover have *sim12-size* (*while* (*c*) *e*) > *sim12-size e′* **by**(*simp*)
ultimately show *?thesis* **by** *auto*
qed
next
case (*bisim1While3 c n c′ xs stk loc pc xcp e*)
note *IH* = *bisim1While3.IH*(*2*)
note *bisim1* = ‹*P*,*c*,*h* ⊢ (*c′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim2* = ‹⋀*xs*. *P*,*e*,*h*⊢ (*e*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bsok* = ‹*bsok* (*while* (*c*) *e*) *n*›
from ‹*True*,*P*,*t* ⊢*1* ⟨*if* (*c′*) (*e*;; *while* (*c*) *e*) *else unit*,(*h*, *xs*)⟩ −*ta*→ ⟨*e′*,(*h′*, *xs′*)⟩› **show** *?case*
proof *cases*
case (*Cond1Red b′*)
note [*simp*] = ‹*e′* = *if* (*b′*) (*e*;; *while* (*c*) *e*) *else unit*›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*c′*,(*h*, *xs*)⟩ −*ta*→ ⟨*b′*,(*h′*, *xs′*)⟩›
from *red* **have** *τmove1 P h* (*if* (*c′*) (*e*;; *while* (*c*) *e*) *else unit*) = *τmove1 P h c′* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
moreover from *red* **have** *call1* (*if* (*c′*) (*e*;;*while* (*c*) *e*) *else unit*) = *call1 c′* **by** *auto*
moreover from *IH*[*OF red*] *bsok*
obtain *pc″ stk″ loc″ xcp″* **where** *bisim*: *P*,*c*,*h′* ⊢ (*b′*, *xs′*) ↔ (*stk″*, *loc″*, *pc″*, *xcp″*)
and *redo*: *?exec ta c c′ b′ h stk loc pc xcp h′ pc″ stk″ loc″ xcp″* **by** *auto*
from *bisim*
have *P*,*while* (*c*) *e*,*h′* ⊢ (*if* (*b′*) (*e*;; *while* (*c*) *e*) *else unit*, *xs′*) ↔ (*stk″*, *loc″*, *pc″*, *xcp″*)
by(*rule bisim1-bisims1.bisim1While3*)
moreover {
assume *no-call2 c pc*
hence *no-call2* (*while* (*c*) *e*) *pc* **by**(*auto simp add*: *no-call2-def*) }
ultimately show *?thesis* **using** *redo*
by(*auto simp del*: *call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)(*blast intro*: *While-τExecrI1 While-τExectI1 exec-move-WhileI1*)+
next
case *Red1CondT*
note [*simp*] = ‹*c′* = *true*› ‹*e′* = *e*;; *while* (*c*) *e*› ‹*ta* = *ε*› ‹*h′* = *h*› ‹*xs′* = *xs*›
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t c h* (*stk*, *loc*, *pc*, *xcp*) ([*Bool True*], *loc*, *length* (*compE2 c*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*while* (*c*) *e*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*Bool True*], *loc*, *length* (*compE2 c*), *None*)
by−(*rule While-τExecrI1*)
moreover have *exec-move-a P t* (*while* (*c*) *e*) *h* ([*Bool True*], *loc*, *length* (*compE2 c*), *None*) *ε h* ([], *loc*, *Suc* (*length* (*compE2 c*)), *None*)
unfolding *exec-move-def* **by**(*rule exec-instr*, *auto*)
moreover have *τmove2* (*compP2 P*) *h* [*Bool True*] (*while* (*c*) *e*) (*length* (*compE2 c*)) *None* **by**(*simp add*: *τmove2-iff*)
ultimately have *τExec-movet-a P t* (*while* (*c*) *e*) *h* (*stk*, *loc*, *pc*, *xcp*) ([], *loc*, *Suc* (*length* (*compE2 c*)), *None*)
by(*auto intro*: *rtranclp-into-tranclp1 τexec-moveI simp add*: *compP2-def*)
moreover have *τmove1 P h* (*if* (*c′*) (*e*;; *while* (*c*) *e*) *else unit*) **by**(*auto simp add*: *τmove1.simps*

τmoves1.simps)
moreover from *bisim1-refl*
have *P*, *while* (*c*) *e*, *h* ⊢ (*e*;; *while* (*c*) *e*, *xs*) ↔ ([], *loc*, *Suc* (*length* (*compE2 c*) + *0*), *None*)
unfolding *s* **by**(*rule bisim1While4*)
ultimately show *?thesis* **by** (*fastforce*)
next
case *Red1CondF*
note [*simp*] = ‹*c′* = *false*› ‹*e′* = *unit*› ‹*ta* = *ε*› ‹*h′* = *h*› ‹*xs′* = *xs*›
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t c h* (*stk*, *loc*, *pc*, *xcp*) ([*Bool False*], *loc*, *length* (*compE2 c*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*while* (*c*) *e*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*Bool False*], *loc*, *length* (*compE2 c*), *None*)
by−(*rule While-τExecrI1*)
moreover have *exec-move-a P t* (*while* (*c*) *e*) *h* ([*Bool False*], *loc*, *length* (*compE2 c*), *None*) *ε h* ([], *loc*, *Suc* (*Suc* (*Suc* (*length* (*compE2 c*) + *length* (*compE2 e*)))), *None*)
by(*auto intro*!: *exec-instr simp add*: *exec-move-def*)
moreover have *τmove2* (*compP2 P*) *h* [*Bool False*] (*while* (*c*) *e*) (*length* (*compE2 c*)) *None* **by**(*simp add*: *τmove2-iff*)
ultimately have *τExec-mover-a P t* (*while* (*c*) *e*) *h* (*stk*, *loc*, *pc*, *xcp*) ([], *loc*, *Suc* (*Suc* (*Suc* (*length* (*compE2 c*) + *length* (*compE2 e*)))), *None*)
by(*auto intro*: *rtranclp.rtrancl-into-rtrancl τexec-moveI simp add*: *compP2-def*)
moreover have *τmove1 P h* (*if* (*false*) (*e*;;*while* (*c*) *e*) *else unit*) **by**(*rule τmove1CondRed*)
moreover have *P*, *while* (*c*) *e*, *h* ⊢ (*unit*, *xs*) ↔ ([], *loc*, (*Suc* (*Suc* (*Suc* (*length* (*compE2 c*) + *length* (*compE2 e*))))), *None*)
unfolding *s* **by**(*rule bisim1While7*)
ultimately show *?thesis* **using** *s* **by** *auto*
next
case (*Cond1Throw a*)
note [*simp*] = ‹*c′* = *Throw a*› ‹*ta* = *ε*› ‹*e′* = *Throw a*› ‹*h′* = *h*› ‹*xs′* = *xs*›
have *τ*: *τmove1 P h* (*if* (*c′*) (*e*;; *while* (*c*) *e*) *else unit*) **by**(*auto intro*: *τmove1CondThrow*)
from *bisim1* **have** *xcp* = ⌊*a*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*a*⌋
with *bisim1*
have *P*, *while* (*c*) *e*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
by(*auto intro*: *bisim1-bisims1.bisim1WhileThrow1*)
thus *?thesis* **using** *τ* **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim1* **obtain** *pc′*
where *τExec-mover-a P t c h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
and *bisim′*: *P*, *c*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t* (*while* (*c*) *e*) *h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
by−(*rule While-τExecrI1*)
moreover from *bisim′*
have *P*, *while* (*c*) *e*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
by−(*rule bisim1WhileThrow1*, *auto*)
ultimately show *?thesis* **using** *τ* **by** *auto*
qed
qed
next

case (*bisim1While4 E n e xs stk loc pc xcp c*)
note *IH* = *bisim1While4.IH*(*2*)
note *bisim2* = ‹*P*,*E*,*h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim1* = ‹⋀*xs*. *P*,*c*,*h* ⊢ (*c*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bsok* = ‹*bsok* (*while* (*c*) *E*) *n*›
from ‹*True*,*P*,*t* ⊢*1* ⟨*e*;; *while* (*c*) *E*,(*h*, *xs*)⟩ −*ta*→ ⟨*e'*,(*h'*, *xs'*)⟩› **show** *?case*
proof *cases*
case (*Seq1Red E'*)
note [*simp*] = ‹*e'* = *E'*;;*while* (*c*) *E*›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e*, (*h*, *xs*)⟩ −*ta*→ ⟨*E'*, (*h'*, *xs'*)⟩›
from *red* **have** *τ*: *τmove1 P h* (*e*;; *while* (*c*) *E*) = *τmove1 P h e* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
with *IH*[*OF red*] *bsok* **obtain** *pc''* *stk''* *loc''* *xcp''*
where *bisim*: *P*,*E*,*h'* ⊢ (*E'*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
and *exec'*: *?exec ta E e E' h stk loc pc xcp h' pc'' stk'' loc'' xcp''* **by** *auto*
have *?exec ta* (*while* (*c*) *E*) (*e*;;*while* (*c*) *E*) (*E'*;;*while* (*c*) *E*) *h stk loc* (*Suc* (*length* (*compE2 c*) + *pc*)) *xcp h'* (*Suc* (*length* (*compE2 c*) + *pc''*)) *stk'' loc'' xcp''*
proof(*cases τmove1 P h* (*e*;; *while* (*c*) *E*))
case *True*
with *exec'* **show** *?thesis* **using** *τ* **by**(*fastforce intro*: *While-τExecrI2 While-τExectI2*)
next
case *False*
with *exec' τ* **obtain** *pc'* *stk'* *loc'* *xcp'*
where *e*: *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
and *e'*: *exec-move-a P t E h* (*stk'*, *loc'*, *pc'*, *xcp'*) (*extTA2JVM* (*compP2 P*) *ta*) *h'* (*stk''*, *loc''*, *pc''*, *xcp''*)
and *τ'*: ¬ *τmove2* (*compP2 P*) *h stk' E pc' xcp'*
and *call*: (*call1 e* = *None* ∨ *no-call2 E pc* ∨ *pc'* = *pc* ∧ *stk'* = *stk* ∧ *loc'* = *loc* ∧ *xcp'* = *xcp*) **by** *auto*
from *e* **have** *τExec-mover-a P t* (*while* (*c*) *E*) *h* (*stk*, *loc*, *Suc* (*length* (*compE2 c*) + *pc*), *xcp*) (*stk'*, *loc'*, *Suc* (*length* (*compE2 c*) + *pc'*), *xcp'*) **by**(*rule While-τExecrI2*)
moreover
from *e'* **have** *exec-move-a P t* (*while* (*c*) *E*) *h* (*stk'*, *loc'*, *Suc* (*length* (*compE2 c*) + *pc'*), *xcp'*) (*extTA2JVM* (*compP2 P*) *ta*) *h'* (*stk''*, *loc''*, *Suc* (*length* (*compE2 c*) + *pc''*), *xcp''*)
by(*rule exec-move-WhileI2*)
moreover from *τ' e'* **have** ¬ *τmove2* (*compP2 P*) *h stk'* (*while* (*c*) *E*) (*Suc* (*length* (*compE2 c*) + *pc'*)) *xcp'*
by(*auto simp add*: *τmove2-iff*)
moreover have *call1* (*e*;; *while* (*c*) *E*) = *call1 e* **by** *simp*
moreover have *no-call2 E pc* ⟹ *no-call2* (*while* (*c*) *E*) (*Suc* (*length* (*compE2 c*) + *pc*))
by(*auto simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *False call* **by**(*auto simp del*: *split-paired-Ex call1.simps calls1.simps*)
qed
with *bisim τ* **show** *?thesis* **by** *auto* (*blast intro*: *bisim1-bisims1.bisim1While4*)+
next
case (*Red1Seq v*)
note [*simp*] = ‹*e* = *Val v*› ‹*ta* = *ε*› ‹*e'* = *while* (*c*) *E*› ‹*h'* = *h*› ‹*xs'* = *xs*›
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 E*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*while* (*c*) *E*) *h* (*stk*, *loc*, *Suc* (*length* (*compE2 c*) + *pc*), *xcp*) ([*v*], *loc*, *Suc* (*length* (*compE2 c*) + *length* (*compE2 E*)), *None*)
by−(*rule While-τExecrI2*)
moreover

have *exec-move-a P t* (*while* (*c*) *E*) *h* ([*v*], *loc*, *Suc* (*length* (*compE2 c*) + *length* (*compE2 E*)), *None*) *ε h* ([], *loc*, *Suc* (*Suc* (*length* (*compE2 c*) + *length* (*compE2 E*))), *None*)
unfolding *exec-move-def* **by**(*rule exec-instr*, *auto*)
moreover have *τmove2* (*compP2 P*) *h* [*v*] (*while* (*c*) *E*) (*Suc* (*length* (*compE2 c*) + *length* (*compE2 E*))) *None* **by**(*simp add*: *τmove2-iff*)
ultimately have *τExec-movet-a P t* (*while* (*c*) *E*) *h* (*stk*, *loc*, *Suc* (*length* (*compE2 c*) + *pc*), *xcp*) ([], *loc*, *Suc* (*Suc* (*length* (*compE2 c*) + *length* (*compE2 E*))), *None*)
by(*auto intro*: *rtranclp-into-tranclp1 τexec-moveI simp add*: *compP2-def*)
moreover
have *P*, *while* (*c*) *E*, *h* ⊢ (*while* (*c*) *E*, *xs*) ↔ ([], *xs*, (*Suc* (*Suc* (*length* (*compE2 c*) + *length* (*compE2 E*)))), *None*)
unfolding *s* **by**(*rule bisim1While6*)
moreover have *τmove1 P h* (*e*;; *while* (*c*) *E*) **by**(*auto intro*: *τmove1SeqRed*)
ultimately show *?thesis* **using** *s* **by**(*auto*)(*blast intro*: *tranclp-into-rtranclp*)
next
case (*Seq1Throw a*)
note [*simp*] = ‹*e* = *Throw a*› ‹*ta* = *ε*› ‹*e′* = *Throw a*› ‹*h′* = *h*› ‹*xs′* = *xs*›
have *τ*: *τmove1 P h* (*e*;; *while* (*c*) *E*) **by**(*auto intro*: *τmove1SeqThrow*)
from *bisim2* **have** *xcp* = ⌊*a*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*a*⌋
with *bisim2*
have *P*, *while* (*c*) *E*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *Suc* (*length* (*compE2 c*) + *pc*), *xcp*)
by(*auto intro*: *bisim1WhileThrow2*)
thus *?thesis* **using** *τ* **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim2* **obtain** *pc′*
where *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
and *bisim′*: *P*, *E*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t* (*while* (*c*) *E*) *h* (*stk*, *loc*, *Suc* (*length* (*compE2 c*) + *pc*), *None*) ([*Addr a*], *loc*, *Suc* (*length* (*compE2 c*) + *pc′*), ⌊*a*⌋)
by−(*rule While-τExecrI2*)
moreover from *bisim′*
have *P*, *while* (*c*) *E*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *Suc* (*length* (*compE2 c*) + *pc′*), ⌊*a*⌋)
by−(*rule bisim1WhileThrow2*, *auto*)
ultimately show *?thesis* **using** *τ* **by** *auto*
qed
qed
next
case (*bisim1While6 c n e xs*)
note *bisim1* = ‹⋀*xs*. *P*,*c*,*h* ⊢ (*c*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim2* = ‹⋀*xs*. *P*,*e*,*h* ⊢ (*e*, *xs*) ↔ ([], *xs*, *0*, *None*)›
from ‹*True*,*P*,*t* ⊢1 ⟨*while* (*c*) *e*,(*h*, *xs*)⟩ −*ta*→ ⟨*e′*,(*h′*, *xs′*)⟩› **show** *?case*
proof *cases*
case *Red1While*
note [*simp*] = ‹*ta* = *ε*› ‹*e′* = *if* (*c*) (*e*;; *while* (*c*) *e*) *else unit*› ‹*h′* = *h*› ‹*xs′* = *xs*›
have *τmove1 P h* (*while* (*c*) *e*) **by**(*rule τmove1WhileRed*)
moreover
have *P*,*while* (*c*) *e*,*h* ⊢ (*if* (*c*) (*e*;; *while* (*c*) *e*) *else unit*, *xs*) ↔ ([], *xs*, *0*, *None*)
by(*rule bisim1-bisims1.bisim1While3*[*OF bisim1-refl*])
moreover have *τExec-movet-a P t* (*while* (*c*) *e*) *h* ([], *xs*, *Suc* (*Suc* (*length* (*compE2 c*) + *length*

(*compE2 e*))), *None*) ([], *xs*, *0*, *None*)
by(*rule τExect1step*)(*auto simp add*: *exec-move-def τmove2-iff intro*: *exec-instr*)
ultimately show *?thesis* **by**(*fastforce*)
qed
next
case *bisim1While7* **thus** *?case* **by** *fastforce*
next
case *bisim1WhileThrow1* **thus** *?case* **by** *auto*
next
case *bisim1WhileThrow2* **thus** *?case* **by** *auto*
next
case (*bisim1Throw1 E n e xs stk loc pc xcp*)
note *IH* = *bisim1Throw1.IH*(*2*)
note *bisim* = ‹*P*,*E*,*h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*throw e*,(*h*, *xs*)⟩ −*ta*→ ⟨*e'*,(*h'*, *xs'*)⟩›
note *bsok* = ‹*bsok* (*throw E*) *n*›
from *red* **show** *?case*
proof *cases*
case (*Throw1Red E'*)
note [*simp*] = ‹*e'* = *throw E'*›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e*, (*h*, *xs*)⟩ −*ta*→ ⟨*E'*, (*h'*, *xs'*)⟩›
from *red* **have** *τmove1 P h* (*throw e*) = *τmove1 P h e* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
moreover have *call1* (*throw e*) = *call1 e* **by** *auto*
moreover from *IH*[*OF red*] *bsok*
obtain *pc''* *stk''* *loc''* *xcp''* **where** *bisim*: *P*,*E*,*h'* ⊢ (*E'*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
and *redo*: *?exec ta E e E' h stk loc pc xcp h' pc'' stk'' loc'' xcp''* **by** *auto*
from *bisim*
have *P*,*throw E*,*h'* ⊢ (*throw E'*, *xs'*) ↔ (*stk''*, *loc''*, *pc''*, *xcp''*)
by(*rule bisim1-bisims1.bisim1Throw1*)
moreover {
assume *no-call2 E pc*
hence *no-call2* (*throw E*) *pc* **by**(*auto simp add*: *no-call2-def*) **}**
ultimately show *?thesis* **using** *redo*
by(*auto simp del*: *call1.simps calls1.simps split*: *if-split-asm split del*: *if-split*)(*blast intro*: *Throw-τExecrI Throw-τExectI exec-move-ThrowI*)+
next
case *Red1ThrowNull*
note [*simp*] = ‹*e* = *null*› ‹*ta* = *ε*› ‹*e'* = *THROW NullPointer*› ‹*h'* = *h*› ‹*xs'* = *xs*›
from *bisim* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *xcp*) ([*Null*], *loc*, *length* (*compE2 E*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*throw E*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*Null*], *loc*, *length* (*compE2 E*), *None*)
by−(*rule Throw-τExecrI*)
also have *τExec-movet-a P t* (*throw E*) *h* ([*Null*], *loc*, *length* (*compE2 E*), *None*) ([*Null*], *loc*, *length* (*compE2 E*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule τExect1step*)(*auto intro*: *exec-instr τmove2-τmoves2.intros simp add*: *exec-move-def*)
also have *P*, *throw E*, *h* ⊢ (*THROW NullPointer*, *xs*) ↔ ([*Null*], *loc*, *length* (*compE2 E*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
unfolding *s* **by**(*rule bisim1ThrowNull*)
moreover have *τmove1 P h* (*throw e*) **by**(*auto intro*: *τmove1ThrowNull*)
ultimately show *?thesis* **by** *auto*
next
case (*Throw1Throw a*)

```
    note [simp] = ‹e = Throw a› ‹ta = ε› ‹e' = Throw a› ‹h' = h› ‹xs' = xs›
    have τ: τmove1 P h (throw (Throw a)) by(rule τmove1ThrowThrow)
    from bisim have xcp = ⌊a⌋ ∨ xcp = None by(auto dest: bisim1-ThrowD)
    thus ?thesis
    proof
      assume xcp = ⌊a⌋
      with bisim show ?thesis using τ by(fastforce intro: bisim1ThrowThrow)
    next
      assume [simp]: xcp = None
      from bisim obtain pc'
        where τExec-mover-a P t E h (stk, loc, pc, None) ([Addr a], loc, pc', ⌊a⌋)
        and bisim: P, E, h ⊢ (Throw a, xs) ↔ ([Addr a], loc, pc', ⌊a⌋) and s: xs = loc
        by(auto dest: bisim1-Throw-τExec-mover)
      hence τExec-mover-a P t (throw E) h (stk, loc, pc, None) ([Addr a], loc, pc', ⌊a⌋)
        by −(rule Throw-τExecrI)
      moreover from bisim have P, throw E, h ⊢ (Throw a, xs) ↔ ([Addr a], loc, pc', ⌊a⌋)
        by(rule bisim1ThrowThrow)
      ultimately show ?thesis using τ by auto
    qed
  qed
next
  case bisim1Throw2 thus ?case by auto
next
  case bisim1ThrowNull thus ?case by auto
next
  case bisim1ThrowThrow thus ?case by auto
next
  case (bisim1Try E n e xs stk loc pc xcp e2 C' V)
  note IH = bisim1Try.IH(2)
  note bisim1 = ‹P,E,h ⊢ (e, xs) ↔ (stk, loc, pc, xcp)›
  note bisim2 = ‹⋀xs. P,e2,h ⊢ (e2, xs) ↔ ([], xs, 0, None)›
  note red = ‹True,P,t ⊢1 ⟨try e catch(C' V) e2,(h, xs)⟩ −ta→ ⟨e',(h', xs')⟩›
  note bsok = ‹bsok (try E catch(C' V) e2) n›
  from red show ?case
  proof cases
    case (Try1Red E')
    note [simp] = ‹e' = try E' catch(C' V) e2›
      and red = ‹True,P,t ⊢1 ⟨e, (h, xs)⟩ −ta→ ⟨E', (h', xs')⟩›
   from red have τmove1 P h (try e catch(C' V) e2) = τmove1 P h e by(auto simp add: τmove1.simps
τmoves1.simps)
    moreover have call1 (try e catch(C' V) e2) = call1 e by auto
    moreover from IH[OF red] bsok
    obtain pc'' stk'' loc'' xcp'' where bisim: P,E,h' ⊢ (E', xs') ↔ (stk'', loc'', pc'', xcp'')
      and redo: ?exec ta E e E' h stk loc pc xcp h' pc'' stk'' loc'' xcp'' by auto
    from bisim
    have P,try E catch(C' V) e2,h' ⊢ (try E' catch(C' V) e2, xs') ↔ (stk'', loc'', pc'', xcp'')
      by(rule bisim1-bisims1.bisim1Try)
    moreover {
      assume no-call2 E pc
      hence no-call2 (try E catch(C' V) e2) pc by(auto simp add: no-call2-def) }
    ultimately show ?thesis using redo
        by(auto simp del: call1.simps calls1.simps split: if-split-asm split del: if-split)(blast intro:
Try-τExecrI1 Try-τExectI1 exec-move-TryI1)+
  next
```

case (*Red1Try v*)
note [*simp*] = ‹*e* = *Val v*› ‹*ta* = *ε*› ‹*e'* = *Val v*› ‹*h'* = *h*› ‹*xs'* = *xs*›
have *τ*: *τmove1 P h* (*try Val v catch*(*C' V*) *e2*) **by**(*rule τmove1TryRed*)
from *bisim1* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 E*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*try E catch*(*C' V*) *e2*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*v*], *loc*, *length* (*compE2 E*), *None*)
by−(*rule Try-τExecrI1*)
also have *τExec-mover-a P t* (*try E catch*(*C' V*) *e2*) *h* ([*v*], *loc*, *length* (*compE2 E*), *None*) ([*v*], *loc*, *length* (*compE2* (*try E catch*(*C' V*) *e2*)), *None*)
by(*rule τExecr1step*)(*auto intro*: *exec-instr simp add*: *exec-move-def τmove2-iff*)
also (*rtranclp-trans*)
have *P*, *try E catch*(*C' V*) *e2*, *h* ⊢ (*Val v*, *xs*) ↔ ([*v*], *xs*, *length* (*compE2* (*try E catch*(*C' V*) *e2*)), *None*)
by(*rule bisim1Val2*) *simp*
ultimately show *?thesis* **using** *s τ* **by**(*auto*)
next
case (*Red1TryCatch a D*)
hence [*simp*]: *e* = *Throw a ta* = *ε e'* = {*V*:*Class C'*=*None*; *e2*} *h'* = *h xs'* = *xs*[*V* := *Addr a*]
and *ha*: *typeof-addr h a* = ⌊*Class-type D*⌋ **and** *sub*: *P* ⊢ *D* ≼* *C'*
and *V*: *V* < *length xs* **by** *auto*
from *bisim1* **have** [*simp*]: *xs* = *loc* **and** *xcp*: *xcp* = ⌊*a*⌋ ∨ *xcp* = *None*
by(*auto dest*: *bisim1-ThrowD*)
from *xcp* **have** *τExec-mover-a P t* (*try E catch*(*C' V*) *e2*) *h* (*stk*, *loc*, *pc*, *xcp*) ([*Addr a*], *loc*, *Suc* (*length* (*compE2 E*)), *None*)
proof
assume [*simp*]: *xcp* = ⌊*a*⌋
with *bisim1* **have** *match-ex-table* (*compP2 P*) (*cname-of h a*) *pc* (*compxE2 E 0 0*) = *None*
by(*auto dest*: *bisim1-xcp-Some-not-caught*[**where** *pc'*=*0*] *simp add*: *compP2-def*)
moreover from *bisim1* **have** *pc* < *length* (*compE2 E*)
by(*auto dest*: *bisim1-ThrowD*)
ultimately show *?thesis* **using** *ha sub* **unfolding** ‹*xcp* = ⌊*a*⌋›
by−(*rule τExecr1step*[*unfolded exec-move-def*, *OF exec-catch*[**where** *d*=*0*, *simplified*]],
auto simp add: *τmove2-iff matches-ex-entry-def compP2-def match-ex-table-append-not-pcs cname-of-def*)
next
assume [*simp*]: *xcp* = *None*
with *bisim1* **obtain** *pc'* **where** *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc'*, ⌊*a*⌋)
and *bisim'*: *P*, *E*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc'*, ⌊*a*⌋) **and** *s*: *xs* = *loc*
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t* (*try E catch*(*C' V*) *e2*) *h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc'*, ⌊*a*⌋)
by−(*rule Try-τExecrI1*)
also from *bisim'* **have** *match-ex-table* (*compP2 P*) (*cname-of h a*) *pc'* (*compxE2 E 0 0*) = *None*
by(*auto dest*: *bisim1-xcp-Some-not-caught*[**where** *pc'*=*0*] *simp add*: *compP2-def*)
with *ha sub bisim1-ThrowD*[*OF bisim'*]
have *τExec-mover-a P t* (*try E catch*(*C' V*) *e2*) *h* ([*Addr a*], *loc*, *pc'*, ⌊*a*⌋) ([*Addr a*], *loc*, *Suc* (*length* (*compE2 E*)), *None*)
by−(*rule τExecr1step*[*unfolded exec-move-def*, *OF exec-catch*[**where** *d*=*0*, *simplified*]], *auto simp add*: *τmove2-iff matches-ex-entry-def compP2-def match-ex-table-append-not-pcs cname-of-def*)
finally (*rtranclp-trans*) **show** *?thesis* **by** *simp*
qed
also let *?pc'* = *Suc* (*length* (*compE2 E*)) **from** *V*

have *exec*: *τExec-movet-a P t* (*try E catch*(*C′ V*) *e2*) *h* ([*Addr a*], *loc*, *?pc′*, *None*) ([], *loc*[*V* := *Addr a*], *Suc ?pc′*, *None*)
by−(*rule τExect1step*[*unfolded exec-move-def*, *OF exec-instr*], *auto simp add*: *nth-append intro*: *τmove2-τmoves2.intros*)
also (*rtranclp-tranclp-tranclp*)
have *bisim′*: *P*,*try E catch*(*C′ V*) *e2*, *h* ⊢ ({*V*:*Class C′*=*None*; *e2*}, *xs*[*V* := *Addr a*]) ↔ ([], *loc*[*V* := *Addr a*], *Suc ?pc′*, *None*)
unfolding ‹*xs* = *loc*› **by**(*rule bisim1TryCatch2*[*OF bisim1-refl*, *simplified*])
moreover have *τmove1 P h* (*try Throw a catch*(*C′ V*) *e2*) **by**(*rule τmove1TryThrow*)
ultimately show *?thesis* **by**(*auto*)(*blast intro*: *tranclp-into-rtranclp*)
next
case (*Red1TryFail a D*)
hence [*simp*]: *e* = *Throw a ta* = *ε e′* = *Throw a h′* = *h xs′* = *xs*
and *ha*: *typeof-addr h a* = ⌊*Class-type D*⌋ **and** *sub*: ¬ *P* ⊢ *D* ⪯* *C′* **by** *auto*
have *τ*: *τmove1 P h* (*try Throw a catch*(*C′ V*) *e2*) **by**(*rule τmove1TryThrow*)
from *bisim1* **have** [*simp*]: *xs* = *loc* **and** *xcp* = ⌊*a*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim1* **have** *pc*: *pc* ≤ *length* (*compE2 E*) **by**(*rule bisim1-pc-length-compE2*)
from ‹*xcp* = ⌊*a*⌋ ∨ *xcp* = *None*› **show** *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*a*⌋
with *bisim1 ha sub*
have *P*,*try E catch*(*C′ V*) *e2*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
by(*auto intro*: *bisim1TryFail*)
thus *?thesis* **using** *τ* **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim1* **obtain** *pc′*
where *τExec-mover-a P t E h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
and *bisim′*: *P*, *E*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t* (*try E catch*(*C′ V*) *e2*) *h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
by−(*rule Try-τExecrI1*)
moreover from *bisim′ ha sub*
have *P*,*try E catch*(*C′ V*) *e2*,*h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
by(*auto intro*: *bisim1TryFail*)
ultimately show *?thesis* **using** *τ* **by** *auto*
qed
qed
next
case (*bisim1TryCatch1 e n a xs stk loc pc D C′ e2 V*)
note *bisim1* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
note *bisim2* = ‹⋀*xs*. *P*,*e2*,*h* ⊢ (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *IH2* = *bisim1TryCatch1.IH*(*6*)
note *ha* = ‹*typeof-addr h a* = ⌊*Class-type D*⌋›
note *sub* = ‹*P* ⊢ *D* ⪯* *C′*›
note *red* = ‹*True*,*P*,*t* ⊢*1* ⟨{*V*:*Class C′*=*None*; *e2*},(*h*, *xs*[*V* := *Addr a*])⟩ −*ta*→ ⟨*e′*,(*h′*, *xs′*)⟩›
note *bsok* = ‹*bsok* (*try e catch*(*C′ V*) *e2*) *n*›
from *bisim1* **have** [*simp*]: *xs* = *loc* **by**(*auto dest*: *bisim1-ThrowD*)
from *red* **show** *?case*
proof *cases*
case (*Block1Red E′*)
note [*simp*] = ‹*e′* = {*V*:*Class C′*=*None*; *E′*}›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e2*, (*h*, *xs*[*V* := *Addr a*])⟩ −*ta*→ ⟨*E′*, (*h′*, *xs′*)⟩›
from *red* **have** *τ*: *τmove1 P h* {*V*:*Class C′*=*None*; *e2*} = *τmove1 P h e2* **by**(*auto simp add*:

τmove1.simps τmoves1.simps)
have *exec*: *τExec-mover-a P t* (*try e catch*(*C′ V*) *e2*) *h* ([*Addr a*], *xs*, *Suc* (*length* (*compE2 e*) + *0*), *None*) ([], *xs*[*V* := *Addr a*], *Suc* (*Suc* (*length* (*compE2 e*) + *0*)), *None*)
by −(*rule τExecr1step*, *auto simp add*: *exec-move-def τmove2-iff intro*: *exec-instr*)
moreover from *IH2*[*OF red*] *bsok* **obtain** *pc″ stk″ loc″ xcp″*
where *bisim′*: *P*,*e2*,*h′* ⊢ (*E′*, *xs′*) ↔ (*stk″*, *loc″*, *pc″*, *xcp″*)
and *exec′*: *?exec ta e2 e2 E′ h* [] (*xs*[*V* := *Addr a*]) *0 None h′ pc″ stk″ loc″ xcp″* **by** *auto*
have *?exec ta* (*try e catch*(*C′ V*) *e2*) {*V*:*Class C′*=*None*; *e2*} {*V*:*Class C′*=*None*; *E′*} *h* [] (*xs*[*V* := *Addr a*]) (*Suc* (*Suc* (*length* (*compE2 e*)))) *None h′* (*Suc* (*Suc* (*length* (*compE2 e*) + *pc″*))) *stk″ loc″ xcp″*
proof(*cases τmove1 P h* {*V*:*Class C′*=*None*; *e2*})
case *True* **with** *τ exec′* **show** *?thesis*
by(*fastforce dest*: *Try-τExecrI2 Try-τExectI2 simp del*: *compE2.simps compEs2.simps*)
next
case *False*
with *τ exec′* **obtain** *pc′ stk′ loc′ xcp′*
where *e*: *τExec-mover-a P t e2 h* ([], *xs*[*V* := *Addr a*], *0*, *None*) (*stk′*, *loc′*, *pc′*, *xcp′*)
and *e′*: *exec-move-a P t e2 h* (*stk′*, *loc′*, *pc′*, *xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk″*, *loc″*, *pc″*, *xcp″*)
and *τ′*: ¬ *τmove2* (*compP2 P*) *h stk′ e2 pc′ xcp′*
and *call*: *call1 e2* = *None* ∨ *no-call2 e2 0* ∨ *pc′* = *0* ∧ *stk′* = [] ∧ *loc′* = *xs*[*V* := *Addr a*] ∧ *xcp′* = *None* **by** *auto*
from *e* **have** *τExec-mover-a P t* (*try e catch*(*C′ V*) *e2*) *h* ([], *xs*[*V* := *Addr a*], *Suc* (*Suc* (*length* (*compE2 e*) + *0*)), *None*) (*stk′*, *loc′*, *Suc* (*Suc* (*length* (*compE2 e*) + *pc′*)), *xcp′*)
by(*rule Try-τExecrI2*)
moreover from *e′*
have *exec-move-a P t* (*try e catch*(*C′ V*) *e2*) *h* (*stk′*, *loc′*, *Suc* (*Suc* (*length* (*compE2 e*) + *pc′*)), *xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk″*, *loc″*, *Suc* (*Suc* (*length* (*compE2 e*) + *pc″*)), *xcp″*)
by(*rule exec-move-TryI2*)
moreover from *τ′* **have** *τmove2* (*compP2 P*) *h stk′* (*try e catch*(*C′ V*) *e2*) (*Suc* (*Suc* (*length* (*compE2 e*) + *pc′*))) *xcp′* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover have *call1* {*V*:*Class C′*=*None*; *e2*} = *call1 e2* **by** *simp*
moreover have *no-call2 e2 0* ⟹ *no-call2* (*try e catch*(*C′ V*) *e2*) (*Suc* (*Suc* (*length* (*compE2 e*))))
by(*auto simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *False call* **by**(*auto simp del*: *split-paired-Ex call1.simps calls1.simps*) *blast*
qed
moreover from *bisim′*
have *P*, *try e catch*(*C′ V*) *e2*, *h′* ⊢ ({*V*:*Class C′*=*None*; *E′*}, *xs′*) ↔ (*stk″*, *loc″*, *Suc* (*Suc* (*length* (*compE2 e*) + *pc″*)), *xcp″*)
by(*rule bisim1TryCatch2*)
moreover have *no-call2* (*try e catch*(*C′ V*) *e2*) (*Suc* (*length* (*compE2 e*))) **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *τ*
by *auto*(*blast intro*: *rtranclp-trans rtranclp-tranclp-tranclp*)+
next
case (*Red1Block u*)
note [*simp*] = ‹*e2* = *Val u*› ‹*ta* = *ε*› ‹*e′* = *Val u*› ‹*h′* = *h*› ‹*xs′* = *xs*[*V* := *Addr a*]›
have *τExec-mover-a P t* (*try e catch*(*C′ V*) *Val u*) *h* ([*Addr a*], *xs*, *Suc* (*length* (*compE2 e*) + *0*), *None*) ([], *xs*[*V* := *Addr a*], *Suc* (*Suc* (*length* (*compE2 e*) + *0*)), *None*)
by −(*rule τExecr1step*, *auto simp add*: *exec-move-def τmove2-iff intro*: *exec-instr*)
also have *τExec-mover-a P t* (*try e catch*(*C′ V*) *Val u*) *h* ([], *xs*[*V* := *Addr a*], *Suc* (*Suc* (*length*

(*compE2 e*) + *0*)), *None*) ([*u*], *xs*[*V* := *Addr a*], *Suc* (*Suc* (*length* (*compE2 e*) + *1*)), *None*)
by −(*rule Try-τExecrI2*[*OF τExecr1step*[*unfolded exec-move-def*, *OF exec-instr*]], *auto simp add*: *τmove2-iff*)
also (*rtranclp-trans*)
have *P*, *try e catch*(*C′ V*) *Val u*, *h* ⊢ (*Val u*, *xs*[*V* := *Addr a*]) ↔ ([*u*], *xs*[*V* := *Addr a*], *length* (*compE2* (*try e catch*(*C′ V*) *Val u*)), *None*)
by(*rule bisim1Val2*) *simp*
moreover have *τmove1 P h* {*V*:*Class C′*=*None*; *Val u*} **by**(*rule τmove1BlockRed*)
ultimately show *?thesis* **by**(*auto*)
next
case (*Block1Throw a′*)
note [*simp*] = ‹*e2* = *Throw a′*› ‹*h′* = *h*› ‹*ta* = *ε*› ‹*e′* = *Throw a′*› ‹*xs′* = *xs*[*V* := *Addr a*]›
have *τmove1 P h* {*V*:*Class C′*=*None*; *Throw a′*} **by**(*rule τmove1BlockThrow*)
moreover have *τExec-mover-a P t* (*try e catch*(*C′ V*) *e2*) *h* ([*Addr a*], *loc*, *Suc* (*length* (*compE2 e*)), *None*)
([*Addr a′*], *xs′*, *Suc* (*Suc* (*Suc* (*length* (*compE2 e*)))), ⌊*a′*⌋)
by(*rule τExecr3step*)(*auto simp add*: *exec-move-def exec-meth-instr τmove2-iff*)
moreover have *P*, *try e catch*(*C′ V*) *Throw a′*, *h* ⊢ (*Throw a′*, *xs′*) ↔ ([*Addr a′*], *xs′*, *Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2* (*addr a′*)))), ⌊*a′*⌋)
by(*rule bisim1TryCatchThrow*)(*rule bisim1Throw2*)
ultimately show *?thesis* **by** *auto*
qed
next
case (*bisim1TryCatch2 e2 n e2′ xs stk loc pc xcp e C′ V*)
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e2′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim1* = ‹⋀*xs*. *P*,*e*,*h* ⊢ (*e*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *IH2* = *bisim1TryCatch2.IH*(*2*)
note *red* = ‹*True*,*P*,*t* ⊢*1* ⟨{*V*:*Class C′*=*None*; *e2′*},(*h*, *xs*)⟩ −*ta*→ ⟨*e′*,(*h′*, *xs′*)⟩›
note *bsok* = ‹*bsok* (*try e catch*(*C′ V*) *e2*) *n*›
from *red* **show** *?case*
proof *cases*
case (*Block1Red E′*)
note [*simp*] = ‹*e′* = {*V*:*Class C′*=*None*; *E′*}›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e2′*, (*h*, *xs*)⟩ −*ta*→ ⟨*E′*, (*h′*, *xs′*)⟩›
from *red* **have** *τ*: *τmove1 P h* {*V*:*Class C′*=*None*; *e2′*} = *τmove1 P h e2′* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *IH2*[*OF red*] *bsok* **obtain** *pc″ stk″ loc″ xcp″*
where *bisim′*: *P*,*e2*,*h′* ⊢ (*E′*, *xs′*) ↔ (*stk″*, *loc″*, *pc″*, *xcp″*)
and *exec′*: *?exec ta e2 e2′ E′ h stk loc pc xcp h′ pc″ stk″ loc″ xcp″* **by** *auto*
have *?exec ta* (*try e catch*(*C′ V*) *e2*) {*V*:*Class C′*=*None*; *e2′*} {*V*:*Class C′*=*None*; *E′*} *h stk loc* (*Suc* (*Suc* (*length* (*compE2 e*) + *pc*))) *xcp h′* (*Suc* (*Suc* (*length* (*compE2 e*) + *pc″*))) *stk″ loc″ xcp″*
proof (*cases τmove1 P h* {*V*:*Class C′*=*None*; *e2′*})
case *True* **with** *τ exec′* **show** *?thesis* **by**(*auto intro*: *Try-τExecrI2 Try-τExectI2*)
next
case *False*
with *τ exec′* **obtain** *pc′ stk′ loc′ xcp′*
where *e*: *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*)
and *e′*: *exec-move-a P t e2 h* (*stk′*, *loc′*, *pc′*, *xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk″*, *loc″*, *pc″*, *xcp″*)
and *τ′*: ¬ *τmove2* (*compP2 P*) *h stk′ e2 pc′ xcp′*
and *call*: *call1 e2′* = *None* ∨ *no-call2 e2 pc* ∨ *pc′* = *pc* ∧ *stk′* = *stk* ∧ *loc′* = *loc* ∧ *xcp′* = *xcp*
by *auto*
from *e* **have** *τExec-mover-a P t* (*try e catch*(*C′ V*) *e2*) *h* (*stk*, *loc*, *Suc* (*Suc* (*length* (*compE2 e*) + *pc*)), *xcp*) (*stk′*, *loc′*, *Suc* (*Suc* (*length* (*compE2 e*) + *pc′*)), *xcp′*)

by(*rule Try-τExecrI2*)
moreover from *e′*
have *exec-move-a P t* (*try e catch*(*C′ V*) *e2*) *h* (*stk′*, *loc′*, *Suc* (*Suc* (*length* (*compE2 e*) + *pc′*)), *xcp′*) (*extTA2JVM* (*compP2 P*) *ta*) *h′* (*stk″*, *loc″*, *Suc* (*Suc* (*length* (*compE2 e*) + *pc″*)), *xcp″*)
by(*rule exec-move-TryI2*)
moreover from *τ′* **have** *τmove2* (*compP2 P*) *h stk′* (*try e catch*(*C′ V*) *e2*) (*Suc* (*Suc* (*length* (*compE2 e*) + *pc′*))) *xcp′* ⟹ *False*
by(*simp add*: *τmove2-iff*)
moreover have *call1* {*V*:*Class C′*=*None*; *e2′*} = *call1 e2′* **by** *simp*
moreover have *no-call2 e2 pc* ⟹ *no-call2* (*try e catch*(*C′ V*) *e2*) (*Suc* (*Suc* (*length* (*compE2 e*) + *pc*)))
by(*auto simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *False call* **by**(*auto simp del*: *split-paired-Ex call1.simps calls1.simps*)
qed
moreover from *bisim′*
have *P*, *try e catch*(*C′ V*) *e2*, *h′* ⊢ ({*V*:*Class C′*=*None*; *E′*}, *xs′*) ↔ (*stk″*, *loc″*, *Suc* (*Suc* (*length* (*compE2 e*) + *pc″*)), *xcp″*)
by(*rule bisim1-bisims1.bisim1TryCatch2*)
ultimately show *?thesis* **using** *τ* **by** *auto blast*+
next
case (*Red1Block u*)
note [*simp*] = ‹*e2′* = *Val u*› ‹*ta* = *ε*› ‹*e′* = *Val u*› ‹*h′* = *h*› ‹*xs′* = *xs*›
from *bisim2* **have** *s*: *xcp* = *None xs* = *loc*
and *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *xcp*) ([*u*], *loc*, *length* (*compE2 e2*), *None*)
by(*auto dest*: *bisim1Val2D1*)
hence *τExec-mover-a P t* (*try e catch*(*C′ V*) *e2*) *h* (*stk*, *loc*, *Suc* (*Suc* (*length* (*compE2 e*) + *pc*)), *xcp*) ([*u*], *loc*, *Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e2*))), *None*)
by −(*rule Try-τExecrI2*)
moreover
have *P*, *try e catch*(*C′ V*) *e2*, *h* ⊢ (*Val u*, *xs*) ↔ ([*u*], *xs*, *length* (*compE2* (*try e catch*(*C′ V*) *e2*)), *None*)
by(*rule bisim1Val2*) *simp*
moreover have *τmove1 P h* {*V*:*Class C′*=*None*; *Val u*} **by**(*rule τmove1BlockRed*)
ultimately show *?thesis* **using** *s* **by** *auto*
next
case (*Block1Throw a*)
note [*simp*] = ‹*e2′* = *Throw a*› ‹*ta* = *ε*› ‹*e′* = *Throw a*› ‹*h′* = *h*› ‹*xs′* = *xs*›
have *τ*: *τmove1 P h* {*V*:*Class C′*=*None*; *e2′*} **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
from *bisim2* **have** *xcp* = ⌊*a*⌋ ∨ *xcp* = *None* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis*
proof
assume [*simp*]: *xcp* = ⌊*a*⌋
with *bisim2*
have *P*, *try e catch*(*C′ V*) *e2*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *Suc* (*Suc* (*length* (*compE2 e*) + *pc*)), *xcp*)
by(*auto intro*: *bisim1TryCatchThrow*)
thus *?thesis* **using** *τ* **by**(*fastforce*)
next
assume [*simp*]: *xcp* = *None*
with *bisim2* **obtain** *pc′*
where *τExec-mover-a P t e2 h* (*stk*, *loc*, *pc*, *None*) ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋)
and *bisim′*: *P*, *e2*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *loc*, *pc′*, ⌊*a*⌋) **and** [*simp*]: *xs* = *loc*
by(*auto dest*: *bisim1-Throw-τExec-mover*)
hence *τExec-mover-a P t* (*try e catch*(*C′ V*) *e2*) *h* (*stk*, *loc*, *Suc* (*Suc* (*length* (*compE2 e*) +

```
pc)), None) ([Addr a], loc, Suc (Suc (length (compE2 e) + pc')), ⌊a⌋)
      by−(rule Try-τExecrI2)
    moreover from bisim'
    have P, try e catch(C' V) e2, h ⊢ (Throw a, xs) ↔ ([Addr a], loc, Suc (Suc (length (compE2 e)
+ pc')), ⌊a⌋)
      by(rule bisim1TryCatchThrow)
    ultimately show ?thesis using τ by auto
  qed
 qed
next
 case bisim1TryFail thus ?case by auto
next
 case bisim1TryCatchThrow thus ?case by auto
next
 case bisims1Nil thus ?case by(auto elim!: reds1.cases)
next
 case (bisims1List1 E n e xs stk loc pc xcp es)
 note IH1 = bisims1List1.IH(2)
 note IH2 = bisims1List1.IH(4)
 note bisim1 = ‹P,E,h ⊢ (e, xs) ↔ (stk, loc, pc, xcp)›
 note bisim2 = ‹⋀xs. P,es,h ⊢ (es, xs) [↔] ([], xs, 0, None)›
 note bsok = ‹bsoks (E # es) n›
 from ‹True,P,t ⊢1 ⟨e # es,(h, xs)⟩ [−ta→] ⟨es',(h', xs')⟩› show ?case
 proof cases
  case (List1Red1 E')
  note [simp] = ‹es' = E' # es›
   and red = ‹True,P,t ⊢1 ⟨e,(h, xs)⟩ −ta→ ⟨E',(h', xs')⟩›
  from red have τ: τmoves1 P h (e # es) = τmove1 P h e by(auto simp add: τmove1.simps
τmoves1.simps)
  moreover from IH1[OF red] bsok
  obtain pc'' stk'' loc'' xcp'' where bisim: P,E,h' ⊢ (E', xs') ↔ (stk'', loc'', pc'', xcp'')
   and redo: ?exec ta E e E' h stk loc pc xcp h' pc'' stk'' loc'' xcp'' by auto
  from bisim
  have P,E#es,h' ⊢ (E'#es, xs') [↔] (stk'', loc'', pc'', xcp'')
   by(rule bisim1-bisims1.bisims1List1)
  moreover {
   assume no-call2 E pc
    hence no-calls2 (E # es) pc ∨ pc = length (compE2 E) by(auto simp add: no-call2-def
no-calls2-def) }
  moreover from red have calls1 (e # es) = call1 e by auto
  ultimately show ?thesis using redo
    apply(auto simp add: exec-move-def exec-moves-def simp del: call1.simps calls1.simps split:
if-split-asm split del: if-split)
  apply(blast intro: τExec-mover-τExec-movesr τExec-movet-τExec-movest intro!: bisim1-bisims1.bisims1List1
elim: τmoves2.cases)+
   done
 next
  case (List1Red2 ES' v)
  note [simp] = ‹es' = Val v # ES'› ‹e = Val v›
   and red = ‹True,P,t ⊢1 ⟨es,(h, xs)⟩ [−ta→] ⟨ES',(h', xs')⟩›
  from bisim1 have s: xs = loc xcp = None
   and exec1: τExec-mover-a P t E h (stk, loc, pc, xcp) ([v], loc, length (compE2 E), None)
   by(auto dest: bisim1Val2D1)
  hence τExec-movesr-a P t (E # es) h (stk, loc, pc, xcp) ([v], loc, length (compE2 E), None)
```

```
      by −(rule τExec-mover-τExec-movesr)
    moreover from IH2[OF red] bsok obtain pc'' stk'' loc'' xcp''
      where bisim': P,es,h' ⊢ (ES', xs') [↔] (stk'', loc'', pc'', xcp'')
      and exec': ?execs ta es es ES' h [] xs 0 None h' pc'' stk'' loc'' xcp'' by auto
   have τ: τmoves1 P h (Val v # es) = τmoves1 P h es by(auto simp add: τmove1.simps τmoves1.simps)
    have ?execs ta (E # es) (Val v # es) (Val v # ES') h [v] xs (length (compE2 E)) None h' (length
(compE2 E) +  pc'') (stk'' @ [v]) loc'' xcp''
    proof(cases τmoves1 P h (Val v # es))
      case True with τ exec' show ?thesis
        using append-τExec-movesr[of [v] [E] - P t es h [] xs 0 None stk'' loc'' pc'' xcp'']
          append-τExec-movest[of [v] [E] - P t es h [] xs 0 None stk'' loc'' pc'' xcp''] by auto
    next
      case False with τ exec' obtain pc' stk' loc' xcp'
        where e: τExec-movesr-a P t es h ([], xs, 0, None) (stk', loc', pc', xcp')
       and e': exec-moves-a P t es h (stk', loc', pc', xcp') (extTA2JVM (compP2 P) ta) h' (stk'', loc'',
pc'', xcp'')
        and τ': ¬ τmoves2 (compP2 P) h stk' es pc' xcp'
        and call: calls1 es = None ∨ no-calls2 es 0 ∨ pc' = 0 ∧ stk' = [] ∧ loc' = xs ∧ xcp' = None
by auto
      from append-τExec-movesr[OF - e, where vs=[v] and es' = [E]]
      have τExec-movesr-a P t (E # es) h ([v], xs, length (compE2 E), None) (stk' @ [v], loc', length
(compE2 E) + pc', xcp') by simp
      moreover from append-exec-moves[OF - e', of [v] [E]]
     have exec-moves-a P t (E # es) h (stk' @ [v], loc', length (compE2 E) + pc', xcp') (extTA2JVM
(compP2 P) ta) h' (stk'' @ [v], loc'', length (compE2 E) + pc'', xcp'')
        by simp
      moreover from τ' e'
      have τmoves2 (compP2 P) h (stk' @ [v]) (E # es) (length (compE2 E) + pc') xcp' ⟹ False
        by(auto simp add: τmoves2-iff τinstr-stk-drop-exec-moves)
      moreover have calls1 (Val v # es) = calls1 es by simp
      moreover have no-calls2 es 0 ⟹ no-calls2 (E # es) (length (compE2 E))
        by(auto simp add: no-calls2-def)
    ultimately show ?thesis using False call by(auto simp del: split-paired-Ex call1.simps calls1.simps)
blast
    qed
    moreover from bisim'
    have P,E # es,h' ⊢ (Val v # ES', xs') [↔] (stk'' @ [v], loc'', length (compE2 E) + pc'', xcp'')
      by(rule bisim1-bisims1.bisims1List2)
    moreover from bisim1 have pc ≠ length (compE2 E) ⟶ no-calls2 (E # es) pc
      by(auto simp add: no-calls2-def dest: bisim-Val-pc-not-Invoke bisim1-pc-length-compE2)
    ultimately show ?thesis using τ exec1 s
      apply(auto simp del: split-paired-Ex call1.simps calls1.simps split: if-split-asm split del: if-split)
     apply(blast intro: τExec-movesr-trans|fastforce elim!: τExec-movesr-trans simp del: split-paired-Ex
call1.simps calls1.simps)+
      done
  qed
next
  case (bisims1List2 ES n es xs stk loc pc xcp e v)
  note IH2 = bisims1List2.IH(2)
  note bisim1 = ‹⋀xs. P,e,h ⊢ (e, xs) ↔ ([], xs, 0, None)›
  note bisim2 = ‹P,ES,h ⊢ (es, xs) [↔] (stk, loc, pc, xcp)›
  note bsok = ‹bsoks (e # ES) n›
  from ‹True,P,t ⊢1 ⟨Val v # es,(h, xs)⟩ [−ta→] ⟨es',(h', xs')⟩› show ?case
  proof cases
```

case (*List1Red2 ES'*)
note [*simp*] = ‹*es'* = *Val v # ES'*›
and *red* = ‹*True*,*P*,*t* ⊢1 ⟨*es*,(*h*, *xs*)⟩ [−*ta*→] ⟨*ES'*,(*h'*, *xs'*)⟩›
from *IH2*[*OF red*] *bsok* **obtain** *pc''* *stk''* *loc''* *xcp''*
where *bisim'*: *P*,*ES*,*h'* ⊢ (*ES'*, *xs'*) [↔] (*stk''*, *loc''*, *pc''*, *xcp''*)
and *exec'*: *?execs ta ES es ES' h stk loc pc xcp h' pc'' stk'' loc'' xcp''* **by** *auto*
have *τ*: *τmoves1 P h* (*Val v # es*) = *τmoves1 P h es* **by**(*auto simp add*: *τmove1.simps τmoves1.simps*)
have *?execs ta* (*e # ES*) (*Val v # es*) (*Val v # ES'*) *h* (*stk* @ [*v*]) *loc* (*length* (*compE2 e*) + *pc*) *xcp h'* (*length* (*compE2 e*) + *pc''*) (*stk''* @ [*v*]) *loc'' xcp''*
proof(*cases τmoves1 P h* (*Val v # es*))
case *True* **with** *τ exec'* **show** *?thesis*
using *append-τExec-movesr*[*of* [*v*] [*e*] - *P t ES h stk*]
append-τExec-movest[*of* [*v*] [*e*] - *P t ES h stk*] **by** *auto*
next
case *False* **with** *τ exec'* **obtain** *pc' stk' loc' xcp'*
where *e*: *τExec-movesr-a P t ES h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
and *e'*: *exec-moves-a P t ES h* (*stk'*, *loc'*, *pc'*, *xcp'*) (*extTA2JVM* (*compP2 P*) *ta*) *h'* (*stk''*, *loc''*, *pc''*, *xcp''*)
and *τ'*: ¬ *τmoves2* (*compP2 P*) *h stk' ES pc' xcp'*
and *call*: *calls1 es* = *None* ∨ *no-calls2 ES pc* ∨ *pc'* = *pc* ∧ *stk'* = *stk* ∧ *loc'* = *loc* ∧ *xcp'* = *xcp* **by** *auto*
from *append-τExec-movesr*[*OF* - *e*, **where** *vs*=[*v*] **and** *es'* = [*e*]]
have *τExec-movesr-a P t* (*e # ES*) *h* (*stk* @ [*v*], *loc*, *length* (*compE2 e*) + *pc*, *xcp*) (*stk'* @ [*v*], *loc'*, *length* (*compE2 e*) + *pc'*, *xcp'*) **by** *simp*
moreover from *append-exec-moves*[*OF* - *e'*, *of* [*v*] [*e*]]
have *exec-moves-a P t* (*e # ES*) *h* (*stk'* @ [*v*], *loc'*, *length* (*compE2 e*) + *pc'*, *xcp'*) (*extTA2JVM* (*compP2 P*) *ta*) *h'* (*stk''* @ [*v*], *loc''*, *length* (*compE2 e*) + *pc''*, *xcp''*) **by** *simp*
moreover from *τ' e'*
have *τmoves2* (*compP2 P*) *h* (*stk'* @ [*v*]) (*e # ES*) (*length* (*compE2 e*) + *pc'*) *xcp'* ⟹ *False*
by(*auto simp add*: *τmoves2-iff τinstr-stk-drop-exec-moves*)
moreover have *calls1* (*Val v # es*) = *calls1 es* **by** *simp*
moreover have *no-calls2 ES pc* ⟹ *no-calls2* (*e # ES*) (*length* (*compE2 e*) + *pc*)
by(*auto simp add*: *no-calls2-def*)
ultimately show *?thesis* **using** *False call* **by**(*auto simp del*: *split-paired-Ex call1.simps calls1.simps*)

qed
moreover from *bisim'*
have *P*,*e # ES*,*h'* ⊢ (*Val v # ES'*, *xs'*) [↔] (*stk''* @ [*v*], *loc''*, *length* (*compE2 e*) + *pc''*, *xcp''*)
by(*rule bisim1-bisims1.bisims1List2*)
ultimately show *?thesis* **using** *τ* **by** *auto blast*+
qed *auto*
qed

end

context *J1-JVM-conf-read* **begin**

lemma *exec-1-simulates-Red1-τ*:
assumes *wf*: *wf-J1-prog P*
and *Red1*: *True*,*P*,*t* ⊢1 ⟨(*e*, *xs*)/*exs*, *h*⟩ −*ta*→ ⟨(*e'*, *xs'*)/*exs'*, *h*⟩
and *bisim*: *bisim1-list1 t h* (*e*, *xs*) *exs xcp frs*
and *τ*: *τMove1 P h* ((*e*, *xs*), *exs*)
shows ∃ *xcp' frs'*. (*if sim12-size e'* < *sim12-size e then τExec-1-dr else τExec-1-dt*) (*compP2 P*) *t*

(*xcp*, *h*, *frs*) (*xcp'*, *h*, *frs'*) ∧ *bisim1-list1 t h* (*e'*,*xs'*) *exs' xcp' frs'*
proof −
from *wf* **have** *wt*: *wf-jvm-prog*$_{compTP\ P}$ (*compP2 P*) **by**(*rule wt-compTP-compP2*)
from *Red1* **show** *?thesis*
proof(*cases*)
case (*red1Red TA*)
note [*simp*] = ‹*ta* = *extTA2J1 P TA*› ‹*exs'* = *exs*›
and *red* = ‹*True*,*P*,*t* ⊢*1* ⟨*e*,(*h*, *xs*)⟩ −*TA*→ ⟨*e'*,(*h*, *xs'*)⟩›
from τ *red* **have** τ': τ*move1 P h e* **by**(*auto elim*: *red1-cases*)
from *bisim* **show** *?thesis*
proof(*cases*)
case (*bl1-Normal stk loc C M pc FRS Ts T body D*)
hence [*simp*]: *frs* = (*stk*, *loc*, *C*, *M*, *pc*) # *FRS*
and *conf*: *compTP P* ⊢ *t*: (*xcp*, *h*, *frs*) √
and *sees*: *P* ⊢ *C sees M*: *Ts*→*T* = ⌊*body*⌋ *in D*
and *bisim*: *P*,*blocks1 0* (*Class D*#*Ts*) *body*,*h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
and *bisims*: *list-all2* (*bisim1-fr P h*) *exs FRS*
and *lenxs*: *max-vars e* ≤ *length xs*
by *auto*
from *sees wf* **have** *bsok* (*blocks1 0* (*Class D* # *Ts*) *body*) *0*
by(*auto dest*!: *sees-wf-mdecl WT1-expr-locks simp add*: *wf-J1-mdecl-def wf-mdecl-def bsok-def*)
from *exec-instr-simulates-red1*[*OF wf bisim red this*] τ' **obtain** *pc' stk' loc' xcp'*
where *exec*: (*if sim12-size e'* < *sim12-size e then* τ*Exec-mover-a else* τ*Exec-movet-a*) *P t body h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
and *b'*: *P*,*blocks1 0* (*Class D*#*Ts*) *body*,*h* ⊢ (*e'*, *xs'*) ↔ (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*auto split*: *if-split-asm simp del*: *blocks1.simps*)
from *exec sees* **have** (*if sim12-size e'* < *sim12-size e then* τ*Exec-1r else* τ*Exec-1t*) (*compP2 P*) *t* (*xcp*, *h*, *frs*) (*xcp'*, *h*, (*stk'*, *loc'*, *C*, *M*, *pc'*) # *FRS*)
by(*auto intro*: τ*Exec-mover-*τ*Exec-1r* τ*Exec-movet-*τ*Exec-1t*)
from *wt this conf* **have** *execd*: (*if sim12-size e'* < *sim12-size e then* τ*Exec-1-dr else* τ*Exec-1-dt*) (*compP2 P*) *t* (*xcp*, *h*, *frs*) (*xcp'*, *h*, (*stk'*, *loc'*, *C*, *M*, *pc'*) # *FRS*)
by(*auto intro*: τ*Exec-1r-*τ*Exec-1-dr* τ*Exec-1t-*τ*Exec-1-dt*)
moreover from *wt execd conf*
have *compTP P* ⊢ *t*: (*xcp'*, *h*, (*stk'*, *loc'*, *C*, *M*, *pc'*) # *FRS*) √
by(*auto intro*: τ*Exec-1-dr-preserves-correct-state* τ*Exec-1-dt-preserves-correct-state split*: *if-split-asm*)
hence *bisim1-list1 t h* (*e'*, *xs'*) *exs xcp'* ((*stk'*, *loc'*, *C*, *M*, *pc'*) # *FRS*)
using *sees b'*
proof
from *red* **have** *max-vars e'* ≤ *max-vars e* **by**(*rule red1-max-vars*)
with *red1-preserves-len*[*OF red*] *lenxs*
show *max-vars e'* ≤ *length xs'* **by** *simp*
qed *fact*
hence *bisim1-list1 t h* (*e'*,*xs'*) *exs' xcp'* ((*stk'*, *loc'*, *C*, *M*, *pc'*) # *FRS*) **by** *simp*
ultimately show *?thesis* **by** *blast*
qed(*insert red*, *auto elim*: *red1-cases*)
next
case (*red1Call a' M' vs' U' Ts' T' body' D'*)
hence [*simp*]: *ta* = ε
and *exs'* [*simp*]: *exs'* = (*e*, *xs*) # *exs*
and *e'*: *e'* = *blocks1 0* (*Class D'*#*Ts'*) *body'*
and *xs'*: *xs'* = *Addr a'* # *vs'* @ *replicate* (*max-vars body'*) *undefined-value*
and *ha'*: *typeof-addr h a'* = ⌊*U'*⌋
and *call*: *call1 e* = ⌊(*a'*, *M'*, *vs'*)⌋ **by** *auto*
note *sees'* = ‹*P* ⊢ *class-type-of U' sees M'*: *Ts'*→*T'* = ⌊*body'*⌋ *in D'*›

note *lenvs'Ts'* = ‹*length vs'* = *length Ts'*›
from *ha' sees-method-decl-above*[*OF sees'*]
have *conf*: *P*,*h* ⊢ *Addr a'* :≤ *ty-of-htype U'* **by**(*auto simp add*: *conf-def*)
note *wt* = *wt-compTP-compP2*[*OF wf*]
from *bisim* **show** *?thesis*
proof(*cases*)
case (*bl1-Normal stk loc C M pc FRS Ts T body D*)
hence [*simp*]: *frs* = (*stk*, *loc*, *C*, *M*, *pc*) # *FRS*
and *conf*: *compTP P* ⊢ *t*: (*xcp*, *h*, *frs*) √
and *sees*: *P* ⊢ *C sees M*: *Ts*→*T* = ⌊*body*⌋ *in D*
and *bisim*: *P*,*blocks1 0* (*Class D*#*Ts*) *body*,*h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
and *bisims*: *list-all2* (*bisim1-fr P h*) *exs FRS*
and *lenxs*: *max-vars e* ≤ *length xs* **by** *auto*
from *call bisim* **have** [*simp*]: *xcp* = *None* **by**(*cases xcp*, *auto dest*: *bisim1-call-xcpNone*)
from *bisim* **have** *b*: *P*,*blocks1 0* (*Class D*#*Ts*) *body*,*h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*) **by** *simp*
from *bisim* **have** *lenloc*: *length xs* = *length loc* **by**(*rule bisim1-length-xs*)
from *sees* **have** *sees''*: *compP2 P* ⊢ *C sees M*:*Ts*→*T* = ⌊(*max-stack body*, *max-vars body*, *compE2 body* @ [*Return*], *compxE2 body 0 0*)⌋ *in D*
unfolding *compP2-def compMb2-def Let-def* **by**(*auto dest*: *sees-method-compP*)
from *sees wf* **have** ¬ *contains-insync* (*blocks1 0* (*Class D* # *Ts*) *body*)
by(*auto dest*!: *sees-wf-mdecl WT1-expr-locks simp add*: *wf-J1-mdecl-def wf-mdecl-def contains-insync-conv*)
with *bisim1-call-τExec-move*[*OF b call*, *of 0 t*] *lenxs* **obtain** *pc' loc' stk'*
where *exec*: *τExec-mover-a P t body h* (*stk*, *loc*, *pc*, *None*) (*rev vs'* @ *Addr a'* # *stk'*, *loc'*, *pc'*, *None*)
and *pc'*: *pc'* < *length* (*compE2 body*) **and** *ins*: *compE2 body* ! *pc'* = *Invoke M'* (*length vs'*)
and *bisim'*: *P*,*blocks1 0* (*Class D*#*Ts*) *body*,*h* ⊢ (*e*, *xs*) ↔ (*rev vs'* @ *Addr a'* # *stk'*, *loc'*, *pc'*, *None*)
by(*auto simp add*: *blocks1-max-vars simp del*: *blocks1.simps*)
let *?f* = (*rev vs'* @ *Addr a'* # *stk'*, *loc'*, *C*, *M*, *pc'*)
from *exec sees*
have *exec1*: *τExec-1r* (*compP2 P*) *t* (*None*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *FRS*) (*None*, *h*, *?f* # *FRS*)
by(*rule τExec-mover-τExec-1r*)
with *wt* **have** *τExec-1-dr* (*compP2 P*) *t* (*None*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *FRS*) (*None*, *h*, *?f* # *FRS*) **using** *conf*
by(*simp*)(*rule τExec-1r-τExec-1-dr*)
also with *wt* **have** *conf'*: *compTP P* ⊢ *t*: (*None*, *h*, *?f* # *FRS*) √ **using** *conf*
by *simp* (*rule τExec-1-dr-preserves-correct-state*)
let *?f'* = ([], *Addr a'* # *vs'* @ (*replicate* (*max-vars body'*) *undefined-value*), *D'*, *M'*, *0*)
from *pc' ins sees sees' ha'*
have (*ε*, *None*, *h*, *?f'* # *?f* # *FRS*) ∈ *exec-instr* (*instrs-of* (*compP2 P*) *C M* ! *pc'*) (*compP2 P*) *t h* (*rev vs'* @ *Addr a'* # *stk'*) *loc' C M pc' FRS*
by(*auto simp add*: *compP2-def compMb2-def nth-append split-beta*)
hence *exec-1* (*compP2 P*) *t* (*None*, *h*, *?f* # *FRS*) *ε* (*None*, *h*, *?f'* # *?f* # *FRS*)
using *exec sees* **by**(*simp add*: *exec-1-iff*)
with *conf'* **have** *execd*: *compP2 P*,*t* ⊢ *Normal* (*None*, *h*, *?f* # *FRS*) −*ε*−*jvmd*→ *Normal* (*None*, *h*, *?f'* # *?f* # *FRS*)
by(*simp add*: *welltyped-commute*[*OF wt*])
hence *check*: *check* (*compP2 P*) (*None*, *h*, *?f* # *FRS*) **by**(*rule jvmd-NormalE*)
have *τmove2* (*compP2 P*) *h* (*rev vs'* @ *Addr a'* # *stk'*) *body pc' None* **using** *pc' ins ha' sees'*
by(*auto simp add*: *τmove2-iff compP2-def dest*: *sees-method-fun*)
with *sees pc' ins* **have** *τMove2* (*compP2 P*) (*None*, *h*, (*rev vs'* @ *Addr a'* # *stk'*, *loc'*, *C*, *M*, *pc'*) # *FRS*)

unfolding *τMove2-compP2*[*OF sees*] **by**(*auto simp add*: *compP2-def compMb2-def*)
with ‹*exec-1* (*compP2 P*) *t* (*None*, *h*, *?f* # *FRS*) *ε* (*None*, *h*, *?f′* # *?f* # *FRS*)› *check*
have *τExec-1-dt* (*compP2 P*) *t* (*None*, *h*, *?f* # *FRS*) (*None*, *h*, *?f′* # *?f* # *FRS*) **by** *fastforce*
also from *execd sees″ sees′ ins ha′ pc′* **have** *compP2 P*,*h* ⊢ *vs′* [:≤] *Ts′*
by(*auto simp add*: *check-def compP2-def split*: *if-split-asm elim*!: *jvmd-NormalE*)
hence *lenvs*: *length vs′* = *length Ts′* **by**(*rule list-all2-lengthD*)
from *wt execd conf′* **have** *compTP P* ⊢ *t*:(*None*, *h*, *?f′* # *?f* # *FRS*) √
by(*rule BV-correct-d-1*)
hence *bisim1-list1 t h* (*blocks1 0* (*Class D′*#*Ts′*) *body′*, *xs′*) ((*e*, *xs*) # *exs*) *None* (*?f′* # *?f* # *FRS*)
proof
from *sees′* **show** *P* ⊢ *D′ sees M′*: *Ts′*→*T′* = ⌊*body′*⌋ *in D′* **by**(*rule sees-method-idemp*)
show *P*,*blocks1 0* (*Class D′*#*Ts′*) *body′*,*h* ⊢ (*blocks1 0* (*Class D′*#*Ts′*) *body′*, *xs′*) ↔
([], *Addr a′* # *vs′* @ *replicate* (*max-vars body′*) *undefined-value*, *0*, *None*)
unfolding *xs′* **by**(*rule bisim1-refl*)
show *max-vars* (*blocks1 0* (*Class D′* # *Ts′*) *body′*) ≤ *length xs′*
unfolding *xs′* **using** *lenvs* **by**(*simp add*: *blocks1-max-vars*)
from *lenxs* **have** (*max-vars e*) ≤ *length xs* **by** *simp*
with *sees bisim′ call* **have** *bisim1-fr P h* (*e*, *xs*) (*rev vs′* @ *Addr a′* # *stk′*, *loc′*, *C*, *M*, *pc′*)
by(*rule bisim1-fr.intros*)
thus *list-all2* (*bisim1-fr P h*) ((*e*, *xs*) # *exs*)
((*rev vs′* @ *Addr a′* # *stk′*, *loc′*, *C*, *M*, *pc′*) # *FRS*)
using *bisims* **by** *simp*
qed
moreover have *ta-bisim wbisim1 ta ε* **by** *simp*
ultimately show *?thesis*
unfolding ‹*frs* = (*stk*, *loc*, *C*, *M*, *pc*) # *FRS*› ‹*xcp* = *None*› *e′ exs′*
by *auto*(*blast intro*: *tranclp-into-rtranclp*)
next
case *bl1-finalVal*
with *call* **show** *?thesis* **by** *simp*
next
case *bl1-finalThrow*
with *call* **show** *?thesis* **by** *simp*
qed
next
case (*red1Return E*)
note [*simp*] = ‹*exs* = (*E*, *xs′*) # *exs′*› ‹*ta* = *ε*› ‹*e′* = *inline-call e E*›
note *wt* = *wt-compTP-compP2*[*OF wf*]
from *bisim* **have** *bisim*: *bisim1-list1 t h* (*e*, *xs*) ((*E*, *xs′*) # *exs′*) *xcp frs* **by** *simp*
thus *?thesis*
proof *cases*
case (*bl1-Normal stk loc C M pc FRS Ts T body D*)
hence [*simp*]: *frs* = (*stk*, *loc*, *C*, *M*, *pc*) # *FRS*
and *conf*: *compTP P* ⊢ *t*: (*xcp*, *h*, *frs*) √
and *sees*: *P* ⊢ *C sees M*: *Ts*→*T* = ⌊*body*⌋ *in D*
and *bisim*: *P*,*blocks1 0* (*Class D*#*Ts*) *body*,*h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
and *bisims*: *list-all2* (*bisim1-fr P h*) ((*E*, *xs′*) # *exs′*) *FRS*
and *lenxs*: *max-vars e* ≤ *length xs* **by** *auto*
from *bisims* **obtain** *f FRS′* **where** [*simp*]: *FRS* = *f* # *FRS′* **by**(*fastforce simp add*: *list-all2-Cons1*)
from *bisims* **have** *bisim1-fr P h* (*E*, *xs′*) *f* **by** *simp*
then obtain *C0 M0 Ts0 T0 body0 D0 stk0 loc0 pc0 a′ M′ vs′*
where [*simp*]: *f* = (*stk0*, *loc0*, *C0*, *M0*, *pc0*)
and *sees0*: *P* ⊢ *C0 sees M0*:*Ts0*→*T0*=⌊*body0*⌋ *in D0*

and *bisim0*: *P*,*blocks1 0* (*Class D0*#*Ts0*) *body0*,*h* ⊢ (*E*, *xs′*) ↔ (*stk0*, *loc0*, *pc0*, *None*)
and *lenxs0*: *max-vars E* ≤ *length xs′*
and *call0*: *call1 E* = ⌊(*a′*, *M′*, *vs′*)⌋
by *cases auto*

let *?ee* = *inline-call e E*

from *bisim0 call0* **have** *pc0*: *pc0* < *length* (*compE2* (*blocks1 0* (*Class D0*#*Ts0*) *body0*))
by(*rule bisim1-call-pcD*)
hence *pc0*: *pc0* < *length* (*compE2 body0*) **by** *simp*
with *sees-method-compP*[*OF sees0*, **where** *f*=λ*C M Ts T*. *compMb2*]
sees-method-compP[*OF sees*, **where** *f*=λ*C M Ts T*. *compMb2*] *conf*
obtain *ST LT* **where** Φ: *compTP P C0 M0* ! *pc0* = ⌊(*ST*, *LT*)⌋
and *conff*: *conf-f* (*compP* (λ*C M Ts T*. *compMb2*) *P*) *h* (*ST*, *LT*) (*compE2 body0* @ [*Return*])
(*stk0*, *loc0*, *C0*, *M0*, *pc0*)
and *ins*: (*compE2 body0* @ [*Return*]) ! *pc0* = *Invoke M* (*length Ts*)
by(*simp add*: *correct-state-def*)(*fastforce simp add*: *compP2-def compMb2-def dest*: *sees-method-fun*)
from *bisim1-callD*[*OF bisim0 call0*, *of M length Ts*] *ins pc0*
have [*simp*]: *M′* = *M* **by** *simp*

from ‹*final e*› **show** *?thesis*
proof(*cases*)
fix *v*
assume [*simp*]: *e* = *Val v*
with *bisim* **have** [*simp*]: *xcp* = *None* **by**(*auto dest*: *bisim-Val-loc-eq-xcp-None*)

from *bisim1Val2D1*[*OF bisim*[*unfolded* ‹*xcp* = *None*› ‹*e* = *Val v*›]]
have τ*Exec-mover-a P t body h* (*stk*, *loc*, *pc*, *None*) ([*v*], *loc*, *length* (*compE2 body*), *None*)
and [*simp*]: *xs* = *loc* **by**(*auto simp del*: *blocks1.simps*)
with *sees* **have** τ*Exec-1r* (*compP2 P*) *t* (*None*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *FRS*) (*None*, *h*, ([*v*],
loc, *C*, *M*, *length* (*compE2 body*)) # *FRS*)
by−(*rule* τ*Exec-mover-*τ*Exec-1r*)
with *conf wt* **have** τ*Exec-1-dr* (*compP2 P*) *t* (*None*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *FRS*) (*None*, *h*,
([*v*], *loc*, *C*, *M*, *length* (*compE2 body*)) # *FRS*)
by(*simp*)(*rule* τ*Exec-1r-*τ*Exec-1-dr*)
moreover with *conf wt* **have** *conf′*: *compTP P* ⊢ *t*:(*None*, *h*, ([*v*], *loc*, *C*, *M*, *length* (*compE2*
body)) # *FRS*) √
by(*simp*)(*rule* τ*Exec-1-dr-preserves-correct-state*)
from *sees sees0*
have *exec*: *exec-1* (*compP2 P*) *t* (*None*, *h*, ([*v*], *loc*, *C*, *M*, *length* (*compE2 body*)) # *FRS*) ε
(*None*, *h*, (*v* # *drop* (*Suc* (*length Ts*)) *stk0*, *loc0*, *C0*, *M0*, *Suc pc0*) # *FRS′*)
by(*simp add*: *exec-1-iff compP2-def compMb2-def*)
moreover with *conf′ wt* **have** *compP2 P*,*t* ⊢ *Normal* (*None*, *h*, ([*v*], *loc*, *C*, *M*, *length* (*compE2*
body)) # *FRS*) −ε−*jvmd*→ *Normal* (*None*, *h*, (*v* # *drop* (*Suc* (*length Ts*)) *stk0*, *loc0*, *C0*, *M0*, *Suc*
pc0) # *FRS′*)
by(*simp add*: *welltyped-commute*)
hence *check* (*compP2 P*) (*None*, *h*, ([*v*], *loc*, *C*, *M*, *length* (*compE2 body*)) # *FRS*)
by(*rule jvmd-NormalE*)
moreover have τ*Move2* (*compP2 P*) (*None*, *h*, ([*v*], *loc*, *C*, *M*, *length* (*compE2 body*)) # *FRS*)
unfolding τ*Move2-compP2*[*OF sees*] **by**(*auto*)
ultimately have τ*Exec-1-dt* (*compP2 P*) *t* (*None*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *FRS*) (*None*, *h*,
(*v* # *drop* (*Suc* (*length Ts*)) *stk0*, *loc0*, *C0*, *M0*, *Suc pc0*) # *FRS′*)
by −(*erule rtranclp-into-tranclp1*,*rule* τ*exec-1-dI*)
moreover from *wt conf′ exec*

have *compTP P* ⊢ *t*:(*None*, *h*, (*v* # *drop* (*Suc* (*length Ts*)) *stk0*, *loc0*, *C0*, *M0*, *Suc pc0*) # *FRS′*) √
by(*rule BV-correct-1*)
hence *bisim1-list1 t h* (*?ee*, *xs′*) *exs′ None* ((*v* # *drop* (*Suc* (*length Ts*)) *stk0*, *loc0*, *C0*, *M0*, *Suc pc0*) # *FRS′*)
using *sees0*
proof
from *bisim1-inline-call-Val*[*OF bisim0 call0*, *of length Ts v*] *ins pc0*
show *P*,*blocks1 0* (*Class D0*#*Ts0*) *body0*,*h* ⊢ (*?ee*, *xs′*) ↔ (*v* # *drop* (*Suc* (*length Ts*)) *stk0*, *loc0*, *Suc pc0*, *None*)
by *simp*
from *lenxs0 max-vars-inline-call*[*of e E*]
show *max-vars* (*inline-call e E*) ≤ *length xs′* **by** *simp*
from *bisims* **show** *list-all2* (*bisim1-fr P h*) *exs′ FRS′* **by** *simp*
qed
ultimately show *?thesis*
by −(*rule exI conjI*|*assumption*|*simp*)+
next
fix *ad*
assume [*simp*]: *e* = *Throw ad*

have ∃ *stk′ pc′*. *τExec-mover-a P t body h* (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc*, *pc′*, ⌊*ad*⌋) ∧
P,*blocks1 0* (*Class D*#*Ts*) *body*,*h* ⊢ (*Throw ad*, *loc*) ↔ (*stk′*, *loc*, *pc′*, ⌊*ad*⌋)
proof(*cases xcp*)
case [*simp*]: *None*
from *bisim1-Throw-τExec-mover*[*OF bisim*[*unfolded None* ‹*e* = *Throw ad*›]] **obtain** *pc′*
where *exec*: *τExec-mover-a P t body h* (*stk*, *loc*, *pc*, *None*) ([*Addr ad*], *loc*, *pc′*, ⌊*ad*⌋)
and *bisim′*: *P*,*blocks1 0* (*Class D*#*Ts*) *body*,*h* ⊢ (*Throw ad*, *xs*) ↔ ([*Addr ad*], *loc*, *pc′*, ⌊*ad*⌋)
and [*simp*]: *xs* = *loc* **by**(*auto simp del*: *blocks1.simps*)
thus *?thesis* **by** *fastforce*
next
case (*Some a′*)
with *bisim* **have** *a′* = *ad xs* = *loc* **by**(*auto dest*: *bisim1-ThrowD*)
thus *?thesis* **using** *bisim Some* **by**(*auto*)
qed
then obtain *stk′ pc′* **where** *exec*: *τExec-mover-a P t body h* (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc*, *pc′*, ⌊*ad*⌋)
and *bisim′*: *P*,*blocks1 0* (*Class D*#*Ts*) *body*,*h* ⊢ (*Throw ad*, *loc*) ↔ (*stk′*, *loc*, *pc′*, ⌊*ad*⌋) **by** *blast*
with *sees* **have** *τExec-1r* (*compP2 P*) *t* (*xcp*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *FRS*) (⌊*ad*⌋, *h*, (*stk′*, *loc*, *C*, *M*, *pc′*) # *FRS*)
by−(*rule τExec-mover-τExec-1r*)
with *conf wt* **have** *τExec-1-dr* (*compP2 P*) *t* (*xcp*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *FRS*) (⌊*ad*⌋, *h*, (*stk′*, *loc*, *C*, *M*, *pc′*) # *FRS*)
by(*simp*)(*rule τExec-1r-τExec-1-dr*)
moreover with *conf wt* **have** *conf′*: *compTP P* ⊢ *t*: (⌊*ad*⌋, *h*, (*stk′*, *loc*, *C*, *M*, *pc′*) # *FRS*) √
by(*simp*)(*rule τExec-1-dr-preserves-correct-state*)
from *bisim1-xcp-Some-not-caught*[*OF bisim′*, *of* λ*C M Ts T*. *compMb2 0 0*] *sees*
have *match*: *match-ex-table* (*compP2 P*) (*cname-of h ad*) *pc′* (*ex-table-of* (*compP2 P*) *C M*) = *None*
by(*simp add*: *compP2-def compMb2-def*)
hence *exec*: *exec-1* (*compP2 P*) *t* (⌊*ad*⌋, *h*, (*stk′*, *loc*, *C*, *M*, *pc′*) # *FRS*) *ε* (⌊*ad*⌋, *h*, *FRS*) **by**(*simp add*: *exec-1-iff*)
moreover

```
      with conf' wt have compP2 P,t ⊢ Normal (⌊ad⌋, h, (stk', loc, C, M, pc') # FRS) −ε−jvmd→
Normal (⌊ad⌋, h, FRS)
        by(simp add: welltyped-commute)
      hence check (compP2 P) (⌊ad⌋, h, (stk', loc, C, M, pc') # FRS) by(rule jvmd-NormalE)
      moreover from bisim' have τMove2 (compP2 P) (⌊ad⌋, h, (stk', loc, C, M, pc') # FRS)
        unfolding τMove2-compP2[OF sees] by(auto dest: bisim1-pc-length-compE2)
       ultimately have τExec-1-dt (compP2 P) t (xcp, h, (stk, loc, C, M, pc) # FRS) (⌊ad⌋, h,
FRS)
        by −(erule rtranclp-into-tranclp1,rule τexec-1-dI)
      moreover from wt conf' exec
      have compTP P ⊢ t: (⌊ad⌋, h, (stk0, loc0, C0, M0, pc0) # FRS') √
        by(simp)(rule BV-correct-1)
      hence bisim1-list1 t h (?ee, xs') exs' ⌊ad⌋ ((stk0, loc0, C0, M0, pc0) # FRS')
        using sees0
      proof
        from bisim1-inline-call-Throw[OF bisim0 call0] ins pc0
        show P,blocks1 0 (Class D0#Ts0) body0,h ⊢ (?ee, xs') ↔ (stk0, loc0, pc0, ⌊ad⌋) by simp
        from lenxs0 max-vars-inline-call[of e E]
        show max-vars ?ee ≤ length xs' by simp
        from bisims Cons show list-all2 (bisim1-fr P h) exs' FRS' by simp
      qed
      moreover from call0 have sim12-size (inline-call (Throw ad) E) > 0 by(cases E) simp-all
      ultimately show ?thesis
        by −(rule exI conjI|assumption|simp)+
    qed
  qed
 qed
qed

lemma exec-1-simulates-Red1-not-τ:
 assumes wf: wf-J1-prog P
 and Red1: True,P,t ⊢1 ⟨(e, xs)/exs, h⟩ −ta→ ⟨(e', xs')/exs', h'⟩
 and bisim: bisim1-list1 t h (e, xs) exs xcp frs
 and τ: ¬ τMove1 P h ((e, xs), exs)
 shows ∃ xcp' frs'. τExec-1-dr (compP2 P) t (xcp, h, frs) (xcp', h, frs') ∧
       (∃ ta' xcp'' frs''. exec-1-d (compP2 P) t (Normal (xcp', h, frs')) ta' (Normal (xcp'', h', frs''))
∧
                    ¬ τMove2 (compP2 P) (xcp', h, frs') ∧ ta-bisim wbisim1 ta ta' ∧
            bisim1-list1 t h' (e',xs') exs' xcp'' frs'') ∧
        (call1 e = None ∨
         (case frs of Nil ⇒ False | (stk, loc, C, M, pc) # FRS ⇒ ∀ M' n. instrs-of (compP2 P) C
M ! pc ≠ Invoke M' n) ∨
          xcp'= xcp ∧ frs' = frs)
using Red1
proof(cases)
 case (red1Red TA)
 hence [simp]: ta = extTA2J1 P TA exs' = exs
  and red: True,P,t ⊢1 ⟨e,(h, xs)⟩ −TA→ ⟨e',(h', xs')⟩ by simp-all
 from red have hext: hext h h' by(auto dest: red1-hext-incr)
 from τ have τ': ¬ τmove1 P h e by(auto intro: τmove1Block)
 note wt = wt-compTP-compP2[OF wf]
 from bisim show ?thesis
 proof(cases)
  case (bl1-Normal stk loc C M pc FRS Ts T body D)
```

hence [*simp*]: *frs* = (*stk*, *loc*, *C*, *M*, *pc*) # *FRS*
and *conf*: *compTP P* ⊢ *t*: (*xcp*, *h*, *frs*) √
and *sees*: *P* ⊢ *C sees M*: *Ts*→*T* = ⌊*body*⌋ *in D*
and *bisim*: *P*,*blocks1 0* (*Class D*#*Ts*) *body*,*h* ⊢ (*e*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
and *bisims*: *list-all2* (*bisim1-fr P h*) *exs FRS*
and *lenxs*: *max-vars e* ≤ *length xs* **by** *auto*
from *sees wf* **have** *bsok* (*blocks1 0* (*Class D* # *Ts*) *body*) *0*
by(*auto dest*!: *sees-wf-mdecl WT1-expr-locks simp add*: *wf-J1-mdecl-def wf-mdecl-def bsok-def*)

from *exec-instr-simulates-red1*[*OF wf bisim red this*] *τ′*
obtain *pc′ stk′ loc′ xcp′ pc″ stk″ loc″ xcp″*
where *exec1*: *τExec-mover-a P t body h* (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*)
and *exec2*: *exec-move-a P t body h* (*stk′*, *loc′*, *pc′*, *xcp′*) (*extTA2JVM* (*compP2 P*) *TA*) *h′* (*stk″*, *loc″*, *pc″*, *xcp″*)
and *τ2*: ¬ *τmove2* (*compP2 P*) *h stk′ body pc′ xcp′*
and *b′*: *P*,*blocks1 0* (*Class D*#*Ts*) *body*, *h′* ⊢ (*e′*, *xs′*) ↔ (*stk″*, *loc″*, *pc″*, *xcp″*)
and *call*: *call1 e* = *None* ∨ *no-call2* (*blocks1 0* (*Class D* # *Ts*) *body*) *pc* ∨ *pc′* = *pc* ∧ *stk′* = *stk* ∧ *loc′* = *loc* ∧ *xcp′* = *xcp*
by(*fastforce simp add*: *exec-move-def simp del*: *blocks1.simps*)
from *exec2* **have** *pc′body*: *pc′* < *length* (*compE2 body*) **by**(*auto*)
from *exec1 sees* **have** *exec1′*: *τExec-1r* (*compP2 P*) *t* (*xcp*, *h*, *frs*) (*xcp′*, *h*, (*stk′*, *loc′*, *C*, *M*, *pc′*) # *FRS*)
by(*auto intro*: *τExec-mover-τExec-1r*)
with *wt* **have** *execd*: *τExec-1-dr* (*compP2 P*) *t* (*xcp*, *h*, *frs*) (*xcp′*, *h*, (*stk′*, *loc′*, *C*, *M*, *pc′*) # *FRS*)
using *conf* **by**(*rule τExec-1r-τExec-1-dr*)
moreover { **fix** *a*
assume [*simp*]: *xcp′* = ⌊*a*⌋
from *exec2 sees-method-compP*[*OF sees*, *of* λ*C M Ts T*. *compMb2*] *pc′body*
have *match-ex-table* (*compP2 P*) (*cname-of h a*) *pc′* (*ex-table-of* (*compP2 P*) *C M*) ≠ *None*
by(*auto simp add*: *exec-move-def compP2-def compMb2-def elim*!: *exec-meth.cases*) }
note *xt* = *this*
with *τ2 sees pc′body* **have** *τ2′*: ¬ *τMove2* (*compP2 P*) (*xcp′*, *h*, (*stk′*, *loc′*, *C*, *M*, *pc′*) # *FRS*)
unfolding *τMove2-compP2*[*OF sees*] **by**(*auto simp add*: *compP2-def compMb2-def τmove2-iff*)
moreover from *exec2 sees*
have *exec2′*: *exec-1* (*compP2 P*) *t* (*xcp′*, *h*, (*stk′*, *loc′*, *C*, *M*, *pc′*) # *FRS*) (*extTA2JVM* (*compP2 P*) *TA*) (*xcp″*, *h′*, (*stk″*, *loc″*, *C*, *M*, *pc″*) # *FRS*)
by(*rule exec-move-exec-1*)
from *wt execd conf* **have** *conf′*: *compTP P* ⊢ *t*: (*xcp′*, *h*, (*stk′*, *loc′*, *C*, *M*, *pc′*) # *FRS*) √
by(*rule τExec-1-dr-preserves-correct-state*)
with *exec2′ wt*
have *exec-1-d* (*compP2 P*) *t* (*Normal* (*xcp′*, *h*, (*stk′*, *loc′*, *C*, *M*, *pc′*) # *FRS*)) (*extTA2JVM* (*compP2 P*) *TA*) (*Normal* (*xcp″*, *h′*, (*stk″*, *loc″*, *C*, *M*, *pc″*) # *FRS*))
by(*simp add*: *welltyped-commute*)
moreover
from *τ2 sees pc′body xt* **have** *τ2′*: ¬ *τMove2* (*compP2 P*) (*xcp′*, *h*, (*stk′*, *loc′*, *C*, *M*, *pc′*) # *FRS*)
unfolding *τMove2-compP2*[*OF sees*] **by**(*auto simp add*: *compP2-def compMb2-def τmove2-iff*)
moreover from *wt conf′ exec2′*
have *conf″*: *compTP P* ⊢ *t*: (*xcp″*, *h′*, (*stk″*, *loc″*, *C*, *M*, *pc″*) # *FRS*) √ **by**(*rule BV-correct-1*)
hence *bisim1-list1 t h′* (*e′*, *xs′*) *exs xcp″* ((*stk″*, *loc″*, *C*, *M*, *pc″*) # *FRS*) **using** *sees b′*
proof
from *red1-preserves-len*[*OF red*] *red1-max-vars*[*OF red*] *lenxs*
show *max-vars e′* ≤ *length xs′* **by** *simp*

from *bisims* **show** *list-all2* (*bisim1-fr P h′*) *exs FRS*

```
      by(rule List.list-all2-mono)(rule bisim1-fr-hext-mono[OF - hext])
    qed
    moreover from conf'' have hconf h' preallocated h' by(auto simp add: correct-state-def)
    with wf red
    have ta-bisim wbisim1 ta (extTA2JVM (compP2 P) TA)
      by(auto intro: ta-bisim-red-extTA2J1-extTA2JVM)
    moreover from call sees-method-compP[OF sees, of λC M Ts T. compMb2]
    have call1 e = None ∨ (case frs of [] ⇒ False | (stk, loc, C, M, pc) # FRS ⇒ ∀ M' n. instrs-of
(compP2 P) C M ! pc ≠ Invoke M' n) ∨ xcp' = xcp ∧ (stk', loc', C, M, pc') # FRS = frs
      by(auto simp add: no-call2-def compP2-def compMb2-def)
    ultimately show ?thesis by −(rule exI conjI|assumption|simp)+
  next
    case bl1-finalVal
    with red show ?thesis by auto
  next
    case bl1-finalThrow
    with red show ?thesis by(auto elim: red1-cases)
  qed
next
  case red1Call
  with τ have False
   by(auto simp add: synthesized-call-def dest!: τmove1-not-call1[where P=P and h=h] dest: sees-method-fun)
  thus ?thesis ..
next
  case red1Return
  with τ have False by auto
  thus ?thesis ..
qed

end

end
```

7.18 Correctness of Stage 2: From JVM to intermediate language

```
theory JVMJ1 imports
  J1JVMBisim
begin

declare split-paired-Ex[simp del]

lemma rec-option-is-case-option: rec-option = case-option
apply (rule ext)+
apply (rename-tac y)
apply (case-tac y)
apply auto
done

context J1-JVM-heap-base begin

lemma assumes ha: typeof-addr h a = ⌊Class-type D⌋
  and subclsObj: P ⊢ D ≼* Throwable
```

shows *bisim1-xcp-τRed*:
⟦ *P*,*e*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋);
match-ex-table (*compP f P*) (*cname-of h a*) *pc* (*compxE2 e 0 0*) = *None*;
∃ *n*. *n* + *max-vars e′* ≤ *length xs* ∧ *B e n* ⟧
⟹ *τred1r P t h* (*e′*, *xs*) (*Throw a*, *loc*) ∧ *P*,*e*,*h* ⊢ (*Throw a*, *loc*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)

and *bisims1-xcp-τReds*:
⟦ *P*,*es*,*h* ⊢ (*es′*, *xs*) [↔] (*stk*, *loc*, *pc*, ⌊*a*⌋);
match-ex-table (*compP f P*) (*cname-of h a*) *pc* (*compxEs2 es 0 0*) = *None*;
∃ *n*. *n* + *max-varss es′* ≤ *length xs* ∧ *Bs es n* ⟧
⟹ ∃ *vs es″*. *τreds1r P t h* (*es′*, *xs*) (*map Val vs* @ *Throw a* # *es″*, *loc*) ∧
P,*es*,*h* ⊢ (*map Val vs* @ *Throw a* # *es″*, *loc*) [↔] (*stk*, *loc*, *pc*, ⌊*a*⌋)
proof(*induct* (*e′*, *xs*) (*stk*, *loc*, *pc*, ⌊*a* :: *'addr*⌋)
and (*es′*, *xs*) (*stk*, *loc*, *pc*, ⌊*a* :: *'addr*⌋)
arbitrary: *e′ xs stk loc pc* **and** *es′ xs stk loc pc rule*: *bisim1-bisims1.inducts*)
case *bisim1NewThrow* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1NewArray* **thus** *?case*
by(*auto intro*: *rtranclp.rtrancl-into-rtrancl New1ArrayThrow bisim1-bisims1.intros dest*: *bisim1-ThrowD elim*!: *NewArray-τred1r-xt*)
next
case *bisim1NewArrayThrow* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1NewArrayFail* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1Cast* **thus** *?case*
by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl Cast1Throw bisim1-bisims1.intros dest*: *bisim1-ThrowD elim*!: *Cast-τred1r-xt*)
next
case *bisim1CastThrow* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1CastFail* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1InstanceOf* **thus** *?case*
by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl InstanceOf1Throw bisim1-bisims1.intros dest*: *bisim1-ThrowD elim*!: *InstanceOf-τred1r-xt*)
next
case *bisim1InstanceOfThrow* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1BinOp1* **thus** *?case*
by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl Bin1OpThrow1 bisim1-bisims1.intros simp add*: *match-ex-table-append dest*: *bisim1-ThrowD elim*!: *BinOp-τred1r-xt1*)
next
case *bisim1BinOp2* **thus** *?case*
by(*clarsimp simp add*: *match-ex-table-append-not-pcs compxE2-size-convs compxE2-stack-xlift-convs match-ex-table-shift-pc-None*)
(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl red1-reds1.intros bisim1BinOpThrow2 elim*!: *BinOp-τred1r-xt2*)
next

case *bisim1BinOpThrow1* **thus** *?case*
by(*auto simp add*: *match-ex-table-append intro*: *bisim1-bisims1.intros*)
next
case (*bisim1BinOpThrow2 e xs stk loc pc e1 bop*)
note *bisim* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
hence *xs* = *loc* **by**(*auto dest*: *bisim1-ThrowD*)
with *bisim* **show** *?case*
by(*auto intro*: *bisim1-bisims1.bisim1BinOpThrow2*)
next
case *bisim1BinOpThrow* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1LAss1* **thus** *?case*
by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl LAss1Throw bisim1-bisims1.intros dest*: *bisim1-ThrowD elim*!: *LAss-τred1r*)
next
case *bisim1LAssThrow* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1AAcc1* **thus** *?case*
by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl AAcc1Throw1 bisim1-bisims1.intros simp add*: *match-ex-table-append dest*: *bisim1-ThrowD elim*!: *AAcc-τred1r-xt1*)
next
case *bisim1AAcc2* **thus** *?case*
by(*clarsimp simp add*: *match-ex-table-append-not-pcs compxE2-size-convs compxE2-stack-xlift-convs match-ex-table-shift-pc-None*)
(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl red1-reds1.intros bisim1AAccThrow2 elim*!: *AAcc-τred1r-xt2*)
next
case *bisim1AAccThrow1* **thus** *?case*
by(*auto simp add*: *match-ex-table-append intro*: *bisim1-bisims1.intros*)
next
case *bisim1AAccThrow2* **thus** *?case*
by(*auto simp add*: *match-ex-table-append intro*: *bisim1-bisims1.intros dest*: *bisim1-ThrowD*)
next
case *bisim1AAccFail* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1AAss1* **thus** *?case*
by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl AAss1Throw1 bisim1-bisims1.intros simp add*: *match-ex-table-append dest*: *bisim1-ThrowD elim*!: *AAss-τred1r-xt1*)
next
case *bisim1AAss2* **thus** *?case*
by(*clarsimp simp add*: *compxE2-size-convs compxE2-stack-xlift-convs*)
(*fastforce simp add*: *match-ex-table-append intro*: *rtranclp.rtrancl-into-rtrancl red1-reds1.intros bisim1AAssThrow2 elim*!: *AAss-τred1r-xt2*)
next
case *bisim1AAss3* **thus** *?case*
by(*clarsimp simp add*: *compxE2-size-convs compxE2-stack-xlift-convs*)
(*fastforce simp add*: *match-ex-table-append intro*: *rtranclp.rtrancl-into-rtrancl red1-reds1.intros bisim1AAssThrow3 elim*!: *AAss-τred1r-xt3*)
next
case *bisim1AAssThrow1* **thus** *?case*
by(*auto simp add*: *match-ex-table-append intro*: *bisim1-bisims1.intros*)
next

case (*bisim1AAssThrow2 e xs stk loc pc i e2*)
note *bisim* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
hence *xs* = *loc* **by**(*auto dest*: *bisim1-ThrowD*)
with *bisim* **show** *?case*
by(*auto intro*: *bisim1-bisims1.bisim1AAssThrow2*)
next
case (*bisim1AAssThrow3 e xs stk loc pc A i*)
note *bisim* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
hence *xs* = *loc* **by**(*auto dest*: *bisim1-ThrowD*)
with *bisim* **show** *?case*
by(*auto intro*: *bisim1-bisims1.bisim1AAssThrow3*)
next
case *bisim1AAssFail* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1ALength* **thus** *?case*
by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl ALength1Throw bisim1-bisims1.intros dest*: *bisim1-ThrowD elim*!: *ALength-*τ*red1r-xt*)
next
case *bisim1ALengthThrow* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1ALengthNull* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1FAcc* **thus** *?case*
by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl FAcc1Throw bisim1-bisims1.intros dest*: *bisim1-ThrowD elim*!: *FAcc-*τ*red1r-xt*)
next
case *bisim1FAccThrow* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1FAccNull* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1FAss1* **thus** *?case*
by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl FAss1Throw1 bisim1-bisims1.intros simp add*: *match-ex-table-app*
dest: *bisim1-ThrowD elim*!: *FAss-*τ*red1r-xt1*)
next
case *bisim1FAss2* **thus** *?case*
by(*clarsimp simp add*: *match-ex-table-append-not-pcs compxE2-size-convs compxE2-stack-xlift-convs*)
(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl red1-reds1.intros bisim1FAssThrow2 elim*!: *FAss-*τ*red1r-xt2*)
next
case *bisim1FAssThrow1* **thus** *?case*
by(*auto simp add*: *match-ex-table-append intro*: *bisim1-bisims1.intros*)
next
case (*bisim1FAssThrow2 e2 xs stk loc pc e*)
note *bisim* = ‹*P*,*e2*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
hence *xs* = *loc* **by**(*auto dest*: *bisim1-ThrowD*)
with *bisim* **show** *?case*
by(*auto intro*: *bisim1-bisims1.bisim1FAssThrow2*)
next
case *bisim1FAssNull* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)

next
case *bisim1CAS1* **thus** *?case*
by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl CAS1Throw bisim1-bisims1.intros simp add*: *match-ex-table-append dest*: *bisim1-ThrowD elim*!: *CAS-τred1r-xt1*)
next
case *bisim1CAS2* **thus** *?case*
by(*clarsimp simp add*: *compxE2-size-convs compxE2-stack-xlift-convs*)
(*fastforce simp add*: *match-ex-table-append intro*: *rtranclp.rtrancl-into-rtrancl red1-reds1.intros bisim1CASThrow2 elim*!: *CAS-τred1r-xt2*)
next
case *bisim1CAS3* **thus** *?case*
by(*clarsimp simp add*: *compxE2-size-convs compxE2-stack-xlift-convs*)
(*fastforce simp add*: *match-ex-table-append intro*: *rtranclp.rtrancl-into-rtrancl red1-reds1.intros bisim1CASThrow3 elim*!: *CAS-τred1r-xt3*)
next
case *bisim1CASThrow1* **thus** *?case*
by(*auto simp add*: *match-ex-table-append intro*: *bisim1-bisims1.intros*)
next
case (*bisim1CASThrow2 e xs stk loc pc i e2*)
note *bisim* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
hence *xs* = *loc* **by**(*auto dest*: *bisim1-ThrowD*)
with *bisim* **show** *?case*
by(*auto intro*: *bisim1-bisims1.bisim1CASThrow2*)
next
case (*bisim1CASThrow3 e xs stk loc pc A i*)
note *bisim* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
hence *xs* = *loc* **by**(*auto dest*: *bisim1-ThrowD*)
with *bisim* **show** *?case*
by(*auto intro*: *bisim1-bisims1.bisim1CASThrow3*)
next
case *bisim1CASFail* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1Call1* **thus** *?case*
by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl Call1ThrowObj bisim1-bisims1.intros simp add*: *match-ex-table-append dest*: *bisim1-ThrowD elim*!: *Call-τred1r-obj*)
next
case *bisim1CallParams* **thus** *?case*
by(*clarsimp simp add*: *match-ex-table-append-not-pcs compxE2-size-convs compxEs2-size-convs compxE2-stack-xlift-convs compxEs2-stack-xlift-convs match-ex-table-shift-pc-None*)
(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl Call1ThrowParams*[*OF refl*] *bisim1CallThrowParams elim*!: *Call-τred1r-param*)
next
case *bisim1CallThrowObj* **thus** *?case*
by(*auto simp add*: *match-ex-table-append intro*: *bisim1-bisims1.intros*)
next
case (*bisim1CallThrowParams es vs es′ xs stk loc pc obj M*)
note *bisim* = ‹*P*,*es*,*h* ⊢ (*map Val vs* @ *Throw a* # *es′*, *xs*) [↔] (*stk*, *loc*, *pc*, ⌊*a*⌋)›
hence *xs* = *loc* **by**(*auto dest*: *bisims1-ThrowD*)
with *bisim* **show** *?case*
by(*auto intro*: *bisim1-bisims1.bisim1CallThrowParams*)
next
case *bisim1CallThrow* **thus** *?case*
by(*auto intro*: *bisim1-bisims1.intros*)

next
case (*bisim1BlockSome4 e e' xs stk loc pc V Ty v*)
from ‹∃ *n*. *n* + *max-vars* {*V*:*Ty*=*None*; *e'*} ≤ *length xs* ∧ $\mathcal{B}$ {*V*:*Ty*=⌊*v*⌋; *e*} *n*›
have *V*: *V* < *length xs* **by** *simp*
from *bisim1BlockSome4* **have** *Red*: *τred1r P t h* (*e'*, *xs*) (*Throw a*, *loc*)
and *bisim*: *P*,*e*,*h* ⊢ (*Throw a*, *loc*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
by(*auto simp add*: *match-ex-table-append-not-pcs compxE2-size-convs compxEs2-size-convs compxE2-stack-xlift-convs compxEs2-stack-xlift-convs match-ex-table-shift-pc-None intro*!: *exI*[**where** *x*=*Suc V*] *elim*: *meta-impE*)
note *len* = *τred1r-preserves-len*[*OF Red*]
from *Red* **have** *τred1r P t h* ({*V*:*Ty*=*None*; *e'*}, *xs*) ({*V*:*Ty*=*None*; *Throw a*}, *loc*) **by**(*auto intro*: *Block-None-τred1r-xt*)
thus *?case* **using** *V len bisim*
by(*auto intro*: *τmove1BlockThrow Block1Throw bisim1BlockThrowSome elim*!: *rtranclp.rtrancl-into-rtrancl*)
next
case *bisim1BlockThrowSome* **thus** *?case*
by(*auto dest*: *bisim1-ThrowD intro*: *bisim1-bisims1.bisim1BlockThrowSome*)
next
case (*bisim1BlockNone e e' xs stk loc pc V Ty*)
hence *Red*: *τred1r P t h* (*e'*, *xs*) (*Throw a*, *loc*)
and *bisim*: *P*,*e*,*h* ⊢ (*Throw a*, *loc*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
by(*auto elim*: *meta-impE intro*!: *exI*[**where** *x*=*Suc V*])
from *Red* **have** *len*: *length loc* = *length xs* **by**(*rule τred1r-preserves-len*)
from ‹∃ *n*. *n* + *max-vars* {*V*:*Ty*=*None*; *e'*} ≤ *length xs* ∧ $\mathcal{B}$ {*V*:*Ty*=*None*; *e*} *n*›
have *V*: *V* < *length xs* **by** *simp*
from *Red* **have** *τred1r P t h* ({*V*:*Ty*=*None*; *e'*}, *xs*) ({*V*:*Ty*=*None*; *Throw a*}, *loc*) **by**(*rule Block-None-τred1r-xt*)
thus *?case* **using** *V len bisim*
by(*auto intro*: *τmove1BlockThrow Block1Throw bisim1BlockThrowNone elim*!: *rtranclp.rtrancl-into-rtrancl*)
next
case *bisim1BlockThrowNone* **thus** *?case*
by(*auto dest*: *bisim1-ThrowD intro*: *bisim1-bisims1.bisim1BlockThrowNone*)
next
case *bisim1Sync1* **thus** *?case*
by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl Synchronized1Throw1 bisim1-bisims1.intros simp add*: *match-ex-table-append dest*: *bisim1-ThrowD elim*!: *Sync-τred1r-xt*)
next
case (*bisim1Sync4 e2 e' xs stk loc pc V e1 a'*)
from ‹*P*,*e2*,*h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
have *pc* < *length* (*compE2 e2*) **by**(*auto dest*!: *bisim1-xcp-pcD*)
with ‹*match-ex-table* (*compP f P*) (*cname-of h a*) (*Suc* (*Suc* (*Suc* (*length* (*compE2 e1*) + *pc*)))) (*compxE2* (*sync*$_V$ (*e1*) *e2*) *0 0*) = *None*› *subclsObj ha*
have *False* **by**(*simp add*: *match-ex-table-append matches-ex-entry-def split*: *if-split-asm*)
thus *?case* **..**
next
case *bisim1Sync10* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1Sync11* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1Sync12* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1Sync14* **thus** *?case*

by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1SyncThrow* **thus** *?case*
by(*auto simp add*: *match-ex-table-append intro*: *bisim1-bisims1.intros*)
next
case *bisim1Seq1* **thus** *?case*
by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl Seq1Throw bisim1-bisims1.intros dest*: *bisim1-ThrowD elim*!: *Seq-τred1r-xt simp add*: *match-ex-table-append*)
next
case *bisim1SeqThrow1* **thus** *?case*
by(*auto simp add*: *match-ex-table-append intro*: *bisim1-bisims1.intros*)
next
case *bisim1Seq2* **thus** *?case*
by(*auto simp add*: *match-ex-table-append-not-pcs compxE2-size-convs compxE2-stack-xlift-convs match-ex-table-shift-pc-None intro*: *bisim1-bisims1.bisim1Seq2*)
next
case *bisim1Cond1* **thus** *?case*
by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl Cond1Throw bisim1-bisims1.intros dest*: *bisim1-ThrowD elim*!: *Cond-τred1r-xt simp add*: *match-ex-table-append*)
next
case *bisim1CondThen* **thus** *?case*
by(*clarsimp simp add*: *match-ex-table-append*)
(*auto simp add*: *match-ex-table-append-not-pcs compxE2-size-convs compxE2-stack-xlift-convs match-ex-table-shift-pc-None intro*: *bisim1-bisims1.bisim1CondThen*)
next
case *bisim1CondElse* **thus** *?case*
by(*clarsimp simp add*: *match-ex-table-append*)
(*auto simp add*: *match-ex-table-append-not-pcs compxE2-size-convs compxE2-stack-xlift-convs match-ex-table-shift-pc-None intro*: *bisim1-bisims1.bisim1CondElse*)
next
case *bisim1CondThrow* **thus** *?case*
by(*auto simp add*: *match-ex-table-append intro*: *bisim1-bisims1.intros*)
next
case *bisim1While3* **thus** *?case*
by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl Cond1Throw bisim1-bisims1.intros dest*: *bisim1-ThrowD elim*!: *Cond-τred1r-xt simp add*: *match-ex-table-append*)
next
case (*bisim1While4 e e′ xs stk loc pc c*)
hence *τred1r P t h* (*e′*, *xs*) (*Throw a*, *loc*) ∧ *P*,*e*,*h* ⊢ (*Throw a*, *loc*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
by(*auto simp add*: *match-ex-table-append-not-pcs compxE2-size-convs compxEs2-size-convs compxE2-stack-xlift-convs compxEs2-stack-xlift-convs match-ex-table-shift-pc-None*)
hence *τred1r P t h* (*e′*;;*while* (*c*) *e*, *xs*) (*Throw a*, *loc*)
P,*while* (*c*) *e*,*h* ⊢ (*Throw a*, *loc*) ↔ (*stk*, *loc*, *Suc* (*length* (*compE2 c*) + *pc*), ⌊*a*⌋)
by(*auto intro*: *rtranclp.rtrancl-into-rtrancl Seq1Throw τmove1SeqThrow bisim1WhileThrow2 elim*!: *Seq-τred1r-xt*)
thus *?case* **..**
next
case *bisim1WhileThrow1* **thus** *?case*
by(*auto simp add*: *match-ex-table-append intro*: *bisim1-bisims1.intros*)
next
case *bisim1WhileThrow2* **thus** *?case*
by(*auto simp add*: *match-ex-table-append intro*: *bisim1-bisims1.intros dest*: *bisim1-ThrowD*)
next
case *bisim1Throw1* **thus** *?case*

by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl Throw1Throw bisim1-bisims1.intros dest*: *bisim1-ThrowD elim*!: *Throw-τred1r-xt*)
next
case *bisim1Throw2* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1ThrowNull* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case *bisim1ThrowThrow* **thus** *?case*
by(*fastforce intro*: *bisim1-bisims1.intros*)
next
case (*bisim1Try e e′ xs stk loc pc C V e2*)
hence *red*: *τred1r P t h* (*e′*, *xs*) (*Throw a*, *loc*)
and *bisim*: *P*,*e*,*h* ⊢ (*Throw a*, *loc*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
by(*auto simp add*: *match-ex-table-append*)
from *red* **have** *Red*: *τred1r P t h* (*try e′ catch*(*C V*) *e2*, *xs*) (*try Throw a catch*(*C V*) *e2*, *loc*) **by**(*rule Try-τred1r-xt*)
from ‹*match-ex-table* (*compP f P*) (*cname-of h a*) *pc* (*compxE2* (*try e catch*(*C V*) *e2*) *0 0*) = *None*›
have ¬ *matches-ex-entry* (*compP f P*) (*cname-of h a*) *pc* (*0*, *length* (*compE2 e*), ⌊*C*⌋, *Suc* (*length* (*compE2 e*)), *0*)
by(*auto simp add*: *match-ex-table-append split*: *if-split-asm*)
moreover from ‹*P*,*e*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
have *pc* < *length* (*compE2 e*) **by**(*auto dest*: *bisim1-xcp-pcD*)
ultimately have *subcls*: ¬ *P* ⊢ *D* ≼* *C* **using** *ha* **by**(*simp add*: *matches-ex-entry-def cname-of-def*)
with *ha* **have** *True*,*P*,*t* ⊢1 ⟨*try Throw a catch*(*C V*) *e2*, (*h*, *loc*)⟩ −ε→ ⟨*Throw a*, (*h*, *loc*)⟩
by −(*rule Red1TryFail*, *auto*)
moreover from *bisim ha subcls*
have *P*,*try e catch*(*C V*) *e2*,*h* ⊢ (*Throw a*, *loc*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
by(*rule bisim1TryFail*)
ultimately show *?case* **using** *Red* **by**(*blast intro*: *rtranclp.rtrancl-into-rtrancl τmove1TryThrow*)
next
case (*bisim1TryCatch2 e2 e′ xs stk loc pc e1 C V*)
hence ∗: *τred1r P t h* (*e′*, *xs*) (*Throw a*, *loc*) ∧ *P*,*e2*,*h* ⊢ (*Throw a*, *loc*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
by(*clarsimp simp add*: *match-ex-table-append matches-ex-entry-def split*: *if-split-asm*)
(*auto simp add*: *match-ex-table-append compxE2-size-convs compxE2-stack-xlift-convs match-ex-table-shift-pc-No elim*: *meta-impE intro*!: *exI*[**where** *x*=*Suc V*])
moreover note *τred1r-preserves-len*[*OF* ∗[*THEN conjunct1*]]
moreover from ‹∃ *n*. *n* + *max-vars* {*V*:*Class C*=*None*; *e′*} ≤ *length xs* ∧ *B* (*try e1 catch*(*C V*) *e2*) *n*›
have *V* < *length xs* **by** *simp*
ultimately show *?case*
by(*fastforce intro*: *rtranclp.rtrancl-into-rtrancl Block1Throw τmove1BlockThrow bisim1TryCatchThrow elim*!: *Block-None-τred1r-xt*)
next
case (*bisim1TryFail e xs stk loc pc C′′ C′ V e2*)
note *bisim* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
hence *xs* = *loc* **by**(*auto dest*: *bisim1-ThrowD*)
with *bisim* ‹*typeof-addr h a* = ⌊*Class-type C′′*⌋› ‹¬ *P* ⊢ *C′′* ≼* *C′*›
show *?case* **by**(*auto intro*: *bisim1-bisims1.bisim1TryFail*)
next
case (*bisim1TryCatchThrow e2 xs stk loc pc e C′ V*)
from ‹*P*,*e2*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)› **have** *xs* = *loc*
by(*auto dest*: *bisim1-ThrowD*)

with ‹*P*,*e2*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)› **show** *?case*
by(*auto intro*: *bisim1-bisims1.bisim1TryCatchThrow*)
next
case (*bisims1List1 e e′ xs stk loc pc es*)
hence *τred1r P t h* (*e′*, *xs*) (*Throw a*, *loc*)
and *bisim*: *P*,*e*,*h* ⊢ (*Throw a*, *loc*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋) **by**(*auto simp add*: *match-ex-table-append*)
hence *τreds1r P t h* (*e′* # *es*, *xs*) (*map Val* [] @ *Throw a* # *es*, *loc*) **by**(*auto intro*: *τred1r-inj-τreds1r*)
moreover from *bisim*
have *P*,*e*#*es*,*h* ⊢ (*Throw a* # *es*, *loc*) [↔] (*stk*, *loc*, *pc*, ⌊*a*⌋)
by(*rule bisim1-bisims1.bisims1List1*)
ultimately show *?case* **by** *fastforce*
next
case (*bisims1List2 es es′ xs stk loc pc e v*)
hence ∃ *vs es″*. *τreds1r P t h* (*es′*, *xs*) (*map Val vs* @ *Throw a* # *es″*, *loc*) ∧ *P*,*es*,*h* ⊢ (*map Val vs* @ *Throw a* # *es″*, *loc*) [↔] (*stk*, *loc*, *pc*, ⌊*a*⌋)
by(*auto simp add*: *match-ex-table-append-not-pcs compxEs2-size-convs compxEs2-stack-xlift-convs match-ex-table-shift-pc-None*)
then obtain *vs es″* **where** *red*: *τreds1r P t h* (*es′*, *xs*) (*map Val vs* @ *Throw a* # *es″*, *loc*)
and *bisim*: *P*,*es*,*h* ⊢ (*map Val vs* @ *Throw a* # *es″*, *loc*) [↔] (*stk*, *loc*, *pc*, ⌊*a*⌋) **by** *blast*
from *red* **have** *τreds1r P t h* (*Val v* # *es′*, *xs*) (*map Val* (*v* # *vs*) @ *Throw a* # *es″*, *loc*)
by(*auto intro*: *τreds1r-cons-τreds1r*)
moreover from *bisim*
have *P*,*e* # *es*,*h* ⊢ (*map Val* (*v* # *vs*) @ *Throw a* # *es″*, *loc*) [↔] (*stk* @ [*v*], *loc*, *length* (*compE2 e*) + *pc*, ⌊*a*⌋)
by(*auto intro*: *bisim1-bisims1.bisims1List2*)
ultimately show *?case* **by** *fastforce*
qed

primrec *conf-xcp′* :: *′m prog* ⇒ *′heap* ⇒ *′addr option* ⇒ *bool* **where**
conf-xcp′ P h None = *True*
| *conf-xcp′ P h* ⌊*a*⌋ = (∃ *D*. *typeof-addr h a* = ⌊*Class-type D*⌋ ∧ *P* ⊢ *D* ⪯* *Throwable*)

lemma *conf-xcp-conf-xcp′*:
conf-xcp P h xcp i ⟹ *conf-xcp′ P h xcp*
by(*cases xcp*) *auto*

lemma *conf-xcp′-compP* [*simp*]: *conf-xcp′* (*compP f P*) = *conf-xcp′ P*
by(*clarsimp simp add*: *fun-eq-iff conf-xcp′-def rec-option-is-case-option*)

end

context *J1-heap-base* **begin**

lemmas *τred1-Val-simps* [*simp*] =
τred1r-Val τred1t-Val τreds1r-map-Val τreds1t-map-Val

end

context *J1-JVM-conf-read* **begin**

lemma assumes *wf*: *wf-J1-prog P*
and *hconf*: *hconf h preallocated h*
and *tconf*: *P*,*h* ⊢ *t* √*t*
shows *red1-simulates-exec-instr*:

⟦ P, E, $h \vdash (e, xs) \leftrightarrow (stk, loc, pc, xcp)$;
$exec\text{-}move\text{-}d\ P\ t\ E\ h\ (stk, loc, pc, xcp)\ ta\ h'\ (stk', loc', pc', xcp')$;
$n + max\text{-}vars\ e \leq length\ xs$; $bsok\ E\ n$; $P,h \vdash stk\ [:\leq]\ ST$; $conf\text{-}xcp'\ (compP2\ P)\ h\ xcp$ ⟧
$\Longrightarrow \exists e''\ xs''.\ P, E, h' \vdash (e'', xs'') \leftrightarrow (stk', loc', pc', xcp') \wedge$
(*if* $\tau move2\ (compP2\ P)\ h\ stk\ E\ pc\ xcp$
then $h' = h \wedge (if\ xcp' = None \longrightarrow pc < pc'\ then\ \tau red1r\ else\ \tau red1t)\ P\ t\ h\ (e, xs)\ (e'', xs'')$
else $\exists ta'\ e'\ xs'.\ \tau red1r\ P\ t\ h\ (e, xs)\ (e', xs') \wedge True,P,t \vdash 1\ \langle e', (h, xs')\rangle\ -ta'\rightarrow\ \langle e'', (h', xs'')\rangle \wedge ta\text{-}bisim\ wbisim1\ (extTA2J1\ P\ ta')\ ta \wedge \neg\ \tau move1\ P\ h\ e' \wedge (call1\ e = None \vee no\text{-}call2\ E\ pc \vee e' = e \wedge xs' = xs)$)
(**is** ⟦ -; *?exec E stk loc pc xcp stk' loc' pc' xcp'*; -; -; -; - ⟧
$\Longrightarrow \exists e''\ xs''.\ P, E, h' \vdash (e'', xs'') \leftrightarrow (stk', loc', pc', xcp') \wedge$ *?red e xs e'' xs'' E stk pc pc' xcp xcp'*)

and *reds1-simulates-exec-instr*:
⟦ P, Es, $h \vdash (es, xs)\ [\leftrightarrow]\ (stk, loc, pc, xcp)$;
$exec\text{-}moves\text{-}d\ P\ t\ Es\ h\ (stk, loc, pc, xcp)\ ta\ h'\ (stk', loc', pc', xcp')$;
$n + max\text{-}varss\ es \leq length\ xs$; $bsoks\ Es\ n$; $P,h \vdash stk\ [:\leq]\ ST$; $conf\text{-}xcp'\ (compP2\ P)\ h\ xcp$ ⟧
$\Longrightarrow \exists es''\ xs''.\ P, Es, h' \vdash (es'', xs'')\ [\leftrightarrow]\ (stk', loc', pc', xcp') \wedge$
(*if* $\tau moves2\ (compP2\ P)\ h\ stk\ Es\ pc\ xcp$
then $h' = h \wedge (if\ xcp' = None \longrightarrow pc < pc'\ then\ \tau reds1r\ else\ \tau reds1t)\ P\ t\ h\ (es, xs)\ (es'', xs'')$
else $\exists ta'\ es'\ xs'.\ \tau reds1r\ P\ t\ h\ (es, xs)\ (es', xs') \wedge True,P,t \vdash 1\ \langle es', (h, xs')\rangle\ [-ta'\rightarrow]\ \langle es'', (h', xs'')\rangle \wedge ta\text{-}bisim\ wbisim1\ (extTA2J1\ P\ ta')\ ta \wedge \neg\ \tau moves1\ P\ h\ es' \wedge (calls1\ es = None \vee no\text{-}calls2\ Es\ pc \vee es' = es \wedge xs' = xs)$)
(**is** ⟦ -; *?execs Es stk loc pc xcp stk' loc' pc' xcp'*; -; -; -; - ⟧
$\Longrightarrow \exists es''\ xs''.\ P, Es, h' \vdash (es'', xs'')\ [\leftrightarrow]\ (stk', loc', pc', xcp') \wedge$ *?reds es xs es'' xs'' Es stk pc pc' xcp xcp'*)
proof(*induction E n e xs stk loc pc xcp* **and** *Es n es xs stk loc pc xcp*
arbitrary: *stk' loc' pc' xcp' ST* **and** *stk' loc' pc' xcp' ST rule*: *bisim1-bisims1-inducts-split*)
case (*bisim1Val2 e n v xs*)
from ‹*?exec e* [*v*] *xs* (*length* (*compE2 e*)) *None stk' loc' pc' xcp'*›
have *False* **by**(*auto dest*: *exec-meth-length-compE2D simp add*: *exec-move-def*)
thus *?case* **..**
next
case (*bisim1New C' n xs*)
note *exec* = ‹*exec-move-d P t* (*new C'*) *h* ([], *xs*, *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)›
have τ: ¬ τ*move2* (*compP2 P*) *h* [] (*new C'*) *0 None* ¬ τ*move1 P h* (*new C'*) **by**(*auto simp add*: τ*move2-iff*)
show *?case*
proof(*cases allocate h* (*Class-type C'*) = {})
case *True*
have *P,new C',h'* ⊢ (*THROW OutOfMemory*, *xs*) ↔ ([], *xs*, *0*, ⌊*addr-of-sys-xcpt OutOfMemory*⌋)
by(*rule bisim1NewThrow*)
with *exec* τ *True* **show** *?thesis*
by(*fastforce intro*: *Red1NewFail elim*!: *exec-meth.cases simp add*: *exec-move-def*)
next
case *False*
have ⋀*a h'*. *P,new C',h'* ⊢ (*addr a*, *xs*) ↔ ([*Addr a*], *xs*, *length* (*compE2* (*new C'*)), *None*)
by(*rule bisim1Val2*) *auto*
thus *?thesis* **using** *exec False* τ
apply(*simp add*: *exec-move-def*)
apply(*erule exec-meth.cases*)
apply *simp-all*
apply *clarsimp*
apply(*auto intro*!: *Red1New exI simp add*: *ta-bisim-def*)

```
    done
  qed
next
  case (bisim1NewThrow C' n xs)
  from ‹?exec (new C') [] xs 0 ⌊addr-of-sys-xcpt OutOfMemory⌋ stk' loc' pc' xcp'›
  have False by(auto elim: exec-meth.cases simp add: exec-move-def)
  thus ?case ..
next
  case (bisim1NewArray e n e' xs stk loc pc xcp U)
  note IH = bisim1NewArray.IH(2)
  note exec = ‹?exec (newA U⌊e⌉) stk loc pc xcp stk' loc' pc' xcp'›
  note bisim = ‹P,e,h ⊢ (e', xs) ↔ (stk, loc, pc, xcp)›
  note len = ‹n + max-vars (newA U⌊e⌉) ≤ length xs›
  note bsok = ‹bsok (newA U⌊e⌉) n›
  from bisim have pc: pc ≤ length (compE2 e) by(rule bisim1-pc-length-compE2)
  show ?case
  proof(cases pc < length (compE2 e))
    case True
    with exec have exec': ?exec e stk loc pc xcp stk' loc' pc' xcp'
      by(auto simp add: exec-move-newArray)
    from True have τmove2 (compP2 P) h stk (newA U⌊e⌉) pc xcp = τmove2 (compP2 P) h stk e
pc xcp by(simp add: τmove2-iff)
    moreover have no-call2 e pc ⟹ no-call2 (newA U⌊e⌉) pc by(simp add: no-call2-def)
    ultimately show ?thesis using IH[OF exec' - - ‹P,h ⊢ stk [:≤] ST› ‹conf-xcp' (compP2 P) h
xcp›] len bsok
      by(fastforce intro: bisim1-bisims1.bisim1NewArray New1ArrayRed elim!: NewArray-τred1r-xt
NewArray-τred1t-xt)
  next
    case False
    with pc have [simp]: pc = length (compE2 e) by simp
    with bisim obtain v where stk: stk = [v] and xcp: xcp = None
      by(auto dest: bisim1-pc-length-compE2D)
    with bisim pc len bsok have red: τred1r P t h (e', xs) (Val v, loc)
      by(auto intro: bisim1-Val-τred1r simp add: bsok-def)
    hence τred1r P t h (newA U⌊e⌉, xs) (newA U⌊Val v⌉, loc) by(rule NewArray-τred1r-xt)
    moreover have τ: ¬ τmove2 (compP2 P) h [v] (newA U⌊e⌉) pc None by(simp add: τmove2-iff)
    moreover have ¬ τmove1 P h (newA U⌊Val v⌉) by auto
    moreover from exec stk xcp obtain I
      where [simp]: v = Intg I by(auto elim!: exec-meth.cases simp add: exec-move-def)
    have ∃ta' e''. P,newA U⌊e⌉,h' ⊢ (e'',loc) ↔ (stk', loc', pc', xcp') ∧ True,P,t ⊢1 ⟨newA U⌊Val
v⌉,(h, loc)⟩ −ta'→ ⟨e'',(h', loc)⟩ ∧ ta-bisim wbisim1 (extTA2J1 P ta') ta
    proof(cases I <s 0)
      case True with exec stk xcp show ?thesis
       by(fastforce elim!: exec-meth.cases intro: bisim1NewArrayFail Red1NewArrayNegative simp add:
exec-move-def)
    next
      case False
      show ?thesis
      proof(cases allocate h (Array-type U (nat (sint I))) = {})
        case True
        with False exec stk xcp show ?thesis
          by(fastforce elim!: exec-meth.cases intro: bisim1NewArrayFail Red1NewArrayFail simp add:
exec-move-def)
      next
```

```
      case False
      have ⋀a h'. P,newA U⌊e⌉,h' ⊢ (addr a, loc) ↔ ([Addr a], loc, length (compE2 (newA U⌊e⌉)),
None)
        by(rule bisim1Val2) simp
      with False ‹¬ I <s 0› exec stk xcp show ?thesis
        apply(simp add: exec-move-def)
        apply(erule exec-meth.cases)
        apply simp-all
        apply clarsimp
        apply(auto intro!: Red1NewArray exI simp add: ta-bisim-def)
        done
    qed
   qed
   moreover have no-call2 (newA U⌊e⌉) pc by(simp add: no-call2-def)
   ultimately show ?thesis using exec stk xcp by fastforce
  qed
next
  case (bisim1NewArrayThrow e n a xs stk loc pc U)
  note exec = ‹?exec (newA U⌊e⌉) stk loc pc ⌊a⌋ stk' loc' pc' xcp'›
  note bisim = ‹P,e,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
  from bisim have pc: pc < length (compE2 e) by(auto dest: bisim1-ThrowD)
  from bisim have match-ex-table (compP2 P) (cname-of h a) (0 + pc) (compxE2 e 0 0) = None
    unfolding compP2-def by(rule bisim1-xcp-Some-not-caught)
  with exec pc have False by (auto elim!: exec-meth.cases simp add: exec-move-def)
  thus ?case ..
next
  case (bisim1NewArrayFail e n U a xs v)
  note exec = ‹?exec (newA U⌊e⌉) [v] xs (length (compE2 e)) ⌊a⌋ stk' loc' pc' xcp'›
 hence False by(auto elim!: exec-meth.cases dest: match-ex-table-pc-length-compE2 simp add: exec-move-def)
  thus ?case ..
next
  case (bisim1Cast e n e' xs stk loc pc xcp U)
  note IH = bisim1Cast.IH(2)
  note exec = ‹?exec (Cast U e) stk loc pc xcp stk' loc' pc' xcp'›
  note bisim = ‹P,e,h ⊢ (e', xs) ↔ (stk, loc, pc, xcp)›
  note len = ‹n + max-vars (Cast U e') ≤ length xs›
  note ST = ‹P,h ⊢ stk [:≤] ST›
  note bsok = ‹bsok (Cast U e) n›
  from bisim have pc: pc ≤ length (compE2 e) by(rule bisim1-pc-length-compE2)
  show ?case
  proof(cases pc < length (compE2 e))
    case True
    with exec have exec': ?exec e stk loc pc xcp stk' loc' pc' xcp' by(auto simp add: exec-move-Cast)
    from True have τmove2 (compP2 P) h stk (Cast U e) pc xcp = τmove2 (compP2 P) h stk e pc
xcp by(simp add: τmove2-iff)
    moreover have no-call2 e pc ⟹ no-call2 (Cast U e) pc by(simp add: no-call2-def)
    ultimately show ?thesis using IH[OF exec' - - ST ‹conf-xcp' (compP2 P) h xcp›] len bsok
      by(fastforce intro: bisim1-bisims1.bisim1Cast Cast1Red elim!: Cast-τred1r-xt Cast-τred1t-xt)
  next
    case False
    with pc have [simp]: pc = length (compE2 e) by simp
    with bisim obtain v where stk: stk = [v] and xcp: xcp = None
      by(auto dest: bisim1-pc-length-compE2D)
    with bisim pc len bsok have red: τred1r P t h (e', xs) (Val v, loc)
```

by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *τred1r P t h* (*Cast U e'*, *xs*) (*Cast U* (*Val v*), *loc*) **by**(*rule Cast-τred1r-xt*)
also from *exec* **have** [*simp*]: $h' = h$ $ta = \varepsilon$ **by**(*auto simp add*: *exec-move-def elim*!: *exec-meth.cases split*: *if-split-asm*)
have $\exists e''$. *P*,*Cast U e*,*h* ⊢ (*e''*,*loc*) ↔ (*stk'*, *loc'*, *pc'*, *xcp'*) ∧ *True*,*P*,*t* ⊢*1* ⟨*Cast U* (*Val v*),(*h*, *loc*)⟩ −ε→ ⟨*e''*,(*h*, *loc*)⟩
proof(*cases P* ⊢ *the* (*typeof*$_h$ *v*) ≤ *U*)
case *False* **with** *exec stk xcp bsok ST* **show** *?thesis*
by(*fastforce simp add*: *compP2-def exec-move-def exec-meth-instr list-all2-Cons1 conf-def intro*: *bisim1CastFail Red1CastFail*)
next
case *True*
have *P*,*Cast U e*,*h* ⊢ (*Val v*, *loc*) ↔ ([*v*], *loc*, *length* (*compE2* (*Cast U e*)), *None*)
by(*rule bisim1Val2*) *simp*
with *exec stk xcp ST True* **show** *?thesis*
by(*fastforce simp add*: *compP2-def exec-move-def exec-meth-instr list-all2-Cons1 conf-def intro*: *Red1Cast*)
qed
then obtain *e''* **where** *bisim'*: *P*,*Cast U e*,*h* ⊢ (*e''*,*loc*) ↔ (*stk'*, *loc'*, *pc'*, *xcp'*)
and *red*: *True*,*P*,*t* ⊢*1* ⟨*Cast U* (*Val v*),(*h*, *loc*)⟩ −ε→ ⟨*e''*,(*h*, *loc*)⟩ **by** *blast*
have *τmove1 P h* (*Cast U* (*Val v*)) **by**(*rule τmove1CastRed*)
with *red* **have** *τred1t P t h* (*Cast U* (*Val v*), *loc*) (*e''*, *loc*) **by**(*auto intro*: *τred1t-1step*)
also have *τ*: *τmove2* (*compP2 P*) *h* [*v*] (*Cast U e*) *pc None* **by**(*simp add*: *τmove2-iff*)
moreover have *no-call2* (*Cast U e*) *pc* **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *exec stk xcp bisim'* **by** *fastforce*
qed
next
case (*bisim1CastThrow e n a xs stk loc pc U*)
note *exec* = ‹*?exec* (*Cast U e*) *stk loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*›
note *bisim* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
from *bisim* **have** *pc*: *pc* < *length* (*compE2 e*) **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim* **have** *match-ex-table* (*compP2 P*) (*cname-of h a*) (*0* + *pc*) (*compxE2 e 0 0*) = *None*
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)
with *exec pc* **have** *False* **by** (*auto elim*!: *exec-meth.cases simp add*: *exec-move-def*)
thus *?case* **..**
next
case (*bisim1CastFail e n U xs v*)
note *exec* = ‹*?exec* (*Cast U e*) [*v*] *xs* (*length* (*compE2 e*)) ⌊*addr-of-sys-xcpt ClassCast*⌋ *stk' loc' pc' xcp'*›
hence *False* **by**(*auto elim*!: *exec-meth.cases dest*: *match-ex-table-pc-length-compE2 simp add*: *exec-move-def*)
thus *?case* **..**
next
case (*bisim1InstanceOf e n e' xs stk loc pc xcp U*)
note *IH* = *bisim1InstanceOf.IH*(*2*)
note *exec* = ‹*?exec* (*e instanceof U*) *stk loc pc xcp stk' loc' pc' xcp'*›
note *bisim* = ‹*P*,*e*,*h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *len* = ‹*n* + *max-vars* (*e' instanceof U*) ≤ *length xs*›
note *ST* = ‹*P*,*h* ⊢ *stk* [:≤] *ST*›
note *bsok* = ‹*bsok* (*e instanceof U*) *n*›
from *bisim* **have** *pc*: *pc* ≤ *length* (*compE2 e*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e*))
case *True*
with *exec* **have** *exec'*: *?exec e stk loc pc xcp stk' loc' pc' xcp'* **by**(*auto simp add*: *exec-move-InstanceOf*)

from *True* **have** $\tau move2\ (compP2\ P)\ h\ stk\ (e\ instanceof\ U)\ pc\ xcp = \tau move2\ (compP2\ P)\ h\ stk\ e\ pc\ xcp$
by(*simp add*: $\tau move2\text{-}iff$)
moreover have *no-call2 e pc* $\Longrightarrow$ *no-call2* (*e instanceof U*) *pc* **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *IH*[*OF exec'* - - *ST* ‹*conf-xcp'* (*compP2 P*) *h xcp*›] *len bsok*
by(*fastforce intro*: *bisim1-bisims1.bisim1InstanceOf InstanceOf1Red elim*!: *InstanceOf-*τ*red1r-xt InstanceOf-*τ*red1t-xt*)
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compE2 e*) **by** *simp*
with *bisim* **obtain** *v* **where** *stk*: *stk* = [*v*] **and** *xcp*: *xcp* = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim pc len bsok* **have** *red*: $\tau red1r\ P\ t\ h\ (e', xs)\ (Val\ v, loc)$
by(*auto intro*: *bisim1-Val-*τ*red1r simp add*: *bsok-def*)
hence $\tau red1r\ P\ t\ h\ (e'\ instanceof\ U, xs)\ ((Val\ v)\ instanceof\ U, loc)$ **by**(*rule InstanceOf-*τ*red1r-xt*)
also let *?v* = *Bool* ($v \neq Null \wedge P \vdash the\ (typeof_h\ v) \leq U$)
from *exec ST stk xcp* **have** [*simp*]: $h' = h\ ta = \varepsilon\ xcp' = None\ loc' = loc\ stk' = [?v]\ pc' = Suc\ pc$
by(*auto simp add*: *exec-move-def list-all2-Cons1 conf-def compP2-def elim*!: *exec-meth.cases split*: *if-split-asm*)
have *bisim'*: $P,e\ instanceof\ U,h \vdash (Val\ ?v, loc) \leftrightarrow ([?v], loc, length\ (compE2\ (e\ instanceof\ U)), None)$
by(*rule bisim1Val2*) *simp*
from *exec stk xcp ST*
have *red*: $True,P,t \vdash 1\ \langle(Val\ v)\ instanceof\ U,(h, loc)\rangle -\varepsilon\rightarrow \langle Val\ ?v\ ,(h, loc)\rangle$
by(*auto simp add*: *compP2-def exec-move-def exec-meth-instr list-all2-Cons1 conf-def intro*: *Red1InstanceOf*)
have $\tau move1\ P\ h\ ((Val\ v)\ instanceof\ U)$ **by**(*rule* τ*move1InstanceOfRed*)
with *red* **have** $\tau red1t\ P\ t\ h\ ((Val\ v)\ instanceof\ U, loc)\ (Val\ ?v, loc)$ **by**(*auto intro*: τ*red1t-1step*)
also have τ: $\tau move2\ (compP2\ P)\ h\ [v]\ (e\ instanceof\ U)\ pc\ None$ **by**(*simp add*: $\tau move2\text{-}iff$)
moreover have *no-call2* (*e instanceof U*) *pc* **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *exec stk xcp bisim'* **by**(*fastforce*)
qed
next
case (*bisim1InstanceOfThrow e n a xs stk loc pc U*)
note *exec* = ‹*?exec* (*e instanceof U*) *stk loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*›
note *bisim* = ‹$P,e,h \vdash (Throw\ a, xs) \leftrightarrow (stk, loc, pc, \lfloor a \rfloor)$›
from *bisim* **have** *pc*: *pc* < *length* (*compE2 e*) **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim* **have** *match-ex-table* (*compP2 P*) (*cname-of h a*) (*0* + *pc*) (*compxE2 e 0 0*) = *None*
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)
with *exec pc* **have** *False* **by** (*auto elim*!: *exec-meth.cases simp add*: *exec-move-def*)
thus *?case* **..**
next
case (*bisim1Val v n xs*)
from ‹*?exec* (*Val v*) [] *xs 0 None stk' loc' pc' xcp'*›
have *stk'* = [*v*] *loc'* = *xs h'* = *h pc'* = *length* (*compE2* (*Val v*)) *xcp'* = *None*
by(*auto elim*: *exec-meth.cases simp add*: *exec-move-def*)
moreover have $P, Val\ v,h \vdash (Val\ v, xs) \leftrightarrow ([v], xs, length\ (compE2\ (Val\ v)), None)$
by(*rule bisim1Val2*) *simp*
moreover have $\tau move2\ (compP2\ P)\ h\ []\ (Val\ v)\ 0\ None$ **by**(*rule* τ*move2Val*)
ultimately show *?case* **by**(*auto*)
next
case (*bisim1Var V n xs*)
note *exec* = ‹*?exec* (*Var V*) [] *xs 0 None stk' loc' pc' xcp'*›
moreover note *len* = ‹*n* + *max-vars* (*Var V*) ≤ *length xs*›

moreover have *τmove2 (compP2 P) h [] (Var V) 0 None τmove1 P h (Var V)*
by(*auto intro*: *τmove1Var simp add*: *τmove2-iff*)
moreover have *P,Var V,h ⊢ (Val (xs ! V), xs) ↔ ([xs ! V], xs, length (compE2 (Var V)), None)*
by(*rule bisim1Val2*) *simp*
ultimately show *?case* **by**(*fastforce elim*!: *exec-meth.cases intro*: *Red1Var r-into-rtranclp simp add*: *exec-move-def*)
next
case (*bisim1BinOp1 e1 n e1' xs stk loc pc xcp e2 bop*)
note *IH1* = *bisim1BinOp1.IH*(*2*)
note *IH2* = *bisim1BinOp1.IH*(*4*)
note *exec* = ‹*?exec* (*e1* «*bop*» *e2*) *stk loc pc xcp stk' loc' pc' xcp'*›
note *bisim1* = ‹*P,e1,h ⊢* (*e1'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim2* = ‹*P,e2,h ⊢* (*e2*, *loc*) ↔ ([], *loc*, *0*, *None*)›
note *len* = ‹*n* + *max-vars* (*e1'* «*bop*» *e2*) ≤ *length xs*›
note *ST* = ‹*P,h ⊢ stk* [:≤] *ST*›
note *bsok* = ‹*bsok* (*e1* «*bop*» *e2*) *n*›
from *bisim1* **have** *pc*: *pc* ≤ *length* (*compE2 e1*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e1*))
case *True*
with *exec* **have** *exec'*: *?exec e1 stk loc pc xcp stk' loc' pc' xcp'* **by**(*auto simp add*: *exec-move-BinOp1*)
from *True* **have** *τ*: *τmove2* (*compP2 P*) *h stk* (*e1* «*bop*» *e2*) *pc xcp* = *τmove2* (*compP2 P*) *h stk e1 pc xcp*
by(*simp add*: *τmove2-iff*)
with *IH1*[*OF exec' - - ST* ‹*conf-xcp'* (*compP2 P*) *h xcp*›] *bisim2 len bsok* **obtain** *e'' xs''*
where *bisim'*: *P,e1,h' ⊢* (*e''*, *xs''*) ↔ (*stk'*, *loc'*, *pc'*, *xcp'*)
and *red*: *?red e1' xs e'' xs'' e1 stk pc pc' xcp xcp'* **by** *auto*
from *bisim'* **have** *P,e1* «*bop*» *e2,h' ⊢* (*e''* «*bop*» *e2*, *xs''*) ↔ (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*rule bisim1-bisims1.bisim1BinOp1*)
moreover from *True* **have** *no-call2* (*e1* «*bop*» *e2*) *pc* = *no-call2 e1 pc* **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *red τ*
by(*fastforce intro*: *Bin1OpRed1 elim*!: *BinOp-τred1r-xt1 BinOp-τred1t-xt1*)
next
case *False*
with *pc* **have** *pc*: *pc* = *length* (*compE2 e1*) **by** *auto*
with *bisim1* **obtain** *v* **where** *e1'*: *is-val e1'* ⟶ *e1'* = *Val v*
and *stk*: *stk* = [*v*] **and** *xcp*: *xcp* = *None* **and** *call*: *call1 e1'* = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim1 pc len bsok* **have** *rede1'*: *τred1r P t h* (*e1'*, *xs*) (*Val v*, *loc*)
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *rede1''*: *τred1r P t h* (*e1'* «*bop*» *e2*, *xs*) (*Val v* «*bop*» *e2*, *loc*) **by**(*rule BinOp-τred1r-xt1*)
moreover from *pc exec stk xcp*
have *exec-meth-d* (*compP2 P*) (*compE2* (*e1* «*bop*» *e2*)) (*compxE2* (*e1* «*bop*» *e2*) *0 0*) *t h* ([] @ [*v*], *loc*, *length* (*compE2 e1*) + *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *compxE2-size-convs compxE2-stack-xlift-convs exec-move-def*)
hence *exec'*: *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*stack-xlift* (*length* [*v*]) (*compxE2 e2 0 0*)) *t h* ([] @ [*v*], *loc*, *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 e1*), *xcp'*)
and *pc'*: *pc'* ≥ *length* (*compE2 e1*) **by**(*safe dest*!: *BinOp-exec2D*)*simp-all*
then obtain *PC'* **where** *PC'*: *pc'* = *length* (*compE2 e1*) + *PC'*
by −(*rule that*[**where** *PC'35*=*pc'* − *length* (*compE2 e1*)], *simp*)
from *exec' bisim2* **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ [*v*]
and *exec''*: *exec-move-d P t e2 h* ([], *loc*, *0*, *None*) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 e1*), *xcp'*)
by(*unfold exec-move-def*)(*drule* (*1*) *exec-meth-stk-split*, *auto*)

with *pc xcp* **have** τ: τ*move2* (*compP2 P*) *h* [*v*] (*e1* «*bop*» *e2*) (*length* (*compE2 e1*)) *None* = τ*move2* (*compP2 P*) *h* [] *e2 0 None*
using τ*instr-stk-drop-exec-move*[**where** *stk*=[] **and** *vs* = [*v*]]
by(*simp add*: τ*move2-iff* τ*instr-stk-drop-exec-move*)
from *bisim1* **have** *length xs* = *length loc* **by**(*rule bisim1-length-xs*)
with *IH2*[*OF exec''*, *of* []] *len bsok* **obtain** *e'' xs''*
where *bisim'*: *P*,*e2*,*h'* ⊢ (*e''*, *xs''*) ↔ (*stk''*, *loc'*, *pc'* − *length* (*compE2 e1*), *xcp'*)
and *red*: *?red e2 loc e'' xs'' e2* [] *0* (*pc'* − *length* (*compE2 e1*)) *None xcp'* **by** *auto*
from *bisim'*
have *P*,*e1* «*bop*» *e2*,*h'* ⊢ (*Val v* «*bop*» *e''*, *xs''*) ↔ (*stk''* @ [*v*], *loc'*, *length* (*compE2 e1*) + (*pc'* − *length* (*compE2 e1*)), *xcp'*)
by(*rule bisim1-bisims1.bisim1BinOp2*)
moreover from *red* τ
have *?red* (*Val v* «*bop*» *e2*) *loc* (*Val v* «*bop*» *e''*) *xs''* (*e1* «*bop*» *e2*) [*v*] (*length* (*compE2 e1*)) *pc' None xcp'*
by(*fastforce intro*: *Bin1OpRed2 elim*!: *BinOp-*τ*red1r-xt2 BinOp-*τ*red1t-xt2 simp add*: *no-call2-def*)
moreover have *no-call2* (*e1* «*bop*» *e2*) (*length* (*compE2 e1*)) **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** τ *stk' pc xcp pc' PC' bisim1 bisim2 e1' stk call* **by**(*fastforce elim*!: *rtranclp-trans*)
qed
next
case (*bisim1BinOp2 e2 n e2' xs stk loc pc xcp e1 bop v1*)
note *IH2* = *bisim1BinOp2.IH*(*2*)
note *exec* = ‹*?exec* (*e1* «*bop*» *e2*) (*stk* @ [*v1*]) *loc* (*length* (*compE2 e1*) + *pc*) *xcp stk' loc' pc' xcp'*›
note *bisim1* = ‹*P*,*e1*,*h* ⊢ (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e2'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *len* = ‹*n* + *max-vars* (*Val v1* «*bop*» *e2'*) ≤ *length xs*›
note *bsok* = ‹*bsok* (*e1* «*bop*» *e2*) *n*›
note *ST* = ‹*P*,*h* ⊢ *stk* @ [*v1*] [:≤] *ST*›
then obtain *ST2 T* **where** *ST* = *ST2* @ [*T*] *P*,*h* ⊢ *stk* [:≤] *ST2 P*,*h* ⊢ *v1* :≤ *T*
by(*auto simp add*: *list-all2-append1 length-Suc-conv*)
from *bisim2* **have** *pc*: *pc* ≤ *length* (*compE2 e2*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e2*))
case *True*
with *exec* **have** *exec'*: *exec-meth-d* (*compP2 P*) (*compE2 e2*) (*stack-xlift* (*length* [*v1*]) (*compxE2 e2 0 0*)) *t h* (*stk* @ [*v1*], *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 e1*), *xcp'*)
and *pc'*: *pc'* ≥ *length* (*compE2 e1*)
by(*unfold exec-move-def*)(*safe dest*!: *BinOp-exec2D*)
from *exec' bisim2* **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ [*v1*]
and *exec''*: *exec-move-d P t e2 h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 e1*), *xcp'*)
by −(*drule* (*1*) *exec-meth-stk-split*, *auto simp add*: *exec-move-def*)
with *True* **have** τ: τ*move2* (*compP2 P*) *h* (*stk* @ [*v1*]) (*e1* «*bop*» *e2*) (*length* (*compE2 e1*) + *pc*) *xcp* = τ*move2* (*compP2 P*) *h stk e2 pc xcp*
by(*auto simp add*: τ*move2-iff* τ*instr-stk-drop-exec-move*)
from *IH2*[*OF exec'' - -* ‹*P*,*h* ⊢ *stk* [:≤] *ST2*› ‹*conf-xcp'* (*compP2 P*) *h xcp*›] *len bsok* **obtain** *e'' xs''*
where *bisim'*: *P*,*e2*,*h'* ⊢ (*e''*, *xs''*) ↔ (*stk''*, *loc'*, *pc'* − *length* (*compE2 e1*), *xcp'*)
and *red*: *?red e2' xs e'' xs'' e2 stk pc* (*pc'* − *length* (*compE2 e1*)) *xcp xcp'* **by** *auto*
from *bisim'* **have** *P*,*e1* «*bop*» *e2*,*h'* ⊢ (*Val v1* «*bop*» *e''*, *xs''*) ↔ (*stk''* @ [*v1*], *loc'*, *length* (*compE2 e1*) + (*pc'* − *length* (*compE2 e1*)), *xcp'*)
by(*rule bisim1-bisims1.bisim1BinOp2*)
with *red* τ *stk' pc' True* **show** *?thesis*

by(*fastforce intro*: *Bin1OpRed2 elim*!: *BinOp-τred1r-xt2 BinOp-τred1t-xt2 split*: *if-split-asm simp add*: *no-call2-def*)
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compE2 e2*) **by** *simp*
with *bisim2* **obtain** *v2* **where** *e2'*: *is-val e2'* ⟶ *e2'* = *Val v2*
and *stk*: *stk* = [*v2*] **and** *xcp*: *xcp* = *None* **and** *call*: *call1 e2'* = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim2 pc len bsok* **have** *red*: *τred1r P t h* (*e2'*, *xs*) (*Val v2*, *loc*)
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *red1*: *τred1r P t h* (*Val v1* «*bop*» *e2'*, *xs*) (*Val v1* «*bop*» *Val v2*, *loc*) **by**(*rule BinOp-τred1r-xt2*)
show *?thesis*
proof(*cases the* (*binop bop v1 v2*))
case (*Inl v*)
note *red1*
also from *exec xcp ST stk Inl*
have *τred1r P t h* (*Val v1* «*bop*» *Val v2*, *loc*) (*Val v*, *loc*)
by(*force simp add*: *exec-move-def exec-meth-instr list-all2-Cons1 conf-def compP2-def dest*: *binop-progress intro*: *r-into-rtranclp Red1BinOp τmove1BinOp*)
also have *τ*: *τmove2* (*compP2 P*) *h* [*v2*, *v1*] (*e1* «*bop*» *e2*) (*length* (*compE2 e1*) + *length* (*compE2 e2*)) *None*
by(*simp add*: *τmove2-iff*)
moreover have *P*,*e1* «*bop*» *e2*,*h* ⊢ (*Val v*, *loc*) ↔ ([*v*], *loc*, *length* (*compE2* (*e1* «*bop*» *e2*)), *None*)
by(*rule bisim1Val2*) *simp*
ultimately show *?thesis* **using** *exec xcp stk call Inl* **by**(*auto simp add*: *exec-move-def exec-meth-instr*)
next
case (*Inr a*)
note *red1*
also from *exec xcp ST stk Inr*
have *τred1r P t h* (*Val v1* «*bop*» *Val v2*, *loc*) (*Throw a*, *loc*)
by(*force simp add*: *exec-move-def exec-meth-instr list-all2-Cons1 conf-def compP2-def dest*: *binop-progress intro*: *r-into-rtranclp Red1BinOpFail τmove1BinOp*)
also have *τ*: *τmove2* (*compP2 P*) *h* [*v2*, *v1*] (*e1* «*bop*» *e2*) (*length* (*compE2 e1*) + *length* (*compE2 e2*)) *None*
by(*simp add*: *τmove2-iff*)
moreover
have *P*,*e1* «*bop*» *e2*,*h* ⊢ (*Throw a*, *loc*) ↔ ([*v2*, *v1*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*), ⌊*a*⌋)
by(*rule bisim1BinOpThrow*)
ultimately show *?thesis* **using** *exec xcp stk call Inr* **by**(*auto simp add*: *exec-move-def exec-meth-instr*)
qed
qed
next
case (*bisim1BinOpThrow1 e1 n a xs stk loc pc e2 bop*)
note *exec* = ‹*?exec* (*e1* «*bop*» *e2*) *stk loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*›
note *bisim1* = ‹*P*,*e1*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
from *bisim1* **have** *pc*: *pc* < *length* (*compE2 e1*) **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim1* **have** *match-ex-table* (*compP2 P*) (*cname-of h a*) (*0* + *pc*) (*compxE2 e1 0 0*) = *None*
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)
with *exec pc* **have** *False* **by**(*auto elim*!: *exec-meth.cases simp add*: *match-ex-table-not-pcs-None exec-move-def*)
thus *?case* **..**
next

case (*bisim1BinOpThrow2 e2 n a xs stk loc pc e1 bop v1*)
note *exec* = ‹*?exec* (*e1* «*bop*» *e2*) (*stk* @ [*v1*]) *loc* (*length* (*compE2 e1*) + *pc*) ⌊*a*⌋ *stk'* *loc'* *pc'* *xcp'*›
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
hence *match-ex-table* (*compP2 P*) (*cname-of h a*) (*length* (*compE2 e1*) + *pc*) (*compxE2 e2* (*length* (*compE2 e1*)) *0*) = *None*
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)
with *exec* **have** *False*
apply(*auto elim*!: *exec-meth.cases simp add*: *match-ex-table-append-not-pcs exec-move-def*)
apply(*auto simp only*: *compxE2-size-convs compxE2-stack-xlift-convs match-ex-table-stack-xlift-eq-Some-conv*)
done
thus *?case* **..**
next
case (*bisim1BinOpThrow e1 n e2 bop a xs v1 v2*)
note ‹*?exec* (*e1* «*bop*» *e2*) [*v1*, *v2*] *xs* (*length* (*compE2 e1*) + *length* (*compE2 e2*)) ⌊*a*⌋ *stk'* *loc'* *pc'* *xcp'*›
hence *False*
by(*auto elim*!: *exec-meth.cases simp add*: *match-ex-table-append-not-pcs compxE2-size-convs exec-move-def*
dest!: *match-ex-table-shift-pcD match-ex-table-pc-length-compE2*)
thus *?case* **..**
next
case (*bisim1LAss1 e n e' xs stk loc pc xcp V*)
note *IH* = *bisim1LAss1.IH*(*2*)
note *exec* = ‹*?exec* (*V* := *e*) *stk loc pc xcp stk'* *loc'* *pc'* *xcp'*›
note *bisim* = ‹*P*,*e*,*h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *len* = ‹*n* + *max-vars* (*V* := *e'*) ≤ *length xs*›
note *bsok* = ‹*bsok* (*V*:=*e*) *n*›
from *bisim* **have** *pc*: *pc* ≤ *length* (*compE2 e*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e*))
case *True*
with *exec* **have** *exec'*: *?exec e stk loc pc xcp stk'* *loc'* *pc'* *xcp'* **by**(*auto simp add*: *exec-move-LAss*)
from *True* **have** *τmove2* (*compP2 P*) *h stk* (*V* := *e*) *pc xcp* = *τmove2* (*compP2 P*) *h stk e pc xcp*
by(*simp add*: *τmove2-iff*)
with *IH*[*OF exec'* - - ‹*P*,*h* ⊢ *stk* [:≤] *ST*› ‹*conf-xcp'* (*compP2 P*) *h xcp*›] *len bsok* **show** *?thesis*
by(*fastforce intro*: *bisim1-bisims1.bisim1LAss1 LAss1Red elim*!: *LAss-τred1r LAss-τred1t simp add*: *no-call2-def*)
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compE2 e*) **by** *simp*
with *bisim* **obtain** *v* **where** *stk*: *stk* = [*v*] **and** *xcp*: *xcp* = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim pc len bsok* **have** *red*: *τred1r P t h* (*e'*, *xs*) (*Val v*, *loc*)
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *τred1r P t h* (*V* := *e'*, *xs*) (*V* := *Val v*, *loc*) **by**(*rule LAss-τred1r*)
also have *τmove1 P h* (*V* := *Val v*) **by**(*rule τmove1LAssRed*)
with *exec stk xcp* **have** *τred1r P t h* (*V* := *Val v*, *loc*) (*unit*, *loc*[*V* := *v*])
by(*auto intro*!: *r-into-rtranclp Red1LAss simp add*: *exec-move-def elim*!: *exec-meth.cases*)
also have *τ*: *τmove2* (*compP2 P*) *h* [*v*] (*V* := *e*) *pc None* **by**(*simp add*: *τmove2-iff*)
moreover have *P*,(*V* := *e*),*h* ⊢ (*unit*, *loc*[*V* := *v*]) ↔ ([], *loc*[*V* := *v*], *Suc* (*length* (*compE2 e*)), *None*)
by(*rule bisim1LAss2*)
ultimately show *?thesis* **using** *exec stk xcp*
by(*fastforce elim*!: *exec-meth.cases simp add*: *exec-move-def*)
qed

next
case (*bisim1LAss2 e n V xs*)
note *bisim* = ‹*P*,*e*,*h* ⊢ (*e*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *exec* = ‹*?exec* (*V* := *e*) [] *xs* (*Suc* (*length* (*compE2 e*))) *None stk′ loc′ pc′ xcp′*›
hence *stk′* = [*Unit*] *loc′* = *xs pc′* = *length* (*compE2* (*V* := *e*)) *xcp′* = *None h′* = *h*
by(*auto elim*!: *exec-meth.cases simp add*: *exec-move-def*)
moreover have *τmove2* (*compP2 P*) *h* [] (*V* := *e*) (*Suc* (*length* (*compE2 e*))) *None* **by**(*simp add*: *τmove2-iff*)
moreover have *P*,*V*:=*e*,*h′* ⊢ (*unit*, *xs*) ↔ ([*Unit*], *xs*, *length* (*compE2* (*V* := *e*)), *None*)
by(*rule bisim1Val2*) *simp*
ultimately show *?case* **by**(*auto*)
next
case (*bisim1LAssThrow e n a xs stk loc pc V*)
note *exec* = ‹*?exec* (*V* := *e*) *stk loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
note *bisim* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
from *bisim* **have** *pc*: *pc* < *length* (*compE2 e*) **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim* **have** *match-ex-table* (*compP2 P*) (*cname-of h a*) (*0* + *pc*) (*compxE2 e 0 0*) = *None*
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)
with *exec pc* **have** *False* **by** (*auto elim*!: *exec-meth.cases simp add*: *exec-move-def*)
thus *?case* **..**
next
case (*bisim1AAcc1 a n a′ xs stk loc pc xcp i*)
note *IH1* = *bisim1AAcc1.IH*(*2*)
note *IH2* = *bisim1AAcc1.IH*(*4*)
note *exec* = ‹*?exec* (*a*⌊*i*⌉) *stk loc pc xcp stk′ loc′ pc′ xcp′*›
note *bisim1* = ‹*P*,*a*,*h* ⊢ (*a′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim2* = ‹*P*,*i*,*h* ⊢ (*i*, *loc*) ↔ ([], *loc*, *0*, *None*)›
note *len* = ‹*n* + *max-vars* (*a′*⌊*i*⌉) ≤ *length xs*›
note *bsok* = ‹*bsok* (*a*⌊*i*⌉) *n*›
from *bisim1* **have** *pc*: *pc* ≤ *length* (*compE2 a*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 a*))
case *True*
with *exec* **have** *exec′*: *?exec a stk loc pc xcp stk′ loc′ pc′ xcp′* **by**(*auto simp add*: *exec-move-AAcc1*)
from *True* **have** *τ*: *τmove2* (*compP2 P*) *h stk* (*a*⌊*i*⌉) *pc xcp* = *τmove2* (*compP2 P*) *h stk a pc xcp* **by**(*auto intro*: *τmove2AAcc1*)
with *IH1*[*OF exec′ - -* ‹*P*,*h* ⊢ *stk* [:≤] *ST*› ‹*conf-xcp′* (*compP2 P*) *h xcp*›] *len bsok* **obtain** *e″ xs″*
where *bisim′*: *P*,*a*,*h′* ⊢ (*e″*, *xs″*) ↔ (*stk′*, *loc′*, *pc′*, *xcp′*)
and *red*: *?red a′ xs e″ xs″ a stk pc pc′ xcp xcp′* **by** *auto*
from *bisim′* **have** *P*,*a*⌊*i*⌉,*h′* ⊢ (*e″*⌊*i*⌉, *xs″*) ↔ (*stk′*, *loc′*, *pc′*, *xcp′*) **by**(*rule bisim1-bisims1.bisim1AAcc1*)
moreover from *True* **have** *no-call2* (*a*⌊*i*⌉) *pc* = *no-call2 a pc* **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *red τ* **by**(*fastforce intro*: *AAcc1Red1 elim*!: *AAcc-τred1r-xt1 AAcc-τred1t-xt1*)
next
case *False*
with *pc* **have** *pc*: *pc* = *length* (*compE2 a*) **by** *auto*
with *bisim1* **obtain** *v* **where** *a′*: *is-val a′* ⟶ *a′* = *Val v*
and *stk*: *stk* = [*v*] **and** *xcp*: *xcp* = *None* **and** *call*: *call1 a′* = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim1 pc len bsok* **have** *rede1′*: *τred1r P t h* (*a′*, *xs*) (*Val v*, *loc*)
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *τred1r P t h* (*a′*⌊*i*⌉, *xs*) (*Val v*⌊*i*⌉, *loc*) **by**(*rule AAcc-τred1r-xt1*)
moreover from *pc exec stk xcp*
have *exec′*: *exec-meth-d* (*compP2 P*) (*compE2 a* @ *compE2 i* @ [*ALoad*]) (*compxE2 a 0 0* @ *shift*

(*length* (*compE2 a*)) (*stack-xlift* (*length* [*v*]) (*compxE2 i 0 0*))) *t h* ([] @ [*v*], *loc*, *length* (*compE2 a*) + *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *compxE2-size-convs compxE2-stack-xlift-convs exec-move-def*)
hence *exec-meth-d* (*compP2 P*) (*compE2 i* @ [*ALoad*]) (*stack-xlift* (*length* [*v*]) (*compxE2 i 0 0*)) *t h* ([] @ [*v*], *loc*, *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 a*), *xcp'*)
by(*rule exec-meth-drop-xt*) *auto*
hence *exec-meth-d* (*compP2 P*) (*compE2 i*) (*stack-xlift* (*length* [*v*]) (*compxE2 i 0 0*)) *t h* ([] @ [*v*], *loc*, *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 a*), *xcp'*)
by(*rule exec-meth-take*) *simp*
with *bisim2* **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ [*v*]
and *exec''*: *exec-move-d P t i h* ([], *loc*, *0*, *None*) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 a*), *xcp'*)
unfolding *exec-move-def* **by**(*blast dest*: *exec-meth-stk-split*)
with *pc xcp* **have** *τ*: *τmove2* (*compP2 P*) *h* ([] @ [*v*]) (*a*⌊*i*⌉) (*length* (*compE2 a*)) *None* = *τmove2* (*compP2 P*) *h* [] *i 0 None*
using *τinstr-stk-drop-exec-move*[**where** *stk*=[] **and** *vs*=[*v*]]
by(*auto simp add*: *τmove2-iff τinstr-stk-drop-exec-move*)
from *bisim1* **have** *length xs* = *length loc* **by**(*rule bisim1-length-xs*)
with *IH2*[*OF exec''*] *len bsok* **obtain** *e'' xs''*
where *bisim'*: *P*,*i*,*h'* ⊢ (*e''*, *xs''*) ↔ (*stk''*, *loc'*, *pc'* − *length* (*compE2 a*), *xcp'*)
and *red*: *?red i loc e'' xs'' i* [] *0* (*pc'* − *length* (*compE2 a*)) *None xcp'* **by**(*fastforce*)
from *bisim'*
have *P*,*a*⌊*i*⌉,*h'* ⊢ (*Val v*⌊*e'*⌉, *xs''*) ↔ (*stk''* @ [*v*], *loc'*, *length* (*compE2 a*) + (*pc'* − *length* (*compE2 a*)), *xcp'*)
by(*rule bisim1-bisims1.bisim1AAcc2*)
moreover from *red τ* **have** *?red* (*Val v*⌊*i*⌉) *loc* (*Val v*⌊*e'*⌉) *xs''* (*a*⌊*i*⌉) [*v*] (*length* (*compE2 a*)) *pc' None xcp'*
by(*fastforce intro*: *AAcc1Red2 elim*!: *AAcc-τred1r-xt2 AAcc-τred1t-xt2 split*: *if-split-asm simp add*: *no-call2-def*)
moreover from *exec'* **have** *pc'* ≥ *length* (*compE2 a*)
by(*rule exec-meth-drop-xt-pc*) *auto*
moreover have *no-call2* (*a*⌊*i*⌉) *pc* **using** *pc* **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *τ stk' pc xcp stk call*
by(*fastforce elim*!: *rtranclp-trans*)+
qed
next
case (*bisim1AAcc2 i n i' xs stk loc pc xcp a v*)
note *IH2* = *bisim1AAcc2.IH*(*2*)
note *exec* = ‹*?exec* (*a*⌊*i*⌉) (*stk* @ [*v*]) *loc* (*length* (*compE2 a*) + *pc*) *xcp stk' loc' pc' xcp'*›
note *bisim2* = ‹*P*,*i*,*h* ⊢ (*i'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *len* = ‹*n* + *max-vars* (*Val v*⌊*i'*⌉) ≤ *length xs*›
note *bsok* = ‹*bsok* (*a*⌊*i*⌉) *n*›
from *bisim2* **have** *pc*: *pc* ≤ *length* (*compE2 i*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 i*))
case *True*
from *exec* **have** *exec'*: *exec-meth-d* (*compP2 P*) (*compE2 a* @ *compE2 i* @ [*ALoad*]) (*compxE2 a 0 0* @ *shift* (*length* (*compE2 a*)) (*stack-xlift* (*length* [*v*]) (*compxE2 i 0 0*))) *t h* (*stk* @ [*v*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *shift-compxE2 stack-xlift-compxE2 exec-move-def*)
hence *exec-meth-d* (*compP2 P*) (*compE2 i* @ [*ALoad*]) (*stack-xlift* (*length* [*v*]) (*compxE2 i 0 0*)) *t h* (*stk* @ [*v*], *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 a*), *xcp'*)
by(*rule exec-meth-drop-xt*) *auto*
hence *exec-meth-d* (*compP2 P*) (*compE2 i*) (*stack-xlift* (*length* [*v*]) (*compxE2 i 0 0*)) *t h* (*stk* @ [*v*], *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 a*), *xcp'*)

using *True* **by**(*rule exec-meth-take*)
with *bisim2* **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ [*v*]
and *exec''*: *exec-move-d P t i h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 a*), *xcp'*)
unfolding *exec-move-def* **by**(*blast dest*: *exec-meth-stk-split*)
with *True* **have** *τ*: *τmove2* (*compP2 P*) *h* (*stk* @ [*v*]) (*a*⌊*i*⌉) (*length* (*compE2 a*) + *pc*) *xcp* = *τmove2* (*compP2 P*) *h stk i pc xcp*
by(*auto simp add*: *τmove2-iff τinstr-stk-drop-exec-move*)
moreover from ‹*P*,*h* ⊢ *stk* @ [*v*] [:≤] *ST*› **obtain** *ST2*
where *P*,*h* ⊢ *stk* [:≤] *ST2* **by**(*auto simp add*: *list-all2-append1*)
from *IH2*[*OF exec''* - - *this* ‹*conf-xcp'* (*compP2 P*) *h xcp*›] *len bsok* **obtain** *e'' xs''*
where *bisim'*: *P*,*i*,*h'* ⊢ (*e''*, *xs''*) ↔ (*stk''*, *loc'*, *pc'* − *length* (*compE2 a*), *xcp'*)
and *red*: *?red i' xs e'' xs'' i stk pc* (*pc'* − *length* (*compE2 a*)) *xcp xcp'* **by** *auto*
from *bisim'* **have** *P*,*a*⌊*i*⌉,*h'* ⊢ (*Val v*⌊*e''*⌉, *xs''*) ↔ (*stk''* @ [*v*], *loc'*, *length* (*compE2 a*) + (*pc'* − *length* (*compE2 a*)), *xcp'*)
by(*rule bisim1-bisims1.bisim1AAcc2*)
moreover from *exec'* **have** *pc'* ≥ *length* (*compE2 a*)
by(*rule exec-meth-drop-xt-pc*) *auto*
moreover have *no-call2 i pc* ⟹ *no-call2* (*a*⌊*i*⌉) (*length* (*compE2 a*) + *pc*)
by(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *red τ stk' True*
by(*fastforce intro*: *AAcc1Red2 elim*!: *AAcc-τred1r-xt2 AAcc-τred1t-xt2 split*: *if-split-asm*)
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compE2 i*) **by** *simp*
with *bisim2* **obtain** *v2* **where** *i'*: *is-val i'* ⟶ *i'* = *Val v2*
and *stk*: *stk* = [*v2*] **and** *xcp*: *xcp* = *None* **and** *call*: *call1 i'* = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim2 pc len bsok* **have** *red*: *τred1r P t h* (*i'*, *xs*) (*Val v2*, *loc*)
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *τred1r P t h* (*Val v*⌊*i'*⌉, *xs*) (*Val v*⌊*Val v2*⌉, *loc*) **by**(*rule AAcc-τred1r-xt2*)
moreover have *τ*: ¬ *τmove2* (*compP2 P*) *h* [*v2*, *v*] (*a*⌊*i*⌉) (*length* (*compE2 a*) + *length* (*compE2 i*)) *None*
by(*simp add*: *τmove2-iff*)
moreover
have ∃ *ta' e''*. *P*,*a*⌊*i*⌉,*h'* ⊢ (*e''*,*loc*) ↔ (*stk'*, *loc'*, *pc'*, *xcp'*) ∧ *True*,*P*,*t* ⊢*1* ⟨*Val v*⌊*Val v2*⌉, (*h*, *loc*)⟩ −*ta'*→ ⟨*e''*,(*h'*, *loc*)⟩ ∧ *ta-bisim wbisim1* (*extTA2J1 P ta'*) *ta*
proof(*cases v* = *Null*)
case *True* **with** *exec stk xcp* **show** *?thesis*
by(*fastforce elim*!: *exec-meth.cases simp add*: *exec-move-def intro*: *bisim1AAccFail Red1AAccNull*)
next
case *False*
with *exec xcp stk* **obtain** *U el A len I* **where** [*simp*]: *v* = *Addr A* **and** *hA*: *typeof-addr h A* = ⌊*Array-type U len*⌋
and [*simp*]: *v2* = *Intg I* **by**(*auto simp add*: *exec-move-def exec-meth-instr is-Ref-def conf-def split*: *if-split-asm*)
show *?thesis*
proof(*cases 0* <=*s I* ∧ *sint I* < *int len*)
case *True*
hence ¬ *I* <*s 0* **by** *auto*
moreover
with *exec xcp stk True hA* **obtain** *v3* **where** *stk'* = [*v3*] *heap-read h A* (*ACell* (*nat* (*sint I*))) *v3*
by(*auto simp add*: *exec-move-def exec-meth-instr is-Ref-def*)
moreover

```
      have P,a⌊i⌉,h' ⊢ (Val v3,loc) ↔ ([v3], loc, length (compE2 (a⌊i⌉)), None)
        by(rule bisim1Val2) simp
      ultimately show ?thesis using exec stk xcp True hA
        by(fastforce elim!: exec-meth.cases intro: Red1AAcc simp add: exec-move-def ta-upd-simps
ta-bisim-def split: if-split-asm)
    next
      case False
      with exec stk xcp hA show ?thesis
        by(fastforce elim!: exec-meth.cases simp add: is-Ref-def exec-move-def intro: bisim1AAccFail
Red1AAccBounds split: if-split-asm)
    qed
  qed
  ultimately show ?thesis using exec xcp stk call by fastforce
 qed
next
 case (bisim1AAccThrow1 A n a xs stk loc pc i)
 note exec = ‹?exec (A⌊i⌉) stk loc pc ⌊a⌋ stk' loc' pc' xcp'›
 note bisim1 = ‹P,A,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
 from bisim1 have pc: pc < length (compE2 A) by(auto dest: bisim1-ThrowD)
 from bisim1 have match-ex-table (compP2 P) (cname-of h a) (0 + pc) (compxE2 A 0 0) = None
  unfolding compP2-def by(rule bisim1-xcp-Some-not-caught)
 with exec pc have False
  by(auto elim!: exec-meth.cases simp add: match-ex-table-not-pcs-None exec-move-def)
 thus ?case ..
next
 case (bisim1AAccThrow2 i n a xs stk loc pc A v)
 note exec = ‹?exec (A⌊i⌉) (stk @ [v]) loc (length (compE2 A) + pc) ⌊a⌋ stk' loc' pc' xcp'›
 note bisim2 = ‹P,i,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
 hence match-ex-table (compP2 P) (cname-of h a) (length (compE2 A) + pc) (compxE2 i (length
(compE2 A)) 0) = None
  unfolding compP2-def by(rule bisim1-xcp-Some-not-caught)
 with exec have False
  apply(auto elim!: exec-meth.cases simp add: match-ex-table-append-not-pcs exec-move-def)
 apply(auto simp only: compxE2-size-convs compxE2-stack-xlift-convs match-ex-table-stack-xlift-eq-Some-conv)
  done
 thus ?case ..
next
 case (bisim1AAccFail a n i ad xs v v')
 note ‹?exec (a⌊i⌉) [v, v'] xs (length (compE2 a) + length (compE2 i)) ⌊ad⌋ stk' loc' pc' xcp'›
 hence False
  by(auto elim!: exec-meth.cases simp add: match-ex-table-append-not-pcs compxE2-size-convs exec-move-def
           dest!: match-ex-table-shift-pcD match-ex-table-pc-length-compE2)
 thus ?case ..
next
 case (bisim1AAss1 a n a' xs stk loc pc xcp i e)
 note IH1 = bisim1AAss1.IH(2)
 note IH2 = bisim1AAss1.IH(4)
 note exec = ‹?exec (a⌊i⌉ := e) stk loc pc xcp stk' loc' pc' xcp'›
 note bisim1 = ‹P,a,h ⊢ (a', xs) ↔ (stk, loc, pc, xcp)›
 note bisim2 = ‹P,i,h ⊢ (i, loc) ↔ ([], loc, 0, None)›
 note len = ‹n + max-vars (a'⌊i⌉ := e) ≤ length xs›
 note bsok = ‹bsok (a⌊i⌉ := e) n›
 from bisim1 have pc: pc ≤ length (compE2 a) by(rule bisim1-pc-length-compE2)
 show ?case
```

proof(*cases pc* < *length* (*compE2 a*))
case *True*
with *exec* **have** *exec'*: *?exec a stk loc pc xcp stk' loc' pc' xcp'* **by**(*simp add*: *exec-move-AAss1*)
from *True* **have** τ: *τmove2* (*compP2 P*) *h stk* (*a*⌊*i*⌋ := *e*) *pc xcp* = *τmove2* (*compP2 P*) *h stk a pc xcp* **by**(*simp add*: *τmove2-iff*)
with *IH1*[*OF exec'* - - ‹*P*,*h* ⊢ *stk* [:≤] *ST*› ‹*conf-xcp'* (*compP2 P*) *h xcp*›] *len bsok* **obtain** *e'' xs''*
where *bisim'*: *P*,*a*,*h'* ⊢ (*e''*, *xs''*) ↔ (*stk'*, *loc'*, *pc'*, *xcp'*)
and *red*: *?red a' xs e'' xs'' a stk pc pc' xcp xcp'* **by** *auto*
from *bisim'* **have** *P*,*a*⌊*i*⌋ := *e*,*h'* ⊢ (*e''*⌊*i*⌋ := *e*, *xs''*) ↔ (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*rule bisim1-bisims1.bisim1AAss1*)
moreover from *True* **have** *no-call2* (*a*⌊*i*⌋ := *e*) *pc* = *no-call2 a pc* **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *red τ* **by**(*fastforce intro*: *AAss1Red1 elim*!: *AAss-τred1r-xt1 AAss-τred1t-xt1*)
next
case *False*
with *pc* **have** *pc*: *pc* = *length* (*compE2 a*) **by** *auto*
with *bisim1* **obtain** *v* **where** *a'*: *is-val a'* ⟶ *a'* = *Val v*
and *stk*: *stk* = [*v*] **and** *xcp*: *xcp* = *None* **and** *call*: *call1 a'* = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim1 pc len bsok* **have** *rede1'*: *τred1r P t h* (*a'*, *xs*) (*Val v*, *loc*)
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *τred1r P t h* (*a'*⌊*i*⌋ := *e*, *xs*) (*Val v*⌊*i*⌋ := *e*, *loc*) **by**(*rule AAss-τred1r-xt1*)
moreover from *pc exec stk xcp*
have *exec'*: *exec-meth-d* (*compP2 P*) (*compE2 a* @ *compE2 e* @ [*AStore*, *Push Unit*]) (*compxE2 a 0 0* @ *shift* (*length* (*compE2 a*)) (*stack-xlift* (*length* [*v*]) (*compxE2 i 0 0*) @ *shift* (*length* (*compE2 i*)) (*compxE2 e 0* (*Suc* (*Suc 0*))))) *t h* ([] @ [*v*], *loc*, *length* (*compE2 a*) + *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *compxE2-size-convs compxE2-stack-xlift-convs exec-move-def*)
hence *exec-meth-d* (*compP2 P*) (*compE2 i* @ *compE2 e* @ [*AStore*, *Push Unit*]) (*stack-xlift* (*length* [*v*]) (*compxE2 i 0 0*) @ *shift* (*length* (*compE2 i*)) (*compxE2 e 0* (*Suc* (*Suc 0*)))) *t h* ([] @ [*v*], *loc*, *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 a*), *xcp'*)
by(*rule exec-meth-drop-xt*) *auto*
hence *exec-meth-d* (*compP2 P*) (*compE2 i*) (*stack-xlift* (*length* [*v*]) (*compxE2 i 0 0*)) *t h* ([] @ [*v*], *loc*, *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 a*), *xcp'*)
by(*rule exec-meth-take-xt*) *simp*
with *bisim2* **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ [*v*]
and *exec''*: *exec-move-d P t i h* ([], *loc*, *0*, *None*) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 a*), *xcp'*)
unfolding *exec-move-def* **by**(*blast dest*: *exec-meth-stk-split*)
with *pc xcp* **have** τ: *τmove2* (*compP2 P*) *h* [*v*] (*a*⌊*i*⌋:= *e*) (*length* (*compE2 a*)) *None* = *τmove2* (*compP2 P*) *h* [] *i 0 None*
using *τinstr-stk-drop-exec-move*[**where** *stk*=[] **and** *vs*=[*v*]]
by(*auto simp add*: *τmove2-iff*)
from *bisim1* **have** *length xs* = *length loc* **by**(*rule bisim1-length-xs*)
with *IH2*[*OF exec''*] *len bsok* **obtain** *e'' xs''*
where *bisim'*: *P*,*i*,*h'* ⊢ (*e''*, *xs''*) ↔ (*stk''*, *loc'*, *pc'* − *length* (*compE2 a*), *xcp'*)
and *red*: *?red i loc e'' xs'' i* [] *0* (*pc'* − *length* (*compE2 a*)) *None xcp'* **by** *fastforce*
from *bisim'*
have *P*,*a*⌊*i*⌋ := *e*,*h'* ⊢ (*Val v*⌊*e''*⌋ := *e*, *xs''*) ↔ (*stk''* @ [*v*], *loc'*, *length* (*compE2 a*) + (*pc'* − *length* (*compE2 a*)), *xcp'*)
by(*rule bisim1-bisims1.bisim1AAss2*)
moreover from *red τ* **have** *?red* (*Val v*⌊*i*⌋ := *e*) *loc* (*Val v*⌊*e''*⌋ := *e*) *xs''* (*a*⌊*i*⌋ := *e*) [*v*] (*length* (*compE2 a*)) *pc' None xcp'*
by(*fastforce intro*: *AAss1Red2 elim*!: *AAss-τred1r-xt2 AAss-τred1t-xt2 split*: *if-split-asm simp add*: *no-call2-def*)

moreover from *exec′* **have** *pc′ ≥ length (compE2 a)*
by(*rule exec-meth-drop-xt-pc*) *auto*
moreover have *no-call2 (a⌊i⌉ := e) pc* **using** *pc* **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *τ stk′ pc xcp stk* **by**(*fastforce elim*!: *rtranclp-trans*)
qed
next
case (*bisim1AAss2 i n i′ xs stk loc pc xcp a e v*)
note *IH2* = *bisim1AAss2.IH*(*2*)
note *IH3* = *bisim1AAss2.IH*(*6*)
note *exec* = ‹*?exec (a⌊i⌉ := e) (stk @ [v]) loc (length (compE2 a) + pc) xcp stk′ loc′ pc′ xcp′*›
note *bisim2* = ‹*P,i,h ⊢ (i′, xs) ↔ (stk, loc, pc, xcp)*›
note *bisim3* = ‹*P,e,h ⊢ (e, loc) ↔ ([], loc, 0, None)*›
note *len* = ‹*n + max-vars (Val v⌊i⌉ := e) ≤ length xs*›
note *bsok* = ‹*bsok (a⌊i⌉ := e) n*›
from *bisim2* **have** *pc*: *pc ≤ length (compE2 i)* **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc < length (compE2 i)*)
case *True*
from *exec* **have** *exec′*: *exec-meth-d (compP2 P) (compE2 a @ compE2 i @ compE2 e @ [AStore, Push Unit]) (compxE2 a 0 0 @ shift (length (compE2 a)) (stack-xlift (length [v]) (compxE2 i 0 0) @ shift (length (compE2 i)) (compxE2 e 0 (Suc (Suc 0))))) t h (stk @ [v], loc, length (compE2 a) + pc, xcp) ta h′ (stk′, loc′, pc′, xcp′)*
by(*simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps exec-move-def*)
hence *exec-meth-d (compP2 P) (compE2 i @ compE2 e @ [AStore, Push Unit]) (stack-xlift (length [v]) (compxE2 i 0 0) @ shift (length (compE2 i)) (compxE2 e 0 (Suc (Suc 0)))) t h (stk @ [v], loc, pc, xcp) ta h′ (stk′, loc′, pc′ − length (compE2 a), xcp′)*
by(*rule exec-meth-drop-xt*) *auto*
hence *exec-meth-d (compP2 P) (compE2 i) (stack-xlift (length [v]) (compxE2 i 0 0)) t h (stk @ [v], loc, pc, xcp) ta h′ (stk′, loc′, pc′ − length (compE2 a), xcp′)*
using *True* **by**(*rule exec-meth-take-xt*)
with *bisim2* **obtain** *stk″* **where** *stk′*: *stk′ = stk″ @ [v]*
and *exec″*: *exec-move-d P t i h (stk, loc, pc, xcp) ta h′ (stk″, loc′, pc′ − length (compE2 a), xcp′)*
unfolding *exec-move-def* **by**(*blast dest*: *exec-meth-stk-split*)
with *True* **have** *τ*: *τmove2 (compP2 P) h (stk @ [v]) (a⌊i⌉ := e) (length (compE2 a) + pc) xcp = τmove2 (compP2 P) h stk i pc xcp*
by(*auto simp add*: *τmove2-iff τinstr-stk-drop-exec-move*)
moreover from ‹*P,h ⊢ stk @ [v] [:≤] ST*› **obtain** *ST2* **where** *P,h ⊢ stk [:≤] ST2* **by**(*auto simp add*: *list-all2-append1*)
from *IH2*[*OF exec″ - - this* ‹*conf-xcp′ (compP2 P) h xcp*›] *len bsok* **obtain** *e″ xs″*
where *bisim′*: *P,i,h′ ⊢ (e″, xs″) ↔ (stk″, loc′, pc′ − length (compE2 a), xcp′)*
and *red*: *?red i′ xs e″ xs″ i stk pc (pc′ − length (compE2 a)) xcp xcp′* **by** *fastforce*
from *bisim′*
have *P,a⌊i⌉ := e,h′ ⊢ (Val v⌊e′⌉ := e, xs″) ↔ (stk″ @ [v], loc′, length (compE2 a) + (pc′ − length (compE2 a)), xcp′)*
by(*rule bisim1-bisims1.bisim1AAss2*)
moreover from *exec′* **have** *pc′ ≥ length (compE2 a)*
by(*rule exec-meth-drop-xt-pc*) *auto*
moreover have *no-call2 i pc ⟹ no-call2 (a⌊i⌉ := e) (length (compE2 a) + pc)* **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *red τ stk′ True*
by(*fastforce intro*: *AAss1Red2 elim*!: *AAss-τred1r-xt2 AAss-τred1t-xt2 split*: *if-split-asm*)
next
case *False*
with *pc* **have** [*simp*]: *pc = length (compE2 i)* **by** *simp*

with *bisim2* **obtain** *v2* **where** *i'*: *is-val i'* ⟶ *i'* = *Val v2*
and *stk*: *stk* = [*v2*] **and** *xcp*: *xcp* = *None* **and** *call*: *call1 i'* = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim2 pc len bsok* **have** *red*: *τred1r P t h* (*i'*, *xs*) (*Val v2*, *loc*)
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *τred1r P t h* (*Val v*⌊*i*⌉ := *e*, *xs*) (*Val v*⌊*Val v2*⌉ := *e*, *loc*) **by**(*rule AAss-τred1r-xt2*)
moreover from *pc exec stk xcp*
have *exec'*: *exec-meth-d* (*compP2 P*) ((*compE2 a* @ *compE2 i*) @ *compE2 e* @ [*AStore*, *Push Unit*]) ((*compxE2 a 0 0* @ *compxE2 i* (*length* (*compE2 a*)) (*Suc 0*)) @ *shift* (*length* (*compE2 a* @ *compE2 i*)) (*stack-xlift* (*length* [*v2*, *v*]) (*compxE2 e 0 0*))) *t h* ([] @ [*v2*, *v*], *loc*, *length* (*compE2 a* @ *compE2 i*) + *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *compxE2-size-convs compxE2-stack-xlift-convs exec-move-def*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e* @ [*AStore*, *Push Unit*]) (*stack-xlift* (*length* [*v2*, *v*]) (*compxE2 e 0 0*)) *t h* ([] @ [*v2*, *v*], *loc*, *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 a* @ *compE2 i*), *xcp'*)
by(*rule exec-meth-drop-xt*) *auto*
hence *exec-meth-d* (*compP2 P*) (*compE2 e*) (*stack-xlift* (*length* [*v2*, *v*]) (*compxE2 e 0 0*)) *t h* ([] @ [*v2*, *v*], *loc*, *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 a* @ *compE2 i*), *xcp'*)
by(*rule exec-meth-take*) *simp*
with *bisim3* **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ [*v2*, *v*]
and *exec''*: *exec-move-d P t e h* ([], *loc*, *0*, *None*) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 a* @ *compE2 i*), *xcp'*)
unfolding *exec-move-def* **by**(*blast dest*: *exec-meth-stk-split*)
with *pc xcp* **have** *τ*: *τmove2* (*compP2 P*) *h* [*v2*, *v*] (*a*⌊*i*⌉:= *e*) (*length* (*compE2 a*) + *length* (*compE2 i*)) *None* = *τmove2* (*compP2 P*) *h* [] *e 0 None*
using *τinstr-stk-drop-exec-move*[**where** *stk*=[] **and** *vs*=[*v2*, *v*]] **by**(*simp add*: *τmove2-iff*)
from *bisim2* **have** *length xs* = *length loc* **by**(*rule bisim1-length-xs*)
with *IH3*[*OF exec''*, *of* []] *len bsok* **obtain** *e'' xs''*
where *bisim'*: *P*,*e*,*h'* ⊢ (*e''*, *xs''*) ↔ (*stk''*, *loc'*, *pc'* − *length* (*compE2 a*) − *length* (*compE2 i*), *xcp'*)
and *red*: *?red e loc e'' xs'' e* [] *0* (*pc'* − *length* (*compE2 a*) − *length* (*compE2 i*)) *None xcp'*
by *auto* (*fastforce simp only*: *length-append diff-diff-left*)
from *bisim'*
have *P*,*a*⌊*i*⌉ := *e*,*h'* ⊢ (*Val v*⌊*Val v2*⌉ := *e''*, *xs''*) ↔ (*stk''* @ [*v2*, *v*], *loc'*, *length* (*compE2 a*) + *length* (*compE2 i*) + (*pc'* − *length* (*compE2 a*) − *length* (*compE2 i*)), *xcp'*)
by(*rule bisim1-bisims1.bisim1AAss3*)
moreover from *red τ*
have *?red* (*Val v*⌊*Val v2*⌉ := *e*) *loc* (*Val v*⌊*Val v2*⌉ := *e''*) *xs''* (*a*⌊*i*⌉ := *e*) [*v2*, *v*] (*length* (*compE2 a*) + *length* (*compE2 i*)) *pc' None xcp'*
by(*fastforce intro*: *AAss1Red3 elim*!: *AAss-τred1r-xt3 AAss-τred1t-xt3 split*: *if-split-asm simp add*: *no-call2-def*)
moreover from *exec'* **have** *pc'* ≥ *length* (*compE2 a* @ *compE2 i*)
by(*rule exec-meth-drop-xt-pc*) *auto*
moreover have *no-call2* (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *pc*) **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *τ stk' pc xcp stk* **by**(*fastforce elim*!: *rtranclp-trans*)
qed
next
case (*bisim1AAss3 e n e' xs stk loc pc xcp a i v v'*)
note *IH3* = *bisim1AAss3.IH*(*2*)
note *exec* = ‹*?exec* (*a*⌊*i*⌉ := *e*) (*stk* @ [*v'*, *v*]) *loc* (*length* (*compE2 a*) + *length* (*compE2 i*) + *pc*) *xcp stk' loc' pc' xcp'*›
note *bisim3* = ‹*P*,*e*,*h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *len* = ‹*n* + *max-vars* (*Val v*⌊*Val v'*⌉ := *e'*) ≤ *length xs*›
note *bsok* = ‹*bsok* (*a*⌊*i*⌉ := *e*) *n*›

from ‹P,h ⊢ stk @ [v', v] [:≤] ST› **obtain** T T' ST'
where [$simp$]: $ST = ST'$ @ [T', T]
and wtv: P,h ⊢ v :≤ T **and** wtv': P,h ⊢ v' :≤ T' **and** ST': P,h ⊢ stk [:≤] ST'
by(*auto simp add*: *list-all2-Cons1 list-all2-append1*)
from *bisim3* **have** pc: $pc \le length\ (compE2\ e)$ **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases* $pc < length\ (compE2\ e)$)
case *True*
from *exec* **have** *exec'*: *exec-meth-d* (*compP2 P*) ((*compE2 a* @ *compE2 i*) @ *compE2 e* @ [*AStore*, *Push Unit*]) ((*compxE2 a 0 0* @ *compxE2 i* (*length* (*compE2 a*)) (*Suc 0*)) @ *shift* (*length* (*compE2 a* @ *compE2 i*)) (*stack-xlift* (*length* [v', v]) (*compxE2 e 0 0*))) *t h* (*stk* @ [v', v], *loc*, *length* (*compE2 a* @ *compE2 i*) + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *shift-compxE2 stack-xlift-compxE2 exec-move-def*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e* @ [*AStore*, *Push Unit*]) (*stack-xlift* (*length* [v', v]) (*compxE2 e 0 0*)) *t h* (*stk* @ [v', v], *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 a* @ *compE2 i*), *xcp'*)
by(*rule exec-meth-drop-xt*) *auto*
hence *exec-meth-d* (*compP2 P*) (*compE2 e*) (*stack-xlift* (*length* [v', v]) (*compxE2 e 0 0*)) *t h* (*stk* @ [v', v], *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 a* @ *compE2 i*), *xcp'*)
using *True* **by**(*rule exec-meth-take*)
with *bisim3* **obtain** *stk''* **where** *stk'*: $stk' = stk''$ @ [v', v]
and *exec''*: *exec-move-d P t e h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 a* @ *compE2 i*), *xcp'*)
unfolding *exec-move-def* **by**(*blast dest*: *exec-meth-stk-split*)
with *True* **have** τ: $\tau move2\ (compP2\ P)\ h\ (stk @ [v', v])\ (a\lfloor i \rceil := e)\ (length\ (compE2\ a) + length\ (compE2\ i) + pc)\ xcp = \tau move2\ (compP2\ P)\ h\ stk\ e\ pc\ xcp$
by(*auto simp add*: τ*move2-iff* τ*instr-stk-drop-exec-move*)
moreover from *IH3*[*OF exec''* - - *ST'* ‹*conf-xcp'* (*compP2 P*) *h xcp*›] *len bsok* **obtain** *e'' xs''*
where *bisim'*: P,e,$h' \vdash (e'', xs'') \leftrightarrow (stk'', loc', pc' - length\ (compE2\ a) - length\ (compE2\ i), xcp')$
and *red*: *?red e' xs e'' xs'' e stk pc* (*pc'* − *length* (*compE2 a*) − *length* (*compE2 i*)) *xcp xcp'*
by *auto*(*fastforce simp only*: *length-append diff-diff-left*)
from *bisim'*
have P,$a\lfloor i \rceil := e$,$h' \vdash (Val\ v\lfloor Val\ v' \rceil := e'', xs'') \leftrightarrow (stk'' @ [v', v], loc', length\ (compE2\ a) + length\ (compE2\ i) + (pc' - length\ (compE2\ a) - length\ (compE2\ i)), xcp')$
by(*rule bisim1-bisims1.bisim1AAss3*)
moreover from *exec'* **have** $pc' \ge length\ (compE2\ a @ compE2\ i)$
by(*rule exec-meth-drop-xt-pc*) *auto*
moreover have *no-call2 e pc* ⟹ *no-call2* ($a\lfloor i \rceil := e$) (*length* (*compE2 a*) + *length* (*compE2 i*) + *pc*)
by(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *red* τ *stk'* *True*
by(*fastforce intro*: *AAss1Red3 elim*!: *AAss-*τ*red1r-xt3 AAss-*τ*red1t-xt3 split*: *if-split-asm*)
next
case *False*
with *pc* **have** [*simp*]: $pc = length\ (compE2\ e)$ **by** *simp*
with *bisim3* **obtain** *v2* **where** *stk*: $stk = [v2]$ **and** *xcp*: $xcp = None$
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim3 pc len bsok* **have** *red*: τ*red1r P t h* (*e'*, *xs*) (*Val v2*, *loc*)
by(*auto intro*: *bisim1-Val-*τ*red1r simp add*: *bsok-def*)
hence τ*red1r P t h* (*Val v*⌊*Val v'*⌉ := *e'*, *xs*) (*Val v*⌊*Val v'*⌉ := *Val v2*, *loc*) **by**(*rule AAss-*τ*red1r-xt3*)
moreover have τ: ¬ $\tau move2\ (compP2\ P)\ h\ [v2, v', v]\ (a\lfloor i \rceil := e)\ (length\ (compE2\ a) + length\ (compE2\ i) + length\ (compE2\ e))\ None$
by(*simp add*: τ*move2-iff*)

moreover
have $\exists ta'\ e''.\ P,a\lfloor i\rceil := e,h' \vdash (e'',loc) \leftrightarrow (stk',\ loc',\ pc',\ xcp') \wedge True,P,t \vdash 1\ \langle Val\ v\lfloor Val\ v'\rceil := Val\ v2,\ (h,\ loc)\rangle\ -ta'\rightarrow\ \langle e'',(h',\ loc)\rangle \wedge ta\text{-}bisim\ wbisim1\ (extTA2J1\ P\ ta')\ ta$
proof(*cases* $v = Null$)
case *True* **with** *exec stk xcp* **show** *?thesis*
by(*fastforce elim*!: *exec-meth.cases simp add*: *exec-move-def intro*: *bisim1AAssFail Red1AAssNull*)
next
case *False*
with *exec stk xcp* **obtain** *U A len I* **where** [*simp*]: $v = Addr\ A\ v' = Intg\ I$
and *hA*: *typeof-addr h A* = $\lfloor$*Array-type U len*$\rfloor$
by(*fastforce simp add*: *exec-move-def exec-meth-instr is-Ref-def*)
from *ST' stk* **obtain** *T3* **where** *wt3'*: $typeof_h\ v2 = \lfloor T3\rfloor$ **by**(*auto simp add*: *list-all2-Cons1 conf-def*)
show *?thesis*
proof(*cases* $0 <=s\ I \wedge sint\ I < int\ len$)
case *True*
note $I = True$
show *?thesis*
proof(*cases* $P \vdash T3 \leq U$)
case *True*
with *exec stk xcp True hA I wt3'* **show** *?thesis*
by(*fastforce elim*!: *exec-meth.cases simp add*: *compP2-def exec-move-def ta-bisim-def ta-upd-simps intro*: *Red1AAss bisim1AAss4 split*: *if-split-asm*)
next
case *False*
with *exec stk xcp True hA I wt3'* **show** *?thesis*
by(*fastforce elim*!: *exec-meth.cases simp add*: *compP2-def exec-move-def intro*: *Red1AAssStore bisim1AAssFail split*: *if-split-asm*)
qed
next
case *False*
with *exec stk xcp hA* **show** *?thesis*
by(*fastforce elim*!: *exec-meth.cases intro*: *bisim1AAssFail Red1AAssBounds simp add*: *exec-move-def split*: *if-split-asm*)
qed
qed
ultimately show *?thesis* **using** *exec xcp stk* **by**(*fastforce simp add*: *no-call2-def*)
qed
next
case (*bisim1AAssThrow1 A n a xs stk loc pc i e*)
note *exec* = ‹*?exec* $(A\lfloor i\rceil := e)$ *stk loc pc* $\lfloor a\rfloor$ *stk' loc' pc' xcp'*›
note *bisim1* = ‹$P,A,h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a\rfloor)$›
from *bisim1* **have** *pc*: $pc < length\ (compE2\ A)$ **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim1* **have** $match\text{-}ex\text{-}table\ (compP2\ P)\ (cname\text{-}of\ h\ a)\ (0 + pc)\ (compxE2\ A\ 0\ 0) = None$
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)
with *exec pc* **have** *False*
by(*auto elim*!: *exec-meth.cases simp add*: *match-ex-table-not-pcs-None exec-move-def*)
thus *?case* **..**
next
case (*bisim1AAssThrow2 i n a xs stk loc pc A e v*)
note *exec* = ‹*?exec* $(A\lfloor i\rceil := e)\ (stk\ @\ [v])\ loc\ (length\ (compE2\ A) + pc)\ \lfloor a\rfloor$ *stk' loc' pc' xcp'*›
note *bisim2* = ‹$P,i,h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a\rfloor)$›
from *bisim2* **have** *pc*: $pc < length\ (compE2\ i)$ **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim2* **have** $match\text{-}ex\text{-}table\ (compP2\ P)\ (cname\text{-}of\ h\ a)\ (length\ (compE2\ A) + pc)\ (compxE2$

i (*length* (*compE2 A*)) *0*) = *None*
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)
with *exec pc* **have** *False*
apply(*auto elim*!: *exec-meth.cases simp add*: *compxE2-stack-xlift-convs compxE2-size-convs exec-move-def*)
apply(*auto simp add*: *match-ex-table-append-not-pcs*)
done
thus *?case* **..**
next
case (*bisim1AAssThrow3 e n a xs stk loc pc A i v' v*)
note *exec* = ‹*?exec* (*A*⌊*i*⌉ := *e*) (*stk* @ [*v'*, *v*]) *loc* (*length* (*compE2 A*) + *length* (*compE2 i*) + *pc*) ⌊*a*⌋ *stk'* *loc'* *pc'* *xcp'*›
note *bisim2* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
from *bisim2* **have** *match-ex-table* (*compP2 P*) (*cname-of h a*) (*length* (*compE2 A*) + *length* (*compE2 i*) + *pc*) (*compxE2 e* (*length* (*compE2 A*) + *length* (*compE2 i*)) *0*) = *None*
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)
with *exec* **have** *False*
apply(*auto elim*!: *exec-meth.cases simp add*: *compxE2-stack-xlift-convs compxE2-size-convs exec-move-def*)
apply(*auto dest*!: *match-ex-table-stack-xliftD match-ex-table-shift-pcD dest*: *match-ex-table-pcsD simp add*: *match-ex-table-append match-ex-table-shift-pc-None*)
done
thus *?case* **..**
next
case (*bisim1AAssFail a n i e ad xs v' v v''*)
note *exec* = ‹*?exec* (*a*⌊*i*⌉ := *e*) [*v'*, *v*, *v''*] *xs* (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)) ⌊*ad*⌋ *stk'* *loc'* *pc'* *xcp'*›
hence *False*
by(*auto elim*!: *exec-meth.cases simp add*: *match-ex-table-append exec-move-def*
dest!: *match-ex-table-shift-pcD match-ex-table-pc-length-compE2*)
thus *?case* **..**
next
case (*bisim1AAss4 a n i e xs*)
have *P*,*a*⌊*i*⌉ := *e*,*h* ⊢ (*unit*, *xs*) ↔ ([*Unit*], *xs*, *length* (*compE2* (*a*⌊*i*⌉ := *e*)), *None*) **by**(*rule bisim1Val2*) *simp*
moreover have *τmove2* (*compP2 P*) *h* [] (*a*⌊*i*⌉ := *e*) (*Suc* (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*))) *None*
by(*simp add*: *τmove2-iff*)
moreover note ‹*?exec* (*a*⌊*i*⌉ := *e*) [] *xs* (*Suc* (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*))) *None* *stk'* *loc'* *pc'* *xcp'*›
ultimately show *?case*
by(*fastforce elim*!: *exec-meth.cases simp add*: *ac-simps exec-move-def*)
next
case (*bisim1ALength a n a' xs stk loc pc xcp*)
note *IH* = *bisim1ALength.IH*(*2*)
note *exec* = ‹*?exec* (*a*·*length*) *stk* *loc* *pc* *xcp* *stk'* *loc'* *pc'* *xcp'*›
note *bisim* = ‹*P*,*a*,*h* ⊢ (*a'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *len* = ‹*n* + *max-vars* (*a'*·*length*) ≤ *length xs*›
note *bsok* = ‹*bsok* (*a*·*length*) *n*›
from *bisim* **have** *pc*: *pc* ≤ *length* (*compE2 a*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 a*))
case *True*
with *exec* **have** *exec'*: *?exec a stk loc pc xcp stk' loc' pc' xcp'* **by**(*auto simp add*: *exec-move-ALength*)
from *True* **have** *τ*: *τmove2* (*compP2 P*) *h stk* (*a*·*length*) *pc xcp* = *τmove2* (*compP2 P*) *h stk a pc xcp* **by**(*simp add*: *τmove2-iff*)

```
    with IH[OF exec' - - ‹P,h ⊢ stk [:≤] ST› ‹conf-xcp' (compP2 P) h xcp›] len bsok obtain e'' xs''
      where bisim': P,a,h' ⊢ (e'', xs'') ↔ (stk', loc', pc', xcp')
      and red: ?red a' xs e'' xs'' a stk pc pc' xcp xcp' by auto
    from bisim' have P,a•length,h' ⊢ (e''•length, xs'') ↔ (stk', loc', pc', xcp')
      by(rule bisim1-bisims1.bisim1ALength)
   with red τ show ?thesis by(fastforce intro: ALength1Red elim!: ALength-τred1r-xt ALength-τred1t-xt
simp add: no-call2-def)
  next
    case False
    with pc have pc: pc = length (compE2 a) by auto
    with bisim obtain v where stk: stk = [v] and xcp: xcp = None
      by(auto dest: bisim1-pc-length-compE2D)
    with bisim pc len bsok have τred1r P t h (a', xs) (Val v, loc)
      by(auto intro: bisim1-Val-τred1r simp add: bsok-def)
    hence τred1r P t h (a'•length, xs) (Val v•length, loc) by(rule ALength-τred1r-xt)
    moreover
     moreover have τ: ¬ τmove2 (compP2 P) h [v] (a•length) (length (compE2 a)) None by(simp
add: τmove2-iff)
     moreover have ∃ta' e''. P,a•length,h' ⊢ (e'',loc) ↔ (stk', loc', pc', xcp') ∧ True,P,t ⊢1 ⟨Val
v•length, (h, loc)⟩ −ta'→ ⟨e'',(h', loc)⟩ ∧ ta-bisim wbisim1 (extTA2J1 P ta') ta
    proof(cases v = Null)
      case True with exec stk xcp pc show ?thesis
      by(fastforce elim!: exec-meth.cases simp add: exec-move-def intro: bisim1ALengthNull Red1ALength-
Null)
    next
      case False
      with exec stk xcp pc ‹P,h ⊢ stk [:≤] ST›
      obtain U A len where [simp]: v = Addr A
        and hA: typeof-addr h A = ⌊Array-type U len⌋
        by(fastforce simp add: exec-move-def exec-meth-instr is-Ref-def list-all2-Cons1)
      have P,a•length,h' ⊢ (Val (Intg (word-of-int (int len))),loc) ↔ ([Intg (word-of-int (int len))], loc,
length (compE2 (a•length)), None)
        by(rule bisim1Val2) simp
      thus ?thesis using exec stk xcp hA pc
        by(fastforce elim!: exec-meth.cases intro: Red1ALength simp add: exec-move-def)
    qed
   ultimately show ?thesis using τ pc xcp stk by(fastforce elim!: rtranclp-trans simp add: no-call2-def)
  qed
next
  case (bisim1ALengthThrow A n a xs stk loc pc)
  note exec = ‹?exec (A•length) stk loc pc ⌊a⌋ stk' loc' pc' xcp'›
  note bisim1 = ‹P,A,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
  from bisim1 have pc: pc < length (compE2 A) by(auto dest: bisim1-ThrowD)
  from bisim1 have match-ex-table (compP2 P) (cname-of h a) (0 + pc) (compxE2 A 0 0) = None
    unfolding compP2-def by(rule bisim1-xcp-Some-not-caught)
  with exec pc have False by(auto elim!: exec-meth.cases simp add: exec-move-def)
  thus ?case ..
next
  case (bisim1ALengthNull a n xs)
  note exec = ‹?exec (a•length) [Null] xs (length (compE2 a)) ⌊addr-of-sys-xcpt NullPointer⌋ stk' loc'
pc' xcp'›
 hence False by(auto elim!: exec-meth.cases dest!: match-ex-table-pc-length-compE2 simp add: exec-move-def)
  thus ?case ..
next
```

case (*bisim1FAcc e n e′ xs stk loc pc xcp F D*)
note *IH* = *bisim1FAcc.IH*(*2*)
note *exec* = ‹*?exec* (*e*·*F*{*D*}) *stk loc pc xcp stk′ loc′ pc′ xcp′*›
note *bisim* = ‹*P*,*e*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *len* = ‹*n* + *max-vars* (*e′*·*F*{*D*}) ≤ *length xs*›
note *bsok* = ‹*bsok* (*e*·*F*{*D*}) *n*›
from *bisim* **have** *pc*: *pc* ≤ *length* (*compE2 e*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e*))
case *True*
with *exec* **have** *exec′*: *?exec e stk loc pc xcp stk′ loc′ pc′ xcp′* **by**(*simp add*: *exec-move-FAcc*)
from *True* **have** *τ*: *τmove2* (*compP2 P*) *h stk* (*e*·*F*{*D*}) *pc xcp* = *τmove2* (*compP2 P*) *h stk e pc xcp* **by**(*simp add*: *τmove2-iff*)
with *IH*[*OF exec′ - -* ‹*P*,*h* ⊢ *stk* [:≤] *ST*› ‹*conf-xcp′* (*compP2 P*) *h xcp*›] *len bsok* **obtain** *e″ xs″*
where *bisim′*: *P*,*e*,*h′* ⊢ (*e″*, *xs″*) ↔ (*stk′*, *loc′*, *pc′*, *xcp′*)
and *red*: *?red e′ xs e″ xs″ e stk pc pc′ xcp xcp′* **by** *auto*
from *bisim′* **have** *P*,*e*·*F*{*D*},*h′* ⊢ (*e″*·*F*{*D*}, *xs″*) ↔ (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*rule bisim1-bisims1.bisim1FAcc*)
with *red τ* **show** *?thesis* **by**(*fastforce intro*: *FAcc1Red elim*!: *FAcc-τred1r-xt FAcc-τred1t-xt simp add*: *no-call2-def*)
next
case *False*
with *pc* **have** *pc*: *pc* = *length* (*compE2 e*) **by** *auto*
with *bisim* **obtain** *v* **where** *stk*: *stk* = [*v*] **and** *xcp*: *xcp* = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim pc len bsok* **have** *τred1r P t h* (*e′*, *xs*) (*Val v*, *loc*)
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *τred1r P t h* (*e′*·*F*{*D*}, *xs*) (*Val v*·*F*{*D*}, *loc*) **by**(*rule FAcc-τred1r-xt*)
moreover have *τ*: ¬ *τmove2* (*compP2 P*) *h* [*v*] (*e*·*F*{*D*}) (*length* (*compE2 e*)) *None* **by**(*simp add*: *τmove2-iff*)
moreover have ∃ *ta′ e″*. *P*,*e*·*F*{*D*},*h′* ⊢ (*e″*,*loc*) ↔ (*stk′*, *loc′*, *pc′*, *xcp′*) ∧ *True*,*P*,*t* ⊢*1* ⟨*Val v*·*F*{*D*}, (*h*, *loc*)⟩ −*ta′*→ ⟨*e″*,(*h′*, *loc*)⟩ ∧ *ta-bisim wbisim1* (*extTA2J1 P ta′*) *ta*
proof(*cases v* = *Null*)
case *True* **with** *exec stk xcp pc* **show** *?thesis*
by(*fastforce elim*!: *exec-meth.cases simp add*: *exec-move-def intro*: *bisim1FAccNull Red1FAccNull*)
next
case *False*
with *exec stk xcp pc* ‹*P*,*h* ⊢ *stk* [:≤] *ST*›
obtain *A* **where** [*simp*]: *v* = *Addr A*
by(*fastforce simp add*: *exec-move-def exec-meth-instr is-Ref-def compP2-def*)
from *exec False pc stk xcp* **obtain** *v′* **where** *v′*: *heap-read h A* (*CField D F*) *v′ stk′* = [*v′*]
by(*auto simp add*: *exec-move-def exec-meth-instr*)
have *P*,*e*·*F*{*D*},*h′* ⊢ (*Val v′*,*loc*) ↔ ([*v′*], *loc*, *length* (*compE2* (*e*·*F*{*D*})), *None*)
by(*rule bisim1Val2*) *simp*
thus *?thesis* **using** *exec stk xcp pc v′*
by(*fastforce elim*!: *exec-meth.cases intro*: *Red1FAcc simp add*: *exec-move-def ta-upd-simps ta-bisim-def*)
qed
ultimately show *?thesis* **using** *τ pc xcp stk* **by**(*fastforce elim*!: *rtranclp-trans simp add*: *no-call2-def*)
qed
next
case (*bisim1FAccThrow e n a xs stk loc pc F D*)
note *exec* = ‹*?exec* (*e*·*F*{*D*}) *stk loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
note *bisim1* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›

```
  from bisim1 have pc: pc < length (compE2 e) by(auto dest: bisim1-ThrowD)
  from bisim1 have match-ex-table (compP2 P) (cname-of h a) (0 + pc) (compxE2 e 0 0) = None
    unfolding compP2-def by(rule bisim1-xcp-Some-not-caught)
  with exec pc have False by(auto elim!: exec-meth.cases simp add: exec-move-def)
  thus ?case ..
next
  case (bisim1FAccNull e n F D xs)
  note exec = ‹?exec (e·F{D}) [Null] xs (length (compE2 e)) ⌊addr-of-sys-xcpt NullPointer⌋ stk' loc'
pc' xcp'›
 hence False by(auto elim!: exec-meth.cases dest!: match-ex-table-pc-length-compE2 simp add: exec-move-def)
  thus ?case ..
next
  case (bisim1FAss1 e n e' xs stk loc pc xcp e2 F D)
  note IH1 = bisim1FAss1.IH(2)
  note IH2 = bisim1FAss1.IH(4)
  note exec = ‹?exec (e·F{D} := e2) stk loc pc xcp stk' loc' pc' xcp'›
  note bisim1 = ‹P,e,h ⊢ (e', xs) ↔ (stk, loc, pc, xcp)›
  note bisim2 = ‹P,e2,h ⊢ (e2, loc) ↔ ([], loc, 0, None)›
  note len = ‹n + max-vars (e'·F{D} := e2) ≤ length xs›
  note bsok = ‹bsok (e·F{D} := e2) n›
  from bisim1 have pc: pc ≤ length (compE2 e) by(rule bisim1-pc-length-compE2)
  show ?case
  proof(cases pc < length (compE2 e))
    case True
    with exec have exec': ?exec e stk loc pc xcp stk' loc' pc' xcp' by(simp add: exec-move-FAss1)
    from True have τ: τmove2 (compP2 P) h stk (e·F{D} := e2) pc xcp = τmove2 (compP2 P) h
stk e pc xcp
      by(simp add: τmove2-iff)
    with IH1[OF exec' - - ‹P,h ⊢ stk [:≤] ST› ‹conf-xcp' (compP2 P) h xcp›] len bsok obtain e'' xs''
      where bisim': P,e,h' ⊢ (e'', xs'') ↔ (stk', loc', pc', xcp')
      and red: ?red e' xs e'' xs'' e stk pc pc' xcp xcp' by auto
    from bisim' have P,e·F{D} := e2,h' ⊢ (e''·F{D} := e2, xs'') ↔ (stk', loc', pc', xcp')
      by(rule bisim1-bisims1.bisim1FAss1)
   with red τ show ?thesis by(fastforce intro: FAss1Red1 elim!: FAss-τred1r-xt1 FAss-τred1t-xt1 simp
add: no-call2-def)
  next
    case False
    with pc have pc: pc = length (compE2 e) by auto
    with bisim1 obtain v where stk: stk = [v] and xcp: xcp = None
      by(auto dest: bisim1-pc-length-compE2D)
    with bisim1 pc len bsok have rede1': τred1r P t h (e', xs) (Val v, loc)
      by(auto intro: bisim1-Val-τred1r simp add: bsok-def)
    hence τred1r P t h (e'·F{D} := e2, xs) (Val v·F{D} := e2, loc) by(rule FAss-τred1r-xt1)
    moreover from pc exec stk xcp
     have exec': exec-meth-d (compP2 P) (compE2 e @ compE2 e2 @ [Putfield F D, Push Unit])
(compxE2 e 0 0 @ shift (length (compE2 e)) (stack-xlift (length [v]) (compxE2 e2 0 0))) t h ([] @ [v],
loc, length (compE2 e) + 0, None) ta h' (stk', loc', pc', xcp')
      by(simp add: compxE2-size-convs compxE2-stack-xlift-convs exec-move-def)
    hence exec-meth-d (compP2 P) (compE2 e2 @ [Putfield F D, Push Unit]) (stack-xlift (length [v])
(compxE2 e2 0 0)) t h ([] @ [v], loc, 0, None) ta h' (stk', loc', pc' − length (compE2 e), xcp')
      by(rule exec-meth-drop-xt) auto
    hence exec-meth-d (compP2 P) (compE2 e2) (stack-xlift (length [v]) (compxE2 e2 0 0)) t h ([] @
[v], loc, 0, None) ta h' (stk', loc', pc' − length (compE2 e), xcp')
      by(rule exec-meth-take) simp
```

with *bisim2* **obtain** *stk''* **where** *stk'*: *stk' = stk'' @ [v]*
and *exec''*: *exec-move-d P t e2 h ([], loc, 0, None) ta h' (stk'', loc', pc' − length (compE2 e), xcp')*
unfolding *exec-move-def* **by**(*blast dest*: *exec-meth-stk-split*)
with *pc xcp* **have** *τ*: *τmove2 (compP2 P) h [v] (e•F{D} := e2) (length (compE2 e)) None = τmove2 (compP2 P) h [] e2 0 None*
using *τinstr-stk-drop-exec-move*[**where** *stk*=[] **and** *vs* = *[v]*] **by**(*simp add*: *τmove2-iff*)
from *bisim1* **have** *length xs = length loc* **by**(*rule bisim1-length-xs*)
with *IH2*[*OF exec''*, *of* []] *len bsok* **obtain** *e'' xs''*
where *bisim'*: *P,e2,h' ⊢ (e'', xs'') ↔ (stk'', loc', pc' − length (compE2 e), xcp')*
and *red*: *?red e2 loc e'' xs'' e2 [] 0 (pc' − length (compE2 e)) None xcp'* **by** *auto*
from *bisim'*
have *P,e•F{D} := e2,h' ⊢ (Val v•F{D} := e'', xs'') ↔ (stk'' @ [v], loc', length (compE2 e) + (pc' − length (compE2 e)), xcp')*
by(*rule bisim1-bisims1.bisim1FAss2*)
moreover from *red τ*
have *?red (Val v•F{D} := e2) loc (Val v•F{D} := e'') xs'' (e•F{D} := e2) [v] (length (compE2 e)) pc' None xcp'*
by(*fastforce intro*: *FAss1Red2 elim*!: *FAss-τred1r-xt2 FAss-τred1t-xt2 split*: *if-split-asm simp add*: *no-call2-def*)
moreover from *exec'* **have** *pc' ≥ length (compE2 e)*
by(*rule exec-meth-drop-xt-pc*) *auto*
moreover have *no-call2 (e•F{D} := e2) pc* **using** *pc* **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *τ stk' pc xcp stk* **by**(*fastforce elim*!: *rtranclp-trans*)
qed
next
case (*bisim1FAss2 e2 n e' xs stk loc pc xcp e F D v*)
note *IH2 = bisim1FAss2.IH*(*2*)
note *exec = ‹?exec (e•F{D} := e2) (stk @ [v]) loc (length (compE2 e) + pc) xcp stk' loc' pc' xcp'›*
note *bisim2 = ‹P,e2,h ⊢ (e', xs) ↔ (stk, loc, pc, xcp)›*
note *len = ‹n + max-vars (Val v•F{D} := e') ≤ length xs›*
note *bsok = ‹bsok (e•F{D} := e2) n›*
note *ST = ‹P,h ⊢ stk @ [v] [:≤] ST›*
then obtain *T ST'* **where** *ST'*: *P,h ⊢ stk [:≤] ST'* **and** *T*: *typeof$_h$ v = ⌊T⌋*
by(*auto simp add*: *list-all2-append1 list-all2-Cons1 conf-def*)

from *bisim2* **have** *pc*: *pc ≤ length (compE2 e2)* **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc < length (compE2 e2)*)
case *True*
from *exec* **have** *exec'*: *exec-meth-d (compP2 P) (compE2 e @ compE2 e2 @ [Putfield F D, Push Unit]) (compxE2 e 0 0 @ shift (length (compE2 e)) (stack-xlift (length [v]) (compxE2 e2 0 0))) t h (stk @ [v], loc, length (compE2 e) + pc, xcp) ta h' (stk', loc', pc', xcp')*
by(*simp add*: *shift-compxE2 stack-xlift-compxE2 exec-move-def*)
hence *exec-meth-d (compP2 P) (compE2 e2 @ [Putfield F D, Push Unit]) (stack-xlift (length [v]) (compxE2 e2 0 0)) t h (stk @ [v], loc, pc, xcp) ta h' (stk', loc', pc' − length (compE2 e), xcp')*
by(*rule exec-meth-drop-xt*) *auto*
hence *exec-meth-d (compP2 P) (compE2 e2) (stack-xlift (length [v]) (compxE2 e2 0 0)) t h (stk @ [v], loc, pc, xcp) ta h' (stk', loc', pc' − length (compE2 e), xcp')*
using *True* **by**(*rule exec-meth-take*)
with *bisim2* **obtain** *stk''* **where** *stk'*: *stk' = stk'' @ [v]*
and *exec''*: *exec-move-d P t e2 h (stk, loc, pc, xcp) ta h' (stk'', loc', pc' − length (compE2 e), xcp')*
unfolding *exec-move-def* **by**(*blast dest*: *exec-meth-stk-split*)

with *True* **have** τ: $\tau move2\ (compP2\ P)\ h\ (stk\ @\ [v])\ (e{\cdot}F\{D\} := e2)\ (length\ (compE2\ e) + pc)\ xcp = \tau move2\ (compP2\ P)\ h\ stk\ e2\ pc\ xcp$
by(*auto simp add*: $\tau move2\text{-}iff\ \tau instr\text{-}stk\text{-}drop\text{-}exec\text{-}move$)
moreover from *IH2*[*OF exec'' - - ST' ‹conf-xcp' (compP2 P) h xcp›*] *len bsok* **obtain** *e'' xs''*
where *bisim'*: $P,e2,h' \vdash (e'', xs'') \leftrightarrow (stk'', loc', pc' - length\ (compE2\ e), xcp')$
and *red*: *?red e' xs e'' xs'' e2 stk pc* $(pc' - length\ (compE2\ e))\ xcp\ xcp'$ **by** *auto*
from *bisim'* **have** $P,e{\cdot}F\{D\} := e2,h' \vdash (Val\ v{\cdot}F\{D\} := e'', xs'') \leftrightarrow (stk'' @ [v], loc', length\ (compE2\ e) + (pc' - length\ (compE2\ e)), xcp')$
by(*rule bisim1-bisims1.bisim1FAss2*)
moreover from *exec'* **have** $pc' \geq length\ (compE2\ e)$
by(*rule exec-meth-drop-xt-pc*) *auto*
ultimately show *?thesis* **using** *red* τ *stk' True*
by(*fastforce intro*: *FAss1Red2 elim*!: *FAss-*τ*red1r-xt2 FAss-*τ*red1t-xt2 split*: *if-split-asm simp add*: *no-call2-def*)
next
case *False*
with *pc* **have** [*simp*]: $pc = length\ (compE2\ e2)$ **by** *simp*
with *bisim2* **obtain** *v2* **where** *stk*: $stk = [v2]$ **and** *xcp*: $xcp = None$
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim2 pc len bsok* **have** *red*: $\tau red1r\ P\ t\ h\ (e', xs)\ (Val\ v2, loc)$
by(*auto intro*: *bisim1-Val-*τ*red1r simp add*: *bsok-def*)
hence $\tau red1r\ P\ t\ h\ (Val\ v{\cdot}F\{D\} := e', xs)\ (Val\ v{\cdot}F\{D\} := Val\ v2, loc)$ **by**(*rule FAss-*τ*red1r-xt2*)
moreover have τ: $\neg\ \tau move2\ (compP2\ P)\ h\ [v2, v]\ (e{\cdot}F\{D\} := e2)\ (length\ (compE2\ e) + length\ (compE2\ e2))\ None$ **by**(*simp add*: $\tau move2\text{-}iff$)
moreover
have $\exists\ ta'\ e''.\ P,e{\cdot}F\{D\} := e2,h' \vdash (e'',loc) \leftrightarrow (stk', loc', pc', xcp') \wedge True,P,t \vdash 1\ \langle Val\ v{\cdot}F\{D\} := Val\ v2, (h, loc)\rangle\ -ta'\rightarrow\ \langle e'',(h', loc)\rangle \wedge ta\text{-}bisim\ wbisim1\ (extTA2J1\ P\ ta')\ ta$
proof(*cases* $v = Null$)
case *True* **with** *exec stk xcp* **show** *?thesis*
by(*fastforce elim*!: *exec-meth.cases simp add*: *exec-move-def intro*: *bisim1FAssNull Red1FAssNull*)
next
case *False* **with** *exec stk xcp T* **show** *?thesis*
by(*fastforce simp add*: *exec-move-def compP2-def exec-meth-instr is-Ref-def ta-upd-simps ta-bisim-def intro*: *bisim1FAss3 Red1FAss*)
qed
ultimately show *?thesis* **using** *exec xcp stk* **by**(*fastforce simp add*: *no-call2-def*)
qed
next
case (*bisim1FAssThrow1 e n a xs stk loc pc e2 F D*)
note *exec* = ‹*?exec* $(e{\cdot}F\{D\} := e2)\ stk\ loc\ pc\ \lfloor a \rfloor\ stk'\ loc'\ pc'\ xcp'$›
note *bisim1* = ‹$P,e,h \vdash (Throw\ a, xs) \leftrightarrow (stk, loc, pc, \lfloor a \rfloor)$›
from *bisim1* **have** *pc*: $pc < length\ (compE2\ e)$ **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim1* **have** $match\text{-}ex\text{-}table\ (compP2\ P)\ (cname\text{-}of\ h\ a)\ (0 + pc)\ (compxE2\ e\ 0\ 0) = None$
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)
with *exec pc* **have** *False*
by(*auto elim*!: *exec-meth.cases simp add*: *exec-move-def match-ex-table-not-pcs-None*)
thus *?case* **..**
next
case (*bisim1FAssThrow2 e2 n a xs stk loc pc e F D v*)
note *exec* = ‹*?exec* $(e{\cdot}F\{D\} := e2)\ (stk\ @\ [v])\ loc\ (length\ (compE2\ e) + pc)\ \lfloor a \rfloor\ stk'\ loc'\ pc'\ xcp'$›
note *bisim2* = ‹$P,e2,h \vdash (Throw\ a, xs) \leftrightarrow (stk, loc, pc, \lfloor a \rfloor)$›
hence $match\text{-}ex\text{-}table\ (compP2\ P)\ (cname\text{-}of\ h\ a)\ (length\ (compE2\ e) + pc)\ (compxE2\ e2\ (length\ (compE2\ e))\ 0) = None$
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)

with *exec* **have** *False*
by(*auto elim*!: *exec-meth.cases simp add*: *compxE2-stack-xlift-convs exec-move-def*)(*auto dest*!: *match-ex-table-stack-xliftD simp add*: *match-ex-table-append-not-pcs*)
thus *?case* **..**
next
case (*bisim1FAssNull e n e2 F D xs v*)
note *exec* = ‹*?exec* (*e*·*F*{*D*} := *e2*) [*v*, *Null*] *xs* (*length* (*compE2 e*) + *length* (*compE2 e2*)) ⌊*addr-of-sys-xcpt NullPointer*⌋ *stk*′ *loc*′ *pc*′ *xcp*′›
hence *False*
by(*auto elim*!: *exec-meth.cases simp add*: *match-ex-table-append-not-pcs compxE2-size-convs exec-move-def dest*!: *match-ex-table-shift-pcD match-ex-table-pc-length-compE2*)
thus *?case* **..**
next
case (*bisim1FAss3 e n e2 F D xs*)
have *P*,*e*·*F*{*D*} := *e2*,*h* ⊢ (*unit*, *xs*) ↔ ([*Unit*], *xs*, *length* (*compE2* (*e*·*F*{*D*} := *e2*)), *None*) **by**(*rule bisim1Val2*) *simp*
moreover have *τmove2* (*compP2 P*) *h* [] (*e*·*F*{*D*} := *e2*) (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e2*))) *None* **by**(*simp add*: *τmove2-iff*)
moreover note ‹*?exec* (*e*·*F*{*D*} := *e2*) [] *xs* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e2*))) *None stk*′ *loc*′ *pc*′ *xcp*′›
ultimately show *?case*
by(*fastforce elim*!: *exec-meth.cases simp add*: *ac-simps exec-move-def*)
next
case (*bisim1CAS1 a n a*′ *xs stk loc pc xcp i e D F*)
note *IH1* = *bisim1CAS1.IH*(*2*)
note *IH2* = *bisim1CAS1.IH*(*4*)
note *exec* = ‹*?exec* (*a*·*compareAndSwap*(*D*·*F*, *i*, *e*)) *stk loc pc xcp stk*′ *loc*′ *pc*′ *xcp*′›
note *bisim1* = ‹*P*,*a*,*h* ⊢ (*a*′, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim2* = ‹*P*,*i*,*h* ⊢ (*i*, *loc*) ↔ ([], *loc*, *0*, *None*)›
note *len* = ‹*n* + *max-vars* - ≤ *length xs*›
note *bsok* = ‹*bsok* (*a*·*compareAndSwap*(*D*·*F*, *i*, *e*)) *n*›
from *bisim1* **have** *pc*: *pc* ≤ *length* (*compE2 a*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 a*))
case *True*
with *exec* **have** *exec*′: *?exec a stk loc pc xcp stk*′ *loc*′ *pc*′ *xcp*′ **by**(*simp add*: *exec-move-CAS1*)
from *True* **have** *τ*: *τmove2* (*compP2 P*) *h stk* (*a*·*compareAndSwap*(*D*·*F*, *i*, *e*)) *pc xcp* = *τmove2* (*compP2 P*) *h stk a pc xcp* **by**(*simp add*: *τmove2-iff*)
with *IH1*[*OF exec*′ - - ‹*P*,*h* ⊢ *stk* [:≤] *ST*› ‹*conf-xcp*′ (*compP2 P*) *h xcp*›] *len bsok* **obtain** *e*′′ *xs*′′
where *bisim*′: *P*,*a*,*h*′ ⊢ (*e*′′, *xs*′′) ↔ (*stk*′, *loc*′, *pc*′, *xcp*′)
and *red*: *?red a*′ *xs e*′′ *xs*′′ *a stk pc pc*′ *xcp xcp*′ **by** *auto*
from *bisim*′ **have** *P*,*a*·*compareAndSwap*(*D*·*F*, *i*, *e*),*h*′ ⊢ (*e*′′·*compareAndSwap*(*D*·*F*, *i*, *e*), *xs*′′) ↔ (*stk*′, *loc*′, *pc*′, *xcp*′)
by(*rule bisim1-bisims1.bisim1CAS1*)
moreover from *True* **have** *no-call2* (*a*·*compareAndSwap*(*D*·*F*, *i*, *e*)) *pc* = *no-call2 a pc* **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *red τ* **by**(*fastforce intro*: *CAS1Red1 elim*!: *CAS-τred1r-xt1 CAS-τred1t-xt1*)
next
case *False*
with *pc* **have** *pc*: *pc* = *length* (*compE2 a*) **by** *auto*
with *bisim1* **obtain** *v* **where** *a*′: *is-val a*′ ⟶ *a*′ = *Val v*
and *stk*: *stk* = [*v*] **and** *xcp*: *xcp* = *None* **and** *call*: *call1 a*′ = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim1 pc len bsok* **have** *rede1*′: *τred1r P t h* (*a*′, *xs*) (*Val v*, *loc*)

by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *τred1r P t h* (*a′*•*compareAndSwap*(*D*•*F*, *i*, *e*), *xs*) (*Val v*•*compareAndSwap*(*D*•*F*, *i*, *e*), *loc*) **by**(*rule CAS-τred1r-xt1*)
moreover from *pc exec stk xcp*
have *exec′*: *exec-meth-d* (*compP2 P*) (*compE2 a* @ *compE2 i* @ *compE2 e* @ [*CAS F D*]) (*compxE2 a 0 0* @ *shift* (*length* (*compE2 a*)) (*stack-xlift* (*length* [*v*]) (*compxE2 i 0 0*) @ *shift* (*length* (*compE2 i*)) (*compxE2 e 0* (*Suc* (*Suc 0*))))) *t h* ([] @ [*v*], *loc*, *length* (*compE2 a*) + *0*, *None*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *compxE2-size-convs compxE2-stack-xlift-convs exec-move-def*)
hence *exec-meth-d* (*compP2 P*) (*compE2 i* @ *compE2 e* @ [*CAS F D*]) (*stack-xlift* (*length* [*v*]) (*compxE2 i 0 0*) @ *shift* (*length* (*compE2 i*)) (*compxE2 e 0* (*Suc* (*Suc 0*)))) *t h* ([] @ [*v*], *loc*, *0*, *None*) *ta h′* (*stk′*, *loc′*, *pc′* − *length* (*compE2 a*), *xcp′*)
by(*rule exec-meth-drop-xt*) *auto*
hence *exec-meth-d* (*compP2 P*) (*compE2 i*) (*stack-xlift* (*length* [*v*]) (*compxE2 i 0 0*)) *t h* ([] @ [*v*], *loc*, *0*, *None*) *ta h′* (*stk′*, *loc′*, *pc′* − *length* (*compE2 a*), *xcp′*)
by(*rule exec-meth-take-xt*) *simp*
with *bisim2* **obtain** *stk″* **where** *stk′*: *stk′* = *stk″* @ [*v*]
and *exec″*: *exec-move-d P t i h* ([], *loc*, *0*, *None*) *ta h′* (*stk″*, *loc′*, *pc′* − *length* (*compE2 a*), *xcp′*)
unfolding *exec-move-def* **by**(*blast dest*: *exec-meth-stk-split*)
with *pc xcp* **have** *τ*: *τmove2* (*compP2 P*) *h* [*v*] (*a*•*compareAndSwap*(*D*•*F*, *i*, *e*)) (*length* (*compE2 a*)) *None* = *τmove2* (*compP2 P*) *h* [] *i 0 None*
using *τinstr-stk-drop-exec-move*[**where** *stk*=[] **and** *vs*=[*v*]]
by(*auto simp add*: *τmove2-iff*)
from *bisim1* **have** *length xs* = *length loc* **by**(*rule bisim1-length-xs*)
with *IH2*[*OF exec″*] *len bsok* **obtain** *e″ xs″*
where *bisim′*: *P*,*i*,*h′* ⊢ (*e″*, *xs″*) ↔ (*stk″*, *loc′*, *pc′* − *length* (*compE2 a*), *xcp′*)
and *red*: *?red i loc e″ xs″ i* [] *0* (*pc′* − *length* (*compE2 a*)) *None xcp′* **by** *fastforce*
from *bisim′*
have *P*,*a*•*compareAndSwap*(*D*•*F*, *i*, *e*),*h′* ⊢ (*Val v*•*compareAndSwap*(*D*•*F*, *e″*, *e*), *xs″*) ↔ (*stk″* @ [*v*], *loc′*, *length* (*compE2 a*) + (*pc′* − *length* (*compE2 a*)), *xcp′*)
by(*rule bisim1-bisims1.bisim1CAS2*)
moreover from *red τ* **have** *?red* (*Val v*•*compareAndSwap*(*D*•*F*, *i*, *e*)) *loc* (*Val v*•*compareAndSwap*(*D*•*F*, *e″*, *e*)) *xs″* (*a*•*compareAndSwap*(*D*•*F*, *i*, *e*)) [*v*] (*length* (*compE2 a*)) *pc′ None xcp′*
by(*fastforce intro*: *CAS1Red2 elim*!: *CAS-τred1r-xt2 CAS-τred1t-xt2 split*: *if-split-asm simp add*: *no-call2-def*)
moreover from *exec′* **have** *pc′* ≥ *length* (*compE2 a*)
by(*rule exec-meth-drop-xt-pc*) *auto*
moreover have *no-call2* (*a*•*compareAndSwap*(*D*•*F*, *i*, *e*)) *pc* **using** *pc* **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *τ stk′ pc xcp stk* **by**(*fastforce elim*!: *rtranclp-trans*)
qed
next
case (*bisim1CAS2 i n i′ xs stk loc pc xcp a e D F v*)
note *IH2* = *bisim1CAS2.IH*(*2*)
note *IH3* = *bisim1CAS2.IH*(*6*)
note *exec* = ‹*?exec* (*a*•*compareAndSwap*(*D*•*F*, *i*, *e*)) (*stk* @ [*v*]) *loc* (*length* (*compE2 a*) + *pc*) *xcp stk′ loc′ pc′ xcp′*›
note *bisim2* = ‹*P*,*i*,*h* ⊢ (*i′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim3* = ‹*P*,*e*,*h* ⊢ (*e*, *loc*) ↔ ([], *loc*, *0*, *None*)›
note *len* = ‹*n* + *max-vars* (*Val v*•*compareAndSwap*(*D*•*F*, *i′*, *e*)) ≤ *length xs*›
note *bsok* = ‹*bsok* (*a*•*compareAndSwap*(*D*•*F*, *i*, *e*)) *n*›
from *bisim2* **have** *pc*: *pc* ≤ *length* (*compE2 i*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 i*))
case *True*

from *exec* **have** *exec'*: *exec-meth-d* (*compP2 P*) (*compE2 a* @ *compE2 i* @ *compE2 e* @ [*CAS F D*]) (*compxE2 a 0 0* @ *shift* (*length* (*compE2 a*)) (*stack-xlift* (*length* [*v*]) (*compxE2 i 0 0*) @ *shift* (*length* (*compE2 i*)) (*compxE2 e 0* (*Suc* (*Suc 0*))))) *t h* (*stk* @ [*v*], *loc*, *length* (*compE2 a*) + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *shift-compxE2 stack-xlift-compxE2 ac-simps exec-move-def*)
hence *exec-meth-d* (*compP2 P*) (*compE2 i* @ *compE2 e* @ [*CAS F D*]) (*stack-xlift* (*length* [*v*]) (*compxE2 i 0 0*) @ *shift* (*length* (*compE2 i*)) (*compxE2 e 0* (*Suc* (*Suc 0*)))) *t h* (*stk* @ [*v*], *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 a*), *xcp'*)
by(*rule exec-meth-drop-xt*) *auto*
hence *exec-meth-d* (*compP2 P*) (*compE2 i*) (*stack-xlift* (*length* [*v*]) (*compxE2 i 0 0*)) *t h* (*stk* @ [*v*], *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 a*), *xcp'*)
using *True* **by**(*rule exec-meth-take-xt*)
with *bisim2* **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ [*v*]
and *exec''*: *exec-move-d P t i h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 a*), *xcp'*)
unfolding *exec-move-def* **by**(*blast dest*: *exec-meth-stk-split*)
with *True* **have** *τ*: *τmove2* (*compP2 P*) *h* (*stk* @ [*v*]) (*a*·*compareAndSwap*(*D*·*F*, *i*, *e*)) (*length* (*compE2 a*) + *pc*) *xcp* = *τmove2* (*compP2 P*) *h stk i pc xcp*
by(*auto simp add*: *τmove2-iff τinstr-stk-drop-exec-move*)
moreover from ‹*P*,*h* ⊢ *stk* @ [*v*] [:≤] *ST*› **obtain** *ST2* **where** *P*,*h* ⊢ *stk* [:≤] *ST2* **by**(*auto simp add*: *list-all2-append1*)
from *IH2*[*OF exec'' - - this* ‹*conf-xcp'* (*compP2 P*) *h xcp*›] *len bsok* **obtain** *e'' xs''*
where *bisim'*: *P*,*i*,*h'* ⊢ (*e''*, *xs''*) ↔ (*stk''*, *loc'*, *pc'* − *length* (*compE2 a*), *xcp'*)
and *red*: *?red i' xs e'' xs'' i stk pc* (*pc'* − *length* (*compE2 a*)) *xcp xcp'* **by** *fastforce*
from *bisim'*
have *P*,*a*·*compareAndSwap*(*D*·*F*, *i*, *e*),*h'* ⊢ (*Val v*·*compareAndSwap*(*D*·*F*, *e''*, *e*), *xs''*) ↔ (*stk''* @ [*v*], *loc'*, *length* (*compE2 a*) + (*pc'* − *length* (*compE2 a*)), *xcp'*)
by(*rule bisim1-bisims1.bisim1CAS2*)
moreover from *exec'* **have** *pc'* ≥ *length* (*compE2 a*)
by(*rule exec-meth-drop-xt-pc*) *auto*
moreover have *no-call2 i pc* ⟹ *no-call2* (*a*·*compareAndSwap*(*D*·*F*, *i*, *e*)) (*length* (*compE2 a*) + *pc*) **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *red τ stk' True*
by(*fastforce intro*: *CAS1Red2 elim*!: *CAS-τred1r-xt2 CAS-τred1t-xt2 split*: *if-split-asm*)
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compE2 i*) **by** *simp*
with *bisim2* **obtain** *v2* **where** *i'*: *is-val i'* ⟶ *i'* = *Val v2*
and *stk*: *stk* = [*v2*] **and** *xcp*: *xcp* = *None* **and** *call*: *call1 i'* = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim2 pc len bsok* **have** *red*: *τred1r P t h* (*i'*, *xs*) (*Val v2*, *loc*)
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *τred1r P t h* (*Val v*·*compareAndSwap*(*D*·*F*, *i'*, *e*), *xs*) (*Val v*·*compareAndSwap*(*D*·*F*, *Val v2*, *e*), *loc*) **by**(*rule CAS-τred1r-xt2*)
moreover from *pc exec stk xcp*
have *exec'*: *exec-meth-d* (*compP2 P*) ((*compE2 a* @ *compE2 i*) @ *compE2 e* @ [*CAS F D*]) ((*compxE2 a 0 0* @ *compxE2 i* (*length* (*compE2 a*)) (*Suc 0*)) @ *shift* (*length* (*compE2 a* @ *compE2 i*)) (*stack-xlift* (*length* [*v2*, *v*]) (*compxE2 e 0 0*))) *t h* ([] @ [*v2*, *v*], *loc*, *length* (*compE2 a* @ *compE2 i*) + *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *compxE2-size-convs compxE2-stack-xlift-convs exec-move-def*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e* @ [*CAS F D*]) (*stack-xlift* (*length* [*v2*, *v*]) (*compxE2 e 0 0*)) *t h* ([] @ [*v2*, *v*], *loc*, *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 a* @ *compE2 i*), *xcp'*)
by(*rule exec-meth-drop-xt*) *auto*
hence *exec-meth-d* (*compP2 P*) (*compE2 e*) (*stack-xlift* (*length* [*v2*, *v*]) (*compxE2 e 0 0*)) *t h* ([] @ [*v2*, *v*], *loc*, *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 a* @ *compE2 i*), *xcp'*)

by(*rule exec-meth-take*) *simp*
with *bisim3* **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ [*v2*, *v*]
and *exec''*: *exec-move-d P t e h* ([], *loc*, *0*, *None*) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 a* @ *compE2 i*), *xcp'*)
unfolding *exec-move-def* **by**(*blast dest*: *exec-meth-stk-split*)
with *pc xcp* **have** *τ*: *τmove2* (*compP2 P*) *h* [*v2*, *v*] (*a*•*compareAndSwap*(*D*•*F*, *i*, *e*)) (*length* (*compE2 a*) + *length* (*compE2 i*)) *None* = *τmove2* (*compP2 P*) *h* [] *e 0 None*
using *τinstr-stk-drop-exec-move*[**where** *stk*=[] **and** *vs*=[*v2*, *v*]] **by**(*simp add*: *τmove2-iff*)
from *bisim2* **have** *length xs* = *length loc* **by**(*rule bisim1-length-xs*)
with *IH3*[*OF exec''*, *of* []] *len bsok* **obtain** *e''* *xs''*
where *bisim'*: *P*,*e*,*h'* ⊢ (*e''*, *xs''*) ↔ (*stk''*, *loc'*, *pc'* − *length* (*compE2 a*) − *length* (*compE2 i*), *xcp'*)
and *red*: *?red e loc e'' xs'' e* [] *0* (*pc'* − *length* (*compE2 a*) − *length* (*compE2 i*)) *None xcp'*
by *auto* (*fastforce simp only*: *length-append diff-diff-left*)
from *bisim'*
have *P*,*a*•*compareAndSwap*(*D*•*F*, *i*, *e*),*h'* ⊢ (*Val v*•*compareAndSwap*(*D*•*F*, *Val v2*, *e''*), *xs''*) ↔ (*stk''* @ [*v2*, *v*], *loc'*, *length* (*compE2 a*) + *length* (*compE2 i*) + (*pc'* − *length* (*compE2 a*) − *length* (*compE2 i*)), *xcp'*)
by(*rule bisim1-bisims1.bisim1CAS3*)
moreover from *red τ*
have *?red* (*Val v*•*compareAndSwap*(*D*•*F*, *Val v2*, *e*)) *loc* (*Val v*•*compareAndSwap*(*D*•*F*, *Val v2*, *e''*)) *xs''* (*a*•*compareAndSwap*(*D*•*F*, *i*, *e*)) [*v2*, *v*] (*length* (*compE2 a*) + *length* (*compE2 i*)) *pc' None xcp'*
by(*fastforce intro*: *CAS1Red3 elim*!: *CAS-τred1r-xt3 CAS-τred1t-xt3 split*: *if-split-asm simp add*: *no-call2-def*)
moreover from *exec'* **have** *pc'* ≥ *length* (*compE2 a* @ *compE2 i*)
by(*rule exec-meth-drop-xt-pc*) *auto*
moreover have *no-call2* (*a*•*compareAndSwap*(*D*•*F*, *i*, *e*)) (*length* (*compE2 a*) + *pc*) **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *τ stk' pc xcp stk* **by**(*fastforce elim*!: *rtranclp-trans*)
qed
next
case (*bisim1CAS3 e n e' xs stk loc pc xcp a i D F v v'*)
note *IH3* = *bisim1CAS3.IH*(*2*)
note *exec* = ‹*?exec* (*a*•*compareAndSwap*(*D*•*F*, *i*, *e*)) (*stk* @ [*v'*, *v*]) *loc* (*length* (*compE2 a*) + *length* (*compE2 i*) + *pc*) *xcp stk' loc' pc' xcp'*›
note *bisim3* = ‹*P*,*e*,*h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *len* = ‹*n* + *max-vars* - ≤ *length xs*›
note *bsok* = ‹*bsok* (*a*•*compareAndSwap*(*D*•*F*, *i*, *e*)) *n*›
from ‹*P*,*h* ⊢ *stk* @ [*v'*, *v*] [:≤] *ST*› **obtain** *T T' ST'*
where [*simp*]: *ST* = *ST'* @ [*T'*, *T*]
and *wtv*: *P*,*h* ⊢ *v* :≤ *T* **and** *wtv'*: *P*,*h* ⊢ *v'* :≤ *T'* **and** *ST'*: *P*,*h* ⊢ *stk* [:≤] *ST'*
by(*auto simp add*: *list-all2-Cons1 list-all2-append1*)
from *bisim3* **have** *pc*: *pc* ≤ *length* (*compE2 e*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e*))
case *True*
from *exec* **have** *exec'*: *exec-meth-d* (*compP2 P*) ((*compE2 a* @ *compE2 i*) @ *compE2 e* @ [*CAS F D*]) ((*compxE2 a 0 0* @ *compxE2 i* (*length* (*compE2 a*)) (*Suc 0*)) @ *shift* (*length* (*compE2 a* @ *compE2 i*)) (*stack-xlift* (*length* [*v'*, *v*]) (*compxE2 e 0 0*))) *t h* (*stk* @ [*v'*, *v*], *loc*, *length* (*compE2 a* @ *compE2 i*) + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *shift-compxE2 stack-xlift-compxE2 exec-move-def*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e* @ [*CAS F D*]) (*stack-xlift* (*length* [*v'*, *v*]) (*compxE2 e 0 0*)) *t h* (*stk* @ [*v'*, *v*], *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 a* @ *compE2 i*), *xcp'*)
by(*rule exec-meth-drop-xt*) *auto*

hence *exec-meth-d* (*compP2 P*) (*compE2 e*) (*stack-xlift* (*length* [*v′*, *v*]) (*compxE2 e 0 0*)) *t h* (*stk* @ [*v′*, *v*], *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′* − *length* (*compE2 a* @ *compE2 i*), *xcp′*)
using *True* **by**(*rule exec-meth-take*)
with *bisim3* **obtain** *stk″* **where** *stk′*: *stk′* = *stk″* @ [*v′*, *v*]
and *exec″*: *exec-move-d P t e h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk″*, *loc′*, *pc′* − *length* (*compE2 a* @ *compE2 i*), *xcp′*)
unfolding *exec-move-def* **by**(*blast dest*: *exec-meth-stk-split*)
with *True* **have** *τ*: *τmove2* (*compP2 P*) *h* (*stk* @ [*v′*, *v*]) (*a*•*compareAndSwap*(*D*•*F*, *i*, *e*)) (*length* (*compE2 a*) + *length* (*compE2 i*) + *pc*) *xcp* = *τmove2* (*compP2 P*) *h stk e pc xcp*
by(*auto simp add*: *τmove2-iff τinstr-stk-drop-exec-move*)
moreover from *IH3*[*OF exec″ - - ST′* ‹*conf-xcp′* (*compP2 P*) *h xcp*›] *len bsok* **obtain** *e″ xs″*
where *bisim′*: *P*,*e*,*h′* ⊢ (*e″*, *xs″*) ↔ (*stk″*, *loc′*, *pc′* − *length* (*compE2 a*) − *length* (*compE2 i*), *xcp′*)
and *red*: *?red e′ xs e″ xs″ e stk pc* (*pc′* − *length* (*compE2 a*) − *length* (*compE2 i*)) *xcp xcp′*
by *auto*(*fastforce simp only*: *length-append diff-diff-left*)
from *bisim′*
have *P*,*a*•*compareAndSwap*(*D*•*F*, *i*, *e*),*h′* ⊢ (*Val v*•*compareAndSwap*(*D*•*F*, *Val v′*, *e″*), *xs″*) ↔ (*stk″* @ [*v′*, *v*], *loc′*, *length* (*compE2 a*) + *length* (*compE2 i*) + (*pc′* − *length* (*compE2 a*) − *length* (*compE2 i*)), *xcp′*)
by(*rule bisim1-bisims1.bisim1CAS3*)
moreover from *exec′* **have** *pc′* ≥ *length* (*compE2 a* @ *compE2 i*)
by(*rule exec-meth-drop-xt-pc*) *auto*
moreover have *no-call2 e pc* ⟹ *no-call2* (*a*•*compareAndSwap*(*D*•*F*, *i*, *e*)) (*length* (*compE2 a*) + *length* (*compE2 i*) + *pc*)
by(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *red τ stk′ True*
by(*fastforce intro*: *CAS1Red3 elim*!: *CAS-τred1r-xt3 CAS-τred1t-xt3 split*: *if-split-asm*)
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compE2 e*) **by** *simp*
with *bisim3* **obtain** *v2* **where** *stk*: *stk* = [*v2*] **and** *xcp*: *xcp* = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim3 pc len bsok* **have** *red*: *τred1r P t h* (*e′*, *xs*) (*Val v2*, *loc*)
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *τred1r P t h* (*Val v*•*compareAndSwap*(*D*•*F*, *Val v′*, *e′*), *xs*) (*Val v*•*compareAndSwap*(*D*•*F*, *Val v′*, *Val v2*), *loc*) **by**(*rule CAS-τred1r-xt3*)
moreover have *τ*: ¬ *τmove2* (*compP2 P*) *h* [*v2*, *v′*, *v*] (*a*•*compareAndSwap*(*D*•*F*, *i*, *e*)) (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*)) *None*
by(*simp add*: *τmove2-iff*)
moreover
have ∃ *ta′ e″*. *P*,*a*•*compareAndSwap*(*D*•*F*, *i*, *e*),*h′* ⊢ (*e″*,*loc*) ↔ (*stk′*, *loc′*, *pc′*, *xcp′*) ∧ *True*,*P*,*t* ⊢*1* ⟨*Val v*•*compareAndSwap*(*D*•*F*, *Val v′*, *Val v2*), (*h*, *loc*)⟩ −*ta′*→ ⟨*e″*,(*h′*, *loc*)⟩ ∧ *ta-bisim wbisim1* (*extTA2J1 P ta′*) *ta*
proof(*cases v* = *Null*)
case *True* **with** *exec stk xcp* **show** *?thesis*
by(*fastforce elim*!: *exec-meth.cases simp add*: *exec-move-def intro*: *bisim1CASFail CAS1Null*)
next
case *False*
have *P*,*a*•*compareAndSwap*(*D*•*F*, *i*, *e*),*h′* ⊢ (*Val* (*Bool b*), *loc*) ↔ ([*Bool b*], *loc*, *length* (*compE2* (*a*•*compareAndSwap*(*D*•*F*, *i*, *e*))), *None*) **for** *b*
by(*rule bisim1Val2*) *simp*
with *False exec stk xcp* **show** *?thesis*
by (*auto elim*!: *exec-meth.cases simp add*: *exec-move-def is-Ref-def intro*: *Red1CASSucceed Red1CASFail*)

```
      (fastforce intro!: Red1CASSucceed Red1CASFail simp add: ta-bisim-def ac-simps)+
    qed
    ultimately show ?thesis using exec xcp stk by(fastforce simp add: no-call2-def)
  qed
next
  case (bisim1CASThrow1 A n a xs stk loc pc i e D F)
  note exec = ‹?exec (A·compareAndSwap(D·F, i, e)) stk loc pc ⌊a⌋ stk' loc' pc' xcp'›
  note bisim1 = ‹P,A,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
  from bisim1 have pc: pc < length (compE2 A) by(auto dest: bisim1-ThrowD)
  from bisim1 have match-ex-table (compP2 P) (cname-of h a) (0 + pc) (compxE2 A 0 0) = None
    unfolding compP2-def by(rule bisim1-xcp-Some-not-caught)
  with exec pc have False
    by(auto elim!: exec-meth.cases simp add: match-ex-table-not-pcs-None exec-move-def)
  thus ?case ..
next
  case (bisim1CASThrow2 i n a xs stk loc pc A e D F v)
  note exec = ‹?exec (A·compareAndSwap(D·F, i, e)) (stk @ [v]) loc (length (compE2 A) + pc) ⌊a⌋
stk' loc' pc' xcp'›
  note bisim2 = ‹P,i,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
  from bisim2 have pc: pc < length (compE2 i) by(auto dest: bisim1-ThrowD)
  from bisim2 have match-ex-table (compP2 P) (cname-of h a) (length (compE2 A) + pc) (compxE2
i (length (compE2 A)) 0) = None
    unfolding compP2-def by(rule bisim1-xcp-Some-not-caught)
  with exec pc have False
   apply(auto elim!: exec-meth.cases simp add: compxE2-stack-xlift-convs compxE2-size-convs exec-move-def)
    apply(auto simp add: match-ex-table-append-not-pcs)
    done
  thus ?case ..
next
  case (bisim1CASThrow3 e n a xs stk loc pc A i D F v' v)
  note exec = ‹?exec (A·compareAndSwap(D·F, i, e)) (stk @ [v', v]) loc (length (compE2 A) + length
(compE2 i) + pc) ⌊a⌋ stk' loc' pc' xcp'›
  note bisim2 = ‹P,e,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
 from bisim2 have match-ex-table (compP2 P) (cname-of h a) (length (compE2 A) + length (compE2
i) + pc) (compxE2 e (length (compE2 A) + length (compE2 i)) 0) = None
    unfolding compP2-def by(rule bisim1-xcp-Some-not-caught)
  with exec have False
   apply(auto elim!: exec-meth.cases simp add: compxE2-stack-xlift-convs compxE2-size-convs exec-move-def)
     apply(auto dest!: match-ex-table-stack-xliftD match-ex-table-shift-pcD dest: match-ex-table-pcsD
simp add: match-ex-table-append match-ex-table-shift-pc-None)
    done
  thus ?case ..
next
  case (bisim1CASFail a n i e D F ad xs v' v v'')
   note exec = ‹?exec (a·compareAndSwap(D·F, i, e)) [v', v, v''] xs (length (compE2 a) + length
(compE2 i) + length (compE2 e)) ⌊ad⌋ stk' loc' pc' xcp'›
  hence False
    by(auto elim!: exec-meth.cases simp add: match-ex-table-append exec-move-def
          dest!: match-ex-table-shift-pcD match-ex-table-pc-length-compE2)
  thus ?case ..
next
  case (bisim1Call1 obj n obj' xs stk loc pc xcp ps M')
  note IH1 = bisim1Call1.IH(2)
  note IH2 = bisim1Call1.IH(4)
```

note *exec* = ‹*?exec* (*obj*·*M′*(*ps*)) *stk loc pc xcp stk′ loc′ pc′ xcp′*›
note *bisim1* = ‹*P*,*obj*,*h* ⊢ (*obj′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim2* = ‹*P*,*ps*,*h* ⊢ (*ps*, *loc*) [↔] ([], *loc*, *0*, *None*)›
note *len* = ‹*n* + *max-vars* (*obj′*·*M′*(*ps*)) ≤ *length xs*›
note *bsok* = ‹*bsok* (*obj*·*M′*(*ps*)) *n*›
from *bisim1* **have** *pc*: *pc* ≤ *length* (*compE2 obj*) **by**(*rule bisim1-pc-length-compE2*)
from *bisim1* **have** *lenxs*: *length xs* = *length loc* **by**(*rule bisim1-length-xs*)
show *?case*
proof(*cases pc* < *length* (*compE2 obj*))
case *True*
with *exec* **have** *exec′*: *?exec obj stk loc pc xcp stk′ loc′ pc′ xcp′* **by**(*simp add*: *exec-move-Call1*)
from *True* **have** *τmove2* (*compP2 P*) *h stk* (*obj*·*M′*(*ps*)) *pc xcp* = *τmove2* (*compP2 P*) *h stk obj pc xcp* **by**(*simp add*: *τmove2-iff*)
moreover from *True* **have** *no-call2* (*obj*·*M′*(*ps*)) *pc* = *no-call2 obj pc* **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis*
using *IH1*[*OF exec′* - - ‹*P*,*h* ⊢ *stk* [:≤] *ST*› ‹*conf-xcp′* (*compP2 P*) *h xcp*›] *bisim2 len bsok*
by(*fastforce intro*: *bisim1-bisims1.bisim1Call1 Call1Obj elim*!: *Call-τred1r-obj Call-τred1t-obj*)
next
case *False*
with *pc* **have** *pc*: *pc* = *length* (*compE2 obj*) **by** *auto*
with *bisim1* **obtain** *v* **where** *stk*: *stk* = [*v*] **and** *xcp*: *xcp* = *None* **and** *call*: *call1 obj′* = *None*
and *v*: *is-val obj′* ⟹ *obj′* = *Val v* ∧ *xs* = *loc*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim1 pc len bsok* **have** *τred1r P t h* (*obj′*, *xs*) (*Val v*, *loc*)
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *red*: *τred1r P t h* (*obj′*·*M′*(*ps*), *xs*) (*Val v*·*M′*(*ps*), *loc*) **by**(*rule Call-τred1r-obj*)
show *?thesis*
proof(*cases ps*)
case (*Cons p ps′*)
from *pc exec stk xcp*
have *exec-meth-d* (*compP2 P*) (*compE2* (*obj*·*M′*(*ps*))) (*compxE2* (*obj*·*M′*(*ps*)) *0 0*) *t h* ([] @ [*v*], *loc*, *length* (*compE2 obj*) + *0*, *None*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *compxE2-size-convs compxE2-stack-xlift-convs exec-move-def*)
hence *exec′*: *exec-meth-d* (*compP2 P*) (*compEs2 ps*) (*stack-xlift* (*length* [*v*]) (*compxEs2 ps 0 0*)) *t h* ([] @ [*v*], *loc*, *0*, *None*) *ta h′* (*stk′*, *loc′*, *pc′* − *length* (*compE2 obj*), *xcp′*)
and *pc′*: *pc′* ≥ *length* (*compE2 obj*) **using** *Cons*
by(*safe dest*!: *Call-execParamD*) *simp-all*
from *exec′ bisim2* **obtain** *stk″* **where** *stk′*: *stk′* = *stk″* @ [*v*]
and *exec″*: *exec-moves-d P t ps h* ([], *loc*, *0*, *None*) *ta h′* (*stk″*, *loc′*, *pc′* − *length* (*compE2 obj*), *xcp′*)
unfolding *exec-moves-def* **by** −(*drule* (*1*) *exec-meth-stk-splits*, *auto*)
with *pc xcp* **have** *τ*: *τmove2* (*compP2 P*) *h* [*v*] (*obj*·*M′*(*ps*)) (*length* (*compE2 obj*)) *None* = *τmoves2* (*compP2 P*) *h* [] *ps 0 None*
using *τinstr-stk-drop-exec-moves*[**where** *stk*=[] **and** *vs*=[*v*]]
by(*auto simp add*: *τmove2-iff τmoves2-iff*)
from *IH2*[*OF exec″*, *of* []] *len lenxs bsok* **obtain** *es″ xs″*
where *bisim′*: *P*,*ps*,*h′* ⊢ (*es″*, *xs″*) [↔] (*stk″*, *loc′*, *pc′* − *length* (*compE2 obj*), *xcp′*)
and *red′*: *?reds ps loc es″ xs″ ps* [] *0* (*pc′* − *length* (*compE2 obj*)) *None xcp′* **by** *auto*
from *bisim′* **have** *P*,*obj*·*M′*(*ps*),*h′* ⊢ (*Val v*·*M′*(*es″*), *xs″*) ↔ (*stk″* @ [*v*], *loc′*, *length* (*compE2 obj*) + (*pc′* − *length* (*compE2 obj*)), *xcp′*)
by(*rule bisim1CallParams*)
moreover from *pc Cons* **have** *no-call2* (*obj*·*M′*(*ps*)) *pc* **by**(*simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *red red′ τ stk′ pc xcp pc′ stk call*
by(*fastforce elim*!: *rtranclp-trans Call-τred1r-param Call-τred1t-param intro*: *Call1Params*

rtranclp-tranclp-tranclp split: *if-split-asm*)
next
case [*simp*]: *Nil*
from *exec pc stk xcp*
have $v = Null \vee (is\text{-}Addr\ v \wedge (\exists\ T\ C'\ Ts'\ Tr'\ D'.\ typeof_h\ v = \lfloor T \rfloor \wedge class\text{-}type\text{-}of'\ T = \lfloor C' \rfloor \wedge P \vdash C'\ sees\ M':Ts' \to Tr' = Native\ in\ D'))$ (**is** - $\vee$ *?rest*)
by(*fastforce elim*!: *exec-meth.cases simp add*: *is-Ref-def exec-move-def compP2-def has-method-def split*: *if-split-asm*)
thus *?thesis*
proof
assume [*simp*]: $v = Null$
have $P,obj{\cdot}M'([]),h \vdash (THROW\ NullPointer, loc) \leftrightarrow ([] @ [Null], loc, length\ (compE2\ obj) + length\ (compEs2\ ([]\ ::\ 'addr\ expr1\ list)), \lfloor addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer \rfloor)$
by(*safe intro*!: *bisim1CallThrow*) *simp-all*
moreover have $True,P,t \vdash 1\ \langle null{\cdot}M'(map\ Val\ []),(h, loc)\rangle\ -\varepsilon\to\ \langle THROW\ NullPointer,(h, loc)\rangle$
by(*rule Red1CallNull*)
moreover have $\tau move1\ P\ h\ (Val\ v{\cdot}M'([]))\ \tau move2\ (compP2\ P)\ h\ [Null]\ (obj{\cdot}M'(ps))\ (length\ (compE2\ obj))\ None$
by(*simp-all add*: $\tau move2\text{-}iff$)
ultimately show *?thesis* **using** *exec pc stk xcp red*
by(*fastforce elim*!: *exec-meth.cases intro*: *rtranclp-trans simp add*: *exec-move-def*)
next
assume *?rest*
then obtain *a Ta Ts' Tr' D'* **where** [*simp*]: $v = Addr\ a$
and *Ta*: *typeof-addr h a* = $\lfloor Ta \rfloor$
and *iec*: $P \vdash class\text{-}type\text{-}of\ Ta\ sees\ M'\text{: } Ts' \to Tr' = Native\ in\ D'$ **by** *auto*
with *exec pc stk xcp*
obtain *ta' va h'' U* **where** *redex*: $(ta', va, h'') \in red\text{-}external\text{-}aggr\ (compP2\ P)\ t\ a\ M'\ []\ h$
and *ret*: $(xcp', h', [(stk', loc', undefined, undefined, pc')]) = extRet2JVM\ 0\ h''\ [Addr\ a]\ loc\ undefined\ undefined\ (length\ (compE2\ obj))\ []\ va$
and *wtext'*: $P,h \vdash a{\cdot}M'([]) : U$
and *ta'*: $ta = extTA2JVM\ (compP2\ P)\ ta'$
by(*fastforce simp add*: *is-Ref-def exec-move-def compP2-def external-WT'-iff exec-meth-instr*)
from *Ta iec* **have** τ: $\tau move1\ P\ h\ (Val\ v{\cdot}M'([])) \longleftrightarrow \tau move2\ (compP2\ P)\ h\ [Addr\ a]\ (obj{\cdot}M'(ps))\ (length\ (compE2\ obj))\ None$
by(*auto simp add*: $\tau move2\text{-}iff$ *compP2-def*)
from *ret* **have** [*simp*]: $h'' = h'$ **by** *simp*
from *wtext'* **have** *wtext''*: $compP2\ P,h \vdash a{\cdot}M'([]) : U$ **by**(*simp add*: *external-WT'-iff compP2-def*)
from *wf* **have** *wf-jvm-prog* (*compP2 P*) **by**(*rule wt-compP2*)
then obtain *wf-md* **where** *wf'*: *wf-prog wf-md* (*compP2 P*)
unfolding *wf-jvm-prog-def* **by**(*blast dest*: *wt-jvm-progD*)
from *tconf* **have** *tconf'*: $compP2\ P,h \vdash t\ \surd t$ **by**(*simp add*: *compP2-def tconf-def*)
from *redex* **have** *redex''*: $compP2\ P,t \vdash \langle a{\cdot}M'([]), h\rangle\ -ta'\to ext\ \langle va, h'\rangle$
by(*simp add*: *WT-red-external-list-conv*[*OF wf' wtext'' tconf'*, *symmetric*])
hence *redex'*: $P,t \vdash \langle a{\cdot}M'([]), h\rangle\ -ta'\to ext\ \langle va, h'\rangle$ **by**(*simp add*: *compP2-def*)
with *Ta iec* **have** $True,P,t \vdash 1\ \langle addr\ a{\cdot}M'(map\ Val\ []), (h, loc)\rangle\ -ta'\to\ \langle extRet2J\ (addr\ a{\cdot}M'(map\ Val\ []))\ va, (h', loc)\rangle$
by −(*rule Red1CallExternal*, *auto*)
moreover have $P,obj{\cdot}M'(ps),h' \vdash (extRet2J\ (addr\ a{\cdot}M'(map\ Val\ []))\ va, loc) \leftrightarrow (stk', loc', pc', xcp')$
proof(*cases va*)
case (*RetVal v*)
have $P,obj{\cdot}M'([]),h' \vdash (Val\ v, loc) \leftrightarrow ([v], loc, length\ (compE2\ (obj{\cdot}M'([]))), None)$

```
          by(rule bisim1Val2) simp
        with ret RetVal show ?thesis by simp
      next
        case (RetExc ad)
          have P,obj·M'([]),h' ⊢ (Throw ad, loc) ↔ ([] @ [v], loc, length (compE2 (obj)) + length
(compEs2 ([] :: 'addr expr1 list)), ⌊ad⌋)
          by(rule bisim1CallThrow) simp
        with ret RetExc show ?thesis by simp
      next
        case RetStaySame
        have P,obj·M'([]),h' ⊢ (addr a·M'([]), loc) ↔ ([Addr a], loc, length (compE2 obj), None)
          by(rule bisim1-bisims1.bisim1Call1)(rule bisim1Val2, simp)
        thus ?thesis using ret RetStaySame by simp
      qed
       moreover from ‹preallocated h› red-external-hext[OF redex'] have preallocated h' by(rule
preallocated-hext)
      from redex'' wtext'' ‹hconf h› have hconf h' by(rule external-call-hconf)
      with ta' redex' ‹preallocated h'›
      have ta-bisim wbisim1 (extTA2J1 P ta') ta by(auto intro: red-external-ta-bisim21[OF wf])
      moreover have τmove1 P h (Val v·M'([])) ⟹ ta' = ε ∧ h' = h using redex' Ta iec
        by(fastforce dest: τexternal'-red-external-heap-unchanged τexternal'-red-external-TA-empty
sees-method-fun simp add: τexternal'-def τexternal-def)
      moreover from v call
      have call1 (obj'·M'(ps)) ≠ None ⟹ Val v·M'(ps) = obj'·M'(ps) ∧ loc = xs
        by(auto split: if-split-asm)
      ultimately show ?thesis using red τ pc xcp stk Ta call iec
        apply(auto simp del: call1.simps calls1.simps)
       apply(rule exI conjI|assumption|erule rtranclp.rtrancl-into-rtrancl rtranclp-into-tranclp1|drule
(1) sees-method-fun|clarsimp)+
        done
    qed
  qed
qed
next
  case (bisim1CallParams ps n ps' xs stk loc pc xcp obj M' v)
  note bisim2 = ‹P,ps,h ⊢ (ps', xs) [↔] (stk, loc, pc, xcp)›
  note bisim1 = ‹P,obj,h ⊢ (obj, xs) ↔ ([], xs, 0, None)›
  note IH2 = bisim1CallParams.IH(2)
  note exec = ‹exec-move-d P t (obj·M'(ps)) h (stk @ [v], loc, length (compE2 obj) + pc, xcp) ta h'
(stk', loc', pc', xcp')›
  note len = ‹n + max-vars (Val v·M'(ps')) ≤ length xs›
  note bsok = ‹bsok (obj·M'(ps)) n›
  from ‹P,h ⊢ stk @ [v] [:≤] ST› obtain T ST' where ST': P,h ⊢ stk [:≤] ST' and T: typeof_h v =
⌊T⌋
    by(auto simp add: list-all2-Cons1 list-all2-append1 conf-def)
  from bisim2 have pc: pc ≤ length (compEs2 ps) by(rule bisims1-pc-length-compEs2)
  show ?case
  proof(cases pc < length (compEs2 ps))
    case True
    from exec have exec-meth-d (compP2 P) (compE2 (obj·M'(ps))) (compxE2 (obj·M'(ps)) 0 0) t h
(stk @ [v], loc, length (compE2 obj) + pc, xcp) ta h' (stk', loc', pc', xcp')
      by(simp add: exec-move-def)
    with True have exec': exec-meth-d (compP2 P) (compEs2 ps) (stack-xlift (length [v]) (compxEs2
ps 0 0)) t h (stk @ [v], loc, pc, xcp) ta h' (stk', loc', pc' − length (compE2 obj), xcp')
```

and *pc′*: *pc′* ≥ *length* (*compE2 obj*) **by**(*safe dest*!: *Call-execParamD*)
from *exec′ bisim2* **obtain** *stk″* **where** *stk′*: *stk′* = *stk″* @ [*v*]
and *exec″*: *exec-moves-d P t ps h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk″*, *loc′*, *pc′* − *length* (*compE2 obj*), *xcp′*)
by(*unfold exec-moves-def*)(*drule* (*1*) *exec-meth-stk-splits*, *auto*)
with *True* **have** *τ*: *τmove2* (*compP2 P*) *h* (*stk* @ [*v*]) (*obj*·*M′*(*ps*)) (*length* (*compE2 obj*) + *pc*) *xcp* = *τmoves2* (*compP2 P*) *h stk ps pc xcp*
by(*auto simp add*: *τmove2-iff τmoves2-iff τinstr-stk-drop-exec-moves*)
from *IH2*[*OF exec″ - - ST′* ‹*conf-xcp′* (*compP2 P*) *h xcp*›] *len bsok* **obtain** *es″ xs″*
where *bisim′*: *P*,*ps*,*h′* ⊢ (*es″*, *xs″*) [↔] (*stk″*, *loc′*, *pc′* − *length* (*compE2 obj*), *xcp′*)
and *red′*: *?reds ps′ xs es″ xs″ ps stk pc* (*pc′* − *length* (*compE2 obj*)) *xcp xcp′* **by** *auto*
from *bisim′* **have** *P*,*obj*·*M′*(*ps*),*h′* ⊢ (*Val v*·*M′*(*es″*), *xs″*) ↔ (*stk″* @ [*v*], *loc′*, *length* (*compE2 obj*) + (*pc′* − *length* (*compE2 obj*)), *xcp′*)
by(*rule bisim1-bisims1.bisim1CallParams*)
moreover from *True* **have** *no-call2* (*obj*·*M′*(*ps*)) (*length* (*compE2 obj*) + *pc*) = *no-calls2 ps pc*
by(*simp add*: *no-calls2-def no-call2-def*)
moreover have *calls1 ps′* = *None* ⟹ *call1* (*Val v*·*M′*(*ps′*)) = *None* ∨ *is-vals ps′* **by** *simp*
ultimately show *?thesis* **using** *red′ τ stk′ pc pc′*
by(*fastforce intro*: *Call1Params elim*!: *Call-τred1r-param Call-τred1t-param split*: *if-split-asm simp add*: *is-vals-conv*)
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compEs2 ps*) **by** *simp*
with *bisim2* **obtain** *vs* **where** [*simp*]: *stk* = *rev vs xcp* = *None*
and *lenvs*: *length vs* = *length ps* **and** *vs*: *is-vals ps′* ⟶ *ps′* = *map Val vs* ∧ *xs* = *loc*
and *call*: *calls1 ps′* = *None*
by(*auto dest*: *bisims1-pc-length-compEs2D*)
with *bisim2 len bsok* **have** *reds*: *τreds1r P t h* (*ps′*, *xs*) (*map Val vs*, *loc*)
by(*auto intro*: *bisims1-Val-τReds1r simp add*: *bsok-def*)
from *exec T lenvs*
have *v* = *Null* ∨ (*is-Addr v* ∧ (∃ *T C′ Ts′ Tr′ D′*. *typeof*$_h$ *v* = ⌊*T*⌋ ∧ *class-type-of′ T* = ⌊*C′*⌋ ∧ *P* ⊢ *C′ sees M′*:*Ts′*→*Tr′* = *Native in D′*)) (**is** - ∨ *?rest*)
by(*fastforce elim*!: *exec-meth.cases simp add*: *is-Ref-def exec-move-def compP2-def has-method-def split*: *if-split-asm*)
thus *?thesis*
proof
assume [*simp*]: *v* = *Null*
hence *τ*: *τmove1 P h* (*Val v*·*M′*(*map Val vs*))
τmove2 (*compP2 P*) *h* (*stk* @ [*v*]) (*obj*·*M′*(*ps*)) (*length* (*compE2 obj*) + *length* (*compEs2 ps*)) *None*
using *lenvs* **by**(*auto simp add*: *τmove2-iff*)
from *lenvs*
have *P*,*obj*·*M′*(*ps*),*h* ⊢ (*THROW NullPointer*, *loc*) ↔ (*rev vs* @ [*Null*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), ⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*safe intro*!: *bisim1CallThrow*) *simp*
moreover have *True*,*P*,*t* ⊢*1* ⟨*null*·*M′*(*map Val vs*),(*h*, *loc*)⟩ −*ε*→ ⟨*THROW NullPointer*,(*h*, *loc*)⟩
by(*rule Red1CallNull*)
ultimately show *?thesis* **using** *exec pc τ lenvs reds*
apply(*auto elim*!: *exec-meth.cases simp add*: *exec-move-def*)
apply(*rule exI conjI*|*assumption*)+
apply(*rule rtranclp.rtrancl-into-rtrancl*)
apply(*erule Call-τred1r-param*)
by *auto*

next
assume *?rest*
then obtain *a Ta C′ Ts′ Tr′ D′* **where** [*simp*]: *v = Addr a*
and *Ta*: *typeof-addr h a = ⌊Ta⌋*
and *iec*: *P ⊢ class-type-of Ta sees M′: Ts′→Tr′ = Native in D′* **by** *auto*
with *exec pc lenvs*
obtain *ta′ va h′′ U Ts Ts′* **where** *redex*: *(ta′, va, h′′) ∈ red-external-aggr (compP2 P) t a M′ vs h*
and *ret*: *(xcp′, h′, [(stk′, loc′, undefined, undefined, pc′)]) = extRet2JVM (length vs) h′′ (rev vs @ [Addr a]) loc undefined undefined (length (compE2 obj) + length (compEs2 ps)) [] va*
and *wtext′*: *P,h ⊢ a·M′(vs) : U*
and *Ts*: *map typeof_h vs = map Some Ts*
and *ta′*: *ta = extTA2JVM (compP2 P) ta′*
by(*fastforce simp add*: *is-Ref-def exec-move-def compP2-def external-WT′-iff exec-meth-instr confs-conv-map*)
have *τ*: *τmove1 P h (Val v·M′(map Val vs)) ⟷ τmove2 (compP2 P) h (stk @ [v]) (obj·M′(ps)) (length (compE2 obj) + length (compEs2 ps)) None*
using *Ta iec lenvs*
by(*auto simp add*: *τmove2-iff map-eq-append-conv compP2-def*)
from *ret* **have** [*simp*]: *h′′ = h′* **by** *simp*
from *wtext′* **have** *wtext′′*: *compP2 P,h ⊢ a·M′(vs) : U* **by**(*simp add*: *compP2-def external-WT′-iff*)
from *wf* **have** *wf-jvm-prog (compP2 P)* **by**(*rule wt-compP2*)
then obtain *wf-md* **where** *wf′*: *wf-prog wf-md (compP2 P)*
unfolding *wf-jvm-prog-def* **by**(*blast dest*: *wt-jvm-progD*)
from *tconf* **have** *tconf′*: *compP2 P,h ⊢ t √t* **by**(*simp add*: *compP2-def tconf-def*)
from *redex* **have** *redex′′*: *compP2 P,t ⊢ ⟨a·M′(vs), h⟩ −ta′→ext ⟨va, h′⟩*
by(*simp add*: *WT-red-external-list-conv*[*OF wf′ wtext′′ tconf′, symmetric*])
hence *redex′*: *P,t ⊢ ⟨a·M′(vs), h⟩ −ta′→ext ⟨va, h′⟩* **by**(*simp add*: *compP2-def*)
with *Ta iec* **have** *True,P,t ⊢1 ⟨addr a·M′(map Val vs), (h, loc)⟩ −ta′→ ⟨extRet2J (addr a·M′(map Val vs)) va, (h′, loc)⟩*
by −(*rule Red1CallExternal, auto*)
moreover have *P,obj·M′(ps),h′ ⊢ (extRet2J (addr a·M′(map Val vs)) va, loc) ↔ (stk′, loc′, pc′, xcp′)*
proof(*cases va*)
case (*RetVal v*)
have *P,obj·M′(ps),h′ ⊢ (Val v, loc) ↔ ([v], loc, length (compE2 (obj·M′(ps))), None)*
by(*rule bisim1Val2*)(*simp*)
with *ret RetVal* **show** *?thesis* **by** *simp*
next
case (*RetExc ad*)
from *lenvs* **have** *length ps = length (rev vs)* **by** *simp*
hence *P,obj·M′(ps),h′ ⊢ (Throw ad, loc) ↔ (rev vs @ [v], loc, length (compE2 (obj)) + length (compEs2 ps), ⌊ad⌋)*
by(*rule bisim1CallThrow*)
with *ret RetExc* **show** *?thesis* **by** *simp*
next
case *RetStaySame*
from *lenvs* **have** *length ps = length vs* **by** *simp*
from *bisims1-map-Val-append*[*OF bisims1Nil this, of P h′ loc*]
have *P,ps,h′ ⊢ (map Val vs, loc) [↔] (rev vs, loc, length (compEs2 ps), None)* **by** *simp*
hence *P,obj·M′(ps),h′ ⊢ (addr a·M′(map Val vs), loc) ↔ (rev vs @ [Addr a], loc, length (compE2 obj) + length (compEs2 ps), None)*
by(*rule bisim1-bisims1.bisim1CallParams*)
thus *?thesis* **using** *ret RetStaySame* **by** *simp*

```
    qed
    moreover from ‹preallocated h› red-external-hext[OF redex'] have preallocated h' by(rule preal-
located-hext)
    from redex'' wtext'' ‹hconf h› have hconf h' by(rule external-call-hconf)
    with ta' redex' ‹preallocated h'›
    have ta-bisim wbisim1 (extTA2J1 P ta') ta by(auto intro: red-external-ta-bisim21[OF wf])
    moreover have τmove1 P h (Val v·M'(map Val vs)) ⟹ ta' = ε ∧ h' = h
      using Ta iec redex'
        by(fastforce dest: τexternal'-red-external-heap-unchanged τexternal'-red-external-TA-empty
sees-method-fun simp add: τexternal'-def τexternal-def map-eq-append-conv)
    moreover from vs call have call1 (Val v·M'(ps')) ≠ None ⟹ ps' = map Val vs ∧ loc = xs
      by(auto split: if-split-asm simp add: is-vals-conv)
    ultimately show ?thesis using reds τ pc Ta call
      apply(auto simp del: split-paired-Ex call1.simps calls1.simps split: if-split-asm simp add:
map-eq-append-conv)
    apply(auto 4 4 simp del: split-paired-Ex call1.simps calls1.simps intro: rtranclp.rtrancl-into-rtrancl[OF
Call-τred1r-param] rtranclp-into-tranclp1[OF Call-τred1r-param])[3]
      apply((assumption|rule exI conjI|erule Call-τred1r-param|simp add: map-eq-append-conv)+)
      done
  qed
 qed
next
 case (bisim1CallThrowObj obj n a xs stk loc pc ps M')
 note exec = ‹?exec (obj·M'(ps)) stk loc pc ⌊a⌋ stk' loc' pc' xcp'›
 note bisim1 = ‹P,obj,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
 from bisim1 have pc: pc < length (compE2 obj) by(auto dest: bisim1-ThrowD)
 from bisim1 have match-ex-table (compP2 P) (cname-of h a) (0 + pc) (compxE2 obj 0 0) = None
   unfolding compP2-def by(rule bisim1-xcp-Some-not-caught)
 with exec pc have False
   by(auto elim!: exec-meth.cases simp add: exec-move-def match-ex-table-not-pcs-None)
 thus ?case ..
next
 case (bisim1CallThrowParams ps n vs a ps' xs stk loc pc obj M' v)
 note exec = ‹?exec (obj·M'(ps)) (stk @ [v]) loc (length (compE2 obj) + pc) ⌊a⌋ stk' loc' pc' xcp'›
 note bisim2 = ‹P,ps,h ⊢ (map Val vs @ Throw a # ps', xs) [↔] (stk, loc, pc, ⌊a⌋)›
 from bisim2 have pc: pc < length (compEs2 ps) by(auto dest: bisims1-ThrowD)
 from bisim2 have match-ex-table (compP2 P) (cname-of h a) (0 + pc) (compxEs2 ps 0 0) = None
   unfolding compP2-def by(rule bisims1-xcp-Some-not-caught)
 with exec pc have False
  apply(auto elim!: exec-meth.cases simp add: compxEs2-size-convs compxEs2-stack-xlift-convs exec-move-def)
  apply(auto simp add: match-ex-table-append dest!: match-ex-table-shift-pcD match-ex-table-stack-xliftD
match-ex-table-pc-length-compE2)
   done
 thus ?case ..
next
 case (bisim1CallThrow ps vs obj n M' a xs v)
 note lenvs = ‹length ps = length vs›
 note exec = ‹?exec (obj·M'(ps)) (vs @ [v]) xs (length (compE2 obj) + length (compEs2 ps)) ⌊a⌋ stk'
loc' pc' xcp'›
 with lenvs have False
  by(auto elim!: exec-meth.cases simp add: match-ex-table-append-not-pcs exec-move-def dest!: match-ex-table-pc-length-compE
 thus ?case ..
next
 case (bisim1BlockSome1 e n V Ty v xs)
```

have *τmove2 (compP2 P) h [] {V:Ty=⌊v⌋; e} 0 None* **by**(*simp add*: *τmove2-iff*)
with ‹*?exec {V:Ty=⌊v⌋; e} [] xs 0 None stk′ loc′ pc′ xcp′*› ‹*P,e,h ⊢ (e, xs) ↔ ([], xs, 0, None)*›
show *?case* **by**(*fastforce elim*!: *exec-meth.cases intro*: *bisim1BlockSome2 simp add*: *exec-move-def*)
next
case (*bisim1BlockSome2 e n V Ty v xs*)
have *τmove2 (compP2 P) h [v] {V:Ty=⌊v⌋; e} (Suc 0) None* **by**(*simp add*: *τmove2-iff*)
with ‹*?exec {V:Ty=⌊v⌋; e} [v] xs (Suc 0) None stk′ loc′ pc′ xcp′*› ‹*P,e,h ⊢ (e, xs) ↔ ([], xs, 0, None)*›
show *?case* **by**(*fastforce intro*: *r-into-rtranclp Block1Some bisim1BlockSome4*[*OF bisim1-refl*] *simp add*: *exec-move-def*
elim!: *exec-meth.cases*)
next
case (*bisim1BlockSome4 e n e′ xs stk loc pc xcp V Ty v*)
note *IH* = *bisim1BlockSome4.IH*(*2*)
note *exec* = ‹*?exec {V:Ty=⌊v⌋; e} stk loc (Suc (Suc pc)) xcp stk′ loc′ pc′ xcp′*›
note *bisim* = ‹*P,e,h ⊢ (e′, xs) ↔ (stk, loc, pc, xcp)*›
note *bsok* = ‹*bsok {V:Ty=⌊v⌋; e} n*›
with ‹*n + max-vars {V:Ty=None; e′} ≤ length xs*›
have *V*: *V < length xs* **and** *len*: *Suc n + max-vars e′ ≤ length xs* **and** [*simp*]: *n = V* **by** *simp-all*
let *?pre* = [*Push v, Store V*]
from *exec* **have** *exec′*: *exec-meth-d (compP2 P) (?pre @ compE2 e) (shift (length ?pre) (compxE2 e 0 0)) t h (stk, loc, length ?pre + pc, xcp) ta h′ (stk′, loc′, pc′, xcp′)*
by(*simp add*: *compxE2-size-convs exec-move-def*)
hence *pc′*: *pc′ ≥ length ?pre*
by(*rule exec-meth-drop-pc*) *auto*
hence *pc″*: *Suc (Suc (pc′ − Suc (Suc 0))) = pc′* **by** *simp*
moreover from *exec′* **have** *exec-move-d P t e h (stk, loc, pc, xcp) ta h′ (stk′, loc′, pc′ − length ?pre, xcp′)*
unfolding *exec-move-def* **by**(*rule exec-meth-drop*) *auto*
from *IH*[*OF this len - ‹P,h ⊢ stk [:≤] ST› ‹conf-xcp′ (compP2 P) h xcp›*] *bsok* **obtain** *e″ xs″*
where *bisim′*: *P,e,h′ ⊢ (e″, xs″) ↔ (stk′, loc′, pc′ − length ?pre, xcp′)*
and *red′*: *?red e′ xs e″ xs″ e stk pc (pc′ − length ?pre) xcp xcp′* **by** *auto*
from *bisim′* **have** *P,{V:Ty=⌊v⌋; e},h′ ⊢ ({V:Ty=None; e″}, xs″) ↔ (stk′, loc′, Suc (Suc (pc′ − length ?pre)), xcp′)*
by(*rule bisim1-bisims1.bisim1BlockSome4*)
moreover from *pc′ pc″* **have** *τmove2 (compP2 P) h stk {V:Ty=⌊v⌋; e} (Suc (Suc pc)) xcp = τmove2 (compP2 P) h stk e pc xcp* **by**(*simp add*: *τmove2-iff*)
moreover from *red′* **have** *length xs″ = length xs*
by(*auto split*: *if-split-asm dest*!: *τred1r-preserves-len τred1t-preserves-len red1-preserves-len*)
ultimately show *?case* **using** *red′ pc″ V*
by(*fastforce elim*!: *Block-None-τred1r-xt Block-None-τred1t-xt intro*: *Block1Red split*: *if-split-asm simp add*: *no-call2-def*)
next
case (*bisim1BlockThrowSome e n a xs stk loc pc V Ty v*)
note *exec* = ‹*?exec {V:Ty=⌊v⌋; e} stk loc (Suc (Suc pc)) ⌊a⌋ stk′ loc′ pc′ xcp′*›
note *bisim* = ‹*P,e,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)*›
from *bisim* **have** *pc*: *pc < length (compE2 e)* **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim* **have** *match-ex-table (compP2 P) (cname-of h a) (0 + pc) (compxE2 e 0 0) = None*
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)
with *exec pc* **have** *False*
apply(*auto elim*!: *exec-meth.cases simp add*: *match-ex-table-not-pcs-None exec-move-def*)
apply(*auto simp only*: *compxE2-size-convs dest*!: *match-ex-table-shift-pcD*)
apply *simp*
done

```
  thus ?case ..
next
  case (bisim1BlockNone e n e' xs stk loc pc xcp V Ty)
  note IH = bisim1BlockNone.IH(2)
  note exec = ‹?exec {V:Ty=None; e} stk loc pc xcp stk' loc' pc' xcp'›
  note bisim = ‹P,e,h ⊢ (e', xs) ↔ (stk, loc, pc, xcp)›
  note bsok = ‹bsok {V:Ty=None; e} n›
  with ‹n + max-vars {V:Ty=None; e'} ≤ length xs›
  have V: V < length xs and len: Suc n + max-vars e' ≤ length xs by simp-all
  have τmove2 (compP2 P) h stk {V:Ty=None; e} pc xcp = τmove2 (compP2 P) h stk e pc xcp
by(simp add: τmove2-iff)
  moreover
  from exec have ?exec e stk loc pc xcp stk' loc' pc' xcp' by(simp add: exec-move-BlockNone)
  from IH[OF this len - ‹P,h ⊢ stk [:≤] ST› ‹conf-xcp' (compP2 P) h xcp›] bsok obtain e'' xs''
    where bisim': P,e,h' ⊢ (e'', xs'') ↔ (stk', loc', pc', xcp')
    and red': ?red e' xs e'' xs'' e stk pc pc' xcp xcp' by auto
  from bisim' have P,{V:Ty=None; e},h' ⊢ ({V:Ty=None; e''}, xs'') ↔ (stk', loc', pc', xcp')
    by(rule bisim1-bisims1.bisim1BlockNone)
  ultimately show ?case using V red'
    by(fastforce elim!: Block1Red Block-None-τred1r-xt Block-None-τred1t-xt simp add: no-call2-def)
next
  case (bisim1BlockThrowNone e n a xs stk loc pc V Ty)
  note exec = ‹?exec {V:Ty=None; e} stk loc pc ⌊a⌋ stk' loc' pc' xcp'›
  note bisim = ‹P,e,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
  from bisim have pc: pc < length (compE2 e) by(auto dest: bisim1-ThrowD)
  from bisim have match-ex-table (compP2 P) (cname-of h a) (0 + pc) (compxE2 e 0 0) = None
    unfolding compP2-def by(rule bisim1-xcp-Some-not-caught)
  with exec pc have False by(auto elim!: exec-meth.cases simp add: exec-move-def)
  thus ?case ..
next
  case (bisim1Sync1 e1 n e' xs stk loc pc xcp e2 V)
  note IH = bisim1Sync1.IH(2)
  note exec = ‹?exec (sync_V (e1) e2) stk loc pc xcp stk' loc' pc' xcp'›
  note bisim = ‹P,e1,h ⊢ (e', xs) ↔ (stk, loc, pc, xcp)›
  note bisim2 = ‹P,e2,h ⊢ (e2, xs) ↔ ([], xs, 0, None)›
  note len = ‹n + max-vars (sync_V (e') e2) ≤ length xs›
  note bsok = ‹bsok (sync_V (e1) e2) n›
  from bisim have pc: pc ≤ length (compE2 e1) by(rule bisim1-pc-length-compE2)
  show ?case
  proof(cases pc < length (compE2 e1))
    case True
    with exec have exec': ?exec e1 stk loc pc xcp stk' loc' pc' xcp' by(simp add: exec-move-Sync1)
    from True have τmove2 (compP2 P) h stk (sync_V (e1) e2) pc xcp = τmove2 (compP2 P) h stk
e1 pc xcp by(simp add: τmove2-iff)
    with IH[OF exec' - - ‹P,h ⊢ stk [:≤] ST› ‹conf-xcp' (compP2 P) h xcp›] len bisim2 bsok show
?thesis
    by(fastforce intro: bisim1-bisims1.bisim1Sync1 Synchronized1Red1 elim!: Sync-τred1r-xt Sync-τred1t-xt
simp add: no-call2-def)
  next
    case False
    with pc have [simp]: pc = length (compE2 e1) by simp
    with bisim obtain v where stk: stk = [v] and xcp: xcp = None
      by(auto dest: bisim1-pc-length-compE2D)
    with bisim pc len bsok have red: τred1r P t h (e', xs) (Val v, loc)
```

by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *τred1r P t h* (*sync*$_V$ (*e'*) *e2*, *xs*) (*sync*$_V$ (*Val v*) *e2*, *loc*) **by**(*rule Sync-τred1r-xt*)
moreover have *τ*: *τmove2* (*compP2 P*) *h* [*v*] (*sync*$_V$ (*e1*) *e2*) *pc None* **by**(*simp add*: *τmove2-iff*)
moreover
have *P*,(*sync*$_V$ (*e1*) *e2*),*h* ⊢ ((*sync*$_V$ (*Val v*) *e2*), *loc*) ↔ ([*v*, *v*], *loc*, *Suc* (*length* (*compE2 e1*)), *None*)
by(*rule bisim1Sync2*)
ultimately show *?thesis* **using** *exec stk xcp*
by(*fastforce elim*!: *exec-meth.cases simp add*: *exec-move-def*)
qed
next
case (*bisim1Sync2 e1 n e2 V v xs*)
note *exec* = ‹*?exec* (*sync*$_V$ (*e1*) *e2*) [*v*, *v*] *xs* (*Suc* (*length* (*compE2 e1*))) *None stk' loc' pc' xcp'*›
note *bisim* = ‹*P*,*e1*,*h* ⊢ (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*)›
have *τmove2* (*compP2 P*) *h* [*v*, *v*] (*sync*$_V$ (*e1*) *e2*) (*Suc* (*length* (*compE2 e1*))) *None* **by**(*rule τmove2Sync3*)
thus *?case* **using** *exec*
by(*fastforce elim*!: *exec-meth.cases intro*: *bisim1Sync3 simp add*: *exec-move-def*)
next
case (*bisim1Sync3 e1 n e2 V v xs*)
note *exec* = ‹*?exec* (*sync*$_V$ (*e1*) *e2*) [*v*] (*xs*[*V* := *v*]) (*Suc* (*Suc* (*length* (*compE2 e1*)))) *None stk' loc' pc' xcp'*›
note *bisim* = ‹*P*,*e1*,*h* ⊢ (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim2* = ‹⋀*xs*. *P*,*e2*,*h* ⊢ (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *len* = ‹*n* + *max-vars* (*sync*$_V$ (*Val v*) *e2*) ≤ *length xs*›
note *bsok* = ‹*bsok* (*sync*$_V$ (*e1*) *e2*) *n*›
with *len* **have** *V*: *V* < *length xs* **by** *simp*
have *τ*: ¬ *τmove2* (*compP2 P*) *h* [*v*] (*sync*$_V$ (*e1*) *e2*) (*Suc* (*Suc* (*length* (*compE2 e1*)))) *None* **by**(*simp add*: *τmove2-iff*)
from *exec* **have** (∃ *a*. *v* = *Addr a* ∧ *stk'* = [] ∧ *loc'* = *xs*[*V* := *v*] ∧ *ta* = {⌈*Lock*→*a*, *SyncLock a*⌉} ∧ *xcp'* = *None* ∧ *pc'* = *Suc* (*Suc* (*Suc* (*length* (*compE2 e1*))))) ∨ (*v* = *Null* ∧ *stk'* = [*v*] ∧ *loc'* = *xs*[*V* := *v*] ∧ *ta* = *ε* ∧ *xcp'* = ⌊*addr-of-sys-xcpt NullPointer*⌋ ∧ *pc'* = *Suc* (*Suc* (*length* (*compE2 e1*)))) (**is** *?c1* ∨ *?c2*)
by(*fastforce elim*!: *exec-meth.cases simp add*: *is-Ref-def expand-finfun-eq fun-eq-iff finfun-upd-apply exec-move-def*)
thus *?case*
proof
assume *?c1*
then obtain *a* **where** [*simp*]: *v* = *Addr a stk'* = [] *loc'* = *xs*[*V* := *v*] *ta* = {⌈*Lock*→*a*, *SyncLock a*⌉} *xcp'* = *None pc'* = *Suc* (*Suc* (*Suc* (*length* (*compE2 e1*)))) **by** *blast*
have *True*,*P*,*t* ⊢*1* ⟨*sync*$_V$ (*addr a*) *e2*, (*h*, *xs*)⟩ −{⌈*Lock*→*a*, *SyncLock a*⌉}→ ⟨*insync*$_V$ (*a*) *e2*,(*h*, *xs*[*V* := *Addr a*])⟩
using *V* **by**(*rule Lock1Synchronized*)
moreover from *bisim2* **have** *P*,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*insync*$_V$ (*a*) *e2*, *xs*[*V* := *Addr a*]) ↔ ([], *xs*[*V* := *Addr a*], *Suc* (*Suc* (*Suc* (*length* (*compE2 e1*)))), *None*)
by(*auto intro*: *bisim1Sync4*[**where** *pc* = *0*, *simplified*])
ultimately show *?case* **using** *exec τ*
by(*fastforce elim*!: *exec-meth.cases split*: *if-split-asm simp add*: *is-Ref-def exec-move-def ta-bisim-def ta-upd-simps*)
next
assume *?c2*
hence [*simp*]: *v* = *Null stk'* = [*v*] *loc'* = *xs*[*V* := *v*] *ta* = *ε* *xcp'* = ⌊*addr-of-sys-xcpt NullPointer*⌋ *pc'* = *Suc* (*Suc* (*length* (*compE2 e1*))) **by** *simp-all*

```
    from V have True,P,t ⊢1 ⟨sync_V (null) e2, (h, xs)⟩ −ε→ ⟨THROW NullPointer,(h, xs[V := Null])⟩
      by(rule Synchronized1Null)
    moreover
    have P,sync_V (e1) e2,h ⊢ (THROW NullPointer, xs[V := Null]) ↔ ([Null], xs[V := Null], Suc (Suc (length (compE2 e1))), ⌊addr-of-sys-xcpt NullPointer⌋)
      by(rule bisim1Sync11)
    ultimately show ?case using exec τ
      by(fastforce elim!: exec-meth.cases split: if-split-asm simp add: is-Ref-def exec-move-def)
  qed
next
  case (bisim1Sync4 e2 n e' xs stk loc pc xcp e1 V a)
  note IH = bisim1Sync4.IH(2)
  note exec = ‹?exec (sync_V (e1) e2) stk loc (Suc (Suc (Suc (length (compE2 e1) + pc)))) xcp stk' loc' pc' xcp'›
  note bisim1 = ‹P,e1,h ⊢ (e1, xs) ↔ ([], xs, 0, None)›
  note bisim2 = ‹P,e2,h ⊢ (e', xs) ↔ (stk, loc, pc, xcp)›
  note len = ‹n + max-vars (insync_V (a) e') ≤ length xs›
  note bsok = ‹bsok (sync_V (e1) e2) n›
  with len have V: V < length xs and len': Suc n + max-vars e' ≤ length xs by simp-all
  from bisim2 have pc: pc ≤ length (compE2 e2) by(rule bisim1-pc-length-compE2)
  let ?pre = compE2 e1 @ [Dup, Store V, MEnter]
  let ?post = [Load V, MExit, Goto 4, Load V, MExit, ThrowExc]
  from exec have exec': exec-meth-d (compP2 P) (?pre @ compE2 e2 @ ?post)
    (compxE2 e1 0 0 @ shift (length ?pre) (compxE2 e2 0 0 @ [(0, length (compE2 e2), None, 3 + length (compE2 e2), 0)])) t
    h (stk, loc, length ?pre + pc, xcp) ta h' (stk', loc', pc', xcp')
    by(simp add: ac-simps eval-nat-numeral shift-compxE2 exec-move-def)
  hence pc': pc' ≥ length ?pre
    by(rule exec-meth-drop-xt-pc[where n'=1]) auto
  from exec' have exec'': exec-meth-d (compP2 P) (compE2 e2 @ ?post)
    (compxE2 e2 0 0 @ [(0, length (compE2 e2), None, 3 + length (compE2 e2), 0)]) t
    h (stk, loc, pc, xcp) ta h' (stk', loc', pc' − length ?pre, xcp')
    by(rule exec-meth-drop-xt[where n=1]) auto
  show ?case
  proof(cases pc < length (compE2 e2))
    case True
    note pc = True
   hence τ: τmove2 (compP2 P) h stk (sync_V (e1) e2) (Suc (Suc (Suc (length (compE2 e1) + pc)))) xcp = τmove2 (compP2 P) h stk e2 pc xcp
      by(simp add: τmove2-iff)
    show ?thesis
     proof(cases xcp = None ∨ (∃a'. xcp = ⌊a'⌋ ∧ match-ex-table (compP2 P) (cname-of h a') pc (compxE2 e2 0 0) ≠ None))
      case False
      then obtain a' where Some: xcp = ⌊a'⌋
        and True: match-ex-table (compP2 P) (cname-of h a') pc (compxE2 e2 0 0) = None by(auto simp del: not-None-eq)
      from Some ‹conf-xcp' (compP2 P) h xcp› obtain D
        where ha': typeof-addr h a' = ⌊Class-type D⌋ and subcls: P ⊢ D ⪯* Throwable by(auto simp add: compP2-def)
      from ha' subcls bisim2 True bsok have τred1r P t h (e', xs) (Throw a', loc)
        using len' unfolding Some compP2-def by(auto dest!: bisim1-xcp-τRed simp add: bsok-def)
       moreover from pc have τmove2 (compP2 P) h stk (sync_V (e1) e2) (Suc (Suc (Suc (pc +
```

length (*compE2 e1*))))) ⌊*a'*⌋
by(*auto intro*: *τmove2xcp*)
moreover
have *P*, *sync*$_V$ (*e1*) *e2*, *h* ⊢ (*insync*$_V$ (*a*) *Throw a'*, *loc*) ↔ ([*Addr a'*], *loc*, *6* + *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
by(*rule bisim1Sync7*)
ultimately show *?thesis* **using** *exec True pc Some ha' subcls*
apply(*auto elim*!: *exec-meth.cases simp add*: *ac-simps eval-nat-numeral match-ex-table-append matches-ex-entry-def compP2-def exec-move-def*)

apply(*simp-all only*: *compxE2-size-convs*, *auto dest*: *match-ex-table-shift-pcD match-ex-table-pc-length-compE2*
apply(*fastforce elim*!: *InSync-τred1r-xt*)
done
next
case *True*
with *exec''* *pc* **have** *exec-meth-d* (*compP2 P*) (*compE2 e2* @ *?post*) (*compxE2 e2 0 0*) *t* *h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'* − *length ?pre*, *xcp'*)
by(*auto elim*!: *exec-meth.cases intro*: *exec-meth.intros simp add*: *match-ex-table-append exec-move-def*)
hence *exec-move-d P t e2 h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'* − *length ?pre*, *xcp'*)
using *pc* **unfolding** *exec-move-def* **by**(*rule exec-meth-take*)
from *IH*[*OF this len'* - ‹*P*,*h* ⊢ *stk* [:≤] *ST*› ‹*conf-xcp'* (*compP2 P*) *h xcp*›] *bsok* **obtain** *e''* *xs''*
where *bisim'*: *P*,*e2*,*h'* ⊢ (*e''*, *xs''*) ↔ (*stk'*, *loc'*, *pc'* − *length ?pre*, *xcp'*)
and *red'*: *?red e' xs e'' xs'' e2 stk pc* (*pc'* − *length ?pre*) *xcp xcp'* **by** *auto*
from *bisim'*
have *P*,*sync*$_V$ (*e1*) *e2*, *h'* ⊢ (*insync*$_V$ (*a*) *e''*, *xs''*) ↔ (*stk'*, *loc'*, *Suc* (*Suc* (*Suc* (*length* (*compE2 e1*) + (*pc'* − *length ?pre*)))), *xcp'*)
by(*rule bisim1-bisims1.bisim1Sync4*)
moreover from *pc'* **have** *Suc* (*Suc* (*Suc* (*length* (*compE2 e1*) + (*pc'* − *Suc* (*Suc* (*Suc* (*length* (*compE2 e1*)))))))) = *pc'*
Suc (*Suc* (*Suc* (*pc'* − *Suc* (*Suc* (*Suc 0*))))) = *pc'*
by *simp-all*
ultimately show *?thesis* **using** *red' τ*
by(*fastforce intro*: *Synchronized1Red2 simp add*: *eval-nat-numeral no-call2-def elim*!: *InSync-τred1r-xt InSync-τred1t-xt split*: *if-split-asm*)
qed
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compE2 e2*) **by** *simp*
with *bisim2* **obtain** *v* **where** [*simp*]: *stk* = [*v*] *xcp* = *None* **by**(*auto dest*: *bisim1-pc-length-compE2D*)
have *τmove2* (*compP2 P*) *h* [*v*] (*sync*$_V$ (*e1*) *e2*) (*Suc* (*Suc* (*Suc* (*length* (*compE2 e1*) + *length* (*compE2 e2*))))) *None* **by**(*simp add*: *τmove2-iff*)
moreover from *bisim2 pc len bsok* **have** *red*: *τred1r P t h* (*e'*, *xs*) (*Val v*, *loc*)
by(*auto intro*!: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *τred1r P t h* (*insync*$_V$ (*a*) *e'*, *xs*) (*insync*$_V$ (*a*) (*Val v*), *loc*) **by**(*rule InSync-τred1r-xt*)
moreover
have *P*,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*insync*$_V$ (*a*) (*Val v*), *loc*) ↔ ([*loc* ! *V*, *v*], *loc*, *4* + *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
by(*rule bisim1Sync5*)
ultimately show *?thesis* **using** *exec*
by(*auto elim*!: *exec-meth.cases simp add*: *eval-nat-numeral exec-move-def*) *blast*
qed
next
case (*bisim1Sync5 e1 n e2 V a v xs*)
note *bisim1* = ‹*P*,*e1*,*h* ⊢ (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*)›

note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *exec* = ‹*?exec* (*sync*$_V$ (*e1*) *e2*) [*xs* ! *V*, *v*] *xs* (*4* + *length* (*compE2 e1*) + *length* (*compE2 e2*)) *None stk′ loc′ pc′ xcp′*›
from ‹*n* + *max-vars* (*insync*$_V$ (*a*) *Val v*) ≤ *length xs*› ‹*bsok* (*sync*$_V$ (*e1*) *e2*) *n*› **have** *V*: *V* < *length xs* **by** *simp*
have *τ*: ¬ *τmove2* (*compP2 P*) *h* [*xs* ! *V*, *v*] (*sync*$_V$ (*e1*) *e2*) (*4* + *length* (*compE2 e1*) + *length* (*compE2 e2*)) *None*
by(*simp add*: *τmove2-iff*)
have *τ′*: ¬ *τmove1 P h* (*insync*$_V$ (*a*) *Val v*) **by** *auto*
from *exec* **have** (∃ *a′*. *xs* ! *V* = *Addr a′*) ∨ *xs* ! *V* = *Null* (**is** *?c1* ∨ *?c2*)
by(*auto elim*!: *exec-meth.cases simp add*: *split-beta is-Ref-def exec-move-def split*: *if-split-asm*)
thus *?case*
proof
assume *?c1*
then obtain *a′* **where** *xsV* [*simp*]: *xs* ! *V* = *Addr a′* **..**
have *P*,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*Val v*, *xs*) ↔ ([*v*], *xs*, *5* + *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
P,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*THROW IllegalMonitorState*,*xs*) ↔ ([*Addr a′*, *v*],*xs*,*4* + *length* (*compE2 e1*) + *length* (*compE2 e2*),⌊*addr-of-sys-xcpt IllegalMonitorState*⌋)
by(*rule bisim1Sync6*, *rule bisim1Sync12*)
moreover from *xsV V* **have** *True*,*P*,*t* ⊢*1* ⟨*insync*$_V$ (*a*) *Val v*, (*h*, *xs*)⟩ −⦃*Unlock*→*a′*, *SyncUnlock a′*⦄→ ⟨*Val v*,(*h*, *xs*)⟩
by(*rule Unlock1Synchronized*)
moreover from *xsV V* **have** *True*,*P*,*t* ⊢*1* ⟨*insync*$_V$ (*a*) *Val v*, (*h*, *xs*)⟩ −⦃*UnlockFail*→*a′*⦄→ ⟨*THROW IllegalMonitorState*,(*h*, *xs*)⟩
by(*rule Unlock1SynchronizedFail*[*OF TrueI*])
ultimately show *?case* **using** *τ τ′ exec*
by (*fastforce elim*!: *exec-meth.cases simp add*: *is-Ref-def ta-bisim-def exec-move-def ac-simps ta-upd-simps*
simp del: *conj.left-commute*)
next
assume *xsV*: *xs* ! *V* = *Null*
have *P*,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*THROW NullPointer*,*xs*) ↔ ([*Null*, *v*],*xs*,*4* + *length* (*compE2 e1*) + *length* (*compE2 e2*),⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule bisim1Sync12*)
thus *?case* **using** *τ τ′ exec xsV V*
by (*fastforce elim*!: *exec-meth.cases intro*: *Unlock1SynchronizedNull simp add*: *is-Ref-def ta-bisim-def ac-simps exec-move-def*
simp del: *conj.left-commute*)
qed
next
case (*bisim1Sync6 e1 n e2 V v x*)
note *bisim1* = ‹*P*,*e1*,*h* ⊢ (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *exec* = ‹*?exec* (*sync*$_V$ (*e1*) *e2*) [*v*] *x* (*5* + *length* (*compE2 e1*) + *length* (*compE2 e2*)) *None stk′ loc′ pc′ xcp′*›
have *τmove2* (*compP2 P*) *h* [*v*] (*sync*$_V$ (*e1*) *e2*) (*5* + *length* (*compE2 e1*) + *length* (*compE2 e2*)) *None* **by**(*simp add*: *τmove2-iff*)
moreover
have *P*,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*Val v*, *x*) ↔ ([*v*], *x*, *length* (*compE2* (*sync*$_V$ (*e1*) *e2*)), *None*)
by(*rule bisim1Val2*) *simp*
moreover have *nat* (*9* + (*int* (*length* (*compE2 e1*)) + *int* (*length* (*compE2 e2*)))) = *9* + *length* (*compE2 e1*) + *length* (*compE2 e2*) **by** *arith*
ultimately show *?case* **using** *exec*

by(*fastforce elim*!: *exec-meth.cases simp add*: *eval-nat-numeral exec-move-def*)
next
case (*bisim1Sync7 e1 n e2 V a a' xs*)
note ‹*?exec* (*sync*$_V$ (*e1*) *e2*) [*Addr a'*] *xs* (*6* + *length* (*compE2 e1*) + *length* (*compE2 e2*)) *None stk' loc' pc' xcp'*›
moreover
have *P*,*sync*$_V$ (*e1*) *e2*,*h'* ⊢ (*insync*$_V$ (*a*) *Throw a'*, *xs*) ↔
([*xs* ! *V*, *Addr a'*], *xs*, *7* + *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
by(*rule bisim1Sync8*)
moreover have *τmove2* (*compP2 P*) *h* [*Addr a'*] (*sync*$_V$ (*e1*) *e2*) (*6* + *length* (*compE2 e1*) + *length* (*compE2 e2*)) *None*
by(*simp add*: *τmove2-iff*)
ultimately show *?case* **by**(*fastforce elim*!: *exec-meth.cases simp add*: *eval-nat-numeral exec-move-def*)
next
case (*bisim1Sync8 e1 n e2 V a a' xs*)
from ‹*n* + *max-vars* (*insync*$_V$ (*a*) *Throw a'*) ≤ *length xs*› ‹*bsok* (*sync*$_V$ (*e1*) *e2*) *n*› **have** *V*: *V* < *length xs* **by** *simp*
note ‹*?exec* (*sync*$_V$ (*e1*) *e2*) [*xs* ! *V*, *Addr a'*] *xs* (*7* + *length* (*compE2 e1*) + *length* (*compE2 e2*)) *None stk' loc' pc' xcp'*›
moreover have ¬ *τmove2* (*compP2 P*) *h* [*xs* ! *V*, *Addr a'*] (*sync*$_V$ (*e1*) *e2*) (*7* + *length* (*compE2 e1*) + *length* (*compE2 e2*)) *None*
by(*simp add*: *τmove2-iff*)
moreover
have *P*,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*Throw a'*, *xs*) ↔ ([*Addr a'*], *xs*, *8* + *length* (*compE2 e1*) + *length* (*compE2 e2*), *None*)
P,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*THROW IllegalMonitorState*,*xs*) ↔ ([*xs* ! *V*, *Addr a'*],*xs*,*7* + *length* (*compE2 e1*) + *length* (*compE2 e2*),⌊*addr-of-sys-xcpt IllegalMonitorState*⌋)
P,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*THROW NullPointer*,*xs*) ↔ ([*Null*, *Addr a'*],*xs*,*7* + *length* (*compE2 e1*) + *length* (*compE2 e2*),⌊*addr-of-sys-xcpt NullPointer*⌋)
by(*rule bisim1Sync9 bisim1Sync14*)+
moreover {
fix *A*
assume *xs* ! *V* = *Addr A*
hence *True*,*P*,*t* ⊢*1* ⟨*insync*$_V$ (*a*) *Throw a'*,(*h*, *xs*)⟩ −{|*Unlock*→*A*, *SyncUnlock A*|}→ ⟨*Throw a'*, (*h*, *xs*)⟩
True,*P*,*t* ⊢*1* ⟨*insync*$_V$ (*a*) *Throw a'*,(*h*, *xs*)⟩ −{|*UnlockFail*→*A*|}→ ⟨*THROW IllegalMonitorState*, (*h*, *xs*)⟩
using *V* **by**(*rule Synchronized1Throw2 Synchronized1Throw2Fail*[*OF TrueI*])+ }
moreover {
assume *xs* ! *V* = *Null*
hence *True*,*P*,*t* ⊢*1* ⟨*insync*$_V$ (*a*) *Throw a'*,(*h*, *xs*)⟩ −*ε*→ ⟨*THROW NullPointer*, (*h*, *xs*)⟩
using *V* **by**(*rule Synchronized1Throw2Null*) }
moreover have ¬ *τmove1 P h* (*insync*$_V$ (*a*) *Throw a'*) **by** *fastforce*
ultimately show *?case*
by(*fastforce elim*!: *exec-meth.cases simp add*: *eval-nat-numeral is-Ref-def ta-bisim-def ta-upd-simps exec-move-def split*: *if-split-asm*)
next
case (*bisim1Sync9 e1 n e2 V a xs*)
note ‹*?exec* (*sync*$_V$ (*e1*) *e2*) [*Addr a*] *xs* (*8* + *length* (*compE2 e1*) + *length* (*compE2 e2*)) *None stk' loc' pc' xcp'*›
moreover
have *P*,*sync*$_V$ (*e1*) *e2*,*h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *xs*, *8* + *length* (*compE2 e1*) + *length* (*compE2 e2*), ⌊*a*⌋)
by(*rule bisim1Sync10*)

moreover have *τmove2 (compP2 P) h [Addr a] (sync*$_V$ *(e1) e2) (8 + length (compE2 e1) + length (compE2 e2)) None*
by(*rule τmove2Sync8*)
ultimately show *?case*
by(*fastforce elim*!: *exec-meth.cases simp add*: *eval-nat-numeral exec-move-def split*: *if-split-asm*)
next
case (*bisim1Sync10 e1 n e2 V a xs*)
from ‹*?exec (sync*$_V$ *(e1) e2) [Addr a] xs (8 + length (compE2 e1) + length (compE2 e2)) ⌊a⌋ stk′ loc′ pc′ xcp′*›
have *False* **by**(*auto elim*!: *exec-meth.cases simp add*: *matches-ex-entry-def match-ex-table-append-not-pcs exec-move-def*)
thus *?case* **..**
next
case (*bisim1Sync11 e1 n e2 V xs*)
from ‹*?exec (sync*$_V$ *(e1) e2) [Null] xs (Suc (Suc (length (compE2 e1)))) ⌊addr-of-sys-xcpt NullPointer⌋ stk′ loc′ pc′ xcp′*›
have *False* **by**(*auto elim*!: *exec-meth.cases simp add*: *matches-ex-entry-def match-ex-table-append-not-pcs exec-move-def*)
thus *?case* **..**
next
case (*bisim1Sync12 e1 n e2 V a xs v v′*)
from ‹*?exec (sync*$_V$ *(e1) e2) [v, v′] xs (4 + length (compE2 e1) + length (compE2 e2)) ⌊a⌋ stk′ loc′ pc′ xcp′*›
have *False* **by**(*auto elim*!: *exec-meth.cases simp add*: *matches-ex-entry-def match-ex-table-append-not-pcs exec-move-def*)
thus *?case* **..**
next
case (*bisim1Sync14 e1 n e2 V a xs v a′*)
from ‹*?exec (sync*$_V$ *(e1) e2) [v, Addr a′] xs (7 + length (compE2 e1) + length (compE2 e2)) ⌊a⌋ stk′ loc′ pc′ xcp′*›
have *False* **by**(*auto elim*!: *exec-meth.cases simp add*: *matches-ex-entry-def match-ex-table-append-not-pcs exec-move-def*)
thus *?case* **..**
next
case *bisim1InSync* **thus** *?case* **by** *simp*
next
case (*bisim1SyncThrow e1 n a xs stk loc pc e2 V*)
note *exec* = ‹*?exec (sync*$_V$ *(e1) e2) stk loc pc ⌊a⌋ stk′ loc′ pc′ xcp′*›
note *bisim1* = ‹*P,e1,h* ⊢ (*Throw a, xs*) ↔ (*stk, loc, pc, ⌊a⌋*)›
from *bisim1* **have** *pc*: *pc* < *length (compE2 e1)* **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim1* **have** *match-ex-table (compP2 P) (cname-of h a) (0 + pc) (compxE2 e1 0 0) = None*
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)
with *exec pc* **have** *False*
apply(*auto elim*!: *exec-meth.cases simp add*: *match-ex-table-append-not-pcs exec-move-def*)
apply(*auto dest*!: *match-ex-table-shift-pcD simp add*: *matches-ex-entry-def*)
done
thus *?case* **..**
next
case (*bisim1Seq1 e1 n e′ xs stk loc pc xcp e2*)
note *IH* = *bisim1Seq1.IH*(*2*)
note *exec* = ‹*?exec (e1;; e2) stk loc pc xcp stk′ loc′ pc′ xcp′*›
note *bisim* = ‹*P,e1,h* ⊢ (*e′, xs*) ↔ (*stk, loc, pc, xcp*)›
note *len* = ‹*n + max-vars (e′;;e2)* ≤ *length xs*›
note *bsok* = ‹*bsok (e1;; e2) n*›

from *bisim* **have** *pc*: *pc* ≤ *length* (*compE2 e1*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e1*))
case *True*
let *?post* = *Pop* # *compE2 e2*
from *exec* **have** *exec'*: *?exec e1 stk loc pc xcp stk' loc' pc' xcp'* **using** *True* **by**(*simp add*: *exec-move-Seq1*)
from *True* **have** *τmove2* (*compP2 P*) *h stk* (*e1*;;*e2*) *pc xcp* = *τmove2* (*compP2 P*) *h stk e1 pc xcp* **by**(*simp add*: *τmove2-iff*)
with *IH*[*OF exec'* - - ‹*P*,*h* ⊢ *stk* [:≤] *ST*› ‹*conf-xcp'* (*compP2 P*) *h xcp*›] *len bsok* **show** *?thesis*
by(*fastforce intro*: *bisim1-bisims1.bisim1Seq1 Seq1Red elim*!: *Seq-τred1r-xt Seq-τred1t-xt simp add*: *no-call2-def*)
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compE2 e1*) **by** *simp*
with *bisim* **obtain** *v* **where** *stk*: *stk* = [*v*] **and** *xcp*: *xcp* = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim pc len bsok* **have** *red*: *τred1r P t h* (*e'*, *xs*) (*Val v*, *loc*)
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *τred1r P t h* (*e'*;; *e2*, *xs*) (*Val v*;;*e2*, *loc*) **by**(*rule Seq-τred1r-xt*)
also have *τmove1 P h* (*Val v*;;*e2*) **by**(*rule τmove1SeqRed*)
hence *τred1r P t h* (*Val v*;;*e2*, *loc*) (*e2*, *loc*) **by**(*auto intro*: *Red1Seq r-into-rtranclp*)
also have *τ*: *τmove2* (*compP2 P*) *h* [*v*] (*e1*;;*e2*) *pc None* **by**(*simp add*: *τmove2-iff*)
moreover from ‹*P*,*e2*,*h* ⊢ (*e2*, *loc*) ↔ ([], *loc*, *0*, *None*)›
have *P*,*e1*;;*e2*,*h* ⊢ (*e2*, *loc*) ↔ ([], *loc*, *Suc* (*length* (*compE2 e1*) + *0*), *None*)
by(*rule bisim1Seq2*)
ultimately show *?thesis* **using** *exec stk xcp*
by(*fastforce elim*!: *exec-meth.cases simp add*: *exec-move-def*)
qed
next
case (*bisim1SeqThrow1 e1 n a xs stk loc pc e2*)
note *exec* = ‹*?exec* (*e1*;;*e2*) *stk loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*›
note *bisim1* = ‹*P*,*e1*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
from *bisim1* **have** *pc*: *pc* < *length* (*compE2 e1*) **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim1* **have** *match-ex-table* (*compP2 P*) (*cname-of h a*) (*0* + *pc*) (*compxE2 e1 0 0*) = *None*
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)
with *exec pc* **have** *False*
by(*auto elim*!: *exec-meth.cases simp add*: *match-ex-table-not-pcs-None exec-move-def*)
thus *?case* **..**
next
case (*bisim1Seq2 e2 n e' xs stk loc pc xcp e1*)
note *IH* = *bisim1Seq2.IH*(*2*)
note *bisim1* = ‹⋀*xs*. *P*,*e1*,*h* ⊢ (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *len* = ‹*n* + *max-vars e'* ≤ *length xs*›
note *exec* = ‹*?exec* (*e1*;; *e2*) *stk loc* (*Suc* (*length* (*compE2 e1*) + *pc*)) *xcp stk' loc' pc' xcp'*›
note *bsok* = ‹*bsok* (*e1*;; *e2*) *n*›
let *?pre* = *compE2 e1* @ [*Pop*]
from *exec* **have** *exec'*: *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e2*) (*compxE2 e1 0 0* @ *shift* (*length ?pre*) (*compxE2 e2 0 0*)) *t h* (*stk*, *loc*, *length ?pre* + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *shift-compxE2 exec-move-def*)
hence *?exec e2 stk loc pc xcp stk' loc'* (*pc'* − *length ?pre*) *xcp'*
unfolding *exec-move-def* **by**(*rule exec-meth-drop-xt*, *auto*)
from *IH*[*OF this len* - ‹*P*,*h* ⊢ *stk* [:≤] *ST*› ‹*conf-xcp'* (*compP2 P*) *h xcp*›] *bsok* **obtain** *e''* *xs''*

where *bisim'*: $P,e2,h' \vdash (e'', xs'') \leftrightarrow (stk', loc', pc' - length\ ?pre, xcp')$
and *red*: *?red e' xs e'' xs'' e2 stk pc* $(pc' - length\ ?pre)$ *xcp xcp'* **by** *auto*
from *bisim'*
have $P,e1;;e2,h' \vdash (e'', xs'') \leftrightarrow (stk', loc', Suc\ (length\ (compE2\ e1) + (pc' - length\ ?pre)), xcp')$
by(*rule bisim1-bisims1.bisim1Seq2*)
moreover have τ: $\tau move2\ (compP2\ P)\ h\ stk\ (e1;;e2)\ (Suc\ (length\ (compE2\ e1) + pc))\ xcp = \tau move2\ (compP2\ P)\ h\ stk\ e2\ pc\ xcp$
by(*simp add*: *τmove2-iff*)
moreover from *exec'* **have** $pc' \geq length\ ?pre$
by(*rule exec-meth-drop-xt-pc*) *auto*
ultimately show *?case* **using** *red* **by**(*fastforce split*: *if-split-asm simp add*: *no-call2-def*)
next
case (*bisim1Cond1 e n e' xs stk loc pc xcp e1 e2*)
note *IH* = *bisim1Cond1.IH*(2)
note *bisim* = ‹$P,e,h \vdash (e', xs) \leftrightarrow (stk, loc, pc, xcp)$›
note *bisim1* = ‹$\bigwedge xs.\ P,e1,h \vdash (e1, xs) \leftrightarrow ([], xs, 0, None)$›
note *bisim2* = ‹$\bigwedge xs.\ P,e2,h \vdash (e2, xs) \leftrightarrow ([], xs, 0, None)$›
from ‹$n + max\text{-}vars\ (if\ (e')\ e1\ else\ e2) \leq length\ xs$›
have *len*: $n + max\text{-}vars\ e' \leq length\ xs$ **by** *simp*
note *exec* = ‹*?exec* (*if* (*e*) *e1 else e2*) *stk loc pc xcp stk' loc' pc' xcp'*›
note *bsok* = ‹*bsok* (*if* (*e*) *e1 else e2*) *n*›
from *bisim* **have** *pc*: $pc \leq length\ (compE2\ e)$ **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases* $pc < length\ (compE2\ e)$)
case *True*
let *?post* = *IfFalse* (2 + *int* (*length* (*compE2 e1*))) # *compE2 e1* @ *Goto* (1 + *int* (*length* (*compE2 e2*))) # *compE2 e2*
from *exec* **have** *exec'*: *?exec e stk loc pc xcp stk' loc' pc' xcp'* **using** *True*
by(*simp add*: *exec-move-Cond1*)
from *True* **have** $\tau move2\ (compP2\ P)\ h\ stk\ (if\ (e)\ e1\ else\ e2)\ pc\ xcp = \tau move2\ (compP2\ P)\ h\ stk\ e\ pc\ xcp$
by(*simp add*: *τmove2-iff*)
with *IH*[*OF exec' - -* ‹$P,h \vdash stk\ [:\leq]\ ST$› ‹*conf-xcp'* (*compP2 P*) *h xcp*›] *len bsok* **show** *?thesis*
by(*fastforce intro*: *bisim1-bisims1.bisim1Cond1 Cond1Red elim*!: *Cond-τred1r-xt Cond-τred1t-xt simp add*: *no-call2-def*)
next
case *False*
with *pc* **have** [*simp*]: $pc = length\ (compE2\ e)$ **by** *simp*
with *bisim* **obtain** *v* **where** *stk*: $stk = [v]$ **and** *xcp*: $xcp = None$
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim pc len bsok* **have** *red*: $\tau red1r\ P\ t\ h\ (e', xs)\ (Val\ v, loc)$
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence $\tau red1r\ P\ t\ h\ (if\ (e')\ e1\ else\ e2, xs)\ (if\ (Val\ v)\ e1\ else\ e2, loc)$ **by**(*rule Cond-τred1r-xt*)
moreover have $\tau move1\ P\ h\ (if\ (Val\ v)\ e1\ else\ e2)$ **by**(*rule τmove1CondRed*)
moreover have τ: $\tau move2\ (compP2\ P)\ h\ [v]\ (if\ (e)\ e1\ else\ e2)\ pc\ None$ **by**(*simp add*: *τmove2-iff*)
moreover from *bisim1*[*of loc*]
have $P,if\ (e)\ e1\ else\ e2,h \vdash (e1, loc) \leftrightarrow ([], loc, Suc\ (length\ (compE2\ e) + 0), None)$
by(*rule bisim1CondThen*)
moreover from *bisim2*[*of loc*]
have $P,if\ (e)\ e1\ else\ e2,h \vdash (e2, loc) \leftrightarrow ([], loc, Suc\ (Suc\ (length\ (compE2\ e) + length\ (compE2\ e1) + 0)), None)$
by(*rule bisim1CondElse*)
moreover have $nat\ (int\ (length\ (compE2\ e)) + (2 + int\ (length\ (compE2\ e1)))) = Suc\ (Suc\ (length\ (compE2\ e) + length\ (compE2\ e1) + 0))$ **by** *simp*

moreover from *exec xcp stk* **have** *typeof*$_h$ *v* = ⌊*Boolean*⌋ **by**(*auto simp add*: *exec-move-def exec-meth-instr*)
ultimately show *?thesis* **using** *exec stk xcp*
by(*fastforce elim*!: *exec-meth.cases intro*: *Red1CondT Red1CondF elim*!: *rtranclp.rtrancl-into-rtrancl simp add*: *eval-nat-numeral exec-move-def*)
qed
next
case (*bisim1CondThen e1 n e' xs stk loc pc xcp e e2*)
note *IH* = *bisim1CondThen.IH*(*2*)
note *bisim1* = ‹*P*,*e1*,*h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim* = ‹⋀*xs*. *P*,*e*,*h* ⊢ (*e*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim2* = ‹⋀*xs*. *P*,*e2*,*h* ⊢ (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *len* = ‹*n* + *max-vars e'* ≤ *length xs*›
note *exec* = ‹*?exec* (*if* (*e*) *e1 else e2*) *stk loc* (*Suc* (*length* (*compE2 e*) + *pc*)) *xcp stk' loc' pc' xcp'*›
note *bsok* = ‹*bsok* (*if* (*e*) *e1 else e2*) *n*›
from *bisim1* **have** *pc*: *pc* ≤ *length* (*compE2 e1*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e1*))
case *True*
let *?pre* = *compE2 e* @ [*IfFalse* (*2* + *int* (*length* (*compE2 e1*)))]
let *?post* = *Goto* (*1* + *int* (*length* (*compE2 e2*))) # *compE2 e2*
from *exec* **have** *exec'*: *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e1* @ *?post*)
(*compxE2 e 0 0* @ *shift* (*length ?pre*) (*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*compxE2 e2* (*Suc 0*) *0*))) *t*
h (*stk*, *loc*, *length ?pre* + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *shift-compxE2 ac-simps exec-move-def*)
hence *exec-meth-d* (*compP2 P*) (*compE2 e1* @ *?post*) (*compxE2 e1 0 0* @ *shift* (*length* (*compE2 e1*)) (*compxE2 e2* (*Suc 0*) *0*)) *t*
h (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'* − *length ?pre*, *xcp'*)
by(*rule exec-meth-drop-xt*) *auto*
hence *exec-move-d P t e1 h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'* − *length ?pre*, *xcp'*)
using *True* **unfolding** *exec-move-def* **by**(*rule exec-meth-take-xt*)
from *IH*[*OF this len* - ‹*P*,*h* ⊢ *stk* [:≤] *ST*› ‹*conf-xcp'* (*compP2 P*) *h xcp*›] *bsok* **obtain** *e'' xs''*
where *bisim'*: *P*,*e1*,*h'* ⊢ (*e''*, *xs''*) ↔ (*stk'*, *loc'*, *pc'* − *length ?pre*, *xcp'*)
and *red*: *?red e' xs e'' xs'' e1 stk pc* (*pc'* − *length ?pre*) *xcp xcp'* **by** *auto*
from *bisim'*
have *P*,*if* (*e*) *e1 else e2*,*h'* ⊢ (*e''*, *xs''*) ↔ (*stk'*, *loc'*, *Suc* (*length* (*compE2 e*) + (*pc'* − *length ?pre*)), *xcp'*)
by(*rule bisim1-bisims1.bisim1CondThen*)
moreover from *True* **have** *τ*: *τmove2* (*compP2 P*) *h stk* (*if* (*e*) *e1 else e2*) (*Suc* (*length* (*compE2 e*) + *pc*)) *xcp* = *τmove2* (*compP2 P*) *h stk e1 pc xcp*
by(*simp add*: *τmove2-iff*)
moreover from *exec'* **have** *pc'* ≥ *length ?pre*
by(*rule exec-meth-drop-xt-pc*) *auto*
ultimately show *?thesis* **using** *red* **by**(*fastforce split*: *if-split-asm simp add*: *no-call2-def*)
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compE2 e1*) **by** *simp*
with *bisim1* **obtain** *v* **where** *stk*: *stk* = [*v*] **and** *xcp*: *xcp* = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim1 pc len bsok* **have** *red*: *τred1r P t h* (*e'*, *xs*) (*Val v*, *loc*)
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
moreover have *τ*: *τmove2* (*compP2 P*) *h* [*v*] (*if* (*e*) *e1 else e2*) (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*))) *None*

by(*simp add*: *τmove2-iff*)
moreover
have *P*,*if* (*e*) *e1 else e2*,*h* ⊢ (*Val v*, *loc*) ↔ ([*v*], *loc*, *length* (*compE2* (*if* (*e*) *e1 else e2*)), *None*)
by(*rule bisim1Val2*) *simp*
moreover have *nat* (*2* + (*int* (*length* (*compE2 e*)) + (*int* (*length* (*compE2 e1*)) + *int* (*length* (*compE2 e2*))))) = *length* (*compE2* (*if* (*e*) *e1 else e2*)) **by** *simp*
ultimately show *?thesis* **using** *exec xcp stk* **by**(*fastforce elim*!: *exec-meth.cases simp add*: *exec-move-def*)
qed
next
case (*bisim1CondElse e2 n e′ xs stk loc pc xcp e e1*)
note *IH* = *bisim1CondElse.IH*(*2*)
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim* = ‹⋀*xs*. *P*,*e*,*h* ⊢ (*e*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim1* = ‹⋀*xs*. *P*,*e1*,*h* ⊢ (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *len* = ‹*n* + *max-vars e′* ≤ *length xs*›
note *exec* = ‹*?exec* (*if* (*e*) *e1 else e2*) *stk loc* (*Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + *pc*))) *xcp stk′ loc′ pc′ xcp′*›
note *bsok* = ‹*bsok* (*if* (*e*) *e1 else e2*) *n*›
let *?pre* = *compE2 e* @ *IfFalse* (*2* + *int* (*length* (*compE2 e1*))) # *compE2 e1* @ [*Goto* (*1* + *int* (*length* (*compE2 e2*)))]
from *exec* **have** *exec′*: *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e2*)
((*compxE2 e 0 0* @ *compxE2 e1* (*Suc* (*length* (*compE2 e*))) *0*) @ *shift* (*length ?pre*) (*compxE2 e2 0 0*)) *t*
h (*stk*, *loc*, *length ?pre* + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *shift-compxE2 ac-simps exec-move-def*)
hence *?exec e2 stk loc pc xcp stk′ loc′* (*pc′* − *length ?pre*) *xcp′*
unfolding *exec-move-def* **by**(*rule exec-meth-drop-xt*) *auto*
from *IH*[*OF this len* - ‹*P*,*h* ⊢ *stk* [:≤] *ST*› ‹*conf-xcp′* (*compP2 P*) *h xcp*›] *bsok* **obtain** *e″ xs″*
where *bisim′*: *P*,*e2*,*h′* ⊢ (*e″*, *xs″*) ↔ (*stk′*, *loc′*, *pc′* − *length ?pre*, *xcp′*)
and *red*: *?red e′ xs e″ xs″ e2 stk pc* (*pc′* − *length ?pre*) *xcp xcp′* **by** *auto*
from *bisim′*
have *P*,*if* (*e*) *e1 else e2*,*h′* ⊢ (*e″*, *xs″*) ↔ (*stk′*, *loc′*, *Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + (*pc′* − *length ?pre*))), *xcp′*)
by(*rule bisim1-bisims1.bisim1CondElse*)
moreover have *τ*: *τmove2* (*compP2 P*) *h stk* (*if* (*e*) *e1 else e2*) (*Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + *pc*))) *xcp* = *τmove2* (*compP2 P*) *h stk e2 pc xcp*
by(*simp add*: *τmove2-iff*)
moreover from *exec′* **have** *pc′* ≥ *length ?pre*
by(*rule exec-meth-drop-xt-pc*) *auto*
moreover hence *Suc* (*Suc* (*pc′* − *Suc* (*Suc 0*))) = *pc′* **by** *simp*
ultimately show *?case* **using** *red* **by**(*fastforce simp add*: *eval-nat-numeral no-call2-def split*: *if-split-asm*)
next
case (*bisim1CondThrow e n a xs stk loc pc e1 e2*)
note *exec* = ‹*?exec* (*if* (*e*) *e1 else e2*) *stk loc pc* ⌊*a*⌋ *stk′ loc′ pc′ xcp′*›
note *bisim* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
from *bisim* **have** *pc*: *pc* < *length* (*compE2 e*) **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim* **have** *match-ex-table* (*compP2 P*) (*cname-of h a*) (*0* + *pc*) (*compxE2 e 0 0*) = *None*
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)
with *exec pc* **have** *False*
by(*auto elim*!: *exec-meth.cases simp add*: *match-ex-table-not-pcs-None exec-move-def*)
thus *?case* **..**
next
case (*bisim1While1 c n e xs*)
note *IH* = *bisim1While1.IH*(*2*)

note *bisim* = ‹$\bigwedge$*xs*. *P*,*c*,*h* ⊢ (*c*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim1* = ‹$\bigwedge$*xs*. *P*,*e*,*h* ⊢ (*e*, *xs*) ↔ ([], *xs*, *0*, *None*)›
from ‹*n* + *max-vars* (*while* (*c*) *e*) ≤ *length xs*›
have *len*: *n* + *max-vars c* ≤ *length xs* **by** *simp*
note *exec* = ‹*?exec* (*while* (*c*) *e*) [] *xs 0 None stk′ loc′ pc′ xcp′*›
note *bsok* = ‹*bsok* (*while* (*c*) *e*) *n*›
let *?post* = *IfFalse* (*int* (*length* (*compE2 e*)) + *3*) # *compE2 e* @ [*Pop*, *Goto* (−*2* + (− *int* (*length* (*compE2 e*)) − *int* (*length* (*compE2 c*)))), *Push Unit*]
from *exec* **have** *?exec c* [] *xs 0 None stk′ loc′ pc′ xcp′* **by**(*simp add*: *exec-move-While1*)
from *IH*[*OF this len*] *bsok* **obtain** *e″ xs″*
where *bisim′*: *P*,*c*,*h′* ⊢ (*e″*, *xs″*) ↔ (*stk′*, *loc′*, *pc′*, *xcp′*)
and *red*: *?red c xs e″ xs″ c* [] *0 pc′ None xcp′* **by** *fastforce*
from *bisim′*
have *P*,*while* (*c*) *e*,*h′* ⊢ (*if* (*e″*) (*e*;;*while*(*c*) *e*) *else unit*, *xs″*) ↔ (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*rule bisim1While3*)
moreover have *True*,*P*,*t* ⊢*1* ⟨*while* (*c*) *e*, (*h*, *xs*)⟩ −*ε*→ ⟨*if* (*c*) (*e*;;*while* (*c*) *e*) *else unit*, (*h*, *xs*)⟩
by(*rule Red1While*)
hence *τred1r P t h* (*while* (*c*) *e*, *xs*) (*if* (*c*) (*e*;;*while* (*c*) *e*) *else unit*, *xs*)
by(*auto intro*: *r-into-rtranclp τmove1WhileRed*)
moreover have *τmove2* (*compP2 P*) *h* [] (*while* (*c*) *e*) *0 None* = *τmove2* (*compP2 P*) *h* [] *c 0 None*
by(*simp add*: *τmove2-iff*)
ultimately show *?case* **using** *red*
by(*fastforce elim*!: *rtranclp-trans rtranclp-tranclp-tranclp intro*: *Cond1Red elim*!: *Cond-τred1r-xt Cond-τred1t-xt simp add*: *no-call2-def*)
next
case (*bisim1While3 c n e′ xs stk loc pc xcp e*)
note *IH* = *bisim1While3.IH*(*2*)
note *bisim* = ‹*P*,*c*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim1* = ‹$\bigwedge$*xs*. *P*,*e*,*h* ⊢ (*e*, *xs*) ↔ ([], *xs*, *0*, *None*)›
from ‹*n* + *max-vars* (*if* (*e′*) (*e*;;*while* (*c*) *e*) *else unit*) ≤ *length xs*›
have *len*: *n* + *max-vars e′* ≤ *length xs* **by** *simp*
note *bsok* = ‹*bsok* (*while* (*c*) *e*) *n*›
note *exec* = ‹*?exec* (*while* (*c*) *e*) *stk loc pc xcp stk′ loc′ pc′ xcp′*›
from *bisim* **have** *pc*: *pc* ≤ *length* (*compE2 c*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 c*))
case *True*
let *?post* = *IfFalse* (*int* (*length* (*compE2 e*)) + *3*) # *compE2 e* @ [*Pop*, *Goto* (−*2* + (− *int* (*length* (*compE2 e*)) − *int* (*length* (*compE2 c*)))), *Push Unit*]
from *exec* **have** *exec-meth-d* (*compP2 P*) (*compE2 c* @ *?post*) (*compxE2 c 0 0* @ *shift* (*length* (*compE2 c*)) (*compxE2 e* (*Suc 0*) *0*)) *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *shift-compxE2 exec-move-def*)
hence *?exec c stk loc pc xcp stk′ loc′ pc′ xcp′*
using *True* **unfolding** *exec-move-def* **by**(*rule exec-meth-take-xt*)
from *IH*[*OF this len* - ‹*P*,*h* ⊢ *stk* [:≤] *ST*› ‹*conf-xcp′* (*compP2 P*) *h xcp*›] *bsok* **obtain** *e″ xs″*
where *bisim′*: *P*,*c*,*h′* ⊢ (*e″*, *xs″*) ↔ (*stk′*, *loc′*, *pc′*, *xcp′*)
and *red*: *?red e′ xs e″ xs″ c stk pc pc′ xcp xcp′* **by** *auto*
from *bisim′*
have *P*,*while* (*c*) *e*,*h′* ⊢ (*if* (*e″*) (*e*;;*while*(*c*) *e*) *else unit*, *xs″*) ↔ (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*rule bisim1-bisims1.bisim1While3*)
moreover have *τmove2* (*compP2 P*) *h stk* (*while* (*c*) *e*) *pc xcp* = *τmove2* (*compP2 P*) *h stk c pc xcp* **using** *True*
by(*simp add*: *τmove2-iff*)
ultimately show *?thesis* **using** *red*

by(*fastforce elim*!: *rtranclp-trans intro*: *Cond1Red elim*!: *Cond-τred1r-xt Cond-τred1t-xt simp add*: *no-call2-def*)
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compE2 c*) **by** *simp*
with *bisim* **obtain** *v* **where** *stk*: *stk* = [*v*] **and** *xcp*: *xcp* = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim pc len bsok* **have** *red*: *τred1r P t h* (*e′*, *xs*) (*Val v*, *loc*)
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *τred1r P t h* (*if* (*e′*) (*e*;; *while* (*c*) *e*) *else unit*, *xs*) (*if* (*Val v*) (*e*;; *while* (*c*) *e*) *else unit*, *loc*)
by(*rule Cond-τred1r-xt*)
moreover have *τ*: *τmove2* (*compP2 P*) *h* [*v*] (*while* (*c*) *e*) (*length* (*compE2 c*)) *None* **by**(*simp add*: *τmove2-iff*)
moreover have *τmove1 P h* (*if* (*Val v*) (*e*;;*while* (*c*) *e*) *else unit*) **by**(*rule τmove1CondRed*)
moreover from *bisim1*[*of loc*]
have *P*,*while* (*c*) *e*,*h* ⊢ (*e*;;*while*(*c*) *e*, *loc*) ↔ ([], *loc*, *Suc* (*length* (*compE2 c*) + *0*), *None*)
by(*rule bisim1While4*)
moreover
have *P*,*while* (*c*) *e*,*h* ⊢ (*unit*, *loc*) ↔ ([], *loc*, *Suc* (*Suc* (*Suc* (*length* (*compE2 c*) + *length* (*compE2 e*)))), *None*)
by(*rule bisim1While7*)
moreover from *exec stk xcp* **have** *typeof*$_h$ *v* = ⌊*Boolean*⌋
by(*auto simp add*: *exec-meth-instr exec-move-def*)
moreover have *nat* (*int* (*length* (*compE2 c*)) + (*3* + *int* (*length* (*compE2 e*)))) = *Suc* (*Suc* (*Suc* (*length* (*compE2 c*) + *length* (*compE2 e*)))) **by** *simp*
ultimately show *?thesis* **using** *exec stk xcp*
by(*fastforce elim*!: *exec-meth.cases rtranclp-trans intro*: *Red1CondT Red1CondF simp add*: *eval-nat-numeral exec-move-def*)
qed
next
case (*bisim1While4 e n e′ xs stk loc pc xcp c*)
note *IH* = *bisim1While4.IH*(*2*)
note *bisim* = ‹⋀*xs*. *P*,*c*,*h* ⊢ (*c*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *bisim1* = ‹*P*,*e*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
from ‹*n* + *max-vars* (*e′*;; *while* (*c*) *e*) ≤ *length xs*›
have *len*: *n* + *max-vars e′* ≤ *length xs* **by** *simp*
note *exec* = ‹*?exec* (*while* (*c*) *e*) *stk loc* (*Suc* (*length* (*compE2 c*) + *pc*)) *xcp stk′ loc′ pc′ xcp′*›
note *bsok* = ‹*bsok* (*while* (*c*) *e*) *n*›
from *bisim1* **have** *pc*: *pc* ≤ *length* (*compE2 e*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e*))
case *True*
let *?pre* = *compE2 c* @ [*IfFalse* (*int* (*length* (*compE2 e*)) + *3*)]
let *?post* = [*Pop*, *Goto* (−*2* + (− *int* (*length* (*compE2 e*)) − *int* (*length* (*compE2 c*)))), *Push Unit*]
from *exec* **have** *exec-meth-d* (*compP2 P*) ((*?pre* @ *compE2 e*) @ *?post*) (*compxE2 c 0 0* @ *shift* (*length ?pre*) (*compxE2 e 0 0*)) *t h* (*stk*, *loc*, *length ?pre* + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*simp add*: *shift-compxE2 exec-move-def*)
hence *exec′*: *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e*) (*compxE2 c 0 0* @ *shift* (*length ?pre*) (*compxE2 e 0 0*)) *t h* (*stk*, *loc*, *length ?pre* + *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*)
by(*rule exec-meth-take*)(*simp add*: *True*)
hence *?exec e stk loc pc xcp stk′ loc′* (*pc′* − *length ?pre*) *xcp′*
unfolding *exec-move-def* **by**(*rule exec-meth-drop-xt*) *auto*
from *IH*[*OF this len* - ‹*P*,*h* ⊢ *stk* [:≤] *ST*› ‹*conf-xcp′* (*compP2 P*) *h xcp*›] *bsok* **obtain** *e″ xs″*
where *bisim′*: *P*,*e*,*h′* ⊢ (*e″*, *xs″*) ↔ (*stk′*, *loc′*, *pc′* − *length ?pre*, *xcp′*)

```
    and red: ?red e' xs e'' xs'' e stk pc (pc' − length ?pre) xcp xcp' by auto
   from red have ?red (e';;while (c) e) xs (e'';;while (c) e) xs'' e stk pc (pc' − length ?pre) xcp xcp'
    by(fastforce intro: Seq1Red elim!: Seq-τred1r-xt Seq-τred1t-xt split: if-split-asm)
   moreover from bisim'
   have P,while (c) e,h' ⊢ (e'';;while(c) e, xs'') ↔ (stk', loc', Suc (length (compE2 c) + (pc' − length
?pre)), xcp')
    by(rule bisim1-bisims1.bisim1While4)
   moreover have τmove2 (compP2 P) h stk (while (c) e) (Suc (length (compE2 c) + pc)) xcp =
τmove2 (compP2 P) h stk e pc xcp
    using True by(simp add: τmove2-iff)
   moreover from exec' have pc' ≥ length ?pre
    by(rule exec-meth-drop-xt-pc) auto
   moreover have no-call2 e pc ⟹ no-call2 (while (c) e) (Suc (length (compE2 c) + pc))
    by(simp add: no-call2-def)
   ultimately show ?thesis
    apply(auto split: if-split-asm)
    apply(fastforce+)[6]
    apply(rule exI conjI|assumption|rule rtranclp.rtrancl-refl|simp)+
    done
 next
   case False
   with pc have [simp]: pc = length (compE2 e) by simp
   with bisim1 obtain v where stk: stk = [v] and xcp: xcp = None
    by(auto dest: bisim1-pc-length-compE2D)
   with bisim1 pc len bsok have red: τred1r P t h (e', xs) (Val v, loc)
    by(auto intro: bisim1-Val-τred1r simp add: bsok-def)
   hence τred1r P t h (e';; while (c) e, xs) (Val v;; while (c) e, loc) by(rule Seq-τred1r-xt)
    moreover have τ: τmove2 (compP2 P) h [v] (while (c) e) (Suc (length (compE2 c) + length
(compE2 e))) None
    by(simp add: τmove2-iff)
   moreover have τmove1 P h (Val v;;while (c) e) by(rule τmove1SeqRed)
   moreover
   have P,while (c) e,h ⊢ (while(c) e, loc) ↔ ([], loc, Suc (Suc (length (compE2 c) + length (compE2
e))), None)
    by(rule bisim1While6)
   ultimately show ?thesis using exec stk xcp
     by(fastforce elim!: exec-meth.cases rtranclp-trans intro: Red1Seq simp add: eval-nat-numeral
exec-move-def)
 qed
next
 case (bisim1While6 c n e xs)
 note exec = ‹?exec (while (c) e) [] xs (Suc (Suc (length (compE2 c) + length (compE2 e)))) None
stk' loc' pc' xcp'›
  moreover have τmove2 (compP2 P) h [] (while (c) e) (Suc (Suc (length (compE2 c) + length
(compE2 e)))) None
   by(simp add: τmove2-iff)
 moreover
 have P,while (c) e,h' ⊢ (if (c) (e;; while (c) e) else unit, xs) ↔ ([], xs, 0, None)
  by(rule bisim1While3[OF bisim1-refl])
 moreover have τred1t P t h (while (c) e, xs) (if (c) (e;; while (c) e) else unit, xs)
  by(rule tranclp.r-into-trancl)(auto intro: Red1While)
 ultimately show ?case
  by(fastforce elim!: exec-meth.cases simp add: exec-move-def)
next
```

```
  case (bisim1While7 c n e xs)
  note ‹?exec (while (c) e) [] xs (Suc (Suc (Suc (length (compE2 c) + length (compE2 e))))) None
stk' loc' pc' xcp'›
  moreover have τmove2 (compP2 P) h [] (while (c) e) (Suc (Suc (Suc (length (compE2 c) + length
(compE2 e))))) None
    by(simp add: τmove2-iff)
  moreover have P,while (c) e,h' ⊢ (unit, xs) ↔ ([Unit], xs, length (compE2 (while (c) e)), None)
    by(rule bisim1Val2) simp
  ultimately show ?case by(fastforce elim!: exec-meth.cases simp add: exec-move-def)
next
  case (bisim1WhileThrow1 c n a xs stk loc pc e)
  note exec = ‹?exec (while (c) e) stk loc pc ⌊a⌋ stk' loc' pc' xcp'›
  note bisim1 = ‹P,c,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
  from bisim1 have pc: pc < length (compE2 c) by(auto dest: bisim1-ThrowD)
  from bisim1 have match-ex-table (compP2 P) (cname-of h a) (0 + pc) (compxE2 c 0 0) = None
    unfolding compP2-def by(rule bisim1-xcp-Some-not-caught)
  with exec pc have False
    by(auto elim!: exec-meth.cases simp add: match-ex-table-not-pcs-None exec-move-def)
  thus ?case ..
next
  case (bisim1WhileThrow2 e n a xs stk loc pc c)
  note exec = ‹?exec (while (c) e) stk loc (Suc (length (compE2 c) + pc)) ⌊a⌋ stk' loc' pc' xcp'›
  note bisim = ‹P,e,h ⊢ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)›
  from bisim have pc: pc < length (compE2 e) by(auto dest: bisim1-ThrowD)
  from bisim have match-ex-table (compP2 P) (cname-of h a) (0 + pc) (compxE2 e 0 0) = None
    unfolding compP2-def by(rule bisim1-xcp-Some-not-caught)
  with exec pc have False
    apply(auto elim!: exec-meth.cases simp add: match-ex-table-not-pcs-None exec-move-def)
    apply(auto dest!: match-ex-table-shift-pcD simp only: compxE2-size-convs)
    apply simp
    done
  thus ?case ..
next
  case (bisim1Throw1 e n e' xs stk loc pc xcp)
  note IH = bisim1Throw1.IH(2)
  note exec = ‹?exec (throw e) stk loc pc xcp stk' loc' pc' xcp'›
  note bisim = ‹P,e,h ⊢ (e', xs) ↔ (stk, loc, pc, xcp)›
  note len = ‹n + max-vars (throw e') ≤ length xs›
  note bsok = ‹bsok (throw e) n›
  from bisim have pc: pc ≤ length (compE2 e) by(rule bisim1-pc-length-compE2)
  show ?case
  proof(cases pc < length (compE2 e))
    case True
    with exec have exec': ?exec e stk loc pc xcp stk' loc' pc' xcp' by(simp add: exec-move-Throw)
   from True have τmove2 (compP2 P) h stk (throw e) pc xcp = τmove2 (compP2 P) h stk e pc xcp
by(simp add: τmove2-iff)
    with IH[OF exec' - - ‹P,h ⊢ stk [:≤] ST› ‹conf-xcp' (compP2 P) h xcp›] len bsok show ?thesis
    by(fastforce intro: bisim1-bisims1.bisim1Throw1 Throw1Red elim!: Throw-τred1r-xt Throw-τred1t-xt
simp add: no-call2-def)
  next
    case False
    with pc have [simp]: pc = length (compE2 e) by simp
    with bisim obtain v where stk: stk = [v] and xcp: xcp = None
      by(auto dest: bisim1-pc-length-compE2D)
```

with *bisim pc len bsok* **have** *red*: τ*red1r P t h* (*e'*, *xs*) (*Val v*, *loc*)
by(*auto intro*: *bisim1-Val-*τ*red1r simp add*: *bsok-def*)
hence τ*red1r P t h* (*throw e'*, *xs*) (*throw* (*Val v*), *loc*) **by**(*rule Throw-*τ*red1r-xt*)
moreover have τ: τ*move2* (*compP2 P*) *h* [*v*] (*throw e*) *pc None* **by**(*simp add*: τ*move2-iff*)
moreover
have $\bigwedge a$. *P,throw e,h* $\vdash$ (*Throw a*, *loc*) $\leftrightarrow$ ([*Addr a*], *loc*, *length* (*compE2 e*), $\lfloor a \rfloor$)
by(*rule bisim1Throw2*)
moreover
have *P,throw e,h* $\vdash$ (*THROW NullPointer*, *loc*) $\leftrightarrow$ ([*Null*], *loc*, *length* (*compE2 e*), $\lfloor$*addr-of-sys-xcpt NullPointer*$\rfloor$)
by(*rule bisim1ThrowNull*)
moreover from *exec stk xcp* ‹*P,h* $\vdash$ *stk* [:$\leq$] *ST*› **obtain** *T'* **where** *T'*: *typeof*$_h$ *v* = $\lfloor T' \rfloor$ *P* $\vdash$ *T'* $\leq$ *Class Throwable*
by(*auto simp add*: *exec-move-def exec-meth-instr list-all2-Cons1 conf-def compP2-def*)
moreover with *T'* **have** *v* $\neq$ *Null* $\Longrightarrow$ $\exists$ *C*. *T'* = *Class C* **by**(*cases T'*)(*auto dest*: *Array-widen*)
moreover have τ*red1r P t h* (*throw null*, *loc*) (*THROW NullPointer*, *loc*)
by(*auto intro*: *r-into-rtranclp Red1ThrowNull* τ*move1ThrowNull*)
ultimately show *?thesis* **using** *exec stk xcp T'* **unfolding** *exec-move-def*
by(*cases v*)(*fastforce elim*!: *exec-meth.cases intro*: *rtranclp-trans*)+
qed
next
case (*bisim1Throw2 e n a xs*)
note *exec* = ‹*?exec* (*throw e*) [*Addr a*] *xs* (*length* (*compE2 e*)) $\lfloor a \rfloor$ *stk' loc' pc' xcp'*›
hence *False* **by**(*auto elim*!: *exec-meth.cases dest*: *match-ex-table-pc-length-compE2 simp add*: *exec-move-def*)
thus *?case* **..**
next
case (*bisim1ThrowNull e n xs*)
note *exec* = ‹*?exec* (*throw e*) [*Null*] *xs* (*length* (*compE2 e*)) $\lfloor$*addr-of-sys-xcpt NullPointer*$\rfloor$ *stk' loc' pc' xcp'*›
hence *False* **by**(*auto elim*!: *exec-meth.cases dest*: *match-ex-table-pc-length-compE2 simp add*: *exec-move-def*)
thus *?case* **..**
next
case (*bisim1ThrowThrow e n a xs stk loc pc*)
note *exec* = ‹*?exec* (*throw e*) *stk loc pc* $\lfloor a \rfloor$ *stk' loc' pc' xcp'*›
note *bisim1* = ‹*P,e,h* $\vdash$ (*Throw a*, *xs*) $\leftrightarrow$ (*stk*, *loc*, *pc*, $\lfloor a \rfloor$)›
from *bisim1* **have** *pc*: *pc* < *length* (*compE2 e*) **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim1* **have** *match-ex-table* (*compP2 P*) (*cname-of h a*) (*0* + *pc*) (*compxE2 e 0 0*) = *None*
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)
with *exec pc* **have** *False* **by**(*auto elim*!: *exec-meth.cases simp add*: *exec-move-def*)
thus *?case* **..**
next
case (*bisim1Try e n e' xs stk loc pc xcp e2 C' V*)
note *IH* = *bisim1Try.IH*(*2*)
note *bisim* = ‹*P,e,h* $\vdash$ (*e'*, *xs*) $\leftrightarrow$ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim1* = ‹$\bigwedge$*xs*. *P,e2,h* $\vdash$ (*e2*, *xs*) $\leftrightarrow$ ([], *xs*, *0*, *None*)›
note *exec* = ‹*?exec* (*try e catch*(*C' V*) *e2*) *stk loc pc xcp stk' loc' pc' xcp'*›
note *bsok* = ‹*bsok* (*try e catch*(*C' V*) *e2*) *n*›
with ‹*n* + *max-vars* (*try e' catch*(*C' V*) *e2*) $\leq$ *length xs*›
have *len*: *n* + *max-vars e'* $\leq$ *length xs* **and** *V*: *V* < *length xs* **by** *simp-all*
from *bisim* **have** *pc*: *pc* $\leq$ *length* (*compE2 e*) **by**(*rule bisim1-pc-length-compE2*)
show *?case*
proof(*cases pc* < *length* (*compE2 e*))
case *True*
note *pc* = *True*

show *?thesis*
proof(*cases xcp = None ∨ (∃ a'. xcp = ⌊a'⌋ ∧ match-ex-table (compP2 P) (cname-of h a') pc (compxE2 e 0 0) ≠ None*))
case *False*
then obtain *a'* **where** *Some*: *xcp = ⌊a'⌋*
and *True*: *match-ex-table (compP2 P) (cname-of h a') pc (compxE2 e 0 0) = None* **by**(*auto simp del*: *not-None-eq*)
from *Some bisim ‹conf-xcp' (compP2 P) h xcp›* **have** *∃ C. typeof_h (Addr a') = ⌊Class C⌋ ∧ P ⊢ C ⪯* Throwable*
by(*auto simp add*: *compP2-def*)
then obtain *C''* **where** *ha'*: *typeof-addr h a' = ⌊Class-type C''⌋*
and *subclsThrow*: *P ⊢ C'' ⪯* Throwable* **by**(*auto*)
with *exec True Some pc* **have** *subcls*: *P ⊢ C'' ⪯* C'*
apply(*auto elim*!: *exec-meth.cases simp add*: *match-ex-table-append compP2-def matches-ex-entry-def exec-move-def cname-of-def split*: *if-split-asm*)
apply(*simp only*: *compxE2-size-convs*, *simp*)
done
moreover from *ha' subclsThrow bsok* **have** *red*: *τred1r P t h (e', xs) (Throw a', loc)*
and *bisim'*: *P,e,h ⊢ (Throw a', loc) ↔ (stk, loc, pc, ⌊a'⌋)* **using** *bisim True len*
unfolding *Some compP2-def* **by**(*auto dest*!: *bisim1-xcp-τRed simp add*: *bsok-def*)
from *red* **have** *lenloc*: *length loc = length xs* **by**(*rule τred1r-preserves-len*)
from *red* **have** *τred1r P t h (try e' catch(C' V) e2, xs) (try (Throw a') catch(C' V) e2, loc)*
by(*rule Try-τred1r-xt*)
hence *τred1r P t h (try e' catch(C' V) e2, xs) ({V:Class C'=None; e2}, loc[V := Addr a'])*
using *ha' subcls V* **unfolding** *lenloc[symmetric]*
by(*auto intro*: *rtranclp.rtrancl-into-rtrancl Red1TryCatch τmove1TryThrow*)
moreover from *pc* **have** *τmove2 (compP2 P) h stk (try e catch(C' V) e2) pc ⌊a'⌋* **by**(*simp add*: *τmove2-iff*)
moreover from *bisim' ha' subcls*
have *P,try e catch(C' V) e2,h ⊢ ({V:Class C'=None; e2}, loc[V := Addr a']) ↔ ([Addr a'], loc, Suc (length (compE2 e)), None)*
by(*rule bisim1TryCatch1*)
ultimately show *?thesis* **using** *exec True pc Some ha' subclsThrow*
apply(*auto elim*!: *exec-meth.cases simp add*: *ac-simps eval-nat-numeral match-ex-table-append matches-ex-entry-def compP2-def exec-move-def cname-of-def*)
apply *fastforce*
apply(*simp-all only*: *compxE2-size-convs*, *auto dest*: *match-ex-table-shift-pcD*)
done
next
case *True*
let *?post = Goto (int (length (compE2 e2)) + 2) # Store V # compE2 e2*
from *exec True* **have** *exec-meth-d (compP2 P) (compE2 e @ ?post) (compxE2 e 0 0 @ shift (length (compE2 e)) (compxE2 e2 (Suc (Suc 0)) 0)) t h (stk, loc, pc, xcp) ta h' (stk', loc', pc', xcp')*
by(*auto elim*!: *exec-meth.cases intro*: *exec-meth.intros simp add*: *match-ex-table-append shift-compxE2 exec-move-def*)
hence *?exec e stk loc pc xcp stk' loc' pc' xcp'*
using *pc* **unfolding** *exec-move-def* **by**(*rule exec-meth-take-xt*)
from *IH[OF this len - ‹P,h ⊢ stk [:≤] ST› ‹conf-xcp' (compP2 P) h xcp›] bsok* **obtain** *e'' xs''*
where *bisim'*: *P,e,h' ⊢ (e'', xs'') ↔ (stk', loc', pc', xcp')*
and *red'*: *?red e' xs e'' xs'' e stk pc pc' xcp xcp'* **by** *auto*
from *bisim'*
have *P,try e catch(C' V) e2,h' ⊢ (try e'' catch(C' V) e2, xs'') ↔ (stk', loc', pc', xcp')*
by(*rule bisim1-bisims1.bisim1Try*)
moreover from *pc* **have** *τmove2 (compP2 P) h stk (try e catch(C' V) e2) pc xcp = τmove2*

(*compP2 P*) *h stk e pc xcp*
by(*simp add*: *τmove2-iff*)
ultimately show *?thesis* **using** *red'* **by**(*fastforce intro*: *Try1Red elim*!: *Try-τred1r-xt Try-τred1t-xt simp add*: *no-call2-def*)
qed
next
case *False*
with *pc* **have** [*simp*]: *pc* = *length* (*compE2 e*) **by** *simp*
with *bisim* **obtain** *v* **where** *stk*: *stk* = [*v*] **and** *xcp*: *xcp* = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim pc len bsok* **have** *red*: *τred1r P t h* (*e'*, *xs*) (*Val v*, *loc*)
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *τred1r P t h* (*try e' catch*(*C' V*) *e2*, *xs*) (*try* (*Val v*) *catch*(*C' V*) *e2*, *loc*) **by**(*rule Try-τred1r-xt*)
hence *τred1r P t h* (*try e' catch*(*C' V*) *e2*, *xs*) (*Val v*, *loc*)
by(*auto intro*: *rtranclp.rtrancl-into-rtrancl Red1Try τmove1TryRed*)
moreover have *τ*: *τmove2* (*compP2 P*) *h* [*v*] (*try e catch*(*C' V*) *e2*) *pc None* **by**(*simp add*: *τmove2-iff*)
moreover
have *P*,*try e catch*(*C' V*) *e2*,*h* ⊢ (*Val v*, *loc*) ↔ ([*v*], *loc*, *length* (*compE2* (*try e catch*(*C' V*) *e2*)), *None*)
by(*rule bisim1Val2*) *simp*
moreover have *nat* (*int* (*length* (*compE2 e*)) + (*int* (*length* (*compE2 e2*)) + 2)) = *length* (*compE2* (*try e catch*(*C' V*) *e2*)) **by** *simp*
ultimately show *?thesis* **using** *exec stk xcp*
by(*fastforce elim*!: *exec-meth.cases simp add*: *exec-move-def*)
qed
next
case (*bisim1TryCatch1 e n a xs stk loc pc C'' C' e2 V*)
note *exec* = ‹*?exec* (*try e catch*(*C' V*) *e2*) [*Addr a*] *loc* (*Suc* (*length* (*compE2 e*))) *None stk' loc' pc' xcp'*›
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e2*, *loc*[*V* := *Addr a*]) ↔ ([], *loc*[*V* := *Addr a*], *0*, *None*)›
note *bisim1* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
hence [*simp*]: *xs* = *loc* **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim2*
have *P*, *try e catch*(*C' V*) *e2*, *h* ⊢ ({*V*:*Class C'*=*None*; *e2*}, *loc*[*V* := *Addr a*]) ↔ ([], *loc*[*V* := *Addr a*], *Suc* (*Suc* (*length* (*compE2 e*) + *0*)), *None*)
by(*rule bisim1TryCatch2*)
moreover have *τmove2* (*compP2 P*) *h* [*Addr a*] (*try e catch*(*C' V*) *e2*) (*Suc* (*length* (*compE2 e*))) *None* **by**(*simp add*: *τmove2-iff*)
ultimately show *?case* **using** *exec* **by**(*fastforce elim*!: *exec-meth.cases simp add*: *exec-move-def*)
next
case (*bisim1TryCatch2 e2 n e' xs stk loc pc xcp e C' V*)
note *IH* = *bisim1TryCatch2.IH*(*2*)
note *bisim2* = ‹*P*,*e2*,*h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim* = ‹⋀*xs*. *P*,*e*,*h* ⊢ (*e*, *xs*) ↔ ([], *xs*, *0*, *None*)›
note *exec* = ‹*?exec* (*try e catch*(*C' V*) *e2*) *stk loc* (*Suc* (*Suc* (*length* (*compE2 e*) + *pc*))) *xcp stk' loc' pc' xcp'*›
note *bsok* = ‹*bsok* (*try e catch*(*C' V*) *e2*) *n*›
with ‹*n* + *max-vars* {*V*:*Class C'*=*None*; *e'*} ≤ *length xs*›
have *len*: *Suc n* + *max-vars e'* ≤ *length xs* **and** *V*: *V* < *length xs* **by** *simp-all*
let *?pre* = *compE2 e* @ [*Goto* (*int* (*length* (*compE2 e2*)) + *2*), *Store V*]
from *exec* **have** *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e2*)
(*compxE2 e 0 0* @ *shift* (*length ?pre*) (*compxE2 e2 0 0*) @ [(*0*, *length* (*compE2 e*), ⌊*C'*⌋, *Suc* (*length*

(*compE2 e*)), *0*)]) *t*
h (*stk*, *loc*, *length ?pre* + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *shift-compxE2 exec-move-def*)
hence *exec'*: *exec-meth-d* (*compP2 P*) (*?pre* @ *compE2 e2*) (*compxE2 e 0 0* @ *shift* (*length ?pre*) (*compxE2 e2 0 0*)) *t*
h (*stk*, *loc*, *length ?pre* + *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*auto elim*!: *exec-meth.cases intro*: *exec-meth.intros simp add*: *match-ex-table-append matches-ex-entry-def*)
hence *?exec e2 stk loc pc xcp stk' loc'* (*pc'* − *length ?pre*) *xcp'*
unfolding *exec-move-def* **by**(*rule exec-meth-drop-xt*) *auto*
from *IH*[*OF this len* - ‹*P*,*h* ⊢ *stk* [:≤] *ST*› ‹*conf-xcp'* (*compP2 P*) *h xcp*›] *bsok* **obtain** *e'' xs''*
where *bisim'*: *P*,*e2*,*h'* ⊢ (*e''*, *xs''*) ↔ (*stk'*, *loc'*, *pc'* − *length ?pre*, *xcp'*)
and *red*: *?red e' xs e'' xs'' e2 stk pc* (*pc'* − *length ?pre*) *xcp xcp'* **by** *auto*
from *red* **have** *length xs'' = length xs*
by(*auto dest*!: *τred1r-preserves-len τred1t-preserves-len red1-preserves-len split*: *if-split-asm*)
with *red V* **have** *?red* {*V*:*Class C'*=*None*; *e'*} *xs* {*V*:*Class C'*=*None*; *e''*} *xs'' e2 stk pc* (*pc'* − *length ?pre*) *xcp xcp'*
by(*fastforce elim*!: *Block-None-τred1r-xt Block-None-τred1t-xt intro*: *Block1Red split*: *if-split-asm*)
moreover
from *bisim'*
have *P*,*try e catch*(*C'* *V*) *e2*,*h'* ⊢ ({*V*:*Class C'*=*None*;*e''*}, *xs''*) ↔ (*stk'*, *loc'*, *Suc* (*Suc* (*length* (*compE2 e*) + (*pc'* − *length ?pre*))), *xcp'*)
by(*rule bisim1-bisims1.bisim1TryCatch2*)
moreover have *τmove2* (*compP2 P*) *h stk* (*try e catch*(*C'* *V*) *e2*) (*Suc* (*Suc* (*length* (*compE2 e*) + *pc*))) *xcp* = *τmove2* (*compP2 P*) *h stk e2 pc xcp*
by(*simp add*: *τmove2-iff*)
moreover from *exec'* **have** *pc'* ≥ *length ?pre*
by(*rule exec-meth-drop-xt-pc*) *auto*
moreover hence *Suc* (*Suc* (*pc'* − *Suc* (*Suc 0*))) = *pc'* **by** *simp*
moreover have *no-call2 e2 pc* ⟹ *no-call2* (*try e catch*(*C'* *V*) *e2*) (*Suc* (*Suc* (*length* (*compE2 e*) + *pc*)))
by(*simp add*: *no-call2-def*)
ultimately show *?case* **using** *red V* **by**(*fastforce simp add*: *eval-nat-numeral split*: *if-split-asm*)
next
case (*bisim1TryFail e n a xs stk loc pc C'' C' e2 V*)
note *bisim* = ‹*P*,*e*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
from *bisim* **have** *pc*: *pc* < *length* (*compE2 e*) **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim* **have** *match-ex-table* (*compP2 P*) (*cname-of h a*) (*0* + *pc*) (*compxE2 e 0 0*) = *None*
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)
with ‹*?exec* (*try e catch*(*C'* *V*) *e2*) *stk loc pc* ⌊*a*⌋ *stk' loc' pc' xcp'*› *pc* ‹*typeof-addr h a* = ⌊*Class-type C''*⌋› ‹¬ *P* ⊢ *C''* ≼* *C'*›
have *False* **by**(*auto elim*!: *exec-meth.cases simp add*: *matches-ex-entry-def compP2-def match-ex-table-append-not-pcs exec-move-def cname-of-def*)
thus *?case* **..**
next
case (*bisim1TryCatchThrow e2 n a xs stk loc pc e C' V*)
note *bisim* = ‹*P*,*e2*,*h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)›
from *bisim* **have** *pc*: *pc* < *length* (*compE2 e2*) **by**(*auto dest*: *bisim1-ThrowD*)
from *bisim* **have** *match-ex-table* (*compP2 P*) (*cname-of h a*) (*0* + *pc*) (*compxE2 e2 0 0*) = *None*
unfolding *compP2-def* **by**(*rule bisim1-xcp-Some-not-caught*)
with ‹*?exec* (*try e catch*(*C'* *V*) *e2*) *stk loc* (*Suc* (*Suc* (*length* (*compE2 e*) + *pc*))) ⌊*a*⌋ *stk' loc' pc' xcp'*› *pc*
have *False* **apply**(*auto elim*!: *exec-meth.cases simp add*: *compxE2-size-convs match-ex-table-append-not-pcs exec-move-def*)
apply(*auto dest*!: *match-ex-table-shift-pcD simp add*: *match-ex-table-append matches-ex-entry-def*

compP2-def)
done
thus *?case* **..**
next
case *bisims1Nil*
hence *False* **by**(*auto elim*!: *exec-meth.cases simp add*: *exec-moves-def*)
thus *?case* **..**
next
case (*bisims1List1 e n e' xs stk loc pc xcp es*)
note *IH1* = *bisims1List1.IH*(*2*)
note *IH2* = *bisims1List1.IH*(*4*)
note *exec* = ‹*?execs* (*e* # *es*) *stk loc pc xcp stk' loc' pc' xcp'*›
note *bisim1* = ‹*P*,*e*,*h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)›
note *bisim2* = ‹*P*,*es*,*h* ⊢ (*es*, *loc*) [↔] ([], *loc*, *0*, *None*)›
note *len* = ‹*n* + *max-varss* (*e'* # *es*) ≤ *length xs*›
note *bsok* = ‹*bsoks* (*e* # *es*) *n*›
from *bisim1* **have** *pc*: *pc* ≤ *length* (*compE2 e*) **by**(*rule bisim1-pc-length-compE2*)
from *bisim1* **have** *lenxs*: *length xs* = *length loc* **by**(*rule bisim1-length-xs*)
show *?case*
proof(*cases pc* < *length* (*compE2 e*))
case *True*
with *exec* **have** *exec'*: *?exec e stk loc pc xcp stk' loc' pc' xcp'*
by(*auto simp add*: *compxEs2-size-convs exec-moves-def exec-move-def intro*: *exec-meth-take-xt*)
from *True* **have** *τmoves2* (*compP2 P*) *h stk* (*e* # *es*) *pc xcp* = *τmove2* (*compP2 P*) *h stk e pc xcp*
by(*simp add*: *τmove2-iff τmoves2-iff*)
moreover from *True* **have** *no-calls2* (*e* # *es*) *pc* = *no-call2 e pc*
by(*simp add*: *no-call2-def no-calls2-def*)
ultimately show *?thesis*
using *IH1*[*OF exec'* - - ‹*P*,*h* ⊢ *stk* [:≤] *ST*› ‹*conf-xcp'* (*compP2 P*) *h xcp*›] *bsok len*
by(*fastforce intro*: *bisim1-bisims1.bisims1List1 elim*!: *τred1r-inj-τreds1r τred1t-inj-τreds1t List1Red1*)
next
case *False*
with *pc* **have** *pc* [*simp*]: *pc* = *length* (*compE2 e*) **by** *simp*
with *bisim1* **obtain** *v* **where** *stk*: *stk* = [*v*] **and** *xcp*: *xcp* = *None*
and *v*: *is-val e'* ⟹ *e'* = *Val v* ∧ *xs* = *loc* **and** *call*: *call1 e'* = *None*
by(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim1 pc len bsok* **have** *red*: *τred1r P t h* (*e'*, *xs*) (*Val v*, *loc*)
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
hence *τreds1r P t h* (*e'* # *es*, *xs*) (*Val v* # *es*, *loc*) **by**(*rule τred1r-inj-τreds1r*)
moreover from *exec stk xcp*
have *exec'*: *exec-meth-d* (*compP2 P*) (*compE2 e* @ *compEs2 es*) (*compxE2 e 0 0* @ *shift* (*length* (*compE2 e*)) (*stack-xlift* (*length* [*v*]) (*compxEs2 es 0 0*))) *t h* ([] @ [*v*], *loc*, *length* (*compE2 e*) + *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
by(*simp add*: *shift-compxEs2 stack-xlift-compxEs2 exec-moves-def*)
hence *exec-meth-d* (*compP2 P*) (*compEs2 es*) (*stack-xlift* (*length* [*v*]) (*compxEs2 es 0 0*)) *t h* ([] @ [*v*], *loc*, *0*, *None*) *ta h'* (*stk'*, *loc'*, *pc'* − *length* (*compE2 e*), *xcp'*)
by(*rule exec-meth-drop-xt*) *auto*
with *bisim2* **obtain** *stk''* **where** *stk'*: *stk'* = *stk''* @ [*v*]
and *exec''*: *exec-moves-d P t es h* ([], *loc*, *0*, *None*) *ta h'* (*stk''*, *loc'*, *pc'* − *length* (*compE2 e*), *xcp'*)
by(*unfold exec-moves-def*)(*drule* (*1*) *exec-meth-stk-splits*, *auto*)
from *IH2*[*OF exec''*] *len lenxs bsok* **obtain** *es'' xs''*
where *bisim'*: *P*,*es*,*h'* ⊢ (*es''*, *xs''*) [↔] (*stk''*, *loc'*, *pc'* − *length* (*compE2 e*), *xcp'*)
and *red'*: *?reds es loc es'' xs'' es* [] *0* (*pc'* − *length* (*compE2 e*)) *None xcp'* **by** *fastforce*

from *bisim′* **have** *P,e # es,h′ ⊢ (Val v # es″, xs″) [↔] (stk″ @ [v], loc′, length (compE2 e) + (pc′ − length (compE2 e)), xcp′)*
by(*rule bisims1List2*)
moreover from *exec″*
have *τmoves2 (compP2 P) h [v] (e # es) (length (compE2 e)) None = τmoves2 (compP2 P) h [] es 0 None*
using *τinstr-stk-drop-exec-moves*[**where** *stk*=[] **and** *vs*=[*v*]] **by**(*simp add*: *τmoves2-iff*)
moreover have *τ*: ⋀*es′*. *τmoves1 P h (Val v # es′) ⟹ τmoves1 P h es′* **by** *simp*
from *exec′* **have** *pc′ ≥ length (compE2 e)*
by(*rule exec-meth-drop-xt-pc*) *auto*
moreover have *no-calls2 es 0 ⟹ no-calls2 (e # es) (length (compE2 e))*
by(*simp add*: *no-calls2-def*)
ultimately show *?thesis* **using** *red′ xcp stk stk′ call v*
apply(*auto simp add*: *split-paired-Ex*)
apply(*blast 25 intro*: *rtranclp-trans rtranclp-tranclp-tranclp τreds1r-cons-τreds1r List1Red2 τreds1t-cons-τreds1t dest*: *τ*)+
done
qed
next
case (*bisims1List2 es n es′ xs stk loc pc xcp e v*)
note *IH = bisims1List2.IH*(*2*)
note *exec = ‹?execs (e # es) (stk @ [v]) loc (length (compE2 e) + pc) xcp stk′ loc′ pc′ xcp′›*
note *bisim1 = ‹P,e,h ⊢ (e, xs) ↔ ([], xs, 0, None)›*
note *bisim2 = ‹P,es,h ⊢ (es′, xs) [↔] (stk, loc, pc, xcp)›*
note *len = ‹n + max-varss (Val v # es′) ≤ length xs›*
note *bsok = ‹bsoks (e # es) n›*
from *‹P,h ⊢ stk @ [v] [:≤] ST›* **obtain** *ST′* **where** *ST′*: *P,h ⊢ stk [:≤] ST′*
by(*auto simp add*: *list-all2-append1*)
from *exec* **have** *exec′*: *exec-meth-d (compP2 P) (compE2 e @ compEs2 es) (compxE2 e 0 0 @ shift (length (compE2 e)) (stack-xlift (length [v]) (compxEs2 es 0 0))) t h (stk @ [v], loc, length (compE2 e) + pc, xcp) ta h′ (stk′, loc′, pc′, xcp′)*
by(*simp add*: *shift-compxEs2 stack-xlift-compxEs2 exec-moves-def*)
hence *exec-meth-d (compP2 P) (compEs2 es) (stack-xlift (length [v]) (compxEs2 es 0 0)) t h (stk @ [v], loc, pc, xcp) ta h′ (stk′, loc′, pc′ − length (compE2 e), xcp′)*
by(*rule exec-meth-drop-xt*) *auto*
with *bisim2* **obtain** *stk″* **where** *stk′*: *stk′ = stk″ @ [v]*
and *exec″*: *exec-moves-d P t es h (stk, loc, pc, xcp) ta h′ (stk″, loc′, pc′ − length (compE2 e), xcp′)*
by(*unfold exec-moves-def*)(*drule* (*1*) *exec-meth-stk-splits*, *auto*)
from *IH*[*OF exec″ - - ST′ ‹conf-xcp′ (compP2 P) h xcp›*] *len bsok* **obtain** *es″ xs″*
where *bisim′*: *P,es,h′ ⊢ (es″, xs″) [↔] (stk″, loc′, pc′ − length (compE2 e), xcp′)*
and *red′*: *?reds es′ xs es″ xs″ es stk pc (pc′ − length (compE2 e)) xcp xcp′* **by** *auto*
from *bisim′* **have** *P,e # es,h′ ⊢ (Val v # es″, xs″) [↔] (stk″ @ [v], loc′, length (compE2 e) + (pc′ − length (compE2 e)), xcp′)*
by(*rule bisim1-bisims1.bisims1List2*)
moreover from *exec″* **have** *τmoves2 (compP2 P) h (stk @ [v]) (e # es) (length (compE2 e) + pc) xcp = τmoves2 (compP2 P) h stk es pc xcp*
by(*auto simp add*: *τmoves2-iff τinstr-stk-drop-exec-moves*)
moreover have *τ*: ⋀*es′*. *τmoves1 P h (Val v # es′) ⟹ τmoves1 P h es′* **by** *simp*
from *exec′* **have** *pc′ ≥ length (compE2 e)*
by(*rule exec-meth-drop-xt-pc*) *auto*
moreover have *no-calls2 es pc ⟹ no-calls2 (e # es) (length (compE2 e) + pc)*
by(*simp add*: *no-calls2-def*)
ultimately show *?case* **using** *red′ stk′*
by(*auto split*: *if-split-asm simp add*: *split-paired-Ex*)(*blast intro*: *rtranclp-trans rtranclp-tranclp-tranclp*

τreds1r-cons-τreds1r List1Red2 τreds1t-cons-τreds1t dest: *τ*)+
qed

end

inductive *sim21-size-aux* :: *nat* ⇒ (*pc* × *'addr option*) ⇒ (*pc* × *'addr option*) ⇒ *bool*
for *len* :: *nat*
where
⟦ *pc1* ≤ *len*; *pc2* ≤ *len*; *xcp1* ≠ *None* ∧ *xcp2* = *None* ∧ *pc1* = *pc2* ∨ *xcp1* = *None* ∧ *pc1* > *pc2* ⟧
⟹ *sim21-size-aux len* (*pc1*, *xcp1*) (*pc2*, *xcp2*)

definition *sim21-size* :: *'addr jvm-prog* ⇒ *'addr jvm-thread-state* ⇒ *'addr jvm-thread-state* ⇒ *bool*
where
sim21-size P xcpfrs xcpfrs' ⟷
(*xcpfrs*, *xcpfrs'*) ∈
inv-image (*less-than* <*lex*> *same-fst* (λ*n*. *True*) (λ*n*. {(*pcxcp*, *pcxcp'*). *sim21-size-aux n pcxcp pcxcp'*}))
(λ(*xcp*, *frs*). (*length frs*, *case frs of* [] ⇒ *undefined*
| (*stk*, *loc*, *C*, *M*, *pc*) # *frs* ⇒ (*length* (*fst* (*snd* (*snd* (*the* (*snd* (*snd* (*snd* (*method P C M*)))))))), *pc*, *xcp*)))

lemma *wfP-sim21-size-aux*: *wfP* (*sim21-size-aux n*)
proof −
let *?f* = λ(*pc*, *xcp*). *case xcp of None* ⇒ *Suc* (2 ∗ (*n* − *pc*)) | *Some* - ⇒ 2 ∗ (*n* − *pc*)
have *wf* {(*m*, *m'*). (*m* :: *nat*) < *m'*} **by**(*rule wf-less*)
hence *wf* (*inv-image* {(*m*, *m'*). *m* < *m'*} *?f*) **by**(*rule wf-inv-image*)
moreover have {(*pcxcp1*, *pcxcp2*). *sim21-size-aux n pcxcp1 pcxcp2*} ⊆ *inv-image* {(*m*, *m'*). *m* < *m'*} *?f*
by(*auto elim*!: *sim21-size-aux.cases*)
ultimately show *?thesis* **unfolding** *wfp-def* **by**(*rule wf-subset*)
qed

lemma *Collect-split-mem*: {(*x*, *y*). (*x*, *y*) ∈ *Q*} = *Q* **by** *simp*

lemma *wfP-sim21-size*: *wfP* (*sim21-size P*)
unfolding *wfp-def Collect-split-mem sim21-size-def* [*abs-def*]
apply(*rule wf-inv-image*)
apply(*rule wf-lex-prod*)
apply(*rule wf-less-than*)
apply(*rule wf-same-fst*)
apply(*rule wfP-sim21-size-aux*[*unfolded wfp-def*])
done

declare *split-beta*[*simp*]

context *J1-JVM-heap-base* **begin**

lemma *bisim1-Invoke-τRed*:
⟦ *P*,*E*,*h* ⊢ (*e*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk'*, *loc*, *pc*, *None*); *pc* < *length* (*compE2 E*);
compE2 E ! *pc* = *Invoke M* (*length vs*); *n* + *max-vars e* ≤ *length xs*; *bsok E n* ⟧
⟹ ∃ *e' xs'*. *τred1r P t h* (*e*, *xs*) (*e'*, *xs'*) ∧ *P*,*E*,*h* ⊢ (*e'*, *xs'*) ↔ (*rev vs* @ *Addr a* # *stk'*, *loc*, *pc*, *None*) ∧ *call1 e'* = ⌊(*a*, *M*, *vs*)⌋
(**is** ⟦ -; -; -; -; - ⟧ ⟹ *?concl e xs E n pc stk' loc*)

and *bisims1-Invoke-τReds*:
⟦ *P*,*Es*,*h* ⊢ (*es*, *xs*) [↔] (*rev vs* @ *Addr a* # *stk′*, *loc*, *pc*, *None*); *pc* < *length* (*compEs2 Es*);
compEs2 Es ! *pc* = *Invoke M* (*length vs*); *n* + *max-varss es* ≤ *length xs*; *bsoks Es n* ⟧
⟹ ∃ *es′ xs′*. *τreds1r P t h* (*es*, *xs*) (*es′*, *xs′*) ∧ *P*,*Es*,*h* ⊢ (*es′*, *xs′*) [↔] (*rev vs* @ *Addr a* # *stk′*, *loc*, *pc*, *None*) ∧ *calls1 es′* = ⌊(*a*, *M*, *vs*)⌋
(**is** ⟦ -; -; -; -; - ⟧ ⟹ *?concls es xs Es n pc stk′ loc*)
proof(*induct E n e xs stk*≡*rev vs* @ *Addr a* # *stk′ loc pc xcp*≡*None*::*′addr option*
and *Es n es xs stk*≡*rev vs* @ *Addr a* # *stk′ loc pc xcp*≡*None*::*′addr option*
arbitrary: *stk′* **and** *stk′ rule*: *bisim1-bisims1-inducts-split*)
case *bisim1Val2* **thus** *?case* **by** *simp*
next
case *bisim1New* **thus** *?case* **by** *simp*
next
case *bisim1NewArray* **thus** *?case*
by(*fastforce split*: *if-split-asm intro*: *bisim1-bisims1.bisim1NewArray dest*: *bisim1-pc-length-compE2 elim*!: *NewArray-τred1r-xt*)
next
case *bisim1Cast* **thus** *?case*
by(*fastforce split*: *if-split-asm intro*: *bisim1-bisims1.bisim1Cast dest*: *bisim1-pc-length-compE2 elim*!: *Cast-τred1r-xt*)
next
case *bisim1InstanceOf* **thus** *?case*
by(*fastforce split*: *if-split-asm intro*: *bisim1-bisims1.bisim1InstanceOf dest*: *bisim1-pc-length-compE2 elim*!: *InstanceOf-τred1r-xt*)
next
case *bisim1Val* **thus** *?case* **by** *simp*
next
case *bisim1Var* **thus** *?case* **by** *simp*
next
case *bisim1BinOp1* **thus** *?case*
apply(*auto split*: *if-split-asm intro*: *bisim1-bisims1.bisim1BinOp1 dest*: *bisim1-pc-length-compE2 elim*: *BinOp-τred1r-xt1*)
apply(*fastforce elim*!: *BinOp-τred1r-xt1 intro*: *bisim1-bisims1.bisim1BinOp1*)
done
next
case (*bisim1BinOp2 e2 n e′ xs stk loc pc e1 bop v1*)
note *IH* = ‹⋀*stk′*. ⟦*stk* = *rev vs* @ *Addr a* # *stk′*; *pc* < *length* (*compE2 e2*); *compE2 e2* ! *pc* = *Invoke M* (*length vs*); *n* + *max-vars e′* ≤ *length xs*; *bsok e2 n*
⟧ ⟹ *?concl e′ xs e2 n pc stk′ loc*›
note *inv* = ‹*compE2* (*e1* «*bop*» *e2*) ! (*length* (*compE2 e1*) + *pc*) = *Invoke M* (*length vs*)›
with ‹*length* (*compE2 e1*) + *pc* < *length* (*compE2* (*e1* «*bop*» *e2*))› **have** *pc*: *pc* < *length* (*compE2 e2*)
by(*auto split*: *bop.splits if-split-asm*)
moreover with *inv* **have** *compE2 e2* ! *pc* = *Invoke M* (*length vs*) **by** *simp*
moreover with ‹*P*,*e2*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)› *pc*
obtain *vs″ v″ stk″* **where** *stk* = *vs″* @ *v″* # *stk″* **and** *length vs″* = *length vs*
by(*auto dest*!: *bisim1-Invoke-stkD*)
with ‹*stk* @ [*v1*] = *rev vs* @ *Addr a* # *stk′*› **obtain** *stk‴*
where *stk‴*: *stk* = *rev vs* @ *Addr a* # *stk‴* **and** *stk*: *stk′* = *stk‴* @ [*v1*]
by(*cases stk′ rule*: *rev-cases*) *auto*
from ‹*n* + *max-vars* (*Val v1* «*bop*» *e′*) ≤ *length xs*› **have** *n* + *max-vars e′* ≤ *length xs* **by** *simp*
moreover from ‹*bsok* (*e1* «*bop*» *e2*) *n*› **have** *bsok e2 n* **by** *simp*
ultimately have *?concl e′ xs e2 n pc stk‴ loc* **using** *stk‴* **by**−(*rule IH*)
then obtain *e″ xs′* **where** *IH′*: *τred1r P t h* (*e′*, *xs*) (*e″*, *xs′*) *call1 e″* = ⌊(*a*, *M*, *vs*)⌋

and *bisim*: $P,e2,h \vdash (e'', xs') \leftrightarrow (rev\ vs\ @\ Addr\ a\ \#\ stk''', loc, pc, None)$ **by** *blast*
from *bisim* **have** $P,e1\ «bop»\ e2,h \vdash (Val\ v1\ «bop»\ e'', xs') \leftrightarrow ((rev\ vs\ @\ Addr\ a\ \#\ stk''')\ @\ [v1], loc, length\ (compE2\ e1) + pc, None)$
by(*rule bisim1-bisims1.bisim1BinOp2*)
with *IH' stk* **show** *?case* **by**(*fastforce elim*!: *BinOp-τred1r-xt2*)
next
case *bisim1LAss1* **thus** *?case*
by(*fastforce split*: *if-split-asm intro*: *bisim1-bisims1.bisim1LAss1 dest*: *bisim1-pc-length-compE2 elim*!: *LAss-τred1r*)
next
case *bisim1LAss2* **thus** *?case* **by** *simp*
next
case *bisim1AAcc1* **thus** *?case*
apply(*auto split*: *if-split-asm intro*: *bisim1-bisims1.bisim1AAcc1 dest*: *bisim1-pc-length-compE2 elim*!: *AAcc-τred1r-xt1*)
apply(*fastforce elim*!: *AAcc-τred1r-xt1 intro*: *bisim1-bisims1.bisim1AAcc1*)
done
next
case (*bisim1AAcc2 e2 n e' xs stk loc pc e1 v1*)
note $IH = ‹\bigwedge stk'.\ ⟦stk = rev\ vs\ @\ Addr\ a\ \#\ stk';\ pc < length\ (compE2\ e2);\ compE2\ e2\ !\ pc = Invoke\ M\ (length\ vs);\ n + max\text{-}vars\ e' \le length\ xs;\ bsok\ e2\ n ⟧ \Longrightarrow ?concl\ e'\ xs\ e2\ n\ pc\ stk'\ loc›$
note $inv = ‹compE2\ (e1⌊e2⌉)\ !\ (length\ (compE2\ e1) + pc) = Invoke\ M\ (length\ vs)›$
with $‹length\ (compE2\ e1) + pc < length\ (compE2\ (e1⌊e2⌉))›$ **have** *pc*: $pc < length\ (compE2\ e2)$
by(*auto split*: *if-split-asm*)
moreover with *inv* **have** $compE2\ e2\ !\ pc = Invoke\ M\ (length\ vs)$ **by** *simp*
moreover with $‹P,e2,h \vdash (e', xs) \leftrightarrow (stk, loc, pc, None)›\ pc$
obtain *vs'' v'' stk''* **where** $stk = vs'' @ v'' \#\ stk''$ **and** $length\ vs'' = length\ vs$
by(*auto dest*!: *bisim1-Invoke-stkD*)
with $‹stk @ [v1] = rev\ vs @ Addr\ a \#\ stk'›$ **obtain** *stk'''*
where *stk'''*: $stk = rev\ vs @ Addr\ a \#\ stk'''$ **and** *stk*: $stk' = stk''' @ [v1]$
by(*cases stk' rule*: *rev-cases*) *auto*
from $‹n + max\text{-}vars\ (Val\ v1⌊e'⌉) \le length\ xs›$ **have** $n + max\text{-}vars\ e' \le length\ xs$ **by** *simp*
moreover from $‹bsok\ (e1⌊e2⌉)\ n›$ **have** $bsok\ e2\ n$ **by** *simp*
ultimately have *?concl e' xs e2 n pc stk''' loc* **using** *stk'''* **by**−(*rule IH*)
then obtain *e'' xs'* **where** *IH'*: $\tau red1r\ P\ t\ h\ (e', xs)\ (e'', xs')\ call1\ e'' = ⌊(a, M, vs)⌋$
and *bisim*: $P,e2,h \vdash (e'', xs') \leftrightarrow (rev\ vs @ Addr\ a \#\ stk''', loc, pc, None)$ **by** *blast*
from *bisim* **have** $P,e1⌊e2⌉,h \vdash (Val\ v1⌊e'⌉, xs') \leftrightarrow ((rev\ vs @ Addr\ a \#\ stk''') @ [v1], loc, length\ (compE2\ e1) + pc, None)$
by(*rule bisim1-bisims1.bisim1AAcc2*)
with *IH' stk* **show** *?case* **by**(*fastforce elim*!: *AAcc-τred1r-xt2*)
next
case (*bisim1AAss1 e n e' xs loc pc e2 e3*)
note $IH = ‹⟦\ pc < length\ (compE2\ e);\ compE2\ e\ !\ pc = Invoke\ M\ (length\ vs);\ n + max\text{-}vars\ e' \le length\ xs;\ bsok\ e\ n ⟧ \Longrightarrow ?concl\ e'\ xs\ e\ n\ pc\ stk'\ loc›$
note $bisim = ‹P,e,h \vdash (e', xs) \leftrightarrow (rev\ vs @ Addr\ a \#\ stk', loc, pc, None)›$
note $len = ‹n + max\text{-}vars\ (e'⌊e2⌉ := e3) \le length\ xs›$
hence *len'*: $n + max\text{-}vars\ e' \le length\ xs$ **by** *simp*
note $inv = ‹compE2\ (e⌊e2⌉ := e3)\ !\ pc = Invoke\ M\ (length\ vs)›$
with $‹pc < length\ (compE2\ (e⌊e2⌉ := e3))›\ bisim$ **have** *pc*: $pc < length\ (compE2\ e)$
by(*auto split*: *if-split-asm dest*: *bisim1-pc-length-compE2*)
moreover with *inv* **have** $compE2\ e\ !\ pc = Invoke\ M\ (length\ vs)$ **by** *simp*
moreover from $‹bsok\ (e⌊e2⌉ := e3)\ n›$ **have** $bsok\ e\ n$ **by** *simp*

ultimately have *?concl e′ xs e n pc stk′ loc* **using** *len′* **by**−(*rule IH*)
thus *?case*
by(*fastforce intro*: *bisim1-bisims1.bisim1AAss1 elim*!: *AAss-τred1r-xt1*)
next
case (*bisim1AAss2 e2 n e′ xs stk loc pc e1 e3 v1*)
note *IH* = ‹⋀*stk′*. ⟦*stk* = *rev vs* @ *Addr a* # *stk′*; *pc* < *length* (*compE2 e2*); *compE2 e2* ! *pc* = *Invoke M* (*length vs*); *n* + *max-vars e′* ≤ *length xs*; *bsok e2 n*
⟧ ⟹ *?concl e′ xs e2 n pc stk′ loc*›
note *inv* = ‹*compE2* (*e1*⌊*e2*⌉ := *e3*) ! (*length* (*compE2 e1*) + *pc*) = *Invoke M* (*length vs*)›
note *bisim* = ‹*P*,*e2*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)›
with *inv* ‹*length* (*compE2 e1*) + *pc* < *length* (*compE2* (*e1*⌊*e2*⌉ := *e3*))› **have** *pc*: *pc* < *length* (*compE2 e2*)
by(*auto split*: *if-split-asm dest*: *bisim1-pc-length-compE2*)
moreover with *inv* **have** *compE2 e2* ! *pc* = *Invoke M* (*length vs*) **by** *simp*
moreover with *bisim pc*
obtain *vs″ v″ stk″* **where** *stk* = *vs″* @ *v″* # *stk″* **and** *length vs″* = *length vs*
by(*auto dest*!: *bisim1-Invoke-stkD*)
with ‹*stk* @ [*v1*] = *rev vs* @ *Addr a* # *stk′*› **obtain** *stk‴*
where *stk‴*: *stk* = *rev vs* @ *Addr a* # *stk‴* **and** *stk*: *stk′* = *stk‴* @ [*v1*]
by(*cases stk′ rule*: *rev-cases*) *auto*
from ‹*n* + *max-vars* (*Val v1*⌊*e′*⌉ := *e3*) ≤ *length xs*› **have** *n* + *max-vars e′* ≤ *length xs* **by** *simp*
moreover from ‹*bsok* (*e1*⌊*e2*⌉ := *e3*) *n*› **have** *bsok e2 n* **by** *simp*
ultimately have *?concl e′ xs e2 n pc stk‴ loc* **using** *stk‴* **by**−(*rule IH*)
then obtain *e″ xs′* **where** *IH′*: *τred1r P t h* (*e′*, *xs*) (*e″*, *xs′*) *call1 e″* = ⌊(*a*, *M*, *vs*)⌋
and *bisim*: *P*,*e2*,*h* ⊢ (*e″*, *xs′*) ↔ (*rev vs* @ *Addr a* # *stk‴*, *loc*, *pc*, *None*) **by** *blast*
from *bisim*
have *P*,*e1*⌊*e2*⌉ := *e3*,*h* ⊢ (*Val v1*⌊*e″*⌉ := *e3*, *xs′*) ↔ ((*rev vs* @ *Addr a* # *stk‴*) @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *None*)
by(*rule bisim1-bisims1.bisim1AAss2*)
with *IH′ stk* **show** *?case* **by**(*fastforce elim*!: *AAss-τred1r-xt2*)
next
case (*bisim1AAss3 e3 n e′ xs stk loc pc e1 e2 v1 v2*)
note *IH* = ‹⋀*stk′*. ⟦*stk* = *rev vs* @ *Addr a* # *stk′*; *pc* < *length* (*compE2 e3*); *compE2 e3* ! *pc* = *Invoke M* (*length vs*); *n* + *max-vars e′* ≤ *length xs*; *bsok e3 n*
⟧ ⟹ *?concl e′ xs e3 n pc stk′ loc*›
note *inv* = ‹*compE2* (*e1*⌊*e2*⌉ := *e3*) ! (*length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*) = *Invoke M* (*length vs*)›
with ‹*length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc* < *length* (*compE2* (*e1*⌊*e2*⌉ := *e3*))›
have *pc*: *pc* < *length* (*compE2 e3*) **by**(*simp add*: *nth-Cons split*: *nat.split-asm if-split-asm*)
moreover with *inv* **have** *compE2 e3* ! *pc* = *Invoke M* (*length vs*) **by** *simp*
moreover with ‹*P*,*e3*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)› *pc*
obtain *vs″ v″ stk″* **where** *stk* = *vs″* @ *v″* # *stk″* **and** *length vs″* = *length vs*
by(*auto dest*!: *bisim1-Invoke-stkD*)
with ‹*stk* @ [*v2*, *v1*] = *rev vs* @ *Addr a* # *stk′*› **obtain** *stk‴*
where *stk‴*: *stk* = *rev vs* @ *Addr a* # *stk‴* **and** *stk*: *stk′* = *stk‴* @ [*v2*, *v1*]
by(*cases stk′ rule*: *rev-cases*) *auto*
from ‹*n* + *max-vars* (*Val v1*⌊*Val v2*⌉ := *e′*) ≤ *length xs*› **have** *n* + *max-vars e′* ≤ *length xs* **by** *simp*
moreover from ‹*bsok* (*e1*⌊*e2*⌉ := *e3*) *n*› **have** *bsok e3 n* **by** *simp*
ultimately have *?concl e′ xs e3 n pc stk‴ loc* **using** *stk‴* **by**−(*rule IH*)
then obtain *e″ xs′* **where** *IH′*: *τred1r P t h* (*e′*, *xs*) (*e″*, *xs′*) *call1 e″* = ⌊(*a*, *M*, *vs*)⌋
and *bisim*: *P*,*e3*,*h* ⊢ (*e″*, *xs′*) ↔ (*rev vs* @ *Addr a* # *stk‴*, *loc*, *pc*, *None*) **by** *blast*
from *bisim*
have *P*,*e1*⌊*e2*⌉ := *e3*,*h* ⊢ (*Val v1*⌊*Val v2*⌉ := *e″*, *xs′*) ↔ ((*rev vs* @ *Addr a* # *stk‴*) @ [*v2*, *v1*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*, *None*)

by −(*rule bisim1-bisims1.bisim1AAss3*, *auto*)
with *IH′ stk* **show** *?case* **by**(*fastforce elim*!: *AAss-τred1r-xt3*)
next
case *bisim1AAss4* **thus** *?case* **by** *simp*
next
case *bisim1ALength* **thus** *?case*
by(*fastforce split*: *if-split-asm intro*: *bisim1-bisims1.bisim1ALength dest*: *bisim1-pc-length-compE2 elim*!: *ALength-τred1r-xt*)
next
case *bisim1FAcc* **thus** *?case*
by(*fastforce split*: *if-split-asm intro*: *bisim1-bisims1.bisim1FAcc dest*: *bisim1-pc-length-compE2 elim*!: *FAcc-τred1r-xt*)
next
case *bisim1FAss1* **thus** *?case*
apply(*auto split*: *if-split-asm intro*: *bisim1-bisims1.bisim1FAss1 dest*: *bisim1-pc-length-compE2 elim*!: *FAss-τred1r-xt1*)
by(*fastforce intro*: *bisim1-bisims1.bisim1FAss1 elim*!: *FAss-τred1r-xt1*)
next
case (*bisim1FAss2 e2 n e′ xs stk loc pc e1 F D v1*)
note *IH* = ‹⋀*stk′*. ⟦*stk* = *rev vs* @ *Addr a* # *stk′*; *pc* < *length* (*compE2 e2*); *compE2 e2* ! *pc* = *Invoke M* (*length vs*); *n* + *max-vars e′* ≤ *length xs*; *bsok e2 n*
⟧ ⟹ *?concl e′ xs e2 n pc stk′ loc*›
note *inv* = ‹*compE2* (*e1*·*F*{*D*} := *e2*) ! (*length* (*compE2 e1*) + *pc*) = *Invoke M* (*length vs*)›
with ‹*length* (*compE2 e1*) + *pc* < *length* (*compE2* (*e1*·*F*{*D*} := *e2*))› **have** *pc*: *pc* < *length* (*compE2 e2*)
by(*simp split*: *if-split-asm nat.split-asm add*: *nth-Cons*)
moreover with *inv* **have** *compE2 e2* ! *pc* = *Invoke M* (*length vs*) **by** *simp*
moreover with ‹*P*,*e2*,*h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *None*)› *pc*
obtain *vs″ v″ stk″* **where** *stk* = *vs″* @ *v″* # *stk″* **and** *length vs″* = *length vs*
by(*auto dest*!: *bisim1-Invoke-stkD*)
with ‹*stk* @ [*v1*] = *rev vs* @ *Addr a* # *stk′*› **obtain** *stk‴*
where *stk‴*: *stk* = *rev vs* @ *Addr a* # *stk‴* **and** *stk*: *stk′* = *stk‴* @ [*v1*]
by(*cases stk′ rule*: *rev-cases*) *auto*
from ‹*n* + *max-vars* (*Val v1*·*F*{*D*} := *e′*) ≤ *length xs*› **have** *n* + *max-vars e′* ≤ *length xs* **by** *simp*
moreover from ‹*bsok* (*e1*·*F*{*D*} := *e2*) *n*› **have** *bsok e2 n* **by** *simp*
ultimately have *?concl e′ xs e2 n pc stk‴ loc* **using** *stk‴* **by**−(*rule IH*)
then obtain *e″ xs′* **where** *IH′*: *τred1r P t h* (*e′*, *xs*) (*e″*, *xs′*) *call1 e″* = ⌊(*a*, *M*, *vs*)⌋
and *bisim*: *P*,*e2*,*h* ⊢ (*e″*, *xs′*) ↔ (*rev vs* @ *Addr a* # *stk‴*, *loc*, *pc*, *None*) **by** *blast*
from *bisim* **have** *P*,*e1*·*F*{*D*} := *e2*,*h* ⊢ (*Val v1*·*F*{*D*} := *e″*, *xs′*) ↔ ((*rev vs* @ *Addr a* # *stk‴*) @ [*v1*], *loc*, *length* (*compE2 e1*) + *pc*, *None*)
by(*rule bisim1-bisims1.bisim1FAss2*)
with *IH′ stk* **show** *?case* **by**(*fastforce elim*!: *FAss-τred1r-xt2*)
next
case *bisim1FAss3* **thus** *?case* **by** *simp*
next
case (*bisim1CAS1 e n e′ xs loc pc e2 e3 D F*)
note *IH* = ‹⟦ *pc* < *length* (*compE2 e*); *compE2 e* ! *pc* = *Invoke M* (*length vs*); *n* + *max-vars e′* ≤ *length xs*; *bsok e n*
⟧ ⟹ *?concl e′ xs e n pc stk′ loc*›
note *bisim* = ‹*P*,*e*,*h* ⊢ (*e′*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk′*, *loc*, *pc*, *None*)›
note *len* = ‹*n* + *max-vars* - ≤ *length xs*›
hence *len′*: *n* + *max-vars e′* ≤ *length xs* **by** *simp*
note *inv* = ‹*compE2* - ! *pc* = *Invoke M* (*length vs*)›
with ‹*pc* < *length* -› *bisim* **have** *pc*: *pc* < *length* (*compE2 e*)

```
    by(auto split: if-split-asm dest: bisim1-pc-length-compE2)
  moreover with inv have compE2 e ! pc = Invoke M (length vs) by simp
  moreover from ‹bsok - n› have bsok e n by simp
  ultimately have ?concl e' xs e n pc stk' loc using len' by−(rule IH)
  thus ?case
    by(fastforce intro: bisim1-bisims1.bisim1CAS1 elim!: CAS-τred1r-xt1)
next
  case (bisim1CAS2 e2 n e' xs stk loc pc e1 e3 D F v1)
  note IH = ‹⋀stk'. ⟦stk = rev vs @ Addr a # stk'; pc < length (compE2 e2); compE2 e2 ! pc =
Invoke M (length vs); n + max-vars e' ≤ length xs; bsok e2 n
                ⟧ ⟹ ?concl e' xs e2 n pc stk' loc›
  note inv = ‹compE2 - ! (length (compE2 e1) + pc) = Invoke M (length vs)›
  note bisim = ‹P,e2,h ⊢ (e', xs) ↔ (stk, loc, pc, None)›
  with inv ‹length (compE2 e1) + pc < length (compE2 -)› have pc: pc < length (compE2 e2)
    by(auto split: if-split-asm dest: bisim1-pc-length-compE2)
  moreover with inv have compE2 e2 ! pc = Invoke M (length vs) by simp
  moreover with bisim pc
  obtain vs'' v'' stk'' where stk = vs'' @ v'' # stk'' and length vs'' = length vs
    by(auto dest!: bisim1-Invoke-stkD)
  with ‹stk @ [v1] = rev vs @ Addr a # stk'› obtain stk'''
    where stk''': stk = rev vs @ Addr a # stk''' and stk: stk' = stk''' @ [v1]
    by(cases stk' rule: rev-cases) auto
  from ‹n + max-vars - ≤ length xs› have n + max-vars e' ≤ length xs by simp
  moreover from ‹bsok - n› have bsok e2 n by simp
  ultimately have ?concl e' xs e2 n pc stk''' loc using stk''' by−(rule IH)
  then obtain e'' xs' where IH': τred1r P t h (e', xs) (e'', xs') call1 e'' = ⌊(a, M, vs)⌋
    and bisim: P,e2,h ⊢ (e'', xs') ↔ (rev vs @ Addr a # stk''', loc, pc, None) by blast
  from bisim
  have P,e1•compareAndSwap(D•F, e2, e3),h ⊢ (Val v1•compareAndSwap(D•F, e'', e3), xs') ↔ ((rev
vs @ Addr a # stk''') @ [v1], loc, length (compE2 e1) + pc, None)
    by(rule bisim1-bisims1.bisim1CAS2)
  with IH' stk show ?case by(fastforce elim!: CAS-τred1r-xt2)
next
  case (bisim1CAS3 e3 n e' xs stk loc pc e1 e2 D F v1 v2)
  note IH = ‹⋀stk'. ⟦stk = rev vs @ Addr a # stk'; pc < length (compE2 e3); compE2 e3 ! pc =
Invoke M (length vs); n + max-vars e' ≤ length xs; bsok e3 n
                ⟧ ⟹ ?concl e' xs e3 n pc stk' loc›
  note inv = ‹compE2 - ! (length (compE2 e1) + length (compE2 e2) + pc) = Invoke M (length vs)›
  with ‹length (compE2 e1) + length (compE2 e2) + pc < length (compE2 -)›
  have pc: pc < length (compE2 e3) by(simp add: nth-Cons split: nat.split-asm if-split-asm)
  moreover with inv have compE2 e3 ! pc = Invoke M (length vs) by simp
  moreover with ‹P,e3,h ⊢ (e', xs) ↔ (stk, loc, pc, None)›  pc
  obtain vs'' v'' stk'' where stk = vs'' @ v'' # stk'' and length vs'' = length vs
    by(auto dest!: bisim1-Invoke-stkD)
  with ‹stk @ [v2, v1] = rev vs @ Addr a # stk'› obtain stk'''
    where stk''': stk = rev vs @ Addr a # stk''' and stk: stk' = stk''' @ [v2, v1]
    by(cases stk' rule: rev-cases) auto
  from ‹n + max-vars - ≤ length xs› have n + max-vars e' ≤ length xs by simp
  moreover from ‹bsok - n› have bsok e3 n by simp
  ultimately have ?concl e' xs e3 n pc stk''' loc using stk''' by−(rule IH)
  then obtain e'' xs' where IH': τred1r P t h (e', xs) (e'', xs') call1 e'' = ⌊(a, M, vs)⌋
    and bisim: P,e3,h ⊢ (e'', xs') ↔ (rev vs @ Addr a # stk''', loc, pc, None) by blast
  from bisim
  have P,e1•compareAndSwap(D•F, e2, e3), h ⊢ (Val v1•compareAndSwap(D•F, Val v2, e''), xs') ↔
```

((*rev vs* @ *Addr a* # *stk'''*) @ [*v2*, *v1*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*, *None*)
by −(*rule bisim1-bisims1.bisim1CAS3*, *auto*)
with *IH' stk* **show** *?case* **by**(*fastforce elim*!: *CAS-τred1r-xt3*)
next
case (*bisim1Call1 obj n obj' xs loc pc ps M'*)
note *IH* = ‹⟦*pc* < *length* (*compE2 obj*); *compE2 obj* ! *pc* = *Invoke M* (*length vs*); *n* + *max-vars obj'* ≤ *length xs*; *bsok obj n* ⟧ ⟹ *?concl obj' xs obj n pc stk' loc*›
note *bisim* = ‹*P*,*obj*,*h* ⊢ (*obj'*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk'*, *loc*, *pc*, *None*)›
note *len* = ‹*n* + *max-vars* (*obj'*·*M'*(*ps*)) ≤ *length xs*›
hence *len'*: *n* + *max-vars obj'* ≤ *length xs* **by** *simp*
from ‹*bsok* (*obj*·*M'*(*ps*)) *n*› **have** *bsok*: *bsok obj n* **by** *simp*
note *inv* = ‹*compE2* (*obj*·*M'*(*ps*)) ! *pc* = *Invoke M* (*length vs*)›
with ‹*pc* < *length* (*compE2* (*obj*·*M'*(*ps*)))› *bisim*
have *pc*: *pc* < *length* (*compE2 obj*) ∨ *ps* = [] ∧ *pc* = *length* (*compE2 obj*)
by(*cases ps*)(*auto split*: *if-split-asm dest*: *bisim1-pc-length-compE2*)
thus *?case*
proof
assume *pc* < *length* (*compE2 obj*)
moreover with *inv* **have** *compE2 obj* ! *pc* = *Invoke M* (*length vs*) **by** *simp*
ultimately have *?concl obj' xs obj n pc stk' loc* **using** *len' bsok* **by**(*rule IH*)
thus *?thesis* **by**(*fastforce intro*: *bisim1-bisims1.bisim1Call1 elim*!: *Call-τred1r-obj*)
next
assume [*simp*]: *ps* = [] ∧ *pc* = *length* (*compE2 obj*)
with *inv* **have** [*simp*]: *vs* = [] *M'* = *M* **by** *simp-all*
with *bisim* **have** [*simp*]: *vs* = [] *stk'* = [] **by**(*auto dest*: *bisim1-pc-length-compE2D*)
with *bisim len' bsok* **have** *τred1r P t h* (*obj'*, *xs*) (*addr a*, *loc*)
by(*auto intro*: *bisim1-Val-τred1r simp add*: *bsok-def*)
moreover
have *P*,*obj*·*M*([]),*h* ⊢ (*addr a*·*M*([]), *loc*) ↔ ([*Addr a*], *loc*, *length* (*compE2 obj*), *None*)
by(*rule bisim1-bisims1.bisim1Call1*[*OF bisim1Val2*]) *simp-all*
ultimately show *?thesis* **by** *auto*(*fastforce elim*!: *Call-τred1r-obj*)
qed
next
case (*bisim1CallParams ps n ps' xs stk loc pc obj M' v*)
note *IH* = ‹⋀*stk'*. ⟦*stk* = *rev vs* @ *Addr a* # *stk'*; *pc* < *length* (*compEs2 ps*); *compEs2 ps* ! *pc* = *Invoke M* (*length vs*); *n* + *max-varss ps'* ≤ *length xs*; *bsoks ps n* ⟧ ⟹ *?concls ps' xs ps n pc stk' loc*›
note *bisim* = ‹*P*,*ps*,*h* ⊢ (*ps'*, *xs*) [↔] (*stk*, *loc*, *pc*, *None*)›
note *len* = ‹*n* + *max-vars* (*Val v*·*M'*(*ps'*)) ≤ *length xs*›
hence *len'*: *n* + *max-varss ps'* ≤ *length xs* **by** *simp*
note *stk* = ‹*stk* @ [*v*] = *rev vs* @ *Addr a* # *stk'*›
note *inv* = ‹*compE2* (*obj*·*M'*(*ps*)) ! (*length* (*compE2 obj*) + *pc*) = *Invoke M* (*length vs*)›
from ‹*bsok* (*obj*·*M'*(*ps*)) *n*› **have** *bsok*: *bsoks ps n* **by** *simp*
from ‹*length* (*compE2 obj*) + *pc* < *length* (*compE2* (*obj*·*M'*(*ps*)))›
have *pc* < *length* (*compEs2 ps*) ∨ *pc* = *length* (*compEs2 ps*) **by**(*auto*)
thus *?case*
proof
assume *pc*: *pc* < *length* (*compEs2 ps*)
moreover with *inv* **have** *compEs2 ps* ! *pc* = *Invoke M* (*length vs*) **by** *simp*
moreover with *bisim pc*
obtain *vs''* *v''* *stk''* **where** *stk* = *vs''* @ *v''* # *stk''* **and** *length vs''* = *length vs*
by(*auto dest*!: *bisims1-Invoke-stkD*)
with ‹*stk* @ [*v*] = *rev vs* @ *Addr a* # *stk'*› **obtain** *stk'''*

where *stk'''*: *stk* = *rev vs* @ *Addr a* # *stk'''* **and** *stk*: *stk'* = *stk'''* @ [*v*]
by(*cases stk' rule*: *rev-cases*) *auto*
note *len' stk'''*
ultimately have *?concls ps' xs ps n pc stk''' loc* **using** *bsok* **by**−(*rule IH*)
then obtain *es'' xs'* **where** *IH'*: *τreds1r P t h* (*ps'*, *xs*) (*es''*, *xs'*) *calls1 es''* = ⌊(*a*, *M*, *vs*)⌋
and *bisim*: *P*,*ps*,*h* ⊢ (*es''*, *xs'*) [↔] (*rev vs* @ *Addr a* # *stk'''*, *loc*, *pc*, *None*) **by** *blast*
from *bisim* **have** *P*,*obj*•*M'*(*ps*),*h* ⊢ (*Val v*•*M'*(*es''*), *xs'*) ↔ ((*rev vs* @ *Addr a* # *stk'''*) @ [*v*], *loc*, *length* (*compE2 obj*) + *pc*, *None*)
by(*rule bisim1-bisims1.bisim1CallParams*)
with *IH' stk* **show** *?case*
by(*fastforce elim*!: *Call-τred1r-param simp add*: *is-vals-conv*)
next
assume [*simp*]: *pc* = *length* (*compEs2 ps*)
from *bisim* **obtain** *vs'* **where** [*simp*]: *stk* = *rev vs'*
and *psvs'*: *length ps* = *length vs'* **by**(*auto dest*: *bisims1-pc-length-compEs2D*)
from *inv* **have** [*simp*]: *M'* = *M* **and** *vsps*: *length vs* = *length ps* **by** *simp-all*
with *stk psvs'* **have** [*simp*]: *v* = *Addr a stk'* = [] *vs'* = *vs* **by** *simp-all*
from *bisim len' bsok* **have** *τreds1r P t h* (*ps'*, *xs*) (*map Val vs*, *loc*)
by(*auto intro*: *bisims1-Val-τReds1r simp add*: *bsoks-def*)
moreover from *bisims1-map-Val-append*[*OF bisims1Nil vsps*[*symmetric*], *simplified*, *of P h loc*]
have *P*,*obj*•*M*(*ps*),*h* ⊢ (*addr a*•*M*(*map Val vs*), *loc*) ↔ (*rev vs* @ [*Addr a*], *loc*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), *None*)
by(*rule bisim1-bisims1.bisim1CallParams*)
ultimately show *?thesis* **by**(*fastforce elim*!: *Call-τred1r-param*)
qed
next
case *bisim1BlockSome1* **thus** *?case* **by** *simp*
next
case *bisim1BlockSome2* **thus** *?case* **by** *simp*
next
case (*bisim1BlockSome4 e n e' xs loc pc V T v*)
note *IH* = ‹⟦*pc* < *length* (*compE2 e*); *compE2 e* ! *pc* = *Invoke M* (*length vs*); *Suc n* + *max-vars e'* ≤ *length xs*; *bsok e* (*Suc n*)
⟧ ⟹ *?concl e' xs e* (*Suc V*) *pc stk' loc*›
from ‹*Suc* (*Suc pc*) < *length* (*compE2* {*V*:*T*=⌊*v*⌋; *e*})› **have** *pc* < *length* (*compE2 e*) **by** *simp*
moreover from ‹*compE2* {*V*:*T*=⌊*v*⌋; *e*} ! *Suc* (*Suc pc*) = *Invoke M* (*length vs*)›
have *compE2 e* ! *pc* = *Invoke M* (*length vs*) **by** *simp*
moreover note *len* = ‹*n* + *max-vars* {*V*:*T*=*None*; *e'*} ≤ *length xs*›
hence *Suc n* + *max-vars e'* ≤ *length xs* **by** *simp*
moreover from ‹*bsok* {*V*:*T*=⌊*v*⌋; *e*} *n*› **have** *bsok e* (*Suc n*) **by** *simp*
ultimately have *?concl e' xs e* (*Suc V*) *pc stk' loc* **by**(*rule IH*)
then obtain *e'' xs'* **where** *red*: *τred1r P t h* (*e'*, *xs*) (*e''*, *xs'*)
and *bisim'*: *P*,*e*,*h* ⊢ (*e''*, *xs'*) ↔ (*rev vs* @ *Addr a* # *stk'*, *loc*, *pc*, *None*)
and *call*: *call1 e''* = ⌊(*a*, *M*, *vs*)⌋ **by** *blast*
from *red* **have** *τred1r P t h* ({*V*:*T*=*None*; *e'*}, *xs*) ({*V*:*T*=*None*; *e''*}, *xs'*) **by**(*rule Block-None-τred1r-xt*)
with *bisim' call* **show** *?case* **by**(*fastforce intro*: *bisim1-bisims1.bisim1BlockSome4*)
next
case (*bisim1BlockNone e n e' xs loc pc V T*)
note *IH* = ‹⟦*pc* < *length* (*compE2 e*); *compE2 e* ! *pc* = *Invoke M* (*length vs*); *Suc n* + *max-vars e'* ≤ *length xs*; *bsok e* (*Suc n*)
⟧ ⟹ *?concl e' xs e* (*Suc V*) *pc stk' loc*›
from ‹*pc* < *length* (*compE2* {*V*:*T*=*None*; *e*})› **have** *pc* < *length* (*compE2 e*) **by** *simp*
moreover from ‹*compE2* {*V*:*T*=*None*; *e*} ! *pc* = *Invoke M* (*length vs*)›
have *compE2 e* ! *pc* = *Invoke M* (*length vs*) **by** *simp*

```
  moreover note len = ‹n + max-vars {V:T=None; e'} ≤ length xs›
  hence Suc n + max-vars e' ≤ length xs by simp
  moreover from ‹bsok {V:T=None; e} n› have bsok e (Suc n) by simp
  ultimately have ?concl e' xs e (Suc V) pc stk' loc by(rule IH)
  then obtain e'' xs' where red: τred1r P t h (e', xs) (e'', xs')
    and bisim': P,e,h ⊢ (e'', xs') ↔ (rev vs @ Addr a # stk', loc, pc, None)
    and call: call1 e'' = ⌊(a, M, vs)⌋ by blast
 from red have τred1r P t h ({V:T=None; e'}, xs) ({V:T=None; e''}, xs') by(rule Block-None-τred1r-xt)
  with bisim' call show ?case by(fastforce intro: bisim1-bisims1.bisim1BlockNone)
next
  case bisim1Sync1 thus ?case
    apply(auto split: if-split-asm intro: bisim1-bisims1.bisim1Sync1 dest: bisim1-pc-length-compE2
elim!: Sync-τred1r-xt)
    by(fastforce split: if-split-asm intro: bisim1-bisims1.bisim1Sync1 elim!: Sync-τred1r-xt)
next
  case bisim1Sync2 thus ?case by simp
next
  case bisim1Sync3 thus ?case by simp
next
  case bisim1Sync4 thus ?case
    apply(auto split: if-split-asm intro: bisim1-bisims1.bisim1Sync4 dest: bisim1-pc-length-compE2
elim!: InSync-τred1r-xt)
    by(fastforce split: if-split-asm intro: bisim1-bisims1.bisim1Sync4 elim!: InSync-τred1r-xt)
next
  case bisim1Sync5 thus ?case by simp
next
  case bisim1Sync6 thus ?case by simp
next
  case bisim1Sync7 thus ?case by simp
next
  case bisim1Sync8 thus ?case by simp
next
  case bisim1Sync9 thus ?case by simp
next
  case bisim1InSync thus ?case by simp
next
  case bisim1Seq1 thus ?case
   apply(auto split: if-split-asm intro: bisim1-bisims1.bisim1Seq1 dest: bisim1-pc-length-compE2 elim!:
Seq-τred1r-xt)
    by(fastforce split: if-split-asm intro: bisim1-bisims1.bisim1Seq1 elim!: Seq-τred1r-xt)
next
  case bisim1Seq2 thus ?case
    by(auto split: if-split-asm intro: bisim1-bisims1.bisim1Seq2 dest: bisim1-pc-length-compE2)
next
  case bisim1Cond1 thus ?case
    apply(clarsimp split: if-split-asm)
     apply(fastforce intro!: exI intro: bisim1-bisims1.bisim1Cond1 elim!: Cond-τred1r-xt)
    by(fastforce dest: bisim1-pc-length-compE2)
next
  case bisim1CondThen thus ?case
    apply(clarsimp split: if-split-asm)
     apply(fastforce intro!: exI intro: bisim1-bisims1.bisim1CondThen)
    by(fastforce dest: bisim1-pc-length-compE2)
next
```

case *bisim1CondElse* **thus** *?case*
by(*clarsimp split*: *if-split-asm*)(*fastforce intro*!: *exI intro*: *bisim1-bisims1.bisim1CondElse*)
next
case *bisim1While1* **thus** *?case* **by** *simp*
next
case *bisim1While3* **thus** *?case*
apply(*auto split*: *if-split-asm intro*: *bisim1-bisims1.bisim1While3 dest*: *bisim1-pc-length-compE2 elim*!: *Cond-τred1r-xt*)
by(*fastforce split*: *if-split-asm intro*: *bisim1-bisims1.bisim1While3 elim*!: *Cond-τred1r-xt*)
next
case *bisim1While4* **thus** *?case*
apply(*auto split*: *if-split-asm intro*: *bisim1-bisims1.bisim1While4 dest*: *bisim1-pc-length-compE2 elim*!: *Seq-τred1r-xt*)
by(*fastforce split*: *if-split-asm intro*: *bisim1-bisims1.bisim1While4 elim*!: *Seq-τred1r-xt*)
next
case *bisim1While6* **thus** *?case* **by** *simp*
next
case *bisim1While7* **thus** *?case* **by** *simp*
next
case *bisim1Throw1* **thus** *?case*
by(*fastforce split*: *if-split-asm intro*: *bisim1-bisims1.bisim1Throw1 dest*: *bisim1-pc-length-compE2 elim*!: *Throw-τred1r-xt*)
next
case *bisim1Try* **thus** *?case*
apply(*auto split*: *if-split-asm intro*: *bisim1-bisims1.bisim1Try dest*: *bisim1-pc-length-compE2 elim*!: *Try-τred1r-xt*)
by(*fastforce split*: *if-split-asm intro*: *bisim1-bisims1.bisim1Try elim*!: *Try-τred1r-xt*)
next
case *bisim1TryCatch1* **thus** *?case* **by** *simp*
next
case (*bisim1TryCatch2 e n e′ xs loc pc e1 C V*)
note *IH* = ‹⟦*pc* < *length* (*compE2 e*); *compE2 e* ! *pc* = *Invoke M* (*length vs*); *Suc n* + *max-vars e′* ≤ *length xs*; *bsok e* (*Suc n*)
⟧ ⟹ *?concl e′ xs e* (*Suc V*) *pc stk′ loc*›
from ‹*Suc* (*Suc* (*length* (*compE2 e1*) + *pc*)) < *length* (*compE2* (*try e1 catch*(*C V*) *e*))›
have *pc* < *length* (*compE2 e*) **by** *simp*
moreover from ‹*compE2* (*try e1 catch*(*C V*) *e*) ! *Suc* (*Suc* (*length* (*compE2 e1*) + *pc*)) = *Invoke M* (*length vs*)›
have *compE2 e* ! *pc* = *Invoke M* (*length vs*) **by** *simp*
moreover note *len* = ‹*n* + *max-vars* {*V*:*Class C*=*None*; *e′*} ≤ *length xs*›
hence *Suc n* + *max-vars e′* ≤ *length xs* **by** *simp*
moreover from ‹*bsok* (*try e1 catch*(*C V*) *e*) *n*› **have** *bsok e* (*Suc n*) **by** *simp*
ultimately have *?concl e′ xs e* (*Suc V*) *pc stk′ loc* **by**(*rule IH*)
then obtain *e″ xs′* **where** *red*: *τred1r P t h* (*e′*, *xs*) (*e″*, *xs′*)
and *bisim′*: *P*,*e*,*h* ⊢ (*e″*, *xs′*) ↔ (*rev vs* @ *Addr a* # *stk′*, *loc*, *pc*, *None*)
and *call*: *call1 e″* = ⌊(*a*, *M*, *vs*)⌋ **by** *blast*
from *red* **have** *τred1r P t h* ({*V*:*Class C*=*None*; *e′*}, *xs*) ({*V*:*Class C*=*None*; *e″*}, *xs′*)
by(*rule Block-None-τred1r-xt*)
with *bisim′ call* **show** *?case* **by**(*fastforce intro*: *bisim1-bisims1.bisim1TryCatch2*)
next
case *bisims1Nil* **thus** *?case* **by** *simp*
next
case (*bisims1List1 e n e′ xs loc pc es*)
note *IH* = ‹⟦*pc* < *length* (*compE2 e*); *compE2 e* ! *pc* = *Invoke M* (*length vs*); *n* + *max-vars e′* ≤

length xs; *bsok e n*
⟧ ⟹ *?concl e′ xs e n pc stk′ loc*›
note *bisim* = ‹*P*,*e*,*h* ⊢ (*e′*, *xs*) ↔ (*rev vs* @ *Addr a* # *stk′*, *loc*, *pc*, *None*)›
note *len* = ‹*n* + *max-varss* (*e′* # *es*) ≤ *length xs*›
hence *len′*: *n* + *max-vars e′* ≤ *length xs* **by** *simp*
from ‹*bsoks* (*e* # *es*) *n*› **have** *bsok*: *bsok e n* **by** *simp*
note *inv* = ‹*compEs2* (*e* # *es*) ! *pc* = *Invoke M* (*length vs*)›
with ‹*pc* < *length* (*compEs2* (*e* # *es*))› *bisim* **have** *pc*: *pc* < *length* (*compE2 e*)
by(*cases es*)(*auto split*: *if-split-asm dest*: *bisim1-pc-length-compE2*)
moreover with *inv* **have** *compE2 e* ! *pc* = *Invoke M* (*length vs*) **by** *simp*
ultimately have *?concl e′ xs e n pc stk′ loc* **using** *len′ bsok* **by**(*rule IH*)
thus *?case* **by**(*fastforce intro*: *bisim1-bisims1*.*bisims1List1 elim*!: *τred1r-inj-τreds1r*)
next
case (*bisims1List2 es n es′ xs stk loc pc e v*)
note *IH* = ‹⋀*stk′*. ⟦*stk* = *rev vs* @ *Addr a* # *stk′*; *pc* < *length* (*compEs2 es*); *compEs2 es* ! *pc* = *Invoke M* (*length vs*); *n* + *max-varss es′* ≤ *length xs*; *bsoks es n*
⟧ ⟹ *?concls es′ xs es n pc stk′ loc*›
note *bisim* = ‹*P*,*es*,*h* ⊢ (*es′*, *xs*) [↔] (*stk*, *loc*, *pc*, *None*)›
note *len* = ‹*n* + *max-varss* (*Val v* # *es′*) ≤ *length xs*›
hence *len′*: *n* + *max-varss es′* ≤ *length xs* **by** *simp*
from ‹*bsoks* (*e* # *es*) *n*› **have** *bsok*: *bsoks es n* **by** *simp*
note *stk* = ‹*stk* @ [*v*] = *rev vs* @ *Addr a* # *stk′*›
note *inv* = ‹*compEs2* (*e* # *es*) ! (*length* (*compE2 e*) + *pc*) = *Invoke M* (*length vs*)›
from ‹*length* (*compE2 e*) + *pc* < *length* (*compEs2* (*e* # *es*))› **have** *pc*: *pc* < *length* (*compEs2 es*)
by *auto*
moreover with *inv* **have** *compEs2 es* ! *pc* = *Invoke M* (*length vs*) **by** *simp*
moreover with *bisim pc*
obtain *vs″ v″ stk″* **where** *stk* = *vs″* @ *v″* # *stk″* **and** *length vs″* = *length vs*
by(*auto dest*!: *bisims1-Invoke-stkD*)
with ‹*stk* @ [*v*] = *rev vs* @ *Addr a* # *stk′*› **obtain** *stk‴*
where *stk‴*: *stk* = *rev vs* @ *Addr a* # *stk‴* **and** *stk*: *stk′* = *stk‴* @ [*v*]
by(*cases stk′ rule*: *rev-cases*) *auto*
note *len′ stk‴*
ultimately have *?concls es′ xs es n pc stk‴ loc* **using** *bsok* **by**−(*rule IH*)
then obtain *es″ xs′* **where** *red*: *τreds1r P t h* (*es′*, *xs*) (*es″*, *xs′*)
and *call*: *calls1 es″* = ⌊(*a*, *M*, *vs*)⌋
and *bisim*: *P*,*es*,*h* ⊢ (*es″*, *xs′*) [↔] (*rev vs* @ *Addr a* # *stk‴*, *loc*, *pc*, *None*) **by** *blast*
from *bisim* **have** *P*,*e*#*es*,*h* ⊢ (*Val v* # *es″*, *xs′*) [↔]
((*rev vs* @ *Addr a* # *stk‴*) @ [*v*], *loc*, *length* (*compE2 e*) + *pc*, *None*)
by(*rule bisim1-bisims1*.*bisims1List2*)
moreover from *red* **have** *τreds1r P t h* (*Val v* # *es′*, *xs*) (*Val v* # *es″*, *xs′*) **by**(*rule τreds1r-cons-τreds1r*)
ultimately show *?case* **using** *stk call* **by** *fastforce*
qed

end

declare *split-beta* [*simp del*]

context *J1-JVM-conf-read* **begin**

lemma *τRed1-simulates-exec-1-τ*:
assumes *wf*: *wf-J1-prog P*
and *exec*: *exec-1-d* (*compP2 P*) *t* (*Normal* (*xcp*, *h*, *frs*)) *ta* (*Normal* (*xcp′*, *h′*, *frs′*))
and *bisim*: *bisim1-list1 t h* (*e*, *xs*) *exs xcp frs*

and τ: $\tau Move2\ (compP2\ P)\ (xcp,\ h,\ frs)$
shows $h = h' \wedge (\exists\, e'\ xs'\ exs'.\ (if\ sim21\text{-}size\ (compP2\ P)\ (xcp',\ frs')\ (xcp,\ frs)\ then\ \tau Red1r\ else\ \tau Red1t)\ P\ t\ h\ ((e,\ xs),\ exs)\ ((e',\ xs'),\ exs') \wedge bisim1\text{-}list1\ t\ h\ (e',\ xs')\ exs'\ xcp'\ frs')$
using *bisim*
proof(*cases*)
case (*bl1-Normal stk loc C M pc FRS Ts T body D*)
hence [*simp*]: $frs = (stk,\ loc,\ C,\ M,\ pc)\ \#\ FRS$
and *conf*: $compTP\ P \vdash t: (xcp,\ h,\ (stk,\ loc,\ C,\ M,\ pc)\ \#\ FRS)\ \surd$
and *sees*: $P \vdash C\ sees\ M: Ts \rightarrow T = \lfloor body \rfloor\ in\ D$
and *bisim*: $P,blocks1\ 0\ (Class\ D\#Ts)\ body,h \vdash (e,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
and *lenxs*: $max\text{-}vars\ e \leq length\ xs$
and *bisims*: $list\text{-}all2\ (bisim1\text{-}fr\ P\ h)\ exs\ FRS$ **by** *auto*
from *sees-method-compP*[*OF sees*, **where** $f{=}\lambda C\ M\ Ts\ T.\ compMb2$]
have *sees'*: $compP2\ P \vdash C\ sees\ M: Ts \rightarrow T = \lfloor (max\text{-}stack\ body,\ max\text{-}vars\ body,\ compE2\ body\ @\ [Return],\ compxE2\ body\ 0\ 0) \rfloor\ in\ D$
by(*simp add*: *compP2-def compMb2-def*)
from *bisim* **have** *pc*: $pc \leq length\ (compE2\ body)$ **by**(*auto dest*: *bisim1-pc-length-compE2*)
from *conf* **have** *hconf*: $hconf\ h\ preallocated\ h$ **by**(*simp-all add*: *correct-state-def*)

from *sees wf* **have** *bsok*: $bsok\ (blocks1\ 0\ (Class\ D\ \#\ Ts)\ body)\ 0$
by(*auto dest*!: *sees-wf-mdecl simp add*: *bsok-def wf-mdecl-def WT1-expr-locks*)

from *exec* **obtain** *check*: $check\ (compP2\ P)\ (xcp,\ h,\ frs)$
and *exec*: $compP2\ P,t \vdash (xcp,\ h,\ frs)\ -ta{-}jvm{\rightarrow}\ (xcp',\ h',\ frs')$
by(*rule jvmd-NormalE*)(*auto simp add*: *exec-1-iff*)

from *wt-compTP-compP2*[*OF wf*] *exec conf*
have *conf'*: $compTP\ P \vdash t{:}(xcp',\ h',\ frs')\ \surd$ **by**(*auto intro*: *BV-correct-1*)

from *conf* **have** *tconf*: $P,h \vdash t\ \surd t$ **unfolding** *correct-state-def*
by(*simp add*: *compP2-def tconf-def*)

show *?thesis*
proof(*cases xcp*)
case [*simp*]: *None*
from *exec* **have** *execi*: $(ta,\ xcp',\ h',\ frs') \in exec\text{-}instr\ (instrs\text{-}of\ (compP2\ P)\ C\ M\ !\ pc)\ (compP2\ P)\ t\ h\ stk\ loc\ C\ M\ pc\ FRS$
by(*simp add*: *exec-1-iff*)
show *?thesis*
proof(*cases* $pc < length\ (compE2\ body)$)
case *True*
with *execi sees'* **have** *execi*: $(ta,\ xcp',\ h',\ frs') \in exec\text{-}instr\ (compE2\ body\ !\ pc)\ (compP2\ P)\ t\ h\ stk\ loc\ C\ M\ pc\ FRS$
by(*simp*)
from τ *sees' True* **have** τi: $\tau move2\ (compP2\ P)\ h\ stk\ body\ pc\ None$ **by**(*simp add*: $\tau move2\text{-}iff$)

show *?thesis*
proof(*cases* $length\ frs' = Suc\ (length\ FRS)$)
case *False*
with *execi sees True compE2-not-Return*[*of body*]
have $(\exists\, M\ n.\ compE2\ body\ !\ pc = Invoke\ M\ n)$
apply(*cases compE2 body* ! *pc*)
apply(*auto split*: *if-split-asm sum.split-asm simp add*: *split-beta compP2-def compMb2-def*)
apply(*metis in-set-conv-nth*)+

done
then obtain *MM n* **where** *ins*: *compE2 body* ! *pc* = *Invoke MM n* **by** *blast*
with *bisim1-Invoke-stkD*[*OF bisim*[*unfolded None*], *of MM n*] *True* **obtain** *vs′ v′ stk′*
where [*simp*]: *stk* = *vs′* @ *v′* # *stk′ n* = *length vs′* **by** *auto*
from *check sees True ins* **have** *is-Ref v′*
by(*auto split*: *if-split-asm simp add*: *split-beta compP2-def compMb2-def check-def*)
moreover from *execi sees True ins False sees′* **have** *v′* ≠ *Null* **by** *auto*
ultimately obtain *a′* **where** [*simp*]: *v′* = *Addr a′* **by**(*auto simp add*: *is-Ref-def*)
from *bisim* **have** *Bisim′*: *P*,*blocks1 0* (*Class D*#*Ts*) *body*,*h* ⊢ (*e*, *xs*) ↔ (*rev* (*rev vs′*) @ *Addr a′* # *stk′*, *loc*, *pc*, *None*)
by *simp*
from *bisim1-Invoke-τRed*[*OF this - - - bsok*, *of MM t*] *True ins lenxs*
obtain *e′ xs′* **where** *red*: *τred1r P t h* (*e*, *xs*) (*e′*, *xs′*)
and *bisim′*: *P*,*blocks1 0* (*Class D*#*Ts*) *body*,*h* ⊢ (*e′*, *xs′*) ↔ (*rev* (*rev vs′*) @ *Addr a′* # *stk′*, *loc*, *pc*, *None*)
and *call′*: *call1 e′* = ⌊(*a′*, *MM*, *rev vs′*)⌋ **by** *auto*
from *red* **have** *Red*: *τRed1r P t h* ((*e*, *xs*), *exs*) ((*e′*, *xs′*), *exs*)
by(*rule τred1r-into-τRed1r*)

from *False execi True check ins sees′* **obtain** *U′ Ts′ T′ meth D′*
where *ha′*: *typeof-addr h a′* = ⌊*U′*⌋
and *Sees′*: *compP2 P* ⊢ *class-type-of U′ sees MM*:*Ts′* → *T′* = ⌊*meth*⌋ *in D′*
by(*auto simp add*: *check-def has-method-def split*: *if-split-asm*)(*auto split*: *extCallRet.split-asm*)
from *sees-method-compPD*[*OF Sees′*[*unfolded compP2-def*]] **obtain** *body′*
where *Sees*: *P* ⊢ *class-type-of U′ sees MM*:*Ts′* → *T′*=⌊*body′*⌋ *in D′*
and [*simp*]: *meth* = (*max-stack body′*, *max-vars body′*, *compE2 body′* @ [*Return*], *compxE2 body′ 0 0*)
by(*auto simp add*: *compMb2-def*)

let *?e* = *blocks1 0* (*Class D′*#*Ts′*) *body′*
let *?xs* = *Addr a′* # *rev vs′* @ *replicate* (*max-vars body′*) *undefined-value*
let *?e′xs′* = (*e′*, *xs′*)
let *?f* = (*stk*, *loc*, *C*, *M*, *pc*)
let *?f′* = ([],*Addr a′* # *rev vs′* @ *replicate* (*max-vars body′*) *undefined-value*, *D′*, *MM*, *0*)

from *execi pc ins False ha′ Sees′ sees′*
have [*simp*]: *xcp′* = *None ta* = *ε frs′* = *?f′* # *?f* # *FRS h′* = *h*
by(*auto split*: *if-split-asm simp add*: *split-beta*)

from *bisim′* **have** *bisim″*: *P*,*blocks1 0* (*Class D*#*Ts*) *body*,*h* ⊢ (*e′*, *xs′*) ↔ (*rev* (*rev vs′*) @ *Addr a′* # *stk′*, *loc*, *pc*, *None*)
by *simp*
have *n* = *length vs′* **by** *simp*
from *conf′ Sees′ ins sees′ True* **have** *n* = *length Ts′*
apply(*auto simp add*: *correct-state-def*)
apply(*drule* (*1*) *sees-method-fun*)+
apply(*auto dest*: *sees-method-idemp sees-method-fun*)
done
with ‹*n* = *length vs′*› **have** *vs′Ts′*: *length* (*rev vs′*) = *length Ts′* **by** *simp*

with *call′ ha′ Sees*
have *True*,*P*,*t* ⊢*1* ⟨(*e′*, *xs′*)/*exs*,*h*⟩ −*ε*→ ⟨(*?e*, *?xs*)/ (*e′*, *xs′*) # *exs*, *h*⟩ **by**(*rule red1Call*)
hence *True*,*P*,*t* ⊢*1* ⟨(*e′*, *xs′*)/*exs*,*h*⟩ −*ε*→ ⟨(*?e*, *?xs*)/ *?e′xs′* # *exs*, *h*⟩ **by**(*simp*)
moreover from *call′ Sees ha′* **have** *τMove1 P h* ((*e′*, *xs′*), *exs*)

by(*auto simp add*: *synthesized-call-def dest*!: *τmove1-not-call1*[**where** *P*=*P* **and** *h*=*h*] *dest*: *sees-method-fun*)
ultimately have *τRed1t P t h* ((*e'*, *xs'*), *exs*) ((*?e*, *?xs*), *?e'xs'* # *exs*) **by** *auto*
moreover have *bisim1-list1 t h* (*?e*, *?xs*) (*?e'xs'* # *exs*) *None* (*?f'* # *?f* # *FRS*)
proof
from *conf'* **show** *compTP P* ⊢ *t*:(*None*, *h*, *?f'* # *?f* # *FRS*) √ **by** *simp*
from *Sees* **show** *P* ⊢ *D' sees MM*: *Ts'*→*T'* = ⌊*body'*⌋ *in D'* **by**(*rule sees-method-idemp*)
show *P*,*blocks1 0* (*Class D'*#*Ts'*) *body'*,*h* ⊢ (*blocks1 0* (*Class D'*#*Ts'*) *body'*, *?xs*) ↔ ([], *Addr a'* # *rev vs'* @ *replicate* (*max-vars body'*) *undefined-value*, *0*, *None*)
by(*rule bisim1-refl*)
show *max-vars* (*blocks1 0* (*Class D'*#*Ts'*) *body'*) ≤ *length ?xs* **using** *vs'Ts'* **by**(*simp add*: *blocks1-max-vars*)
from *sees* **have** *bisim1-fr P h ?e'xs' ?f*
proof
show *P*,*blocks1 0* (*Class D*#*Ts*) *body*,*h* ⊢ (*e'*, *xs'*) ↔ (*stk*, *loc*, *pc*, *None*)
using *bisim''* **by** *simp*
from *call'* **show** *call1 e'* = ⌊(*a'*, *MM*, *rev vs'*)⌋ **.**
from *red* **have** *xs'xs*: *length xs'* = *length xs* **by**(*rule τred1r-preserves-len*)
with *red lenxs* **show** *max-vars e'* ≤ *length xs'* **by**(*auto dest*: *τred1r-max-vars*)
qed
with *bisims* **show** *list-all2* (*bisim1-fr P h*) (*?e'xs'* # *exs*) (*?f* # *FRS*) **by** *simp*
qed
ultimately show *?thesis* **using** *Red*
by *auto*(*blast intro*: *rtranclp-trans rtranclp-tranclp-tranclp tranclp-into-rtranclp*)+
next
case *True*
note *pc* = ‹*pc* < *length* (*compE2 body*)›
with *execi True* **have** ∃ *stk' loc' pc'*. *frs'* = (*stk'*, *loc'*, *C*, *M*, *pc'*) # *FRS*
by(*cases* (*compE2 body* @ [*Return*]) ! *pc*)(*auto split*: *if-split-asm sum.split-asm simp*: *split-beta*, *auto split*: *extCallRet.splits*)
then obtain *stk' loc' pc'* **where** [*simp*]: *frs'* = (*stk'*, *loc'*, *C*, *M*, *pc'*) # *FRS* **by** *blast*
from *conf* **obtain** *ST* **where** *compP2 P*,*h* ⊢ *stk* [:≤] *ST* **by**(*auto simp add*: *correct-state-def conf-f-def2*)
hence *ST*: *P*,*h* ⊢ *stk* [:≤] *ST* **by**(*rule List.list-all2-mono*)(*simp add*: *compP2-def*)
from *execi sees pc check*
have *exec'*: *exec-move-d P t* (*blocks1 0* (*Class D*#*Ts*) *body*) *h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, *pc'*, *xcp'*)
apply(*auto simp add*: *compP2-def compMb2-def exec-move-def check-def exec-meth-instr split*: *if-split-asm sum.split-asm*)
apply(*cases compE2 body* ! *pc*)
apply(*auto simp add*: *neq-Nil-conv split-beta split*: *if-split-asm sum.split-asm*)
apply(*force split*: *extCallRet.split-asm*)
apply(*cases compE2 body* ! *pc*, *auto simp add*: *split-beta neq-Nil-conv split*: *if-split-asm sum.split-asm*)
done
from *red1-simulates-exec-instr*[*OF wf hconf tconf bisim this - bsok ST*] *lenxs τi* **obtain** *e'' xs''*
where *bisim'*: *P*,*blocks1 0* (*Class D*#*Ts*) *body*,*h'* ⊢ (*e''*, *xs''*) ↔ (*stk'*, *loc'*, *pc'*, *xcp'*)
and *red*: (*if xcp'* = *None* ⟶ *pc* < *pc' then τred1r else τred1t*) *P t h* (*e*, *xs*) (*e''*, *xs''*) **and** [*simp*]: *h'* = *h*
by(*auto simp del*: *blocks1.simps*)
have *Red*: (*if sim21-size* (*compP2 P*) (*xcp'*, *frs'*) (*xcp*, *frs*) *then τRed1r else τRed1t*) *P t h* ((*e*, *xs*), *exs*) ((*e''*, *xs''*), *exs*)
proof(*cases xcp'* = *None* ⟶ *pc* < *pc'*)
case *True*

```
        from bisim bisim' have pc ≤ Suc (length (compE2 body)) pc' ≤ Suc (length (compE2 body))
          by(auto dest: bisim1-pc-length-compE2)
        moreover {
          fix a assume xcp' = ⌊a⌋
          with exec' have pc = pc' by(auto dest: exec-move-raise-xcp-pcD) }
        ultimately have sim21-size (compP2 P) (xcp', frs') (xcp, frs) using sees True
              by(auto simp add: sim21-size-def)(auto simp add: compP2-def compMb2-def intro!:
sim21-size-aux.intros)
        with red True show ?thesis by simp(rule τred1r-into-τRed1r)
      next
        case False
        thus ?thesis using red by(auto intro: τred1t-into-τRed1t τred1r-into-τRed1r)
      qed
      moreover from red lenxs
      have max-vars e'' ≤ length xs''
        apply(auto dest: τred1r-max-vars τred1r-preserves-len τred1t-max-vars τred1t-preserves-len
split: if-split-asm)
       apply(frule τred1r-max-vars τred1t-max-vars, drule τred1r-preserves-len τred1t-preserves-len,
simp)+
        done
      with conf' sees bisim'
      have bisim1-list1 t h (e'', xs'') exs xcp' ((stk', loc', C, M, pc') # FRS)
        unfolding ‹frs' = (stk', loc', C, M, pc') # FRS› ‹h' = h›
        using bisims by(rule bisim1-list1.bl1-Normal)
      ultimately show ?thesis by(auto split del: if-split)
    qed
  next
    case False
    with pc have [simp]: pc = length (compE2 body) by simp
    with execi sees have [simp]: xcp' = None
    by(cases compE2 body ! pc)(auto split: if-split-asm simp add: compP2-def compMb2-def split-beta)
    from bisim have Bisim: P,blocks1 0 (Class D#Ts) body,h ⊢ (e, xs) ↔ (stk, loc, length (compE2
(blocks1 0 (Class D#Ts) body)), None) by simp
    then obtain v where [simp]: stk = [v] by(blast dest: bisim1-pc-length-compE2D)
    with Bisim lenxs bsok have red: τred1r P t h (e, xs) (Val v, loc)
      by clarify (erule bisim1-Val-τred1r[where n=0], simp-all add: bsok-def)
    hence Red: τRed1r P t h ((e, xs), exs) ((Val v, loc), exs) by(rule τred1r-into-τRed1r)
    show ?thesis
    proof(cases FRS)
      case [simp]: Nil
      with bisims have [simp]: exs = [] by simp
      with exec sees' have [simp]: ta = ε xcp' = None h' = h frs' = []
        by(auto simp add: exec-1-iff)
      from hconf have bisim1-list1 t h (Val v, loc) [] None [] by(rule bl1-finalVal)
      then show ?thesis using Red
        by(auto intro: rtranclp.rtrancl-into-rtrancl rtranclp-into-tranclp1 simp del: τRed1-conv simp
add: sim21-size-def)
    next
      case (Cons f' FRS')
      then obtain stk'' loc'' C'' M'' pc''
        where [simp]: FRS = (stk'', loc'', C'', M'', pc'') # FRS' by(cases f') fastforce
      from bisims obtain e'' xs'' EXS' where [simp]: exs = (e'', xs'') # EXS'
        by(auto simp add: list-all2-Cons2)
      with bisims have bisim1-fr P h (e'', xs'') (stk'', loc'', C'', M'', pc'') by simp
```

```
      then obtain E'' Ts'' T'' body'' D'' a'' M''' vs''
        where [simp]: e'' = E''
        and sees'': P ⊢ C'' sees M'':Ts''→T'' = ⌊body''⌋ in D''
        and bisim'': P,blocks1 0 (Class D''#Ts'') body'',h ⊢ (E'', xs'') ↔ (stk'', loc'', pc'', None)
        and call'': call1 E'' = ⌊(a'', M''', vs'')⌋
        and lenxs'': max-vars E'' ≤ length xs''
        by(cases) fastforce
      let ?ee' = inline-call (Val v) E''
      let ?e' = ?ee'
      let ?xs' = xs''

      from bisim'' call'' have pc'': pc'' < length (compE2 (blocks1 0 (Class D''#Ts'') body''))
        by(rule bisim1-call-pcD)
      hence pc'': pc'' < length (compE2 body'') by simp
      with sees-method-compP[OF sees'', where f=λC M Ts T. compMb2]
        sees-method-compP[OF sees, where f=λC M Ts T. compMb2] conf
      obtain ST LT where Φ: compTP P C'' M'' ! pc'' = ⌊(ST, LT)⌋
       and conff: conf-f (compP (λC M Ts T. compMb2) P) h (ST, LT) (compE2 body'' @ [Return])
(stk'', loc'', C'', M'', pc'')
        and ins: (compE2 body'' @ [Return]) ! pc'' = Invoke M (length Ts)
      unfolding correct-state-def by(fastforce simp add: compP2-def compMb2-def dest: sees-method-fun)
      from bisim1-callD[OF bisim'' call'', of M length Ts] ins pc''
      have [simp]: M''' = M by simp

      from call'' have call1 E'' = ⌊(a'', M''', vs'')⌋ by simp
      have True,P,t ⊢1 ⟨(Val v, loc)/(E'', xs'') # EXS',h⟩ −ε→
                       ⟨(inline-call (Val v) E'', xs'')/EXS', h⟩
        by(rule red1Return) simp
      hence True,P,t ⊢1 ⟨(Val v, loc)/(E'', xs'') # EXS',h⟩ −ε→ ⟨(?e', ?xs')/EXS', h⟩
        by simp
      moreover have τMove1 P h ((Val v, loc), (E'', xs'') # EXS') by auto
      ultimately have τRed1 P t h ((Val v, loc), (E'', xs'') # EXS') ((?e', ?xs'), EXS') by auto
      moreover from exec sees have [simp]: ta = ε h' = h
        and [simp]: frs' = (v # drop (length Ts + 1) stk'', loc'', C'', M'', pc'' + 1) # FRS'
        by(auto simp add: compP2-def compMb2-def exec-1-iff)

      have bisim1-list1 t h (?e', ?xs') EXS' None ((v # drop (length Ts + 1) stk'', loc'', C'', M'',
pc'' + 1) # FRS')
      proof
        from conf' show compTP P ⊢ t: (None, h, (v # drop (length Ts + 1) stk'', loc'', C'', M'',
pc'' + 1) # FRS') √ by simp
        from sees'' show P ⊢ C'' sees M'': Ts''→T'' = ⌊body''⌋ in D'' .
        from bisim1-inline-call-Val[OF bisim'' call'', of length Ts v] ins pc''
          show P,blocks1 0 (Class D''#Ts'') body'',h ⊢ (inline-call (Val v) E'', xs'') ↔ (v # drop
(length Ts + 1) stk'', loc'', pc'' + 1, None) by simp
        from lenxs'' max-vars-inline-call[of Val v E'']
        show max-vars (inline-call (Val v) E'') ≤ length xs'' by simp
        from bisims show list-all2 (bisim1-fr P h) EXS' FRS' by simp
      qed
      ultimately show ?thesis using Red
        by(auto simp add: τRed1-conv intro: rtranclp-into-tranclp1 rtranclp.rtrancl-into-rtrancl)
    qed
  qed
next
```

case [*simp*]: (*Some a′*)
from *exec* **have** *execs*: (*xcp′*, *h′*, *frs′*) = *exception-step* (*compP2 P*) *a′ h* (*stk*, *loc*, *C*, *M*, *pc*) *FRS*
and [*simp*]: *ta* = *ε* **by**(*auto simp add*: *exec-1-iff*)
from *conf* **have** *confxcp′*: *conf-xcp′ P h xcp*
unfolding *correct-state-def* **by**(*auto simp add*: *compP2-def*)
then obtain *D′* **where** *ha′*: *typeof-addr h a′* = ⌊*Class-type D′*⌋ **and** *subclsD′*: *P* ⊢ *D′* ⪯* *Throwable*
by *auto*
from *bisim* **have** *pc*: *pc* < *length* (*compE2 body*) **by**(*auto dest*: *bisim1-xcp-pcD*)
show *?thesis*
proof(*cases match-ex-table* (*compP2 P*) (*cname-of h a′*) *pc* (*ex-table-of* (*compP2 P*) *C M*))
case *None*
from *bisim* **have** *pc*: *pc* < *length* (*compE2 body*) **by**(*auto dest*: *bisim1-xcp-pcD*)
with *sees′ None* **have** *match*: *match-ex-table* (*compP2 P*) (*cname-of h a′*) *pc* (*compxE2 body 0 0*) = *None*
by(*auto*)
with *execs sees′* **have** [*simp*]: *ta* = *ε xcp′* = ⌊*a′*⌋ *h′* = *h frs′* = *FRS* **using** *match sees′* **by** *auto*
from *conf* **obtain** *CCC* **where** *ha*: *typeof-addr h a′* = ⌊*Class-type CCC*⌋ **and** *subcls*: *P* ⊢ *CCC* ⪯* *Throwable*
unfolding *correct-state-def* **by**(*auto simp add*: *conf-f-def2 compP2-def*)
from *bisim1-xcp-τRed*[*OF ha subcls bisim*[*unfolded Some*], *of* λ*C M Ts T*. *compMb2*] *match lenxs bsok*
have *red*: *τred1r P t h* (*e*, *xs*) (*Throw a′*, *loc*)
and *b′*: *P*,*blocks1 0* (*Class D*#*Ts*) *body*,*h* ⊢ (*Throw a′*, *loc*) ↔ (*stk*, *loc*, *pc*, ⌊*a′*⌋)
by(*auto simp add*: *compP2-def bsok-def*)
from *red* **have** *Red*: *τRed1r P t h* ((*e*, *xs*), *exs*) ((*Throw a′*, *loc*), *exs*)
by(*rule τred1r-into-τRed1r*)
show *?thesis*
proof(*cases FRS*)
case (*Cons f′ FRS′*)
then obtain *stk″ loc″ C″ M″ pc″*
where [*simp*]: *FRS* = (*stk″*, *loc″*, *C″*, *M″*, *pc″*) # *FRS′* **by**(*cases f′*) *fastforce*
from *bisims* **obtain** *e″ xs″ EXS′* **where** [*simp*]: *exs* = (*e″*, *xs″*) # *EXS′*
by(*auto simp add*: *list-all2-Cons2*)
with *bisims* **have** *bisim1-fr P h* (*e″*, *xs″*) (*stk″*, *loc″*, *C″*, *M″*, *pc″*) **by** *simp*
then obtain *E″ Ts″ T″ body″ D″ a″ M‴ vs″*
where [*simp*]: *e″* = *E″*
and *sees″*: *P* ⊢ *C″ sees M″*:*Ts″*→*T″* = ⌊*body″*⌋ *in D″*
and *bisim″*: *P*,*blocks1 0* (*Class D″*#*Ts″*) *body″*,*h* ⊢ (*E″*, *xs″*) ↔ (*stk″*, *loc″*, *pc″*, *None*)
and *call″*: *call1 E″* = ⌊(*a″*, *M‴*, *vs″*)⌋
and *lenxs″*: *max-vars E″* ≤ *length xs″*
by(*cases*) *fastforce*
let *?ee′* = *inline-call* (*Throw a′*) *E″*
let *?e′* = *?ee′*
let *?xs′* = *xs″*

from *bisim″ call″* **have** *pc″*: *pc″* < *length* (*compE2* (*blocks1 0* (*Class D″*#*Ts″*) *body″*))
by(*rule bisim1-call-pcD*)
hence *pc″*: *pc″* < *length* (*compE2 body″*) **by** *simp*
with *sees-method-compP*[*OF sees″*, **where** *f*=λ*C M Ts T*. *compMb2*]
sees-method-compP[*OF sees*, **where** *f*=λ*C M Ts T*. *compMb2*] *conf*
obtain *ST LT* **where** Φ: *compTP P C″ M″* ! *pc″* = ⌊(*ST*, *LT*)⌋
and *conff*: *conf-f* (*compP* (λ*C M Ts T*. *compMb2*) *P*) *h* (*ST*, *LT*) (*compE2 body″* @ [*Return*]) (*stk″*, *loc″*, *C″*, *M″*, *pc″*)
and *ins*: (*compE2 body″* @ [*Return*]) ! *pc″* = *Invoke M* (*length Ts*)

unfolding *correct-state-def*
by(*fastforce simp add*: *compP2-def compMb2-def dest*: *sees-method-fun*)
from *bisim1-callD*[*OF bisim'' call''*, *of M length Ts*] *ins pc''*
have [*simp*]: $M''' = M$ **by** *simp*

have *True*,*P*,*t* ⊢*1* ⟨(*Throw a'*, *loc*)/(*E''*, *xs''*) # *EXS'*,*h*⟩ −*ε*→ ⟨(*inline-call* (*Throw a'*) *E''*, *xs''*)/*EXS'*, *h*⟩
by(*rule red1Return*) *simp*
moreover have *τMove1 P h* ((*Throw a'*, *loc*), (*E''*, *xs''*) # *EXS'*) **by** *fastforce*
ultimately have *τRed1 P t h* ((*Throw a'*, *loc*), (*E''*, *xs''*) # *EXS'*) ((*?e'*, *?xs'*), *EXS'*) **by** *simp*
moreover
have *bisim1-list1 t h* (*?e'*, *?xs'*) *EXS'* ⌊*a'*⌋ ((*stk''*, *loc''*, *C''*, *M''*, *pc''*) # *FRS'*)
proof
from *conf'* **show** *compTP P* ⊢ *t*: (⌊*a'*⌋, *h*, (*stk''*, *loc''*, *C''*, *M''*, *pc''*) # *FRS'*) √ **by** *simp*
from *sees''* **show** *P* ⊢ *C''* *sees M''*: *Ts''*→*T''* = ⌊*body''*⌋ *in D''* **.**
from *bisim1-inline-call-Throw*[*OF bisim'' call''*, *of length Ts a'*] *ins pc''*
show *P*,*blocks1 0* (*Class D''*#*Ts''*) *body''*,*h* ⊢ (*inline-call* (*Throw a'*) *E''*, *xs''*) ↔ (*stk''*, *loc''*, *pc''*, ⌊*a'*⌋)
by *simp*
from *lenxs''* *max-vars-inline-call*[*of Throw a' E''*]
show *max-vars* (*inline-call* (*Throw a'*) *E''*) ≤ *length xs''* **by** *simp*
from *bisims* **show** *list-all2* (*bisim1-fr P h*) *EXS' FRS'* **by** *simp*
qed
ultimately show *?thesis* **using** *Red*
by(*auto simp del*: *τRed1-conv intro*: *rtranclp.rtrancl-into-rtrancl rtranclp-into-tranclp1*)
next
case [*simp*]: *Nil*
with *bisims* **have** [*simp*]: *exs* = [] **by** *simp*
from *hconf* **have** *bisim1-list1 t h* (*Throw a'*, *loc*) [] ⌊*a'*⌋ [] **by**(*rule bl1-finalThrow*)
thus *?thesis* **using** *Red*
by(*auto simp del*: *τRed1-conv intro*: *rtranclp.rtrancl-into-rtrancl rtranclp-into-tranclp1 simp add*: *sim21-size-def*)
qed
next
case (*Some pcd*)
then obtain *pch d* **where** *match*: *match-ex-table* (*compP2 P*) (*cname-of h a'*) *pc* (*ex-table-of* (*compP2 P*) *C M*) = ⌊(*pch*, *d*)⌋
by(*cases pcd*) *auto*
with *τ sees' pc* **have** *τ'*: *τmove2* (*compP2 P*) *h stk body pc* ⌊*a'*⌋ **by**(*simp add*: *compP2-def compMb2-def τmove2-iff*)
from *match execs* **have** [*simp*]: *h'* = *h xcp'* = *None*
frs' = (*Addr a'* # *drop* (*length stk* − *d*) *stk*, *loc*, *C*, *M*, *pch*) # *FRS* **by** *simp-all*
from *bisim match sees'*
have *d* ≤ *length stk* **by**(*auto intro*: *bisim1-match-Some-stk-length simp add*: *compP2-def compMb2-def*)
with *match sees'*
have *execm*: *exec-move-d P t* (*blocks1 0* (*Class D*#*Ts*) *body*) *h* (*stk*, *loc*, *pc*, ⌊*a'*⌋) *ta h'* (*Addr a'* # *drop* (*length stk* − *d*) *stk*, *loc*, *pch*, *None*)
by(*auto simp add*: *exec-move-def exec-meth-xcpt*)
from *conf* **obtain** *ST* **where** *compP2 P*,*h* ⊢ *stk* [:≤] *ST* **by**(*auto simp add*: *correct-state-def conf-f-def2*)
hence *ST*: *P*,*h* ⊢ *stk* [:≤] *ST* **by**(*rule List.list-all2-mono*)(*simp add*: *compP2-def*)
from *red1-simulates-exec-instr*[*OF wf hconf tconf bisim*[*unfolded* ‹*xcp* = ⌊*a'*⌋›] *execm* - *bsok ST*] *lenxs ha' subclsD' τ'*
obtain *e''* *xs''*

where b': P,*blocks1 0* (*Class D*#*Ts*) *body*,$h \vdash (e'', xs'') \leftrightarrow$ (*Addr a'* # *drop* (*length stk* − *d*) *stk*, *loc*, *pch*, *None*)
and *red*: (*if pc* < *pch then* τ*red1r else* τ*red1t*) *P t h* (*e*, *xs*) (*e''*, *xs''*) **and** [*simp*]: $h' = h$
by(*auto split*: *if-split-asm intro*: τ*move2xcp simp add*: *compP2-def simp del*: *blocks1.simps*)
have *Red*: (*if sim21-size* (*compP2 P*) (*xcp'*, *frs'*) (*xcp*, *frs*) *then* τ*Red1r else* τ*Red1t*) *P t h* ((*e*, *xs*), *exs*) ((*e''*, *xs''*), *exs*)
proof(*cases pc* < *pch*)
case *True*
from *bisim b'* **have** *pc* ≤ *Suc* (*length* (*compE2 body*)) *pch* ≤ *Suc* (*length* (*compE2 body*))
by(*auto dest*: *bisim1-pc-length-compE2*)
with *sees True* **have** *sim21-size* (*compP2 P*) (*xcp'*, *frs'*) (*xcp*, *frs*)
by(*auto simp add*: *sim21-size-def*)(*auto simp add*: *compP2-def compMb2-def intro*: *sim21-size-aux.intros*)
with *red True* **show** *?thesis* **by** *simp*(*rule* τ*red1r-into-*τ*Red1r*)
next
case *False*
thus *?thesis* **using** *red* **by**(*auto intro*: τ*red1t-into-*τ*Red1t* τ*red1r-into-*τ*Red1r*)
qed
moreover from *red lenxs*
have *max-vars e''* ≤ *length xs''*
apply(*auto dest*: τ*red1r-max-vars* τ*red1r-preserves-len* τ*red1t-max-vars* τ*red1t-preserves-len split*: *if-split-asm*)
apply(*frule* τ*red1r-max-vars* τ*red1t-max-vars*, *drule* τ*red1r-preserves-len* τ*red1t-preserves-len*, *simp*)+
done
with *conf' sees b'*
have *bisim1-list1 t h* (*e''*, *xs''*) *exs None* ((*Addr a'* # *drop* (*length stk* − *d*) *stk*, *loc*, *C*, *M*, *pch*) # *FRS*)
using *bisims* **unfolding** ‹$h' = h$› ‹$xcp' = None$›
‹$frs' = (Addr\ a' \# drop\ (length\ stk - d)\ stk, loc, C, M, pch) \# FRS$›
by *rule*
ultimately show *?thesis* **by**(*auto split del*: *if-split*)
qed
qed
qed(*insert exec*, *auto simp add*: *exec-1-iff elim*!: *jvmd-NormalE*)

lemma τ*Red1-simulates-exec-1-not-*τ:
assumes *wf*: *wf-J1-prog P*
and *exec*: *exec-1-d* (*compP2 P*) *t* (*Normal* (*xcp*, *h*, *frs*)) *ta* (*Normal* (*xcp'*, *h'*, *frs'*))
and *bisim*: *bisim1-list1 t h* (*e*, *xs*) *exs xcp frs*
and τ: ¬ τ*Move2* (*compP2 P*) (*xcp*, *h*, *frs*)
shows ∃ *e' xs' exs' ta' e'' xs'' exs''*. τ*Red1r P t h* ((*e*, *xs*), *exs*) ((*e'*, *xs'*), *exs'*) ∧
True,*P*,*t* ⊢*1* ⟨(*e'*, *xs'*)/*exs'*, *h*⟩ −*ta'*→ ⟨(*e''*, *xs''*)/*exs''*, *h'*⟩ ∧
¬ τ*Move1 P h* ((*e'*, *xs'*), *exs'*) ∧ *ta-bisim wbisim1 ta' ta* ∧
bisim1-list1 t h' (*e''*, *xs''*) *exs''* *xcp' frs'* ∧
(*call1 e* = *None* ∨
(*case frs of Nil* ⇒ *False* | (*stk*, *loc*, *C*, *M*, *pc*) # *FRS* ⇒ ∀ *M' n*. *instrs-of* (*compP2 P*) *C M* ! *pc* ≠ *Invoke M' n*) ∨
e' = *e* ∧ *xs'* = *xs* ∧ *exs'* = *exs*)

using *bisim*
proof *cases*
case (*bl1-Normal stk loc C M pc FRS Ts T body D*)
hence [*simp*]: *frs* = (*stk*, *loc*, *C*, *M*, *pc*) # *FRS*
and *conf*: *compTP P* ⊢ *t*: (*xcp*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *FRS*) √

and *sees*: $P \vdash C$ *sees* M: $Ts \to T = \lfloor body \rfloor$ *in* D
and *bisim*: P,*blocks1* 0 (*Class* D#Ts) *body*,$h \vdash (e, xs) \leftrightarrow (stk, loc, pc, xcp)$
and *lenxs*: *max-vars* $e \leq$ *length* xs
and *bisims*: *list-all2* (*bisim1-fr* P h) *exs FRS* **by** *auto*

from *sees-method-compP*[*OF sees*, **where** f=$\lambda C\ M\ Ts\ T$. *compMb2*]
have *sees'*: *compP2* $P \vdash C$ *sees* M: $Ts \to T = \lfloor$(*max-stack body*, *max-vars body*, *compE2 body* @ [*Return*], *compxE2 body* 0 0)$\rfloor$ *in* D
by(*simp add*: *compP2-def compMb2-def*)
from *bisim* **have** *pc*: $pc \leq$ *length* (*compE2 body*) **by**(*auto dest*: *bisim1-pc-length-compE2*)
from *conf* **have** *hconf*: *hconf* h *preallocated* h **and** *tconf*: P,$h \vdash t\ \surd t$
unfolding *correct-state-def* **by**(*simp-all add*: *compP2-def tconf-def*)

from *sees wf* **have** *bsok*: *bsok* (*blocks1* 0 (*Class* D # Ts) *body*) 0
by(*auto dest*!: *sees-wf-mdecl simp add*: *bsok-def wf-mdecl-def WT1-expr-locks*)

from *exec* **obtain** *check*: *check* (*compP2* P) (xcp, h, *frs*)
and *exec*: *compP2* P,$t \vdash$ (xcp, h, *frs*) $-ta-jvm\to$ (xcp', h', *frs'*)
by(*rule jvmd-NormalE*)(*auto simp add*: *exec-1-iff*)

from *wt-compTP-compP2*[*OF wf*] *exec conf*
have *conf'*: *compTP* $P \vdash t$: (xcp', h', *frs'*) $\surd$ **by**(*auto intro*: *BV-correct-1*)

show *?thesis*
proof(*cases xcp*)
case [*simp*]: *None*
from *exec* **have** *execi*: (ta, xcp', h', *frs'*) $\in$ *exec-instr* (*instrs-of* (*compP2* P) $C\ M$! pc) (*compP2* P) $t\ h\ stk\ loc\ C\ M\ pc\ FRS$
by(*simp add*: *exec-1-iff*)
show *?thesis*
proof(*cases length frs'* = *Suc* (*length FRS*))
case *True*
with *pc execi sees'* **have** *pc*: $pc <$ *length* (*compE2 body*)
by(*auto split*: *if-split-asm simp add*: *split-beta*)
with *execi sees'* **have** *execi*: (ta, xcp', h', *frs'*) $\in$ *exec-instr* (*compE2 body* ! pc) (*compP2* P) $t\ h\ stk\ loc\ C\ M\ pc\ FRS$
by(*simp*)
from τ *sees' True pc* **have** τi: $\neg\ \tau$*move2* (*compP2* P) $h\ stk\ body\ pc\ None$ **by**(*simp add*: τ*move2-iff*)
from *execi True sees' pc* **have** $\exists\, stk'\ loc'\ pc'$. *frs'* = ($stk'$, loc', C, M, pc') # *FRS*
by(*cases* (*compE2 body* @ [*Return*]) ! pc)(*auto split*: *if-split-asm sum.split-asm simp add*: *split-beta*, *auto split*: *extCallRet.splits*)
then obtain $stk'\ loc'\ pc'$ **where** [*simp*]: *frs'* = (stk', loc', C, M, pc') # *FRS* **by** *blast*
from *conf* **obtain** ST **where** *compP2* P,$h \vdash stk$ [:≤] ST **by**(*auto simp add*: *correct-state-def conf-f-def2*)
hence ST: P,$h \vdash stk$ [:≤] ST **by**(*rule List.list-all2-mono*)(*simp add*: *compP2-def*)
from *execi sees True check pc*
have *exec'*: *exec-move-d* $P\ t$ (*blocks1* 0 (*Class* D#Ts) *body*) h (stk, loc, pc, xcp) $ta\ h'$ (stk', loc', pc', xcp')
apply(*auto simp add*: *compP2-def compMb2-def exec-move-def check-def exec-meth-instr split*: *if-split-asm sum.split-asm*)
apply(*cases compE2 body* ! pc)
apply(*auto simp add*: *neq-Nil-conv split-beta split*: *if-split-asm sum.split-asm*)
apply(*force split*: *extCallRet.split-asm*)
apply(*cases compE2 body* ! pc, *auto simp add*: *split-beta neq-Nil-conv split*: *if-split-asm*

sum.split-asm)
done
from *red1-simulates-exec-instr*[*OF wf hconf tconf bisim this - bsok ST*] *lenxs* τi **obtain** e'' xs'' ta' e' xs'
where *bisim'*: *P,blocks1 0* (*Class D#Ts*) *body,h'* $\vdash$ (e'', xs'') $\leftrightarrow$ (*stk'*, *loc'*, *pc'*, *xcp'*)
and *red1*: τ*red1r P t h* (*e*, *xs*) (*e'*, *xs'*) **and** *red2*: *True,P,t* $\vdash$*1* $\langle e', (h, xs')\rangle$ $-ta'\rightarrow$ $\langle e'', (h', xs'')\rangle$
and τ*1*: $\neg$ τ*move1 P h e'* **and** *tabisim*: *ta-bisim wbisim1* (*extTA2J1 P ta'*) *ta*
and *call*: *call1 e* = *None* $\vee$ *no-call2* (*blocks1 0* (*Class D # Ts*) *body*) *pc* $\vee$ $e' = e \wedge xs' = xs$
by(*fastforce simp del*: *blocks1.simps*)
from *red1* **have** *Red1*: τ*Red1r P t h* ((*e*, *xs*), *exs*) ((*e'*, *xs'*), *exs*)
by(*rule* τ*red1r-into-*τ*Red1r*)
moreover from *red2* **have** *True,P,t* $\vdash$*1* $\langle (e', xs')/exs, h\rangle$ $-extTA2J1\ P\ ta'\rightarrow$ $\langle (e'', xs'')/exs, h'\rangle$
by(*rule red1Red*)
moreover from τ*1 red2* **have** $\neg$ τ*Move1 P h* ((*e'*, *xs'*), *exs*) **by** *auto*
moreover from τ*red1r-max-vars*[*OF red1*] *lenxs* τ*red1r-preserves-len*[*OF red1*]
have *max-vars e'* $\leq$ *length xs'* **by** *simp*
with *red1-preserves-len*[*OF red2*] *red1-max-vars*[*OF red2*]
have *max-vars e''* $\leq$ *length xs''* **by** *simp*
with *conf' sees bisim'*
have *bisim1-list1 t h'* (e'', xs'') *exs xcp'* ((*stk'*, *loc'*, *C*, *M*, *pc'*) # *FRS*)
unfolding ‹*frs'* = (*stk'*, *loc'*, *C*, *M*, *pc'*) # *FRS*›
proof
from *red2* **have** *hext h h'* **by**(*auto dest*: *red1-hext-incr*)
from *bisims* **show** *list-all2* (*bisim1-fr P h'*) *exs FRS*
by(*rule List.list-all2-mono*)(*erule bisim1-fr-hext-mono*[*OF* - ‹*hext h h'*›])
qed
moreover from *call sees'*
have *call1 e* = *None* $\vee$ ($\forall$ *M' n. instrs-of* (*compP2 P*) *C M* ! *pc* $\neq$ *Invoke M' n*) $\vee$ $e' = e \wedge xs' = xs$
by(*auto simp add*: *no-call2-def*)
ultimately show *?thesis* **using** *tabisim*
by(*fastforce simp del*: *split-paired-Ex*)
next
case *False*
with *execi sees pc compE2-not-Return*[*of body*]
have (*pc* = *length* (*compE2 body*) $\vee$ ($\exists$ *M n. compE2 body* ! *pc* = *Invoke M n*)) $\wedge$ *xcp'* = *None*
apply(*cases compE2 body* ! *pc*)
apply(*auto split*: *if-split-asm sum.split-asm simp add*: *split-beta compP2-def compMb2-def*)
apply(*auto split*: *extCallRet.splits*)
apply(*metis in-set-conv-nth*)+
done
hence [*simp*]: *xcp'* = *None*
and *pc* = *length* (*compE2 body*) $\vee$ ($\exists$ *M n. compE2 body* ! *pc* = *Invoke M n*) **by** *simp-all*
moreover
{ assume [*simp*]: *pc* = *length* (*compE2 body*)
with *sees-method-compP*[*OF sees*, **where** *f*=λ*C M Ts T. compMb2*] τ **have** *False* **by**(*auto simp add*: *compMb2-def compP2-def*)
hence *?thesis* **.. }**
moreover {
assume $\exists$ *M n. compE2 body* ! *pc* = *Invoke M n*
and *pc* $\neq$ *length* (*compE2 body*)
with *pc* **obtain** *MM n* **where** *ins*: *compE2 body* ! *pc* = *Invoke MM n*
and *pc*: *pc* < *length* (*compE2 body*) **by** *auto*
with *bisim1-Invoke-stkD*[*OF bisim*[*unfolded None*], *of MM n*] **obtain** *vs' v' stk'*

```
        where [simp]: stk = vs' @ v' # stk' n = length vs' by auto
       with False τ sees' execi pc ins have False by auto (auto split: extCallRet.split-asm) }
      ultimately show ?thesis by blast
    qed
  next
    case [simp]: (Some ad)
    from bisim have pc: pc < length (compE2 body) by(auto dest: bisim1-xcp-pcD)
    with τ sees' have False by auto
    thus ?thesis ..
  qed
qed(insert exec, auto simp add: exec-1-iff elim!: jvmd-NormalE)

end

end
```

7.19 Correctness of Stage 2: The multithreaded setting

```
theory Correctness2
imports
  J1JVM
  JVMJ1
  ../BV/BVProgressThreaded
begin

declare Listn.lesub-list-impl-same-size[simp del]

context J1-JVM-heap-conf-base begin

lemma bisim1-list1-has-methodD: bisim1-list1 t h ex exs xcp ((stk, loc, C, M, pc) # frs) ⟹ P ⊢ C
has M
by(fastforce elim!: bisim1-list1.cases intro: has-methodI)

end

declare compP-has-method [simp]

sublocale J1-JVM-heap-conf-base < Red1-exec:
   delay-bisimulation-base mred1 P t mexec (compP2 P) t wbisim1 t ta-bisim wbisim1 τMOVE1 P
τMOVE2 (compP2 P)
  for t
.

sublocale J1-JVM-heap-conf-base < Red1-execd: delay-bisimulation-base
  mred1 P t
  mexecd (compP2 P) t
  wbisim1 t
  ta-bisim wbisim1
  τMOVE1 P
  τMOVE2 (compP2 P)
  for t
.
```

context *JVM-heap-base* **begin**

lemma *τexec-1-d-silent-move*:
τexec-1-d P t (*xcp*, *h*, *frs*) (*xcp'*, *h'*, *frs'*)
⟹ *τtrsys.silent-move* (*mexecd P t*) (*τMOVE2 P*) ((*xcp*, *frs*), *h*) ((*xcp'*, *frs'*), *h'*)
apply(*rule τtrsys.silent-move.intros*)
apply *auto*
apply(*rule exec-1-d-NormalI*)
apply(*auto simp add*: *exec-1-iff exec-d-def*)
done

lemma *silent-move-τexec-1-d*:
τtrsys.silent-move (*mexecd P t*) (*τMOVE2 P*) ((*xcp*, *frs*), *h*) ((*xcp'*, *frs'*), *h'*)
⟹ *τexec-1-d P t* (*xcp*, *h*, *frs*) (*xcp'*, *h'*, *frs'*)
apply(*erule τtrsys.silent-move.cases*)
apply *clarsimp*
apply(*erule jvmd-NormalE*)
apply(*auto simp add*: *exec-1-iff*)
done

lemma *τExec-1-dr-rtranclpD*:
τExec-1-dr P t (*xcp*, *h*, *frs*) (*xcp'*, *h'*, *frs'*)
⟹ *τtrsys.silent-moves* (*mexecd P t*) (*τMOVE2 P*) ((*xcp*, *frs*), *h*) ((*xcp'*, *frs'*), *h'*)
by(*induct rule*: *rtranclp-induct3*)(*blast intro*: *rtranclp.rtrancl-into-rtrancl τexec-1-d-silent-move*)+

lemma *τExec-1-dt-tranclpD*:
τExec-1-dt P t (*xcp*, *h*, *frs*) (*xcp'*, *h'*, *frs'*)
⟹ *τtrsys.silent-movet* (*mexecd P t*) (*τMOVE2 P*) ((*xcp*, *frs*), *h*) ((*xcp'*, *frs'*), *h'*)
by(*induct rule*: *tranclp-induct3*)(*blast intro*: *tranclp.trancl-into-trancl τexec-1-d-silent-move*)+

lemma *rtranclp-τExec-1-dr*:
τtrsys.silent-moves (*mexecd P t*) (*τMOVE2 P*) ((*xcp*, *frs*), *h*) ((*xcp'*, *frs'*), *h'*)
⟹ *τExec-1-dr P t* (*xcp*, *h*, *frs*) (*xcp'*, *h'*, *frs'*)
by(*induct rule*: *rtranclp-induct*[*of* - ((*ax*, *ay*), *az*) ((*bx*, *by*), *bz*), *split-rule*, *consumes 1*])(*blast intro*: *rtranclp.rtrancl-into-rtrancl silent-move-τexec-1-d*)+

lemma *tranclp-τExec-1-dt*:
τtrsys.silent-movet (*mexecd P t*) (*τMOVE2 P*) ((*xcp*, *frs*), *h*) ((*xcp'*, *frs'*), *h'*)
⟹ *τExec-1-dt P t* (*xcp*, *h*, *frs*) (*xcp'*, *h'*, *frs'*)
by(*induct rule*: *tranclp-induct*[*of* - ((*ax*, *ay*), *az*) ((*bx*, *by*), *bz*), *split-rule*, *consumes 1*])(*blast intro*: *tranclp.trancl-into-trancl silent-move-τexec-1-d*)+

lemma *τExec-1-dr-conv-rtranclp*:
τExec-1-dr P t (*xcp*, *h*, *frs*) (*xcp'*, *h'*, *frs'*) =
τtrsys.silent-moves (*mexecd P t*) (*τMOVE2 P*) ((*xcp*, *frs*), *h*) ((*xcp'*, *frs'*), *h'*)
by(*blast intro*: *τExec-1-dr-rtranclpD rtranclp-τExec-1-dr*)

lemma *τExec-1-dt-conv-tranclp*:
τExec-1-dt P t (*xcp*, *h*, *frs*) (*xcp'*, *h'*, *frs'*) =
τtrsys.silent-movet (*mexecd P t*) (*τMOVE2 P*) ((*xcp*, *frs*), *h*) ((*xcp'*, *frs'*), *h'*)
by(*blast intro*: *τExec-1-dt-tranclpD tranclp-τExec-1-dt*)

end

context *J1-JVM-conf-read* **begin**

lemma *Red1-execd-weak-bisim*:
assumes *wf*: *wf-J1-prog P*
shows *delay-bisimulation-measure* (*mred1 P t*) (*mexecd* (*compP2 P*) *t*) (*wbisim1 t*) (*ta-bisim wbisim1*) (*τMOVE1 P*) (*τMOVE2* (*compP2 P*)) (λ(((*e*, *xs*), *exs*), *h*) (((*e′*, *xs′*), *exs′*), *h′*). *sim12-size e* < *sim12-size e′*) (λ(*xcpfrs*, *h*) (*xcpfrs′*, *h*). *sim21-size* (*compP2 P*) *xcpfrs xcpfrs′*)
proof
fix *s1 s2 s1′*
assume *wbisim1 t s1 s2* **and** *τtrsys.silent-move* (*mred1 P t*) (*τMOVE1 P*) *s1 s1′*
moreover obtain *e xs exs h* **where** *s1*: *s1* = (((*e*, *xs*), *exs*), *h*) **by**(*cases s1*) *auto*
moreover obtain *e′ xs′ exs′ h1′* **where** *s1′*: *s1′* = (((*e′*, *xs′*), *exs′*), *h1′*) **by**(*cases s1′*) *auto*
moreover obtain *xcp frs h2* **where** *s2*: *s2* = ((*xcp*, *frs*), *h2*) **by**(*cases s2*) *auto*
ultimately have [*simp*]: *h2* = *h* **and** *red*: *True*,*P*,*t* ⊢1 ⟨(*e*, *xs*)/*exs*,*h*⟩ −*ε*→ ⟨(*e′*, *xs′*)/*exs′*,*h1′*⟩
and *τ*: *τMove1 P h* ((*e*, *xs*), *exs*) **and** *bisim*: *bisim1-list1 t h* (*e*, *xs*) *exs xcp frs* **by**(*auto*)
from *red τ bisim* **have** *h1′* [*simp*]: *h1′* = *h* **by**(*auto dest*: *τmove1-heap-unchanged elim*!: *Red1.cases bisim1-list1.cases*)
from *exec-1-simulates-Red1-τ*[*OF wf red*[*unfolded h1′*] *bisim τ*] **obtain** *xcp′ frs′*
where *exec*: (*if sim12-size e′* < *sim12-size e then τExec-1-dr else τExec-1-dt*) (*compP2 P*) *t* (*xcp*, *h*, *frs*) (*xcp′*, *h*, *frs′*)
and *bisim′*: *bisim1-list1 t h* (*e′*, *xs′*) *exs′ xcp′ frs′* **by** *blast*
from *exec* **have** (*if* (λ(((*e*, *xs*), *exs*), *h*) (((*e′*, *xs′*), *exs′*), *h′*). *sim12-size e* < *sim12-size e′*) (((*e′*, *xs′*), *exs′*), *h*) (((*e*, *xs*), *exs*), *h*) *then τtrsys.silent-moves* (*mexecd* (*compP2 P*) *t*) (*τMOVE2* (*compP2 P*)) *else τtrsys.silent-movet* (*mexecd* (*compP2 P*) *t*) (*τMOVE2* (*compP2 P*))) ((*xcp*, *frs*), *h*) ((*xcp′*, *frs′*), *h*)
by(*auto simp add*: *τExec-1-dr-conv-rtranclp τExec-1-dt-conv-tranclp*)
thus *wbisim1 t s1′ s2* ∧ (λ(((*e*, *xs*), *exs*), *h*) (((*e′*, *xs′*), *exs′*), *h′*). *sim12-size e* < *sim12-size e′*)$^{++}$ *s1′ s1* ∨
(∃ *s2′*. (*τtrsys.silent-movet* (*mexecd* (*compP2 P*) *t*) (*τMOVE2* (*compP2 P*))) *s2 s2′* ∧ *wbisim1 t s1′ s2′*)
using *bisim′ s1 s1′ s2*
by −(*rule delay-bisimulation-base.simulation-silent1I′*, *auto split del*: *if-split*)
next
fix *s1 s2 s2′*
assume *wbisim1 t s1 s2* **and** *τtrsys.silent-move* (*mexecd* (*compP2 P*) *t*) (*τMOVE2* (*compP2 P*)) *s2 s2′*
moreover obtain *e xs exs h1* **where** *s1*: *s1* = (((*e*, *xs*), *exs*), *h1*) **by**(*cases s1*) *auto*
moreover obtain *xcp frs h* **where** *s2*: *s2* = ((*xcp*, *frs*), *h*) **by**(*cases s2*) *auto*
moreover obtain *xcp′ frs′ h2′* **where** *s2′*: *s2′* = ((*xcp′*, *frs′*), *h2′*) **by**(*cases s2′*) *auto*
ultimately have [*simp*]: *h1* = *h* **and** *exec*: *exec-1-d* (*compP2 P*) *t* (*Normal* (*xcp*, *h*, *frs*)) *ε* (*Normal* (*xcp′*, *h2′*, *frs′*))
and *τ*: *τMove2* (*compP2 P*) (*xcp*, *h*, *frs*) **and** *bisim*: *bisim1-list1 t h* (*e*, *xs*) *exs xcp frs* **by**(*auto*)
from *τRed1-simulates-exec-1-τ*[*OF wf exec bisim τ*]
obtain *e′ xs′ exs′* **where** [*simp*]: *h2′* = *h*
and *red*: (*if sim21-size* (*compP2 P*) (*xcp′*, *frs′*) (*xcp*, *frs*) *then τRed1r else τRed1t*) *P t h* ((*e*, *xs*), *exs*) ((*e′*, *xs′*), *exs′*)
and *bisim′*: *bisim1-list1 t h* (*e′*, *xs′*) *exs′ xcp′ frs′* **by** *blast*
from *red* **have** (*if* ((λ(*xcpfrs*, *h*) (*xcpfrs′*, *h*). *sim21-size* (*compP2 P*) *xcpfrs xcpfrs′*) ((*xcp′*, *frs′*), *h2′*) ((*xcp*, *frs*), *h*)) *then τtrsys.silent-moves* (*mred1 P t*) (*τMOVE1 P*) *else τtrsys.silent-movet* (*mred1 P t*) (*τMOVE1 P*)) (((*e*, *xs*), *exs*), *h*) (((*e′*, *xs′*), *exs′*), *h*)
by(*auto dest*: *τRed1r-rtranclpD τRed1t-tranclpD*)
thus *wbisim1 t s1 s2′* ∧ (λ(*xcpfrs*, *h*) (*xcpfrs′*, *h*). *sim21-size* (*compP2 P*) *xcpfrs xcpfrs′*)$^{++}$ *s2′ s2* ∨
(∃ *s1′*. *τtrsys.silent-movet* (*mred1 P t*) (*τMOVE1 P*) *s1 s1′* ∧ *wbisim1 t s1′ s2′*)

using *bisim′ s1 s2 s2′*
by −(*rule delay-bisimulation-base.simulation-silent2I′*, *auto split del*: *if-split*)
next
fix *s1 s2 tl1 s1′*
assume *wbisim1 t s1 s2* **and** *mred1 P t s1 tl1 s1′* **and** ¬ *τMOVE1 P s1 tl1 s1′*
moreover obtain *e xs exs h* **where** *s1*: *s1* = (((*e*, *xs*), *exs*), *h*) **by**(*cases s1*) *auto*
moreover obtain *e′ xs′ exs′ h1′* **where** *s1′*: *s1′* = (((*e′*, *xs′*), *exs′*), *h1′*) **by**(*cases s1′*) *auto*
moreover obtain *xcp frs h2* **where** *s2*: *s2* = ((*xcp*, *frs*), *h2*) **by**(*cases s2*) *auto*
ultimately have [*simp*]: *h2* = *h* **and** *red*: *True*,*P*,*t* ⊢*1* ⟨(*e*, *xs*)/*exs*,*h*⟩ −*tl1*→ ⟨(*e′*, *xs′*)/*exs′*,*h1′*⟩
and *τ*: ¬ *τMove1 P h* ((*e*, *xs*), *exs*) **and** *bisim*: *bisim1-list1 t h* (*e*, *xs*) *exs xcp frs*
by(*fastforce elim*!: *Red1.cases dest*: *red1-τ-taD*)+
from *exec-1-simulates-Red1-not-τ*[*OF wf red bisim τ*] **obtain** *ta′ xcp′ frs′ xcp″ frs″*
where *exec1*: *τExec-1-dr* (*compP2 P*) *t* (*xcp*, *h*, *frs*) (*xcp′*, *h*, *frs′*)
and *exec2*: *exec-1-d* (*compP2 P*) *t* (*Normal* (*xcp′*, *h*, *frs′*)) *ta′* (*Normal* (*xcp″*, *h1′*, *frs″*))
and *τ′*: ¬ *τMove2* (*compP2 P*) (*xcp′*, *h*, *frs′*)
and *bisim′*: *bisim1-list1 t h1′* (*e′*, *xs′*) *exs′ xcp″ frs″*
and *ta′*: *ta-bisim wbisim1 tl1 ta′* **by** *blast*
from *exec1* **have** *τtrsys.silent-moves* (*mexecd* (*compP2 P*) *t*) (*τMOVE2* (*compP2 P*)) ((*xcp*, *frs*), *h*) ((*xcp′*, *frs′*), *h*)
by(*rule τExec-1-dr-rtranclpD*)
thus ∃ *s2′ s2″ tl2*. *τtrsys.silent-moves* (*mexecd* (*compP2 P*) *t*) (*τMOVE2* (*compP2 P*)) *s2 s2′* ∧
mexecd (*compP2 P*) *t s2′ tl2 s2″* ∧ ¬ *τMOVE2* (*compP2 P*) *s2′ tl2 s2″* ∧
wbisim1 t s1′ s2″ ∧ *ta-bisim wbisim1 tl1 tl2*
using *bisim′ exec2 τ′ s1 s1′ s2 ta′* **unfolding** ‹*h2* = *h*›
apply(*subst* (*1 2*) *split-paired-Ex*)
apply(*subst* (*1 2*) *split-paired-Ex*)
by *clarify* ((*rule exI conjI*|*assumption*)+, *auto*)
next
fix *s1 s2 tl2 s2′*
assume *wbisim1 t s1 s2* **and** *mexecd* (*compP2 P*) *t s2 tl2 s2′* **and** ¬ *τMOVE2* (*compP2 P*) *s2 tl2 s2′*
moreover obtain *e xs exs h1* **where** *s1*: *s1* = (((*e*, *xs*), *exs*), *h1*) **by**(*cases s1*) *auto*
moreover obtain *xcp frs h* **where** *s2*: *s2* = ((*xcp*, *frs*), *h*) **by**(*cases s2*) *auto*
moreover obtain *xcp′ frs′ h2′* **where** *s2′*: *s2′* = ((*xcp′*, *frs′*), *h2′*) **by**(*cases s2′*) *auto*
ultimately have [*simp*]: *h1* = *h* **and** *exec*: *exec-1-d* (*compP2 P*) *t* (*Normal* (*xcp*, *h*, *frs*)) *tl2* (*Normal* (*xcp′*, *h2′*, *frs′*))
and *τ*: ¬ *τMove2* (*compP2 P*) (*xcp*, *h*, *frs*) **and** *bisim*: *bisim1-list1 t h* (*e*, *xs*) *exs xcp frs*
apply *auto*
apply(*erule jvmd-NormalE*)
apply(*cases xcp*)
apply *auto*
apply(*rename-tac stk loc C M pc frs*)
apply(*case-tac instrs-of* (*compP2 P*) *C M* ! *pc*)
apply(*simp-all split*: *if-split-asm*)
apply(*auto dest*!: *τexternal-red-external-aggr-TA-empty simp add*: *check-def has-method-def τexternal-def τexternal′-def*)
done
from *τRed1-simulates-exec-1-not-τ*[*OF wf exec bisim τ*] **obtain** *e′ xs′ exs′ ta′ e″ xs″ exs″*
where *red1*: *τRed1r P t h* ((*e*, *xs*), *exs*) ((*e′*, *xs′*), *exs′*)
and *red2*: *True*,*P*,*t* ⊢*1* ⟨(*e′*, *xs′*)/*exs′*,*h*⟩ −*ta′*→ ⟨(*e″*, *xs″*)/*exs″*,*h2′*⟩
and *τ′*: ¬ *τMove1 P h* ((*e′*, *xs′*), *exs′*) **and** *ta′*: *ta-bisim wbisim1 ta′ tl2*
and *bisim′*: *bisim1-list1 t h2′* (*e″*, *xs″*) *exs″ xcp′ frs′* **by** *blast*
from *red1* **have** *τtrsys.silent-moves* (*mred1 P t*) (*τMOVE1 P*) (((*e*, *xs*), *exs*), *h*) (((*e′*, *xs′*), *exs′*), *h*)

by(*rule* τ*Red1r-rtranclpD*)
thus $\exists$ *s1' s1'' tl1*. τ*trsys.silent-moves* (*mred1 P t*) (τ*MOVE1 P*) *s1 s1'* $\wedge$ *mred1 P t s1' tl1 s1''* $\wedge$
$\neg$ τ*MOVE1 P s1' tl1 s1''* $\wedge$ *wbisim1 t s1'' s2'* $\wedge$ *ta-bisim wbisim1 tl1 tl2*
using *bisim' red2* τ' *s1 s2 s2'* ‹*h1 = h*› *ta'*
apply −
apply(*rule exI*[**where** *x*=(((*e'*, *xs'*), *exs'*), *h*)])
apply(*rule exI*[**where** *x*=(((*e''*, *xs''*), *exs''*), *h2'*)])
apply(*rule exI*[**where** *x*=*ta'*])
apply *auto*
done
next
have *wf* (*inv-image* {(*x*, *y*). *x* < *y*} (λ(((*e*, *xs*), *exs*), *h*). *sim12-size e*))
by(*rule wf-inv-image*)(*rule wf-less*)
also have *inv-image* {(*x*, *y*). *x* < *y*} (λ(((*e*, *xs*), *exs*), *h*). *sim12-size e*) =
{(*x*, *y*). (λ(((*e*, *xs*), *exs*), *h*) (((*e'*, *xs'*), *exs'*), *h'*). *sim12-size e* < *sim12-size e'*) *x y*} **by** *auto*
finally show *wfP* (λ(((*e*, *xs*), *exs*), *h*) (((*e'*, *xs'*), *exs'*), *h'*). *sim12-size e* < *sim12-size e'*)
unfolding *wfp-def* **.**
next
from *wfP-sim21-size*
have *wf* {(*xcpfrs*, *xcpfrs'*). *sim21-size* (*compP2 P*) *xcpfrs xcpfrs'*} **by**(*unfold wfp-def*)
hence *wf* (*inv-image* {(*xcpfrs*, *xcpfrs'*). *sim21-size* (*compP2 P*) *xcpfrs xcpfrs'*} *fst*) **by**(*rule wf-inv-image*)
also have *inv-image* {(*xcpfrs*, *xcpfrs'*). *sim21-size* (*compP2 P*) *xcpfrs xcpfrs'*} *fst* =
{((*xcpfrs*, *h*), (*xcpfrs'*, *h*)). *sim21-size* (*compP2 P*) *xcpfrs xcpfrs'*} **by** *auto*
also have ... = {(*x*, *y*). (λ(*xcpfrs*, *h*) (*xcpfrs'*, *h*). *sim21-size* (*compP2 P*) *xcpfrs xcpfrs'*) *x y*}
by(*auto*)
finally show *wfP* (λ(*xcpfrs*, *h*) (*xcpfrs'*, *h*). *sim21-size* (*compP2 P*) *xcpfrs xcpfrs'*)
unfolding *wfp-def* **.**
qed

lemma *Red1-execd-delay-bisim*:
assumes *wf*: *wf-J1-prog P*
shows *delay-bisimulation-diverge* (*mred1 P t*) (*mexecd* (*compP2 P*) *t*) (*wbisim1 t*) (*ta-bisim wbisim1*)
(τ*MOVE1 P*) (τ*MOVE2* (*compP2 P*))
proof −
interpret *delay-bisimulation-measure*
mred1 P t mexecd (*compP2 P*) *t wbisim1 t ta-bisim wbisim1* τ*MOVE1 P* τ*MOVE2* (*compP2 P*)
λ(((*e*, *xs*), *exs*), *h*) (((*e'*, *xs'*), *exs'*), *h'*). *sim12-size e* < *sim12-size e'*
λ(*xcpfrs*, *h*) (*xcpfrs'*, *h*). *sim21-size* (*compP2 P*) *xcpfrs xcpfrs'*
using *wf* **by**(*rule Red1-execd-weak-bisim*)
show *?thesis* **by**(*unfold-locales*)
qed

end

definition *bisim-wait1JVM* ::
'addr jvm-prog ⇒ (*'addr expr1* × *'addr locals1*) × (*'addr expr1* × *'addr locals1*) *list* ⇒ *'addr jvm-thread-state* ⇒ *bool*
where
bisim-wait1JVM P ≡
λ((*e1*, *xs1*), *exs1*) (*xcp*, *frs*). *call1 e1* ≠ *None* ∧
(*case frs of Nil* ⇒ *False* | (*stk*, *loc*, *C*, *M*, *pc*) # *frs'* ⇒ ∃ *M' n*. *instrs-of P C M* ! *pc* = *Invoke M' n*)

sublocale *J1-JVM-heap-conf-base* < *Red1-execd*:

FWbisimulation-base
final-expr1
mred1 P
JVM-final
mexecd (compP2 P)
convert-RA
wbisim1
bisim-wait1JVM (compP2 P)
.

sublocale *JVM-heap-base* < *execd-mthr*:
τmultithreaded
JVM-final
mexecd P
convert-RA
τMOVE2 P
for *P*
by(*unfold-locales*)

sublocale *J1-JVM-heap-conf-base* < *Red1-execd*:
FWdelay-bisimulation-base
final-expr1
mred1 P
JVM-final
mexecd (compP2 P)
convert-RA
wbisim1
bisim-wait1JVM (compP2 P)
τMOVE1 P
τMOVE2 (compP2 P)
by(*unfold-locales*)

context *J1-JVM-conf-read* **begin**

theorem *Red1-exec1-FWwbisim*:
assumes *wf*: *wf-J1-prog P*
shows *FWdelay-bisimulation-diverge final-expr1 (mred1 P) JVM-final (mexecd (compP2 P)) wbisim1 (bisim-wait1JVM (compP2 P)) (τMOVE1 P) (τMOVE2 (compP2 P))*
proof −
let *?exec* = *mexecd (compP2 P)*
let *?τexec* = λ*t*. *τtrsys.silent-moves (mexecd (compP2 P) t) (τMOVE2 (compP2 P))*
let *?τred* = λ*t*. *τtrsys.silent-moves (mred1 P t) (τMOVE1 P)*
interpret *delay-bisimulation-diverge*
mred1 P t ?exec t wbisim1 t ta-bisim wbisim1 τMOVE1 P τMOVE2 (compP2 P)
for *t*
using *wf* **by**(*rule Red1-execd-delay-bisim*)
show *?thesis*
proof
fix *t s1 s2*
assume *wbisim1 t s1 s2* (λ(*x1*, *m*). *final-expr1 x1*) *s1*
moreover obtain *e xs exs m1* **where** [*simp*]: *s1* = (((*e*, *xs*), *exs*), *m1*) **by**(*cases s1*) *auto*
moreover obtain *xcp frs m2* **where** [*simp*]: *s2* = ((*xcp*, *frs*), *m2*) **by**(*cases s2*) *auto*
ultimately have [*simp*]: *m2* = *m1 exs* = []
and *bisim1-list1 t m1* (*e*, *xs*) [] *xcp frs*

```
    and final e by auto
  from ‹bisim1-list1 t m1 (e, xs) [] xcp frs› ‹final e›
  show ∃ s2'. ?τexec t s2 s2' ∧ wbisim1 t s1 s2' ∧ (λ(x2, m). JVM-final x2) s2'
  proof cases
    case (bl1-Normal stk loc C M pc frs' Ts T body D)
    hence [simp]: frs = [(stk, loc, C, M, pc)]
      and conf: compTP P ⊢ t:(xcp, m1, frs) √
      and sees: P ⊢ C sees M: Ts→T = ⌊body⌋ in D
      and bisim: P,blocks1 0 (Class D # Ts) body,m1 ⊢ (e, xs) ↔ (stk, loc, pc, xcp)
      and var: max-vars e ≤ length xs by auto
    from ‹final e› show ?thesis
    proof cases
      fix v
      assume [simp]: e = Val v
      with bisim have [simp]: xcp = None xs = loc
        and exec: τExec-mover-a P t (blocks1 0 (Class D # Ts) body) m1 (stk, loc, pc, xcp) ([v], loc,
length (compE2 body), None)
        by(auto dest!: bisim1Val2D1)
      from exec have τExec-mover-a P t body m1 (stk, loc, pc, xcp) ([v], loc, length (compE2 body),
None)
        unfolding τExec-mover-blocks1 .
      with sees have τExec-1r (compP2 P) t (xcp, m1, frs) (None, m1, [([v], loc, C, M, length
(compE2 body))])
        by(auto intro: τExec-mover-τExec-1r)
      with wt-compTP-compP2[OF wf]
     have execd: τExec-1-dr (compP2 P) t (xcp, m1, frs) (None, m1, [([v], loc, C, M, length (compE2
body))])
        using conf by(rule τExec-1r-τExec-1-dr)
      also from sees-method-compP[OF sees, of λC M Ts T. compMb2] sees max-stack1[of body]
      have τexec-1-d (compP2 P) t (None, m1, [([v], loc, C, M, length (compE2 body))]) (None, m1,
[])
      by(auto simp add: τexec-1-d-def compP2-def compMb2-def check-def has-methodI intro: exec-1I)
      finally have ?τexec t s2 ((None, []), m1)
        unfolding τExec-1-dr-conv-rtranclp by simp
      moreover have JVM-final (None, []) by simp
     moreover from conf have hconf m1 preallocated m1 unfolding correct-state-def by(simp-all)
      hence wbisim1 t s1 ((None, []), m1) by(auto intro: bisim1-list1.intros)
      ultimately show ?thesis by blast
    next
      fix a
      assume [simp]: e = throw (addr a)
        hence ∃ stk' loc' pc'. τExec-mover-a P t body m1 (stk, loc, pc, xcp) (stk', loc', pc', ⌊a⌋) ∧
P,blocks1 0 (Class D # Ts) body,m1 ⊢ (Throw a, xs) ↔ (stk', loc', pc', ⌊a⌋)
      proof(cases xcp)
        case None
        with bisim show ?thesis
       by(fastforce dest!: bisim1-Throw-τExec-movet simp del: blocks1.simps intro: tranclp-into-rtranclp)
      next
        case (Some a')
        with bisim have a = a' by(auto dest: bisim1-ThrowD)
        with Some bisim show ?thesis by(auto)
      qed
      then obtain stk' loc' pc'
        where exec: τExec-mover-a P t body m1 (stk, loc, pc, xcp) (stk', loc', pc', ⌊a⌋)
```

```
        and bisim': P,blocks1 0 (Class D # Ts) body,m1 ⊢ (throw (addr a), xs) ↔ (stk', loc', pc',
⌊a⌋) by blast
      with sees have τExec-1r (compP2 P) t (xcp, m1, frs) (⌊a⌋, m1, [(stk', loc', C, M, pc')])
        by(auto intro: τExec-mover-τExec-1r)
      with wt-compTP-compP2[OF wf]
      have execd: τExec-1-dr (compP2 P) t (xcp, m1, frs) (⌊a⌋, m1, [(stk', loc', C, M, pc')])
        using conf by(rule τExec-1r-τExec-1-dr)
      also {
        from bisim1-xcp-Some-not-caught[OF bisim', of λC M Ts T. compMb2 0 0]
        have match-ex-table (compP2 P) (cname-of m1 a) pc' (compxE2 body 0 0) = None by(simp
add: compP2-def)
        moreover from bisim' have pc' < length (compE2 body) by(auto dest: bisim1-ThrowD)
        ultimately have τexec-1 (compP2 P) t (⌊a⌋, m1, [(stk', loc', C, M, pc')]) (⌊a⌋, m1, [])
          using sees-method-compP[OF sees, of λC M Ts T. compMb2] sees
          by(auto simp add: τexec-1-def compP2-def compMb2-def has-methodI intro: exec-1I)
        moreover from wt-compTP-compP2[OF wf] execd conf
      have compTP P ⊢ t:(⌊a⌋, m1, [(stk', loc', C, M, pc')]) √ by(rule τExec-1-dr-preserves-correct-state)
        ultimately have τexec-1-d (compP2 P) t (⌊a⌋, m1, [(stk', loc', C, M, pc')]) (⌊a⌋, m1, [])
          using wt-compTP-compP2[OF wf]
        by(auto simp add: τexec-1-def τexec-1-d-def welltyped-commute[symmetric] elim: jvmd-NormalE)
}
      finally have ?τexec t s2 ((⌊a⌋, []), m1)
        unfolding τExec-1-dr-conv-rtranclp by simp
      moreover have JVM-final (⌊a⌋, []) by simp
      moreover from conf have hconf m1 preallocated m1 by(simp-all add: correct-state-def)
      hence wbisim1 t s1 ((⌊a⌋, []), m1) by(auto intro: bisim1-list1.intros)
      ultimately show ?thesis by blast
    qed
  qed(auto intro!: exI bisim1-list1.intros)
 next
  fix t s1 s2
  assume wbisim1 t s1 s2 (λ(x2, m). JVM-final x2) s2
  moreover obtain e xs exs m1 where [simp]: s1 = (((e, xs), exs), m1) by(cases s1) auto
  moreover obtain xcp frs m2 where [simp]: s2 = ((xcp, frs), m2) by(cases s2) auto
  ultimately have [simp]: m2 = m1 exs = [] frs = []
    and bisim: bisim1-list1 t m1 (e, xs) [] xcp [] by(auto elim: bisim1-list1.cases)
  hence final e by(auto elim: bisim1-list1.cases)
  thus ∃s1'. ?τred t s1 s1' ∧ wbisim1 t s1' s2 ∧ (λ(x1, m). final-expr1 x1) s1' using bisim by auto
 next
  fix t' x m1 xx m2 t x1 x2 x1' ta1 x1'' m1' x2' ta2 x2'' m2'
  assume b: wbisim1 t' (x, m1) (xx, m2) and b': wbisim1 t (x1, m1) (x2, m2)
    and τred: ?τred t (x1, m1) (x1', m1)
    and red: mred1 P t (x1', m1) ta1 (x1'', m1')
    and ¬ τMOVE1 P (x1', m1) ta1 (x1'', m1')
    and τexec: ?τexec t (x2, m2) (x2', m2)
    and exec: ?exec t (x2', m2) ta2 (x2'', m2')
    and ¬ τMOVE2 (compP2 P) (x2', m2) ta2 (x2'', m2')
    and b2: wbisim1 t (x1'', m1') (x2'', m2')
  from red have hext m1 m1' by(auto simp add: split-beta intro: Red1-hext-incr)
  moreover from b2 have m1' = m2' by(cases x1'', cases x2'') simp
  moreover from b2 have hconf m2'
    by(cases x1'', cases x2'')(auto elim!: bisim1-list1.cases simp add: correct-state-def)
  moreover from b' exec have preallocated m2
    by(cases x1, cases x2)(auto elim!: bisim1-list1.cases simp add: correct-state-def)
```

moreover from *b′ τred red* **have** *tconf*: *compP2 P,m2 ⊢ t √t*
by(*cases x1*, *cases x2*)(*auto elim*!: *bisim1-list1.cases Red1.cases simp add*: *correct-state-def τmreds1-Val-Nil τmreds1-Throw-Nil*)
from *τexec* **have** *τexec′*: *τExec-1-dr* (*compP2 P*) *t* (*fst x2*, *m2*, *snd x2*) (*fst x2′*, *m2*, *snd x2′*)
unfolding *τExec-1-dr-conv-rtranclp* **by** *simp*
with *b′ tconf* **have** *compTP P ⊢ t*: (*fst x2′*, *m2*, *snd x2′*) *√*
using ‹*preallocated m2*›
apply(*cases x1*, *cases x2*)
apply(*erule τExec-1-dr-preserves-correct-state*[*OF wt-compTP-compP2*[*OF wf*]])
apply(*auto elim*!: *bisim1-list1.cases simp add*: *correct-state-def*)
done
ultimately show *wbisim1 t′* (*x*, *m1′*) (*xx*, *m2′*) **using** *b exec*
apply(*cases x*, *cases xx*)
apply(*auto elim*!: *bisim1-list1.cases intro*!: *bisim1-list1.intros simp add*: *split-beta intro*: *preallocated-hext*)
apply(*erule* (*2*) *correct-state-heap-change*[*OF wt-compTP-compP2*[*OF wf*]])
apply(*erule* (*1*) *bisim1-hext-mono*)
apply(*erule List.list-all2-mono*)
apply(*erule* (*1*) *bisim1-fr-hext-mono*)
done
next
fix *t x1 m1 x2 m2 x1′ ta1 x1″ m1′ x2′ ta2 x2″ m2′ w*
assume *b*: *wbisim1 t* (*x1*, *m1*) (*x2*, *m2*)
and *τred*: *?τred t* (*x1*, *m1*) (*x1′*, *m1*)
and *red*: *mred1 P t* (*x1′*, *m1*) *ta1* (*x1″*, *m1′*)
and ¬ *τMOVE1 P* (*x1′*, *m1*) *ta1* (*x1″*, *m1′*)
and *τexec*: *?τexec t* (*x2*, *m2*) (*x2′*, *m2*)
and *exec*: *?exec t* (*x2′*, *m2*) *ta2* (*x2″*, *m2′*)
and ¬ *τMOVE2* (*compP2 P*) (*x2′*, *m2*) *ta2* (*x2″*, *m2′*)
and *b′*: *wbisim1 t* (*x1″*, *m1′*) (*x2″*, *m2′*)
and *ta-bisim wbisim1 ta1 ta2*
and *Suspend*: *Suspend w ∈ set* ⦃*ta1*⦄$_w$ *Suspend w ∈ set* ⦃*ta2*⦄$_w$
from *red Suspend*
have *call1* (*fst* (*fst x1″*)) *≠ None*
by(*cases x1′*)(*cases x1″*, *auto dest*: *Red1-Suspend-is-call*)
moreover from *mexecd-Suspend-Invoke*[*OF exec Suspend*(*2*)]
obtain *xcp stk loc C M pc frs′ M′ n* **where** *x2″* = (*xcp*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs′*)
instrs-of (*compP2 P*) *C M* ! *pc* = *Invoke M′ n* **by** *blast*
ultimately show *bisim-wait1JVM* (*compP2 P*) *x1″ x2″*
by(*simp add*: *bisim-wait1JVM-def split-beta*)
next
fix *t x1 m1 x2 m2 ta1 x1′ m1′*
assume *wbisim1 t* (*x1*, *m1*) (*x2*, *m2*)
and *bisim-wait1JVM* (*compP2 P*) *x1 x2*
and *mred1 P t* (*x1*, *m1*) *ta1* (*x1′*, *m1′*)
and *wakeup*: *Notified ∈ set* ⦃*ta1*⦄$_w$ ∨ *WokenUp ∈ set* ⦃*ta1*⦄$_w$
moreover obtain *e1 xs1 exs1* **where** [*simp*]: *x1* = ((*e1*, *xs1*), *exs1*) **by**(*cases x1*) *auto*
moreover obtain *xcp frs* **where** [*simp*]: *x2* = (*xcp*, *frs*) **by**(*cases x2*)
moreover obtain *e1′ xs1′ exs1′* **where** [*simp*]: *x1′* = ((*e1′*, *xs1′*), *exs1′*) **by**(*cases x1′*) *auto*
ultimately have [*simp*]: *m1* = *m2*
and *bisim*: *bisim1-list1 t m2* (*e1*, *xs1*) *exs1 xcp frs*
and *red*: *True,P,t ⊢1* ⟨(*e1*, *xs1*)/*exs1*, *m2*⟩ −*ta1*→ ⟨(*e1′*, *xs1′*)/*exs1′*, *m1′*⟩
and *call*: *call1 e1 ≠ None*
case frs of [] ⇒ *False* | (*stk*, *loc*, *C*, *M*, *pc*) # *frs′* ⇒ ∃ *M′ n*. *instrs-of* (*compP2 P*) *C M*

! *pc* = *Invoke M′ n*
by(*auto simp add*: *bisim-wait1JVM-def split-def*)
from *red wakeup* **have** ¬ *τMove1 P m2* ((*e1*, *xs1*), *exs1*)
by(*auto elim*!: *Red1.cases dest*: *red1-τ-taD simp add*: *split-beta ta-upd-simps*)
from *exec-1-simulates-Red1-not-τ*[*OF wf red bisim this*] *call*
show ∃ *ta2 x2′ m2′*. *mexecd* (*compP2 P*) *t* (*x2*, *m2*) *ta2* (*x2′*, *m2′*) ∧ *wbisim1 t* (*x1′*, *m1′*) (*x2′*, *m2′*) ∧ *ta-bisim wbisim1 ta1 ta2*
by(*auto simp del*: *not-None-eq simp add*: *split-paired-Ex ta-bisim-def ta-upd-simps split*: *list.split-asm*)
next
fix *t x1 m1 x2 m2 ta2 x2′ m2′*
assume *wbisim1 t* (*x1*, *m1*) (*x2*, *m2*)
and *bisim-wait1JVM* (*compP2 P*) *x1 x2*
and *mexecd* (*compP2 P*) *t* (*x2*, *m2*) *ta2* (*x2′*, *m2′*)
and *wakeup*: *Notified* ∈ *set* ⦃*ta2*⦄*w* ∨ *WokenUp* ∈ *set* ⦃*ta2*⦄*w*
moreover obtain *e1 xs1 exs1* **where** [*simp*]: *x1* = ((*e1*, *xs1*), *exs1*) **by**(*cases x1*) *auto*
moreover obtain *xcp frs* **where** [*simp*]: *x2* = (*xcp*, *frs*) **by**(*cases x2*)
moreover obtain *xcp′ frs′* **where** [*simp*]: *x2′* = (*xcp′*, *frs′*) **by**(*cases x2′*)
ultimately have [*simp*]: *m1* = *m2*
and *bisim*: *bisim1-list1 t m2* (*e1*, *xs1*) *exs1 xcp frs*
and *exec*: *compP2 P*,*t* ⊢ *Normal* (*xcp*, *m2*, *frs*) −*ta2*−*jvmd*→ *Normal* (*xcp′*, *m2′*, *frs′*)
and *call*: *call1 e1* ≠ *None*
case frs of [] ⇒ *False* | (*stk*, *loc*, *C*, *M*, *pc*) # *frs′* ⇒ ∃ *M′ n*. *instrs-of* (*compP2 P*) *C M*
! *pc* = *Invoke M′ n*
by(*auto simp add*: *bisim-wait1JVM-def split-def*)
from *exec wakeup* **have** ¬ *τMove2* (*compP2 P*) (*xcp*, *m2*, *frs*)
by(*auto dest*: *τexec-1-taD simp add*: *split-beta ta-upd-simps*)
from *τRed1-simulates-exec-1-not-τ*[*OF wf exec bisim this*] *call*
show ∃ *ta1 x1′ m1′*. *mred1 P t* (*x1*, *m1*) *ta1* (*x1′*, *m1′*) ∧ *wbisim1 t* (*x1′*, *m1′*) (*x2′*, *m2′*) ∧ *ta-bisim wbisim1 ta1 ta2*
by(*auto simp del*: *not-None-eq simp add*: *split-paired-Ex ta-bisim-def ta-upd-simps split*: *list.split-asm*)
next
show (∃ *x*. *final-expr1 x*) ⟷ (∃ *x*. *JVM-final x*)
by(*auto simp add*: *split-paired-Ex final-iff*)
qed
qed

end

sublocale *J1-JVM-heap-conf-base* < *Red1-mexecd*:
FWbisimulation-base
final-expr1
mred1 P
JVM-final
mexecd (*compP2 P*)
convert-RA
wbisim1
bisim-wait1JVM (*compP2 P*)
.

context *J1-JVM-heap-conf* **begin**

lemma *bisim-J1-JVM-start*:
assumes *wf*: *wf-J1-prog P*
and *wf-start*: *wf-start-state P C M vs*

shows *Red1-execd.mbisim* (*J1-start-state P C M vs*) (*JVM-start-state* (*compP2 P*) *C M vs*)
proof −
from *wf-start* **obtain** *Ts T body D* **where** *start*: *start-heap-ok*
and *sees*: $P \vdash C$ *sees* M:*Ts*→*T*=⌊*body*⌋ *in D* **and** *conf*: *P*,*start-heap* ⊢ *vs* [:≤] *Ts* **by** *cases*

let *?e* = *blocks1 0* (*Class D*#*Ts*) *body*
let *?xs* = *Null* # *vs* @ *replicate* (*max-vars body*) *undefined-value*

from *sees-wf-mdecl*[*OF wf sees*] **obtain** *T*′
where *B*: $\mathcal{B}$ *body* (*Suc* (*length Ts*))
and *wt*: *P*,*Class D* # *Ts* ⊢*1 body* :: *T*′
and *da*: $\mathcal{D}$ *body* ⌊{..*length Ts*}⌋
and *sv*: *syncvars body*
by(*auto simp add*: *wf-mdecl-def*)

have *P*,*?e*,*start-heap* ⊢ (*?e*, *?xs*) ↔ ([], *?xs*, *0*, *None*) **by**(*rule bisim1-refl*)
moreover
from *wf* **have** *wf*′: *wf-jvm-prog*$_{compTP\ P}$ (*compP2 P*) **by**(*rule wt-compTP-compP2*)
from *sees-method-compP*[*OF sees*, *of* λ*C M Ts T*. *compMb2*]
have *sees*′: *compP2 P* ⊢ *C sees M*: *Ts*→*T* = ⌊*compMb2 body*⌋ *in D* **by**(*simp add*: *compP2-def*)
from *conf* **have** *compP2 P*,*start-heap* ⊢ *vs* [:≤] *Ts* **by**(*simp add*: *compP2-def heap-base.compP-confs*)
from *BV-correct-initial*[*OF wf*′ *start sees*′ *this*] *sees*′
have *compTP P* ⊢ *start-tid*:(*None*, *start-heap*, [([], *?xs*, *D*, *M*, *0*)]) √
by(*simp add*: *JVM-start-state*′*-def compP2-def compMb2-def*)
hence *bisim1-list1 start-tid start-heap* (*?e*, *?xs*) [] *None* [([], *?xs*, *D*, *M*, *0*)]
using *sees-method-idemp*[*OF sees*]
proof
show *P*,*?e*,*start-heap* ⊢ (*?e*, *?xs*) ↔ ([], *?xs*, *0*, *None*)
by(*rule bisim1-refl*)
show *max-vars ?e* ≤ *length ?xs* **using** *conf*
by(*auto simp add*: *blocks1-max-vars dest*: *list-all2-lengthD*)
qed *simp*
thus *?thesis*
using *sees sees*′ **unfolding** *start-state-def*
by −(*rule Red1-execd.mbisimI*, *auto split*: *if-split-asm intro*: *wset-thread-okI simp add*: *compP2-def compMb2-def*)
qed

lemmas *τred1-Val-simps* = *τred1r-Val τred1t-Val τreds1r-map-Val τreds1t-map-Val*

end

end

7.20 Indexing variables in variable lists

theory *ListIndex* **imports** *Main* **begin**

In order to support local variables and arbitrarily nested blocks, the local variables are arranged as an indexed list. The outermost local variable ("this") is the first element in the list, the most recently created local variable the last element. When descending into a block structure, a corresponding list *Vs* of variable names is maintained. To find the index of some variable *V*, we have to find the index of the *last* occurrence of *V* in *Vs*. This is what *index*

does:

primrec *index* :: *'a list* ⇒ *'a* ⇒ *nat*
where
  index [] *y* = *0*
| *index* (*x*#*xs*) *y* =
  (*if x=y then if x* ∈ *set xs then index xs y* + *1 else 0 else index xs y* + *1*)

definition *hidden* :: *'a list* ⇒ *nat* ⇒ *bool*
where *hidden xs i* ≡ *i* < *size xs* ∧ *xs*!*i* ∈ *set*(*drop* (*i*+*1*) *xs*)

7.20.1 *index*

lemma [*simp*]: *index* (*xs* @ [*x*]) *x* = *size xs*

lemma [*simp*]: (*index* (*xs* @ [*x*]) *y* = *size xs*) = (*x* = *y*)

lemma [*simp*]: *x* ∈ *set xs* ⟹ *xs* ! *index xs x* = *x*

lemma [*simp*]: *x* ∉ *set xs* ⟹ *index xs x* = *size xs*

lemma *index-size-conv*[*simp*]: (*index xs x* = *size xs*) = (*x* ∉ *set xs*)

lemma *size-index-conv*[*simp*]: (*size xs* = *index xs x*) = (*x* ∉ *set xs*)

lemma (*index xs x* < *size xs*) = (*x* ∈ *set xs*)

lemma [*simp*]: ⟦ *y* ∈ *set xs*; *x* ≠ *y* ⟧ ⟹ *index* (*xs* @ [*x*]) *y* = *index xs y*

lemma *index-less-size*[*simp*]: *x* ∈ *set xs* ⟹ *index xs x* < *size xs*
lemma *index-less-aux*: ⟦*x* ∈ *set xs*; *size xs* ≤ *n*⟧ ⟹ *index xs x* < *n*

lemma [*simp*]: *x* ∈ *set xs* ∨ *y* ∈ *set xs* ⟹ (*index xs x* = *index xs y*) = (*x* = *y*)

lemma *inj-on-index*: *inj-on* (*index xs*) (*set xs*)

lemma *index-drop*: ⋀*x i*. ⟦ *x* ∈ *set xs*; *index xs x* < *i* ⟧ ⟹ *x* ∉ *set*(*drop i xs*)

7.20.2 *hidden*

lemma *hidden-index*: *x* ∈ *set xs* ⟹ *hidden* (*xs* @ [*x*]) (*index xs x*)

lemma *hidden-inacc*: *hidden xs i* ⟹ *index xs x* ≠ *i*

lemma [*simp*]: *hidden xs i* ⟹ *hidden* (*xs*@[*x*]) *i*

lemma *fun-upds-apply*: ⋀*m ys*.
  (*m*(*xs*[↦]*ys*)) *x* =
  (*let xs'* = *take* (*size ys*) *xs*
   in if x ∈ *set xs' then Some*(*ys* ! *index xs' x*) *else m x*)

lemma *map-upds-apply-eq-Some*:
  ((*m*(*xs*[↦]*ys*)) *x* = *Some y*) =
  (*let xs'* = *take* (*size ys*) *xs*
   in if x ∈ *set xs' then ys* ! *index xs' x* = *y else m x* = *Some y*)

lemma *map-upds-upd-conv-index*:
⟦*x* ∈ *set xs*; *size xs* ≤ *size ys* ⟧
⟹ *m*(*xs*[↦]*ys*, *x*↦*y*) = *m*(*xs*[↦]*ys*[*index xs x* := *y*])
lemma *image-index*:
A ⊆ *set*(*xs*@[*x*]) ⟹ *index* (*xs* @ [*x*]) ‘ *A* =
(*if x* ∈ *A then insert* (*size xs*) (*index xs* ‘ (*A*−{*x*})) *else index xs* ‘ *A*)
lemma *index-le-lengthD*: *index xs x* < *length xs* ⟹ *x* ∈ *set xs*
by(*erule contrapos-pp*)(*simp*)

lemma *not-hidden-index-nth*: ⟦ *i* < *length Vs*; ¬ *hidden Vs i* ⟧ ⟹ *index Vs* (*Vs* ! *i*) = *i*
by(*induct Vs arbitrary*: *i*)(*auto split*: *if-split-asm nat.split-asm simp add*: *nth-Cons hidden-def*)

lemma *hidden-snoc-nth*:
assumes *len*: *i* < *length Vs*
shows *hidden* (*Vs* @ [*Vs* ! *i*]) *i*
proof(*cases hidden Vs i*)
case *True* **thus** *?thesis* **by** *simp*
next
case *False*
with *len* **have** *index Vs* (*Vs* ! *i*) = *i* **by**(*rule not-hidden-index-nth*)
moreover from *len* **have** *hidden* (*Vs* @ [*Vs* ! *i*]) (*index Vs* (*Vs* ! *i*))
by(*auto intro*: *hidden-index*)
ultimately show *?thesis* **by** *simp*
qed

lemma *map-upds-Some-eq-nth-index*:
assumes [*Vs* [↦] *vs*] *V* = *Some v length Vs* ≤ *length vs*
shows *vs* ! *index Vs V* = *v*
proof −
from ‹[*Vs* [↦] *vs*] *V* = *Some v*› **have** *V* ∈ *set Vs*
by −(*rule classical*, *auto*)
with ‹[*Vs* [↦] *vs*] *V* = *Some v*› ‹*length Vs* ≤ *length vs*› **show** *?thesis*
proof(*induct Vs arbitrary*: *vs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs ys*)
note *IH* = ‹⋀*vs*. ⟦ [*xs* [↦] *vs*] *V* = *Some v*; *length xs* ≤ *length vs*; *V* ∈ *set xs* ⟧ ⟹ *vs* ! *index xs*
V = *v*›
from ‹[*x* # *xs* [↦] *ys*] *V* = *Some v*› **obtain** *y Ys* **where** *ys* = *y* # *Ys* **by**(*cases ys*, *auto*)
with ‹*length* (*x* # *xs*) ≤ *length ys*› **have** *length xs* ≤ *length Ys* **by** *simp*
show *?case*
proof(*cases V* ∈ *set xs*)
case *True*
with ‹[*x* # *xs* [↦] *ys*] *V* = *Some v*› ‹*length xs* ≤ *length Ys*› ‹*ys* = *y* # *Ys*›
have [*xs* [↦] *Ys*] *V* = *Some v*
apply(*auto simp add*: *map-upds-def map-of-eq-None-iff set-zip image-Collect split*: *if-split-asm*)
apply(*clarsimp simp add*: *in-set-conv-decomp*)
apply(*hypsubst-thin*)
apply(*erule-tac x*=*length ys* **in** *allE*)
by(*simp*)
with *IH*[*OF this* ‹*length xs* ≤ *length Ys*› *True*] ‹*ys* = *y* # *Ys*› *True*
show *?thesis* **by**(*simp*)
next

```
    case False with ‹V ∈ set (x # xs)› have x = V by auto
    with False ‹[x # xs [↦] ys] V = Some v› ‹ys = y # Ys› have y = v
     by(auto)
    with False ‹x = V› ‹ys = y # Ys›
    show ?thesis by(simp)
  qed
 qed
qed

end
```

7.21 Compilation Stage 1

```
theory Compiler1 imports
  PCompiler
  J1State
  ListIndex
begin

definition fresh-var :: vname list ⇒ vname
  where fresh-var Vs = sum-list (STR ''V'' # Vs)

lemma fresh-var-fresh: fresh-var Vs ∉ set Vs
proof −
  have V ∈ set Vs ⟹ length (String.explode V) < length (String.explode (fresh-var Vs)) for V
    by (induction Vs) (auto simp add: fresh-var-def Literal.rep-eq)
  then show ?thesis
    by auto
qed
```

Replacing variable names by indices.

```
function compE1  :: vname list ⇒ 'addr expr      ⇒ 'addr expr1
  and compEs1 :: vname list ⇒ 'addr expr list ⇒ 'addr expr1 list
where
  compE1 Vs (new C) = new C
| compE1 Vs (newA T⌊e⌉) = newA T⌊compE1 Vs e⌉
| compE1 Vs (Cast T e) = Cast T (compE1 Vs e)
| compE1 Vs (e instanceof T) = (compE1 Vs e) instanceof T
| compE1 Vs (Val v) = Val v
| compE1 Vs (Var V) = Var(index Vs V)
| compE1 Vs (e«bop»e') = (compE1 Vs e)«bop»(compE1 Vs e')
| compE1 Vs (V:=e) = (index Vs V):= (compE1 Vs e)
| compE1 Vs (a⌊i⌉) = (compE1 Vs a)⌊compE1 Vs i⌉
| compE1 Vs (a⌊i⌉:=e) = (compE1 Vs a)⌊compE1 Vs i⌉:=compE1 Vs e
| compE1 Vs (a·length) = compE1 Vs a·length
| compE1 Vs (e·F{D}) = compE1 Vs e·F{D}
| compE1 Vs (e·F{D}:=e') = compE1 Vs e·F{D}:=compE1 Vs e'
| compE1 Vs (e·compareAndSwap(D·F, e', e'')) = compE1 Vs e·compareAndSwap(D·F, compE1 Vs
e', compE1 Vs e'')
| compE1 Vs (e·M(es)) = (compE1 Vs e)·M(compEs1 Vs es)
| compE1 Vs {V:T=vo; e} = {(size Vs):T=vo; compE1 (Vs@[V]) e}
| compE1 Vs (sync_U (o') e) = sync_(length Vs) (compE1 Vs o') (compE1 (Vs@[fresh-var Vs]) e)
| compE1 Vs (insync_U (a) e) = insync_(length Vs) (a) (compE1 (Vs@[fresh-var Vs]) e)
```

| *compE1 Vs* (*e1*;;*e2*) = (*compE1 Vs e1*);;(*compE1 Vs e2*)
| *compE1 Vs* (*if* (*b*) *e1 else e2*) = (*if* (*compE1 Vs b*) (*compE1 Vs e1*) *else* (*compE1 Vs e2*))
| *compE1 Vs* (*while* (*b*) *e*) = (*while* (*compE1 Vs b*) (*compE1 Vs e*))
| *compE1 Vs* (*throw e*) = *throw* (*compE1 Vs e*)
| *compE1 Vs* (*try e1 catch*(*C V*) *e2*) = *try*(*compE1 Vs e1*) *catch*(*C* (*size Vs*)) (*compE1* (*Vs*@[*V*]) *e2*)

| *compEs1 Vs* [] = []
| *compEs1 Vs* (*e*#*es*) = *compE1 Vs e* # *compEs1 Vs es*
by *pat-completeness auto*
termination
apply(*relation case-sum* (λ*p*. *size* (*snd p*)) (λ*p*. *size-list size* (*snd p*)) <∗*mlex*∗> {})
apply(*rule wf-mlex*[*OF wf-empty*])
apply(*rule mlex-less*, *simp*)+
done

lemmas *compE1-compEs1-induct* =
compE1-compEs1.induct[*case-names New NewArray Cast InstanceOf Val Var BinOp LAss AAcc AAss ALen FAcc FAss CAS Call Block Synchronized InSynchronized Seq Cond While throw TryCatch Nil Cons*]

lemma *compEs1-conv-map* [*simp*]: *compEs1 Vs es* = *map* (*compE1 Vs*) *es*
by(*induct es*) *simp-all*

lemmas *compEs1-map-Val* = *compEs1-conv-map*

lemma *compE1-eq-Val* [*simp*]: *compE1 Vs e* = *Val v* ⟷ *e* = *Val v*
apply(*cases e*, *auto*)
done

lemma *Val-eq-compE1* [*simp*]: *Val v* = *compE1 Vs e* ⟷ *e* = *Val v*
apply(*cases e*, *auto*)
done

lemma *compEs1-eq-map-Val* [*simp*]: *compEs1 Vs es* = *map Val vs* ⟷ *es* = *map Val vs*
apply(*induct es arbitrary*: *vs*)
apply(*auto*, *blast*)
done

lemma *compE1-eq-Var* [*simp*]: *compE1 Vs e* = *Var V* ⟷ (∃ *V'*. *e* = *Var V'* ∧ *V* = *index Vs V'*)
by(*cases e*, *auto*)

lemma *compE1-eq-Call* [*simp*]:
compE1 Vs e = *obj*•*M*(*params*) ⟷ (∃ *obj' params'*. *e* = *obj'*•*M*(*params'*) ∧ *compE1 Vs obj'* = *obj* ∧ *compEs1 Vs params'* = *params*)
by(*cases e*, *auto*)

lemma *length-compEs2* [*simp*]:
length (*compEs1 Vs es*) = *length es*
by(*simp add*: *compEs1-conv-map*)

lemma fixes *e* :: *'addr expr* **and** *es* :: *'addr expr list*
shows *expr-locks-compE1* [*simp*]: *expr-locks* (*compE1 Vs e*) = *expr-locks e*
and *expr-lockss-compEs1* [*simp*]: *expr-lockss* (*compEs1 Vs es*) = *expr-lockss es*
by(*induct Vs e* **and** *Vs es rule*: *compE1-compEs1.induct*)(*auto intro*: *ext*)

```
lemma fixes e :: 'addr expr and es :: 'addr expr list
  shows contains-insync-compE1 [simp]: contains-insync (compE1 Vs e) = contains-insync e
  and contains-insyncs-compEs1 [simp]: contains-insyncs (compEs1 Vs es) = contains-insyncs es
by(induct Vs e and Vs es rule: compE1-compEs1.induct)simp-all

lemma fixes e :: 'addr expr and es :: 'addr expr list
  shows max-vars-compE1: max-vars (compE1 Vs e) = max-vars e
  and max-varss-compEs1: max-varss (compEs1 Vs es) = max-varss es
apply(induct Vs e and Vs es rule: compE1-compEs1.induct)
apply(auto)
done

lemma fixes e :: 'addr expr and es :: 'addr expr list
  shows B: size Vs = n ⟹ B (compE1 Vs e) n
  and Bs: size Vs = n ⟹ Bs (compEs1 Vs es) n
apply(induct Vs e and Vs es arbitrary: n and n rule: compE1-compEs1.induct)
apply auto
done

lemma fixes e :: 'addr expr and es :: 'addr expr list
  shows fv-compE1: fv e ⊆ set Vs ⟹ fv (compE1 Vs e) = (index Vs) ‘ (fv e)
  and fvs-compEs1: fvs es ⊆ set Vs ⟹ fvs (compEs1 Vs es) = (index Vs) ‘ (fvs es)
proof(induct Vs e and Vs es rule: compE1-compEs1-induct)
  case (Block Vs V ty vo exp)
  have IH: fv exp ⊆ set (Vs @ [V]) ⟹ fv (compE1 (Vs @ [V]) exp) = index (Vs @ [V]) ‘ fv exp by
fact
  from ‹fv {V:ty=vo; exp} ⊆ set Vs› have fv': fv exp ⊆ set (Vs @ [V]) by auto
  from IH[OF this] have IH': fv (compE1 (Vs @ [V]) exp) = index (Vs @ [V]) ‘ fv exp .
  have fv (compE1 (Vs @ [V]) exp) − {length Vs} = index Vs ‘ (fv exp − {V})
  proof(rule equalityI[OF subsetI subsetI])
    fix x
    assume x: x ∈ fv (compE1 (Vs @ [V]) exp) − {length Vs}
    hence x ≠ length Vs by simp
    from x IH' have x ∈ index (Vs @ [V]) ‘ fv exp by simp
    thus x ∈ index Vs ‘ (fv exp − {V})
    proof(rule imageE)
      fix y
      assume [simp]: x = index (Vs @ [V]) y
        and y: y ∈ fv exp
      have y ≠ V
      proof
        assume [simp]: y = V
        hence x = length Vs by simp
        with ‹x ≠ length Vs› show False by contradiction
      qed
      moreover with fv' y have y ∈ set Vs by auto
      ultimately have index (Vs @ [V]) y = index Vs y by(simp)
      thus ?thesis using y ‹y ≠ V› by auto
    qed
  next
    fix x
    assume x: x ∈ index Vs ‘ (fv exp − {V})
    thus x ∈ fv (compE1 (Vs @ [V]) exp) − {length Vs}
```

```
    proof(rule imageE)
      fix y
      assume [simp]: x = index Vs y
        and y: y ∈ fv exp − {V}
      with fv′ have y ∈ set Vs y ≠ V by auto
      hence index Vs y = index (Vs @ [V]) y by simp
      with y have x ∈ index (Vs @ [V]) ‘ fv exp by auto
      thus ?thesis using IH′ ‹y ∈ set Vs› by simp
    qed
  qed
  thus ?case by simp
next
  case (Synchronized Vs V exp1 exp2)
  have IH1: fv exp1 ⊆ set Vs ⟹ fv (compE1 Vs exp1) = index Vs ‘ fv exp1
    and IH2: fv exp2 ⊆ set (Vs @ [fresh-var Vs]) ⟹ fv (compE1 (Vs @ [fresh-var Vs]) exp2) = index
(Vs @ [fresh-var Vs]) ‘ fv exp2
    by fact+
  from ‹fv (sync V (exp1) exp2) ⊆ set Vs› have fv1: fv exp1 ⊆ set Vs
    and fv2: fv exp2 ⊆ set Vs by auto
  from fv2 have fv2′: fv exp2 ⊆ set (Vs @ [fresh-var Vs]) by auto
  have index (Vs @ [fresh-var Vs]) ‘ fv exp2 = index Vs ‘ fv exp2
  proof(rule equalityI[OF subsetI subsetI])
    fix x
    assume x: x ∈ index (Vs @ [fresh-var Vs]) ‘ fv exp2
    thus x ∈ index Vs ‘ fv exp2
    proof(rule imageE)
      fix y
      assume [simp]: x = index (Vs @ [fresh-var Vs]) y
        and y: y ∈ fv exp2
      from y fv2 have y ∈ set Vs by auto
      moreover hence y ≠ (fresh-var Vs) by(auto simp add: fresh-var-fresh)
      ultimately show ?thesis using y by(auto)
    qed
  next
    fix x
    assume x: x ∈ index Vs ‘ fv exp2
    thus x ∈ index (Vs @ [fresh-var Vs]) ‘ fv exp2
    proof(rule imageE)
      fix y
      assume [simp]: x = index Vs y
        and y: y ∈ fv exp2
      from y fv2 have y ∈ set Vs by auto
      moreover hence y ≠ (fresh-var Vs) by(auto simp add: fresh-var-fresh)
      ultimately have index Vs y = index (Vs @ [fresh-var Vs]) y by simp
      thus ?thesis using y by(auto)
    qed
  qed
  with IH1[OF fv1] IH2[OF fv2′] show ?case by(auto)
next
  case (InSynchronized Vs V a exp)
  have IH: fv exp ⊆ set (Vs @ [fresh-var Vs]) ⟹ fv (compE1 (Vs @ [fresh-var Vs]) exp) = index (Vs
@ [fresh-var Vs]) ‘ fv exp
    by fact
  from ‹fv (insync V (a) exp) ⊆ set Vs› have fv: fv exp ⊆ set Vs by simp
```

```
  hence fv': fv exp ⊆ set (Vs @ [fresh-var Vs]) by auto
  have index (Vs @ [fresh-var Vs]) ‘ fv exp = index Vs ‘ fv exp
  proof(rule equalityI[OF subsetI subsetI])
    fix x
    assume x ∈ index (Vs @ [fresh-var Vs]) ‘ fv exp
    thus x ∈ index Vs ‘ fv exp
    proof(rule imageE)
      fix y
      assume [simp]: x = index (Vs @ [fresh-var Vs]) y
        and y: y ∈ fv exp
      from y fv have y ∈ set Vs by auto
      moreover hence y ≠ (fresh-var Vs) by(auto simp add: fresh-var-fresh)
      ultimately have index (Vs @ [fresh-var Vs]) y = index Vs y by simp
      thus ?thesis using y by simp
    qed
  next
    fix x
    assume x ∈ index Vs ‘ fv exp
    thus x ∈ index (Vs @ [fresh-var Vs]) ‘ fv exp
    proof(rule imageE)
      fix y
      assume [simp]: x = index Vs y
        and y: y ∈ fv exp
      from y fv have y ∈ set Vs by auto
      moreover hence y ≠ (fresh-var Vs) by(auto simp add: fresh-var-fresh)
      ultimately have index Vs y = index (Vs @ [fresh-var Vs]) y by simp
      thus ?thesis using y by auto
    qed
  qed
  with IH[OF fv'] show ?case by simp
next
  case (TryCatch Vs exp1 C V exp2)
  have IH1: fv exp1 ⊆ set Vs ⟹ fv (compE1 Vs exp1) = index Vs ‘ fv exp1
    and IH2: fv exp2 ⊆ set (Vs @ [V]) ⟹ fv (compE1 (Vs @ [V]) exp2) = index (Vs @ [V]) ‘ fv
exp2
    by fact+
  from ‹fv (try exp1 catch(C V) exp2) ⊆ set Vs› have fv1: fv exp1 ⊆ set Vs
    and fv2: fv exp2 ⊆ set (Vs @ [V]) by auto
  have index (Vs @ [V]) ‘ fv exp2 − {length Vs} = index Vs ‘ (fv exp2 − {V})
  proof(rule equalityI[OF subsetI subsetI])
    fix x
    assume x: x ∈ index (Vs @ [V]) ‘ fv exp2 − {length Vs}
    hence x ≠ length Vs by simp
    from x have x ∈ index (Vs @ [V]) ‘ fv exp2 by auto
    thus x ∈ index Vs ‘ (fv exp2 − {V})
    proof(rule imageE)
      fix y
      assume [simp]: x = index (Vs @ [V]) y
        and y: y ∈ fv exp2
      have y ≠ V
      proof
        assume [simp]: y = V
        hence x = length Vs by simp
        with ‹x ≠ length Vs› show False by contradiction
```

```
      qed
      moreover with fv2 y have y ∈ set Vs by auto
      ultimately have index (Vs @ [V]) y = index Vs y by(simp)
      thus ?thesis using y ‹y ≠ V› by auto
    qed
  next
    fix x
    assume x: x ∈ index Vs ‘ (fv exp2 − {V})
    thus x ∈ index (Vs @ [V]) ‘ fv exp2 − {length Vs}
    proof(rule imageE)
      fix y
      assume [simp]: x = index Vs y
        and y: y ∈ fv exp2 − {V}
      with fv2 have y ∈ set Vs y ≠ V by auto
      hence index Vs y = index (Vs @ [V]) y by simp
      with y have x ∈ index (Vs @ [V]) ‘ fv exp2 by auto
      thus ?thesis using ‹y ∈ set Vs› by simp
    qed
  qed
  with IH1[OF fv1] IH2[OF fv2] show ?case by auto
qed(auto)

lemma fixes e :: 'addr expr and es :: 'addr expr list
  shows syncvars-compE1: fv e ⊆ set Vs ⟹ syncvars (compE1 Vs e)
  and syncvarss-compEs1: fvs es ⊆ set Vs ⟹ syncvarss (compEs1 Vs es)
proof(induct Vs e and Vs es rule: compE1-compEs1-induct)
  case (Block Vs V ty vo exp)
  from ‹fv {V:ty=vo; exp} ⊆ set Vs› have fv exp ⊆ set (Vs @ [V]) by auto
   from ‹fv exp ⊆ set (Vs @ [V]) ⟹ syncvars (compE1 (Vs @ [V]) exp)›[OF this] show ?case
by(simp)
next
  case (Synchronized Vs V exp1 exp2)
  note IH1 = ‹fv exp1 ⊆ set Vs ⟹ syncvars (compE1 Vs exp1)›
  note IH2 = ‹fv exp2 ⊆ set (Vs @ [fresh-var Vs]) ⟹ syncvars (compE1 (Vs @ [fresh-var Vs]) exp2)›
  from ‹fv (sync_V (exp1) exp2) ⊆ set Vs› have fv1: fv exp1 ⊆ set Vs
    and fv2: fv exp2 ⊆ set Vs and fv2': fv exp2 ⊆ set (Vs @ [fresh-var Vs]) by auto
  have length Vs ∉ index (Vs @ [fresh-var Vs]) ‘ fv exp2
  proof
    assume length Vs ∈ index (Vs @ [fresh-var Vs]) ‘ fv exp2
    thus False
    proof(rule imageE)
      fix x
      assume x: length Vs = index (Vs @ [fresh-var Vs]) x
        and x': x ∈ fv exp2
      from x' fv2 have x ∈ set Vs x ≠ (fresh-var Vs) by(auto simp add: fresh-var-fresh)
      with x show ?thesis by(simp)
    qed
  qed
  with IH1[OF fv1] IH2[OF fv2'] fv2' show ?case by(simp add: fv-compE1)
next
  case (InSynchronized Vs V a exp)
  note IH = ‹fv exp ⊆ set (Vs @ [fresh-var Vs]) ⟹ syncvars (compE1 (Vs @ [fresh-var Vs]) exp)›
  from ‹fv (insync_V (a) exp) ⊆ set Vs› have fv: fv exp ⊆ set Vs
    and fv': fv exp ⊆ set (Vs @ [fresh-var Vs]) by auto
```

```
  have length Vs ∉ index (Vs @ [fresh-var Vs]) ‘ fv exp
  proof
    assume length Vs ∈ index (Vs @ [fresh-var Vs]) ‘ fv exp
    thus False
    proof(rule imageE)
      fix x
      assume x: length Vs = index (Vs @ [fresh-var Vs]) x
        and x': x ∈ fv exp
      from x' fv have x ∈ set Vs x ≠ (fresh-var Vs) by(auto simp add: fresh-var-fresh)
      with x show ?thesis by(simp)
    qed
  qed
  with IH[OF fv'] fv' show ?case by(simp add: fv-compE1)
next
  case (TryCatch Vs exp1 C V exp2)
  note IH1 = ‹fv exp1 ⊆ set Vs ⟹ syncvars (compE1 Vs exp1)›
  note IH2 = ‹fv exp2 ⊆ set (Vs @ [V]) ⟹ syncvars (compE1 (Vs @ [V]) exp2)›
  from ‹fv (try exp1 catch(C V) exp2) ⊆ set Vs› have fv1: fv exp1 ⊆ set Vs
    and fv2: fv exp2 ⊆ set (Vs @ [V]) by auto
  from IH1[OF fv1] IH2[OF fv2] show ?case by auto
qed auto

lemma (in heap-base) synthesized-call-compP [simp]:
  synthesized-call (compP f P) h aMvs = synthesized-call P h aMvs
by(simp add: synthesized-call-def)


primrec fin1 :: 'addr expr ⇒ 'addr expr1
where
  fin1 (Val v) = Val v
| fin1 (throw e) = throw (fin1 e)

lemma comp-final: final e ⟹ compE1 Vs e = fin1 e
by(erule finalE, simp-all)

lemma fixes e :: 'addr expr and es :: 'addr expr list
  shows [simp]: max-vars (compE1 Vs e) = max-vars e
  and max-varss (compEs1 Vs es) = max-varss es
by (induct Vs e and Vs es rule: compE1-compEs1-induct)(simp-all)
```

Compiling programs:

```
definition compP1 :: 'addr J-prog ⇒ 'addr J1-prog
where
  compP1 ≡ compP (λC M Ts T (pns,body). compE1 (this#pns) body)

declare compP1-def[simp]

end
```

7.22 The bisimulation relation betwenn source and intermediate language

```
theory J0J1Bisim imports
```

J1
J1WellForm
Compiler1
../J/JWellForm
J0
begin

7.22.1 Correctness of program compilation

primrec *unmod* :: *'addr expr1* ⇒ *nat* ⇒ *bool*
and *unmods* :: *'addr expr1 list* ⇒ *nat* ⇒ *bool*
where
unmod (*new C*) *i* = *True*
| *unmod* (*newA T*⌊*e*⌋) *i* = *unmod e i*
| *unmod* (*Cast C e*) *i* = *unmod e i*
| *unmod* (*e instanceof T*) *i* = *unmod e i*
| *unmod* (*Val v*) *i* = *True*
| *unmod* (*e1* «*bop*» *e2*) *i* = (*unmod e1 i* ∧ *unmod e2 i*)
| *unmod* (*Var i*) *j* = *True*
| *unmod* (*i*:=*e*) *j* = (*i* ≠ *j* ∧ *unmod e j*)
| *unmod* (*a*⌊*i*⌋) *j* = (*unmod a j* ∧ *unmod i j*)
| *unmod* (*a*⌊*i*⌋:=*e*) *j* = (*unmod a j* ∧ *unmod i j* ∧ *unmod e j*)
| *unmod* (*a*·*length*) *j* = *unmod a j*
| *unmod* (*e*·*F*{*D*}) *i* = *unmod e i*
| *unmod* (*e1*·*F*{*D*}:=*e2*) *i* = (*unmod e1 i* ∧ *unmod e2 i*)
| *unmod* (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *i* = (*unmod e1 i* ∧ *unmod e2 i* ∧ *unmod e3 i*)
| *unmod* (*e*·*M*(*es*)) *i* = (*unmod e i* ∧ *unmods es i*)
| *unmod* {*j*:*T*=*vo*; *e*} *i* = ((*i* = *j* ⟶ *vo* = *None*) ∧ *unmod e i*)
| *unmod* (*sync*$_V$ (*o'*) *e*) *i* = (*unmod o' i* ∧ *unmod e i* ∧ *i* ≠ *V*)
| *unmod* (*insync*$_V$ (*a*) *e*) *i* = *unmod e i*
| *unmod* (*e1*;;*e2*) *i* = (*unmod e1 i* ∧ *unmod e2 i*)
| *unmod* (*if* (*e*) e_1 *else* e_2) *i* = (*unmod e i* ∧ *unmod* e_1 *i* ∧ *unmod* e_2 *i*)
| *unmod* (*while* (*e*) *c*) *i* = (*unmod e i* ∧ *unmod c i*)
| *unmod* (*throw e*) *i* = *unmod e i*
| *unmod* (*try* e_1 *catch*(*C i*) e_2) *j* = (*unmod* e_1 *j* ∧ (*if i*=*j then False else unmod* e_2 *j*))

| *unmods* ([]) *i* = *True*
| *unmods* (*e*#*es*) *i* = (*unmod e i* ∧ *unmods es i*)

lemma *unmods-map-Val* [*simp*]: *unmods* (*map Val vs*) *V*
by(*induct vs*) *simp-all*

lemma fixes *e* :: *'addr expr* **and** *es* :: *'addr expr list*
shows *hidden-unmod*: *hidden Vs i* ⟹ *unmod* (*compE1 Vs e*) *i*
and *hidden-unmods*: *hidden Vs i* ⟹ *unmods* (*compEs1 Vs es*) *i*
apply(*induct Vs e* **and** *Vs es rule*: *compE1-compEs1-induct*)
apply (*simp-all add*:*hidden-inacc*)
apply(*auto simp add*:*hidden-def*)
done

lemma *unmod-extRet2J* [*simp*]: *unmod e i* ⟹ *unmod* (*extRet2J e va*) *i*
by(*cases va*) *simp-all*

lemma *max-dest*: (*n* :: *nat*) + *max a b* ≤ *c* ⟹ *n* + *a* ≤ *c* ∧ *n* + *b* ≤ *c*

apply(*auto simp add*: *max-def split*: *if-split-asm*)
done

declare *max-dest* [*dest*!]

lemma fixes *e* :: *'addr expr* **and** *es* :: *'addr expr list*
shows *fv-unmod-compE1*: ⟦ *i* < *length Vs*; *Vs* ! *i* ∉ *fv e* ⟧ ⟹ *unmod* (*compE1 Vs e*) *i*
and *fvs-unmods-compEs1*: ⟦ *i* < *length Vs*; *Vs* ! *i* ∉ *fvs es* ⟧ ⟹ *unmods* (*compEs1 Vs es*) *i*
proof(*induct Vs e* **and** *Vs es rule*: *compE1-compEs1-induct*)
case (*Block Vs V ty vo exp*)
note *IH* = ‹⟦*i* < *length* (*Vs* @ [*V*]); (*Vs* @ [*V*]) ! *i* ∉ *fv exp* ⟧ ⟹ *unmod* (*compE1* (*Vs* @ [*V*]) *exp*) *i›*
note *len* = ‹*i* < *length Vs›*
hence *i*: *i* < *length* (*Vs* @ [*V*]) **by** *simp*
show *?case*
proof(*cases Vs* ! *i* = *V*)
case *True*
from *len* **have** *hidden* (*Vs* @ [*Vs* ! *i*]) *i* **by**(*rule hidden-snoc-nth*)
with *len True* **show** *?thesis* **by**(*auto intro*: *hidden-unmod*)
next
case *False*
with ‹*Vs* ! *i* ∉ *fv* {*V*:*ty*=*vo*; *exp*}› *len* **have** (*Vs* @ [*V*]) ! *i* ∉ *fv exp*
by(*auto simp add*: *nth-append*)
from *IH*[*OF i this*] *len* **show** *?thesis* **by**(*auto*)
qed
next
case (*TryCatch Vs e1 C V e2*)
note *IH1* = ‹⟦*i* < *length Vs*; *Vs* ! *i* ∉ *fv e1* ⟧ ⟹ *unmod* (*compE1 Vs e1*) *i›*
note *IH2* = ‹⟦*i* < *length* (*Vs* @ [*V*]); (*Vs* @ [*V*]) ! *i* ∉ *fv e2* ⟧ ⟹ *unmod* (*compE1* (*Vs* @ [*V*]) *e2*) *i›*
note *len* = ‹*i* < *length Vs›*
hence *i*: *i* < *length* (*Vs* @ [*V*]) **by** *simp*
have *unmod* (*compE1* (*Vs* @ [*V*]) *e2*) *i*
proof(*cases Vs* ! *i* = *V*)
case *True*
from *len* **have** *hidden* (*Vs* @ [*Vs* ! *i*]) *i* **by**(*rule hidden-snoc-nth*)
with *len True* **show** *?thesis* **by**(*auto intro*: *hidden-unmod*)
next
case *False*
with ‹*Vs* ! *i* ∉ *fv* (*try e1 catch*(*C V*) *e2*)› *len* **have** (*Vs* @ [*V*]) ! *i* ∉ *fv e2*
by(*auto simp add*: *nth-append*)
from *IH2*[*OF i this*] *len* **show** *?thesis* **by**(*auto*)
qed
with *IH1*[*OF len*] ‹*Vs* ! *i* ∉ *fv* (*try e1 catch*(*C V*) *e2*)› *len* **show** *?case* **by**(*auto*)
qed(*auto dest*: *index-le-lengthD simp add*: *nth-append*)

lemma *hidden-lengthD*: *hidden Vs i* ⟹ *i* < *length Vs*
by(*simp add*: *hidden-def*)

lemma fixes *e* :: *'addr expr1* **and** *es* :: *'addr expr1 list*
shows *fv-B-unmod*: ⟦ *V* ∉ *fv e*; *B e n*; *V* < *n* ⟧ ⟹ *unmod e V*
and *fvs-Bs-unmods*: ⟦ *V* ∉ *fvs es*; *Bs es n*; *V* < *n* ⟧ ⟹ *unmods es V*
by(*induct e* **and** *es arbitrary*: *n* **and** *n rule*: *unmod.induct unmods.induct*) *auto*

lemma assumes *fin*: *final e′*
shows *unmod-inline-call*: *unmod* (*inline-call e′ e*) *V* ⟷ *unmod e V*
and *unmods-inline-calls*: *unmods* (*inline-calls e′ es*) *V* ⟷ *unmods es V*
apply(*induct e* **and** *es rule*: *unmod.induct unmods.induct*)
apply(*insert fin*)
apply(*auto simp add*: *is-vals-conv*)
done

7.22.2 The delay bisimulation relation

Delay bisimulation for expressions

inductive *bisim* :: *vname list* ⇒ *'addr expr* ⇒ *'addr expr1* ⇒ *'addr val list* ⇒ *bool*
and *bisims* :: *vname list* ⇒ *'addr expr list* ⇒ *'addr expr1 list* ⇒ *'addr val list* ⇒ *bool*
where
bisimNew: *bisim Vs* (*new C*) (*new C*) *xs*
| *bisimNewArray*: *bisim Vs e e′ xs* ⟹ *bisim Vs* (*newA T*⌊*e*⌉) (*newA T*⌊*e′*⌉) *xs*
| *bisimCast*: *bisim Vs e e′ xs* ⟹ *bisim Vs* (*Cast T e*) (*Cast T e′*) *xs*
| *bisimInstanceOf*: *bisim Vs e e′ xs* ⟹ *bisim Vs* (*e instanceof T*) (*e′ instanceof T*) *xs*
| *bisimVal*: *bisim Vs* (*Val v*) (*Val v*) *xs*
| *bisimBinOp1*:
⟦ *bisim Vs e e′ xs*; ¬ *is-val e*; ¬ *contains-insync e″* ⟧ ⟹ *bisim Vs* (*e* «*bop*» *e″*) (*e′* «*bop*» *compE1 Vs e″*) *xs*
| *bisimBinOp2*: *bisim Vs e e′ xs* ⟹ *bisim Vs* (*Val v* «*bop*» *e*) (*Val v* «*bop*» *e′*) *xs*
| *bisimVar*: *bisim Vs* (*Var V*) (*Var* (*index Vs V*)) *xs*
| *bisimLAss*: *bisim Vs e e′ xs* ⟹ *bisim Vs* (*V*:=*e*) (*index Vs V*:=*e′*) *xs*
| *bisimAAcc1*: ⟦ *bisim Vs a a′ xs*; ¬ *is-val a*; ¬ *contains-insync i* ⟧ ⟹ *bisim Vs* (*a*⌊*i*⌉) (*a′*⌊*compE1 Vs i*⌉) *xs*
| *bisimAAcc2*: *bisim Vs i i′ xs* ⟹ *bisim Vs* (*Val v*⌊*i*⌉) (*Val v*⌊*i′*⌉) *xs*
| *bisimAAss1*:
⟦ *bisim Vs a a′ xs*; ¬ *is-val a*; ¬ *contains-insync i*; ¬ *contains-insync e* ⟧
⟹ *bisim Vs* (*a*⌊*i*⌉:=*e*) (*a′*⌊*compE1 Vs i*⌉:=*compE1 Vs e*) *xs*
| *bisimAAss2*: ⟦ *bisim Vs i i′ xs*; ¬ *is-val i*; ¬ *contains-insync e* ⟧ ⟹ *bisim Vs* (*Val v*⌊*i*⌉:=*e*) (*Val v*⌊*i′*⌉:=*compE1 Vs e*) *xs*
| *bisimAAss3*: *bisim Vs e e′ xs* ⟹ *bisim Vs* (*Val v*⌊*Val i*⌉ := *e*) (*Val v*⌊*Val i*⌉ := *e′*) *xs*
| *bisimALength*: *bisim Vs a a′ xs* ⟹ *bisim Vs* (*a*·*length*) (*a′*·*length*) *xs*
| *bisimFAcc*: *bisim Vs e e′ xs* ⟹ *bisim Vs* (*e*·*F*{*D*}) (*e′*·*F*{*D*}) *xs*
| *bisimFAss1*: ⟦ *bisim Vs e e′ xs*; ¬ *is-val e*; ¬ *contains-insync e″* ⟧ ⟹ *bisim Vs* (*e*·*F*{*D*}:=*e″*) (*e′*·*F*{*D*}:=*compE1 Vs e″*) *xs*
| *bisimFAss2*: *bisim Vs e e′ xs* ⟹ *bisim Vs* (*Val v*·*F*{*D*} := *e*) (*Val v*·*F*{*D*} := *e′*) *xs*
| *bisimCAS1*: ⟦ *bisim Vs e e′ xs*; ¬ *is-val e*; ¬ *contains-insync e2*; ¬ *contains-insync e3* ⟧
⟹ *bisim Vs* (*e*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) (*e′*·*compareAndSwap*(*D*·*F*, *compE1 Vs e2*, *compE1 Vs e3*)) *xs*
| *bisimCAS2*: ⟦ *bisim Vs e e′ xs*; ¬ *is-val e*; ¬ *contains-insync e3* ⟧
⟹ *bisim Vs* (*Val v*·*compareAndSwap*(*D*·*F*, *e*, *e3*)) (*Val v*·*compareAndSwap*(*D*·*F*, *e′*, *compE1 Vs e3*)) *xs*
| *bisimCAS3*: *bisim Vs e e′ xs* ⟹ *bisim Vs* (*Val v*·*compareAndSwap*(*D*·*F*, *Val v′*, *e*)) (*Val v*·*compareAndSwap*(*D*·*F*, *Val v′*, *e′*)) *xs*
| *bisimCallObj*: ⟦ *bisim Vs e e′ xs*; ¬ *is-val e*; ¬ *contains-insyncs es* ⟧ ⟹ *bisim Vs* (*e*·*M*(*es*)) (*e′*·*M*(*compEs1 Vs es*)) *xs*
| *bisimCallParams*: *bisims Vs es es′ xs* ⟹ *bisim Vs* (*Val v*·*M*(*es*)) (*Val v*·*M*(*es′*)) *xs*
| *bisimBlockNone*: *bisim* (*Vs*@[*V*]) *e e′ xs* ⟹ *bisim Vs* {*V*:*T*=*None*; *e*} {(*length Vs*):*T*=*None*; *e′*} *xs*
| *bisimBlockSome*: ⟦ *bisim* (*Vs*@[*V*]) *e e′* (*xs*[*length Vs* := *v*]) ⟧ ⟹ *bisim Vs* {*V*:*T*=⌊*v*⌋; *e*} {(*length Vs*):*T*=⌊*v*⌋; *e′*} *xs*
| *bisimBlockSomeNone*: ⟦ *bisim* (*Vs*@[*V*]) *e e′ xs*; *xs* ! (*length Vs*) = *v* ⟧ ⟹ *bisim Vs* {*V*:*T*=⌊*v*⌋; *e*}

{(*length Vs*):*T*=*None*; *e′*} *xs*
| *bisimSynchronized*:
⟦ *bisim Vs o′ o″ xs*; ¬ *contains-insync e* ⟧
⟹ *bisim Vs* (*sync*(*o′*) *e*) (*sync*$_{length\ Vs}$(*o″*) (*compE1* (*Vs*@[*fresh-var Vs*]) *e*)) *xs*
| *bisimInSynchronized*:
⟦ *bisim* (*Vs*@[*fresh-var Vs*]) *e e′ xs*; *xs* ! *length Vs* = *Addr a* ⟧ ⟹ *bisim Vs* (*insync*(*a*) *e*) (*insync*$_{length\ Vs}$(*a*) *e′*) *xs*
| *bisimSeq*: ⟦ *bisim Vs e e′ xs*; ¬ *contains-insync e″* ⟧ ⟹ *bisim Vs* (*e*;;*e″*) (*e′*;;*compE1 Vs e″*) *xs*
| *bisimCond*:
⟦ *bisim Vs e e′ xs*; ¬ *contains-insync e1*; ¬ *contains-insync e2* ⟧
⟹ *bisim Vs* (*if* (*e*) *e1 else e2*) (*if* (*e′*) (*compE1 Vs e1*) *else* (*compE1 Vs e2*)) *xs*
| *bisimWhile*:
⟦ ¬ *contains-insync b*; ¬ *contains-insync e* ⟧ ⟹ *bisim Vs* (*while* (*b*) *e*) (*while* (*compE1 Vs b*) (*compE1 Vs e*)) *xs*
| *bisimThrow*: *bisim Vs e e′ xs* ⟹ *bisim Vs* (*throw e*) (*throw e′*) *xs*
| *bisimTryCatch*:
⟦ *bisim Vs e e′ xs*; ¬ *contains-insync e″* ⟧
⟹ *bisim Vs* (*try e catch*(*C V*) *e″*) (*try e′ catch*(*C* (*length Vs*)) *compE1* (*Vs*@[*V*]) *e″*) *xs*

| *bisimsNil*: *bisims Vs* [] [] *xs*
| *bisimsCons1*: ⟦ *bisim Vs e e′ xs*; ¬ *is-val e*; ¬ *contains-insyncs es* ⟧ ⟹ *bisims Vs* (*e* # *es*) (*e′* # *compEs1 Vs es*) *xs*
| *bisimsCons2*: *bisims Vs es es′ xs* ⟹ *bisims Vs* (*Val v* # *es*) (*Val v* # *es′*) *xs*

declare *bisimNew* [*iff*]
declare *bisimVal* [*iff*]
declare *bisimVar* [*iff*]
declare *bisimWhile* [*iff*]
declare *bisimsNil* [*iff*]

declare *bisim-bisims.intros* [*intro*!]
declare *bisimsCons1* [*rule del*, *intro*] *bisimsCons2* [*rule del*, *intro*]
bisimBinOp1 [*rule del*, *intro*] *bisimAAcc1* [*rule del*, *intro*]
bisimAAss1 [*rule del*, *intro*] *bisimAAss2* [*rule del*, *intro*]
bisimFAss1 [*rule del*, *intro*]
bisimCAS1 [*rule del*, *intro*] *bisimCAS2* [*rule del*, *intro*]
bisimCallObj [*rule del*, *intro*]

inductive-cases *bisim-safe-cases* [*elim*!]:
bisim Vs (*new C*) *e′ xs*
bisim Vs (*newA T*⌊*e*⌉) *e′ xs*
bisim Vs (*Cast T e*) *e′ xs*
bisim Vs (*e instanceof T*) *e′ xs*
bisim Vs (*Val v*) *e′ xs*
bisim Vs (*Var V*) *e′ xs*
bisim Vs (*V*:=*e*) *e′ xs*
bisim Vs (*Val v*⌊*i*⌉) *e′ xs*
bisim Vs (*Val v*⌊*Val v′*⌉ := *e*) *e′ xs*
bisim Vs (*Val v*·*compareAndSwap*(*D*·*F*, *Val v′*, *e*)) *e′ xs*
bisim Vs (*a*·*length*) *e′ xs*
bisim Vs (*e*·*F*{*D*}) *e′ xs*
bisim Vs (*sync*(*o′*) *e*) *e′ xs*
bisim Vs (*insync*(*a*) *e*) *e′ xs*
bisim Vs (*e*;;*e′*) *e″ xs*

bisim Vs (*if* (*b*) *e1 else e2*) *e′ xs*
bisim Vs (*while* (*b*) *e*) *e′ xs*
bisim Vs (*throw e*) *e′ xs*
bisim Vs (*try e catch*(*C V*) *e′*) *e″ xs*
bisim Vs e′ (*new C*) *xs*
bisim Vs e′ (*newA T*⌊*e*⌉) *xs*
bisim Vs e′ (*Cast T e*) *xs*
bisim Vs e′ (*e instanceof T*) *xs*
bisim Vs e′ (*Val v*) *xs*
bisim Vs e′ (*Var V*) *xs*
bisim Vs e′ (*V*:=*e*) *xs*
bisim Vs e′ (*Val v*⌊*i*⌉) *xs*
bisim Vs e′ (*Val v*⌊*Val v′*⌉ := *e*) *xs*
bisim Vs e′ (*Val v*·*compareAndSwap*(*D*·*F*, *Val v′*, *e*)) *xs*
bisim Vs e′ (*a*·*length*) *xs*
bisim Vs e′ (*e*·*F*{*D*}) *xs*
bisim Vs e′ (*sync*$_V$ (*o′*) *e*) *xs*
bisim Vs e′ (*insync*$_V$ (*a*) *e*) *xs*
bisim Vs e″ (*e*;;*e′*) *xs*
bisim Vs e′ (*if* (*b*) *e1 else e2*) *xs*
bisim Vs e′ (*while* (*b*) *e*) *xs*
bisim Vs e′ (*throw e*) *xs*
bisim Vs e″ (*try e catch*(*C V*) *e′*) *xs*

inductive-cases *bisim-cases* [*elim*]:
bisim Vs (*e1* «*bop*» *e2*) *e′ xs*
bisim Vs (*a*⌊*i*⌉) *e′ xs*
bisim Vs (*a*⌊*i*⌉:=*e*) *e′ xs*
bisim Vs (*e*·*F*{*D*}:=*e′*) *e″ xs*
bisim Vs (*e*·*compareAndSwap*(*D*·*F*, *e′*, *e″*)) *e‴ xs*
bisim Vs (*e*·*M*(*es*)) *e′ xs*
bisim Vs {*V*:*T*=*vo*; *e*} *e′ xs*
bisim Vs e′ (*e1* «*bop*» *e2*) *xs*
bisim Vs e′ (*a*⌊*i*⌉) *xs*
bisim Vs e′ (*a*⌊*i*⌉:=*e*) *xs*
bisim Vs e″ (*e*·*F*{*D*}:=*e′*) *xs*
bisim Vs e‴ (*e*·*compareAndSwap*(*D*·*F*, *e′*, *e″*)) *xs*
bisim Vs e′ (*e*·*M*(*es*)) *xs*
bisim Vs e′ {*V*:*T*=*vo*; *e*} *xs*

inductive-cases *bisims-safe-cases* [*elim*!]:
bisims Vs [] *es xs*
bisims Vs es [] *xs*

inductive-cases *bisims-cases* [*elim*]:
bisims Vs (*e* # *es*) *es′ xs*
bisims Vs es′ (*e* # *es*) *xs*

Delay bisimulation for call stacks

inductive *bisim01* :: *′addr expr* ⇒ *′addr expr1* × *′addr locals1* ⇒ *bool*
where
⟦ *bisim* [] *e e′ xs*; *fv e* = {}; $\mathcal{D}$ *e* ⌊{}⌋; *max-vars e′* ≤ *length xs*; *call e* = ⌊*aMvs*⌋; *call1 e′* = ⌊*aMvs*⌋ ⟧
⟹ *bisim01 e* (*e′*, *xs*)

inductive *bisim-list* :: *'addr expr list* ⇒ (*'addr expr1* × *'addr locals1*) *list* ⇒ *bool*
where
bisim-listNil: *bisim-list* [] []
| *bisim-listCons*:
⟦ *bisim-list es exs'*; *bisim* [] *e e' xs*;
fv e = {}; 𝒟 *e* ⌊{}⌋;
max-vars e' ≤ *length xs*;
call e = ⌊*aMvs*⌋; *call1 e'* = ⌊*aMvs*⌋ ⟧
⟹ *bisim-list* (*e* # *es*) ((*e'*, *xs*) # *exs'*)

inductive-cases *bisim-list-cases* [*elim*!]:
bisim-list [] *exs'*
bisim-list (*ex* # *exs*) *exs'*
bisim-list exs (*ex'* # *exs'*)

fun *bisim-list1* ::
'addr expr × *'addr expr list* ⇒ (*'addr expr1* × *'addr locals1*) × (*'addr expr1* × *'addr locals1*) *list* ⇒ *bool*
where
bisim-list1 (*e*, *es*) ((*e1*, *xs1*), *exs1*) ⟷
bisim-list es exs1 ∧ *bisim* [] *e e1 xs1* ∧ *fv e* = {} ∧ 𝒟 *e* ⌊{}⌋ ∧ *max-vars e1* ≤ *length xs1*

definition *bisim-red0-Red1* ::
(*'addr expr* × *'addr expr list*) × *'heap*)
⇒ ((((*'addr expr1* × *'addr locals1*) × (*'addr expr1* × *'addr locals1*) *list*) × *'heap*) ⇒ *bool*
where *bisim-red0-Red1* ≡ (λ(*es*, *h*) (*exs*, *h'*). *bisim-list1 es exs* ∧ *h* = *h'*)

abbreviation *ta-bisim01* ::
(*'addr*, *'thread-id*, *'heap*) *J0-thread-action* ⇒ (*'addr*, *'thread-id*, *'heap*) *J1-thread-action* ⇒ *bool*
where
ta-bisim01 ≡ *ta-bisim* (λ*t*. *bisim-red0-Red1*)

definition *bisim-wait01* ::
(*'addr expr* × *'addr expr list*) ⇒ (*'addr expr1* × *'addr locals1*) × (*'addr expr1* × *'addr locals1*) *list*
⇒ *bool*
where *bisim-wait01* ≡ λ(*e0*, *es0*) ((*e1*, *xs1*), *exs1*). *call e0* ≠ *None* ∧ *call1 e1* ≠ *None*

lemma *bisim-list1I*[*intro?*]:
⟦ *bisim-list es exs1*; *bisim* [] *e e1 xs1*; *fv e* = {};
𝒟 *e* ⌊{}⌋; *max-vars e1* ≤ *length xs1* ⟧
⟹ *bisim-list1* (*e*, *es*) ((*e1*, *xs1*), *exs1*)
by *simp*

lemma *bisim-list1E*[*elim?*]:
assumes *bisim-list1* (*e*, *es*) ((*e1*, *xs1*), *exs1*)
obtains *bisim-list es exs1 bisim* [] *e e1 xs1 fv e* = {} 𝒟 *e* ⌊{}⌋ *max-vars e1* ≤ *length xs1*
using *assms* **by** *auto*

lemma *bisim-list1-elim*:
assumes *bisim-list1 es' exs*
obtains *e es e1 xs1 exs1*
where *es'* = (*e*, *es*) *exs* = ((*e1*, *xs1*), *exs1*)
and *bisim-list es exs1 bisim* [] *e e1 xs1 fv e* = {} 𝒟 *e* ⌊{}⌋ *max-vars e1* ≤ *length xs1*
using *assms* **by**(*cases es'*)(*cases exs*, *fastforce*)

declare *bisim-list1.simps* [*simp del*]

lemma *bisims-map-Val-conv* [*simp*]: *bisims Vs* (*map Val vs*) *es xs* = (*es* = *map Val vs*)
apply(*induct vs arbitrary*: *es*)
apply(*fastforce*)
apply(*simp*)
apply(*rule iffI*)
apply(*erule bisims-cases*, *auto*)
done

declare *compEs1-conv-map* [*simp del*]

lemma *bisim-contains-insync*: *bisim Vs e e' xs* ⟹ *contains-insync e* = *contains-insync e'*
and *bisims-contains-insyncs*: *bisims Vs es es' xs* ⟹ *contains-insyncs es* = *contains-insyncs es'*
by(*induct rule*: *bisim-bisims.inducts*)(*auto*)

lemma *bisims-map-Val-Throw*:
bisims Vs (*map Val vs* @ *Throw a* # *es*) *es' xs* ⟷ *es'* = *map Val vs* @ *Throw a* # *compEs1 Vs es* ∧ ¬ *contains-insyncs es*
apply(*induct vs arbitrary*: *es'*)
apply(*simp*)
apply(*fastforce simp add*: *compEs1-conv-map*)
apply(*fastforce elim*!: *bisims-cases intro*: *bisimsCons2*)
done

lemma *compE1-bisim* [*intro*]: ⟦ *fv e* ⊆ *set Vs*; ¬ *contains-insync e* ⟧ ⟹ *bisim Vs e* (*compE1 Vs e*) *xs*
and *compEs1-bisims* [*intro*]: ⟦ *fvs es* ⊆ *set Vs*; ¬ *contains-insyncs es* ⟧ ⟹ *bisims Vs es* (*compEs1 Vs es*) *xs*
proof(*induct Vs e* **and** *Vs es arbitrary*: *xs* **and** *xs rule*: *compE1-compEs1-induct*)
case (*BinOp Vs exp1 bop exp2 x*)
thus *?case* **by**(*cases is-val exp1*)(*auto*)
next
case (*AAcc Vs exp1 exp2 x*)
thus *?case* **by**(*cases is-val exp1*)(*auto*)
next
case (*AAss Vs exp1 exp2 exp3 x*)
thus *?case* **by**(*cases is-val exp1*, *cases is-val exp2*, *fastforce*+)
next
case (*FAss Vs exp1 F D exp2 x*)
thus *?case* **by**(*cases is-val exp1*, *auto*)
next
case (*CAS Vs e1 D F e2 e3 x*)
thus *?case* **by**(*cases is-val e1*, *cases is-val e2*, *fastforce*+)
next
case (*Call Vs obj M params x*)
thus *?case* **by**(*cases is-val obj*)(*auto*)
next
case (*Block Vs V T vo exp xs*)
from ‹*fv* {*V*:*T*=*vo*; *exp*} ⊆ *set Vs*› **have** *fv exp* ⊆ *set* (*Vs*@[*V*]) **by**(*auto*)
with *Block* **show** *?case* **by**(*cases vo*)(*auto*)
next
case (*Cons Vs exp list x*)

thus *?case* **by**(*cases is-val exp*)(*auto intro*!: *bisimsCons2*)
qed(*auto*)

lemma *bisim-hidden-unmod*: ⟦ *bisim Vs e e′ xs*; *hidden Vs i* ⟧ ⟹ *unmod e′ i*
and *bisims-hidden-unmods*: ⟦ *bisims Vs es es′ xs*; *hidden Vs i* ⟧ ⟹ *unmods es′ i*
by(*induct rule*: *bisim-bisims.inducts*)(*auto intro*: *hidden-unmod hidden-unmods dest*: *hidden-inacc hidden-lengthD*)

lemma *bisim-fv-unmod*: ⟦ *bisim Vs e e′ xs*; *i* < *length Vs*; *Vs* ! *i* ∉ *fv e* ⟧ ⟹ *unmod e′ i*
and *bisims-fvs-unmods*: ⟦ *bisims Vs es es′ xs*; *i* < *length Vs*; *Vs* ! *i* ∉ *fvs es* ⟧ ⟹ *unmods es′ i*
proof(*induct rule*: *bisim-bisims.inducts*)
case (*bisimBlockNone Vs V e e′ xs T*)
note *len* = ‹*i* < *length Vs*›
have *unmod e′ i*
proof(*cases Vs* ! *i* = *V*)
case *True*
from *len* **have** *hidden* (*Vs* @ [*Vs* ! *i*]) *i* **by**(*rule hidden-snoc-nth*)
with *len True* ‹*bisim* (*Vs* @ [*V*]) *e e′ xs*› **show** *?thesis* **by**(*auto intro*: *bisim-hidden-unmod*)
next
case *False*
with *bisimBlockNone* **show** *?thesis* **by**(*auto simp add*: *nth-append*)
qed
thus *?case* **by** *simp*
next
case (*bisimBlockSome Vs V e e′ xs v T*)
note *len* = ‹*i* < *length Vs*›
show *?case*
proof(*cases Vs* ! *i* = *V*)
case *True*
from *len* **have** *hidden* (*Vs* @ [*Vs* ! *i*]) *i* **by**(*rule hidden-snoc-nth*)
with *len True* ‹*bisim* (*Vs* @ [*V*]) *e e′* (*xs*[*length Vs* := *v*])›
show *?thesis* **by**(*auto intro*: *bisim-hidden-unmod*)
next
case *False*
with *bisimBlockSome* **show** *?thesis* **by**(*auto simp add*: *nth-append*)
qed
next
case (*bisimBlockSomeNone Vs V e e′ xs v T*)
note *len* = ‹*i* < *length Vs*›
show *?case*
proof(*cases Vs* ! *i* = *V*)
case *True*
from *len* **have** *hidden* (*Vs* @ [*Vs* ! *i*]) *i* **by**(*rule hidden-snoc-nth*)
with *len True* ‹*bisim* (*Vs* @ [*V*]) *e e′ xs*›
show *?thesis* **by**(*auto intro*: *bisim-hidden-unmod*)
next
case *False*
with *bisimBlockSomeNone* **show** *?thesis* **by**(*auto simp add*: *nth-append*)
qed
qed(*fastforce dest*: *fv-unmod-compE1 fvs-unmods-compEs1 index-le-lengthD simp add*: *nth-append*)+

lemma *bisim-extRet2J* [*intro*!]: *bisim Vs e e′ xs* ⟹ *bisim Vs* (*extRet2J e va*) (*extRet2J1 e′ va*) *xs*
by(*cases va*) *auto*

lemma *bisims-map-Val-conv2* [*simp*]: *bisims Vs es* (*map Val vs*) *xs* = (*es* = *map Val vs*)
apply(*induct vs arbitrary*: *es*)
apply(*fastforce elim*!: *bisims-cases*)+
done

lemma *bisims-map-Val-Throw2*:
bisims Vs es' (*map Val vs* @ *Throw a* # *es*) *xs* ⟷
(∃ *es''*. *es'* = *map Val vs* @ *Throw a* # *es''* ∧ *es* = *compEs1 Vs es''* ∧ ¬ *contains-insyncs es''*)
apply(*induct vs arbitrary*: *es'*)
apply(*simp*)
apply(*fastforce simp add*: *compEs1-conv-map*)
apply(*fastforce elim*!: *bisims-cases*)
done

lemma *hidden-bisim-unmod*: ⟦ *bisim Vs e e' xs*; *hidden Vs i* ⟧ ⟹ *unmod e' i*
and *hidden-bisims-unmods*: ⟦ *bisims Vs es es' xs*; *hidden Vs i* ⟧ ⟹ *unmods es' i*
apply(*induct rule*: *bisim-bisims.inducts*)
apply(*auto simp add*:*hidden-inacc intro*: *hidden-unmod hidden-unmods*)
apply(*auto simp add*: *hidden-def*)
done

lemma *bisim-list-list-all2-conv*:
bisim-list es exs' ⟷ *list-all2 bisim01 es exs'*
proof
assume *bisim-list es exs'*
thus *list-all2 bisim01 es exs'*
by *induct*(*auto intro*!: *bisim01.intros*)
next
assume *list-all2 bisim01 es exs'*
thus *bisim-list es exs'*
by(*induct es arbitrary*: *exs'*)(*auto intro*!: *bisim-listCons bisim-listNil elim*!: *bisim01.cases simp add*: *list-all2-Cons1*)
qed

lemma *bisim-list-extTA2J0-extTA2J1*:
assumes *wf*: *wf-J-prog P*
and *sees*: *P* ⊢ *C sees M*:[]→*T* = ⌊*meth*⌋ *in D*
shows *bisim-list1* (*extNTA2J0 P* (*C*, *M*, *a*)) (*extNTA2J1* (*compP1 P*) (*C*, *M*, *a*))
proof −
obtain *pns body* **where** *meth* = (*pns*, *body*) **by**(*cases meth*)
with *sees* **have** *sees*: *P* ⊢ *C sees M*:[]→*T* = ⌊(*pns*, *body*)⌋ *in D* **by** *simp*
moreover let *?xs* = *Addr a* # *replicate* (*max-vars body*) *undefined-value*
let *?e'* = {*0*:*Class D*=*None*; *compE1* (*this* # *pns*) *body*}
have *bisim-list1* ({*this*:*Class D*=⌊*Addr a*⌋; *body*}, []) ((*?e'*, *?xs*), [])
proof
show *bisim-list* [] [] ..
from *sees-wf-mdecl*[*OF wf-prog-wwf-prog*[*OF wf*] *sees*] **have** *fv body* ⊆ *set* [*this*] *pns* = []
by(*auto simp add*: *wf-mdecl-def*)
thus *fv* ({*this*:*Class D*=⌊*Addr a*⌋; *body*}) = {} **by** *simp*
from *sees-wf-mdecl*[*OF wf sees*] **obtain** *T'* **where** *P*,[*this* ↦ *Class D*] ⊢ *body* :: *T' this* ∉ *set pns*
and *D body* ⌊*dom* [*this* ↦ *Addr a*]⌋ **by**(*auto simp add*: *wf-mdecl-def*)
hence ¬ *contains-insync body* **by**(*auto simp add*: *contains-insync-conv dest*: *WT-expr-locks*)
with ‹*fv body* ⊆ *set* [*this*]›
have *bisim* ([] @ [*this*]) *body* (*compE1* (*this* # *pns*) *body*) *?xs*

unfolding *append.simps* ‹*pns* = []› **by**(*rule compE1-bisim*)
hence *bisim* [] {*this*:*Class D*=⌊*Addr a*⌋; *body*} {*length* ([] :: *String.literal list*):*Class D*=*None*; *compE1* (*this* # *pns*) *body*} *?xs*
by(*rule bisimBlockSomeNone*)(*simp*)
thus *bisim* [] ({*this*:*Class D*=⌊*Addr a*⌋; *body*}) *?e'* *?xs* **by** *simp*
from ‹$\mathcal{D}$ *body* ⌊*dom* [*this* ↦ *Addr a*]⌋› **show** $\mathcal{D}$ ({*this*:*Class D*=⌊*Addr a*⌋; *body*}) ⌊{}⌋ **by** *simp*
show *max-vars ?e'* ≤ *length ?xs* **by** *simp*
qed
ultimately show *?thesis* **by**(*simp*)
qed

lemma *bisim-max-vars*: *bisim Vs e e' xs* ⟹ *max-vars e* = *max-vars e'*
and *bisims-max-varss*: *bisims Vs es es' xs* ⟹ *max-varss es* = *max-varss es'*
apply(*induct rule*: *bisim-bisims.inducts*)
apply(*auto simp add*: *max-vars-compE1 max-varss-compEs1*)
done

lemma *bisim-call*: *bisim Vs e e' xs* ⟹ *call e* = *call e'*
and *bisims-calls*: *bisims Vs es es' xs* ⟹ *calls es* = *calls es'*
apply(*induct rule*: *bisim-bisims.inducts*)
apply(*auto simp add*: *is-vals-conv*)
done

lemma *bisim-call-None-call1*: ⟦ *bisim Vs e e' xs*; *call e* = *None* ⟧ ⟹ *call1 e'* = *None*
and *bisims-calls-None-calls1*: ⟦ *bisims Vs es es' xs*; *calls es* = *None* ⟧ ⟹ *calls1 es'* = *None*
by(*induct rule*: *bisim-bisims.inducts*)(*auto simp add*: *is-vals-conv split*: *if-split-asm*)

lemma *bisim-call1-Some-call*:
⟦ *bisim Vs e e' xs*; *call1 e'* = ⌊*aMvs*⌋ ⟧ ⟹ *call e* = ⌊*aMvs*⌋

and *bisims-calls1-Some-calls*:
⟦ *bisims Vs es es' xs*; *calls1 es'* = ⌊*aMvs*⌋ ⟧ ⟹ *calls es* = ⌊*aMvs*⌋
by(*induct rule*: *bisim-bisims.inducts*)(*auto simp add*: *is-vals-conv split*: *if-split-asm*)

lemma *blocks-bisim*:
assumes *bisim*: *bisim* (*Vs* @ *pns*) *e e' xs*
and *length*: *length vs* = *length pns length Ts* = *length pns*
and *xs*: ∀ *i*. *i* < *length vs* ⟶ *xs* ! (*i* + *length Vs*) = *vs* ! *i*
shows *bisim Vs* (*blocks pns Ts vs e*) (*blocks1* (*length Vs*) *Ts e'*) *xs*
using *bisim length xs*
proof(*induct pns Ts vs e arbitrary*: *e' Vs rule*: *blocks.induct*)
case (*1 V Vs T Ts v vs e e' VS*)
note *IH* = ‹⋀*e' Vsa*. ⟦*bisim* (*Vsa* @ *Vs*) *e e' xs*;
length vs = *length Vs*; *length Ts* = *length Vs*; ∀ *i*<*length vs*. *xs* ! (*i* + *length Vsa*) = *vs* ! *i*⟧
⟹ *bisim Vsa* (*blocks Vs Ts vs e*) (*blocks1* (*length Vsa*) *Ts e'*) *xs*›
note *xs* = ‹∀ *i*<*length* (*v* # *vs*). *xs* ! (*i* + *length VS*) = (*v* # *vs*) ! *i*›
hence *xs'*: ∀ *i*<*length vs*. *xs* ! (*i* + *length* (*VS* @ [*V*])) = *vs* ! *i* **and** *v*: *xs* ! *length VS* = *v* **by**(*auto*)
from ‹*bisim* (*VS* @ *V* # *Vs*) *e e' xs*› **have** *bisim* ((*VS* @ [*V*]) @ *Vs*) *e e' xs* **by** *simp*
from *IH*[*OF this - - xs'*] ‹*length* (*v* # *vs*) = *length* (*V* # *Vs*)› ‹*length* (*T* # *Ts*) = *length* (*V* # *Vs*)›
have *bisim* (*VS* @ [*V*]) (*blocks Vs Ts vs e*) (*blocks1* (*length* (*VS* @ [*V*])) *Ts e'*) *xs*
by *auto*
hence *bisim VS* ({*V*:*T*=⌊*v*⌋; *blocks Vs Ts vs e*}) {*length VS*:*T*=*None*; *blocks1* (*length* (*VS* @ [*V*]))

Ts e'} *xs*
using *v* **by**(*rule bisimBlockSomeNone*)
thus *?case* **by** *simp*
qed(*auto*)

lemma fixes *e* :: ('*a*,'*b*,'*addr*) *exp* **and** *es* :: ('*a*,'*b*,'*addr*) *exp list*
shows *inline-call-max-vars*: *call e* = ⌊*aMvs*⌋ ⟹ *max-vars* (*inline-call e' e*) ≤ *max-vars e* + *max-vars e'*
and *inline-calls-max-varss*: *calls es* = ⌊*aMvs*⌋ ⟹ *max-varss* (*inline-calls e' es*) ≤ *max-varss es* + *max-vars e'*
by(*induct e* **and** *es rule*: *call.induct calls.induct*)(*auto*)

lemma assumes *final E bisim VS E E' xs*
shows *inline-call-compE1*: *call e* = ⌊*aMvs*⌋ ⟹ *inline-call E'* (*compE1 Vs e*) = *compE1 Vs* (*inline-call E e*)
and *inline-calls-compEs1*: *calls es* = ⌊*aMvs*⌋ ⟹ *inline-calls E'* (*compEs1 Vs es*) = *compEs1 Vs* (*inline-calls E es*)
proof(*induct Vs e* **and** *Vs es rule*: *compE1-compEs1-induct*)
case (*Call Vs obj M params*)
note *IHobj* = ‹*call obj* = ⌊*aMvs*⌋ ⟹ *inline-call E'* (*compE1 Vs obj*) = *compE1 Vs* (*inline-call E obj*)›
note *IHparams* = ‹*calls params* = ⌊*aMvs*⌋ ⟹ *inline-calls E'* (*compEs1 Vs params*) = *compEs1 Vs* (*inline-calls E params*)›
obtain *a M' vs* **where** [*simp*]: *aMvs* = (*a*, *M'*, *vs*) **by** (*cases aMvs*, *auto*)
with ‹*call* (*obj*·*M*(*params*)) = ⌊*aMvs*⌋› **have** *call* (*obj*·*M*(*params*)) = ⌊(*a*, *M'*, *vs*)⌋ **by** *simp*
thus *?case*
proof(*induct rule*: *call-callE*)
case *CallObj*
with *IHobj* **have** *inline-call E'* (*compE1 Vs obj*) = *compE1 Vs* (*inline-call E obj*) **by** *auto*
with *CallObj* **show** *?case* **by** *auto*
next
case (*CallParams v*)
with *IHparams* **have** *inline-calls E'* (*compEs1 Vs params*) = *compEs1 Vs* (*inline-calls E params*)
by *auto*
with *CallParams* **show** *?case* **by**(*auto simp add*: *is-vals-conv*)
next
case *Call*
with ‹*final E*› ‹*bisim VS E E' xs*› **show** *?case* **by**(*auto simp add*: *is-vals-conv*)
qed
qed(*auto split*: *if-split-asm*)

lemma assumes *bisim*: *bisim VS E E' XS*
and *final*: *final E*
shows *bisim-inline-call*:
⟦ *bisim Vs e e' xs*; *call e* = ⌊*aMvs*⌋; *fv e* ⊆ *set Vs* ⟧
⟹ *bisim Vs* (*inline-call E e*) (*inline-call E' e'*) *xs*

and *bisims-inline-calls*:
⟦ *bisims Vs es es' xs*; *calls es* = ⌊*aMvs*⌋; *fvs es* ⊆ *set Vs* ⟧
⟹ *bisims Vs* (*inline-calls E es*) (*inline-calls E' es'*) *xs*
proof(*induct rule*: *bisim-bisims.inducts*)
case (*bisimBinOp1 Vs e e' xs bop e''*)
thus *?case* **by**(*cases is-val* (*inline-call E e*))(*fastforce*)+
next

```
  case (bisimAAcc1 Vs a a' xs i)
  thus ?case by(cases is-val (inline-call E a))(fastforce)+
next
  case (bisimAAss1 Vs a a' xs i e)
  thus ?case by(cases is-val (inline-call E a), cases is-val i)(fastforce)+
next
  case (bisimAAss2 Vs i i' xs a e)
  thus ?case by(cases is-val (inline-call E i))(fastforce)+
next
  case (bisimFAss1 Vs e e' xs F D e'')
  thus ?case by(cases is-val (inline-call E e))(fastforce)+
next
  case (bisimCAS1 Vs e e' xs e2 e3 D F)
  thus ?case
    apply(cases is-val (inline-call E e))
     apply(cases is-val e2)
      apply(fastforce)
     apply clarsimp
     apply(safe; clarsimp?)
     apply auto
    done
next
  case (bisimCAS2 Vs e e' xs e3 v D F)
  thus ?case by(cases is-val (inline-call E e); safe?; clarsimp; fastforce)
next
  case (bisimCallObj Vs e e' xs es M)
  obtain a M' vs where aMvs = (a, M', vs) by(cases aMvs, auto)
  with ‹call (e•M(es)) = ⌊aMvs⌋› have call (e•M(es)) = ⌊(a, M', vs)⌋ by simp
  thus ?case
  proof(induct rule: call-callE)
    case CallObj
    with ‹fv (e•M(es)) ⊆ set Vs› ‹aMvs = (a, M', vs)›
      ‹⟦call e = ⌊aMvs⌋; fv e ⊆ set Vs⟧ ⟹ bisim Vs (inline-call E e) (inline-call E' e') xs›
    have IH': bisim Vs (inline-call E e) (inline-call E' e') xs by(auto)
    with ‹bisim Vs e e' xs› ‹fv (e•M(es)) ⊆ set Vs› CallObj ‹¬ contains-insyncs es› show ?thesis
      by(cases is-val (inline-call E e))(fastforce)+
  next
    case (CallParams v)
    hence inline-calls E' (compEs1 Vs es) = compEs1 Vs (inline-calls E es)
      by −(rule inline-calls-compEs1[OF final bisim])
    moreover from ‹fv (e•M(es)) ⊆ set Vs› final fvs-inline-calls[of E es]
    have fvs (inline-calls E es) ⊆ set Vs by(auto elim!: final.cases)
     moreover note CallParams ‹bisim Vs e e' xs› ‹fv (e•M(es)) ⊆ set Vs› ‹¬ contains-insyncs es›
final
    ultimately show ?case by(auto simp add: is-vals-conv final-iff)
  next
    case Call
    with final bisim ‹bisim Vs e e' xs› show ?case by(auto simp add: is-vals-conv)
  qed
next
  case (bisimCallParams Vs es es' xs v M)
  obtain a M' vs where [simp]: aMvs = (a, M', vs) by(cases aMvs, auto)
  with ‹call (Val v•M(es)) = ⌊aMvs⌋› have call (Val v•M(es)) = ⌊(a, M', vs)⌋ by simp
  thus ?case
```

```
  proof(induct rule: call-callE)
    case CallObj thus ?case by simp
  next
    case (CallParams v')
    with ‹ ⟦calls es = ⌊aMvs⌋; fvs es ⊆ set Vs⟧ ⟹ bisims Vs (inline-calls E es) (inline-calls E' es')
xs› ‹fv (Val v·M(es)) ⊆ set Vs›
    have bisims Vs (inline-calls E es) (inline-calls E' es') xs by(auto)
    with final bisim ‹bisims Vs es es' xs› show ?case by(auto simp add: is-vals-conv)
  next
    case Call
    with final bisim ‹bisims Vs es es' xs› show ?case by(auto)
  qed
next
  case (bisimsCons1 Vs e e' xs es)
  thus ?case by(cases is-val (inline-call E e))(fastforce)+
qed(fastforce)+

declare hyperUn-ac [simp del]

lemma sqInt-lem3: ⟦ A ⊑ A'; B ⊑ B' ⟧ ⟹ A ⊓ B ⊑ A' ⊓ B'
by(auto simp add: hyperset-defs)

lemma sqUn-lem3: ⟦ A ⊑ A'; B ⊑ B' ⟧ ⟹ A ⊔ B ⊑ A' ⊔ B'
by(auto simp add: hyperset-defs)

lemma A-inline-call: call e = ⌊aMvs⌋ ⟹ 𝒜 e ⊑ 𝒜 (inline-call e' e)
  and As-inline-calls: calls es = ⌊aMvs⌋ ⟹ 𝒜s es ⊑ 𝒜s (inline-calls e' es)
proof(induct e and es rule: call.induct calls.induct)
  case (Call obj M params)
  obtain a M' vs where [simp]: aMvs = (a, M', vs) by(cases aMvs, auto)
  with ‹call (obj·M(params)) = ⌊aMvs⌋› have call (obj·M(params)) = ⌊(a, M', vs)⌋ by simp
  thus ?case
  proof(induct rule: call-callE)
    case CallObj
    with ‹call obj = ⌊aMvs⌋ ⟹ 𝒜 obj ⊑ 𝒜 (inline-call e' obj)›
    show ?case by(auto intro: sqUn-lem)
  next
    case CallParams
    with ‹calls params = ⌊aMvs⌋ ⟹ 𝒜s params ⊑ 𝒜s (inline-calls e' params)›
    show ?case by(auto intro: sqUn-lem)
  next
    case Call
    thus ?case by(auto simp add: hyperset-defs)
  qed
next
  case Block thus ?case by(fastforce intro: diff-lem)
next
  case throw thus ?case by(simp add: hyperset-defs)
next
  case TryCatch thus ?case by(auto intro: sqInt-lem)
qed(fastforce intro: sqUn-lem sqUn-lem2)+

lemma assumes final e'
  shows defass-inline-call: ⟦ call e = ⌊aMvs⌋; 𝒟 e A ⟧ ⟹ 𝒟 (inline-call e' e) A
```

```
  and defasss-inline-calls: ⟦ calls es = ⌊aMvs⌋; 𝒟s es A ⟧ ⟹ 𝒟s (inline-calls e′ es) A
proof(induct e and es arbitrary: A and A rule: call.induct calls.induct)
  case (Call obj M params A)
  obtain a M′ vs where [simp]: aMvs = (a, M′, vs) by(cases aMvs, auto)
  with ‹call (obj·M(params)) = ⌊aMvs⌋› have call (obj·M(params)) = ⌊(a, M′, vs)⌋  by simp
  thus ?case
  proof(cases rule: call-callE)
    case CallObj
    with ‹𝒟 (obj·M(params)) A› ‹⟦call obj = ⌊aMvs⌋; 𝒟 obj A⟧ ⟹ 𝒟 (inline-call e′ obj) A›
    have 𝒟 (inline-call e′ obj) A by simp
    moreover from A-inline-call[OF CallObj, of e′]
    have A ⊔ (𝒜 obj) ⊑ A ⊔ (𝒜 (inline-call e′ obj)) by(rule sqUn-lem2)
   with ‹𝒟 (obj·M(params)) A› have 𝒟s params (A ⊔ 𝒜 (inline-call e′ obj)) by(auto elim: Ds-mono′)
    ultimately show ?thesis using CallObj by auto
  next
    case (CallParams v)
     with ‹𝒟 (obj·M(params)) A› ‹⟦calls params = ⌊aMvs⌋; 𝒟s params A⟧ ⟹ 𝒟s (inline-calls e′
params) A›
    have 𝒟s (inline-calls e′ params) A by(simp)
    with CallParams show ?thesis by(auto)
  next
    case Call
    with ‹final e′› show ?thesis by(auto elim!: D-mono′ simp add: hyperset-defs)
  qed
next
  case (Cons-exp exp exps A)
  show ?case
  proof(cases is-val exp)
    case True
    with ‹𝒟s (exp # exps) A› ‹⟦calls exps = ⌊aMvs⌋; 𝒟s exps A⟧ ⟹ 𝒟s (inline-calls e′ exps) A›
      ‹calls (exp # exps) = ⌊aMvs⌋›
    have 𝒟s (inline-calls e′ exps) A by(auto)
    with True show ?thesis by(auto)
  next
    case False
    with ‹⟦call exp = ⌊aMvs⌋; 𝒟 exp A⟧ ⟹ 𝒟 (inline-call e′ exp) A› ‹calls (exp # exps) = ⌊aMvs⌋›
‹𝒟s (exp # exps) A›
    have 𝒟 (inline-call e′ exp) A by auto
    moreover from False ‹calls (exp # exps) = ⌊aMvs⌋› have 𝒜 exp ⊑ 𝒜 (inline-call e′ exp)
      by(auto intro: A-inline-call)
    hence A ⊔ 𝒜 exp ⊑ A ⊔ 𝒜 (inline-call e′ exp) by(rule sqUn-lem2)
    with ‹𝒟s (exp # exps) A› have 𝒟s exps (A ⊔ 𝒜 (inline-call e′ exp))
      by(auto intro: Ds-mono′)
    ultimately show ?thesis using False by(auto)
  qed
qed(fastforce split: if-split-asm elim: D-mono′ intro: sqUn-lem2 sqUn-lem A-inline-call)+

lemma bisim-B: bisim Vs e E xs ⟹ ℬ E (length Vs)
  and bisims-Bs: bisims Vs es Es xs ⟹ ℬs Es (length Vs)
apply(induct rule: bisim-bisims.inducts)
apply(auto intro: ℬ ℬs)
done

lemma bisim-expr-locks-eq: bisim Vs e e′ xs ⟹ expr-locks e = expr-locks e′
```

and *bisims-expr-lockss-eq*: *bisims Vs es es' xs* $\Longrightarrow$ *expr-lockss es* = *expr-lockss es'*
by(*induct rule*: *bisim-bisims.inducts*)(*auto intro*!: *ext*)

lemma *bisim-list-expr-lockss-eq*: *bisim-list es exs'* $\Longrightarrow$ *expr-lockss es* = *expr-lockss* (*map fst exs'*)
apply(*induct rule*: *bisim-list.induct*)
apply(*auto dest*: *bisim-expr-locks-eq*)
done

context *J1-heap-base* **begin**

lemma [*simp*]:
fixes *e* :: ('*a*, '*b*, '*addr*) *exp* **and** *es* :: ('*a*, '*b*, '*addr*) *exp list*
shows *τmove1-compP*: *τmove1* (*compP f P*) *h e* = *τmove1 P h e*
and *τmoves1-compP*: *τmoves1* (*compP f P*) *h es* = *τmoves1 P h es*
by(*induct e* **and** *es rule*: *τmove1.induct τmoves1.induct*) *auto*

lemma *τMove1-compP* [*simp*]: *τMove1* (*compP f P*) = *τMove1 P*
by(*intro ext*) *auto*

lemma *red1-preserves-unmod*:
⟦ *uf*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; *unmod e i* ⟧ $\Longrightarrow$ (*lcl s'*) ! *i* = (*lcl s*) ! *i*

and *reds1-preserves-unmod*:
⟦ *uf*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *unmods es i* ⟧ $\Longrightarrow$ (*lcl s'*) ! *i* = (*lcl s*) ! *i*
apply(*induct rule*: *red1-reds1.inducts*)
apply(*auto split*: *if-split-asm*)
done

lemma *red1-unmod-preserved*:
⟦ *uf*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; *unmod e i* ⟧ $\Longrightarrow$ *unmod e' i*
and *reds1-unmods-preserved*:
⟦ *uf*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *unmods es i* ⟧ $\Longrightarrow$ *unmods es' i*
by(*induct rule*: *red1-reds1.inducts*)(*auto split*: *if-split-asm*)

lemma *τred1t-unmod-preserved*:
⟦ *τred1gt uf P t h* (*e*, *xs*) (*e'*, *xs'*); *unmod e i* ⟧ $\Longrightarrow$ *unmod e' i*
by(*induct rule*: *tranclp-induct2*)(*auto intro*: *red1-unmod-preserved*)

lemma *τred1r-unmod-preserved*:
⟦ *τred1gr uf P t h* (*e*, *xs*) (*e'*, *xs'*); *unmod e i* ⟧ $\Longrightarrow$ *unmod e' i*
by(*induct rule*: *rtranclp-induct2*)(*auto intro*: *red1-unmod-preserved*)

lemma *τred1t-preserves-unmod*:
⟦*τred1gt uf P t h* (*e*, *xs*) (*e'*, *xs'*); *unmod e i*; *i* < *length xs* ⟧
$\Longrightarrow$ *xs'* ! *i* = *xs* ! *i*
apply(*induct rule*: *tranclp-induct2*)
apply(*auto dest*: *red1-preserves-unmod*)
apply(*drule red1-preserves-unmod*)
apply(*erule* (*1*) *τred1t-unmod-preserved*)
apply(*drule τred1t-preserves-len*)
apply *auto*
done

lemma *τred1'r-preserves-unmod*:

⟦*τred1gr uf P t h* (*e*, *xs*) (*e′*, *xs′*); *unmod e i*; *i* < *length xs* ⟧
⟹ *xs′* ! *i* = *xs* ! *i*
apply(*induct rule*: *converse-rtranclp-induct2*)
apply(*auto dest*: *red1-preserves-unmod red1-unmod-preserved red1-preserves-len*)
apply(*frule* (*1*) *red1-unmod-preserved*)
apply(*frule red1-preserves-len*)
apply(*frule* (*1*) *red1-preserves-unmod*)
apply *auto*
done

end

context *J-heap-base* **begin**

lemma [*simp*]:
fixes *e* :: ('*a*, '*b*, '*addr*) *exp* **and** *es* :: ('*a*, '*b*, '*addr*) *exp list*
shows *τmove0-compP*: *τmove0* (*compP f P*) *h e* = *τmove0 P h e*
and *τmoves0-compP*: *τmoves0* (*compP f P*) *h es* = *τmoves0 P h es*
by(*induct e* **and** *es rule*: *τmove0.induct τmoves0.induct*) *auto*

lemma *τMove0-compP* [*simp*]: *τMove0* (*compP f P*) = *τMove0 P*
by(*intro ext*) *auto*

end

end

7.23 Unlocking a sync block never fails

theory *Correctness1Threaded* **imports**
J0J1Bisim
../Framework/FWInitFinLift
begin

definition *lock-oks1* ::
('*addr*,'*thread-id*) *locks*
⇒ ('*addr*,'*thread-id*,(('*a*,'*b*,'*addr*) *exp* × '*c*) × ((('*a*,'*b*,'*addr*) *exp* × '*c*) *list*) *thread-info* ⇒ *bool*
where
⋀*ln*. *lock-oks1 ls ts* ≡ ∀ *t*. (*case* (*ts t*) *of None* ⇒ (∀ *l*. *has-locks* (*ls* $ *l*) *t* = *0*)
| ⌊((*ex*, *exs*), *ln*)⌋ ⇒ (∀ *l*. *has-locks* (*ls* $ *l*) *t* + *ln* $ *l* = *expr-lockss* (*map fst* (*ex* # *exs*)) *l*))

primrec *el-loc-ok* :: '*addr expr1* ⇒ '*addr locals1* ⇒ *bool*
and *els-loc-ok* :: '*addr expr1 list* ⇒ '*addr locals1* ⇒ *bool*
where
el-loc-ok (*new C*) *xs* ⟷ *True*
| *el-loc-ok* (*newA T*⌊*e*⌉) *xs* ⟷ *el-loc-ok e xs*
| *el-loc-ok* (*Cast T e*) *xs* ⟷ *el-loc-ok e xs*
| *el-loc-ok* (*e instanceof T*) *xs* ⟷ *el-loc-ok e xs*
| *el-loc-ok* (*e*«*bop*»*e′*) *xs* ⟷ *el-loc-ok e xs* ∧ *el-loc-ok e′ xs*
| *el-loc-ok* (*Var V*) *xs* ⟷ *True*
| *el-loc-ok* (*Val v*) *xs* ⟷ *True*
| *el-loc-ok* (*V* := *e*) *xs* ⟷ *el-loc-ok e xs*

| *el-loc-ok* (*a*⌊*i*⌉) *xs* ⟷ *el-loc-ok a xs* ∧ *el-loc-ok i xs*
| *el-loc-ok* (*a*⌊*i*⌉ := *e*) *xs* ⟷ *el-loc-ok a xs* ∧ *el-loc-ok i xs* ∧ *el-loc-ok e xs*
| *el-loc-ok* (*a*·*length*) *xs* ⟷ *el-loc-ok a xs*
| *el-loc-ok* (*e*·*F*{*D*}) *xs* ⟷ *el-loc-ok e xs*
| *el-loc-ok* (*e*·*F*{*D*} := *e*′) *xs* ⟷ *el-loc-ok e xs* ∧ *el-loc-ok e*′ *xs*
| *el-loc-ok* (*e*·*compareAndSwap*(*D*·*F*, *e*′, *e*′′)) *xs* ⟷ *el-loc-ok e xs* ∧ *el-loc-ok e*′ *xs* ∧ *el-loc-ok e*′′ *xs*
| *el-loc-ok* (*e*·*M*(*ps*)) *xs* ⟷ *el-loc-ok e xs* ∧ *els-loc-ok ps xs*
| *el-loc-ok* {*V*:*T*=*vo*; *e*} *xs* ⟷ (*case vo of None* ⇒ *el-loc-ok e xs* | ⌊*v*⌋ ⇒ *el-loc-ok e* (*xs*[*V* := *v*]))
| *el-loc-ok* (*sync*$_V$(*e*) *e*′) *xs* ⟷ *el-loc-ok e xs* ∧ *el-loc-ok e*′ *xs* ∧ *unmod e*′ *V*
| *el-loc-ok* (*insync*$_V$(*a*) *e*) *xs* ⟷ *xs* ! *V* = *Addr a* ∧ *el-loc-ok e xs* ∧ *unmod e V*
| *el-loc-ok* (*e*;;*e*′) *xs* ⟷ *el-loc-ok e xs* ∧ *el-loc-ok e*′ *xs*
| *el-loc-ok* (*if* (*b*) *e else e*′) *xs* ⟷ *el-loc-ok b xs* ∧ *el-loc-ok e xs* ∧ *el-loc-ok e*′ *xs*
| *el-loc-ok* (*while* (*b*) *c*) *xs* ⟷ *el-loc-ok b xs* ∧ *el-loc-ok c xs*
| *el-loc-ok* (*throw e*) *xs* ⟷ *el-loc-ok e xs*
| *el-loc-ok* (*try e catch*(*C V*) *e*′) *xs* ⟷ *el-loc-ok e xs* ∧ *el-loc-ok e*′ *xs*

| *els-loc-ok* [] *xs* ⟷ *True*
| *els-loc-ok* (*e* # *es*) *xs* ⟷ *el-loc-ok e xs* ∧ *els-loc-ok es xs*

lemma *el-loc-okI*: ⟦ ¬ *contains-insync e*; *syncvars e*; ℬ *e n* ⟧ ⟹ *el-loc-ok e xs*
and *els-loc-okI*: ⟦ ¬ *contains-insyncs es*; *syncvarss es*; ℬ*s es n* ⟧ ⟹ *els-loc-ok es xs*
by(*induct e* **and** *es arbitrary*: *xs n* **and** *xs n rule*: *el-loc-ok.induct els-loc-ok.induct*)(*auto intro*: *fv-B-unmod*)

lemma *el-loc-ok-compE1*: ⟦ ¬ *contains-insync e*; *fv e* ⊆ *set Vs* ⟧ ⟹ *el-loc-ok* (*compE1 Vs e*) *xs*
and *els-loc-ok-compEs1*: ⟦ ¬ *contains-insyncs es*; *fvs es* ⊆ *set Vs* ⟧ ⟹ *els-loc-ok* (*compEs1 Vs es*) *xs*
by(*auto intro*: *el-loc-okI els-loc-okI syncvars-compE1 syncvarss-compEs1* ℬ ℬ*s simp del*: *compEs1-conv-map*)

lemma shows *el-loc-ok-not-contains-insync-local-change*:
⟦ ¬ *contains-insync e*; *el-loc-ok e xs* ⟧ ⟹ *el-loc-ok e xs*′
and *els-loc-ok-not-contains-insyncs-local-change*:
⟦ ¬ *contains-insyncs es*; *els-loc-ok es xs* ⟧ ⟹ *els-loc-ok es xs*′
by(*induct e* **and** *es arbitrary*: *xs xs*′ **and** *xs xs*′ *rule*: *el-loc-ok.induct els-loc-ok.induct*)(*fastforce*)+

lemma *el-loc-ok-update*: ⟦ ℬ *e n*; *V* < *n* ⟧ ⟹ *el-loc-ok e* (*xs*[*V* := *v*]) = *el-loc-ok e xs*
and *els-loc-ok-update*: ⟦ ℬ*s es n*; *V* < *n* ⟧ ⟹ *els-loc-ok es* (*xs*[*V* := *v*]) = *els-loc-ok es xs*
apply(*induct e* **and** *es arbitrary*: *n xs* **and** *n xs rule*: *el-loc-ok.induct els-loc-ok.induct*)
apply(*auto simp add*: *list-update-swap*)
done

lemma *els-loc-ok-map-Val* [*simp*]:
els-loc-ok (*map Val vs*) *xs*
by(*induct vs*) *auto*

lemma *els-loc-ok-map-Val-append* [*simp*]:
els-loc-ok (*map Val vs* @ *es*) *xs* = *els-loc-ok es xs*
by(*induct vs*) *auto*

lemma *el-loc-ok-extRet2J* [*simp*]:
el-loc-ok e xs ⟹ *el-loc-ok* (*extRet2J e va*) *xs*
by(*cases va*) *auto*

definition *el-loc-ok1* :: ((*nat*, *nat*, *'addr*) *exp* × *'addr locals1*) × ((*nat*, *nat*, *'addr*) *exp* × *'addr locals1*) *list* ⇒ *bool*

where *el-loc-ok1* = (λ((*e*, *xs*), *exs*). *el-loc-ok e xs* ∧ *sync-ok e* ∧ (∀(*e*,*xs*)∈*set exs*. *el-loc-ok e xs* ∧ *sync-ok e*))

lemma *el-loc-ok1-simps*:
el-loc-ok1 ((*e*, *xs*), *exs*) = (*el-loc-ok e xs* ∧ *sync-ok e* ∧ (∀(*e*,*xs*)∈*set exs*. *el-loc-ok e xs* ∧ *sync-ok e*))
by(*simp add*: *el-loc-ok1-def*)

lemma *el-loc-ok-blocks1* [*simp*]:
el-loc-ok (*blocks1 n Ts body*) *xs* = *el-loc-ok body xs*
by(*induct n Ts body rule*: *blocks1.induct*) *auto*

lemma *sync-oks-blocks1* [*simp*]: *sync-ok* (*blocks1 n Ts e*) = *sync-ok e*
by(*induct n Ts e rule*: *blocks1.induct*) *auto*

lemma assumes *fin*: *final e′*
shows *el-loc-ok-inline-call*: *el-loc-ok e xs* ⟹ *el-loc-ok* (*inline-call e′ e*) *xs*
and *els-loc-ok-inline-calls*: *els-loc-ok es xs* ⟹ *els-loc-ok* (*inline-calls e′ es*) *xs*
apply(*induct e* **and** *es arbitrary*: *xs* **and** *xs rule*: *el-loc-ok.induct els-loc-ok.induct*)
apply(*insert fin*)
apply(*auto simp add*: *unmod-inline-call*)
done

lemma assumes *sync-ok e′*
shows *sync-ok-inline-call*: *sync-ok e* ⟹ *sync-ok* (*inline-call e′ e*)
and *sync-oks-inline-calls*: *sync-oks es* ⟹ *sync-oks* (*inline-calls e′ es*)
apply(*induct e* **and** *es rule*: *sync-ok.induct sync-oks.induct*)
apply(*insert* ‹*sync-ok e′*›)
apply *auto*
done

lemma *bisim-sync-ok*:
bisim Vs e e′ xs ⟹ *sync-ok e*
bisim Vs e e′ xs ⟹ *sync-ok e′*

and *bisims-sync-oks*:
bisims Vs es es′ xs ⟹ *sync-oks es*
bisims Vs es es′ xs ⟹ *sync-oks es′*
apply(*induct rule*: *bisim-bisims.inducts*)
apply(*auto intro*: *not-contains-insync-sync-ok not-contains-insyncs-sync-oks simp del*: *compEs1-conv-map*)
done

lemma assumes *final e′*
shows *expr-locks-inline-call-final*:
expr-locks (*inline-call e′ e*) = *expr-locks e*
and *expr-lockss-inline-calls-final*:
expr-lockss (*inline-calls e′ es*) = *expr-lockss es*
apply(*induct e* **and** *es rule*: *expr-locks.induct expr-lockss.induct*)
apply(*insert* ‹*final e′*›)
apply(*auto simp add*: *is-vals-conv intro*: *ext*)
done

lemma *lock-oks1I*:
⟦ ⋀*t l*. *ts t* = *None* ⟹ *has-locks* (*ls* $ *l*) *t* = *0*;
⋀*t e x exs ln l*. *ts t* = ⌊(((*e*, *x*), *exs*), *ln*)⌋ ⟹ *has-locks* (*ls* $ *l*) *t* + *ln* $ *l*= *expr-locks e l* +

expr-lockss (*map fst exs*) *l* ⟧
⟹ *lock-oks1 ls ts*
apply(*fastforce simp add*: *lock-oks1-def*)
done

lemma *lock-oks1E*:
⟦ *lock-oks1 ls ts*;
∀ *t*. *ts t* = *None* ⟶ (∀ *l*. *has-locks* (*ls* $ *l*) *t* = *0*) ⟹ *Q*;
∀ *t e x exs ln*. *ts t* = ⌊(((*e*, *x*), *exs*), *ln*)⌋ ⟶ (∀ *l*. *has-locks* (*ls* $ *l*) *t* + *ln* $ *l* = *expr-locks e l* + *expr-lockss* (*map fst exs*) *l*) ⟹ *Q* ⟧
⟹ *Q*
by(*fastforce simp add*: *lock-oks1-def*)

lemma *lock-oks1D1*:
⟦ *lock-oks1 ls ts*; *ts t* = *None* ⟧ ⟹ ∀ *l*. *has-locks* (*ls* $ *l*) *t* = *0*
apply(*simp add*: *lock-oks1-def*)
apply(*erule-tac x*=*t* **in** *allE*)
apply(*auto*)
done

lemma *lock-oks1D2*:
⋀*ln*. ⟦ *lock-oks1 ls ts*; *ts t* = ⌊(((*e*, *x*), *exs*), *ln*)⌋ ⟧
⟹ ∀ *l*. *has-locks* (*ls* $ *l*) *t* + *ln* $ *l* = *expr-locks e l* + *expr-lockss* (*map fst exs*) *l*
apply(*fastforce simp add*: *lock-oks1-def*)
done

lemma *lock-oks1-thr-updI*:
⋀*ln*. ⟦ *lock-oks1 ls ts*; *ts t* = ⌊(((*e*, *xs*), *exs*), *ln*)⌋;
∀ *l*. *expr-locks e l* + *expr-lockss* (*map fst exs*) *l* = *expr-locks e′ l* + *expr-lockss* (*map fst exs′*) *l* ⟧
⟹ *lock-oks1 ls* (*ts*(*t* ↦ (((*e′*, *xs′*), *exs′*), *ln*)))
by(*rule lock-oks1I*)(*auto split*: *if-split-asm dest*: *lock-oks1D2 lock-oks1D1*)

definition *mbisim-Red1′-Red1* ::
(('*addr*,'*thread-id*,('*addr expr1* × '*addr locals1*) × ('*addr expr1* × '*addr locals1*) *list*,'*heap*,'*addr*) *state*,
('*addr*,'*thread-id*,('*addr expr1* × '*addr locals1*) × ('*addr expr1* × '*addr locals1*) *list*,'*heap*,'*addr*) *state*) *bisim*
where
mbisim-Red1′-Red1 s1 s2 =
(*s1* = *s2* ∧ *lock-oks1* (*locks s1*) (*thr s1*) ∧ *ts-ok* (λ*t exexs h*. *el-loc-ok1 exexs*) (*thr s1*) (*shr s1*))

lemma *sync-ok-blocks*:
⟦ *length vs* = *length pns*; *length Ts* = *length pns*⟧
⟹ *sync-ok* (*blocks pns Ts vs body*) = *sync-ok body*
by(*induct pns Ts vs body rule*: *blocks.induct*) *auto*

context *J1-heap-base* **begin**

lemma *red1-True-into-red1-False*:
⟦ *True*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩; *el-loc-ok e* (*lcl s*) ⟧
⟹ *False*,*P*,*t* ⊢1 ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩ ∨ (∃ *l*. *ta* = {⌈*UnlockFail*→*l*⌉} ∧ *expr-locks e l* > *0*)
and *reds1-True-into-reds1-False*:
⟦ *True*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es′*, *s′*⟩; *els-loc-ok es* (*lcl s*) ⟧

$\Longrightarrow$ *False*,*P*,*t* $\vdash$1 $\langle es,\ s\rangle$ $[-ta\rightarrow]$ $\langle es',\ s'\rangle$ $\vee$ $(\exists\, l.\ ta = \{\!|UnlockFail{\rightarrow}l|\!\} \wedge$ *expr-lockss es l* $> 0)$
apply(*induct rule*: *red1-reds1.inducts*)
apply(*auto intro*: *red1-reds1.intros split*: *if-split-asm*)
done

lemma *Red1-True-into-Red1-False*:
assumes *True*,*P*,*t* $\vdash$1 $\langle ex/exs,shr\ s\rangle$ $-ta\rightarrow$ $\langle ex'/exs',m'\rangle$
and *el-loc-ok1* $(ex,\ exs)$
shows *False*,*P*,*t* $\vdash$1 $\langle ex/exs,shr\ s\rangle$ $-ta\rightarrow$ $\langle ex'/exs',m'\rangle$ $\vee$
$(\exists\, l.\ ta = \{\!|UnlockFail{\rightarrow}l|\!\} \wedge$ *expr-lockss* $(fst\ ex\ \#\ map\ fst\ exs)\ l > 0)$
using *assms*
by(*cases*)(*auto dest*: *Red1.intros red1-True-into-red1-False simp add*: *el-loc-ok1-def ta-upd-simps*)

lemma shows *red1-preserves-el-loc-ok*:
$[\![$ *uf*,*P*,*t* $\vdash$1 $\langle e,\ s\rangle$ $-ta\rightarrow$ $\langle e',\ s'\rangle$; *sync-ok e*; *el-loc-ok e* $(lcl\ s)$ $]\!] \Longrightarrow$ *el-loc-ok e'* $(lcl\ s')$

and *reds1-preserves-els-loc-ok*:
$[\![$ *uf*,*P*,*t* $\vdash$1 $\langle es,\ s\rangle$ $[-ta\rightarrow]$ $\langle es',\ s'\rangle$; *sync-oks es*; *els-loc-ok es* $(lcl\ s)$ $]\!] \Longrightarrow$ *els-loc-ok es'* $(lcl\ s')$
proof(*induct rule*: *red1-reds1.inducts*)
case (*Synchronized1Red2 e s ta e' s' V a*)
from ‹*el-loc-ok* (*insync*$_V$ (a) $e)$ $(lcl\ s)$›
have *el-loc-ok e* $(lcl\ s)$ *unmod e V lcl s* ! *V* = *Addr a* **by** *auto*
from ‹*sync-ok* (*insync*$_V$ (a) $e)$› **have** *sync-ok e* **by** *simp*
hence *el-loc-ok e'* $(lcl\ s')$
using ‹*el-loc-ok e* $(lcl\ s)$›
by(*rule Synchronized1Red2*)
moreover from ‹*uf*,*P*,*t* $\vdash$1 $\langle e,s\rangle$ $-ta\rightarrow$ $\langle e',s'\rangle$› ‹*unmod e V*› **have** *unmod e' V*
by(*rule red1-unmod-preserved*)
moreover from *red1-preserves-unmod*[*OF* ‹*uf*,*P*,*t* $\vdash$1 $\langle e,s\rangle$ $-ta\rightarrow$ $\langle e',s'\rangle$› ‹*unmod e V*›] ‹*lcl s* ! *V* = *Addr a*›
have *lcl s'* ! *V* = *Addr a* **by** *simp*
ultimately show *?case* **by** *auto*
qed(*auto elim*: *el-loc-ok-not-contains-insync-local-change els-loc-ok-not-contains-insyncs-local-change*)

lemma *red1-preserves-sync-ok*: $[\![$ *uf*,*P*,*t* $\vdash$1 $\langle e,\ s\rangle$ $-ta\rightarrow$ $\langle e',\ s'\rangle$; *sync-ok e* $]\!] \Longrightarrow$ *sync-ok e'*
and *reds1-preserves-sync-oks*: $[\![$ *uf*,*P*,*t* $\vdash$1 $\langle es,\ s\rangle$ $[-ta\rightarrow]$ $\langle es',\ s'\rangle$; *sync-oks es* $]\!] \Longrightarrow$ *sync-oks es'*
by(*induct rule*: *red1-reds1.inducts*)(*auto elim*: *not-contains-insync-sync-ok*)

lemma *Red1-preserves-el-loc-ok1*:
assumes *wf*: *wf-J1-prog P*
shows $[\![$ *uf*,*P*,*t* $\vdash$1 $\langle ex/exs,m\rangle$ $-ta\rightarrow$ $\langle ex'/exs',m'\rangle$; *el-loc-ok1* $(ex,\ exs)$ $]\!] \Longrightarrow$ *el-loc-ok1* $(ex',\ exs')$
apply(*erule Red1.cases*)
apply(*auto simp add*: *el-loc-ok1-def dest*: *red1-preserves-el-loc-ok red1-preserves-sync-ok intro*: *el-loc-ok-inline-call*
sync-ok-inline-call)
apply(*fastforce dest*!: *sees-wf-mdecl*[*OF wf*] *simp add*: *wf-mdecl-def intro*!: *el-loc-okI dest*: *WT1-not-contains-insync*
intro: *not-contains-insync-sync-ok*)+
done

lemma assumes *wf*: *wf-J1-prog P*
shows *red1-el-loc-ok1-new-thread*:
$[\![$ *uf*,*P*,*t* $\vdash$1 $\langle e,\ s\rangle$ $-ta\rightarrow$ $\langle e',\ s'\rangle$; *NewThread t'* $(C,\ M,\ a)$ $h \in set\ \{\!|ta|\!\}_t$ $]\!]$
$\Longrightarrow$ *el-loc-ok1* $((\{0{:}Class\ (fst\ (method\ P\ C\ M)){=}None;\ the\ (snd\ (snd\ (snd\ (method\ P\ C\ M))))\},$
xs), $[])$

and *reds1-el-loc-ok1-new-thread*:
⟦ *uf*,*P*,*t* ⊢1 ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *NewThread t'* (*C*, *M*, *a*) *h* ∈ *set* {|*ta*|}*t* ⟧
⟹ *el-loc-ok1* (({*0*:*Class* (*fst* (*method P C M*))=*None*; *the* (*snd* (*snd* (*snd* (*method P C M*))))}, *xs*), [])
proof(*induct rule*: *red1-reds1.inducts*)
case *Red1CallExternal* **thus** *?case*
apply(*auto dest*!: *red-external-new-thread-sees*[*OF wf*] *simp add*: *el-loc-ok1-simps*)
apply(*auto dest*!: *sees-wf-mdecl*[*OF wf*] *WT1-not-contains-insync simp add*: *wf-mdecl-def intro*!: *el-loc-okI not-contains-insync-sync-ok*)
done
qed *auto*

lemma *Red1-el-loc-ok1-new-thread*:
assumes *wf*: *wf-J1-prog P*
shows ⟦ *uf*,*P*,*t* ⊢1 ⟨*ex*/*exs*,*m*⟩ −*ta*→ ⟨*ex'*/*exs'*,*m'*⟩; *NewThread t' exexs m'* ∈ *set* {|*ta*|}*t* ⟧
⟹ *el-loc-ok1 exexs*
by(*erule Red1.cases*)(*fastforce elim*: *red1-el-loc-ok1-new-thread*[*OF wf*] *simp add*: *ta-upd-simps*)+

lemma *Red1-el-loc-ok*:
assumes *wf*: *wf-J1-prog P*
shows *lifting-wf final-expr1* (*mred1g uf P*) (λ*t exexs h*. *el-loc-ok1 exexs*)
by(*unfold-locales*)(*auto elim*: *Red1-preserves-el-loc-ok1*[*OF wf*] *Red1-el-loc-ok1-new-thread*[*OF wf*])

lemma *mred1-eq-mred1'*:
assumes *lok*: *lock-oks1* (*locks s*) (*thr s*)
and *elo*: *ts-ok* (λ*t exexs h*. *el-loc-ok1 exexs*) (*thr s*) (*shr s*)
and *tst*: *thr s t* = ⌊(*exexs*, *no-wait-locks*)⌋
and *aoe*: *Red1-mthr.actions-ok s t ta*
shows *mred1 P t* (*exexs*, *shr s*) *ta* = *mred1' P t* (*exexs*, *shr s*) *ta*
proof(*intro ext iffI*)
fix *xm'*
assume *mred1 P t* (*exexs*, *shr s*) *ta xm'*
moreover obtain *ex exs* **where** *exexs* [*simp*]: *exexs* = (*ex*, *exs*) **by**(*cases exexs*)
moreover obtain *ex' exs' m'* **where** *xm'* [*simp*]: *xm'* = ((*ex'*, *exs'*), *m'*) **by**(*cases xm'*) *auto*
ultimately have *red*: *True*,*P*,*t* ⊢1 ⟨*ex*/*exs*,*shr s*⟩ −*ta*→ ⟨*ex'*/*exs'*,*m'*⟩ **by** *simp*
from *elo tst* **have** *el-loc-ok1* (*ex*, *exs*) **by**(*auto dest*: *ts-okD*)
from *Red1-True-into-Red1-False*[*OF red this*]
have *False*,*P*,*t* ⊢1 ⟨*ex*/*exs*,*shr s*⟩ −*ta*→ ⟨*ex'*/*exs'*,*m'*⟩
proof
assume ∃*l*. *ta* = {|*UnlockFail*→*l*|} ∧ 0 < *expr-lockss* (*fst ex* # *map fst exs*) *l*
then obtain *l* **where** *ta*: *ta* = {|*UnlockFail*→*l*|}
and *el*: *expr-lockss* (*fst ex* # *map fst exs*) *l* > *0* **by** *blast*
from *aoe* **have** *lock-actions-ok* (*locks s* $ *l*) *t* ({|*ta*|}*l* $ *l*)
by(*auto simp add*: *lock-ok-las-def*)
with *ta* **have** *has-locks* (*locks s* $ *l*) *t* = *0* **by** *simp*
with *lok tst* **have** *expr-lockss* (*map fst* (*ex* # *exs*)) *l* = *0*
by(*cases ex*)(*auto 4 6 simp add*: *lock-oks1-def*)
with *el* **have** *False* **by** *simp*
thus *?thesis* **..**
qed
thus *mred1' P t* (*exexs*, *shr s*) *ta xm'* **by** *simp*
next
fix *xm'*
assume *mred1' P t* (*exexs*, *shr s*) *ta xm'*

thus *mred1 P t (exexs, shr s) ta xm'*
by(*cases xm'*)(*auto simp add*: *split-beta intro*: *Red1-False-into-Red1-True*)
qed

lemma *Red1-mthr-eq-Red1-mthr'*:
assumes *lok*: *lock-oks1 (locks s) (thr s)*
and *elo*: *ts-ok (λt exexs h. el-loc-ok1 exexs) (thr s) (shr s)*
shows *Red1-mthr.redT True P s = Red1-mthr.redT False P s*
proof(*intro ext*)
fix *tta s'*
show *Red1-mthr.redT True P s tta s' = Red1-mthr.redT False P s tta s'* (**is** *?lhs = ?rhs*)
proof
assume *?lhs* **thus** *?rhs*
proof *cases*
case (*redT-normal t x ta x' m'*)
from ‹*mred1 P t (x, shr s) ta (x', m')*› **have** *mred1' P t (x, shr s) ta (x', m')*
unfolding *mred1-eq-mred1'*[*OF lok elo* ‹*thr s t = ⌊(x, no-wait-locks)⌋*› ‹*Red1-mthr.actions-ok s t ta*›] **.**
thus *?thesis* **using** *redT-normal*(*3−*) **unfolding** ‹*tta = (t, ta)*› **..**
next
case (*redT-acquire t x ln n*)
from *this*(*2−*) **show** *?thesis* **unfolding** *redT-acquire*(*1*) **..**
qed
next
assume *?rhs* **thus** *?lhs*
proof(*cases*)
case (*redT-normal t x ta x' m'*)
from ‹*mred1' P t (x, shr s) ta (x', m')*› **have** *mred1 P t (x, shr s) ta (x', m')*
unfolding *mred1-eq-mred1'*[*OF lok elo* ‹*thr s t = ⌊(x, no-wait-locks)⌋*› ‹*Red1-mthr.actions-ok s t ta*›] **.**
thus *?thesis* **using** *redT-normal*(*3−*) **unfolding** ‹*tta = (t, ta)*› **..**
next
case (*redT-acquire t x ln n*)
from *this*(*2−*) **show** *?thesis* **unfolding** *redT-acquire*(*1*) **..**
qed
qed
qed

lemma assumes *wf*: *wf-J1-prog P*
shows *expr-locks-new-thread1*:
⟦ *uf*,*P*,*t* ⊢1 ⟨*e*,*s*⟩ −*TA*→ ⟨*e'*,*s'*⟩; *NewThread t' (ex, exs) h* ∈ *set* (*map* (*convert-new-thread-action* (*extNTA2J1 P*)) {|*TA*|}*t*) ⟧
⟹ *expr-lockss* (*map fst* (*ex* # *exs*)) = (λ*ad*. *0*)
and *expr-lockss-new-thread1*:
⟦ *uf*,*P*,*t* ⊢1 ⟨*es*,*s*⟩ [−*TA*→] ⟨*es'*,*s'*⟩; *NewThread t' (ex, exs) h* ∈ *set* (*map* (*convert-new-thread-action* (*extNTA2J1 P*)) {|*TA*|}*t*) ⟧
⟹ *expr-lockss* (*map fst* (*ex* # *exs*)) = (λ*ad*. *0*)
proof(*induct rule*: *red1-reds1.inducts*)
case (*Red1CallExternal s a T M vs ta va h' e' s'*)
then obtain *C fs ad* **where** *subThread*: *P* ⊢ *C* ⪯* *Thread* **and** *ext*: *extNTA2J1 P* (*C*, *run*, *ad*) = (*ex*, *exs*)
by(*fastforce dest*: *red-external-new-thread-sub-thread*)
from *sub-Thread-sees-run*[*OF wf subThread*] **obtain** *D body*
where *sees*: *P* ⊢ *C sees run*: []→*Void* = ⌊*body*⌋ *in D* **by** *auto*

```
  from sees-wf-mdecl[OF wf this] obtain T where P,[Class D] ⊢1 body :: T
    by(auto simp add: wf-mdecl-def)
  hence ¬ contains-insync body by(rule WT1-not-contains-insync)
  hence expr-locks body = (λad. 0) by(auto simp add: contains-insync-conv fun-eq-iff)
  with sees ext show ?case by(auto)
qed auto

lemma assumes wf: wf-J1-prog P
  shows red1-update-expr-locks:
  ⟦ False,P,t ⊢1 ⟨e, s⟩ −ta→ ⟨e', s'⟩; sync-ok e; el-loc-ok e (lcl s) ⟧
  ⟹ upd-expr-locks (int o expr-locks e) {|ta|}l = int o expr-locks e'

  and reds1-update-expr-lockss:
  ⟦ False,P,t ⊢1 ⟨es, s⟩ [−ta→] ⟨es', s'⟩; sync-oks es; els-loc-ok es (lcl s) ⟧
  ⟹ upd-expr-locks (int o expr-lockss es) {|ta|}l = int o expr-lockss es'
proof −
  have ⟦ False,P,t ⊢1 ⟨e, s⟩ −ta→ ⟨e', s'⟩; sync-ok e; el-loc-ok e (lcl s) ⟧
       ⟹ upd-expr-locks (λad. 0) {|ta|}l = (λad. (int o expr-locks e') ad − (int o expr-locks e) ad)
    and ⟦ False,P,t ⊢1 ⟨es, s⟩ [−ta→] ⟨es', s'⟩; sync-oks es; els-loc-ok es (lcl s) ⟧
       ⟹ upd-expr-locks (λad. 0) {|ta|}l = (λad. (int o expr-lockss es') ad − (int o expr-lockss es) ad)
  proof(induct rule: red1-reds1.inducts)
    case Red1CallExternal thus ?case
      by(auto simp add: fun-eq-iff contains-insync-conv contains-insyncs-conv finfun-upd-apply elim!:
red-external.cases)
  qed(fastforce simp add: fun-eq-iff contains-insync-conv contains-insyncs-conv finfun-upd-apply)+
  hence ⟦ False,P,t ⊢1 ⟨e, s⟩ −ta→ ⟨e', s'⟩; sync-ok e; el-loc-ok e (lcl s) ⟧
       ⟹ upd-expr-locks (λad. 0 + (int ∘ expr-locks e) ad) {|ta|}l = int ∘ expr-locks e'
    and ⟦ False,P,t ⊢1 ⟨es, s⟩ [−ta→] ⟨es', s'⟩; sync-oks es; els-loc-ok es (lcl s) ⟧
       ⟹ upd-expr-locks (λad. 0 + (int ∘ expr-lockss es) ad) {|ta|}l = int ∘ expr-lockss es'
    by(fastforce simp only: upd-expr-locks-add)+
  thus ⟦ False,P,t ⊢1 ⟨e, s⟩ −ta→ ⟨e', s'⟩; sync-ok e; el-loc-ok e (lcl s) ⟧
       ⟹ upd-expr-locks (int o expr-locks e) {|ta|}l = int o expr-locks e'
    and ⟦ False,P,t ⊢1 ⟨es, s⟩ [−ta→] ⟨es', s'⟩; sync-oks es; els-loc-ok es (lcl s) ⟧
       ⟹ upd-expr-locks (int o expr-lockss es) {|ta|}l = int o expr-lockss es'
    by(auto simp add: o-def)
qed

lemma Red1'-preserves-lock-oks:
  assumes wf: wf-J1-prog P
  and Red: Red1-mthr.redT False P s1 ta1 s1'
  and loks: lock-oks1 (locks s1) (thr s1)
  and sync: ts-ok (λt exexs h. el-loc-ok1 exexs) (thr s1) (shr s1)
  shows lock-oks1 (locks s1') (thr s1')
using Red
proof(cases rule: Red1-mthr.redT.cases)
  case (redT-normal t x ta x' m')
  note [simp] = ‹ta1 = (t, ta)›
  obtain ex exs where x: x = (ex, exs) by (cases x)
  obtain ex' exs' where x': x' = (ex', exs') by (cases x')
  note thrst = ‹thr s1 t = ⌊(x, no-wait-locks)⌋›
  note aoe = ‹Red1-mthr.actions-ok s1 t ta›
  from ‹mred1' P t (x, shr s1) ta (x', m')›
  have red: False,P,t ⊢1 ⟨ex/exs,shr s1⟩ −ta→ ⟨ex'/exs',m'⟩
    unfolding x x' by simp-all
```

note *s1'* = ‹*redT-upd s1 t ta x' m' s1'*›
moreover from *red*
have *lock-oks1* (*locks s1'*) (*thr s1'*)
proof *cases*
case (*red1Red e x TA e' x'*)
note [*simp*] = ‹*ex* = (*e*, *x*)› ‹*ta* = *extTA2J1 P TA*› ‹*ex'* = (*e'*, *x'*)› ‹*exs'* = *exs*›
and *red* = ‹*False*,*P*,*t* ⊢*1* ⟨*e*,(*shr s1*, *x*)⟩ −*TA*→ ⟨*e'*,(*m'*, *x'*)⟩›
{ fix *t'*
assume *None*: ((*redT-updTs* (*thr s1*) (*map* (*convert-new-thread-action* (*extNTA2J1 P*)) {|*TA*|}*t*))(*t* ↦ (((*e'*, *x'*), *exs*), *redT-updLns* (*locks s1*) *t* (*snd* (*the* (*thr s1 t*))) {|*TA*|}*l*))) *t'* = *None*
{ fix *l*
from *aoe* **have** *lock-actions-ok* (*locks s1* $ *l*) *t* ({|*ta*|}*l* $ *l*) **by**(*auto simp add*: *lock-ok-las-def*)
with *None* **have** *has-locks* ((*redT-updLs* (*locks s1*) *t* {|*ta*|}*l*) $ *l*) *t'* = *has-locks* (*locks s1* $ *l*) *t'*
by(*auto split*: *if-split-asm*)
also from *loks None* **have** *has-locks* (*locks s1* $ *l*) *t'* = *0* **unfolding** *lock-oks1-def*
by(*force split*: *if-split-asm dest*!: *redT-updTs-None*)
finally have *has-locks* (*upd-locks* (*locks s1* $ *l*) *t* ({|*TA*|}*l* $ *l*)) *t'* = *0* **by** *simp* **}**
hence ∀ *l*. *has-locks* (*upd-locks* (*locks s1* $ *l*) *t* ({|*TA*|}*l* $ *l*)) *t'* = *0* **.. }**
moreover {
fix *t' eX eXS LN*
assume *Some*: ((*redT-updTs* (*thr s1*) (*map* (*convert-new-thread-action* (*extNTA2J1 P*)) {|*TA*|}*t*))(*t* ↦ (((*e'*, *x'*), *exs*), *redT-updLns* (*locks s1*) *t* (*snd* (*the* (*thr s1 t*))) {|*TA*|}*l*))) *t'* = ⌊((*eX*, *eXS*), *LN*)⌋
{ fix *l*
from *aoe* **have** *lao*: *lock-actions-ok* (*locks s1* $ *l*) *t* ({|*ta*|}*l* $ *l*) **by**(*auto simp add*: *lock-ok-las-def*)
have *has-locks* ((*redT-updLs* (*locks s1*) *t* {|*ta*|}*l*) $ *l*) *t'* + *LN* $ *l* = *expr-lockss* (*map fst* (*eX* # *eXS*)) *l*
proof(*cases t* = *t'*)
case *True*
from *loks thrst x*
have *has-locks* (*locks s1* $ *l*) *t* = *expr-locks e l* + *expr-lockss* (*map fst exs*) *l*
by(*force simp add*: *lock-oks1-def*)
hence *lock-expr-locks-ok* (*locks s1* $ *l*) *t 0* (*int* (*expr-locks e l* + *expr-lockss* (*map fst exs*) *l*))
by(*simp add*: *lock-expr-locks-ok-def*)
with *lao* **have** *lock-expr-locks-ok* (*upd-locks* (*locks s1* $ *l*) *t* ({|*ta*|}*l* $ *l*)) *t* (*upd-threadRs 0* (*locks s1* $ *l*) *t* ({|*ta*|}*l* $ *l*))
(*upd-expr-lock-actions* (*int* (*expr-locks e l* + *expr-lockss* (*map fst exs*) *l*)) ({|*ta*|}*l* $ *l*))
by(*rule upd-locks-upd-expr-lock-preserve-lock-expr-locks-ok*)
moreover from *sync thrst x* **have** *sync-ok e el-loc-ok e x*
unfolding *el-loc-ok1-def* **by**(*auto dest*: *ts-okD*)
with *red1-update-expr-locks*[*OF wf red*]
have *upd-expr-locks* (*int* ∘ *expr-locks e*) {|*TA*|}*l* = *int* ∘ *expr-locks e'* **by**(*simp*)
hence *upd-expr-lock-actions* (*int* (*expr-locks e l*)) ({|*TA*|}*l* $ *l*) = *int* (*expr-locks e' l*)
by(*simp add*: *upd-expr-locks-def fun-eq-iff*)
ultimately show *?thesis* **using** *lao Some thrst x True*
by(*auto simp add*: *lock-expr-locks-ok-def upd-expr-locks-def*)
next
case *False*
from *aoe* **have** *tok*: *thread-oks* (*thr s1*) {|*ta*|}*t* **by** *auto*
show *?thesis*
proof(*cases thr s1 t'* = *None*)
case *True*
with *Some tok False* **obtain** *m*
where *nt*: *NewThread t'* (*eX*, *eXS*) *m* ∈ *set* (*map* (*convert-new-thread-action* (*extNTA2J1 P*)) {|*TA*|}*t*)

```
          and [simp]: LN = no-wait-locks by(auto dest: redT-updTs-new-thread)
        note expr-locks-new-thread1[OF wf red nt]
        moreover from loks True have has-locks (locks s1 $ l) t' = 0
          by(force simp add: lock-oks1-def)
        ultimately show ?thesis using lao False by simp
      next
        case False
        with Some ‹t ≠ t'› tok
        have thr s1 t' = ⌊((eX, eXS), LN)⌋ by(fastforce dest: redT-updTs-Some[OF - tok])
        with loks tok lao ‹t ≠ t'› show ?thesis by(cases eX)(auto simp add: lock-oks1-def)
      qed
    qed }
  hence ∀ l. has-locks ((redT-updLs (locks s1) t {|ta|}_l) $ l) t' + LN $ l = expr-lockss (map fst (eX
# eXS)) l .. }
 ultimately show ?thesis using s1' unfolding lock-oks1-def x' by(clarsimp simp del: fun-upd-apply)
next
  case (red1Call e a M vs U Ts T body D x)
  from wf ‹P ⊢ class-type-of U sees M: Ts→T = ⌊body⌋ in D›
  obtain T' where P,Class D # Ts ⊢1 body :: T'
    by(auto simp add: wf-mdecl-def dest!: sees-wf-mdecl)
  hence expr-locks (blocks1 0 (Class D#Ts) body) = (λl. 0)
   by(auto simp add: expr-locks-blocks1 contains-insync-conv fun-eq-iff dest!: WT1-not-contains-insync)
  thus ?thesis using red1Call thrst loks s1'
    unfolding lock-oks1-def x' x
    by auto force+
next
  case (red1Return e' x' e x)
  thus ?thesis using thrst loks s1'
    unfolding lock-oks1-def x' x
    apply(auto simp add: redT-updWs-def elim!: rtrancl3p-cases)
     apply(erule-tac x=t in allE)
     apply(erule conjE)
     apply(erule disjE)
      apply(force simp add: expr-locks-inline-call-final ac-simps)
     apply(fastforce simp add: expr-locks-inline-call-final)
    apply hypsubst-thin
    apply(erule-tac x=ta in allE)
    apply fastforce
    done
qed
moreover from sync ‹mred1' P t (x, shr s1) ta (x', m')› thrst aoe s1'
have ts-ok (λt exexs h. el-loc-ok1 exexs) (thr s1') (shr s1')
  by(auto intro: lifting-wf.redT-updTs-preserves[OF Red1-el-loc-ok[OF wf]])
ultimately show ?thesis by simp
next
  case (redT-acquire t x n ln)
  thus ?thesis using loks unfolding lock-oks1-def
    apply auto
     apply force
    apply(case-tac ln $ l::nat)
     apply simp
     apply(erule allE)
     apply(erule conjE)
     apply(erule allE)+
```

apply(*erule* (*1*) *impE*)
apply(*erule-tac* *x*=*l* **in** *allE*)
apply *fastforce*
apply(*erule may-acquire-allE*)
apply(*erule allE*)
apply(*erule-tac* *x*=*l* **in** *allE*)
apply(*erule impE*)
apply *simp*
apply(*simp only*: *has-locks-acquire-locks-conv*)
apply(*erule conjE*)
apply(*erule allE*)+
apply(*erule* (*1*) *impE*)
apply(*erule-tac* *x*=*l* **in** *allE*)
apply *simp*
done
qed

lemma *Red1'-Red1-bisimulation*:
assumes *wf*: *wf-J1-prog P*
shows *bisimulation* (*Red1-mthr.redT False P*) (*Red1-mthr.redT True P*) *mbisim-Red1'-Red1* (=)
proof
fix *s1 s2 tl1 s1'*
assume *mbisim-Red1'-Red1 s1 s2* **and** *Red1-mthr.redT False P s1 tl1 s1'*
thus ∃ *s2' tl2*. *Red1-mthr.redT True P s2 tl2 s2'* ∧ *mbisim-Red1'-Red1 s1' s2'* ∧ *tl1* = *tl2*
by(*cases tl1*)(*auto simp add*: *mbisim-Red1'-Red1-def Red1-mthr-eq-Red1-mthr' simp del*: *split-paired-Ex elim*: *Red1'-preserves-lock-oks*[*OF wf*] *lifting-wf.redT-preserves*[*OF Red1-el-loc-ok*, *OF wf*])
next
fix *s1 s2 tl2 s2'*
assume *mbisim-Red1'-Red1 s1 s2 Red1-mthr.redT True P s2 tl2 s2'*
thus ∃ *s1' tl1*. *Red1-mthr.redT False P s1 tl1 s1'* ∧ *mbisim-Red1'-Red1 s1' s2'* ∧ *tl1* = *tl2*
by(*cases tl2*)(*auto simp add*: *mbisim-Red1'-Red1-def Red1-mthr-eq-Red1-mthr' simp del*: *split-paired-Ex elim*: *Red1'-preserves-lock-oks*[*OF wf*] *lifting-wf.redT-preserves*[*OF Red1-el-loc-ok*, *OF wf*])
qed

lemma *Red1'-Red1-bisimulation-final*:
wf-J1-prog P
⟹ *bisimulation-final* (*Red1-mthr.redT False P*) (*Red1-mthr.redT True P*)
mbisim-Red1'-Red1 (=) *Red1-mthr.mfinal Red1-mthr.mfinal*
apply(*intro-locales*)
apply(*erule Red1'-Red1-bisimulation*)
apply(*unfold-locales*)
apply(*auto simp add*: *mbisim-Red1'-Red1-def*)
done

lemma *bisim-J1-J1-start*:
assumes *wf*: *wf-J1-prog P*
and *wf-start*: *wf-start-state P C M vs*
shows *mbisim-Red1'-Red1* (*J1-start-state P C M vs*) (*J1-start-state P C M vs*)
proof −
from *wf-start* **obtain** *Ts T body D*
where *sees*: *P* ⊢ *C sees M*:*Ts*→*T*=⌊*body*⌋ *in D*
and *conf*: *P*,*start-heap* ⊢ *vs* [:≤] *Ts*
by *cases*
let *?e* = *blocks1 0* (*Class C*#*Ts*) *body*

let *?xs* = *Null* # *vs* @ *replicate* (*max-vars body*) *undefined-value*

from *sees-wf-mdecl*[*OF wf sees*] **obtain** *T'*
where *B*: $\mathcal{B}$ *body* (*Suc* (*length Ts*))
and *wt*: *P*,*Class D* # *Ts* ⊢*1 body* :: *T'*
and *da*: $\mathcal{D}$ *body* ⌊{..*length Ts*}⌋
and *sv*: *syncvars body*
by(*auto simp add*: *wf-mdecl-def*)

from *wt* **have** *expr-locks ?e* = (λ-. *0*) **by**(*auto intro*: *WT1-expr-locks*)
thus *?thesis* **using** *da sees sv B*
unfolding *start-state-def*
by(*fastforce simp add*: *mbisim-Red1'-Red1-def lock-oks1-def el-loc-ok1-def contains-insync-conv intro*!: *ts-okI expr-locks-sync-ok split*: *if-split-asm intro*: *el-loc-okI*)
qed

lemma *Red1'-Red1-bisim-into-weak*:
assumes *wf*: *wf-J1-prog P*
shows *bisimulation-into-delay* (*Red1-mthr.redT False P*) (*Red1-mthr.redT True P*) *mbisim-Red1'-Red1* (=) (*Red1-mthr.m*τ*move P*) (*Red1-mthr.m*τ*move P*)
proof −
interpret *b*: *bisimulation Red1-mthr.redT False P Red1-mthr.redT True P mbisim-Red1'-Red1* (=)
by(*rule Red1'-Red1-bisimulation*[*OF wf*])
show *?thesis* **by**(*unfold-locales*)(*simp add*: *mbisim-Red1'-Red1-def*)
qed

end

sublocale *J1-heap-base* < *Red1-mthr*:
*if-*τ*multithreaded-wf*
final-expr1
mred1g uf P
convert-RA
τ*MOVE1 P*
for *uf P*
by(*unfold-locales*)

context *J1-heap-base* **begin**

abbreviation *if-lock-oks1* ::
('*addr*,'*thread-id*) *locks*
⇒ ('*addr*,'*thread-id*,(*status* × (('*a*,'*b*,'*addr*) *exp* × '*c*) × (('*a*,'*b*,'*addr*) *exp* × '*c*) *list*)) *thread-info*
⇒ *bool*
where
if-lock-oks1 ls ts ≡ *lock-oks1 ls* (*init-fin-descend-thr ts*)

definition *if-mbisim-Red1'-Red1* ::
(('*addr*,'*thread-id*,*status* × (('*addr expr1* × '*addr locals1*) × ('*addr expr1* × '*addr locals1*) *list*),'*heap*,'*addr*) *state*,
('*addr*,'*thread-id*,*status* × (('*addr expr1* × '*addr locals1*) × ('*addr expr1* × '*addr locals1*) *list*),'*heap*,'*addr*) *state*) *bisim*
where
if-mbisim-Red1'-Red1 s1 s2 ⟷
s1 = *s2* ∧ *if-lock-oks1* (*locks s1*) (*thr s1*) ∧ *ts-ok* (*init-fin-lift* (λ*t exexs h*. *el-loc-ok1 exexs*)) (*thr s1*)

(*shr s1*)

lemma *if-mbisim-Red1'-Red1-imp-mbisim-Red1'-Red1*:
if-mbisim-Red1'-Red1 s1 s2 ⟹ *mbisim-Red1'-Red1* (*init-fin-descend-state s1*) (*init-fin-descend-state s2*)
by(*auto simp add*: *mbisim-Red1'-Red1-def if-mbisim-Red1'-Red1-def ts-ok-init-fin-descend-state*)

lemma *if-Red1-mthr-imp-if-Red1-mthr'*:
assumes *lok*: *if-lock-oks1* (*locks s*) (*thr s*)
and *elo*: *ts-ok* (*init-fin-lift* (λ*t exexs h*. *el-loc-ok1 exexs*)) (*thr s*) (*shr s*)
and *Red*: *Red1-mthr.if.redT uf P s tta s'*
shows *Red1-mthr.if.redT* (¬ *uf*) *P s tta s'*
using *Red*
proof(*cases*)
case (*redT-acquire t x ln n*)
from *this*(*2*−) **show** *?thesis* **unfolding** *redT-acquire*(*1*) **..**
next
case (*redT-normal t x ta x' m'*)
note *aok* = ‹*Red1-mthr.if.actions-ok s t ta*›
and *tst* = ‹*thr s t* = ⌊(*x*, *no-wait-locks*)⌋›
from ‹*Red1-mthr.init-fin uf P t* (*x*, *shr s*) *ta* (*x'*, *m'*)›
have *Red1-mthr.init-fin* (¬ *uf*) *P t* (*x*, *shr s*) *ta* (*x'*, *m'*)
proof(*cases*)
case *InitialThreadAction* **show** *?thesis* **unfolding** *InitialThreadAction* **..**
next
case (*ThreadFinishAction exexs*)
from ‹*final-expr1 exexs*› **show** *?thesis* **unfolding** *ThreadFinishAction* **..**
next
case (*NormalAction exexs ta' exexs'*)
let *?s* = *init-fin-descend-state s*

from *lok* **have** *lock-oks1* (*locks ?s*) (*thr ?s*) **by**(*simp*)
moreover from *elo* **have** *elo*: *ts-ok* (λ*t exexs h*. *el-loc-ok1 exexs*) (*thr ?s*) (*shr ?s*)
by(*simp add*: *ts-ok-init-fin-descend-state*)
moreover from *tst* ‹*x* = (*Running*, *exexs*)›
have *thr ?s t* = ⌊(*exexs*, *no-wait-locks*)⌋ **by** *simp*
moreover from *aok* **have** *Red1-mthr.actions-ok ?s t ta'*
using ‹*ta* = *convert-TA-initial* (*convert-obs-initial ta'*)› **by** *auto*
ultimately have *mred1 P t* (*exexs*, *shr ?s*) *ta'* = *mred1' P t* (*exexs*, *shr ?s*) *ta'*
by(*rule mred1-eq-mred1'*)
with ‹*mred1g uf P t* (*exexs*, *shr s*) *ta'* (*exexs'*, *m'*)›
have *mred1g* (¬ *uf*) *P t* (*exexs*, *shr s*) *ta'* (*exexs'*, *m'*)
by(*cases uf*) *simp-all*
thus *?thesis* **unfolding** *NormalAction*(*1*−*3*) **by**(*rule Red1-mthr.init-fin.NormalAction*)
qed
thus *?thesis* **using** *tst aok* ‹*redT-upd s t ta x' m' s'*› **unfolding** ‹*tta* = (*t*, *ta*)› **..**
qed

lemma *if-Red1-mthr-eq-if-Red1-mthr'*:
assumes *lok*: *if-lock-oks1* (*locks s*) (*thr s*)
and *elo*: *ts-ok* (*init-fin-lift* (λ*t exexs h*. *el-loc-ok1 exexs*)) (*thr s*) (*shr s*)
shows *Red1-mthr.if.redT True P s* = *Red1-mthr.if.redT False P s*
using *if-Red1-mthr-imp-if-Red1-mthr'*[*OF assms*, *of True P*, *simplified*]
if-Red1-mthr-imp-if-Red1-mthr'[*OF assms*, *of False P*, *simplified*]

by(*blast del*: *equalityI*)

lemma *if-Red1-el-loc-ok*:
assumes *wf*: *wf-J1-prog P*
shows *lifting-wf Red1-mthr.init-fin-final* (*Red1-mthr.init-fin uf P*) (*init-fin-lift* (λ*t exexs h. el-loc-ok1 exexs*))
by(*rule lifting-wf.lifting-wf-init-fin-lift*)(*rule Red1-el-loc-ok*[*OF wf*])

lemma *if-Red1'-preserves-if-lock-oks*:
assumes *wf*: *wf-J1-prog P*
and *Red*: *Red1-mthr.if.redT False P s1 ta1 s1'*
and *loks*: *if-lock-oks1* (*locks s1*) (*thr s1*)
and *sync*: *ts-ok* (*init-fin-lift* (λ*t exexs h. el-loc-ok1 exexs*)) (*thr s1*) (*shr s1*)
shows *if-lock-oks1* (*locks s1'*) (*thr s1'*)
proof −
let *?s1* = *init-fin-descend-state s1*
let *?s1'* = *init-fin-descend-state s1'*
from *loks* **have** *loks'*: *lock-oks1* (*locks ?s1*) (*thr ?s1*) **by** *simp*
from *sync* **have** *sync'*: *ts-ok* (λ*t exexs h. el-loc-ok1 exexs*) (*thr ?s1*) (*shr ?s1*)
by(*simp add*: *ts-ok-init-fin-descend-state*)
from *Red* **show** *?thesis*
proof(*cases*)
case (*redT-acquire t x n ln*)
hence *Red1-mthr.redT False P ?s1* (*t*, *K$* [], [], [], [], [], *convert-RA ln*) *?s1'*
by(*cases x*)(*auto intro*!: *Red1-mthr.redT.redT-acquire simp add*: *init-fin-descend-thr-def*)
with *wf* **have** *lock-oks1* (*locks ?s1'*) (*thr ?s1'*) **using** *loks' sync'* **by**(*rule Red1'-preserves-lock-oks*)
thus *?thesis* **by** *simp*
next
case (*redT-normal t sx ta sx' m'*)
note *tst* = ‹*thr s1 t* = ⌊(*sx*, *no-wait-locks*)⌋›
from ‹*Red1-mthr.init-fin False P t* (*sx*, *shr s1*) *ta* (*sx'*, *m'*)›
show *?thesis*
proof(*cases*)
case (*InitialThreadAction x*) **thus** *?thesis* **using** *redT-normal loks*
by(*cases x*)(*auto 4 3 simp add*: *init-fin-descend-thr-def redT-updLns-def expand-finfun-eq fun-eq-iff intro*: *lock-oks1-thr-updI*)
next
case (*ThreadFinishAction x*) **thus** *?thesis* **using** *redT-normal loks*
by(*cases x*)(*auto 4 3 simp add*: *init-fin-descend-thr-def redT-updLns-def expand-finfun-eq fun-eq-iff intro*: *lock-oks1-thr-updI*)
next
case (*NormalAction x ta' x'*)
note *ta* = ‹*ta* = *convert-TA-initial* (*convert-obs-initial ta'*)›
from ‹*mred1' P t* (*x*, *shr s1*) *ta'* (*x'*, *m'*)›
have *mred1' P t* (*x*, *shr ?s1*) *ta'* (*x'*, *m'*) **by** *simp*
moreover have *tst'*: *thr ?s1 t* = ⌊(*x*, *no-wait-locks*)⌋
using *tst* ‹*sx* = (*Running*, *x*)› **by** *simp*
moreover have *Red1-mthr.actions-ok ?s1 t ta'*
using *ta* ‹*Red1-mthr.if.actions-ok s1 t ta*› **by** *simp*
moreover from ‹*redT-upd s1 t ta sx' m' s1'*› *tst tst' ta* ‹*sx'* = (*Running*, *x'*)›
have *redT-upd ?s1 t ta' x' m' ?s1'* **by** *auto*
ultimately have *Red1-mthr.redT False P ?s1* (*t*, *ta'*) *?s1'* ..
with *wf* **have** *lock-oks1* (*locks ?s1'*) (*thr ?s1'*) **using** *loks' sync'* **by**(*rule Red1'-preserves-lock-oks*)
thus *?thesis* **by** *simp*

```
    qed
  qed
qed

lemma Red1'-Red1-if-bisimulation:
  assumes wf: wf-J1-prog P
  shows bisimulation (Red1-mthr.if.redT False P) (Red1-mthr.if.redT True P) if-mbisim-Red1'-Red1
(=)
proof
  fix s1 s2 tl1 s1'
  assume if-mbisim-Red1'-Red1 s1 s2 and Red1-mthr.if.redT False P s1 tl1 s1'
  thus ∃ s2' tl2. Red1-mthr.if.redT True P s2 tl2 s2' ∧ if-mbisim-Red1'-Red1 s1' s2' ∧ tl1 = tl2
    by(cases tl1)(auto simp add: if-mbisim-Red1'-Red1-def if-Red1-mthr-eq-if-Red1-mthr' simp del:
split-paired-Ex elim: if-Red1'-preserves-if-lock-oks[OF wf] lifting-wf.redT-preserves[OF if-Red1-el-loc-ok,
OF wf])
next
  fix s1 s2 tl2 s2'
  assume if-mbisim-Red1'-Red1 s1 s2 Red1-mthr.if.redT True P s2 tl2 s2'
  thus ∃ s1' tl1. Red1-mthr.if.redT False P s1 tl1 s1' ∧ if-mbisim-Red1'-Red1 s1' s2' ∧ tl1 = tl2
    by(cases tl2)(auto simp add: if-mbisim-Red1'-Red1-def if-Red1-mthr-eq-if-Red1-mthr' simp del:
split-paired-Ex elim: if-Red1'-preserves-if-lock-oks[OF wf] lifting-wf.redT-preserves[OF if-Red1-el-loc-ok,
OF wf])
qed

lemma if-bisim-J1-J1-start:
  assumes wf: wf-J1-prog P
  and wf-start: wf-start-state P C M vs
  shows if-mbisim-Red1'-Red1 (init-fin-lift-state status (J1-start-state P C M vs)) (init-fin-lift-state
status (J1-start-state P C M vs))
proof −
  from assms have mbisim-Red1'-Red1 (J1-start-state P C M vs) (J1-start-state P C M vs) by(rule
bisim-J1-J1-start)
  thus ?thesis
   by(simp add: if-mbisim-Red1'-Red1-def mbisim-Red1'-Red1-def)(simp add: init-fin-lift-state-conv-simps
init-fin-descend-thr-def thr-init-fin-list-state' o-def map-option.compositionality map-option.identity split-beta)
qed

lemma if-Red1'-Red1-bisim-into-weak:
  assumes wf: wf-J1-prog P
 shows bisimulation-into-delay (Red1-mthr.if.redT False P) (Red1-mthr.if.redT True P) if-mbisim-Red1'-Red1
(=) (Red1-mthr.if.mτmove P) (Red1-mthr.if.mτmove P)
proof −
 interpret b: bisimulation Red1-mthr.if.redT False P Red1-mthr.if.redT True P if-mbisim-Red1'-Red1
(=)
    by(rule Red1'-Red1-if-bisimulation[OF wf])
  show ?thesis by(unfold-locales)(simp add: if-mbisim-Red1'-Red1-def)
qed

lemma if-Red1'-Red1-bisimulation-final:
  wf-J1-prog P
  ⟹ bisimulation-final (Red1-mthr.if.redT False P) (Red1-mthr.if.redT True P)
      if-mbisim-Red1'-Red1 (=) Red1-mthr.if.mfinal Red1-mthr.if.mfinal
apply(intro-locales)
 apply(erule Red1'-Red1-if-bisimulation)
```

apply(*unfold-locales*)
apply(*auto simp add*: *if-mbisim-Red1'-Red1-def*)
done

end

end

7.24 Semantic Correctness of Stage 1

theory *Correctness1* **imports**
J0J1Bisim
../J/DefAssPreservation
begin

lemma *finals-map-Val* [*simp*]: *finals* (*map Val vs*)
by(*simp add*: *finals-iff*)

context *J-heap-base* **begin**

lemma *τred0r-preserves-defass*:
assumes *wf*: *wf-J-prog P*
shows ⟦ *τred0r extTA P t h* (*e*, *xs*) (*e'*, *xs'*); $\mathcal{D}$ *e* ⌊*dom xs*⌋ ⟧ ⟹ $\mathcal{D}$ *e'* ⌊*dom xs'*⌋
by(*induct rule*: *rtranclp-induct2*)(*auto dest*: *red-preserves-defass*[*OF wf*])

lemma *τred0t-preserves-defass*:
assumes *wf*: *wf-J-prog P*
shows ⟦ *τred0t extTA P t h* (*e*, *xs*) (*e'*, *xs'*); $\mathcal{D}$ *e* ⌊*dom xs*⌋ ⟧ ⟹ $\mathcal{D}$ *e'* ⌊*dom xs'*⌋
by(*rule τred0r-preserves-defass*[*OF wf*])(*rule tranclp-into-rtranclp*)

end

lemma *LAss-lem*:
⟦*x* ∈ *set xs*; *size xs* ≤ *size ys* ⟧
⟹ *m1* ⊆$_m$ *m2*(*xs*[↦]*ys*) ⟹ *m1*(*x*↦*y*) ⊆$_m$ *m2*(*xs*[↦]*ys*[*index xs x* := *y*])
apply(*simp add*:*map-le-def*)
apply(*simp add*:*fun-upds-apply index-less-aux eq-sym-conv*)
done

lemma *Block-lem*:
fixes *l* :: *'a* ⇀ *'b*
assumes *0*: *l* ⊆$_m$ [*Vs* [↦] *ls*]
and *1*: *l'* ⊆$_m$ [*Vs* [↦] *ls'*, *V*↦*v*]
and *hidden*: *V* ∈ *set Vs* ⟹ *ls* ! *index Vs V* = *ls'* ! *index Vs V*
and *size*: *size ls* = *size ls'* *size Vs* < *size ls'*
shows *l'*(*V* := *l V*) ⊆$_m$ [*Vs* [↦] *ls'*]
proof −
have *l'*(*V* := *l V*) ⊆$_m$ [*Vs* [↦] *ls'*, *V*↦*v*](*V* := *l V*)
using *1* **by**(*rule map-le-upd*)
also have ... = [*Vs* [↦] *ls'*](*V* := *l V*) **by** *simp*
also have ... ⊆$_m$ [*Vs* [↦] *ls'*]
proof (*cases l V*)
case *None* **thus** *?thesis* **by** *simp*

next
case (*Some w*)
hence [*Vs* [↦] *ls*] *V* = *Some w*
using *0* **by**(*force simp add*: *map-le-def split*:*if-splits*)
hence *VinVs*: *V* ∈ *set Vs* **and** *w*: *w* = *ls* ! *index Vs V*
using *size* **by**(*auto simp add*:*fun-upds-apply split*:*if-splits*)
hence *w* = *ls'* ! *index Vs V* **using** *hidden*[*OF VinVs*] **by** *simp*
hence [*Vs* [↦] *ls'*](*V* := *l V*) = [*Vs* [↦] *ls'*]
using *Some size VinVs* **by**(*simp add*:*index-less-aux map-upds-upd-conv-index*)
thus *?thesis* **by** *simp*
qed
finally show *?thesis* **.**
qed

7.24.1 Correctness proof

locale *J0-J1-heap-base* =
J?: *J-heap-base* +
J1?: *J1-heap-base* +
constrains *addr2thread-id* :: ('*addr* :: *addr*) ⇒ '*thread-id*
and *thread-id2addr* :: '*thread-id* ⇒ '*addr*
and *empty-heap* :: '*heap*
and *allocate* :: '*heap* ⇒ *htype* ⇒ ('*heap* × '*addr*) *set*
and *typeof-addr* :: '*heap* ⇒ '*addr* ⇀ *htype*
and *heap-read* :: '*heap* ⇒ '*addr* ⇒ *addr-loc* ⇒ '*addr val* ⇒ *bool*
and *heap-write* :: '*heap* ⇒ '*addr* ⇒ *addr-loc* ⇒ '*addr val* ⇒ '*heap* ⇒ *bool*
begin

lemma *ta-bisim01-extTA2J0-extTA2J1*:
assumes *wf*: *wf-J-prog P*
and *nt*: ⋀*n T C M a h*. ⟦ *n* < *length* {|*ta*|}$_t$; {|*ta*|}$_t$! *n* = *NewThread T* (*C*, *M*, *a*) *h* ⟧
⟹ *typeof-addr h a* = ⌊*Class-type C*⌋ ∧ (∃ *T meth D*. *P* ⊢ *C sees M*:[]→*T* =⌊*meth*⌋ *in D*)
shows *ta-bisim01* (*extTA2J0 P ta*) (*extTA2J1* (*compP1 P*) *ta*)
apply(*simp add*: *ta-bisim-def ta-upd-simps*)
apply(*auto intro*!: *list-all2-all-nthI*)
apply(*case-tac* {|*ta*|}$_t$! *n*)
apply(*auto simp add*: *bisim-red0-Red1-def*)
apply(*drule* (*1*) *nt*)
apply(*clarify*)
apply(*erule bisim-list-extTA2J0-extTA2J1*[*OF wf*, *simplified*])
done

lemma *red-external-ta-bisim01*:
⟦ *wf-J-prog P*; *P*,*t* ⊢ ⟨*a*·*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h'*⟩ ⟧ ⟹ *ta-bisim01* (*extTA2J0 P ta*) (*extTA2J1* (*compP1 P*) *ta*)
apply(*rule ta-bisim01-extTA2J0-extTA2J1*, *assumption*)
apply(*drule* (*1*) *red-external-new-thread-sees*, *auto simp add*: *in-set-conv-nth*)
apply(*drule red-ext-new-thread-heap*, *auto simp add*: *in-set-conv-nth*)
done

lemmas *τred1t-expr* =
NewArray-τred1t-xt Cast-τred1t-xt InstanceOf-τred1t-xt BinOp-τred1t-xt1 BinOp-τred1t-xt2 LAss-τred1t
AAcc-τred1t-xt1 AAcc-τred1t-xt2 AAss-τred1t-xt1 AAss-τred1t-xt2 AAss-τred1t-xt3
ALength-τred1t-xt FAcc-τred1t-xt FAss-τred1t-xt1 FAss-τred1t-xt2

CAS-τred1t-xt1 CAS-τred1t-xt2 CAS-τred1t-xt3 Call-τred1t-obj
Call-τred1t-param Block-None-τred1t-xt Block-τred1t-Some Sync-τred1t-xt InSync-τred1t-xt
Seq-τred1t-xt Cond-τred1t-xt Throw-τred1t-xt Try-τred1t-xt

lemmas *τred1r-expr* =
NewArray-τred1r-xt Cast-τred1r-xt InstanceOf-τred1r-xt BinOp-τred1r-xt1 BinOp-τred1r-xt2 LAss-τred1r
AAcc-τred1r-xt1 AAcc-τred1r-xt2 AAss-τred1r-xt1 AAss-τred1r-xt2 AAss-τred1r-xt3
ALength-τred1r-xt FAcc-τred1r-xt FAss-τred1r-xt1 FAss-τred1r-xt2
CAS-τred1r-xt1 CAS-τred1r-xt2 CAS-τred1r-xt3 Call-τred1r-obj
Call-τred1r-param Block-None-τred1r-xt Block-τred1r-Some Sync-τred1r-xt InSync-τred1r-xt
Seq-τred1r-xt Cond-τred1r-xt Throw-τred1r-xt Try-τred1r-xt

definition *sim-move01* ::
'addr J1-prog ⇒ 'thread-id ⇒ ('addr, 'thread-id, 'heap) J0-thread-action ⇒ 'addr expr ⇒ 'addr expr1 ⇒ 'heap
⇒ 'addr locals1 ⇒ ('addr, 'thread-id, 'heap) external-thread-action ⇒ 'addr expr1 ⇒ 'heap ⇒ 'addr locals1 ⇒ bool
where
sim-move01 P t ta0 e0 e h xs ta e' h' xs' ⟷ ¬ final e0 ∧
(if τmove0 P h e0 then h' = h ∧ ta0 = ε ∧ ta = ε ∧ τred1't P t h (e, xs) (e', xs')
else ta-bisim01 ta0 (extTA2J1 P ta) ∧
(if call e0 = None ∨ call1 e = None
then (∃ e'' xs''. τred1'r P t h (e, xs) (e'', xs'') ∧ False,P,t ⊢1 ⟨e'', (h, xs'')⟩ −ta→ ⟨e', (h', xs')⟩ ∧
¬ τmove1 P h e'')
else False,P,t ⊢1 ⟨e, (h, xs)⟩ −ta→ ⟨e', (h', xs')⟩ ∧ ¬ τmove1 P h e))

definition *sim-moves01* ::
'addr J1-prog ⇒ 'thread-id ⇒ ('addr, 'thread-id, 'heap) J0-thread-action ⇒ 'addr expr list ⇒ 'addr expr1 list ⇒ 'heap
⇒ 'addr locals1 ⇒ ('addr, 'thread-id, 'heap) external-thread-action ⇒ 'addr expr1 list ⇒ 'heap ⇒ 'addr locals1 ⇒ bool
where
sim-moves01 P t ta0 es0 es h xs ta es' h' xs' ⟷ ¬ finals es0 ∧
(if τmoves0 P h es0 then h' = h ∧ ta0 = ε ∧ ta = ε ∧ τreds1't P t h (es, xs) (es', xs')
else ta-bisim01 ta0 (extTA2J1 P ta) ∧
(if calls es0 = None ∨ calls1 es = None
then (∃ es'' xs''. τreds1'r P t h (es, xs) (es'', xs'') ∧ False,P,t ⊢1 ⟨es'', (h, xs'')⟩ [−ta→] ⟨es', (h', xs')⟩ ∧
¬ τmoves1 P h es'')
else False,P,t ⊢1 ⟨es, (h, xs)⟩ [−ta→] ⟨es', (h', xs')⟩ ∧ ¬ τmoves1 P h es))

declare *τred1t-expr* [*elim*!] *τred1r-expr*[*elim*!]

lemma *sim-move01-expr*:
assumes *sim-move01 P t ta0 e0 e h xs ta e' h' xs'*
shows
sim-move01 P t ta0 (newA T⌊e0⌉) (newA T⌊e⌉) h xs ta (newA T⌊e'⌉) h' xs'
sim-move01 P t ta0 (Cast T e0) (Cast T e) h xs ta (Cast T e') h' xs'
sim-move01 P t ta0 (e0 instanceof T) (e instanceof T) h xs ta (e' instanceof T) h' xs'
sim-move01 P t ta0 (e0 «bop» e2) (e «bop» e2') h xs ta (e' «bop» e2') h' xs'
sim-move01 P t ta0 (Val v «bop» e0) (Val v «bop» e) h xs ta (Val v «bop» e') h' xs'
sim-move01 P t ta0 (V := e0) (V' := e) h xs ta (V' := e') h' xs'
sim-move01 P t ta0 (e0⌊e2⌉) (e⌊e2'⌉) h xs ta (e'⌊e2'⌉) h' xs'

sim-move01 P t ta0 (Val v⌊e0⌉) (Val v⌊e⌉) h xs ta (Val v⌊e'⌉) h' xs'
sim-move01 P t ta0 (e0⌊e2⌉ := e3) (e⌊e2'⌉ := e3') h xs ta (e'⌊e2'⌉ := e3') h' xs'
sim-move01 P t ta0 (Val v⌊e0⌉ := e3) (Val v⌊e⌉ := e3') h xs ta (Val v⌊e'⌉ := e3') h' xs'
sim-move01 P t ta0 (AAss (Val v) (Val v') e0) (AAss (Val v) (Val v') e) h xs ta (AAss (Val v) (Val v') e') h' xs'
sim-move01 P t ta0 (e0·length) (e·length) h xs ta (e'·length) h' xs'
sim-move01 P t ta0 (e0·F{D}) (e·F'{D'}) h xs ta (e'·F'{D'}) h' xs'
sim-move01 P t ta0 (FAss e0 F D e2) (FAss e F' D' e2') h xs ta (FAss e' F' D' e2') h' xs'
sim-move01 P t ta0 (FAss (Val v) F D e0) (FAss (Val v) F' D' e) h xs ta (FAss (Val v) F' D' e') h' xs'
sim-move01 P t ta0 (CompareAndSwap e0 D F e2 e3) (CompareAndSwap e D F e2' e3') h xs ta (CompareAndSwap e' D F e2' e3') h' xs'
sim-move01 P t ta0 (CompareAndSwap (Val v) D F e0 e3) (CompareAndSwap (Val v) D F e e3') h xs ta (CompareAndSwap (Val v) D F e' e3') h' xs'
sim-move01 P t ta0 (CompareAndSwap (Val v) D F (Val v') e0) (CompareAndSwap (Val v) D F (Val v') e) h xs ta (CompareAndSwap (Val v) D F (Val v') e') h' xs'
sim-move01 P t ta0 (e0·M(es)) (e·M(es')) h xs ta (e'·M(es')) h' xs'
sim-move01 P t ta0 ({V:T=vo; e0}) ({V':T=None; e}) h xs ta ({V':T=None; e'}) h' xs'
sim-move01 P t ta0 (sync(e0) e2) (sync$_{V'}$(e) e2') h xs ta (sync$_{V'}$(e') e2') h' xs'
sim-move01 P t ta0 (insync(a) e0) (insync$_{V'}$(a') e) h xs ta (insync$_{V'}$(a') e') h' xs'
sim-move01 P t ta0 (e0;;e2) (e;;e2') h xs ta (e';;e2') h' xs'
sim-move01 P t ta0 (if (e0) e2 else e3) (if (e) e2' else e3') h xs ta (if (e') e2' else e3') h' xs'
sim-move01 P t ta0 (throw e0) (throw e) h xs ta (throw e') h' xs'
sim-move01 P t ta0 (try e0 catch(C V) e2) (try e catch(C' V') e2') h xs ta (try e' catch(C' V') e2') h' xs'
using *assms*
apply(*simp-all add*: *sim-move01-def final-iff τred1r-Val τred1t-Val split*: *if-split-asm split del*: *if-split*)
apply(*fastforce simp add*: *final-iff τred1r-Val τred1t-Val split*!: *if-splits intro*: *red1-reds1.intros*)+
done

lemma *sim-moves01-expr*:
sim-move01 P t ta0 e0 e h xs ta e' h' xs' ⟹ sim-moves01 P t ta0 (e0 # es2) (e # es2') h xs ta (e' # es2') h' xs'
sim-moves01 P t ta0 es0 es h xs ta es' h' xs' ⟹ sim-moves01 P t ta0 (Val v # es0) (Val v # es) h xs ta (Val v # es') h' xs'
apply(*simp-all add*: *sim-move01-def sim-moves01-def final-iff finals-iff Cons-eq-append-conv τred1t-Val τred1r-Val split*: *if-split-asm split del*: *if-split*)
apply(*auto simp add*: *Cons-eq-append-conv τred1t-Val τred1r-Val split*!: *if-splits intro*: *List1Red1 List1Red2 τred1t-inj-τreds1t τred1r-inj-τreds1r τreds1t-cons-τreds1t τreds1r-cons-τreds1r*)
apply(*force elim*!: *τred1r-inj-τreds1r List1Red1*)
apply(*force elim*!: *τred1r-inj-τreds1r List1Red1*)
apply(*force elim*!: *τred1r-inj-τreds1r List1Red1*)
apply(*force elim*!: *τred1r-inj-τreds1r List1Red1*)
apply(*force elim*!: *τreds1r-cons-τreds1r intro*!: *List1Red2*)
apply(*force elim*!: *τreds1r-cons-τreds1r intro*!: *List1Red2*)
done

lemma *sim-move01-CallParams*:
sim-moves01 P t ta0 es0 es h xs ta es' h' xs'
⟹ sim-move01 P t ta0 (Val v·M(es0)) (Val v·M(es)) h xs ta (Val v·M(es')) h' xs'
apply(*clarsimp simp add*: *sim-move01-def sim-moves01-def τreds1r-map-Val τreds1t-map-Val is-vals-conv split*: *if-split-asm split del*: *if-split*)
apply(*fastforce simp add*: *sim-move01-def sim-moves01-def τreds1r-map-Val τreds1t-map-Val intro*: *Call-τred1r-param Call1Params*)

apply(*rule conjI*, *fastforce*)
apply(*split if-split*)
apply(*rule conjI*)
apply(*clarsimp simp add*: *finals-iff*)
apply(*clarify*)
apply(*split if-split*)
apply(*rule conjI*)
apply(*simp del*: *call.simps calls.simps call1.simps calls1.simps*)
apply(*fastforce simp add*: *sim-move01-def sim-moves01-def* τ*red1r-Val* τ*red1t-Val* τ*reds1r-map-Val-Throw intro*: *Call-*τ*red1r-param Call1Params split*: *if-split-asm*)
apply(*fastforce split*: *if-split-asm simp add*: *is-vals-conv* τ*reds1r-map-Val* τ*reds1r-map-Val-Throw*)
apply(*rule conjI*, *fastforce*)
apply(*fastforce simp add*: *sim-move01-def sim-moves01-def* τ*red1r-Val* τ*red1t-Val* τ*reds1t-map-Val* τ*reds1r-map-Val is-vals-conv intro*: *Call-*τ*red1r-param Call1Params split*: *if-split-asm*)
done

lemma *sim-move01-reds*:
⟦ (*h'*, *a*) ∈ *allocate h* (*Class-type C*); *ta0* = {|*NewHeapElem a* (*Class-type C*)|}; *ta* = {|*NewHeapElem a* (*Class-type C*)|} ⟧
⟹ *sim-move01 P t ta0* (*new C*) (*new C*) *h xs ta* (*addr a*) *h' xs*
allocate h (*Class-type C*) = {} ⟹ *sim-move01 P t* ε (*new C*) (*new C*) *h xs* ε (*THROW OutOfMemory*) *h xs*
⟦ (*h'*, *a*) ∈ *allocate h* (*Array-type T* (*nat* (*sint i*))); *0* <=*s i*;
ta0 = {|*NewHeapElem a* (*Array-type T* (*nat* (*sint i*)))|}; *ta* = {|*NewHeapElem a* (*Array-type T* (*nat* (*sint i*)))|} ⟧
⟹ *sim-move01 P t ta0* (*newA T*⌊*Val* (*Intg i*)⌉) (*newA T*⌊*Val* (*Intg i*)⌉) *h xs ta* (*addr a*) *h' xs*
i <*s 0* ⟹ *sim-move01 P t* ε (*newA T*⌊*Val* (*Intg i*)⌉) (*newA T*⌊*Val* (*Intg i*)⌉) *h xs* ε (*THROW NegativeArraySize*) *h xs*
⟦ *allocate h* (*Array-type T* (*nat* (*sint i*))) = {}; *0* <=*s i* ⟧
⟹ *sim-move01 P t* ε (*newA T*⌊*Val* (*Intg i*)⌉) (*newA T*⌊*Val* (*Intg i*)⌉) *h xs* ε (*THROW OutOfMemory*) *h xs*
⟦ *typeof*$_h$ *v* = ⌊*U*⌋; *P* ⊢ *U* ≤ *T* ⟧
⟹ *sim-move01 P t* ε (*Cast T* (*Val v*)) (*Cast T* (*Val v*)) *h xs* ε (*Val v*) *h xs*
⟦ *typeof*$_h$ *v* = ⌊*U*⌋; ¬ *P* ⊢ *U* ≤ *T* ⟧
⟹ *sim-move01 P t* ε (*Cast T* (*Val v*)) (*Cast T* (*Val v*)) *h xs* ε (*THROW ClassCast*) *h xs*
⟦ *typeof*$_h$ *v* = ⌊*U*⌋; *b* ⟷ *v* ≠ *Null* ∧ *P* ⊢ *U* ≤ *T* ⟧
⟹ *sim-move01 P t* ε ((*Val v*) *instanceof T*) ((*Val v*) *instanceof T*) *h xs* ε (*Val* (*Bool b*)) *h xs*
binop bop v1 v2 = *Some* (*Inl v*) ⟹ *sim-move01 P t* ε ((*Val v1*) «*bop*» (*Val v2*)) (*Val v1* «*bop*» *Val v2*) *h xs* ε (*Val v*) *h xs*
binop bop v1 v2 = *Some* (*Inr a*) ⟹ *sim-move01 P t* ε ((*Val v1*) «*bop*» (*Val v2*)) (*Val v1* «*bop*» *Val v2*) *h xs* ε (*Throw a*) *h xs*
⟦ *xs*!*V* = *v*; *V* < *size xs* ⟧ ⟹ *sim-move01 P t* ε (*Var V'*) (*Var V*) *h xs* ε (*Val v*) *h xs*
V < *length xs* ⟹ *sim-move01 P t* ε (*V'* := *Val v*) (*V* := *Val v*) *h xs* ε *unit h* (*xs*[*V* := *v*])
sim-move01 P t ε (*null*⌊*Val v*⌉) (*null*⌊*Val v*⌉) *h xs* ε (*THROW NullPointer*) *h xs*
⟦ *typeof-addr h a* = ⌊*Array-type T n*⌋; *i* <*s 0* ∨ *sint i* ≥ *int n* ⟧
⟹ *sim-move01 P t* ε (*addr a*⌊*Val* (*Intg i*)⌉) ((*addr a*)⌊*Val* (*Intg i*)⌉) *h xs* ε (*THROW ArrayIndexOutOfBounds*) *h xs*
⟦ *typeof-addr h a* = ⌊*Array-type T n*⌋; *0* <=*s i*; *sint i* < *int n*;
heap-read h a (*ACell* (*nat* (*sint i*))) *v*;
ta0 = {|*ReadMem a* (*ACell* (*nat* (*sint i*))) *v*|};
ta = {|*ReadMem a* (*ACell* (*nat* (*sint i*))) *v*|} ⟧
⟹ *sim-move01 P t ta0* (*addr a*⌊*Val* (*Intg i*)⌉) ((*addr a*)⌊*Val* (*Intg i*)⌉) *h xs ta* (*Val v*) *h xs*
sim-move01 P t ε (*null*⌊*Val v*⌉ := *Val v'*) (*null*⌊*Val v*⌉ := *Val v'*) *h xs* ε (*THROW NullPointer*) *h xs*
⟦ *typeof-addr h a* = ⌊*Array-type T n*⌋; *i* <*s 0* ∨ *sint i* ≥ *int n* ⟧

$\Longrightarrow$ *sim-move01 P t* ε (*AAss* (*addr a*) (*Val* (*Intg i*)) (*Val v*)) (*AAss* (*addr a*) (*Val* (*Intg i*)) (*Val v*)) *h xs* ε (*THROW ArrayIndexOutOfBounds*) *h xs*

$[\![$ *typeof-addr h a* = $\lfloor$*Array-type T n*$\rfloor$; 0 <=*s i*; *sint i* < *int n*; *typeof*$_h$ *v* = $\lfloor$*U*$\rfloor$; $\neg$ (*P* $\vdash$ *U* $\leq$ *T*) $]\!]$

$\Longrightarrow$ *sim-move01 P t* ε (*AAss* (*addr a*) (*Val* (*Intg i*)) (*Val v*)) (*AAss* (*addr a*) (*Val* (*Intg i*)) (*Val v*)) *h xs* ε (*THROW ArrayStore*) *h xs*

$[\![$ *typeof-addr h a* = $\lfloor$*Array-type T n*$\rfloor$; 0 <=*s i*; *sint i* < *int n*; *typeof*$_h$ *v* = *Some U*; *P* $\vdash$ *U* $\leq$ *T*; *heap-write h a* (*ACell* (*nat* (*sint i*))) *v h*′; *ta0* = $\{\!|$*WriteMem a* (*ACell* (*nat* (*sint i*))) *v*$|\!\}$; *ta* = $\{\!|$*WriteMem a* (*ACell* (*nat* (*sint i*))) *v* $|\!\}$ $]\!]$

$\Longrightarrow$ *sim-move01 P t ta0* (*AAss* (*addr a*) (*Val* (*Intg i*)) (*Val v*)) (*AAss* (*addr a*) (*Val* (*Intg i*)) (*Val v*)) *h xs ta unit h*′ *xs*

typeof-addr h a = $\lfloor$*Array-type T n*$\rfloor$ $\Longrightarrow$ *sim-move01 P t* ε (*addr a*·*length*) (*addr a*·*length*) *h xs* ε (*Val* (*Intg* (*word-of-int* (*int n*)))) *h xs*

sim-move01 P t ε (*null*·*length*) (*null*·*length*) *h xs* ε (*THROW NullPointer*) *h xs*

$[\![$ *heap-read h a* (*CField D F*) *v*; *ta0* = $\{\!|$*ReadMem a* (*CField D F*) *v*$|\!\}$; *ta* = $\{\!|$*ReadMem a* (*CField D F*) *v*$|\!\}$ $]\!]$

$\Longrightarrow$ *sim-move01 P t ta0* (*addr a*·*F*{*D*}) (*addr a*·*F*{*D*}) *h xs ta* (*Val v*) *h xs*

sim-move01 P t ε (*null*·*F*{*D*}) (*null*·*F*{*D*}) *h xs* ε (*THROW NullPointer*) *h xs*

$[\![$ *heap-write h a* (*CField D F*) *v h*′; *ta0* = $\{\!|$*WriteMem a* (*CField D F*) *v*$|\!\}$; *ta* = $\{\!|$*WriteMem a* (*CField D F*) *v*$|\!\}$ $]\!]$

$\Longrightarrow$ *sim-move01 P t ta0* (*addr a*·*F*{*D*} := *Val v*) (*addr a*·*F*{*D*} := *Val v*) *h xs ta unit h*′ *xs*

sim-move01 P t ε (*null*·*compareAndSwap*(*D*·*F*, *Val v*, *Val v*′)) (*null*·*compareAndSwap*(*D*·*F*, *Val v*, *Val v*′)) *h xs* ε (*THROW NullPointer*) *h xs*

$[\![$ *heap-read h a* (*CField D F*) *v*″; *heap-write h a* (*CField D F*) *v*′ *h*′; *v*″ = *v*; *ta0* = $\{\!|$*ReadMem a* (*CField D F*) *v*″, *WriteMem a* (*CField D F*) *v*′$|\!\}$; *ta* = $\{\!|$*ReadMem a* (*CField D F*) *v*″, *WriteMem a* (*CField D F*) *v*′$|\!\}$ $]\!]$

$\Longrightarrow$ *sim-move01 P t ta0* (*addr a*·*compareAndSwap*(*D*·*F*, *Val v*, *Val v*′)) (*addr a*·*compareAndSwap*(*D*·*F*, *Val v*, *Val v*′)) *h xs ta true h*′ *xs*

$[\![$ *heap-read h a* (*CField D F*) *v*″; *v*″ $\neq$ *v*; *ta0* = $\{\!|$*ReadMem a* (*CField D F*) *v*″$|\!\}$; *ta* = $\{\!|$*ReadMem a* (*CField D F*) *v*″$|\!\}$ $]\!]$

$\Longrightarrow$ *sim-move01 P t ta0* (*addr a*·*compareAndSwap*(*D*·*F*, *Val v*, *Val v*′)) (*addr a*·*compareAndSwap*(*D*·*F*, *Val v*, *Val v*′)) *h xs ta false h xs*

sim-move01 P t ε (*null*·*F*{*D*} := *Val v*) (*null*·*F*{*D*} := *Val v*) *h xs* ε (*THROW NullPointer*) *h xs*

sim-move01 P t ε ({*V*′:*T*=*vo*; *Val u*}) ({*V*:*T*=*None*; *Val u*}) *h xs* ε (*Val u*) *h xs*

V < *length xs* $\Longrightarrow$ *sim-move01 P t* ε (*sync*(*null*) *e0*) (*sync*$_V$ (*null*) *e1*) *h xs* ε (*THROW NullPointer*) *h* (*xs*[*V* := *Null*])

sim-move01 P t ε (*Val v*;;*e0*) (*Val v*;; *e1*) *h xs* ε *e1 h xs*

sim-move01 P t ε (*if* (*true*) *e0 else e0*′) (*if* (*true*) *e1 else e1*′) *h xs* ε *e1 h xs*

sim-move01 P t ε (*if* (*false*) *e0 else e0*′) (*if* (*false*) *e1 else e1*′) *h xs* ε *e1*′ *h xs*

sim-move01 P t ε (*throw null*) (*throw null*) *h xs* ε (*THROW NullPointer*) *h xs*

sim-move01 P t ε (*try* (*Val v*) *catch*(*C V*′) *e0*) (*try* (*Val v*) *catch*(*C V*) *e1*) *h xs* ε (*Val v*) *h xs*

$[\![$ *typeof-addr h a* = $\lfloor$*Class-type D*$\rfloor$; *P* $\vdash$ *D* $\preceq^*$ *C*; *V* < *length xs* $]\!]$

$\Longrightarrow$ *sim-move01 P t* ε (*try* (*Throw a*) *catch*(*C V*′) *e0*) (*try* (*Throw a*) *catch*(*C V*) *e1*) *h xs* ε ({*V*:*Class C*=*None*; *e1*}) *h* (*xs*[*V* := *Addr a*])

sim-move01 P t ε (*newA T*$\lfloor$*Throw a*$\rceil$) (*newA T*$\lfloor$*Throw a*$\rceil$) *h xs* ε (*Throw a*) *h xs*

sim-move01 P t ε (*Cast T* (*Throw a*)) (*Cast T* (*Throw a*)) *h xs* ε (*Throw a*) *h xs*

sim-move01 P t ε ((*Throw a*) *instanceof T*) ((*Throw a*) *instanceof T*) *h xs* ε (*Throw a*) *h xs*

sim-move01 P t ε ((*Throw a*) «*bop*» *e0*) ((*Throw a*) «*bop*» *e1*) *h xs* ε (*Throw a*) *h xs*

sim-move01 P t ε (*Val v* «*bop*» (*Throw a*)) (*Val v* «*bop*» (*Throw a*)) *h xs* ε (*Throw a*) *h xs*

sim-move01 P t ε (*V*′ := *Throw a*) (*V* := *Throw a*) *h xs* ε (*Throw a*) *h xs*

sim-move01 P t ε (*Throw a*$\lfloor$*e0*$\rceil$) (*Throw a*$\lfloor$*e1*$\rceil$) *h xs* ε (*Throw a*) *h xs*

sim-move01 P t ε (*Val v*$\lfloor$*Throw a*$\rceil$) (*Val v*$\lfloor$*Throw a*$\rceil$) *h xs* ε (*Throw a*) *h xs*

sim-move01 P t ε (*Throw a*$\lfloor$*e0*$\rceil$:= *e0*′) (*Throw a*$\lfloor$*e1*$\rceil$:= *e1*′) *h xs* ε (*Throw a*) *h xs*

sim-move01 P t ε (*Val v*$\lfloor$*Throw a*$\rceil$:= *e0*) (*Val v*$\lfloor$*Throw a*$\rceil$:= *e1*) *h xs* ε (*Throw a*) *h xs*

sim-move01 P t ε (Val v⌊Val v'⌉ := Throw a) (Val v⌊Val v'⌉ := Throw a) h xs ε (Throw a) h xs
sim-move01 P t ε (Throw a·length) (Throw a·length) h xs ε (Throw a) h xs
sim-move01 P t ε (Throw a·F{D}) (Throw a·F{D}) h xs ε (Throw a) h xs
sim-move01 P t ε (Throw a·F{D} := e0) (Throw a·F{D} := e1) h xs ε (Throw a) h xs
sim-move01 P t ε (Val v·F{D} := Throw a) (Val v·F{D} := Throw a) h xs ε (Throw a) h xs
sim-move01 P t ε (Throw a·compareAndSwap(D·F, e2, e3)) (Throw a·compareAndSwap(D·F, e2', e3')) h xs ε (Throw a) h xs
sim-move01 P t ε (Val v·compareAndSwap(D·F, Throw a, e3)) (Val v·compareAndSwap(D·F, Throw a, e3')) h xs ε (Throw a) h xs
sim-move01 P t ε (Val v·compareAndSwap(D·F, Val v', Throw a)) (Val v·compareAndSwap(D·F, Val v', Throw a)) h xs ε (Throw a) h xs
sim-move01 P t ε (Throw a·M(es0)) (Throw a·M(es1)) h xs ε (Throw a) h xs
sim-move01 P t ε ({V':T=vo; Throw a}) ({V:T=None; Throw a}) h xs ε (Throw a) h xs
sim-move01 P t ε (sync(Throw a) e0) (sync$_V$(Throw a) e1) h xs ε (Throw a) h xs
sim-move01 P t ε (Throw a;;e0) (Throw a;;e1) h xs ε (Throw a) h xs
sim-move01 P t ε (if (Throw a) e0 else e0') (if (Throw a) e1 else e1') h xs ε (Throw a) h xs
sim-move01 P t ε (throw (Throw a)) (throw (Throw a)) h xs ε (Throw a) h xs
apply(*simp-all add*: *sim-move01-def ta-bisim-def split*: *if-split-asm split del*: *if-split*)
apply(*fastforce intro*: *red1-reds1.intros*)+
done

lemma *sim-move01-ThrowParams*:
sim-move01 P t ε (Val v·M(map Val vs @ Throw a # es0)) (Val v·M(map Val vs @ Throw a # es1)) h xs ε (Throw a) h xs
apply(*simp add*: *sim-move01-def split del*: *if-split*)
apply(*rule conjI*, *fastforce*)
apply(*split if-split*)
apply(*rule conjI*)
apply(*fastforce intro*: *red1-reds1.intros*)
apply(*fastforce simp add*: *sim-move01-def intro*: *red1-reds1.intros*)
done

lemma *sim-move01-CallNull*:
sim-move01 P t ε (null·M(map Val vs)) (null·M(map Val vs)) h xs ε (THROW NullPointer) h xs
by(*fastforce simp add*: *sim-move01-def map-eq-append-conv intro*: *red1-reds1.intros*)

lemma *sim-move01-SyncLocks*:
⟦ *V < length xs*; *ta0 = {|Lock→a, SyncLock a|}*; *ta = {|Lock→a, SyncLock a|}* ⟧
⟹ *sim-move01 P t ta0 (sync(addr a) e0) (sync$_V$ (addr a) e1) h xs ta (insync$_V$ (a) e1) h (xs[V := Addr a])*
⟦ *xs ! V = Addr a'*; *V < length xs*; *ta0 = {|Unlock→a', SyncUnlock a'|}*; *ta = {|Unlock→a', SyncUnlock a'|}* ⟧
⟹ *sim-move01 P t ta0 (insync(a') (Val v)) (insync$_V$ (a) (Val v)) h xs ta (Val v) h xs*
⟦ *xs ! V = Addr a'*; *V < length xs*; *ta0 = {|Unlock→a', SyncUnlock a'|}*; *ta = {|Unlock→a', SyncUnlock a'|}* ⟧
⟹ *sim-move01 P t ta0 (insync(a') (Throw a'')) (insync$_V$ (a) (Throw a'')) h xs ta (Throw a'') h xs*
by(*fastforce simp add*: *sim-move01-def ta-bisim-def expand-finfun-eq fun-eq-iff finfun-upd-apply ta-upd-simps intro*: *red1-reds1.intros*[*simplified*] *split*: *if-split-asm*)+

lemma *sim-move01-TryFail*:
⟦ *typeof-addr h a = ⌊Class-type D⌋*; ¬ *P ⊢ D ⪯* C* ⟧
⟹ *sim-move01 P t ε (try (Throw a) catch(C V') e0) (try (Throw a) catch(C V) e1) h xs ε (Throw a) h xs*
by(*auto simp add*: *sim-move01-def intro*!: *Red1TryFail*)

lemma *sim-move01-BlockSome*:
⟦ *sim-move01 P t ta0 e0 e h* (*xs*[*V* := *v*]) *ta e′ h′ xs′*; *V* < *length xs* ⟧
⟹ *sim-move01 P t ta0* ({*V′*:*T*=⌊*v*⌋; *e0*}) ({*V*:*T*=⌊*v*⌋; *e*}) *h xs ta* ({*V*:*T*=*None*; *e′*}) *h′ xs′*
V < *length xs* ⟹ *sim-move01 P t ε* ({*V′*:*T*=⌊*v*⌋; *Val u*}) ({*V*:*T*=⌊*v*⌋; *Val u*}) *h xs ε* (*Val u*) *h* (*xs*[*V* := *v*])
V < *length xs* ⟹ *sim-move01 P t ε* ({*V′*:*T*=⌊*v*⌋; *Throw a*}) ({*V*:*T*=⌊*v*⌋; *Throw a*}) *h xs ε* (*Throw a*) *h* (*xs*[*V* := *v*])
apply(*auto simp add*: *sim-move01-def*)
apply(*split if-split-asm*)
apply(*fastforce intro*: *intro*: *converse-rtranclp-into-rtranclp Block1Some Block1Red Block-*τ*red1r-Some*)
apply(*fastforce intro*: *intro*: *converse-rtranclp-into-rtranclp Block1Some Block1Red Block-*τ*red1r-Some*)
apply(*fastforce simp add*: *sim-move01-def intro*!: τ*red1t-2step*[*OF Block1Some*] τ*red1r-1step*[*OF Block1Some*] *Red1Block Block1Throw*)+
done

lemmas *sim-move01-intros* =
sim-move01-expr sim-move01-reds sim-move01-ThrowParams sim-move01-CallNull sim-move01-TryFail sim-move01-BlockSome sim-move01-CallParams

declare *sim-move01-intros*[*intro*]

lemma *sim-move01-preserves-len*: *sim-move01 P t ta0 e0 e h xs ta e′ h′ xs′* ⟹ *length xs′* = *length xs*
by(*fastforce simp add*: *sim-move01-def split*: *if-split-asm dest*: τ*red1r-preserves-len* τ*red1t-preserves-len red1-preserves-len*)

lemma *sim-move01-preserves-unmod*:
⟦ *sim-move01 P t ta0 e0 e h xs ta e′ h′ xs′*; *unmod e i*; *i* < *length xs* ⟧ ⟹ *xs′* ! *i* = *xs* ! *i*
apply(*auto simp add*: *sim-move01-def split*: *if-split-asm dest*: τ*red1t-preserves-unmod*)
apply(*frule* (*2*) τ*red1′r-preserves-unmod*)
apply(*frule* (*1*) τ*red1r-unmod-preserved*)
apply(*frule* τ*red1r-preserves-len*)
apply(*auto dest*: *red1-preserves-unmod*)
apply(*frule* (*2*) τ*red1′r-preserves-unmod*)
apply(*frule* (*1*) τ*red1r-unmod-preserved*)
apply(*frule* τ*red1r-preserves-len*)
apply(*auto dest*: *red1-preserves-unmod*)
done

lemma assumes *wf*: *wf-J-prog P*
shows *red1-simulates-red-aux*:
⟦ *extTA2J0 P*,*P*,*t* ⊢ ⟨*e1*, *S*⟩ −*TA*→ ⟨*e1′*, *S′*⟩; *bisim vs e1 e2 XS*; *fv e1* ⊆ *set vs*;
lcl S ⊆$_m$ [*vs* [↦] *XS*]; *length vs* + *max-vars e1* ≤ *length XS*;
∀ *aMvs*. *call e1* = ⌊*aMvs*⌋ ⟶ *synthesized-call P* (*hp S*) *aMvs* ⟧
⟹ ∃ *ta e2′ XS′*. *sim-move01* (*compP1 P*) *t TA e1 e2* (*hp S*) *XS ta e2′* (*hp S′*) *XS′* ∧ *bisim vs e1′ e2′ XS′* ∧ *lcl S′* ⊆$_m$ [*vs* [↦] *XS′*]
(**is** ⟦ -; -; -; -; -; *?synth e1 S* ⟧ ⟹ *?concl e1 e2 S XS e1′ S′ TA vs*)

and *reds1-simulates-reds-aux*:
⟦ *extTA2J0 P*,*P*,*t* ⊢ ⟨*es1*, *S*⟩ [−*TA*→] ⟨*es1′*, *S′*⟩; *bisims vs es1 es2 XS*; *fvs es1* ⊆ *set vs*;
lcl S ⊆$_m$ [*vs* [↦] *XS*]; *length vs* + *max-varss es1* ≤ *length XS*;
∀ *aMvs*. *calls es1* = ⌊*aMvs*⌋ ⟶ *synthesized-call P* (*hp S*) *aMvs* ⟧
⟹ ∃ *ta es2′ xs′*. *sim-moves01* (*compP1 P*) *t TA es1 es2* (*hp S*) *XS ta es2′* (*hp S′*) *xs′* ∧ *bisims vs es1′ es2′ xs′* ∧ *lcl S′* ⊆$_m$ [*vs* [↦] *xs′*]

(**is** ⟦ -; -; -; -; -; -; *?synths es1 S* ⟧ ⟹ *?concls es1 es2 S XS es1′ S′ TA vs*)
proof(*induct arbitrary*: *vs e2 XS* **and** *vs es2 XS rule*: *red-reds.inducts*)
case (*BinOpRed1 e s ta e′ s′ bop e2 Vs E2 xs*)
note *IH* = ‹⋀*vs e2 XS*. ⟦*bisim vs e e2 XS*; *fv e* ⊆ *set vs*; *lcl s* ⊆$_m$ [*vs* [↦] *XS*]; *length vs* + *max-vars e* ≤ *length XS*;
?synth e s ⟧ ⟹ *?concl e e2 s XS e′ s′ ta vs*›
from ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩› **have** ¬ *is-val e* **by** *auto*
with ‹*bisim Vs* (*e* «*bop*» *e2*) *E2 xs*› **obtain** *E*
where *E2* = *E* «*bop*» *compE1 Vs e2* **and** *bisim Vs e E xs* **and** ¬ *contains-insync e2* **by** *auto*
with *IH*[*of Vs E xs*] ‹*fv* (*e* «*bop*» *e2*) ⊆ *set Vs*› ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]› ‹¬ *is-val e*›
‹*length Vs* + *max-vars* (*e* «*bop*» *e2*) ≤ *length xs*› ‹*?synth* (*e* «*bop*» *e2*) *s*› ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩›
show *?case* **by**(*cases is-val e′*)(*fastforce elim*!: *sim-move01-expr*)+
next
case (*BinOpRed2 e s ta e′ s′ v bop Vs E2 xs*)
note *IH* = ‹⋀*vs e2 XS*. ⟦*bisim vs e e2 XS*; *fv e* ⊆ *set vs*; *lcl s* ⊆$_m$ [*vs* [↦] *XS*]; *length vs* + *max-vars e* ≤ *length XS*;
?synth e s ⟧ ⟹ *?concl e e2 s XS e′ s′ ta vs*›
from ‹*bisim Vs* (*Val v* «*bop*» *e*) *E2 xs*› **obtain** *E*
where *E2* = *Val v* «*bop*» *E* **and** *bisim Vs e E xs* **by** *auto*
with *IH*[*of Vs E xs*] ‹*fv* (*Val v* «*bop*» *e*) ⊆ *set Vs*› ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]›
‹*length Vs* + *max-vars* (*Val v* «*bop*» *e*) ≤ *length xs*› ‹*?synth* (*Val v* «*bop*» *e*) *s*› ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩›
show *?case* **by**(*fastforce elim*!: *sim-move01-expr*)
next
case *RedVar* **thus** *?case*
by(*fastforce simp add*: *index-less-aux map-le-def fun-upds-apply intro*!: *exI dest*: *bspec*)
next
case *RedLAss* **thus** *?case*
by(*fastforce intro*: *index-less-aux LAss-lem intro*!: *exI simp del*: *fun-upd-apply*)
next
case (*AAccRed1 a s ta a′ s′ i Vs E2 xs*)
note *IH* = ‹⋀*vs e2 XS*. ⟦*bisim vs a e2 XS*; *fv a* ⊆ *set vs*; *lcl s* ⊆$_m$ [*vs* [↦] *XS*]; *length vs* + *max-vars a* ≤ *length XS*;
?synth a s ⟧ ⟹ *?concl a e2 s XS a′ s′ ta vs*›
from ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*a*,*s*⟩ −*ta*→ ⟨*a′*,*s′*⟩› **have** ¬ *is-val a* **by** *auto*
with ‹*bisim Vs* (*a*⌊*i*⌉) *E2 xs*› **obtain** *E*
where *E2* = *E*⌊*compE1 Vs i*⌉ **and** *bisim Vs a E xs* **and** ¬ *contains-insync i* **by** *auto*
with *IH*[*of Vs E xs*] ‹*fv* (*a*⌊*i*⌉) ⊆ *set Vs*› ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]› ‹¬ *is-val a*›
‹*length Vs* + *max-vars* (*a*⌊*i*⌉) ≤ *length xs*› ‹*?synth* (*a*⌊*i*⌉) *s*› ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*a*,*s*⟩ −*ta*→ ⟨*a′*,*s′*⟩›
show *?case* **by**(*cases is-val a′*)(*fastforce elim*!: *sim-move01-expr*)+
next
case (*AAccRed2 i s ta i′ s′ a Vs E2 xs*)
note *IH* = ‹⋀*vs e2 XS*. ⟦*bisim vs i e2 XS*; *fv i* ⊆ *set vs*; *lcl s* ⊆$_m$ [*vs* [↦] *XS*]; *length vs* + *max-vars i* ≤ *length XS*;
?synth i s ⟧ ⟹ *?concl i e2 s XS i′ s′ ta vs*›
from ‹*bisim Vs* (*Val a*⌊*i*⌉) *E2 xs*› **obtain** *E*
where *E2* = *Val a*⌊*E*⌉ **and** *bisim Vs i E xs* **by** *auto*
with *IH*[*of Vs E xs*] ‹*fv* (*Val a*⌊*i*⌉) ⊆ *set Vs*› ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]›
‹*length Vs* + *max-vars* (*Val a*⌊*i*⌉) ≤ *length xs*› ‹*?synth* (*Val a*⌊*i*⌉) *s*› ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*i*,*s*⟩ −*ta*→ ⟨*i′*,*s′*⟩›
show *?case* **by**(*fastforce elim*!: *sim-move01-expr*)
next
case *RedAAcc* **thus** *?case* **by**(*auto simp del*: *split-paired-Ex*)

next
case (*AAssRed1 a s ta a′ s′ i e Vs E2 xs*)
note *IH* = ‹⋀*vs e2 XS*. ⟦*bisim vs a e2 XS*; *fv a* ⊆ *set vs*; *lcl s* ⊆$_m$ [*vs* [↦] *XS*]; *length vs* + *max-vars a* ≤ *length XS*;
?synth a s ⟧ ⟹ *?concl a e2 s XS a′ s′ ta vs*›
from ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*a*,*s*⟩ −*ta*→ ⟨*a′*,*s′*⟩› **have** ¬ *is-val a* **by** *auto*
with ‹*bisim Vs* (*a*⌊*i*⌉:=*e*) *E2 xs*› **obtain** *E*
where *E2*: *E2* = *E*⌊*compE1 Vs i*⌉:=*compE1 Vs e* **and** *bisim Vs a E xs*
and *sync*: ¬ *contains-insync i* ¬ *contains-insync e* **by** *auto*
with *IH*[*of Vs E xs*] ‹*fv* (*a*⌊*i*⌉:=*e*) ⊆ *set Vs*› ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]› ‹¬ *is-val a*› ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*a*,*s*⟩ −*ta*→ ⟨*a′*,*s′*⟩›
‹*length Vs* + *max-vars* (*a*⌊*i*⌉:=*e*) ≤ *length xs*› ‹*?synth* (*a*⌊*i*⌉:=*e*) *s*›
obtain *ta′ e2′ xs′*
where *IH′*: *sim-move01* (*compP1 P*) *t ta a E* (*hp s*) *xs ta′ e2′* (*hp s′*) *xs′*
bisim Vs a′ e2′ xs′ lcl s′ ⊆$_m$ [*Vs* [↦] *xs′*]
by *auto*
show *?case*
proof(*cases is-val a′*)
case *True*
from ‹*fv* (*a*⌊*i*⌉:=*e*) ⊆ *set Vs*› *sync*
have *bisim Vs i* (*compE1 Vs i*) *xs′ bisim Vs e* (*compE1 Vs e*) *xs′* **by** *auto*
with *IH′ E2 True sync* ‹¬ *is-val a*› ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*a*,*s*⟩ −*ta*→ ⟨*a′*,*s′*⟩› **show** *?thesis*
by(*cases is-val i*)(*fastforce elim*!: *sim-move01-expr*)+
next
case *False* **with** *IH′ E2 sync* ‹¬ *is-val a*› ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*a*,*s*⟩ −*ta*→ ⟨*a′*,*s′*⟩›
show *?thesis* **by**(*fastforce elim*!: *sim-move01-expr*)
qed
next
case (*AAssRed2 i s ta i′ s′ a e Vs E2 xs*)
note *IH* = ‹⋀*vs e2 XS*. ⟦*bisim vs i e2 XS*; *fv i* ⊆ *set vs*; *lcl s* ⊆$_m$ [*vs* [↦] *XS*]; *length vs* + *max-vars i* ≤ *length XS*;
?synth i s ⟧ ⟹ *?concl i e2 s XS i′ s′ ta vs*›
from ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*i*,*s*⟩ −*ta*→ ⟨*i′*,*s′*⟩› **have** ¬ *is-val i* **by** *auto*
with ‹*bisim Vs* (*Val a*⌊*i*⌉ := *e*) *E2 xs*› **obtain** *E*
where *E2* = *Val a*⌊*E*⌉:=*compE1 Vs e* **and** *bisim Vs i E xs* **and** ¬ *contains-insync e* **by** *auto*
with *IH*[*of Vs E xs*] ‹*fv* (*Val a*⌊*i*⌉:=*e*) ⊆ *set Vs*› ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]› ‹¬ *is-val i*› ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*i*,*s*⟩ −*ta*→ ⟨*i′*,*s′*⟩›
‹*length Vs* + *max-vars* (*Val a*⌊*i*⌉:=*e*) ≤ *length xs*› ‹*?synth* (*Val a*⌊*i*⌉:=*e*) *s*›
show *?case* **by**(*cases is-val i′*)(*fastforce elim*!: *sim-move01-expr*)+
next
case (*AAssRed3 e s ta e′ s′ a i Vs E2 xs*)
note *IH* = ‹⋀*vs e2 XS*. ⟦*bisim vs e e2 XS*; *fv e* ⊆ *set vs*; *lcl s* ⊆$_m$ [*vs* [↦] *XS*]; *length vs* + *max-vars e* ≤ *length XS*;
?synth e s ⟧ ⟹ *?concl e e2 s XS e′ s′ ta vs*›
from ‹*bisim Vs* (*Val a*⌊*Val i*⌉ := *e*) *E2 xs*› **obtain** *E*
where *E2* = *Val a*⌊*Val i*⌉:=*E* **and** *bisim Vs e E xs* **by** *auto*
with *IH*[*of Vs E xs*] ‹*fv* (*Val a*⌊*Val i*⌉:=*e*) ⊆ *set Vs*› ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]› ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩›
‹*length Vs* + *max-vars* (*Val a*⌊*Val i*⌉:=*e*) ≤ *length xs*› ‹*?synth* (*Val a*⌊*Val i*⌉:=*e*) *s*›
show *?case* **by**(*fastforce elim*!: *sim-move01-expr*)
next
case *RedAAssStore* **thus** *?case* **by**(*auto intro*!: *exI*)
next
case (*FAssRed1 e s ta e′ s′ F D e2 Vs E2 xs*)

note *IH* = ‹⋀*vs e2 XS*. ⟦*bisim vs e e2 XS*; *fv e* ⊆ *set vs*; *lcl s* ⊆$_m$ [*vs* [↦] *XS*]; *length vs* + *max-vars e* ≤ *length XS*;
?synth e s ⟧ ⟹ *?concl e e2 s XS e′ s′ ta vs*›
from ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩› **have** ¬ *is-val e* **by** *auto*
with ‹*bisim Vs* (*e*·*F*{*D*} := *e2*) *E2 xs*› **obtain** *E*
where *E2* = *E*·*F*{*D*} := *compE1 Vs e2* **and** *bisim Vs e E xs* **and** ¬ *contains-insync e2* **by** *auto*
with *IH*[*of Vs E xs*] ‹*fv* (*e*·*F*{*D*} := *e2*) ⊆ *set Vs*› ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]› ‹¬ *is-val e*› ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩›
‹*length Vs* + *max-vars* (*e*·*F*{*D*} := *e2*) ≤ *length xs*› ‹*?synth* (*e*·*F*{*D*} := *e2*) *s*›
show *?case* **by**(*cases is-val e′*)(*fastforce elim*!: *sim-move01-expr*)+
next
case (*FAssRed2 e s ta e′ s′ v F D Vs E2 xs*)
note *IH* = ‹⋀*vs e2 XS*. ⟦*bisim vs e e2 XS*; *fv e* ⊆ *set vs*; *lcl s* ⊆$_m$ [*vs* [↦] *XS*]; *length vs* + *max-vars e* ≤ *length XS*;
?synth e s ⟧ ⟹ *?concl e e2 s XS e′ s′ ta vs*›
from ‹*bisim Vs* (*Val v*·*F*{*D*} := *e*) *E2 xs*› **obtain** *E*
where *E2* = *Val v*·*F*{*D*} := *E* **and** *bisim Vs e E xs* **by** *auto*
with *IH*[*of Vs E xs*] ‹*fv* (*Val v*·*F*{*D*} := *e*) ⊆ *set Vs*› ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]› ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩›
‹*length Vs* + *max-vars* (*Val v*·*F*{*D*} := *e*) ≤ *length xs*› ‹*?synth* (*Val v*·*F*{*D*} := *e*) *s*›
show *?case* **by**(*fastforce elim*!: *sim-move01-expr*)
next
case (*CASRed1 e s ta e′ s′ D F e2 e3 Vs E2 xs*)
note *IH* = ‹⋀*vs e2 XS*. ⟦*bisim vs e e2 XS*; *fv e* ⊆ *set vs*; *lcl s* ⊆$_m$ [*vs* [↦] *XS*]; *length vs* + *max-vars e* ≤ *length XS*;
?synth e s ⟧ ⟹ *?concl e e2 s XS e′ s′ ta vs*›
from ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩› **have** ¬ *is-val e* **by** *auto*
with ‹*bisim Vs - E2 xs*› **obtain** *E*
where *E2*: *E2* = *E*·*compareAndSwap*(*D*·*F*, *compE1 Vs e2*, *compE1 Vs e3*) **and** *bisim Vs e E xs*
and *sync*: ¬ *contains-insync e2* ¬ *contains-insync e3* **by**(*auto*)
with *IH*[*of Vs E xs*] ‹*fv - ⊆ set Vs*› ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]› ‹¬ *is-val e*› ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩›
‹*length Vs* + *max-vars - ≤ length xs*› ‹*?synth - s*›
obtain *ta′ e2′ xs′*
where *IH′*: *sim-move01* (*compP1 P*) *t ta e E* (*hp s*) *xs ta′ e2′* (*hp s′*) *xs′*
bisim Vs e′ e2′ xs′ lcl s′ ⊆$_m$ [*Vs* [↦] *xs′*]
by *auto*
show *?case*
proof(*cases is-val e′*)
case *True*
from ‹*fv - ⊆ set Vs*› *sync*
have *bisim Vs e2* (*compE1 Vs e2*) *xs′ bisim Vs e3* (*compE1 Vs e3*) *xs′* **by** *auto*
with *IH′ E2 True sync* ‹¬ *is-val e*› ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩› **show** *?thesis*
by(*cases is-val e2*)(*fastforce elim*!: *sim-move01-expr*)+
next
case *False* **with** *IH′ E2 sync* ‹¬ *is-val e*› ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩›
show *?thesis* **by**(*fastforce elim*!: *sim-move01-expr*)
qed
next
case (*CASRed2 e s ta e′ s′ v D F e3 Vs E2 xs*)
note *IH* = ‹⋀*vs e2 XS*. ⟦*bisim vs e e2 XS*; *fv e* ⊆ *set vs*; *lcl s* ⊆$_m$ [*vs* [↦] *XS*]; *length vs* + *max-vars e* ≤ *length XS*;
?synth e s ⟧ ⟹ *?concl e e2 s XS e′ s′ ta vs*›
from ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩› **have** ¬ *is-val e* **by** *auto*

with ‹*bisim Vs - E2 xs*› **obtain** *E*
where *E2* = *Val v·compareAndSwap(D·F, E, compE1 Vs e3)* **and** *bisim Vs e E xs* **and** ¬ *contains-insync e3* **by**(*auto*)
with *IH*[*of Vs E xs*] ‹*fv - ⊆ set Vs*› ‹*lcl s ⊆m* [*Vs* [↦] *xs*]› ‹¬ *is-val e*› ‹*extTA2J0 P,P,t ⊢* ⟨*e,s*⟩ *−ta→* ⟨*e′,s′*⟩›
‹*length Vs + max-vars - ≤ length xs*› ‹*?synth - s*›
show *?case* **by**(*cases is-val e′*)(*fastforce elim*!: *sim-move01-expr*)+
next
case (*CASRed3 e s ta e′ s′ v D F v′ Vs E2 xs*)
note *IH* = ‹⋀*vs e2 XS*. ⟦*bisim vs e e2 XS*; *fv e ⊆ set vs*; *lcl s ⊆m* [*vs* [↦] *XS*]; *length vs + max-vars e ≤ length XS*;
?synth e s ⟧ ⟹ *?concl e e2 s XS e′ s′ ta vs*›
from ‹*bisim Vs - E2 xs*› **obtain** *E*
where *E2* = *Val v·compareAndSwap(D·F, Val v′, E)* **and** *bisim Vs e E xs* **by** *auto*
with *IH*[*of Vs E xs*] ‹*fv - ⊆ set Vs*› ‹*lcl s ⊆m* [*Vs* [↦] *xs*]› ‹*extTA2J0 P,P,t ⊢* ⟨*e,s*⟩ *−ta→* ⟨*e′,s′*⟩›
‹*length Vs + max-vars - ≤ length xs*› ‹*?synth - s*›
show *?case* **by**(*fastforce elim*!: *sim-move01-expr*)
next
case (*CallObj e s ta e′ s′ M es Vs E2 xs*)
note *IH* = ‹⋀*vs e2 XS*. ⟦*bisim vs e e2 XS*; *fv e ⊆ set vs*; *lcl s ⊆m* [*vs* [↦] *XS*]; *length vs + max-vars e ≤ length XS*;
?synth e s ⟧ ⟹ *?concl e e2 s XS e′ s′ ta vs*›
from ‹*extTA2J0 P,P,t ⊢* ⟨*e,s*⟩ *−ta→* ⟨*e′,s′*⟩› **have** ¬ *is-val e* **by** *auto*
with ‹*bisim Vs* (*e·M*(*es*)) *E2 xs*› **obtain** *E*
where *E2* = *E·M*(*compEs1 Vs es*) **and** *bisim Vs e E xs* **and** ¬ *contains-insyncs es*
by(*auto simp add*: *compEs1-conv-map*)
with *IH*[*of Vs E xs*] ‹*fv* (*e·M*(*es*)) ⊆ *set Vs*› ‹*lcl s ⊆m* [*Vs* [↦] *xs*]›
‹*length Vs + max-vars* (*e·M*(*es*)) ≤ *length xs*› ‹*?synth* (*e·M*(*es*)) *s*›
show *?case* **by**(*cases is-val e′*)(*fastforce elim*!: *sim-move01-expr split*: *if-split-asm*)+
next
case (*CallParams es s ta es′ s′ v M Vs E2 xs*)
note *IH* = ‹⋀*vs es2 XS*. ⟦*bisims vs es es2 XS*; *fvs es ⊆ set vs*; *lcl s ⊆m* [*vs* [↦] *XS*]; *length vs + max-varss es ≤ length XS*;
?synths es s ⟧ ⟹ *?concls es es2 s XS es′ s′ ta vs*›
from ‹*bisim Vs* (*Val v·M*(*es*)) *E2 xs*› **obtain** *Es*
where *E2* = *Val v·M*(*Es*) **and** *bisims Vs es Es xs* **by** *auto*
moreover from ‹*extTA2J0 P,P,t ⊢* ⟨*es,s*⟩ [*−ta→*] ⟨*es′,s′*⟩› **have** ¬ *is-vals es* **by** *auto*
with ‹*?synth* (*Val v·M*(*es*)) *s*› **have** *?synths es s* **by**(*auto*)
moreover note *IH*[*of Vs Es xs*] ‹*fv* (*Val v·M*(*es*)) ⊆ *set Vs*› ‹*lcl s ⊆m* [*Vs* [↦] *xs*]›
‹*length Vs + max-vars* (*Val v·M*(*es*)) ≤ *length xs*›
ultimately show *?case* **by**(*fastforce elim*!: *sim-move01-CallParams*)
next
case (*RedCall s a U M Ts T pns body D vs Vs E2 xs*)
from ‹*typeof-addr* (*hp s*) *a* = ⌊*U*⌋›
have *call* (*addr a·M*(*map Val vs*)) = ⌊(*a, M, vs*)⌋ **by** *auto*
with ‹*?synth* (*addr a·M*(*map Val vs*)) *s*› **have** *synthesized-call P* (*hp s*) (*a, M, vs*) **by** *auto*
with ‹*typeof-addr* (*hp s*) *a* = ⌊*U*⌋› ‹*P ⊢ class-type-of U sees M*: *Ts→T* = ⌊(*pns, body*)⌋ *in D*›
have *False* **by**(*auto simp add*: *synthesized-call-conv dest*: *sees-method-fun*)
thus *?case* **..**
next
case (*RedCallExternal s a T M Ts Tr D vs ta va h′ ta′ e′ s′ Vs E2 xs*)
from ‹*bisim Vs* (*addr a·M*(*map Val vs*)) *E2 xs*› **have** *E2* = *addr a·M*(*map Val vs*) **by** *auto*
moreover note ‹*P ⊢ class-type-of T sees M*: *Ts→Tr* = *Native in D*› ‹*typeof-addr* (*hp s*) *a* = ⌊*T*⌋›
‹*ta′* = *extTA2J0 P ta*›

‹e' = *extRet2J* (*addr* $a{\cdot}M(map\ Val\ vs)$) va› ‹s' = (h', *lcl* s)› ‹$P,t \vdash \langle a{\cdot}M(vs),hp\ s\rangle\ -ta{\rightarrow}ext\ \langle va,h'\rangle$›
‹*lcl* $s \subseteq_m [Vs\ [\mapsto]\ xs]$›
moreover from *wf* ‹$P,t \vdash \langle a{\cdot}M(vs),hp\ s\rangle\ -ta{\rightarrow}ext\ \langle va,h'\rangle$›
have *ta-bisim01* (*extTA2J0 P ta*) (*extTA2J1* (*compP1 P*) *ta*) **by**(*rule red-external-ta-bisim01*)
moreover from ‹$P,t \vdash \langle a{\cdot}M(vs),hp\ s\rangle\ -ta{\rightarrow}ext\ \langle va,h'\rangle$› ‹*typeof-addr* (*hp s*) $a = \lfloor T \rfloor$›
‹$P \vdash$ *class-type-of T sees M*: $Ts{\rightarrow}Tr = Native$ *in D*›
have τ*external-defs D M* $\Longrightarrow h' = hp\ s \wedge ta = \varepsilon$
by(*fastforce dest*: τ*external'-red-external-heap-unchanged* τ*external'-red-external-TA-empty simp add*: τ*external'-def* τ*external-def*)
ultimately show *?case*
by(*cases va*)(*fastforce intro*!: *exI*[**where** *x*=*ta*] *intro*: *Red1CallExternal simp add*: *map-eq-append-conv sim-move01-def dest*: *sees-method-fun simp del*: *split-paired-Ex*)+
next
case (*BlockRed e h x V vo ta e' h' x' T Vs E2 xs*)
note *IH* = ‹$\bigwedge$*vs e2 XS*. ⟦*bisim vs e e2 XS*; *fv e* $\subseteq$ *set vs*; *lcl* (h, $x(V := vo)$) $\subseteq_m$ [*vs* [$\mapsto$] *XS*];
length vs + *max-vars e* $\leq$ *length XS*; *?synth e* (h, ($x(V := vo)$))⟧
$\Longrightarrow$ *?concl e e2* (h, $x(V := vo)$) *XS e'* (h', x') *ta vs*›
note *red* = ‹*extTA2J0 P*,*P*,*t* $\vdash \langle e,(h,\ x(V := vo))\rangle\ -ta{\rightarrow}\ \langle e',(h',\ x')\rangle$›
note *len* = ‹*length Vs* + *max-vars* {V:T=*vo*; *e*} $\leq$ *length xs*›
from ‹*fv* {V:T=*vo*; *e*} $\subseteq$ *set Vs*› **have** *fv*: *fv e* $\subseteq$ *set* (*Vs*@[V]) **by** *auto*
from ‹*bisim Vs* {V:T=*vo*; *e*} *E2 xs*› **show** *?case*
proof(*cases rule*: *bisim-cases*(7)[*consumes 1*, *case-names BlockNone BlockSome BlockSomeNone*])
case (*BlockNone E'*)
with *red IH*[*of Vs*@[V] *E' xs*] *fv* ‹*lcl* (h, x) $\subseteq_m$ [*Vs* [$\mapsto$] *xs*]›
‹*length Vs* + *max-vars* {V:T=*vo*; *e*} $\leq$ *length xs*› ‹*?synth* {V:T=*vo*; *e*} (h, x)›
obtain *TA' e2' xs'* **where** *red'*: *sim-move01* (*compP1 P*) *t ta e E' h xs TA' e2' h' xs'*
and *bisim'*: *bisim* (*Vs* @ [V]) *e' e2' xs' x'* $\subseteq_m$ [*Vs* @ [V] [$\mapsto$] *xs'*] **by** *auto*
from *red'* ‹*length Vs* + *max-vars* {V:T=*vo*; *e*} $\leq$ *length xs*›
have *length* (*Vs*@[V]) + *max-vars e* $\leq$ *length xs'*
by(*fastforce simp add*: *sim-move01-def dest*: *red1-preserves-len* τ*red1t-preserves-len* τ*red1r-preserves-len split*: *if-split-asm*)
with ‹$x' \subseteq_m$ [*Vs* @ [V] [$\mapsto$] *xs'*]› **have** $x' \subseteq_m$ [*Vs* [$\mapsto$] *xs'*, $V \mapsto$ *xs'* ! *length Vs*] **by**(*simp*)
moreover
{ **assume** $V \in$ *set Vs*
hence *hidden* (*Vs* @ [V]) (*index Vs V*) **by**(*rule hidden-index*)
with ‹*bisim* (*Vs* @ [V]) *e E' xs*› **have** *unmod E'* (*index Vs V*)
by −(*rule hidden-bisim-unmod*)
moreover from ‹*length Vs* + *max-vars* {V:T=*vo*; *e*} $\leq$ *length xs*› ‹$V \in$ *set Vs*›
have *index Vs V* < *length xs* **by**(*auto intro*: *index-less-aux*)
ultimately have *xs* ! *index Vs V* = *xs'* ! *index Vs V*
using *sim-move01-preserves-unmod*[*OF red'*] **by**(*simp*) }
moreover from *red'* **have** *length xs* = *length xs'* **by**(*rule sim-move01-preserves-len*[*symmetric*])
ultimately have *rel*: $x'(V := x\ V) \subseteq_m$ [*Vs* [$\mapsto$] *xs'*]
using ‹*lcl* (h, x) $\subseteq_m$ [*Vs* [$\mapsto$] *xs*]› ‹*length Vs* + *max-vars* {V:T=*vo*; *e*} $\leq$ *length xs*›
by(*auto intro*: *Block-lem*)
show *?thesis*
proof(*cases x' V*)
case *None*
with *red' bisim' BlockNone len*
show *?thesis* **by**(*fastforce simp del*: *split-paired-Ex fun-upd-apply intro*: *rel*)
next
case (*Some v*)
moreover

```
    with ‹x' ⊆m [Vs @ [V] [↦] xs']› have [Vs @ [V] [↦] xs'] V = ⌊v⌋
      by(auto simp add: map-le-def dest: bspec)
    moreover
    from ‹length Vs + max-vars {V:T=vo; e} ≤ length xs› have length Vs < length xs by auto
    ultimately have xs' ! length Vs = v using ‹length xs = length xs'› by(simp)
    with red' bisim' BlockNone Some len
    show ?thesis by(fastforce simp del: fun-upd-apply intro: rel)
   qed
  next
   case (BlockSome E' v)
   with red IH[of Vs@[V] E' xs[length Vs := v]] fv ‹lcl (h, x) ⊆m [Vs [↦] xs]›
    ‹length Vs + max-vars {V:T=vo; e} ≤ length xs› ‹?synth {V:T=vo; e} (h, x)›
   obtain TA' e2' xs' where red': sim-move01 (compP1 P) t ta e E' h (xs[length Vs := v]) TA' e2'
h' xs'
    and bisim': bisim (Vs @ [V]) e' e2' xs' x' ⊆m [Vs @ [V] [↦] xs'] by auto
   from red' ‹length Vs + max-vars {V:T=vo; e} ≤ length xs›
   have length (Vs@[V]) + max-vars e ≤ length xs' by(auto dest: sim-move01-preserves-len)
   with ‹x' ⊆m [Vs @ [V] [↦] xs']› have x' ⊆m [Vs [↦] xs', V ↦ xs' ! length Vs] by(simp)
   moreover
   { assume V ∈ set Vs
    hence hidden (Vs @ [V]) (index Vs V) by(rule hidden-index)
    with ‹bisim (Vs @ [V]) e E' (xs[length Vs := v])› have unmod E' (index Vs V)
     by −(rule hidden-bisim-unmod)
    moreover from ‹length Vs + max-vars {V:T=vo; e} ≤ length xs› ‹V ∈ set Vs›
    have index Vs V < length xs by(auto intro: index-less-aux)
    moreover from ‹length Vs + max-vars {V:T=vo; e} ≤ length xs› ‹V ∈ set Vs›
    have (xs[length Vs := v]) ! index Vs V = xs ! index Vs V by(simp)
    ultimately have xs ! index Vs V = xs' ! index Vs V
     using sim-move01-preserves-unmod[OF red', of index Vs V] by(simp) }
   moreover from red' have length xs = length xs' by(auto dest: sim-move01-preserves-len)
   ultimately have rel: x'(V := x V) ⊆m [Vs [↦] xs']
    using ‹lcl (h, x) ⊆m [Vs [↦] xs]› ‹length Vs + max-vars {V:T=vo; e} ≤ length xs›
    by(auto intro: Block-lem)
   from BlockSome red obtain v' where Some: x' V = ⌊v'⌋ by(auto dest!: red-lcl-incr)
   with ‹x' ⊆m [Vs @ [V] [↦] xs']› have [Vs @ [V] [↦] xs'] V = ⌊v'⌋
    by(auto simp add: map-le-def dest: bspec)
   moreover
   from ‹length Vs + max-vars {V:T=vo; e} ≤ length xs› have length Vs < length xs by auto
   ultimately have xs' ! length Vs = v' using ‹length xs = length xs'› by(simp)
   with red' bisim' BlockSome Some ‹length Vs < length xs›
   show ?thesis by(fastforce simp del: fun-upd-apply intro: rel)
  next
   case (BlockSomeNone E')
   with red IH[of Vs@[V] E' xs] fv ‹lcl (h, x) ⊆m [Vs [↦] xs]›
    ‹length Vs + max-vars {V:T=vo; e} ≤ length xs› ‹?synth {V:T=vo; e} (h, x)›
   obtain TA' e2' xs' where red': sim-move01 (compP1 P) t ta e E' h xs TA' e2' h' xs'
    and IH': bisim (Vs @ [V]) e' e2' xs' x' ⊆m [Vs @ [V] [↦] xs'] by auto
   from red' ‹length Vs + max-vars {V:T=vo; e} ≤ length xs›
   have length (Vs@[V]) + max-vars e ≤ length xs' by(auto dest: sim-move01-preserves-len)
   with ‹x' ⊆m [Vs @ [V] [↦] xs']› have x' ⊆m [Vs [↦] xs', V ↦ xs' ! length Vs] by(simp)
   moreover
   { assume V ∈ set Vs
    hence hidden (Vs @ [V]) (index Vs V) by(rule hidden-index)
    with ‹bisim (Vs @ [V]) e E' xs› have unmod E' (index Vs V)
```

```
      by −(rule hidden-bisim-unmod)
    moreover from ‹length Vs + max-vars {V:T=vo; e} ≤ length xs› ‹V ∈ set Vs›
    have index Vs V < length xs by(auto intro: index-less-aux)
    moreover from ‹length Vs + max-vars {V:T=vo; e} ≤ length xs› ‹V ∈ set Vs›
    have (xs[length Vs := v]) ! index Vs V = xs ! index Vs V by(simp)
    ultimately have xs ! index Vs V = xs' ! index Vs V
      using sim-move01-preserves-unmod[OF red', of index Vs V] by(simp) }
  moreover from red' have length xs = length xs' by(auto dest: sim-move01-preserves-len)
  ultimately have rel: x'(V := x V) ⊆m [Vs [↦] xs']
    using ‹lcl (h, x) ⊆m [Vs [↦] xs]› ‹length Vs + max-vars {V:T=vo; e} ≤ length xs›
    by(auto intro: Block-lem)
  from BlockSomeNone red obtain v' where Some: x' V = ⌊v'⌋ by(auto dest!: red-lcl-incr)
  with ‹x' ⊆m [Vs @ [V] [↦] xs']› have [Vs @ [V] [↦] xs'] V = ⌊v'⌋
    by(auto simp add: map-le-def dest: bspec)
  moreover
  from ‹length Vs + max-vars {V:T=vo; e} ≤ length xs› have length Vs < length xs by auto
  ultimately have xs' ! length Vs = v' using ‹length xs = length xs'› by(simp)
  with red' IH' BlockSomeNone Some ‹length Vs < length xs›
  show ?thesis by(fastforce simp del: fun-upd-apply intro: rel)
 qed
next
 case (RedBlock V T vo u s Vs E2 xs)
 from ‹bisim Vs {V:T=vo; Val u} E2 xs› obtain vo'
  where [simp]: E2 = {length Vs:T=vo'; Val u} by auto
 from RedBlock show ?case
 proof(cases vo)
  case (Some v)
  with ‹bisim Vs {V:T=vo; Val u} E2 xs›
  have vo': case vo' of None ⇒ xs ! length Vs = v | Some v' ⇒ v = v' by auto
  have sim-move01 (compP1 P) t ε {V:T=vo; Val u} E2 (hp s) xs ε (Val u) (hp s) (xs[length Vs
:= v])
  proof(cases vo')
   case None with vo'
   have xs[length Vs := v] = xs by auto
   thus ?thesis using Some None by auto
  next
   case Some
   from ‹length Vs + max-vars {V:T=vo; Val u} ≤ length xs› have length Vs < length xs by simp
   with vo' Some show ?thesis using ‹vo = Some v› by auto
  qed
  thus ?thesis using RedBlock by fastforce
 qed fastforce
next
 case SynchronizedNull thus ?case by fastforce
next
 case (LockSynchronized a e s Vs E2 xs)
 from ‹bisim Vs (sync(addr a) e) E2 xs›
 have E2: E2 = sync_{length Vs} (addr a) (compE1 (Vs@[fresh-var Vs]) e)
  and sync: ¬ contains-insync e by auto
 moreover have fresh-var Vs ∉ set Vs by(rule fresh-var-fresh)
 with ‹fv (sync(addr a) e) ⊆ set Vs› have fresh-var Vs ∉ fv e by auto
 from E2 ‹fv (sync(addr a) e) ⊆ set Vs› sync
 have bisim (Vs@[fresh-var Vs]) e (compE1 (Vs@[fresh-var Vs]) e) (xs[length Vs := Addr a])
  by(auto intro!: compE1-bisim)
```

hence *bisim Vs* (*insync*(*a*) *e*) (*insync*$_{length\ Vs}$ (*a*) (*compE1* (*Vs*@[*fresh-var Vs*]) *e*)) (*xs*[*length Vs* := *Addr a*])
using ‹*fresh-var Vs* ∉ *fv e*› ‹*length Vs* + *max-vars* (*sync*(*addr a*) *e*) ≤ *length xs*› **by** *auto*
moreover from ‹*length Vs* + *max-vars* (*sync*(*addr a*) *e*) ≤ *length xs*›
have *False*,*compP1 P*,*t* ⊢*1* ⟨*sync*$_{length\ Vs}$ (*addr a*) (*compE1* (*Vs*@[*fresh-var Vs*]) *e*), (*hp s*, *xs*)⟩
−{⟦*Lock*→*a*, *SyncLock a*⟧}→
⟨*insync*$_{length\ Vs}$ (*a*) (*compE1* (*Vs*@[*fresh-var Vs*]) *e*), (*hp s*, *xs*[*length Vs* := *Addr a*])⟩
by −(*rule Lock1Synchronized*, *auto*)
hence *sim-move01* (*compP1 P*) *t* {⟦*Lock*→*a*, *SyncLock a*⟧} (*sync*(*addr a*) *e*) *E2* (*hp s*) *xs* {⟦*Lock*→*a*, *SyncLock a*⟧} (*insync*$_{length\ Vs}$ (*a*) (*compE1* (*Vs*@[*fresh-var Vs*]) *e*)) (*hp s*) (*xs*[*length Vs* := *Addr a*])
using *E2* **by**(*fastforce simp add*: *sim-move01-def ta-bisim-def*)
moreover have *zip Vs* (*xs*[*length Vs* := *Addr a*]) = (*zip Vs xs*)[*length Vs* := (*arbitrary*, *Addr a*)]
by(*rule sym*)(*simp add*: *update-zip*)
hence *zip Vs* (*xs*[*length Vs* := *Addr a*]) = *zip Vs xs* **by** *simp*
with ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]› **have** *lcl s* ⊆$_m$ [*Vs* [↦] *xs*[*length Vs* := *Addr a*]]
by(*auto simp add*: *map-le-def map-upds-def*)
ultimately show *?case* **using** ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]› **by** *fastforce*
next
case (*SynchronizedRed2 e s ta e′ s′ a Vs E2 xs*)
note *IH* = ‹⋀*vs e2 XS*. ⟦*bisim vs e e2 XS*; *fv e* ⊆ *set vs*; *lcl s* ⊆$_m$ [*vs* [↦] *XS*]; *length vs* + *max-vars e* ≤ *length XS*;
?synth e s ⟧ ⟹ *?concl e e2 s XS e′ s′ ta vs*›
from ‹*bisim Vs* (*insync*(*a*) *e*) *E2 xs*› **obtain** *E*
where *E2*: *E2* = *insync*$_{length\ Vs}$ (*a*) *E* **and** *bisim*: *bisim* (*Vs*@[*fresh-var Vs*]) *e E xs*
and *xsa*: *xs* ! *length Vs* = *Addr a* **by** *auto*
from ‹*fv* (*insync*(*a*) *e*) ⊆ *set Vs*› *fresh-var-fresh*[*of Vs*] **have** *fv*: *fresh-var Vs* ∉ *fv e* **by** *auto*
from ‹*length Vs* + *max-vars* (*insync*(*a*) *e*) ≤ *length xs*› **have** *length Vs* < *length xs* **by** *simp*
{ assume *lcl s* (*fresh-var Vs*) ≠ *None*
then obtain *v* **where** *lcl s* (*fresh-var Vs*) = ⌊*v*⌋ **by** *auto*
with ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]› **have** [*Vs* [↦] *xs*] (*fresh-var Vs*) = ⌊*v*⌋
by(*auto simp add*: *map-le-def dest*: *bspec*)
hence *fresh-var Vs* ∈ *set Vs*
by(*auto simp add*: *map-upds-def set-zip dest*!: *map-of-SomeD*)
moreover have *fresh-var Vs* ∉ *set Vs* **by**(*rule fresh-var-fresh*)
ultimately have *False* **by** *contradiction* **}**
hence *lcl s* (*fresh-var Vs*) = *None* **by**(*cases lcl s* (*fresh-var Vs*), *auto*)
hence (*lcl s*)(*fresh-var Vs* := *None*) = *lcl s* **by**(*auto intro*: *ext*)
moreover from ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]›
have (*lcl s*)(*fresh-var Vs* := *None*) ⊆$_m$ [*Vs* [↦] *xs*, *fresh-var Vs* ↦ *xs* ! *length Vs*] **by**(*simp*)
ultimately have *lcl s* ⊆$_m$ [*Vs* @ [*fresh-var Vs*] [↦] *xs*]
using ‹*length Vs* < *length xs*› **by**(*auto*)
with *IH*[*of Vs*@[*fresh-var Vs*] *E xs*] ‹*fv* (*insync*(*a*) *e*) ⊆ *set Vs*› *bisim*
‹*length Vs* + *max-vars* (*insync*(*a*) *e*) ≤ *length xs*› ‹*?synth* (*insync*(*a*) *e*) *s*›
obtain *TA′ e2′ xs′* **where** *IH′*: *sim-move01* (*compP1 P*) *t ta e E* (*hp s*) *xs TA′ e2′* (*hp s′*) *xs′*
bisim (*Vs* @ [*fresh-var Vs*]) *e′ e2′ xs′ lcl s′* ⊆$_m$ [*Vs* @ [*fresh-var Vs*] [↦] *xs′*] **by** *auto*
from ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩› **have** *dom* (*lcl s′*) ⊆ *dom* (*lcl s*) ∪ *fv e* **by**(*auto dest*: *red-dom-lcl*)
with *fv* ‹*lcl s* (*fresh-var Vs*) = *None*› **have** (*fresh-var Vs*) ∉ *dom* (*lcl s′*) **by** *auto*
hence *lcl s′* (*fresh-var Vs*) = *None* **by** *auto*
moreover from *IH′* **have** *length xs* = *length xs′* **by**(*auto dest*: *sim-move01-preserves-len*)
moreover note ‹*lcl s′* ⊆$_m$ [*Vs* @ [*fresh-var Vs*] [↦] *xs′*]› ‹*length Vs* < *length xs*›
ultimately have *lcl s′* ⊆$_m$ [*Vs* [↦] *xs′*] **by**(*auto simp add*: *map-le-def dest*: *bspec*)
moreover from *bisim fv* **have** *unmod E* (*length Vs*) **by**(*auto intro*: *bisim-fv-unmod*)
with *IH′* ‹*length Vs* < *length xs*› **have** *xs* ! *length Vs* = *xs′* ! *length Vs*

by(*auto dest*!: *sim-move01-preserves-unmod*)
with *xsa* **have** *xs'* ! *length Vs* = *Addr a* **by** *simp*
ultimately show *?case* **using** *IH' E2* **by**(*fastforce*)
next
case (*UnlockSynchronized a v s Vs E2 xs*)
from ‹*bisim Vs* (*insync*(*a*) *Val v*) *E2 xs*› **have** *E2*: *E2* = $insync_{length\ Vs}$ (*a*) *Val v*
and *xsa*: *xs* ! *length Vs* = *Addr a* **by** *auto*
moreover from ‹*length Vs* + *max-vars* (*insync*(*a*) *Val v*) ≤ *length xs*› *xsa*
have *False*,*compP1 P*,*t* ⊢1 ⟨$insync_{length\ Vs}$ (*a*) (*Val v*), (*hp s*, *xs*)⟩ −{|*Unlock*→*a*, *SyncUnlock a*|}→ ⟨*Val v*, (*hp s*, *xs*)⟩
by−(*rule Unlock1Synchronized*, *auto*)
hence *sim-move01* (*compP1 P*) *t* {|*Unlock*→*a*, *SyncUnlock a*|} (*insync*(*a*) *Val v*) ($insync_{length\ Vs}$ (*a*) *Val v*) (*hp s*) *xs* {|*Unlock*→*a*, *SyncUnlock a*|} (*Val v*) (*hp s*) *xs*
by(*fastforce simp add*: *sim-move01-def ta-bisim-def*)
ultimately show *?case* **using** ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]› **by** *fastforce*
next
case (*RedWhile b c s Vs E2 xs*)
from ‹*bisim Vs* (*while* (*b*) *c*) *E2 xs*› **have** *E2* = *while* (*compE1 Vs b*) (*compE1 Vs c*)
and *sync*: ¬ *contains-insync b* ¬ *contains-insync c* **by** *auto*
moreover have *False*,*compP1 P*,*t* ⊢1 ⟨*while*(*compE1 Vs b*) (*compE1 Vs c*), (*hp s*, *xs*)⟩
−*ε*→ ⟨*if* (*compE1 Vs b*) (*compE1 Vs c*;;*while*(*compE1 Vs b*) (*compE1 Vs c*)) *else unit*, (*hp s*, *xs*)⟩
by(*rule Red1While*)
hence *sim-move01* (*compP1 P*) *t ε* (*while* (*b*) *c*) (*while* (*compE1 Vs b*) (*compE1 Vs c*)) (*hp s*) *xs ε* (*if* (*compE1 Vs b*) (*compE1 Vs c*;;*while*(*compE1 Vs b*) (*compE1 Vs c*)) *else unit*) (*hp s*) *xs*
by(*auto simp add*: *sim-move01-def*)
moreover from ‹*fv* (*while* (*b*) *c*) ⊆ *set Vs*› *sync*
have *bisim Vs* (*if* (*b*) (*c*;; *while* (*b*) *c*) *else unit*)
(*if* (*compE1 Vs b*) (*compE1 Vs* (*c*;; *while*(*b*) *c*)) *else* (*compE1 Vs unit*)) *xs*
by −(*rule bisimCond*, *auto*)
ultimately show *?case* **using** ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]›
by(*simp*)((*rule exI*)+, *erule conjI*, *auto*)
next
case (*RedTryCatch s a D C V e2 Vs E2 xs*)
thus *?case* **by**(*auto intro*!: *exI*)(*auto intro*!: *compE1-bisim*)
next
case *RedTryFail* **thus** *?case* **by**(*auto intro*!: *exI*)
next
case (*ListRed1 e s ta e' s' es Vs ES2 xs*)
note *IH* = ‹⋀*vs e2 XS*. ⟦*bisim vs e e2 XS*; *fv e* ⊆ *set vs*; *lcl s* ⊆$_m$ [*vs* [↦] *XS*]; *length vs* + *max-vars e* ≤ *length XS*;
?synth e s⟧ ⟹ *?concl e e2 s XS e' s' ta vs*›
from ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e'*,*s'*⟩› **have** ¬ *is-val e* **by** *auto*
with ‹*bisims Vs* (*e* # *es*) *ES2 xs*› **obtain** *E'*
where *bisim Vs e E' xs* **and** *ES2*: *ES2* = *E'* # *compEs1 Vs es*
and *sync*: ¬ *contains-insyncs es* **by**(*auto simp add*: *compEs1-conv-map*)
with *IH*[*of Vs E' xs*] ‹*fvs* (*e* # *es*) ⊆ *set Vs*› ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]› ‹*extTA2J0 P*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e'*,*s'*⟩›
‹*length Vs* + *max-varss* (*e* # *es*) ≤ *length xs*› ‹*?synths* (*e* # *es*) *s*› ‹¬ *is-val e*›
show *?case* **by**(*cases is-val e'*)(*fastforce elim*!: *sim-moves01-expr split*: *if-split-asm*)+
next
case (*ListRed2 es s ta es' s' v Vs ES2 xs*)
thus *?case* **by**(*fastforce elim*!: *bisims-cases elim*!: *sim-moves01-expr*)
next

case *CallThrowParams* **thus** *?case*
by(*fastforce elim*!:*bisim-cases simp add*: *bisims-map-Val-Throw*)
next
case (*BlockThrow V T vo a s Vs e2 xs*) **thus** *?case*
by(*cases vo*)(*fastforce elim*!: *bisim-cases*)+
next
case (*SynchronizedThrow2 a ad s Vs E2 xs*)
from ‹*bisim Vs* (*insync*(*a*) *Throw ad*) *E2 xs*› **have** *xs* ! *length Vs* = *Addr a* **by** *auto*
with ‹*length Vs* + *max-vars* (*insync*(*a*) *Throw ad*) ≤ *length xs*›
have *False*,*compP1 P*,*t* ⊢*1* ⟨*insync*$_{length\ Vs}$ (*a*) *Throw ad*, (*hp s*, *xs*)⟩ −{|*Unlock*→*a*, *SyncUnlock a*|}→ ⟨*Throw ad*, (*hp s*, *xs*)⟩
by−(*rule Synchronized1Throw2*, *auto*)
hence *sim-move01* (*compP1 P*) *t* {|*Unlock*→*a*, *SyncUnlock a*|} (*insync*(*a*) *Throw ad*) (*insync*$_{length\ Vs}$ (*a*) *Throw ad*) (*hp s*) *xs* {|*Unlock*→*a*, *SyncUnlock a*|} (*Throw ad*) (*hp s*) *xs*
by(*fastforce simp add*: *sim-move01-def ta-bisim-def fun-eq-iff expand-finfun-eq finfun-upd-apply ta-upd-simps split*: *if-split-asm*)
moreover note ‹*lcl s* ⊆$_m$ [*Vs* [↦] *xs*]› ‹*bisim Vs* (*insync*(*a*) *Throw ad*) *E2 xs*›
ultimately show *?case* **by**(*fastforce*)
next
case *InstanceOfRed* **thus** *?case* **by**(*fastforce*)
next
case *RedInstanceOf* **thus** *?case* **by**(*auto intro*!: *exI*)
next
case *InstanceOfThrow* **thus** *?case* **by** *fastforce*
qed(*fastforce simp del*: *fun-upd-apply split*: *if-split-asm*)+

end

declare *max-dest* [*iff del*]

declare *split-paired-Ex* [*simp del*]

primrec *countInitBlock* :: ('*a*, '*b*, '*addr*) *exp* ⇒ *nat*
and *countInitBlocks* :: ('*a*, '*b*, '*addr*) *exp list* ⇒ *nat*
where
countInitBlock (*new C*) = *0*
| *countInitBlock* (*newA T*⌊*e*⌉) = *countInitBlock e*
| *countInitBlock* (*Cast T e*) = *countInitBlock e*
| *countInitBlock* (*e instanceof T*) = *countInitBlock e*
| *countInitBlock* (*Val v*) = *0*
| *countInitBlock* (*Var V*) = *0*
| *countInitBlock* (*V* := *e*) = *countInitBlock e*
| *countInitBlock* (*e* «*bop*» *e*′) = *countInitBlock e* + *countInitBlock e*′
| *countInitBlock* (*a*⌊*i*⌉) = *countInitBlock a* + *countInitBlock i*
| *countInitBlock* (*AAss a i e*) = *countInitBlock a* + *countInitBlock i* + *countInitBlock e*
| *countInitBlock* (*a*·*length*) = *countInitBlock a*
| *countInitBlock* (*e*·*F*{*D*}) = *countInitBlock e*
| *countInitBlock* (*FAss e F D e*′) = *countInitBlock e* + *countInitBlock e*′
| *countInitBlock* (*e*·*compareAndSwap*(*D*·*F*, *e*′, *e*′′)) =
countInitBlock e + *countInitBlock e*′ + *countInitBlock e*′′
| *countInitBlock* (*e*·*M*(*es*)) = *countInitBlock e* + *countInitBlocks es*
| *countInitBlock* ({*V*:*T*=*vo*; *e*}) = (*case vo of None* ⇒ *0* | *Some v* ⇒ *1*) + *countInitBlock e*
| *countInitBlock* (*sync*$_{V'}$ (*e*) *e*′) = *countInitBlock e* + *countInitBlock e*′
| *countInitBlock* (*insync*$_{V'}$ (*ad*) *e*) = *countInitBlock e*

| *countInitBlock* (*e*;;*e'*) = *countInitBlock e* + *countInitBlock e'*
| *countInitBlock* (*if* (*e*) *e1 else e2*) = *countInitBlock e* + *countInitBlock e1* + *countInitBlock e2*
| *countInitBlock* (*while*(*b*) *e*) = *countInitBlock b* + *countInitBlock e*
| *countInitBlock* (*throw e*) = *countInitBlock e*
| *countInitBlock* (*try e catch*(*C V*) *e'*) = *countInitBlock e* + *countInitBlock e'*

| *countInitBlocks* [] = *0*
| *countInitBlocks* (*e* # *es*) = *countInitBlock e* + *countInitBlocks es*

context *J0-J1-heap-base* **begin**

lemmas *τred0r-expr* =
NewArray-τred0r-xt Cast-τred0r-xt InstanceOf-τred0r-xt BinOp-τred0r-xt1 BinOp-τred0r-xt2 LAss-τred0r
AAcc-τred0r-xt1 AAcc-τred0r-xt2 AAss-τred0r-xt1 AAss-τred0r-xt2 AAss-τred0r-xt3
ALength-τred0r-xt FAcc-τred0r-xt FAss-τred0r-xt1 FAss-τred0r-xt2
CAS-τred0r-xt1 CAS-τred0r-xt2 CAS-τred0r-xt3 Call-τred0r-obj
Call-τred0r-param Block-τred0r-xt Sync-τred0r-xt InSync-τred0r-xt
Seq-τred0r-xt Cond-τred0r-xt Throw-τred0r-xt Try-τred0r-xt

lemmas *τred0t-expr* =
NewArray-τred0t-xt Cast-τred0t-xt InstanceOf-τred0t-xt BinOp-τred0t-xt1 BinOp-τred0t-xt2 LAss-τred0t
AAcc-τred0t-xt1 AAcc-τred0t-xt2 AAss-τred0t-xt1 AAss-τred0t-xt2 AAss-τred0t-xt3
ALength-τred0t-xt FAcc-τred0t-xt FAss-τred0t-xt1 FAss-τred0t-xt2
CAS-τred0t-xt1 CAS-τred0t-xt2 CAS-τred0t-xt3 Call-τred0t-obj
Call-τred0t-param Block-τred0t-xt Sync-τred0t-xt InSync-τred0t-xt
Seq-τred0t-xt Cond-τred0t-xt Throw-τred0t-xt Try-τred0t-xt

declare *τred0r-expr* [*elim*!]
declare *τred0t-expr* [*elim*!]

definition *sim-move10* ::
'addr J-prog ⇒ *'thread-id* ⇒ (*'addr*, *'thread-id*, *'heap*) *external-thread-action* ⇒ *'addr expr1* ⇒ *'addr expr1* ⇒ *'addr expr*
⇒ *'heap* ⇒ *'addr locals* ⇒ (*'addr*, *'thread-id*, *'heap*) *J0-thread-action* ⇒ *'addr expr* ⇒ *'heap* ⇒ *'addr locals* ⇒ *bool*
where
sim-move10 P t ta1 e1 e1' e h xs ta e' h' xs' ⟷ ¬ *final e1* ∧
(*if τmove1 P h e1 then* (*τred0t* (*extTA2J0 P*) *P t h* (*e*, *xs*) (*e'*, *xs'*) ∨ *countInitBlock e1'* < *countInitBlock e1* ∧ *e'* = *e* ∧ *xs'* = *xs*) ∧ *h'* = *h* ∧ *ta1* = ε ∧ *ta* = ε
else ta-bisim01 ta (*extTA2J1* (*compP1 P*) *ta1*) ∧
(*if call e* = *None* ∨ *call1 e1* = *None*
then (∃ *e'' xs''*. *τred0r* (*extTA2J0 P*) *P t h* (*e*, *xs*) (*e''*, *xs''*) ∧ *extTA2J0 P*,*P*,*t* ⊢ ⟨*e''*, (*h*, *xs''*)⟩ −*ta*→ ⟨*e'*, (*h'*, *xs'*)⟩ ∧ *no-call P h e''* ∧ ¬ *τmove0 P h e''*)
else extTA2J0 P,*P*,*t* ⊢ ⟨*e*, (*h*, *xs*)⟩ −*ta*→ ⟨*e'*, (*h'*, *xs'*)⟩ ∧ *no-call P h e* ∧ ¬ *τmove0 P h e*))

definition *sim-moves10* ::
'addr J-prog ⇒ *'thread-id* ⇒ (*'addr*, *'thread-id*, *'heap*) *external-thread-action* ⇒ *'addr expr1 list* ⇒ *'addr expr1 list*
⇒ *'addr expr list* ⇒ *'heap* ⇒ *'addr locals* ⇒ (*'addr*, *'thread-id*, *'heap*) *J0-thread-action* ⇒ *'addr expr list* ⇒ *'heap*
⇒ *'addr locals* ⇒ *bool*
where
sim-moves10 P t ta1 es1 es1' es h xs ta es' h' xs' ⟷ ¬ *finals es1* ∧
(*if τmoves1 P h es1 then* (*τreds0t* (*extTA2J0 P*) *P t h* (*es*, *xs*) (*es'*, *xs'*) ∨ *countInitBlocks es1'* <

countInitBlocks es1 ∧ *es'* = *es* ∧ *xs'* = *xs*) ∧ *h'* = *h* ∧ *ta1* = ε ∧ *ta* = ε
else ta-bisim01 ta (*extTA2J1* (*compP1 P*) *ta1*) ∧
(*if calls es* = *None* ∨ *calls1 es1* = *None*
then (∃ *es'' xs''*. τ*reds0r* (*extTA2J0 P*) *P t h* (*es*, *xs*) (*es''*, *xs''*) ∧ *extTA2J0 P*,*P*,*t* ⊢ ⟨*es''*, (*h*, *xs''*)⟩ [−*ta*→] ⟨*es'*, (*h'*, *xs'*)⟩ ∧ *no-calls P h es''* ∧ ¬ τ*moves0 P h es''*)
else extTA2J0 P,*P*,*t* ⊢ ⟨*es*, (*h*, *xs*)⟩ [−*ta*→] ⟨*es'*, (*h'*, *xs'*)⟩ ∧ *no-calls P h es* ∧ ¬ τ*moves0 P h es*))

lemma *sim-move10-expr*:
assumes *sim-move10 P t ta1 e1 e1' e h xs ta e' h' xs'*
shows
sim-move10 P t ta1 (*newA T*⌊*e1*⌉) (*newA T*⌊*e1'*⌉) (*newA T*⌊*e*⌉) *h xs ta* (*newA T*⌊*e'*⌉) *h' xs'*
sim-move10 P t ta1 (*Cast T e1*) (*Cast T e1'*) (*Cast T e*) *h xs ta* (*Cast T e'*) *h' xs'*
sim-move10 P t ta1 (*e1 instanceof T*) (*e1' instanceof T*) (*e instanceof T*) *h xs ta* (*e' instanceof T*) *h' xs'*
sim-move10 P t ta1 (*e1* «*bop*» *e2*) (*e1'* «*bop*» *e2*) (*e* «*bop*» *e2'*) *h xs ta* (*e'* «*bop*» *e2'*) *h' xs'*
sim-move10 P t ta1 (*Val v* «*bop*» *e1*) (*Val v* «*bop*» *e1'*) (*Val v* «*bop*» *e*) *h xs ta* (*Val v* «*bop*» *e'*) *h' xs'*
sim-move10 P t ta1 (*V* := *e1*) (*V* := *e1'*) (*V'* := *e*) *h xs ta* (*V'* := *e'*) *h' xs'*
sim-move10 P t ta1 (*e1*⌊*e2*⌉) (*e1'*⌊*e2*⌉) (*e*⌊*e2'*⌉) *h xs ta* (*e'*⌊*e2'*⌉) *h' xs'*
sim-move10 P t ta1 (*Val v*⌊*e1*⌉) (*Val v*⌊*e1'*⌉) (*Val v*⌊*e*⌉) *h xs ta* (*Val v*⌊*e'*⌉) *h' xs'*
sim-move10 P t ta1 (*e1*⌊*e2*⌉ := *e3*) (*e1'*⌊*e2*⌉ := *e3*) (*e*⌊*e2'*⌉ := *e3'*) *h xs ta* (*e'*⌊*e2'*⌉ := *e3'*) *h' xs'*
sim-move10 P t ta1 (*Val v*⌊*e1*⌉ := *e3*) (*Val v*⌊*e1'*⌉ := *e3*) (*Val v*⌊*e*⌉ := *e3'*) *h xs ta* (*Val v*⌊*e'*⌉ := *e3'*) *h' xs'*
sim-move10 P t ta1 (*AAss* (*Val v*) (*Val v'*) *e1*) (*AAss* (*Val v*) (*Val v'*) *e1'*) (*AAss* (*Val v*) (*Val v'*) *e*) *h xs ta* (*AAss* (*Val v*) (*Val v'*) *e'*) *h' xs'*
sim-move10 P t ta1 (*e1*•*length*) (*e1'*•*length*) (*e*•*length*) *h xs ta* (*e'*•*length*) *h' xs'*
sim-move10 P t ta1 (*e1*•*F*{*D*}) (*e1'*•*F*{*D*}) (*e*•*F'*{*D'*}) *h xs ta* (*e'*•*F'*{*D'*}) *h' xs'*
sim-move10 P t ta1 (*FAss e1 F D e2*) (*FAss e1' F D e2*) (*FAss e F' D' e2'*) *h xs ta* (*FAss e' F' D' e2'*) *h' xs'*
sim-move10 P t ta1 (*FAss* (*Val v*) *F D e1*) (*FAss* (*Val v*) *F D e1'*) (*FAss* (*Val v*) *F' D' e*) *h xs ta* (*FAss* (*Val v*) *F' D' e'*) *h' xs'*
sim-move10 P t ta1 (*CompareAndSwap e1 F D e2 e3*) (*CompareAndSwap e1' F D e2 e3*) (*CompareAndSwap e F' D' e2' e3'*) *h xs ta* (*CompareAndSwap e' F' D' e2' e3'*) *h' xs'*
sim-move10 P t ta1 (*CompareAndSwap* (*Val v*) *F D e1 e3*) (*CompareAndSwap* (*Val v*) *F D e1' e3*) (*CompareAndSwap* (*Val v*) *F' D' e e3'*) *h xs ta* (*CompareAndSwap* (*Val v*) *F' D' e' e3'*) *h' xs'*
sim-move10 P t ta1 (*CompareAndSwap* (*Val v*) *F D* (*Val v'*) *e1*) (*CompareAndSwap* (*Val v*) *F D* (*Val v'*) *e1'*) (*CompareAndSwap* (*Val v*) *F' D'* (*Val v'*) *e*) *h xs ta* (*CompareAndSwap* (*Val v*) *F' D'* (*Val v'*) *e'*) *h' xs'*
sim-move10 P t ta1 (*e1*•*M*(*es*)) (*e1'*•*M*(*es*)) (*e*•*M*(*es'*)) *h xs ta* (*e'*•*M*(*es'*)) *h' xs'*
sim-move10 P t ta1 (*sync*$_V$(*e1*) *e2*) (*sync*$_V$(*e1'*) *e2*) (*sync*(*e*) *e2'*) *h xs ta* (*sync*(*e'*) *e2'*) *h' xs'*
sim-move10 P t ta1 (*insync*$_V$(*a*) *e1*) (*insync*$_V$(*a*) *e1'*) (*insync*(*a'*) *e*) *h xs ta* (*insync*(*a'*) *e'*) *h' xs'*
sim-move10 P t ta1 (*e1*;;*e2*) (*e1'*;;*e2*) (*e*;;*e2'*) *h xs ta* (*e'*;;*e2'*) *h' xs'*
sim-move10 P t ta1 (*if* (*e1*) *e2 else e3*) (*if* (*e1'*) *e2 else e3*) (*if* (*e*) *e2' else e3'*) *h xs ta* (*if* (*e'*) *e2' else e3'*) *h' xs'*
sim-move10 P t ta1 (*throw e1*) (*throw e1'*) (*throw e*) *h xs ta* (*throw e'*) *h' xs'*
sim-move10 P t ta1 (*try e1 catch*(*C V*) *e2*) (*try e1' catch*(*C V*) *e2*) (*try e catch*(*C' V'*) *e2'*) *h xs ta* (*try e' catch*(*C' V'*) *e2'*) *h' xs'*
using *assms*
apply(*simp-all add*: *sim-move10-def final-iff split del*: *if-split split*: *if-split-asm*)
apply(*fastforce simp*: τ*red0t-Val* τ*red0r-Val intro*: *red-reds.intros split*!: *if-splits*)+
done

lemma *sim-moves10-expr*:

sim-move10 P t ta1 e1 e1' e h xs ta e' h' xs' ⟹ *sim-moves10 P t ta1 (e1 # es2) (e1' # es2) (e # es2') h xs ta (e' # es2') h' xs'*
sim-moves10 P t ta1 es1 es1' es h xs ta es' h' xs' ⟹ *sim-moves10 P t ta1 (Val v # es1) (Val v # es1') (Val v # es) h xs ta (Val v # es') h' xs'*
unfolding *sim-moves10-def sim-move10-def final-iff finals-iff*
apply(*simp-all add*: *Cons-eq-append-conv split del*: *if-split split*: *if-split-asm*)
apply(*safe intro*!: *if-split*)
apply(*fastforce simp add*: *is-vals-conv* τ*reds0t-map-Val* τ*reds0r-map-Val* τ*red0t-Val* τ*red0r-Val intro*!: τ*red0r-inj-*τ*reds0r* τ*reds0r-cons-*τ*reds0r* τ*red0t-inj-*τ*reds0t* τ*reds0t-cons-*τ*reds0t ListRed1 ListRed2 split*: *if-split-asm*)+
done

lemma *sim-move10-CallParams*:
sim-moves10 P t ta1 es1 es1' es h xs ta es' h' xs'
⟹ *sim-move10 P t ta1 (Val v·M(es1)) (Val v·M(es1')) (Val v·M(es)) h xs ta (Val v·M(es')) h' xs'*
unfolding *sim-move10-def sim-moves10-def*
apply(*simp split*: *if-split-asm split del*: *if-split add*: *is-vals-conv*)
apply(*fastforce simp add*: τ*red0t-Val* τ*red0r-Val* τ*reds0t-map-Val* τ*reds0r-map-Val intro*: *Call-*τ*red0r-param Call-*τ*red0t-param CallParams split*: *if-split-asm split del*: *if-split intro*!: *if-split*)
apply(*rule conjI*)
apply *fastforce*
apply(*rule if-intro*)
apply *fastforce*
apply(*clarsimp split del*: *if-split*)
apply(*rule if-intro*)
apply(*clarsimp split*: *if-split-asm simp add*: *is-vals-conv*)
apply(*rule exI conjI*)+
apply(*erule Call-*τ*red0r-param*)
apply(*fastforce intro*: *CallParams*)
apply(*rule exI conjI*)+
apply(*erule Call-*τ*red0r-param*)
apply(*fastforce intro*!: *CallParams*)
apply(*clarsimp split del*: *if-split split*: *if-split-asm simp add*: *is-vals-conv* τ*reds0r-map-Val*)
apply *fastforce*
apply(*rule conjI*)
apply *fastforce*
apply(*rule if-intro*)
apply *fastforce*
apply(*rule conjI*)
apply *fastforce*
apply(*rule if-intro*)
apply(*clarsimp split*: *if-split-asm*)
apply(*clarsimp split*: *if-split-asm split del*: *if-split simp add*: *is-vals-conv*)
apply(*fastforce intro*: *CallParams*)
done

lemma *sim-move10-Block*:
sim-move10 P t ta1 e1 e1' e h (*xs*(*V'* := *vo*)) *ta e' h' xs'*
⟹ *sim-move10 P t ta1* ({*V*:*T*=*None*; *e1*}) ({*V*:*T*=*None*; *e1'*}) ({*V'*:*T*=*vo*; *e*}) *h xs ta* ({*V'*:*T*=*xs' V'*; *e'*}) *h'* (*xs'*(*V'* := *xs V'*))
proof −
assume *sim-move10 P t ta1 e1 e1' e h* (*xs*(*V'* := *vo*)) *ta e' h' xs'*
moreover {
fix *e'' xs''*

assume *extTA2J0 P,P,t* ⊢ ⟨*e''*,(*h*, *xs''*)⟩ −*ta*→ ⟨*e'*,(*h'*, *xs'*)⟩
hence *extTA2J0 P,P,t* ⊢ ⟨*e''*,(*h*, *xs''*(*V'* := *xs V'*, *V'* := *xs''* *V'*))⟩ −*ta*→ ⟨*e'*,(*h'*, *xs'*)⟩ **by** *simp*
from *BlockRed*[*OF this*, *of T*]
have *extTA2J0 P,P,t* ⊢ ⟨{*V'*:*T*=*xs''* *V'*; *e''*},(*h*, *xs''*(*V'* := *xs V'*))⟩ −*ta*→ ⟨{*V'*:*T*=*xs'* *V'*; *e'*},(*h'*, *xs'*(*V'* := *xs V'*))⟩
by *simp* **}**
ultimately show *?thesis*
by(*fastforce simp add*: *sim-move10-def final-iff split*: *if-split-asm*)
qed

lemma *sim-move10-reds*:
⟦ (*h'*, *a*) ∈ *allocate h* (*Class-type C*); *ta1* = {|*NewHeapElem a* (*Class-type C*)|}; *ta* = {|*NewHeapElem a* (*Class-type C*)|} ⟧
⟹ *sim-move10 P t ta1* (*new C*) *e1'* (*new C*) *h xs ta* (*addr a*) *h' xs*
allocate h (*Class-type C*) = {} ⟹ *sim-move10 P t* ε (*new C*) *e1'* (*new C*) *h xs* ε (*THROW OutOfMemory*) *h xs*
⟦ (*h'*, *a*) ∈ *allocate h* (*Array-type T* (*nat* (*sint i*))); *0* <=*s i*;
ta1 = {|*NewHeapElem a* (*Array-type T* (*nat* (*sint i*)))|}; *ta* = {|*NewHeapElem a* (*Array-type T* (*nat* (*sint i*)))|} ⟧
⟹ *sim-move10 P t ta1* (*newA T*⌊*Val* (*Intg i*)⌉) *e1'* (*newA T*⌊*Val* (*Intg i*)⌉) *h xs ta* (*addr a*) *h' xs*
i <*s 0* ⟹ *sim-move10 P t* ε (*newA T*⌊*Val* (*Intg i*)⌉) *e1'* (*newA T*⌊*Val* (*Intg i*)⌉) *h xs* ε (*THROW NegativeArraySize*) *h xs*
⟦ *allocate h* (*Array-type T* (*nat* (*sint i*))) = {}; *0* <=*s i* ⟧
⟹ *sim-move10 P t* ε (*newA T*⌊*Val* (*Intg i*)⌉) *e1'* (*newA T*⌊*Val* (*Intg i*)⌉) *h xs* ε (*THROW OutOfMemory*) *h xs*
⟦ *typeof*$_h$ *v* = ⌊*U*⌋; *P* ⊢ *U* ≤ *T* ⟧
⟹ *sim-move10 P t* ε (*Cast T* (*Val v*)) *e1'* (*Cast T* (*Val v*)) *h xs* ε (*Val v*) *h xs*
⟦ *typeof*$_h$ *v* = ⌊*U*⌋; ¬ *P* ⊢ *U* ≤ *T* ⟧
⟹ *sim-move10 P t* ε (*Cast T* (*Val v*)) *e1'* (*Cast T* (*Val v*)) *h xs* ε (*THROW ClassCast*) *h xs*
⟦ *typeof*$_h$ *v* = ⌊*U*⌋; *b* ⟷ *v* ≠ *Null* ∧ *P* ⊢ *U* ≤ *T* ⟧
⟹ *sim-move10 P t* ε ((*Val v*) *instanceof T*) *e1'* ((*Val v*) *instanceof T*) *h xs* ε (*Val* (*Bool b*)) *h xs*
binop bop v1 v2 = *Some* (*Inl v*) ⟹ *sim-move10 P t* ε ((*Val v1*) «*bop*» (*Val v2*)) *e1'* (*Val v1* «*bop*» *Val v2*) *h xs* ε (*Val v*) *h xs*
binop bop v1 v2 = *Some* (*Inr a*) ⟹ *sim-move10 P t* ε ((*Val v1*) «*bop*» (*Val v2*)) *e1'* (*Val v1* «*bop*» *Val v2*) *h xs* ε (*Throw a*) *h xs*
xs V = ⌊*v*⌋ ⟹ *sim-move10 P t* ε (*Var V'*) *e1'* (*Var V*) *h xs* ε (*Val v*) *h xs*
sim-move10 P t ε (*V'* := *Val v*) *e1'* (*V* := *Val v*) *h xs* ε *unit h* (*xs*(*V* ↦ *v*))
sim-move10 P t ε (*null*⌊*Val v*⌉) *e1'* (*null*⌊*Val v*⌉) *h xs* ε (*THROW NullPointer*) *h xs*
⟦ *typeof-addr h a* = ⌊*Array-type T n*⌋; *i* <*s 0* ∨ *sint i* ≥ *int n* ⟧
⟹ *sim-move10 P t* ε (*addr a*⌊*Val* (*Intg i*)⌉) *e1'* ((*addr a*)⌊*Val* (*Intg i*)⌉) *h xs* ε (*THROW ArrayIndexOutOfBounds*) *h xs*
⟦ *typeof-addr h a* = ⌊*Array-type T n*⌋; *0* <=*s i*; *sint i* < *int n*;
heap-read h a (*ACell* (*nat* (*sint i*))) *v*;
ta1 = {|*ReadMem a* (*ACell* (*nat* (*sint i*))) *v*|}; *ta* = {|*ReadMem a* (*ACell* (*nat* (*sint i*))) *v*|} ⟧
⟹ *sim-move10 P t ta1* (*addr a*⌊*Val* (*Intg i*)⌉) *e1'* ((*addr a*)⌊*Val* (*Intg i*)⌉) *h xs ta* (*Val v*) *h xs*
sim-move10 P t ε (*null*⌊*Val v*⌉ := *Val v'*) *e1'* (*null*⌊*Val v*⌉ := *Val v'*) *h xs* ε (*THROW NullPointer*) *h xs*
⟦ *typeof-addr h a* = ⌊*Array-type T n*⌋; *i* <*s 0* ∨ *sint i* ≥ *int n* ⟧
⟹ *sim-move10 P t* ε (*AAss* (*addr a*) (*Val* (*Intg i*)) (*Val v*)) *e1'* (*AAss* (*addr a*) (*Val* (*Intg i*)) (*Val v*)) *h xs* ε (*THROW ArrayIndexOutOfBounds*) *h xs*
⟦ *typeof-addr h a* = ⌊*Array-type T n*⌋; *0* <=*s i*; *sint i* < *int n*; *typeof*$_h$ *v* = ⌊*U*⌋; ¬ (*P* ⊢ *U* ≤ *T*) ⟧
⟹ *sim-move10 P t* ε (*AAss* (*addr a*) (*Val* (*Intg i*)) (*Val v*)) *e1'* (*AAss* (*addr a*) (*Val* (*Intg i*)) (*Val v*)) *h xs* ε (*THROW ArrayStore*) *h xs*
⟦ *typeof-addr h a* = ⌊*Array-type T n*⌋; *0* <=*s i*; *sint i* < *int n*; *typeof*$_h$ *v* = *Some U*; *P* ⊢ *U* ≤ *T*;

heap-write h a (ACell (nat (sint i))) v h′;
ta1 = {|WriteMem a (ACell (nat (sint i))) v|}; ta = {|WriteMem a (ACell (nat (sint i))) v|} ⟧
⟹ sim-move10 P t ta1 (AAss (addr a) (Val (Intg i)) (Val v)) e1′ (AAss (addr a) (Val (Intg i)) (Val v)) h xs ta unit h′ xs
typeof-addr h a = ⌊Array-type T n⌋ ⟹ sim-move10 P t ε (addr a·length) e1′ (addr a·length) h xs ε (Val (Intg (word-of-nat n))) h xs
sim-move10 P t ε (null·length) e1′ (null·length) h xs ε (THROW NullPointer) h xs
⟦ heap-read h a (CField D F) v; ta1 = {|ReadMem a (CField D F) v|}; ta = {|ReadMem a (CField D F) v|} ⟧
⟹ sim-move10 P t ta1 (addr a·F{D}) e1′ (addr a·F{D}) h xs ta (Val v) h xs
sim-move10 P t ε (null·F{D}) e1′ (null·F{D}) h xs ε (THROW NullPointer) h xs
⟦ heap-write h a (CField D F) v h′; ta1 = {|WriteMem a (CField D F) v|}; ta = {|WriteMem a (CField D F) v|} ⟧
⟹ sim-move10 P t ta1 (addr a·F{D} := Val v) e1′ (addr a·F{D} := Val v) h xs ta unit h′ xs
sim-move10 P t ε (null·compareAndSwap(D·F, Val v, Val v′)) e1′ (null·compareAndSwap(D·F, Val v, Val v′)) h xs ε (THROW NullPointer) h xs
⟦ heap-read h a (CField D F) v″; heap-write h a (CField D F) v′ h′; v″ = v;
ta1 = {| ReadMem a (CField D F) v″, WriteMem a (CField D F) v′ |}; ta = {| ReadMem a (CField D F) v″, WriteMem a (CField D F) v′ |} ⟧
⟹ sim-move10 P t ta1 (addr a·compareAndSwap(D·F, Val v, Val v′)) e1′ (addr a·compareAndSwap(D·F, Val v, Val v′)) h xs ta true h′ xs
⟦ heap-read h a (CField D F) v″; v″ ≠ v;
ta1 = {| ReadMem a (CField D F) v″ |}; ta = {| ReadMem a (CField D F) v″ |} ⟧
⟹ sim-move10 P t ta1 (addr a·compareAndSwap(D·F, Val v, Val v′)) e1′ (addr a·compareAndSwap(D·F, Val v, Val v′)) h xs ta false h xs

sim-move10 P t ε (null·F{D} := Val v) e1′ (null·F{D} := Val v) h xs ε (THROW NullPointer) h xs
sim-move10 P t ε ({V′:T=None; Val u}) e1′ ({V:T=vo; Val u}) h xs ε (Val u) h xs
sim-move10 P t ε ({V′:T=⌊v⌋; e}) ({V′:T=None; e}) ({V:T=vo; e′}) h xs ε ({V:T=vo; e′}) h xs

sim-move10 P t ε (sync$_{V'}$(null) e0) e1′ (sync(null) e1) h xs ε (THROW NullPointer) h xs
sim-move10 P t ε (Val v;;e0) e1′ (Val v;; e1) h xs ε e1 h xs
sim-move10 P t ε (if (true) e0 else e0′) e1′ (if (true) e1 else e2) h xs ε e1 h xs
sim-move10 P t ε (if (false) e0 else e0′) e1′ (if (false) e1 else e2) h xs ε e2 h xs
sim-move10 P t ε (throw null) e1′ (throw null) h xs ε (THROW NullPointer) h xs
sim-move10 P t ε (try (Val v) catch(C V′) e0) e1′ (try (Val v) catch(C V) e1) h xs ε (Val v) h xs
⟦ typeof-addr h a = ⌊Class-type D⌋; P ⊢ D ≼ C ⟧*
⟹ sim-move10 P t ε (try (Throw a) catch(C V′) e0) e1′ (try (Throw a) catch(C V) e1) h xs ε ({V:Class C=⌊Addr a⌋; e1}) h xs
sim-move10 P t ε (newA T⌊Throw a⌉) e1′ (newA T⌊Throw a⌉) h xs ε (Throw a) h xs
sim-move10 P t ε (Cast T (Throw a)) e1′ (Cast T (Throw a)) h xs ε (Throw a) h xs
sim-move10 P t ε ((Throw a) instanceof T) e1′ ((Throw a) instanceof T) h xs ε (Throw a) h xs
sim-move10 P t ε ((Throw a) «bop» e0) e1′ ((Throw a) «bop» e1) h xs ε (Throw a) h xs
sim-move10 P t ε (Val v «bop» (Throw a)) e1′ (Val v «bop» (Throw a)) h xs ε (Throw a) h xs
sim-move10 P t ε (V′ := Throw a) e1′ (V := Throw a) h xs ε (Throw a) h xs
sim-move10 P t ε (Throw a⌊e0⌉) e1′ (Throw a⌊e1⌉) h xs ε (Throw a) h xs
sim-move10 P t ε (Val v⌊Throw a⌉) e1′ (Val v⌊Throw a⌉) h xs ε (Throw a) h xs
sim-move10 P t ε (Throw a⌊e0⌉ := e0′) e1′ (Throw a⌊e1⌉ := e2) h xs ε (Throw a) h xs
sim-move10 P t ε (Val v⌊Throw a⌉ := e0) e1′ (Val v⌊Throw a⌉ := e1) h xs ε (Throw a) h xs
sim-move10 P t ε (Val v⌊Val v′⌉ := Throw a) e1′ (Val v⌊Val v′⌉ := Throw a) h xs ε (Throw a) h xs
sim-move10 P t ε (Throw a·length) e1′ (Throw a·length) h xs ε (Throw a) h xs
sim-move10 P t ε (Throw a·F{D}) e1′ (Throw a·F{D}) h xs ε (Throw a) h xs
sim-move10 P t ε (Throw a·F{D} := e0) e1′ (Throw a·F{D} := e1) h xs ε (Throw a) h xs

sim-move10 P t ε (Val v·F{D} := Throw a) e1′ (Val v·F{D} := Throw a) h xs ε (Throw a) h xs
sim-move10 P t ε (CompareAndSwap (Throw a) D F e0 e0′) e1′ (Throw a·compareAndSwap(D·F, e1′′, e1′′′)) h xs ε (Throw a) h xs
sim-move10 P t ε (CompareAndSwap (Val v) D F (Throw a) e0′) e1′ (Val v·compareAndSwap(D·F, Throw a, e1′′)) h xs ε (Throw a) h xs
sim-move10 P t ε (CompareAndSwap (Val v) D F (Val v′) (Throw a)) e1′ (Val v·compareAndSwap(D·F, Val v′, Throw a)) h xs ε (Throw a) h xs
sim-move10 P t ε (Throw a·M(es0)) e1′ (Throw a·M(es1)) h xs ε (Throw a) h xs
sim-move10 P t ε (Val v·M(map Val vs @ Throw a # es0)) e1′ (Val v·M(map Val vs @ Throw a # es1)) h xs ε (Throw a) h xs
sim-move10 P t ε ({V′:T=None; Throw a}) e1′ ({V:T=vo; Throw a}) h xs ε (Throw a) h xs
sim-move10 P t ε (sync$_{V'}$(Throw a) e0) e1′ (sync(Throw a) e1) h xs ε (Throw a) h xs
sim-move10 P t ε (Throw a;;e0) e1′ (Throw a;;e1) h xs ε (Throw a) h xs
sim-move10 P t ε (if (Throw a) e0 else e0′) e1′ (if (Throw a) e1 else e2) h xs ε (Throw a) h xs
sim-move10 P t ε (throw (Throw a)) e1′ (throw (Throw a)) h xs ε (Throw a) h xs
apply(*fastforce simp add*: *sim-move10-def no-calls-def no-call-def ta-bisim-def intro*: *red-reds.intros*)+
done

lemma *sim-move10-CallNull*:
sim-move10 P t ε (null·M(map Val vs)) e1′ (null·M(map Val vs)) h xs ε (THROW NullPointer) h xs
by(*fastforce simp add*: *sim-move10-def map-eq-append-conv intro*: *RedCallNull*)

lemma *sim-move10-SyncLocks*:
⟦ ta1 = {⌈Lock→a, SyncLock a⌋}; ta = {⌈Lock→a, SyncLock a⌋} ⟧
⟹ sim-move10 P t ta1 (sync$_V$(addr a) e0) e1′ (sync(addr a) e1) h xs ta (insync(a) e1) h xs
⟦ ta1 = {⌈Unlock→a, SyncUnlock a⌋}; ta = {⌈Unlock→a, SyncUnlock a⌋} ⟧
⟹ sim-move10 P t ta1 (insync$_V$(a′) (Val v)) e1′ (insync(a) (Val v)) h xs ta (Val v) h xs
⟦ ta1 = {⌈Unlock→a, SyncUnlock a⌋}; ta = {⌈Unlock→a, SyncUnlock a⌋} ⟧
⟹ sim-move10 P t ta1 (insync$_V$(a′) (Throw a′′)) e1′ (insync(a) (Throw a′′)) h xs ta (Throw a′′) h xs
by(*fastforce simp add*: *sim-move10-def ta-bisim-def ta-upd-simps intro*: *red-reds.intros*[*simplified*])+

lemma *sim-move10-TryFail*:
⟦ typeof-addr h a = ⌊Class-type D⌋; ¬ P ⊢ D ⪯ C ⟧*
⟹ sim-move10 P t ε (try (Throw a) catch(C V′) e0) e1′ (try (Throw a) catch(C V) e1) h xs ε (Throw a) h xs
by(*auto simp add*: *sim-move10-def intro*!: *RedTryFail*)

lemmas *sim-move10-intros* =
sim-move10-expr sim-move10-reds sim-move10-CallNull sim-move10-TryFail sim-move10-Block sim-move10-Call

lemma *sim-move10-preserves-defass*:
assumes *wf*: *wf-J-prog P*
shows *⟦ sim-move10 P t ta1 e1 e1′ e h xs ta e′ h′ xs′; 𝒟 e ⌊dom xs⌋ ⟧ ⟹ 𝒟 e′ ⌊dom xs′⌋*
by(*auto simp add*: *sim-move10-def split*: *if-split-asm dest*!: *τred0r-preserves-defass*[*OF wf*] *τred0t-preserves-defass*[*OF wf*] *red-preserves-defass*[*OF wf*])

declare *sim-move10-intros*[*intro*]

lemma assumes *wf*: *wf-J-prog P*
shows *red-simulates-red1-aux*:
⟦ False,compP1 P,t ⊢1 ⟨e1, S⟩ −TA→ ⟨e1′, S′⟩; bisim vs e2 e1 (lcl S); fv e2 ⊆ set vs;
x ⊆$_m$ [vs [↦] lcl S]; length vs + max-vars e1 ≤ length (lcl S);

$\mathcal{D}$ *e2* ⌊*dom x*⌋ ⟧
⟹ ∃ *ta e2′ x′*. *sim-move10 P t TA e1 e1′ e2* (*hp S*) *x ta e2′* (*hp S′*) *x′* ∧ *bisim vs e2′ e1′* (*lcl S′*) ∧ *x′* ⊆$_m$ [*vs* [↦] *lcl S′*]
(**is** ⟦ -; -; -; -; -; - ⟧ ⟹ *?concl e1 e1′ e2 S x TA S′ e1′ vs*)

and *reds-simulates-reds1-aux*:
⟦ *False,compP1 P,t* ⊢1 ⟨*es1*, *S*⟩ [−*TA*→] ⟨*es1′*, *S′*⟩; *bisims vs es2 es1* (*lcl S*); *fvs es2* ⊆ *set vs*;
x ⊆$_m$ [*vs* [↦] *lcl S*]; *length vs* + *max-varss es1* ≤ *length* (*lcl S*);
$\mathcal{D}$*s es2* ⌊*dom x*⌋ ⟧
⟹ ∃ *ta es2′ x′*. *sim-moves10 P t TA es1 es1′ es2* (*hp S*) *x ta es2′* (*hp S′*) *x′* ∧ *bisims vs es2′ es1′* (*lcl S′*) ∧ *x′* ⊆$_m$ [*vs* [↦] *lcl S′*]
(**is** ⟦ -; -; -; -; -; - ⟧ ⟹ *?concls es1 es1′ es2 S x TA S′ es1′ vs*)
proof(*induct arbitrary*: *vs e2 x* **and** *vs es2 x rule*: *red1-reds1.inducts*)
case (*Bin1OpRed1 e s ta e′ s′ bop e2 Vs E2 X*)
note *IH* = ‹⋀*vs e2 x*. ⟦ *bisim vs e2 e* (*lcl s*); *fv e2* ⊆ *set vs*;
x ⊆$_m$ [*vs* [↦] *lcl s*]; *length vs* + *max-vars e* ≤ *length* (*lcl s*); $\mathcal{D}$ *e2* ⌊*dom x*⌋⟧
⟹ *?concl e e′ e2 s x ta s′ e′ vs*›
from ‹*False,compP1 P,t* ⊢1 ⟨*e,s*⟩ −*ta*→ ⟨*e′,s′*⟩› **have** ¬ *is-val e* **by** *auto*
with ‹*bisim Vs E2* (*e* «*bop*» *e2*) (*lcl s*)› **obtain** *E E2′*
where *E2*: *E2* = *E* «*bop*» *E2′ e2* = *compE1 Vs E2′* **and** *bisim Vs E e* (*lcl s*)
and *sync*: ¬ *contains-insync E2′*
by(*auto elim*!: *bisim-cases*)
moreover note *IH*[*of Vs E X*] ‹*fv E2* ⊆ *set Vs*› ‹*X* ⊆$_m$ ⌈*Vs* [↦] *lcl s*⌉›
‹*length Vs* + *max-vars* (*e* «*bop*» *e2*) ≤ *length* (*lcl s*)› ‹$\mathcal{D}$ *E2* ⌊*dom X*⌋›
ultimately obtain *TA′ e2′ x′* **where** *sim-move10 P t ta e e′ E* (*hp s*) *X TA′ e2′* (*hp s′*) *x′*
bisim Vs e2′ e′ (*lcl s′*) *x′* ⊆$_m$ ⌈*Vs* [↦] *lcl s′*⌉ **by**(*auto*)
with *E2* ‹*fv E2* ⊆ *set Vs*› *sync* **show** *?case* **by**(*cases is-val e2′*)(*auto intro*: *BinOpRed1*)
next
case (*Bin1OpRed2 e s ta e′ s′ v bop Vs E2 X*)
note *IH* = ‹⋀*vs e2 x*. ⟦ *bisim vs e2 e* (*lcl s*); *fv e2* ⊆ *set vs*;
x ⊆$_m$ [*vs* [↦] *lcl s*]; *length vs* + *max-vars e* ≤ *length* (*lcl s*); $\mathcal{D}$ *e2* ⌊*dom x*⌋⟧
⟹ *?concl e e′ e2 s x ta s′ e′ vs*›
from ‹*bisim Vs E2* (*Val v* «*bop*» *e*) (*lcl s*)› **obtain** *E*
where *E2*: *E2* = *Val v* «*bop*» *E* **and** *bisim Vs E e* (*lcl s*) **by**(*auto*)
moreover note *IH*[*of Vs E X*] ‹*fv E2* ⊆ *set Vs*› ‹*X* ⊆$_m$ ⌈*Vs* [↦] *lcl s*⌉›
‹*length Vs* + *max-vars* (*Val v* «*bop*» *e*) ≤ *length* (*lcl s*)› ‹$\mathcal{D}$ *E2* ⌊*dom X*⌋› *E2*
ultimately show *?case* **by**(*auto intro*: *BinOpRed2*)
next
case (*Red1Var s V v Vs E2 X*)
from ‹*bisim Vs E2* (*Var V*) (*lcl s*)› ‹*fv E2* ⊆ *set Vs*›
obtain *V′* **where** *E2* = *Var V′ V′* = *Vs* ! *V V* = *index Vs V′* **by**(*clarify*, *simp*)
from ‹*E2* = *Var V′*› ‹$\mathcal{D}$ *E2* ⌊*dom X*⌋›
obtain *v′* **where** *X V′* = ⌊*v′*⌋ **by**(*auto simp add*: *hyperset-defs*)
with ‹*X* ⊆$_m$ ⌈*Vs* [↦] *lcl s*⌉› **have** ⌈*Vs* [↦] *lcl s*⌉ *V′* = ⌊*v′*⌋ **by**(*rule map-le-SomeD*)
with ‹*length Vs* + *max-vars* (*Var V*) ≤ *length* (*lcl s*)›
have *lcl s* ! (*index Vs V′*) = *v′* **by**(*auto intro*: *map-upds-Some-eq-nth-index*)
with ‹*V* = *index Vs V′*› ‹*lcl s* ! *V* = *v*› **have** *v* = *v′* **by** *simp*
with ‹*X V′* = ⌊*v′*⌋› ‹*E2* = *Var V′*› ‹*X* ⊆$_m$ ⌈*Vs* [↦] *lcl s*⌉›
show *?case* **by**(*fastforce intro*: *RedVar*)
next
case (*LAss1Red e s ta e′ s′ V Vs E2 X*)
note *IH* = ‹⋀*vs e2 x*. ⟦ *bisim vs e2 e* (*lcl s*); *fv e2* ⊆ *set vs*;
x ⊆$_m$ [*vs* [↦] *lcl s*]; *length vs* + *max-vars e* ≤ *length* (*lcl s*); $\mathcal{D}$ *e2* ⌊*dom x*⌋⟧
⟹ *?concl e e′ e2 s x ta s′ e′ vs*›

from ‹*bisim Vs E2* (*V*:=*e*) (*lcl s*)› **obtain** *E V′*
where *E2*: *E2* = (*V′*:=*E*) *V* = *index Vs V′* **and** *bisim Vs E e* (*lcl s*) **by** *auto*
with *IH*[*of Vs E X*] ‹*fv E2* ⊆ *set Vs*› ‹*X* ⊆*m* [*Vs* [↦] *lcl s*]›
‹*length Vs* + *max-vars* (*V*:=*e*) ≤ *length* (*lcl s*)› ‹*D E2* ⌊*dom X*⌋›
E2 **show** *?case* **by**(*auto intro*: *LAssRed*)
next
case (*Red1LAss V l v h Vs E2 X*)
from ‹*bisim Vs E2* (*V*:=*Val v*) (*lcl* (*h*, *l*))› **obtain** *V′* **where** *E2* = (*V′* := *Val v*) *V* = *index Vs V′* **by**(*auto*)
moreover with ‹*fv E2* ⊆ *set Vs*› ‹*X* ⊆*m* [*Vs* [↦] *lcl* (*h*, *l*)]› ‹*length Vs* + *max-vars* (*V*:=*Val v*) ≤ *length* (*lcl* (*h*, *l*))›
have *X*(*V′* ↦ *v*) ⊆*m* [*Vs* [↦] *l*[*index Vs V′* := *v*]] **by**(*auto intro*: *LAss-lem*)
ultimately show *?case* **by**(*auto intro*: *RedLAss simp del*: *fun-upd-apply*)
next
case (*AAcc1Red1 a s ta a′ s′ i Vs E2 X*)
note *IH* = ‹⋀*vs e2 x*. ⟦ *bisim vs e2 a* (*lcl s*); *fv e2* ⊆ *set vs*;
x ⊆*m* [*vs* [↦] *lcl s*]; *length vs* + *max-vars a* ≤ *length* (*lcl s*); *D e2* ⌊*dom x*⌋⟧
⟹ *?concl a a′ e2 s x ta s′ a′ vs*›
from ‹*False*,*compP1 P*,*t* ⊢*1* ⟨*a*,*s*⟩ −*ta*→ ⟨*a′*,*s′*⟩› **have** ¬ *is-val a* **by** *auto*
with ‹*bisim Vs E2* (*a*⌊*i*⌉) (*lcl s*)› **obtain** *E E2′*
where *E2*: *E2* = *E*⌊*E2′*⌉ *i* = *compE1 Vs E2′* **and** *bisim Vs E a* (*lcl s*)
and *sync*: ¬ *contains-insync E2′* **by**(*fastforce*)
moreover note *IH*[*of Vs E X*] ‹*fv E2* ⊆ *set Vs*› ‹*X* ⊆*m* [*Vs* [↦] *lcl s*]›
‹*length Vs* + *max-vars* (*a*⌊*i*⌉) ≤ *length* (*lcl s*)› ‹*D E2* ⌊*dom X*⌋›
ultimately obtain *TA′ e2′ x′* **where** *sim-move10 P t ta a a′ E* (*hp s*) *X TA′ e2′* (*hp s′*) *x′*
bisim Vs e2′ a′ (*lcl s′*) *x′* ⊆*m* [*Vs* [↦] *lcl s′*] **by**(*auto*)
with *E2* ‹*fv E2* ⊆ *set Vs*› *sync* **show** *?case*
by(*cases is-val e2′*)(*auto intro*: *AAccRed1*)
next
case (*AAcc1Red2 i s ta i′ s′ a Vs E2 X*)
note *IH* = ‹⋀*vs e2 x*. ⟦ *bisim vs e2 i* (*lcl s*); *fv e2* ⊆ *set vs*;
x ⊆*m* [*vs* [↦] *lcl s*]; *length vs* + *max-vars i* ≤ *length* (*lcl s*); *D e2* ⌊*dom x*⌋⟧
⟹ *?concl i i′ e2 s x ta s′ i′ vs*›
from ‹*bisim Vs E2* (*Val a*⌊*i*⌉) (*lcl s*)› **obtain** *E*
where *E2*: *E2* = *Val a*⌊*E*⌉ **and** *bisim Vs E i* (*lcl s*) **by**(*auto*)
moreover note *IH*[*of Vs E X*] ‹*fv E2* ⊆ *set Vs*› ‹*X* ⊆*m* [*Vs* [↦] *lcl s*]› *E2*
‹*length Vs* + *max-vars* (*Val a*⌊*i*⌉) ≤ *length* (*lcl s*)› ‹*D E2* ⌊*dom X*⌋›
ultimately show *?case* **by**(*auto intro*: *AAccRed2*)
next
case *Red1AAcc* **thus** *?case* **by**(*fastforce intro*: *RedAAcc simp del*: *fun-upd-apply*)
next
case (*AAss1Red1 a s ta a′ s′ i e Vs E2 X*)
note *IH* = ‹⋀*vs e2 x*. ⟦ *bisim vs e2 a* (*lcl s*); *fv e2* ⊆ *set vs*;
x ⊆*m* [*vs* [↦] *lcl s*]; *length vs* + *max-vars a* ≤ *length* (*lcl s*); *D e2* ⌊*dom x*⌋ ⟧
⟹ *?concl a a′ e2 s x ta s′ a′ vs*›
from ‹*False*,*compP1 P*,*t* ⊢*1* ⟨*a*,*s*⟩ −*ta*→ ⟨*a′*,*s′*⟩› **have** ¬ *is-val a* **by** *auto*
with ‹*bisim Vs E2* (*a*⌊*i*⌉:=*e*) (*lcl s*)› **obtain** *E E2′ E2″*
where *E2*: *E2* = *E*⌊*E2′*⌉:=*E2″ i* = *compE1 Vs E2′ e* = *compE1 Vs E2″* **and** *bisim Vs E a* (*lcl s*)
and *sync*: ¬ *contains-insync E2′* ¬ *contains-insync E2″* **by**(*fastforce*)
moreover note *IH*[*of Vs E X*] ‹*fv E2* ⊆ *set Vs*› ‹*X* ⊆*m* [*Vs* [↦] *lcl s*]›
‹*length Vs* + *max-vars* (*a*⌊*i*⌉:=*e*) ≤ *length* (*lcl s*)› ‹*D E2* ⌊*dom X*⌋›
ultimately obtain *TA′ e2′ x′* **where** *IH′*: *sim-move10 P t ta a a′ E* (*hp s*) *X TA′ e2′* (*hp s′*) *x′*
bisim Vs e2′ a′ (*lcl s′*) *x′* ⊆*m* [*Vs* [↦] *lcl s′*] **by**(*auto*)
show *?case*

```
  proof(cases is-val e2')
    case True
    from E2 ‹fv E2 ⊆ set Vs› sync have bisim Vs E2' i (lcl s') bisim Vs E2'' e (lcl s') by auto
    with IH' E2 True sync show ?thesis
      by(cases is-val E2')(fastforce intro: AAssRed1)+
  next
    case False with IH' E2 sync show ?thesis by(fastforce intro: AAssRed1)
  qed
next
  case (AAss1Red2 i s ta i' s' a e Vs E2 X)
  note IH = ‹⋀vs e2 x. ⟦ bisim vs e2 i (lcl s); fv e2 ⊆ set vs;
            x ⊆m [vs [↦] lcl s]; length vs + max-vars i ≤ length (lcl s); 𝒟 e2 ⌊dom x⌋⟧
           ⟹ ?concl i i' e2 s x ta s' i' vs›
  from ‹False,compP1 P,t ⊢1 ⟨i,s⟩ −ta→ ⟨i',s'⟩› have ¬ is-val i by auto
  with ‹bisim Vs E2 (Val a⌊i⌉:=e) (lcl s)› obtain E E2'
    where E2: E2 = Val a⌊E⌉:=E2' e = compE1 Vs E2' and bisim Vs E i (lcl s)
    and sync: ¬ contains-insync E2' by(fastforce)
  moreover note IH[of Vs E X] ‹fv E2 ⊆ set Vs› ‹X ⊆m [Vs [↦] lcl s]›
    ‹length Vs + max-vars (Val a⌊i⌉:=e) ≤ length (lcl s)› ‹𝒟 E2 ⌊dom X⌋›
  ultimately obtain TA' e2' x' where sim-move10 P t ta i i' E (hp s) X TA' e2' (hp s') x'
    bisim Vs e2' i' (lcl s') x' ⊆m [Vs [↦] lcl s'] by(auto)
  with E2 ‹fv E2 ⊆ set Vs› sync show ?case
    by(cases is-val e2')(fastforce intro: AAssRed2)+
next
  case (AAss1Red3 e s ta e' s' a i Vs E2 X)
  note IH = ‹⋀vs e2 x. ⟦ bisim vs e2 e (lcl s); fv e2 ⊆ set vs;
            x ⊆m [vs [↦] lcl s]; length vs + max-vars e ≤ length (lcl s); 𝒟 e2 ⌊dom x⌋⟧
           ⟹ ?concl e e' e2 s x ta s' e' vs›
  from ‹bisim Vs E2 (Val a⌊Val i⌉:=e) (lcl s)› obtain E
    where E2: E2 = Val a⌊Val i⌉:=E and bisim Vs E e (lcl s) by(fastforce)
  moreover note IH[of Vs E X] ‹fv E2 ⊆ set Vs› ‹X ⊆m [Vs [↦] lcl s]› E2
    ‹length Vs + max-vars (Val a⌊Val i⌉:=e) ≤ length (lcl s)› ‹𝒟 E2 ⌊dom X⌋›
  ultimately show ?case by(fastforce intro: AAssRed3)
next
  case Red1AAssStore thus ?case by(auto)((rule exI conjI)+, auto)
next
  case Red1AAss thus ?case
    by(fastforce simp del: fun-upd-apply)
next
  case (FAss1Red1 e s ta e' s' F D e2 Vs E2 X)
  note IH = ‹⋀vs e2 x. ⟦ bisim vs e2 e (lcl s); fv e2 ⊆ set vs;
            x ⊆m [vs [↦] lcl s]; length vs + max-vars e ≤ length (lcl s); 𝒟 e2 ⌊dom x⌋⟧
           ⟹ ?concl e e' e2 s x ta s' e' vs›
  from ‹False,compP1 P,t ⊢1 ⟨e,s⟩ −ta→ ⟨e',s'⟩› have ¬ is-val e by auto
  with ‹bisim Vs E2 (e·F{D}:=e2) (lcl s)› obtain E E2'
    where E2: E2 = E·F{D}:=E2' e2 = compE1 Vs E2' and bisim Vs E e (lcl s)
    and sync: ¬ contains-insync E2' by(fastforce)
  with IH[of Vs E X] ‹fv E2 ⊆ set Vs› ‹X ⊆m [Vs [↦] lcl s]›
    ‹length Vs + max-vars (e·F{D}:=e2) ≤ length (lcl s)› ‹𝒟 E2 ⌊dom X⌋›
  obtain TA' e2' x' where sim-move10 P t ta e e' E (hp s) X TA' e2' (hp s') x'
    bisim Vs e2' e' (lcl s') x' ⊆m [Vs [↦] lcl s'] by(fastforce)
  with E2 ‹fv E2 ⊆ set Vs› sync show ?case by(cases is-val e2')(auto intro: FAssRed1)
next
  case (FAss1Red2 e s ta e' s' v F D Vs E2 X)
```

note *IH* = ‹$\bigwedge$*vs e2 x.* ⟦ *bisim vs e2 e* (*lcl s*); *fv e2* ⊆ *set vs*;
x ⊆$_m$ [*vs* [↦] *lcl s*]; *length vs* + *max-vars e* ≤ *length* (*lcl s*); $\mathcal{D}$ *e2* ⌊*dom x*⌋⟧
⟹ *?concl e e′ e2 s x ta s′ e′ vs*›
from ‹*bisim Vs E2* (*Val v*·*F*{*D*}:=*e*) (*lcl s*)› **obtain** *E*
where *E2*: *E2* = (*Val v*·*F*{*D*}:=*E*) **and** *bisim Vs E e* (*lcl s*) **by**(*fastforce*)
with *IH*[*of Vs E X*] ‹*fv E2* ⊆ *set Vs*› ‹*X* ⊆$_m$ [*Vs* [↦] *lcl s*]›
‹*length Vs* + *max-vars* (*Val v*·*F*{*D*}:=*e*) ≤ *length* (*lcl s*)› ‹$\mathcal{D}$ *E2* ⌊*dom X*⌋›
E2 **show** *?case* **by**(*fastforce intro*: *FAssRed2*)
next
case (*CAS1Red1 e s ta e′ s′ D F e2 e3 Vs E2 X*)
note *IH* = ‹$\bigwedge$*vs e2 x.* ⟦ *bisim vs e2 e* (*lcl s*); *fv e2* ⊆ *set vs*;
x ⊆$_m$ [*vs* [↦] *lcl s*]; *length vs* + *max-vars e* ≤ *length* (*lcl s*); $\mathcal{D}$ *e2* ⌊*dom x*⌋ ⟧
⟹ *?concl e e′ e2 s x ta s′ e′ vs*›
from ‹*False*,*compP1 P*,*t* ⊢*1* ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩› **have** ¬ *is-val e* **by** *auto*
with ‹*bisim Vs E2* - (*lcl s*)› **obtain** *E E2′ E2″*
where *E2*: *E2* = *E*·*compareAndSwap*(*D*·*F*, *E2′*, *E2″*) *e2* = *compE1 Vs E2′ e3* = *compE1 Vs E2″* **and** *bisim Vs E e* (*lcl s*)
and *sync*: ¬ *contains-insync E2′* ¬ *contains-insync E2″* **by**(*fastforce*)
moreover note *IH*[*of Vs E X*] ‹*fv E2* ⊆ *set Vs*› ‹*X* ⊆$_m$ [*Vs* [↦] *lcl s*]›
‹*length Vs* + *max-vars* - ≤ *length* (*lcl s*)› ‹$\mathcal{D}$ *E2* ⌊*dom X*⌋›
ultimately obtain *TA′ e2′ x′* **where** *IH′*: *sim-move10 P t ta e e′ E* (*hp s*) *X TA′ e2′* (*hp s′*) *x′*
bisim Vs e2′ e′ (*lcl s′*) *x′* ⊆$_m$ [*Vs* [↦] *lcl s′*] **by**(*auto*)
show *?case*
proof(*cases is-val e2′*)
case *True*
from *E2* ‹*fv E2* ⊆ *set Vs*› *sync* **have** *bisim Vs E2′ e2* (*lcl s′*) *bisim Vs E2″ e3* (*lcl s′*) **by** *auto*
with *IH′ E2 True sync* **show** *?thesis* **by**(*cases is-val E2′*)(*fastforce*)+
next
case *False* **with** *IH′ E2 sync* **show** *?thesis* **by**(*fastforce*)
qed
next
case (*CAS1Red2 e s ta e′ s′ v D F e3 Vs E2 X*)
note *IH* = ‹$\bigwedge$*vs e2 x.* ⟦ *bisim vs e2 e* (*lcl s*); *fv e2* ⊆ *set vs*;
x ⊆$_m$ [*vs* [↦] *lcl s*]; *length vs* + *max-vars e* ≤ *length* (*lcl s*); $\mathcal{D}$ *e2* ⌊*dom x*⌋⟧
⟹ *?concl e e′ e2 s x ta s′ e′ vs*›
from ‹*False*,*compP1 P*,*t* ⊢*1* ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩› **have** ¬ *is-val e* **by** *auto*
with ‹*bisim Vs E2* - (*lcl s*)› **obtain** *E E2′*
where *E2*: *E2* = (*Val v*·*compareAndSwap*(*D*·*F*, *E*, *E2′*)) *e3* = *compE1 Vs E2′* **and** *bisim Vs E e* (*lcl s*)
and *sync*: ¬ *contains-insync E2′* **by**(*auto*)
moreover note *IH*[*of Vs E X*] ‹*fv E2* ⊆ *set Vs*› ‹*X* ⊆$_m$ [*Vs* [↦] *lcl s*]›
‹*length Vs* + *max-vars* - ≤ *length* (*lcl s*)› ‹$\mathcal{D}$ *E2* ⌊*dom X*⌋›
ultimately obtain *TA′ e2′ x′* **where** *sim-move10 P t ta e e′ E* (*hp s*) *X TA′ e2′* (*hp s′*) *x′*
bisim Vs e2′ e′ (*lcl s′*) *x′* ⊆$_m$ [*Vs* [↦] *lcl s′*] **by**(*auto*)
with *E2* ‹*fv E2* ⊆ *set Vs*› *sync* **show** *?case*
by(*cases is-val e2′*)(*fastforce*)+
next
case (*CAS1Red3 e s ta e′ s′ v D F v′ Vs E2 X*)
note *IH* = ‹$\bigwedge$*vs e2 x.* ⟦ *bisim vs e2 e* (*lcl s*); *fv e2* ⊆ *set vs*;
x ⊆$_m$ [*vs* [↦] *lcl s*]; *length vs* + *max-vars e* ≤ *length* (*lcl s*); $\mathcal{D}$ *e2* ⌊*dom x*⌋⟧
⟹ *?concl e e′ e2 s x ta s′ e′ vs*›
from ‹*bisim Vs E2* - (*lcl s*)› **obtain** *E*
where *E2*: *E2* = (*Val v*·*compareAndSwap*(*D*·*F*, *Val v′*, *E*)) **and** *bisim Vs E e* (*lcl s*) **by**(*fastforce*)
moreover note *IH*[*of Vs E X*] ‹*fv E2* ⊆ *set Vs*› ‹*X* ⊆$_m$ [*Vs* [↦] *lcl s*]› *E2*

```
    ‹length Vs + max-vars - ≤ length (lcl s)› ‹𝒟 E2 ⌊dom X⌋›
  ultimately show ?case by(fastforce)
next
  case (Call1Obj e s ta e' s' M es Vs E2 X)
  note IH = ‹⋀vs e2 x. ⟦ bisim vs e2 e (lcl s); fv e2 ⊆ set vs;
          x ⊆m [vs [↦] lcl s]; length vs + max-vars e ≤ length (lcl s);
          𝒟 e2 ⌊dom x⌋ ⟧ ⟹ ?concl e e' e2 s x ta s' e' vs›
  from ‹False,compP1 P,t ⊢1 ⟨e,s⟩ −ta→ ⟨e',s'⟩› ‹bisim Vs E2 (e·M(es)) (lcl s)›
  obtain E es' where E2: E2 = E·M(es') es = compEs1 Vs es' and bisim Vs E e (lcl s)
    and sync: ¬ contains-insyncs es' by(auto elim!: bisim-cases simp add: compEs1-conv-map)
  with IH[of Vs E X] ‹fv E2 ⊆ set Vs› ‹X ⊆m [Vs [↦] lcl s]›
    ‹length Vs + max-vars (e·M(es)) ≤ length (lcl s)› ‹𝒟 E2 ⌊dom X⌋›
  obtain TA' e2' x' where IH': sim-move10 P t ta e e' E (hp s) X TA' e2' (hp s') x'
    bisim Vs e2' e' (lcl s') x' ⊆m [Vs [↦] lcl s'] by(fastforce)
  with E2 ‹fv E2 ⊆ set Vs› ‹E2 = E·M(es')› sync show ?case
    by(cases is-val e2')(auto intro: CallObj)
next
  case (Call1Params es s ta es' s' v M Vs E2 X)
  note IH = ‹⋀vs es2 x. ⟦ bisims vs es2 es (lcl s); fvs es2 ⊆ set vs;
          x ⊆m [vs [↦] lcl s]; length vs + max-varss es ≤ length (lcl s); 𝒟s es2 ⌊dom x⌋⟧
        ⟹ ?concls es es' es2 s x ta s' es' vs›
  from ‹bisim Vs E2 (Val v·M(es)) (lcl s)› obtain Es
    where E2 = Val v ·M(Es) bisims Vs Es es (lcl s) by(auto)
  with IH[of Vs Es X] ‹fv E2 ⊆ set Vs› ‹X ⊆m [Vs [↦] lcl s]›
    ‹length Vs + max-vars (Val v·M(es)) ≤ length (lcl s)› ‹𝒟 E2 ⌊dom X⌋›
    ‹E2 = Val v ·M(Es)› show ?case by(fastforce intro: CallParams)
next
  case (Red1CallExternal s a T M Ts Tr D vs ta va h' e' s' Vs E2 X)
  from ‹bisim Vs E2 (addr a·M(map Val vs)) (lcl s)› have E2: E2 = addr a·M(map Val vs) by auto
  moreover from ‹compP1 P ⊢ class-type-of T sees M: Ts→Tr = Native in D›
  have P ⊢ class-type-of T sees M: Ts→Tr = Native in D by simp
  moreover from ‹compP1 P,t ⊢ ⟨a·M(vs),hp s⟩ −ta→ext ⟨va,h'⟩›
  have P,t ⊢ ⟨a·M(vs),hp s⟩ −ta→ext ⟨va,h'⟩ by simp
  moreover from wf ‹P,t ⊢ ⟨a·M(vs),hp s⟩ −ta→ext ⟨va,h'⟩›
  have ta-bisim01 (extTA2J0 P ta) (extTA2J1 (compP1 P) ta)
    by(rule red-external-ta-bisim01)
  moreover note ‹typeof-addr (hp s) a = ⌊T⌋› ‹e' = extRet2J1 (addr a·M(map Val vs)) va› ‹s' =
(h', lcl s)›
  moreover from ‹typeof-addr (hp s) a = ⌊T⌋› ‹P,t ⊢ ⟨a·M(vs),hp s⟩ −ta→ext ⟨va,h'⟩›
    ‹P ⊢ class-type-of T sees M: Ts→Tr = Native in D›
  have τexternal-defs D M ⟹ ta = ε ∧ h' = hp s
    by(fastforce dest: τexternal'-red-external-heap-unchanged τexternal'-red-external-TA-empty simp
add: τexternal'-def τexternal-def)
  ultimately show ?case using ‹X ⊆m [Vs [↦] lcl s]›
  by(fastforce intro!: exI[where x=extTA2J0 P ta] intro: RedCallExternal simp add: map-eq-append-conv
sim-move10-def synthesized-call-def dest: sees-method-fun del: disjCI intro: disjI1 disjI2)
next
  case (Block1Red e h x ta e' h' x' V T Vs E2 X)
  note IH = ‹⋀vs e2 xa. ⟦ bisim vs e2 e (lcl (h, x)); fv e2 ⊆ set vs; xa ⊆m [vs [↦] lcl (h, x)];
                length vs + max-vars e ≤ length (lcl (h, x)); 𝒟 e2 ⌊dom xa⌋⟧
        ⟹ ?concl e e' e2 (h, x) xa ta (h', x') e' vs›
  from ‹False,compP1 P,t ⊢1 ⟨e,(h, x)⟩ −ta→ ⟨e',(h', x')⟩›
  have length x = length x' by(auto dest: red1-preserves-len)
  with ‹length Vs + max-vars {V:T=None; e} ≤ length (lcl (h, x))›
```

```
  have length Vs < length x' by simp
  from ‹bisim Vs E2 {V:T=None; e} (lcl (h, x))›
  show ?case
  proof(cases rule: bisim-cases(14)[consumes 1, case-names BlockNone BlockSome BlockSomeNone])
    case (BlockNone V' E)
    with ‹fv E2 ⊆ set Vs› have fv E ⊆ set (Vs@[V']) by auto
    with IH[of Vs@[V'] E X(V' := None)] BlockNone ‹fv E2 ⊆ set Vs› ‹X ⊆m [Vs [↦] lcl (h, x)]›
      ‹length Vs + max-vars {V:T=None; e} ≤ length (lcl (h, x))› ‹D E2 ⌊dom X⌋›
    obtain TA' e2' X' where IH': sim-move10 P t ta e e' E h (X(V' := None)) TA' e2' h' X'
      bisim (Vs @ [V']) e2' e' x' X' ⊆m [Vs @ [V'] [↦] x']
      by(fastforce simp del: fun-upd-apply)
    { assume V' ∈ set Vs
      hence hidden (Vs @ [V']) (index Vs V') by(rule hidden-index)
      with ‹bisim (Vs @ [V']) E e (lcl (h, x))› have unmod e (index Vs V')
        by(auto intro: hidden-bisim-unmod)
      moreover from ‹length Vs + max-vars {V:T=None; e} ≤ length (lcl (h, x))› ‹V' ∈ set Vs›
      have index Vs V' < length x by(auto intro: index-less-aux)
      ultimately have x ! index Vs V' = x' ! index Vs V'
        using red1-preserves-unmod[OF ‹False,compP1 P,t ⊢1 ⟨e,(h, x)⟩ −ta→ ⟨e',(h', x')⟩›]
        by(simp) }
    with ‹length Vs + max-vars {V:T=None; e} ≤ length (lcl (h, x))›
      ‹X' ⊆m [Vs @ [V'] [↦] x']› ‹length x = length x'› ‹X ⊆m [Vs [↦] lcl (h, x)]›
    have rel: X'(V' := X V') ⊆m [Vs [↦] x'] by(auto intro: Block-lem)

    show ?thesis
    proof(cases X' V')
      case None
      with BlockNone IH' rel show ?thesis by(fastforce intro: BlockRed)
    next
      case (Some v)
      with ‹X' ⊆m [Vs @ [V'] [↦] x']› ‹length Vs < length x'›
      have x' ! length Vs = v by(auto dest: map-le-SomeD)
      with BlockNone IH' rel Some show ?thesis by(fastforce intro: BlockRed)
    qed
  next
    case BlockSome thus ?thesis by simp
  next
    case (BlockSomeNone V' E)
    with ‹fv E2 ⊆ set Vs› have fv E ⊆ set (Vs@[V']) by auto
    with IH[of Vs@[V'] E X(V' ↦ x ! length Vs)] BlockSomeNone ‹fv E2 ⊆ set Vs› ‹X ⊆m [Vs [↦]
lcl (h, x)]›
      ‹length Vs + max-vars {V:T=None; e} ≤ length (lcl (h, x))› ‹D E2 ⌊dom X⌋›
    obtain TA' e2' X' where IH': sim-move10 P t ta e e' E h (X(V' ↦ x ! length Vs)) TA' e2' h' X'
      bisim (Vs @ [V']) e2' e' x' X' ⊆m [Vs @ [V'] [↦] x']
      by(fastforce simp del: fun-upd-apply)
    { assume V' ∈ set Vs
      hence hidden (Vs @ [V']) (index Vs V') by(rule hidden-index)
      with ‹bisim (Vs @ [V']) E e (lcl (h, x))› have unmod e (index Vs V')
        by(auto intro: hidden-bisim-unmod)
      moreover from ‹length Vs + max-vars {V:T=None; e} ≤ length (lcl (h, x))› ‹V' ∈ set Vs›
      have index Vs V' < length x by(auto intro: index-less-aux)
      ultimately have x ! index Vs V' = x' ! index Vs V'
        using red1-preserves-unmod[OF ‹False,compP1 P,t ⊢1 ⟨e,(h, x)⟩ −ta→ ⟨e',(h', x')⟩›]
        by(simp) }
```

```
    with ‹length Vs + max-vars {V:T=None; e} ≤ length (lcl (h, x))›
      ‹X' ⊆m [Vs @ [V'] [↦] x']› ‹length x = length x'› ‹X ⊆m [Vs [↦] lcl (h, x)]›
    have rel: X'(V' := X V') ⊆m [Vs [↦] x'] by(auto intro: Block-lem)
    from ‹sim-move10 P t ta e e' E h (X(V' ↦ x ! length Vs)) TA' e2' h' X'›
    obtain v' where X' V' = ⌊v'⌋
      by(auto simp: sim-move10-def split: if-split-asm dest!: τred0t-lcl-incr τred0r-lcl-incr red-lcl-incr
subsetD)
    with ‹X' ⊆m [Vs @ [V'] [↦] x']› ‹length Vs < length x'›
    have x' ! length Vs = v' by(auto dest: map-le-SomeD)
    with BlockSomeNone IH' rel ‹X' V' = ⌊v'⌋›
    show ?thesis by(fastforce intro: BlockRed)
  qed
next
  case(Block1Some V xs T v e h)
  from ‹bisim vs e2 {V:T=⌊v⌋; e} (lcl (h, xs))› obtain e' V' where e2 = {V':T=⌊v⌋; e'}
    and V = length vs bisim (vs @ [V']) e' e (xs[length vs := v]) by(fastforce)
  moreover have sim-move10 P t ε {length vs:T=⌊v⌋; e} {length vs:T=None; e} {V':T=⌊v⌋; e'} h
x ε {V':T=⌊v⌋; e'} h x
    by(auto)
  moreover from ‹bisim (vs @ [V']) e' e (xs[length vs := v])›
    ‹length vs + max-vars {V:T=⌊v⌋; e} ≤ length (lcl (h, xs))›
  have bisim vs {V':T=⌊v⌋; e'} {length vs:T=None; e} (xs[length vs := v]) by auto
  moreover from ‹x ⊆m [vs [↦] lcl (h, xs)]› ‹length vs + max-vars {V:T=⌊v⌋; e} ≤ length (lcl (h,
xs))›
  have x ⊆m [vs [↦] xs[length vs := v]] by auto
  ultimately show ?case by auto
next
  case (Lock1Synchronized V xs a e h Vs E2 X)
  note len = ‹length Vs + max-vars (sync_V (addr a) e) ≤ length (lcl (h, xs))›
  from ‹bisim Vs E2 (sync_V (addr a) e) (lcl (h, xs))› obtain E
    where E2: E2 = sync(addr a) E e = compE1 (Vs@[fresh-var Vs]) E
    and sync: ¬ contains-insync E and [simp]: V = length Vs by auto
  moreover
  have extTA2J0 P,P,t ⊢ ⟨sync(addr a) E, (h, X)⟩ −{|Lock→a, SyncLock a|}→ ⟨insync(a) E, (h, X)⟩
    by(rule LockSynchronized)
  moreover from ‹fv E2 ⊆ set Vs› fresh-var-fresh[of Vs] sync len
  have bisim Vs (insync(a) E) (insync_{length Vs} (a) e) (xs[length Vs := Addr a])
    unfolding ‹e = compE1 (Vs@[fresh-var Vs]) E› ‹E2 = sync(addr a) E›
    by −(rule bisimInSynchronized,rule compE1-bisim, auto)
  moreover have zip Vs (xs[length Vs := Addr a]) = (zip Vs xs)[length Vs := (arbitrary, Addr a)]
    by(rule sym)(simp add: update-zip)
  hence zip Vs (xs[length Vs := Addr a]) = zip Vs xs by simp
  with ‹X ⊆m [Vs [↦] (lcl (h, xs))]› have X ⊆m [Vs [↦] xs[length Vs := Addr a]]
    by(auto simp add: map-le-def map-upds-def)
  ultimately show ?case by(fastforce intro: sim-move10-SyncLocks)
next
  case (Synchronized1Red2 e s ta e' s' V a Vs E2 X)
  note IH = ‹⋀vs e2 x. ⟦ bisim vs e2 e (lcl s); fv e2 ⊆ set vs;
          x ⊆m [vs [↦] lcl s]; length vs + max-vars e ≤ length (lcl s);
          𝒟 e2 ⌊dom x⌋ ⟧ ⟹ ?concl e e' e2 s x ta s' e' vs›
  from ‹bisim Vs E2 (insync_V (a) e) (lcl s)› obtain E
    where E2: E2 = insync(a) E and bisim: bisim (Vs@[fresh-var Vs]) E e (lcl s)
    and xsa: lcl s ! length Vs = Addr a and [simp]: V = length Vs by auto
  with ‹fv E2 ⊆ set Vs› fresh-var-fresh[of Vs] have fv: (fresh-var Vs) ∉ fv E by auto
```

from ‹*length Vs + max-vars* (*insync*$_V$ (*a*) *e*) ≤ *length* (*lcl s*)› **have** *length Vs* < *length* (*lcl s*) **by** *simp*
{ assume *X* (*fresh-var Vs*) ≠ *None*
then obtain *v* **where** *X* (*fresh-var Vs*) = ⌊*v*⌋ **by** *auto*
with ‹*X* ⊆$_m$ [*Vs* [↦] *lcl s*]› **have** [*Vs* [↦] *lcl s*] (*fresh-var Vs*) = ⌊*v*⌋
by(*auto simp add*: *map-le-def dest*: *bspec*)
hence (*fresh-var Vs*) ∈ *set Vs*
by(*auto simp add*: *map-upds-def set-zip dest*!: *map-of-SomeD*)
moreover have (*fresh-var Vs*) ∉ *set Vs* **by**(*rule fresh-var-fresh*)
ultimately have *False* **by** *contradiction* **}**
hence *X* (*fresh-var Vs*) = *None* **by**(*cases X* (*fresh-var Vs*), *auto*)
hence *X*(*fresh-var Vs* := *None*) = *X* **by**(*auto intro*: *ext*)
moreover from ‹*X* ⊆$_m$ [*Vs* [↦] *lcl s*]›
have *X*(*fresh-var Vs* := *None*) ⊆$_m$ [*Vs* [↦] *lcl s*, (*fresh-var Vs*) ↦ (*lcl s*) ! *length Vs*] **by**(*simp*)
ultimately have *X* ⊆$_m$ [*Vs* @ [*fresh-var Vs*] [↦] *lcl s*]
using ‹*length Vs* < *length* (*lcl s*)› **by**(*auto*)
moreover note *IH*[*of Vs*@[*fresh-var Vs*] *E X*] *bisim E2* ‹*fv E2* ⊆ *set Vs*› ‹*X* ⊆$_m$ [*Vs* [↦] *lcl s*]›
‹*length Vs + max-vars* (*insync*$_V$ (*a*) *e*) ≤ *length* (*lcl s*)› ‹$\mathcal{D}$ *E2* ⌊*dom X*⌋›
ultimately obtain *TA*′ *e2*′ *x*′ **where** *IH*′: *sim-move10 P t ta e e*′ *E* (*hp s*) *X TA*′ *e2*′ (*hp s*′) *x*′
bisim (*Vs* @ [*fresh-var Vs*]) *e2*′ *e*′ (*lcl s*′) *x*′ ⊆$_m$ [*Vs* @ [*fresh-var Vs*] [↦] *lcl s*′] **by** *auto*
hence *dom x*′ ⊆ *dom X* ∪ *fv E*
by(*fastforce iff del*: *domIff simp add*: *sim-move10-def dest*: *red-dom-lcl* *τred0r-dom-lcl*[*OF wf-prog-wwf-prog*[*OF wf*]] *τred0t-dom-lcl*[*OF wf-prog-wwf-prog*[*OF wf*]] *τred0r-fv-subset*[*OF wf-prog-wwf-prog*[*OF wf*]] *split*: *if-split-asm*)
with *fv* ‹*X* (*fresh-var Vs*) = *None*› **have** (*fresh-var Vs*) ∉ *dom x*′ **by** *auto*
hence *x*′ (*fresh-var Vs*) = *None* **by** *auto*
moreover from ‹*False*,*compP1 P*,*t* ⊢*1* ⟨*e*,*s*⟩ −*ta*→ ⟨*e*′,*s*′⟩›
have *length* (*lcl s*) = *length* (*lcl s*′) **by**(*auto dest*: *red1-preserves-len*)
moreover note ‹*x*′ ⊆$_m$ [*Vs* @ [*fresh-var Vs*] [↦] *lcl s*′]› ‹*length Vs* < *length* (*lcl s*)›
ultimately have *x*′ ⊆$_m$ [*Vs* [↦] *lcl s*′] **by**(*auto simp add*: *map-le-def dest*: *bspec*)
moreover from *bisim fv* **have** *unmod e* (*length Vs*) **by**(*auto intro*: *bisim-fv-unmod*)
with ‹*False*,*compP1 P*,*t* ⊢*1* ⟨*e*,*s*⟩ −*ta*→ ⟨*e*′,*s*′⟩› ‹*length Vs* < *length* (*lcl s*)›
have *lcl s* ! *length Vs* = *lcl s*′ ! *length Vs*
by(*auto dest*!: *red1-preserves-unmod*)
with *xsa* **have** *lcl s*′ ! *length Vs* = *Addr a* **by** *simp*
ultimately show *?case* **using** *IH*′ *E2* **by**(*auto intro*: *SynchronizedRed2*)
next
case (*Unlock1Synchronized xs V a*′ *a v h Vs E2 X*)
from ‹*bisim Vs E2* (*insync*$_V$ (*a*) *Val v*) (*lcl* (*h*, *xs*))›
have *E2*: *E2* = *insync*(*a*) *Val v V* = *length Vs xs* ! *length Vs* = *Addr a* **by** *auto*
moreover with ‹*xs* ! *V* = *Addr a*′› **have** [*simp*]: *a*′ = *a* **by** *simp*
have *extTA2J0 P*,*P*,*t* ⊢ ⟨*insync*(*a*) (*Val v*), (*h*, *X*)⟩ −{|*Unlock*→*a*, *SyncUnlock a*|}→ ⟨*Val v*, (*h*, *X*)⟩
by(*rule UnlockSynchronized*)
ultimately show *?case* **using** ‹*X* ⊆$_m$ [*Vs* [↦] *lcl* (*h*, *xs*)]› **by**(*fastforce intro*: *sim-move10-SyncLocks*)
next
case (*Unlock1SynchronizedNull xs V a v h Vs E2 X*)
from ‹*bisim Vs E2* (*insync*$_V$ (*a*) *Val v*) (*lcl* (*h*, *xs*))›
have *V* = *length Vs xs* ! *length Vs* = *Addr a* **by**(*auto*)
with ‹*xs* ! *V* = *Null*› **have** *False* **by** *simp*
thus *?case* **..**
next
case (*Unlock1SynchronizedFail xs V A*′ *a*′ *v h Vs E2 X*)
from ‹*False*› **show** *?case* **..**
next

case (*Red1While b c s Vs E2 X*)
from ‹*bisim Vs E2* (*while* (*b*) *c*) (*lcl s*)› **obtain** *B C*
where *E2*: *E2* = *while* (*B*) *C b* = *compE1 Vs B c* = *compE1 Vs C*
and *sync*: ¬ *contains-insync B* ¬ *contains-insync C* **by** *auto*
moreover have *extTA2J0 P*,*P*,*t* ⊢ ⟨*while* (*B*) *C*, (*hp s*, *X*)⟩ −*ε*→ ⟨*if* (*B*) (*C*;;*while* (*B*) *C*) *else unit*, (*hp s*, *X*)⟩
by(*rule RedWhile*)
hence *sim-move10 P t ε* (*while* (*compE1 Vs B*) (*compE1 Vs C*)) (*if* (*compE1 Vs B*) (*compE1 Vs C*;;*while* (*compE1 Vs B*) (*compE1 Vs C*)) *else unit*) (*while* (*B*) *C*) (*hp s*) *X ε* (*if* (*B*) (*C*;;*while* (*B*) *C*) *else unit*) (*hp s*) *X*
by(*auto simp add*: *sim-move10-def*)
moreover from ‹*fv E2* ⊆ *set Vs*› *E2 sync*
have *bisim Vs* (*if* (*B*) (*C*;; *while* (*B*) *C*) *else unit*)
(*if* (*compE1 Vs B*) (*compE1 Vs* (*C*;; *while*(*B*) *C*)) *else* (*compE1 Vs unit*)) (*lcl s*)
by −(*rule bisimCond*, *auto*)
ultimately show *?case* **using** ‹*X* ⊆*m* [*Vs* [↦] *lcl s*]›
by(*simp*)(*rule exI*, *rule exI*, *rule exI*, *erule conjI*, *auto*)
next
case (*Red1TryCatch h a D C V x e2 Vs E2 X*)
from ‹*bisim Vs E2* (*try Throw a catch*(*C V*) *e2*) (*lcl* (*h*, *x*))›
obtain *E2′ V′* **where** *E2* = *try Throw a catch*(*C V′*) *E2′ V* = *length Vs e2* = *compE1* (*Vs* @ [*V′*]) *E2′*
and *sync*: ¬ *contains-insync E2′* **by**(*auto*)
with ‹*fv E2* ⊆ *set Vs*› **have** *fv E2′* ⊆ *set* (*Vs* @[*V′*]) **by** *auto*
with ‹*e2* = *compE1* (*Vs* @ [*V′*]) *E2′*› *sync* **have** *bisim* (*Vs* @[*V′*]) *E2′ e2* (*x*[*V* := *Addr a*])
by(*auto intro*!: *compE1-bisim*)
with ‹*V* = *length Vs*› ‹*length Vs* + *max-vars* (*try Throw a catch*(*C V*) *e2*) ≤ *length* (*lcl* (*h*, *x*))›
have *bisim Vs* {*V′*:*Class C*=⌊*Addr a*⌋; *E2′*} {*length Vs*:*Class C*=*None*; *e2*} (*x*[*V* := *Addr a*])
by(*auto intro*: *bisimBlockSomeNone*)
moreover from ‹*length Vs* + *max-vars* (*try Throw a catch*(*C V*) *e2*) ≤ *length* (*lcl* (*h*, *x*))›
have [*Vs* [↦] *x*[*length Vs* := *Addr a*]] = [*Vs* [↦] *x*] **by** *simp*
with ‹*X* ⊆*m* [*Vs* [↦] *lcl* (*h*, *x*)]› **have** *X* ⊆*m* [*Vs* [↦] *x*[*length Vs* := *Addr a*]] **by** *simp*
moreover note ‹*e2* = *compE1* (*Vs* @ [*V′*]) *E2′*› ‹*E2* = *try Throw a catch*(*C V′*) *E2′*›
‹*typeof-addr h a* = ⌊*Class-type D*⌋› ‹*compP1 P* ⊢ *D* ≼* *C*› ‹*V* = *length Vs*›
ultimately show *?case* **by**(*auto intro*!: *exI*)
next
case *Red1TryFail* **thus** *?case* **by**(*auto intro*!: *exI sim-move10-TryFail*)
next
case (*List1Red1 e s ta e′ s′ es Vs ES2 X*)
note *IH* = ‹⋀*vs e2 x*. ⟦ *bisim vs e2 e* (*lcl s*); *fv e2* ⊆ *set vs*;
x ⊆*m* [*vs* [↦] *lcl s*]; *length vs* + *max-vars e* ≤ *length* (*lcl s*); *D e2* ⌊*dom x*⌋⟧
⟹ ∃ *TA′ e2′ x′*. *sim-move10 P t ta e e′ e2* (*hp s*) *x TA′ e2′* (*hp s′*) *x′* ∧
bisim vs e2′ e′ (*lcl s′*) ∧ *x′* ⊆*m* [*vs* [↦] *lcl s′*]›
from ‹*bisims Vs ES2* (*e* # *es*) (*lcl s*)› ‹*False*,*compP1 P*,*t* ⊢*1* ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩›
obtain *E ES* **where** *ES2* = *E* # *ES* ¬ *is-val E es* = *compEs1 Vs ES bisim Vs E e* (*lcl s*)
and *sync*: ¬ *contains-insyncs ES* **by**(*auto elim*!: *bisims-cases simp add*: *compEs1-conv-map*)
with *IH*[*of Vs E X*] ‹*fvs ES2* ⊆ *set Vs*› ‹*X* ⊆*m* [*Vs* [↦] *lcl s*]›
‹*length Vs* + *max-varss* (*e* # *es*) ≤ *length* (*lcl s*)› ‹*Ds ES2* ⌊*dom X*⌋›
obtain *TA′ e2′ x′* **where** *IH′*: *sim-move10 P t ta e e′ E* (*hp s*) *X TA′ e2′* (*hp s′*) *x′*
bisim Vs e2′ e′ (*lcl s′*) *x′* ⊆*m* [*Vs* [↦] *lcl s′*] **by** *fastforce*
show *?case*
proof(*cases is-val e2′*)
case *False*
with *IH′* ‹*ES2* = *E* # *ES*› ‹*es* = *compEs1 Vs ES*› *sync* **show** *?thesis* **by**(*auto intro*: *sim-moves10-expr*)

```
  next
    case True
    from ‹fvs ES2 ⊆ set Vs› ‹ES2 = E # ES› ‹es = compEs1 Vs ES› sync
    have bisims Vs ES es (lcl s') by(auto intro: compEs1-bisims)
   with IH' True ‹ES2 = E # ES› ‹es = compEs1 Vs ES› show ?thesis by(auto intro: sim-moves10-expr)
  qed
next
  case (List1Red2 es s ta es' s' v Vs ES2 X)
  note IH = ‹⋀vs es2 x. ⟦bisims vs es2 es (lcl s); fvs es2 ⊆ set vs;
          x ⊆m [vs [↦] lcl s]; length vs + max-varss es ≤ length (lcl s); 𝒟s es2 ⌊dom x⌋⟧
        ⟹ ∃ TA' es2' x'. sim-moves10 P t ta es es' es2 (hp s) x TA' es2' (hp s') x' ∧ bisims vs es2'
es' (lcl s') ∧ x' ⊆m [vs [↦] lcl s']›
  from ‹bisims Vs ES2 (Val v # es) (lcl s)› obtain ES where ES2 = Val v # ES bisims Vs ES es
(lcl s)
    by(auto elim!: bisims-cases)
  with IH[of Vs ES X] ‹fvs ES2 ⊆ set Vs› ‹X ⊆m [Vs [↦] lcl s]›
    ‹length Vs + max-varss (Val v # es) ≤ length (lcl s)› ‹𝒟s ES2 ⌊dom X⌋›
    ‹ES2 = Val v # ES› show ?case by(fastforce intro: sim-moves10-expr)
next
  case Call1ThrowParams
 thus ?case by(fastforce intro: CallThrowParams elim!: bisim-cases simp add: bisims-map-Val-Throw2)
next
  case (Synchronized1Throw2 xs V a' a ad h Vs E2 X)
  from ‹bisim Vs E2 (insync_V (a) Throw ad) (lcl (h, xs))›
  have xs ! length Vs = Addr a and V = length Vs by auto
  with ‹xs ! V = Addr a'› have [simp]: a' = a by simp
  have extTA2J0 P,P,t ⊢ ⟨insync(a) Throw ad, (h, X)⟩ −⦃Unlock→a, SyncUnlock a⦄→ ⟨Throw ad,
(h, X)⟩
    by(rule SynchronizedThrow2)
  with ‹X ⊆m [Vs [↦] lcl (h, xs)]› ‹bisim Vs E2 (insync_V (a) Throw ad) (lcl (h, xs))›
  show ?case by(auto intro: sim-move10-SyncLocks intro!: exI)
next
  case (Synchronized1Throw2Null xs V a a' h Vs E2 X)
  from ‹bisim Vs E2 (insync_V (a) Throw a') (lcl (h, xs))›
  have V = length Vs xs ! length Vs = Addr a by(auto)
  with ‹xs ! V = Null› have False by simp
  thus ?case ..
next
  case (Synchronized1Throw2Fail xs V A' a' a h Vs E2 X)
  from ‹False› show ?case ..
next
  case InstanceOf1Red thus ?case by auto(blast)
next
  case Red1InstanceOf thus ?case by hypsubst-thin auto
next
  case InstanceOf1Throw thus ?case by auto
next
  case CAS1Throw thus ?case by fastforce
next
  case CAS1Throw2 thus ?case by fastforce
next
  case CAS1Throw3 thus ?case by fastforce
qed(simp-all del: fun-upd-apply, (fastforce intro: red-reds.intros simp del: fun-upd-apply simp add:
finfun-upd-apply)+)
```

lemma *bisim-call-Some-call1*:
⟦ *bisim Vs e e' xs*; *call e = ⌊aMvs⌋*; *length Vs + max-vars e' ≤ length xs* ⟧
⟹ ∃ *e'' xs'*. *τred1'r P t h* (*e'*, *xs*) (*e''*, *xs'*) ∧ *call1 e'' = ⌊aMvs⌋* ∧
bisim Vs e e'' xs' ∧ *take* (*length Vs*) *xs* = *take* (*length Vs*) *xs'*

and *bisims-calls-Some-calls1*:
⟦ *bisims Vs es es' xs*; *calls es = ⌊aMvs⌋*; *length Vs + max-varss es' ≤ length xs* ⟧
⟹ ∃ *es'' xs'*. *τreds1'r P t h* (*es'*, *xs*) (*es''*, *xs'*) ∧ *calls1 es'' = ⌊aMvs⌋* ∧
bisims Vs es es'' xs' ∧ *take* (*length Vs*) *xs* = *take* (*length Vs*) *xs'*
proof(*induct rule*: *bisim-bisims.inducts*)
case *bisimCallParams* **thus** *?case*
by(*fastforce simp add*: *is-vals-conv split*: *if-split-asm*)
next
case *bisimBlockNone* **thus** *?case* **by**(*fastforce intro*: *take-eq-take-le-eq*)
next
case (*bisimBlockSome Vs V e e' xs v T*)
from ‹*length Vs + max-vars* {*length Vs*:*T*=⌊*v*⌋; *e'*} ≤ *length xs*›
have *τred1'r P t h* ({*length Vs*:*T*=⌊*v*⌋; *e'*}, *xs*) ({*length Vs*:*T*=*None*; *e'*}, *xs*[*length Vs* := *v*])
by(*auto intro*!: *τred1r-1step Block1Some*)
also from *bisimBlockSome* **obtain** *e'' xs'*
where *τred1'r P t h* (*e'*, *xs*[*length Vs* := *v*]) (*e''*, *xs'*)
and *call1 e'' = ⌊aMvs⌋ bisim* (*Vs* @ [*V*]) *e e'' xs'*
and *take* (*length* (*Vs* @ [*V*])) (*xs*[*length Vs* := *v*]) = *take* (*length* (*Vs* @ [*V*])) *xs'* **by** *auto*
hence *τred1'r P t h* ({*length Vs*:*T*=*None*; *e'*}, *xs*[*length Vs* := *v*]) ({*length Vs*:*T*=*None*; *e''*}, *xs'*)
by *auto*
also from ‹*call1 e'' = ⌊aMvs⌋*› **have** *call1* {*length Vs*:*T*=*None*; *e''*} = *⌊aMvs⌋* **by** *simp*
moreover from ‹*take* (*length* (*Vs* @ [*V*])) (*xs*[*length Vs* := *v*]) = *take* (*length* (*Vs* @ [*V*])) *xs'*›
have *take* (*length Vs*) *xs* = *take* (*length Vs*) *xs'*
by(*auto dest*: *take-eq-take-le-eq*[**where** *m*=*length Vs*] *simp add*: *take-update-cancel*)
moreover {
have *xs'* ! *length Vs* = *take* (*length* (*Vs* @ [*V*])) *xs'* ! *length Vs* **by** *simp*
also note ‹*take* (*length* (*Vs* @ [*V*])) (*xs*[*length Vs* := *v*]) = *take* (*length* (*Vs* @ [*V*])) *xs'*›[*symmetric*]
also have *take* (*length* (*Vs* @ [*V*])) (*xs*[*length Vs* := *v*]) ! *length Vs* = *v*
using ‹*length Vs + max-vars* {*length Vs*:*T*=⌊*v*⌋; *e'*} ≤ *length xs*› **by** *simp*
finally have *bisim Vs* {*V*:*T*=⌊*v*⌋; *e*} {*length Vs*:*T*=*None*; *e''*} *xs'*
using ‹*bisim* (*Vs* @ [*V*]) *e e'' xs'*› **by** *auto* }
ultimately show *?case* **by** *blast*
next
case (*bisimBlockSomeNone Vs V e e' xs v T*)
then obtain *e'' xs'* **where** *τred1'r P t h* (*e'*, *xs*) (*e''*, *xs'*) *call1 e'' = ⌊aMvs⌋ bisim* (*Vs* @ [*V*]) *e e''* *xs'*
and *take* (*length* (*Vs* @ [*V*])) *xs* = *take* (*length* (*Vs* @ [*V*])) *xs'* **by** *auto*
hence *τred1'r P t h* ({*length Vs*:*T*=*None*; *e'*}, *xs*) ({*length Vs*:*T*=*None*; *e''*}, *xs'*) **by** *auto*
moreover from ‹*call1 e'' = ⌊aMvs⌋*› **have** *call1* ({*length Vs*:*T*=*None*; *e''*}) = *⌊aMvs⌋* **by** *simp*
moreover from ‹*take* (*length* (*Vs* @ [*V*])) *xs* = *take* (*length* (*Vs* @ [*V*])) *xs'*›
have *take* (*length Vs*) *xs* = *take* (*length Vs*) *xs'* **by**(*auto intro*: *take-eq-take-le-eq*)
moreover {
have *xs'* ! *length Vs* = *take* (*length* (*Vs* @ [*V*])) *xs'* ! *length Vs* **by** *simp*
also note ‹*take* (*length* (*Vs* @ [*V*])) *xs* = *take* (*length* (*Vs* @ [*V*])) *xs'*›[*symmetric*]
also have *take* (*length* (*Vs* @ [*V*])) *xs* ! *length Vs* = *v* **using** ‹*xs* ! *length Vs* = *v*› **by** *simp*
finally have *bisim Vs* {*V*:*T*=⌊*v*⌋; *e*} {*length Vs*:*T*=*None*; *e''*} *xs'*
using ‹*bisim* (*Vs* @ [*V*]) *e e'' xs'*› **by** *auto* }
ultimately show *?case* **by** *blast*

next
case (*bisimInSynchronized Vs e e' xs a*)
then obtain *e'' xs'* **where** *τred1'r P t h* (*e'*, *xs*) (*e''*, *xs'*) *call1 e'' = ⌊aMvs⌋ bisim* (*Vs* @ [*fresh-var Vs*]) *e e'' xs'*
and *take* (*Suc* (*length Vs*)) *xs* = *take* (*Suc* (*length Vs*)) *xs'* **by** *auto*
hence *τred1'r P t h* (*insync*$_{length\ Vs}$ (*a*) *e'*, *xs*) (*insync*$_{length\ Vs}$ (*a*) *e''*, *xs'*) **by** *auto*
moreover from ‹*call1 e'' = ⌊aMvs⌋*› **have** *call1* (*insync*$_{length\ Vs}$ (*a*) *e''*) = *⌊aMvs⌋* **by** *simp*
moreover from ‹*take* (*Suc* (*length Vs*)) *xs* = *take* (*Suc* (*length Vs*)) *xs'*›
have *take* (*length Vs*) *xs* = *take* (*length Vs*) *xs'* **by**(*auto intro*: *take-eq-take-le-eq*)
moreover {
have *xs'* ! *length Vs* = *take* (*Suc* (*length Vs*)) *xs'* ! *length Vs* **by** *simp*
also note ‹*take* (*Suc* (*length Vs*)) *xs* = *take* (*Suc* (*length Vs*)) *xs'*›[*symmetric*]
also have *take* (*Suc* (*length Vs*)) *xs* ! *length Vs* = *Addr a*
using ‹*xs* ! *length Vs* = *Addr a*› **by** *simp*
finally have *bisim Vs* (*insync*(*a*) *e*) (*insync*$_{length\ Vs}$ (*a*) *e''*) *xs'*
using ‹*bisim* (*Vs* @ [*fresh-var Vs*]) *e e'' xs'*› **by** *auto* **}**
ultimately show *?case* **by** *blast*
next
case *bisimsCons1* **thus** *?case* **by**(*fastforce intro*!: *τred1r-inj-τreds1r*)
next
case *bisimsCons2* **thus** *?case* **by**(*fastforce intro*!: *τreds1r-cons-τreds1r*)
qed *fastforce+*

lemma *sim-move01-into-Red1*:
sim-move01 P t ta e E' h xs ta' e2' h' xs'
⟹ *if τMove0 P h* (*e*, *es1*)
then τRed1't P t h ((*E'*, *xs*), *exs2*) ((*e2'*, *xs'*), *exs2*) ∧ *ta* = *ε* ∧ *h* = *h'*
else ∃ *ex2' exs2' ta'*. *τRed1'r P t h* ((*E'*, *xs*), *exs2*) (*ex2'*, *exs2'*) ∧
(*call e* = *None* ∨ *call1 E'* = *None* ∨ *ex2'* = (*E'*, *xs*) ∧ *exs2'* = *exs2*) ∧
False,*P*,*t* ⊢*1* ⟨*ex2'*/*exs2'*,*h*⟩ −*ta'*→ ⟨(*e2'*, *xs'*)/*exs2*,*h'*⟩ ∧
¬ *τMove1 P h* (*ex2'*, *exs2'*) ∧ *ta-bisim01 ta ta'*
apply(*auto simp add*: *sim-move01-def intro*: *τred1t-into-τRed1t τred1r-into-τRed1r red1Red split*: *if-split-asm*)
apply(*fastforce intro*: *red1Red intro*!: *τred1r-into-τRed1r*)+
done

lemma *sim-move01-max-vars-decr*:
sim-move01 P t ta e0 e h xs ta' e' h' xs' ⟹ *max-vars e'* ≤ *max-vars e*
by(*fastforce simp add*: *sim-move01-def split*: *if-split-asm dest*: *τred1t-max-vars red1-max-vars-decr τred1r-max-vars*)

lemma *Red1-simulates-red0*:
assumes *wf*: *wf-J-prog P*
and *red*: *P*,*t* ⊢*0* ⟨*e1*/*es1*, *h*⟩ −*ta*→ ⟨*e1'*/*es1'*, *h'*⟩
and *bisiml*: *bisim-list1* (*e1*, *es1*) (*ex2*, *exs2*)
shows ∃ *ex2'' exs2''*. *bisim-list1* (*e1'*, *es1'*) (*ex2''*, *exs2''*) ∧
(*if τMove0 P h* (*e1*, *es1*)
then τRed1't (*compP1 P*) *t h* (*ex2*, *exs2*) (*ex2''*, *exs2''*) ∧ *ta* = *ε* ∧ *h* = *h'*
else ∃ *ex2' exs2' ta'*. *τRed1'r* (*compP1 P*) *t h* (*ex2*, *exs2*) (*ex2'*, *exs2'*) ∧
(*call e1* = *None* ∨ *call1* (*fst ex2*) = *None* ∨ *ex2'* = *ex2* ∧ *exs2'* = *exs2*) ∧
False,*compP1 P*,*t* ⊢*1* ⟨*ex2'*/*exs2'*, *h*⟩ −*ta'*→ ⟨*ex2''*/*exs2''*, *h'*⟩ ∧
¬ *τMove1 P h* (*ex2'*, *exs2'*) ∧ *ta-bisim01 ta ta'*)
(**is** ∃ *ex2'' exs2''* . - ∧ *?red ex2'' exs2''*)
using *red*

proof(*cases*)
case (*red0Red XS'*)
note [*simp*] = ‹*es1'* = *es1*›
and *red* = ‹*extTA2J0 P,P,t* ⊢ ⟨*e1*,(*h*, *Map.empty*)⟩ −*ta*→ ⟨*e1'*,(*h'*, *XS'*)⟩›
and *notsynth* = ‹∀ *aMvs*. *call e1* = ⌊*aMvs*⌋ ⟶ *synthesized-call P h aMvs*›
from *bisiml* **obtain** *E xs* **where** *ex2*: *ex2* = (*E*, *xs*)
and *bisim*: *bisim* [] *e1 E xs* **and** *fv*: *fv e1* = {}
and *length*: *max-vars E* ≤ *length xs* **and** *bsl*: *bisim-list es1 exs2*
and *D*: *D e1* ⌊{}⌋ **by**(*auto elim*!: *bisim-list1-elim*)
from *bisim-max-vars*[*OF bisim*] *length red1-simulates-red-aux*[*OF wf red bisim*] *fv notsynth*
obtain *ta' e2' xs'* **where** *sim*: *sim-move01* (*compP1 P*) *t ta e1 E h xs ta' e2' h' xs'*
and *bisim'*: *bisim* [] *e1' e2' xs'* **and** *XS'*: *XS'* ⊆$_m$ *Map.empty* **by** *auto*
from *sim-move01-into-Red1*[*OF sim*, *of es1 exs2*]
have *?red* (*e2'*, *xs'*) *exs2* **unfolding** *ex2* **by** *auto*
moreover {
note *bsl bisim'* **moreover**
from *fv red-fv-subset*[*OF wf-prog-wwf-prog*[*OF wf*] *red*]
have *fv e1'* = {} **by** *simp*
moreover from *red D* **have** *D e1'* ⌊*dom XS'*⌋
by(*auto dest*: *red-preserves-defass*[*OF wf*] *split*: *if-split-asm*)
with *red-dom-lcl*[*OF red*] ‹*fv e1* = {}› **have** *D e1'* ⌊{}⌋
by(*auto elim*!: *D-mono'* *simp add*: *hyperset-defs*)
moreover from *sim* **have** *length xs* = *length xs'* *max-vars e2'* ≤ *max-vars E*
by(*auto dest*: *sim-move01-preserves-len sim-move01-max-vars-decr*)
with *length* **have** *length'*: *max-vars e2'* ≤ *length xs'* **by**(*auto*)
ultimately have *bisim-list1* (*e1'*, *es1*) ((*e2'*, *xs'*), *exs2*) **by**(*rule bisim-list1I*) }
ultimately show *?thesis* **using** *ex2* **by**(*simp split del*: *if-split*)(*rule exI conjI*|*assumption*)+
next
case (*red0Call a M vs U Ts T pns body D*)
note [*simp*] = ‹*ta* = *ε*› ‹*h'* = *h*›
and *es1'* = ‹*es1'* = *e1* # *es1*›
and *e1'* = ‹*e1'* = *blocks* (*this* # *pns*) (*Class D* # *Ts*) (*Addr a* # *vs*) *body*›
and *call* = ‹*call e1* = ⌊(*a*, *M*, *vs*)⌋›
and *ha* = ‹*typeof-addr h a* = ⌊*U*⌋›
and *sees* = ‹*P* ⊢ *class-type-of U sees M*: *Ts*→*T* = ⌊(*pns*, *body*)⌋ *in D*›
and *len* = ‹*length vs* = *length pns*› ‹*length Ts* = *length pns*›
from *bisiml* **obtain** *E xs* **where** *ex2*: *ex2* = (*E*, *xs*)
and *bisim*: *bisim* [] *e1 E xs* **and** *fv*: *fv e1* = {}
and *length*: *max-vars E* ≤ *length xs* **and** *bsl*: *bisim-list es1 exs2*
and *D*: *D e1* ⌊{}⌋ **by**(*auto elim*!: *bisim-list1-elim*)

from *bisim-call-Some-call1*[*OF bisim call*, *of compP1 P t h*] *length*
obtain *e' xs'* **where** *red*: *τred1'r* (*compP1 P*) *t h* (*E*, *xs*) (*e'*, *xs'*)
and *call1 e'* = ⌊(*a*, *M*, *vs*)⌋ *bisim* [] *e1 e' xs'*
and *take 0 xs* = *take 0 xs'* **by** *auto*

let *?e1'* = *blocks* (*this* # *pns*) (*Class D* # *Ts*) (*Addr a* # *vs*) *body*
let *?e2'* = *blocks1 0* (*Class D*#*Ts*) (*compE1* (*this* # *pns*) *body*)
let *?xs2'* = *Addr a* # *vs* @ *replicate* (*max-vars* (*compE1* (*this* # *pns*) *body*)) *undefined-value*
let *?exs2'* = (*e'*, *xs'*) # *exs2*

have *τRed1'r* (*compP1 P*) *t h* ((*E*, *xs*), *exs2*) ((*e'*, *xs'*), *exs2*)
using *red* **by**(*rule τred1r-into-τRed1r*)
moreover {

note ‹*call1 e′* = ⌊(*a*, *M*, *vs*)⌋›
moreover note *ha*
moreover have *compP1 P* ⊢ *class-type-of U sees M*:*Ts* → *T* = *map-option* (λ(*pns*, *body*). *compE1* (*this* # *pns*) *body*) ⌊(*pns*, *body*)⌋ *in D*
using *sees* **unfolding** *compP1-def* **by**(*rule sees-method-compP*)
hence *sees′*: *compP1 P* ⊢ *class-type-of U sees M*:*Ts* → *T* = ⌊*compE1* (*this* # *pns*) *body*⌋ *in D* **by** *simp*
moreover from *len* **have** *length vs* = *length Ts* **by** *simp*
ultimately have *False*,*compP1 P*,*t* ⊢*1* ⟨(*e′*, *xs′*)/*exs2*,*h*⟩ −*ε*→ ⟨(*?e2′*, *?xs2′*)/*?exs2′*, *h*⟩ **by**(*rule red1Call*)
moreover have *τMove1 P h* ((*e′*, *xs′*), *exs2*) **using** ‹*call1 e′* = ⌊(*a*, *M*, *vs*)⌋› *ha sees*
by(*auto simp add*: *synthesized-call-def τexternal′-def dest*: *sees-method-fun dest*!: *τmove1-not-call1*[**where** *P*=*P* **and** *h*=*h*])
ultimately have *τRed1′* (*compP1 P*) *t h* ((*e′*, *xs′*), *exs2*) ((*?e2′*, *?xs2′*), *?exs2′*) **by** *auto* **}**
ultimately have *τRed1′t* (*compP1 P*) *t h* ((*E*, *xs*), *exs2*) ((*?e2′*, *?xs2′*), *?exs2′*) **by**(*rule rtranclp-into-tranclp1*)
moreover
{ from *red* **have** *length xs′* = *length xs* **by**(*rule τred1r-preserves-len*)
moreover from *red* **have** *max-vars e′* ≤ *max-vars E* **by**(*rule τred1r-max-vars*)
ultimately have *max-vars e′* ≤ *length xs′* **using** *length* **by** *simp* **}**
with *bsl* ‹*bisim* [] *e1 e′ xs′*› *fv D* **have** *bisim-list* (*e1* # *es1*) *?exs2′*
using ‹*call e1* = ⌊(*a*, *M*, *vs*)⌋› ‹*call1 e′* = ⌊(*a*, *M*, *vs*)⌋› **by**(*rule bisim-listCons*)
hence *bisim-list1* (*e1′*, *es1′*) ((*?e2′*, *?xs2′*), *?exs2′*)
unfolding *e1′ es1′*
proof(*rule bisim-list1I*)
from *wf-prog-wwf-prog*[*OF wf*] *sees*
have *wf-mdecl wwf-J-mdecl P D* (*M*,*Ts*,*T*,⌊(*pns*,*body*)⌋) **by**(*rule sees-wf-mdecl*)
hence *fv′*: *fv body* ⊆ *set pns* ∪ {*this*} **by**(*auto simp add*: *wf-mdecl-def*)
moreover from ‹*P* ⊢ *class-type-of U sees M*: *Ts*→*T* = ⌊(*pns*, *body*)⌋ *in D*› **have** ¬ *contains-insync body*
by(*auto dest*!: *sees-wf-mdecl*[*OF wf*] *WT-expr-locks simp add*: *wf-mdecl-def contains-insync-conv*)
ultimately have *bisim* ([*this*] @ *pns*) *body* (*compE1* ([*this*] @ *pns*) *body*) *?xs2′*
by −(*rule compE1-bisim*, *auto*)
with ‹*length vs* = *length pns*› ‹*length Ts* = *length pns*›
have *bisim* ([] @ [*this*]) (*blocks pns Ts vs body*) (*blocks1* (*Suc 0*) *Ts* (*compE1* (*this* # *pns*) *body*)) *?xs2′*
by −(*drule blocks-bisim*,*auto simp add*: *nth-append*)
from *bisimBlockSomeNone*[*OF this*, *of Addr a Class D*]
show *bisim* [] *?e1′ ?e2′ ?xs2′* **by** *simp*
from *fv′ len* **show** *fv ?e1′* = {} **by** *auto*

from *wf sees*
have *wf-mdecl wf-J-mdecl P D* (*M*,*Ts*,*T*,⌊(*pns*,*body*)⌋) **by**(*rule sees-wf-mdecl*)
hence *D body* ⌊*set pns* ∪ {*this*}⌋ **by**(*auto simp add*: *wf-mdecl-def*)
with ‹*length vs* = *length pns*› ‹*length Ts* = *length pns*›
have *D* (*blocks pns Ts vs body*) ⌊*dom* [*this* ↦ *Addr a*]⌋ **by**(*auto*)
thus *D ?e1′* ⌊{}⌋ **by** *auto*

from *len* **show** *max-vars ?e2′* ≤ *length ?xs2′* **by**(*auto simp add*: *blocks1-max-vars*)
qed
moreover have *τMove0 P h* (*e1*, *es1*) **using** *call ha sees*
by(*fastforce simp add*: *synthesized-call-def τexternal′-def dest*: *sees-method-fun τmove0-callD*[**where** *P*=*P* **and** *h*=*h*])
ultimately show *?thesis* **using** *ex2* ‹*call e1* = ⌊(*a*, *M*, *vs*)⌋›

```
    by(simp del: τMove1.simps)(rule exI conjI|assumption)+
next
  case (red0Return e)
  note es1 = ‹es1 = e # es1'› and e1' = ‹e1' = inline-call e1 e›
    and [simp] = ‹ta = ε› ‹h' = h›
    and fin = ‹final e1›
  from bisiml es1 obtain E' xs' E xs exs' aMvs where ex2: ex2 = (E', xs') and exs2: exs2 = (E,
xs) # exs'
    and bisim': bisim [] e1 E' xs'
    and bisim: bisim [] e E xs
    and fv: fv e = {}
    and length: max-vars E ≤ length xs
    and bisiml: bisim-list es1' exs'
    and D: 𝒟 e ⌊{}⌋
    and call: call e = ⌊aMvs⌋
    and call1: call1 E = ⌊aMvs⌋
    by(fastforce elim: bisim-list1-elim)
  let ?e2' = inline-call E' E

  from ‹final e1› bisim' have final E' by(auto)
  hence red': False,compP1 P,t ⊢1 ⟨ex2/exs2, h⟩ −ε→ ⟨(?e2', xs)/exs', h⟩
    unfolding ex2 exs2 by(rule red1Return)
  moreover have τMove0 P h (e1, es1) = τMove1 P h ((E', xs'), exs2)
    using ‹final e1› ‹final E'› by auto
  moreover {
    note bisiml
    moreover
    from bisim' fin bisim
    have bisim [] (inline-call e1 e) (inline-call E' E) xs
      using call by(rule bisim-inline-call)(simp add: fv)
    moreover from fv-inline-call[of e1 e] fv fin
    have fv (inline-call e1 e) = {} by auto
    moreover from fin have 𝒟 (inline-call e1 e) ⌊{}⌋
      using call D by(rule defass-inline-call)
    moreover have max-vars ?e2' ≤ max-vars E + max-vars E' by(rule inline-call-max-vars1)
    with ‹final E'› length have max-vars ?e2' ≤ length xs by(auto elim!: final.cases)
    ultimately have bisim-list1 (inline-call e1 e, es1') ((?e2', xs), exs')
      by(rule bisim-list1I) }
  ultimately show ?thesis using e1' ‹final e1› ‹final E'› ex2
    apply(simp del: τMove0.simps τMove1.simps)
    apply(rule exI conjI impI|assumption)+
     apply(rule tranclp.r-into-trancl, simp)
    apply blast
    done
qed

lemma sim-move10-into-red0:
  assumes wwf: wwf-J-prog P
  and sim: sim-move10 P t ta e2 e2' e h Map.empty ta' e' h' x'
  and fv: fv e = {}
  shows if τmove1 P h e2
        then (τRed0t P t h (e, es) (e', es) ∨ countInitBlock e2' < countInitBlock e2 ∧ e' = e ∧ x' =
Map.empty) ∧ ta = ε ∧ h = h'
        else ∃ e'' ta'. τRed0r P t h (e, es) (e'', es) ∧
```

```
                    (call1 e2 = None ∨ call e = None ∨ e'' = e) ∧
                    P,t ⊢0 ⟨e''/es,h⟩ −ta'→ ⟨e'/es,h'⟩ ∧
                    ¬ τMove0 P h (e'', es) ∧ ta-bisim01 ta' (extTA2J1 (compP1 P) ta)
proof(cases τmove1 P h e2)
  case True
  with sim have ¬ final e2
    and red: τred0t (extTA2J0 P) P t h (e, Map.empty) (e', x') ∨
          countInitBlock e2' < countInitBlock e2 ∧ e' = e ∧ x' = Map.empty
    and [simp]: h' = h ta = ε ta' = ε by(simp-all add: sim-move10-def)
  from red have τRed0t P t h (e, es) (e', es) ∨
              countInitBlock e2' < countInitBlock e2 ∧ e' = e ∧ x' = Map.empty
  proof
    assume red: τred0t (extTA2J0 P) P t h (e, Map.empty) (e', x')
    from τred0t-fv-subset[OF wwf red] τred0t-dom-lcl[OF wwf red] fv
    have dom x' ⊆ {} by(auto split: if-split-asm)
    hence x' = Map.empty by auto
    with red have τred0t (extTA2J0 P) P t h (e, Map.empty) (e', Map.empty) by simp
    with wwf have τRed0t P t h (e, es) (e', es)
      using fv by(rule τred0t-into-τRed0t)
    thus ?thesis ..
  qed simp
  with True show ?thesis by simp
next
  case False
  with sim obtain e'' xs'' where ¬ final e2
    and τred: τred0r (extTA2J0 P) P t h (e, Map.empty) (e'', xs'')
    and red: extTA2J0 P,P,t ⊢ ⟨e'',(h, xs'')⟩ −ta'→ ⟨e',(h', x')⟩
    and call: call1 e2 = None ∨ call e = None ∨ e'' = e
    and ¬ τmove0 P h e'' ta-bisim01 ta' (extTA2J1 (compP1 P) ta) no-call P h e''
    by(auto simp add: sim-move10-def split: if-split-asm)
  from τred0r-fv-subset[OF wwf τred] τred0r-dom-lcl[OF wwf τred] fv
  have dom xs'' ⊆ {} by(auto)
  hence xs'' = Map.empty by(auto)
  with τred have τred0r (extTA2J0 P) P t h (e, Map.empty) (e'', Map.empty) by simp
  with wwf have τRed0r P t h (e, es) (e'', es)
    using fv by(rule τred0r-into-τRed0r)
  moreover from red ‹xs'' = Map.empty›
  have extTA2J0 P,P,t ⊢ ⟨e'',(h, Map.empty)⟩ −ta'→ ⟨e',(h', x')⟩ by simp
  from red0Red[OF this] ‹no-call P h e''›
  have P,t ⊢0 ⟨e''/es,h⟩ −ta'→ ⟨e'/es,h'⟩ by(simp add: no-call-def)
  moreover from ‹¬ τmove0 P h e''› red
  have ¬ τMove0 P h (e'', es) by auto
  ultimately show ?thesis using False ‹ta-bisim01 ta' (extTA2J1 (compP1 P) ta)› call
    by(simp del: τMove0.simps) blast
qed

lemma red0-simulates-Red1:
  assumes wf: wf-J-prog P
  and red: False,compP1 P,t ⊢1 ⟨ex2/exs2, h⟩ −ta→ ⟨ex2'/exs2', h'⟩
  and bisiml: bisim-list1 (e, es) (ex2, exs2)
  shows ∃ e' es'. bisim-list1 (e', es') (ex2', exs2') ∧
                (if τMove1 P h (ex2, exs2)
                   then (τRed0t P t h (e, es) (e', es') ∨ countInitBlock (fst ex2') < countInitBlock (fst
ex2) ∧ exs2' = exs2 ∧ e' = e ∧ es' = es) ∧
```

```
                ta = ε ∧ h = h'
            else ∃ e'' es'' ta'. τRed0r P t h (e, es) (e'', es'') ∧
                                 (call1 (fst ex2) = None ∨ call e = None ∨ e'' = e ∧ es'' = es) ∧
                                 P,t ⊢0 ⟨e''/es'', h⟩ −ta'→ ⟨e'/es', h'⟩ ∧
                                 ¬ τMove0 P h (e'', es'') ∧ ta-bisim01 ta' ta)
  (is ∃ e' es' . - ∧ ?red e' es')
using red
proof(cases)
  case (red1Red E xs TA E' xs')
  note red = ‹False,compP1 P,t ⊢1 ⟨E,(h, xs)⟩ −TA→ ⟨E',(h', xs')⟩›
    and ex2 = ‹ex2 = (E, xs)›
    and ex2' = ‹ex2' = (E', xs')›
    and [simp] = ‹ta = extTA2J1 (compP1 P) TA› ‹exs2' = exs2›
  from bisiml ex2 have bisim: bisim [] e E xs and fv: fv e = {}
    and length: max-vars E ≤ length xs and bsl: bisim-list es exs2
    and D: 𝒟 e ⌊{}⌋ by(auto elim: bisim-list1-elim)
  from red-simulates-red1-aux[OF wf red, simplified, OF bisim, of Map.empty] fv length D
  obtain TA' e2' x' where red': sim-move10 P t TA E E' e h Map.empty TA' e2' h' x'
    and bisim'': bisim [] e2' E' xs' and lcl': x' ⊆m Map.empty by auto
  from red have ¬ final E by auto
  with sim-move10-into-red0[OF wf-prog-wwf-prog[OF wf] red', of es] fv ex2 ex2'
  have red'': ?red e2' es by fastforce
  moreover {
    note bsl bisim''
    moreover from red' fv have fv e2' = {}
    by(fastforce simp add: sim-move10-def split: if-split-asm dest: τred0r-fv-subset[OF wf-prog-wwf-prog[OF
wf]] τred0t-fv-subset[OF wf-prog-wwf-prog[OF wf]] red-fv-subset[OF wf-prog-wwf-prog[OF wf]])
    moreover from red' have dom x' ⊆ dom (Map.empty) ∪ fv e
      unfolding sim-move10-def
        apply(auto split: if-split-asm del: subsetI dest: τred0r-dom-lcl[OF wf-prog-wwf-prog[OF wf]]
τred0t-dom-lcl[OF wf-prog-wwf-prog[OF wf]])
      apply(frule-tac [1−2] τred0r-fv-subset[OF wf-prog-wwf-prog[OF wf]])
      apply(auto dest!: τred0r-dom-lcl[OF wf-prog-wwf-prog[OF wf]] red-dom-lcl del: subsetI, blast+)
      done
    with fv have dom x' ⊆ {} by(auto)
    hence x' = Map.empty by(auto)
    with D red' have 𝒟 e2' ⌊{}⌋
      by(auto dest!: sim-move10-preserves-defass[OF wf] split: if-split-asm)
    moreover from red have length xs' = length xs by(auto dest: red1-preserves-len)
    with red1-max-vars[OF red] length
    have max-vars E' ≤ length xs' by simp
    ultimately have bisim-list1 (e2', es) ((E', xs'), exs2)
      by(rule bisim-list1I) }
  ultimately show ?thesis using ex2'
    by(clarsimp split: if-split-asm)(rule exI conjI|assumption|simp)+
next
  case (red1Call E a M vs U Ts T body D xs)
  note [simp] = ‹ex2 = (E, xs)› ‹h' = h› ‹ta = ε›
    and ex2' = ‹ex2' = (blocks1 0 (Class D#Ts) body, Addr a # vs @ replicate (max-vars body)
undefined-value)›
    and exs' = ‹exs2' = (E, xs) # exs2›
    and call = ‹call1 E = ⌊(a, M, vs)⌋› and ha = ‹typeof-addr h a = ⌊U⌋›
    and sees = ‹compP1 P ⊢ class-type-of U sees M: Ts→T = ⌊body⌋ in D›
    and len = ‹length vs = length Ts›
```

let *?e2'* = *blocks1 0* (*Class D#Ts*) *body*
let *?xs2'* = *Addr a # vs @ replicate* (*max-vars body*) *undefined-value*
from *bisiml* **have** *bisim*: *bisim* [] *e E xs* **and** *fv*: *fv e* = {}
and *length*: *max-vars E* ≤ *length xs* **and** *bsl*: *bisim-list es exs2*
and *D*: *𝒟 e* ⌊{}⌋ **by**(*auto elim*: *bisim-list1-elim*)

from *bisim* ‹*call1 E* = ⌊(*a*, *M*, *vs*)⌋›
have *call e* = ⌊(*a*, *M*, *vs*)⌋ **by**(*rule bisim-call1-Some-call*)
moreover note *ha*
moreover from ‹*compP1 P* ⊢ *class-type-of U sees M*: *Ts*→*T* = ⌊*body*⌋ *in D*›
obtain *pns Jbody* **where** *sees'*: *P* ⊢ *class-type-of U sees M*: *Ts*→*T* = ⌊(*pns*, *Jbody*)⌋ *in D*
and *body*: *body* = *compE1* (*this # pns*) *Jbody*
by(*auto dest*: *sees-method-compPD*)
let *?e'* = *blocks* (*this # pns*) (*Class D # Ts*) (*Addr a # vs*) *Jbody*
note *sees'* **moreover from** *wf sees'* **have** *length Ts* = *length pns*
by(*auto dest*!: *sees-wf-mdecl simp add*: *wf-mdecl-def*)
with *len* **have** *length vs* = *length pns length Ts* = *length pns* **by** *simp-all*
ultimately have *red'*: *P*,*t* ⊢*0* ⟨*e*/*es*, *h*⟩ −*ε*→ ⟨*?e'*/*e*#*es*, *h*⟩ **by**(*rule red0Call*)
moreover from ‹*call e* = ⌊(*a*, *M*, *vs*)⌋› *ha sees'*
have *τMove0 P h* (*e*, *es*)
by(*fastforce simp add*: *synthesized-call-def dest*: *τmove0-callD*[**where** *P*=*P* **and** *h*=*h*] *sees-method-fun*)
ultimately have *τRed0t P t h* (*e*, *es*) (*?e'*, *e*#*es*) **by** *auto*
moreover
from *bsl bisim fv D length* ‹*call e* = ⌊(*a*, *M*, *vs*)⌋› ‹*call1 E* = ⌊(*a*, *M*, *vs*)⌋›
have *bisim-list* (*e # es*) ((*E*, *xs*) *# exs2*) **by**(*rule bisim-list.intros*)
hence *bisim-list1* (*?e'*, *e # es*) (*ex2'*, (*E*, *xs*) *# exs2*) **unfolding** *ex2'*
proof(*rule bisim-list1I*)
from *wf-prog-wwf-prog*[*OF wf*] *sees'*
have *wf-mdecl wwf-J-mdecl P D* (*M*,*Ts*,*T*,⌊(*pns*,*Jbody*)⌋) **by**(*rule sees-wf-mdecl*)
hence *fv Jbody* ⊆ *set pns* ∪ {*this*} **by**(*auto simp add*: *wf-mdecl-def*)
moreover from *sees'* **have** ¬ *contains-insync Jbody*
by(*auto dest*!: *sees-wf-mdecl*[*OF wf*] *WT-expr-locks simp add*: *wf-mdecl-def contains-insync-conv*)
ultimately have *bisim* ([] @ *this # pns*) *Jbody* (*compE1* ([] @ *this # pns*) *Jbody*) *?xs2'*
by −(*rule compE1-bisim*, *auto*)
with ‹*length vs* = *length Ts*› ‹*length Ts* = *length pns*› *body*
have *bisim* [] *?e'* (*blocks1* (*length* ([] :: *vname list*)) (*Class D # Ts*) *body*) *?xs2'*
by −(*rule blocks-bisim*, *auto simp add*: *nth-append nth-Cons'*)
thus *bisim* [] *?e' ?e2' ?xs2'* **by** *simp*
from ‹*length vs* = *length Ts*› ‹*length Ts* = *length pns*› ‹*fv Jbody* ⊆ *set pns* ∪ {*this*}›
show *fv ?e'* = {} **by** *auto*
from *wf sees'*
have *wf-mdecl wf-J-mdecl P D* (*M*,*Ts*,*T*,⌊(*pns*,*Jbody*)⌋) **by**(*rule sees-wf-mdecl*)
hence *𝒟 Jbody* ⌊*set pns* ∪ {*this*}⌋ **by**(*auto simp add*: *wf-mdecl-def*)
with ‹*length vs* = *length Ts*› ‹*length Ts* = *length pns*›
have *𝒟* (*blocks pns Ts vs Jbody*) ⌊*dom* [*this* ↦ *Addr a*]⌋ **by**(*auto*)
thus *𝒟 ?e'* ⌊{}⌋ **by** *simp*
from *len* **show** *max-vars ?e2'* ≤ *length ?xs2'* **by**(*simp add*: *blocks1-max-vars*)
qed
moreover have *τMove1 P h* (*ex2*, *exs2*) **using** *ha* ‹*call1 E* = ⌊(*a*, *M*, *vs*)⌋› *sees'*
by(*auto simp add*: *synthesized-call-def τexternal'-def dest*!: *τmove1-not-call1*[**where** *P*=*P* **and** *h*=*h*] *dest*: *sees-method-fun*)
ultimately show *?thesis* **using** *exs'*
by(*simp del*: *τMove1.simps τMove0.simps*)(*rule exI conjI rtranclp.rtrancl-refl*|*assumption*)+
next

```
  case (red1Return E' x' E x)
  note [simp] = ‹h' = h› ‹ta = ε›
    and ex2 = ‹ex2 = (E', x')› and exs2 = ‹exs2 = (E, x) # exs2'›
    and ex2' = ‹ex2' = (inline-call E' E, x)›
    and fin = ‹final E'›
  from bisiml ex2 exs2 obtain e' es' aMvs where es: es = e' # es'
    and bsl: bisim-list es' exs2'
    and bisim: bisim [] e E' x'
    and bisim': bisim [] e' E x
    and fv: fv e' = {}
    and length: max-vars E ≤ length x
    and D: 𝒟 e' ⌊{}⌋
    and call: call e' = ⌊aMvs⌋
    and call1: call1 E = ⌊aMvs⌋
    by(fastforce elim!: bisim-list1-elim)

  from ‹final E'› bisim have fin': final e by(auto)
  hence P,t ⊢0 ⟨e/e' # es',h⟩ -ε→ ⟨inline-call e e'/es',h⟩ by(rule red0Return)
  moreover from bisim fin' bisim' call
  have bisim [] (inline-call e e') (inline-call E' E) x
    by(rule bisim-inline-call)(simp add: fv)
  with bsl have bisim-list1 (inline-call e e', es') (ex2', exs2') unfolding ex2'
  proof(rule bisim-list1I)
    from fin' fv-inline-call[of e e'] fv show fv (inline-call e e') = {} by auto
    from fin' show 𝒟 (inline-call e e') ⌊{}⌋ using call D by(rule defass-inline-call)

    from call1-imp-call[OF call1]
    have max-vars (inline-call E' E) ≤ max-vars E + max-vars E'
      by(rule inline-call-max-vars)
    with fin length show max-vars (inline-call E' E) ≤ length x by(auto elim!: final.cases)
  qed
  moreover have τMove1 P h (ex2, exs2) = τMove0 P h (e, es) using ex2 call1 call fin fin' by(auto)
  ultimately show ?thesis using es
    by(simp del: τMove1.simps τMove0.simps) blast
qed

end

sublocale J0-J1-heap-base < red0-Red1': FWdelay-bisimulation-base
  final-expr0
  mred0 P
  final-expr1
  mred1' (compP1 P)
  convert-RA
  λt. bisim-red0-Red1
  bisim-wait01
  τMOVE0 P
  τMOVE1 (compP1 P)
  for P
.

context J0-J1-heap-base begin

lemma delay-bisimulation-red0-Red1:
```

assumes *wf*: *wf-J-prog P*
shows *delay-bisimulation-measure* (*mred0 P t*) (*mred1' (compP1 P) t*) *bisim-red0-Red1* (*ta-bisim* (λ*t*. *bisim-red0-Red1*)) (τ*MOVE0 P*) (τ*MOVE1* (*compP1 P*)) (λ*es es'*. *False*) (λ(((*e'*, *xs'*), *exs'*), *h'*) (((*e*, *xs*), *exs*), *h*). *countInitBlock e'*< *countInitBlock e*)
(**is** *delay-bisimulation-measure - - - - - - ?μ1 ?μ2*)
proof(*unfold-locales*)
fix *s1 s2 s1'*
assume *bisim-red0-Red1 s1 s2 red0-mthr.silent-move P t s1 s1'*
moreover obtain *ex1 exs1 h1* **where** *s1*: *s1* = ((*ex1*, *exs1*), *h1*) **by**(*cases s1*, *auto*)
moreover obtain *ex1' exs1' h1'* **where** *s1'*: *s1'* = ((*ex1'*, *exs1'*), *h1'*) **by**(*cases s1'*, *auto*)
moreover obtain *ex2 exs2 h2* **where** *s2*: *s2* = ((*ex2*, *exs2*), *h2*) **by**(*cases s2*, *auto*)
ultimately have *bisim*: *bisim-list1* (*ex1*, *exs1*) (*ex2*, *exs2*)
and *heap*: *h1* = *h2*
and *red*: *P*,*t* ⊢*0* ⟨*ex1*/*exs1*,*h1*⟩ −ε→ ⟨*ex1'*/*exs1'*,*h1'*⟩
and τ: τ*Move0 P h1* (*ex1*, *exs1*)
by(*auto simp add*: *bisim-red0-Red1-def red0-mthr.silent-move-iff*)
from *Red1-simulates-red0*[*OF wf red bisim*] τ
obtain *ex2'' exs2''* **where** *bisim'*: *bisim-list1* (*ex1'*, *exs1'*) (*ex2''*, *exs2''*)
and *red'*: τ*Red1't* (*compP1 P*) *t h1* (*ex2*, *exs2*) (*ex2''*, *exs2''*)
and [*simp*]: *h1'* = *h1* **by** *auto*
from τ*Red1't-into-Red1'-*τ*mthr-silent-movet*[*OF red'*] *bisim' heap*
have ∃ *s2'*. *Red1-mthr.silent-movet False* (*compP1 P*) *t s2 s2'* ∧ *bisim-red0-Red1 s1' s2'*
unfolding *s2 s1'* **by**(*auto simp add*: *bisim-red0-Red1-def*)
thus *bisim-red0-Red1 s1' s2* ∧ *?μ1*^++ *s1' s1* ∨ (∃ *s2'*. *Red1-mthr.silent-movet False* (*compP1 P*) *t s2 s2'* ∧ *bisim-red0-Red1 s1' s2'*) **..**
next
fix *s1 s2 s2'*
assume *bisim-red0-Red1 s1 s2 Red1-mthr.silent-move False* (*compP1 P*) *t s2 s2'*
moreover obtain *ex1 exs1 h1* **where** *s1*: *s1* = ((*ex1*, *exs1*), *h1*) **by**(*cases s1*, *auto*)
moreover obtain *ex2 exs2 h2* **where** *s2*: *s2* = ((*ex2*, *exs2*), *h2*) **by**(*cases s2*, *auto*)
moreover obtain *ex2' exs2' h2'* **where** *s2'*: *s2'* = ((*ex2'*, *exs2'*), *h2'*) **by**(*cases s2'*, *auto*)
ultimately have *bisim*: *bisim-list1* (*ex1*, *exs1*) (*ex2*, *exs2*)
and *heap*: *h1* = *h2*
and *red*: *False*,*compP1 P*,*t* ⊢*1* ⟨*ex2*/*exs2*,*h2*⟩ −ε→ ⟨*ex2'*/*exs2'*,*h2'*⟩
and τ: τ*Move1 P h2* (*ex2*, *exs2*)
by(*fastforce simp add*: *bisim-red0-Red1-def Red1-mthr.silent-move-iff*)+
from *red0-simulates-Red1*[*OF wf red bisim*] τ
obtain *e' es'* **where** *bisim'*: *bisim-list1* (*e'*, *es'*) (*ex2'*, *exs2'*)
and *red'*: τ*Red0t P t h2* (*ex1*, *exs1*) (*e'*, *es'*) ∨
countInitBlock (*fst ex2'*) < *countInitBlock* (*fst ex2*) ∧ *exs2'* = *exs2* ∧ *e'* = *ex1* ∧ *es'* = *exs1*
and [*simp*]: *h2'* = *h2* **by** *auto*
from *red'*
show *bisim-red0-Red1 s1 s2'* ∧ *?μ2*^++ *s2' s2* ∨ (∃ *s1'*. *red0-mthr.silent-movet P t s1 s1'* ∧ *bisim-red0-Red1 s1' s2'*)
(**is** *?refl* ∨ *?step*)
proof
assume τ*Red0t P t h2* (*ex1*, *exs1*) (*e'*, *es'*)
from τ*Red0t-into-red0-*τ*mthr-silent-movet*[*OF this*] *bisim' heap*
have *?step* **unfolding** *s1 s2'* **by**(*auto simp add*: *bisim-red0-Red1-def*)
thus *?thesis* **..**
next
assume *countInitBlock* (*fst ex2'*) < *countInitBlock* (*fst ex2*) ∧ *exs2'* = *exs2* ∧ *e'* = *ex1* ∧ *es'* = *exs1*

```
    hence ?refl using bisim' heap unfolding s1 s2' s2
      by (auto simp add: bisim-red0-Red1-def split-beta)
    thus ?thesis ..
  qed
next
  fix s1 s2 ta1 s1'
  assume bisim-red0-Red1 s1 s2 and mred0 P t s1 ta1 s1' and τ: ¬ τMOVE0 P s1 ta1 s1'
  moreover obtain ex1 exs1 h1 where s1: s1 = ((ex1, exs1), h1) by(cases s1, auto)
  moreover obtain ex1' exs1' h1' where s1': s1' = ((ex1', exs1'), h1') by(cases s1', auto)
  moreover obtain ex2 exs2 h2 where s2: s2 = ((ex2, exs2), h2) by(cases s2, auto)
  ultimately have heap: h2 = h1
    and bisim: bisim-list1 (ex1, exs1) (ex2, exs2)
    and red: P,t ⊢0 ⟨ex1/exs1,h1⟩ −ta1→ ⟨ex1'/exs1',h1'⟩
    by(auto simp add: bisim-red0-Red1-def)
  from τ have ¬ τMove0 P h1 (ex1, exs1) unfolding s1
    using red by(auto elim!: red0.cases dest: red-τ-taD[where extTA=extTA2J0 P, OF extTA2J0-ε])
  with Red1-simulates-red0[OF wf red bisim]
  obtain ex2'' exs2'' ex2' exs2' ta'
    where bisim': bisim-list1 (ex1', exs1') (ex2'', exs2'')
    and red': τRed1'r (compP1 P) t h1 (ex2, exs2) (ex2', exs2')
    and red'': False,compP1 P,t ⊢1 ⟨ex2'/exs2',h1⟩ −ta'→ ⟨ex2''/exs2'',h1'⟩
    and τ': ¬ τMove1 P h1 (ex2', exs2')
    and ta: ta-bisim01 ta1 ta' by fastforce
  from red'' have mred1' (compP1 P) t ((ex2', exs2'), h1) ta' ((ex2'', exs2''), h1') by auto
  moreover from τ' have ¬ τMOVE1 (compP1 P) ((ex2', exs2'), h1) ta' ((ex2'', exs2''), h1') by
simp
  moreover from red' have Red1-mthr.silent-moves False (compP1 P) t s2 ((ex2', exs2'), h1)
    unfolding s2 heap by(rule τRed1'r-into-Red1'-τmthr-silent-moves)
  moreover from bisim' have bisim-red0-Red1 ((ex1', exs1'), h1') ((ex2'', exs2''), h1')
    by(auto simp add: bisim-red0-Red1-def)
  ultimately
  show ∃ s2' s2'' ta2. Red1-mthr.silent-moves False (compP1 P) t s2 s2' ∧ mred1' (compP1 P) t s2'
ta2 s2'' ∧
            ¬ τMOVE1 (compP1 P) s2' ta2 s2'' ∧ bisim-red0-Red1 s1' s2'' ∧ ta-bisim01 ta1 ta2
    using ta unfolding s1' by blast
next
  fix s1 s2 ta2 s2'
  assume bisim-red0-Red1 s1 s2 and mred1' (compP1 P) t s2 ta2 s2' and τ: ¬ τMOVE1 (compP1
P) s2 ta2 s2'
  moreover obtain ex1 exs1 h1 where s1: s1 = ((ex1, exs1), h1) by(cases s1, auto)
  moreover obtain ex2 exs2 h2 where s2: s2 = ((ex2, exs2), h2) by(cases s2, auto)
  moreover obtain ex2' exs2' h2' where s2': s2' = ((ex2', exs2'), h2') by(cases s2', auto)
  ultimately have heap: h2 = h1
    and bisim: bisim-list1 (ex1, exs1) (ex2, exs2)
    and red: False,compP1 P,t ⊢1 ⟨ex2/exs2,h1⟩ −ta2→ ⟨ex2'/exs2',h2'⟩
    by(auto simp add: bisim-red0-Red1-def)
  from τ heap have ¬ τMove1 P h2 (ex2, exs2) unfolding s2
    using red by(fastforce elim!: Red1.cases dest: red1-τ-taD)
  with red0-simulates-Red1[OF wf red bisim]
  obtain e' es' e'' es'' ta'
    where bisim': bisim-list1 (e', es') (ex2', exs2')
    and red': τRed0r P t h1 (ex1, exs1) (e'', es'')
    and red'': P,t ⊢0 ⟨e''/es'',h1⟩ −ta'→ ⟨e'/es',h2'⟩
    and τ': ¬ τMove0 P h1 (e'', es'')
```

and *ta*: *ta-bisim01 ta′ ta2* **using** *heap* **by** *fastforce*
from *red″* **have** *mred0 P t* ((*e″*, *es″*), *h1*) *ta′* ((*e′*, *es′*), *h2′*) **by** *auto*
moreover from *red′* **have** *red0-mthr.silent-moves P t* ((*ex1*, *exs1*), *h1*) ((*e″*, *es″*), *h1*)
by(*rule τRed0r-into-red0-τmthr-silent-moves*)
moreover from *τ′* **have** ¬ *τMOVE0 P* ((*e″*, *es″*), *h1*) *ta′* ((*e′*, *es′*), *h2′*) **by** *simp*
moreover from *bisim′* **have** *bisim-red0-Red1* ((*e′*, *es′*), *h2′*) ((*ex2′*, *exs2′*), *h2′*)
by(*auto simp add*: *bisim-red0-Red1-def*)
ultimately
show ∃ *s1′ s1″ ta1*. *red0-mthr.silent-moves P t s1 s1′* ∧
mred0 P t s1′ ta1 s1″ ∧ ¬ *τMOVE0 P s1′ ta1 s1″* ∧
bisim-red0-Red1 s1″ s2′ ∧ *ta-bisim01 ta1 ta2*
using *ta* **unfolding** *s1 s2′* **by** *blast*
next
show *wfP ?μ1* **by** *auto*
next
have *wf* (*measure countInitBlock*) **..**
hence *wf* (*inv-image* (*measure countInitBlock*) (λ(((*e′*, *xs′*), *exs′*), *h′*). *e′*)) **..**
also have *inv-image* (*measure countInitBlock*) (λ(((*e′*, *xs′*), *exs′*), *h′*). *e′*) = {(*s2′*, *s2*). *?μ2 s2′ s2*}
by(*simp add*: *inv-image-def split-beta*)
finally show *wfP ?μ2* **by**(*simp add*: *wfp-def*)
qed

lemma *delay-bisimulation-diverge-red0-Red1*:
assumes *wf-J-prog P*
shows *delay-bisimulation-diverge* (*mred0 P t*) (*mred1′* (*compP1 P*) *t*) *bisim-red0-Red1* (*ta-bisim* (λ*t*. *bisim-red0-Red1*)) (*τMOVE0 P*) (*τMOVE1* (*compP1 P*))
proof −
interpret *delay-bisimulation-measure*
mred0 P t mred1′ (*compP1 P*) *t bisim-red0-Red1 ta-bisim* (λ*t*. *bisim-red0-Red1*) *τMOVE0 P τMOVE1* (*compP1 P*)
λ*es es′*. *False* λ(((*e′*, *xs′*), *exs′*), *h′*) (((*e*, *xs*), *exs*), *h*). *countInitBlock e′*< *countInitBlock e*
using *assms* **by**(*rule delay-bisimulation-red0-Red1*)
show *?thesis* **by** *unfold-locales*
qed

lemma *red0-Red1′-FWweak-bisim*:
assumes *wf*: *wf-J-prog P*
shows *FWdelay-bisimulation-diverge final-expr0* (*mred0 P*) *final-expr1* (*mred1′* (*compP1 P*))
(λ*t*. *bisim-red0-Red1*) *bisim-wait01* (*τMOVE0 P*) (*τMOVE1* (*compP1 P*))
proof −
interpret *delay-bisimulation-diverge*
mred0 P t
mred1′ (*compP1 P*) *t*
bisim-red0-Red1
ta-bisim (λ*t*. *bisim-red0-Red1*) *τMOVE0 P τMOVE1* (*compP1 P*)
for *t*
using *wf* **by**(*rule delay-bisimulation-diverge-red0-Red1*)
show *?thesis*
proof
fix *t* **and** *s1* **and** *s2* :: (('*addr expr1* × '*addr locals1*) × ('*addr expr1* × '*addr locals1*) *list*) × '*heap*
assume *bisim-red0-Red1 s1 s2* (λ(*x1*, *m*). *final-expr0 x1*) *s1*
moreover hence (λ(*x2*, *m*). *final-expr1 x2*) *s2*
by(*cases s1*)(*cases s2*,*auto simp add*: *bisim-red0-Red1-def final-iff elim*!: *bisim-list1-elim elim*: *bisim-list.cases*)

ultimately show $\exists s2'.$ *Red1-mthr.silent-moves False* (*compP1 P*) *t s2 s2'* $\wedge$ *bisim-red0-Red1 s1 s2'* $\wedge$ (λ(*x2*, *m*). *final-expr1 x2*) *s2'*
by *blast*
next
fix *t s1* **and** *s2* :: ((*'addr expr1* $\times$ *'addr locals1*) $\times$ (*'addr expr1* $\times$ *'addr locals1*) *list*) $\times$ *'heap*
assume *bisim-red0-Red1 s1 s2* (λ(*x2*, *m*). *final-expr1 x2*) *s2*
moreover hence (λ(*x1*, *m*). *final-expr0 x1*) *s1*
by(*cases s1*)(*cases s2*,*auto simp add*: *bisim-red0-Red1-def final-iff elim*!: *bisim-list1-elim elim*: *bisim-list.cases*)
ultimately show $\exists s1'.$ *red0-mthr.silent-moves P t s1 s1'* $\wedge$ *bisim-red0-Red1 s1' s2* $\wedge$ (λ(*x1*, *m*). *final-expr0 x1*) *s1'*
by *blast*
next
fix *t' x m1 x' m2 t x1 x2 x1' ta1 x1'' m1' x2' ta2 x2'' m2'*
assume *b*: *bisim-red0-Red1* (*x*, *m1*) (*x'*, *m2*)
and *bo*: *bisim-red0-Red1* (*x1*, *m1*) (*x2*, *m2*)
and τ*red1*: *red0-mthr.silent-moves P t* (*x1*, *m1*) (*x1'*, *m1*)
and *red1*: *mred0 P t* (*x1'*, *m1*) *ta1* (*x1''*, *m1'*)
and τ*1*: $\neg$ τ*MOVE0 P* (*x1'*, *m1*) *ta1* (*x1''*, *m1'*)
and τ*red2*: *Red1-mthr.silent-moves False* (*compP1 P*) *t* (*x2*, *m2*) (*x2'*, *m2*)
and *red2*: *mred1'* (*compP1 P*) *t* (*x2'*, *m2*) *ta2* (*x2''*, *m2'*)
and *bo'*: *bisim-red0-Red1* (*x1''*, *m1'*) (*x2''*, *m2'*)
and *tb*: *ta-bisim* (λ*t*. *bisim-red0-Red1*) *ta1 ta2*
from *b* **have** *m1* = *m2* **by**(*auto simp add*: *bisim-red0-Red1-def split-beta*)
moreover from *bo'* **have** *m1'* = *m2'* **by**(*auto simp add*: *bisim-red0-Red1-def split-beta*)
ultimately show *bisim-red0-Red1* (*x*, *m1'*) (*x'*, *m2'*) **using** *b*
by(*auto simp add*: *bisim-red0-Red1-def split-beta*)
next
fix *t x1 m1 x2 m2 x1' ta1 x1'' m1' x2' ta2 x2'' m2' w*
assume *bisim-red0-Red1* (*x1*, *m1*) (*x2*, *m2*)
and *red0-mthr.silent-moves P t* (*x1*, *m1*) (*x1'*, *m1*)
and *red0*: *mred0 P t* (*x1'*, *m1*) *ta1* (*x1''*, *m1'*)
and $\neg$ τ*MOVE0 P* (*x1'*, *m1*) *ta1* (*x1''*, *m1'*)
and *Red1-mthr.silent-moves False* (*compP1 P*) *t* (*x2*, *m2*) (*x2'*, *m2*)
and *red1*: *mred1'* (*compP1 P*) *t* (*x2'*, *m2*) *ta2* (*x2''*, *m2'*)
and $\neg$ τ*MOVE1* (*compP1 P*) (*x2'*, *m2*) *ta2* (*x2''*, *m2'*)
and *bisim-red0-Red1* (*x1''*, *m1'*) (*x2''*, *m2'*)
and *ta-bisim01 ta1 ta2*
and *Suspend*: *Suspend w* $\in$ *set* $\{\!|$*ta1*$|\!\}_w$ *Suspend w* $\in$ *set* $\{\!|$*ta2*$|\!\}_w$
from *red0 red1 Suspend* **show** *bisim-wait01 x1'' x2''*
by(*cases x2'*)(*cases x2''*, *auto dest*: *Red-Suspend-is-call Red1-Suspend-is-call simp add*: *split-beta bisim-wait01-def is-call-def*)
next
fix *t x1 m1 x2 m2 ta1 x1' m1'*
assume *bisim-red0-Red1* (*x1*, *m1*) (*x2*, *m2*)
and *bisim-wait01 x1 x2*
and *mred0 P t* (*x1*, *m1*) *ta1* (*x1'*, *m1'*)
and *wakeup*: *Notified* $\in$ *set* $\{\!|$*ta1*$|\!\}_w$ $\vee$ *WokenUp* $\in$ *set* $\{\!|$*ta1*$|\!\}_w$
moreover obtain *e0 es0* **where** [*simp*]: *x1* = (*e0*, *es0*) **by**(*cases x1*)
moreover obtain *e0' es0'* **where** [*simp*]: *x1'* = (*e0'*, *es0'*) **by**(*cases x1'*)
moreover obtain *e1 xs1 exs1* **where** [*simp*]: *x2* = ((*e1*, *xs1*), *exs1*) **by**(*cases x2*) *auto*
ultimately have *bisim*: *bisim-list1* (*e0*, *es0*) ((*e1*, *xs1*), *exs1*)
and *m1*: *m2* = *m1*
and *call*: *call e0* $\neq$ *None call1 e1* $\neq$ *None*

```
      and red: P,t ⊢0 ⟨e0/es0, m1⟩ −ta1→ ⟨e0'/es0', m1'⟩
      by(auto simp add: bisim-wait01-def bisim-red0-Red1-def)
    from red wakeup have ¬ τMove0 P m1 (e0, es0)
      by(auto elim!: red0.cases dest: red-τ-taD[where extTA=extTA2J0 P, simplified])
    with Red1-simulates-red0[OF wf red bisim] call m1
    show ∃ta2 x2' m2'. mred1' (compP1 P) t (x2, m2) ta2 (x2', m2') ∧ bisim-red0-Red1 (x1', m1')
(x2', m2') ∧ ta-bisim01 ta1 ta2
      by(auto simp add: bisim-red0-Red1-def)
  next
    fix t x1 m1 x2 m2 ta2 x2' m2'
    assume bisim-red0-Red1 (x1, m1) (x2, m2)
      and bisim-wait01 x1 x2
      and mred1' (compP1 P) t (x2, m2) ta2 (x2', m2')
      and wakeup: Notified ∈ set ⦃ta2⦄w ∨ WokenUp ∈ set ⦃ta2⦄w
    moreover obtain e0 es0 where [simp]: x1 = (e0, es0) by(cases x1)
    moreover obtain e1 xs1 exs1 where [simp]: x2 = ((e1, xs1), exs1) by(cases x2) auto
    moreover obtain e1' xs1' exs1' where [simp]: x2' = ((e1', xs1'), exs1') by(cases x2') auto
    ultimately have bisim: bisim-list1 (e0, es0) ((e1, xs1), exs1)
      and m1: m2 = m1
      and call: call e0 ≠ None call1 e1 ≠ None
      and red: False,compP1 P,t ⊢1 ⟨(e1, xs1)/exs1, m1⟩ −ta2→ ⟨(e1', xs1')/exs1', m2'⟩
      by(auto simp add: bisim-wait01-def bisim-red0-Red1-def)
    from red wakeup have ¬ τMove1 P m1 ((e1, xs1), exs1)
      by(auto elim!: Red1.cases dest: red1-τ-taD)
    with red0-simulates-Red1[OF wf red bisim] m1 call
    show ∃ta1 x1' m1'. mred0 P t (x1, m1) ta1 (x1', m1') ∧ bisim-red0-Red1 (x1', m1') (x2', m2')
∧ ta-bisim01 ta1 ta2
      by(auto simp add: bisim-red0-Red1-def)
  next
    show (∃x. final-expr0 x) ⟷ (∃x. final-expr1 x)
      by(auto simp add: split-paired-Ex final-iff)
  qed
qed

lemma bisim-J0-J1-start:
  assumes wf: wf-J-prog P
  and start: wf-start-state P C M vs
  shows red0-Red1'.mbisim (J0-start-state P C M vs) (J1-start-state (compP1 P) C M vs)
proof −
  from start obtain Ts T pns body D
    where sees: P ⊢ C sees M:Ts→T=⌊(pns,body)⌋ in D
    and conf: P,start-heap ⊢ vs [:≤] Ts
    by cases auto
  from conf have vs: length vs = length Ts by(rule list-all2-lengthD)
  from sees-wf-mdecl[OF wf-prog-wwf-prog[OF wf] sees]
  have fvbody: fv body ⊆ {this} ∪ set pns and len: length pns = length Ts
    by(auto simp add: wf-mdecl-def)
  with vs have fv: fv (blocks pns Ts vs body) ⊆ {this} by auto
  have wfCM: wf-J-mdecl P D (M, Ts, T, pns, body)
    using sees-wf-mdecl[OF wf sees] by(auto simp add: wf-mdecl-def)
  then obtain T' where wtbody: P,[this # pns [↦] Class D # Ts] ⊢ body :: T' by auto
  hence elbody: expr-locks body = (λl. 0) by(rule WT-expr-locks)
  hence cisbody: ¬ contains-insync body by(auto simp add: contains-insync-conv)
  from wfCM len vs have dabody: 𝒟 (blocks pns Ts vs body) ⌊{this}⌋ by auto
```

from *sees* **have** *sees1*: *compP1 P* ⊢ *C sees M*:*Ts*→*T* = ⌊*compE1* (*this* # *pns*) *body*⌋ *in D*
by(*auto dest*: *sees-method-compP*[**where** *f*=(λ*C M Ts T* (*pns*, *body*). *compE1* (*this* # *pns*) *body*)])

let *?e* = *blocks1 0* (*Class C*#*Ts*) (*compE1* (*this* # *pns*) *body*)
let *?xs* = *Null* # *vs* @ *replicate* (*max-vars body*) *undefined-value*
from *fvbody cisbody len vs*
have *bisim* [] (*blocks* (*this* # *pns*) (*Class D* # *Ts*) (*Null* # *vs*) *body*) (*blocks1* (*length* ([] :: *vname list*)) (*Class D* # *Ts*) (*compE1* (*this* # *pns*) *body*)) *?xs*
by−(*rule blocks-bisim*,(*fastforce simp add*: *nth-Cons*′ *nth-append*)+)
with *fv dabody len vs elbody sees sees1*
show *?thesis* **unfolding** *start-state-def*
by(*auto intro*!: *red0-Red1*′.*mbisimI split*: *if-split-asm*)(*auto simp add*: *bisim-red0-Red1-def blocks1-max-vars intro*!: *bisim-list.intros bisim-list1I wset-thread-okI*)
qed

end

end

7.25 Preservation of well-formedness from source code to intermediate language

theory *JJ1WellForm* **imports**
../J/JWellForm
J1WellForm
Compiler1
begin

The compiler preserves well-formedness. Is less trivial than it may appear. We start with two simple properties: preservation of well-typedness

lemma assumes *wf*: *wf-prog wfmd P*
shows *compE1-pres-wt*: ⟦ *P*,[*Vs*[↦]*Ts*] ⊢ *e* :: *U*; *size Ts* = *size Vs* ⟧ ⟹ *compP f P*,*Ts* ⊢*1 compE1 Vs e* :: *U*
and *compEs1-pres-wt*: ⟦ *P*,[*Vs*[↦]*Ts*] ⊢ *es* [::] *Us*; *size Ts* = *size Vs* ⟧ ⟹ *compP f P*,*Ts* ⊢*1 compEs1 Vs es* [::] *Us*
proof(*induct Vs e* **and** *Vs es arbitrary*: *Ts U* **and** *Ts Us rule*: *compE1-compEs1-induct*)
case *Var* **thus** *?case* **by**(*fastforce simp*:*map-upds-apply-eq-Some split*:*if-split-asm*)
next
case *LAss* **thus** *?case* **by**(*fastforce simp*:*map-upds-apply-eq-Some split*:*if-split-asm*)
next
case *Call* **thus** *?case*
by(*fastforce dest*: *sees-method-compP*[**where** *f* = *f*])
next
case *Block* **thus** *?case* **by**(*fastforce simp*:*nth-append*)
next
case (*Synchronized Vs V exp1 exp2 Ts U*)
note *IH1* = ‹⋀*Ts U*. ⟦*P*,[*Vs* [↦] *Ts*] ⊢ *exp1* :: *U*;
length Ts = *length Vs*⟧ ⟹ *compP f P*,*Ts* ⊢*1 compE1 Vs exp1* :: *U*›
note *IH2* = ‹⋀*Ts U*. ⟦*P*,[(*Vs* @ [*fresh-var Vs*]) [↦] *Ts*] ⊢ *exp2* :: *U*; *length Ts* = *length* (*Vs* @ [*fresh-var Vs*])⟧
⟹ *compP f P*,*Ts* ⊢*1 compE1* (*Vs* @ [*fresh-var Vs*]) *exp2* :: *U*›
note *length* = ‹*length Ts* = *length Vs*›
from ‹*P*,[*Vs* [↦] *Ts*] ⊢ *sync*$_V$ (*exp1*) *exp2* :: *U*›

obtain *U1* **where** *wt1*: *P*,[*Vs* [↦] *Ts*] ⊢ *exp1* :: *U1*
and *wt2*: *P*,[*Vs* [↦] *Ts*] ⊢ *exp2* :: *U*
and *U1*: *is-refT U1 U1* ≠ *NT*
by(*auto*)
from *IH1*[*of Ts U1*] *wt1 length*
have *wt1'*: *compP f P*,*Ts* ⊢*1 compE1 Vs exp1* :: *U1* **by** *simp*
from *length fresh-var-fresh*[*of Vs*] **have** [*Vs* [↦] *Ts*] ⊆*m* [*Vs* @ [*fresh-var Vs*] [↦] *Ts* @ [*Class Object*]]
by(*auto simp add*: *map-le-def fun-upd-def*)
with *wt2* **have** *P*,[*Vs*@[*fresh-var Vs*] [↦] *Ts* @ [*Class Object*]] ⊢ *exp2* :: *U*
by(*rule wt-env-mono*)
with *length IH2*[*of Ts* @ [*Class Object*] *U*]
have *compP f P*,*Ts* @ [*Class Object*] ⊢*1 compE1* (*Vs* @ [*fresh-var Vs*]) *exp2* :: *U* **by** *simp*
with *wt1' U1* **show** *?case* **by**(*auto*)
next
case (*TryCatch Vs exp1 C V exp2*)
note *IH1* = ‹⋀*Ts U*. ⟦*P*,[*Vs* [↦] *Ts*] ⊢ *exp1* :: *U*; *length Ts* = *length Vs*⟧ ⟹ *compP f P*,*Ts* ⊢*1 compE1 Vs exp1* :: *U*›
note *IH2* = ‹⋀*Ts U*. ⟦*P*,[(*Vs* @ [*V*]) [↦] *Ts*] ⊢ *exp2* :: *U*; *length Ts* = *length* (*Vs* @ [*V*])⟧ ⟹ *compP f P*,*Ts* ⊢*1 compE1* (*Vs* @ [*V*]) *exp2* :: *U*›
note *length* = ‹*length Ts* = *length Vs*›
with ‹*P*,[*Vs* [↦] *Ts*] ⊢ *try exp1 catch*(*C V*) *exp2* :: *U*›
have *wt1*: *P*,[*Vs* [↦] *Ts*] ⊢ *exp1* :: *U* **and** *wt2*: *P*,[(*Vs*@[*V*]) [↦] (*Ts*@[*Class C*])] ⊢ *exp2* :: *U*
and *C*: *P* ⊢ *C* ⪯* *Throwable* **by**(*auto simp add*: *nth-append*)
from *wf C* **have** *is-class P C* **by**(*rule is-class-sub-Throwable*)
with *IH1*[*OF wt1 length*] *IH2*[*OF wt2*] *length* **show** *?case* **by**(*auto*)
qed(*fastforce*)+

and the correct block numbering:

The main complication is preservation of definite assignment $\mathcal{D}$.

lemma fixes *e* :: *'addr expr* **and** *es* :: *'addr expr list*
shows *A-compE1-None*[*simp*]: $\mathcal{A}$ *e* = *None* ⟹ $\mathcal{A}$ (*compE1 Vs e*) = *None*
and *As-compEs1-None*: $\mathcal{A}s$ *es* = *None* ⟹ $\mathcal{A}s$ (*compEs1 Vs es*) = *None*
apply(*induct Vs e* **and** *Vs es rule*: *compE1-compEs1-induct*)
apply(*auto simp*:*hyperset-defs*)
done

lemma fixes *e* :: *'addr expr* **and** *es* :: *'addr expr list*
shows *A-compE1*: ⟦ $\mathcal{A}$ *e* = ⌊*A*⌋; *fv e* ⊆ *set Vs* ⟧ ⟹ $\mathcal{A}$ (*compE1 Vs e*) = ⌊*index Vs* ‘ *A*⌋
and *As-compEs1*: ⟦ $\mathcal{A}s$ *es* = ⌊*A*⌋; *fvs es* ⊆ *set Vs* ⟧ ⟹ $\mathcal{A}s$ (*compEs1 Vs es*) = ⌊*index Vs* ‘ *A*⌋
proof(*induct Vs e* **and** *Vs es arbitrary*: *A* **and** *A rule*: *compE1-compEs1-induct*)
case (*Block Vs V' T vo e*)
hence *fv e* ⊆ *set* (*Vs*@[*V'*]) **by** *fastforce*
moreover obtain *B* **where** $\mathcal{A}$ *e* = ⌊*B*⌋
using *Block.prems* **by**(*simp add*: *hyperset-defs*)
moreover from *calculation* **have** *B* ⊆ *set* (*Vs*@[*V'*]) **by**(*auto dest*!:*A-fv*)
ultimately show *?case* **using** *Block*
by(*auto simp add*: *hyperset-defs image-index*)
next
case (*Synchronized Vs V exp1 exp2 A*)
have *IH1*: ⋀*A*. ⟦$\mathcal{A}$ *exp1* = ⌊*A*⌋; *fv exp1* ⊆ *set Vs*⟧ ⟹ $\mathcal{A}$ (*compE1 Vs exp1*) = ⌊*index Vs* ‘ *A*⌋ **by** *fact*
have *IH2*: ⋀*A*. ⟦$\mathcal{A}$ *exp2* = ⌊*A*⌋; *fv exp2* ⊆ *set* (*Vs* @ [*fresh-var Vs*])⟧ ⟹ $\mathcal{A}$ (*compE1* (*Vs* @ [*fresh-var Vs*]) *exp2*) = ⌊*index* (*Vs* @ [*fresh-var Vs*]) ‘ *A*⌋ **by** *fact*
from ‹*fv* (*sync*$_V$ (*exp1*) *exp2*) ⊆ *set Vs*›

```
  have fv1: fv exp1 ⊆ set Vs
    and fv2: fv exp2 ⊆ set Vs by auto
  from ‹A (sync V (exp1) exp2) = ⌊A⌋› have A: A exp1 ⊔ A exp2 = ⌊A⌋ by(simp)
   then obtain A1 A2 where A1: A exp1 = ⌊A1⌋ and A2: A exp2 = ⌊A2⌋ by(auto simp add:
hyperset-defs)
  from A2 fv2 have A2 ⊆ set Vs by(auto dest: A-fv del: subsetI)
  with fresh-var-fresh[of Vs] have (fresh-var Vs) ∉ A2 by(auto)
  from fv2 have fv exp2 ⊆ set (Vs @ [fresh-var Vs]) by auto
  with IH2[OF A2] have A (compE1 (Vs @ [fresh-var Vs]) exp2) = ⌊index (Vs @ [fresh-var Vs]) ‘
A2⌋ by(auto)
  with IH1[OF A1 fv1] A[symmetric] ‹A2 ⊆ set Vs› ‹(fresh-var Vs) ∉ A2› A1 A2 show ?case
    by(auto simp add: image-index)
next
  case (InSynchronized Vs V a exp A)
  have IH: ⋀A. ⟦A exp = ⌊A⌋; fv exp ⊆ set (Vs @ [fresh-var Vs])⟧ ⟹ A (compE1 (Vs @ [fresh-var
Vs]) exp) = ⌊index (Vs @ [fresh-var Vs]) ‘ A⌋ by fact
  from ‹A (insync V (a) exp) = ⌊A⌋› have A: A exp = ⌊A⌋ by simp
  from ‹fv (insync V (a) exp) ⊆ set Vs› have fv: fv exp ⊆ set Vs by simp
  from A fv have A ⊆ set Vs by(auto dest: A-fv del: subsetI)
  with fresh-var-fresh[of Vs] have (fresh-var Vs) ∉ A by(auto)
  from fv IH[OF A] have  A (compE1 (Vs @ [fresh-var Vs]) exp) = ⌊index (Vs @ [fresh-var Vs]) ‘
A⌋ by simp
  with ‹A ⊆ set Vs› ‹(fresh-var Vs) ∉ A› show ?case by(simp add: image-index)
next
  case (TryCatch Vs e1 C V' e2)
  hence fve2: fv e2 ⊆ set (Vs@[V']) by auto
  show ?case
  proof (cases A e1)
    assume A1: A e1 = None
    then obtain A2 where A2: A e2 = ⌊A2⌋ using TryCatch
      by(simp add:hyperset-defs)
    hence A2 ⊆ set (Vs@[V']) using TryCatch.prems A-fv[OF A2] by simp blast
    thus ?thesis using TryCatch fve2 A1 A2
      by(auto simp add:hyperset-defs image-index)
  next
    fix A1 assume A1: A e1 =  ⌊A1⌋
    show ?thesis
    proof (cases  A e2)
      assume A2: A e2 = None
      then show ?case using TryCatch A1 by(simp add:hyperset-defs)
    next
      fix A2 assume A2: A e2 = ⌊A2⌋
      have A1 ⊆ set Vs using TryCatch.prems A-fv[OF A1] by simp blast
      moreover
      have A2 ⊆ set (Vs@[V']) using TryCatch.prems A-fv[OF A2] by simp blast
      ultimately show ?thesis using TryCatch A1 A2
        by(fastforce simp add: hyperset-defs image-index
          Diff-subset-conv inj-on-image-Int[OF inj-on-index])
    qed
  qed
next
  case (Cond Vs e e1 e2)
  { assume A e = None ∨ A e1 = None ∨ A e2 = None
    hence ?case using Cond by(auto simp add:hyperset-defs image-Un)
```

```
  }
  moreover
  { fix A A1 A2
    assume 𝒜 e = ⌊A⌋ and A1: 𝒜 e1 = ⌊A1⌋ and A2: 𝒜 e2 = ⌊A2⌋
    moreover
    have A1 ⊆ set Vs using Cond.prems A-fv[OF A1] by simp blast
    moreover
    have A2 ⊆ set Vs using Cond.prems A-fv[OF A2] by simp blast
    ultimately have ?case using Cond
      by(auto simp add:hyperset-defs image-Un
          inj-on-image-Int[OF inj-on-index])
  }
  ultimately show ?case by fastforce
qed (auto simp add:hyperset-defs)

lemma fixes e :: ('a, 'b, 'addr) exp and es :: ('a, 'b, 'addr) exp list
  shows D-None [iff]: 𝒟 e None
  and Ds-None [iff]: 𝒟s es None
by(induct e and es rule: 𝒟.induct 𝒟s.induct)(simp-all)

declare Un-ac [simp]

lemma fixes e :: 'addr expr and es :: 'addr expr list
  shows D-index-compE1: ⟦ A ⊆ set Vs; fv e ⊆ set Vs ⟧ ⟹ 𝒟 e ⌊A⌋ ⟹ 𝒟 (compE1 Vs e) ⌊index
Vs ‘ A⌋
  and Ds-index-compEs1: ⟦ A ⊆ set Vs; fvs es ⊆ set Vs ⟧ ⟹ 𝒟s es ⌊A⌋ ⟹ 𝒟s (compEs1 Vs es)
⌊index Vs ‘ A⌋
proof(induct e and es arbitrary: A Vs and A Vs rule: 𝒟.induct 𝒟s.induct)
  case (BinOp e1 bop e2)
  hence IH1: 𝒟 (compE1 Vs e1) ⌊index Vs ‘ A⌋ by simp
  show ?case
  proof (cases 𝒜 e1)
    case None thus ?thesis using BinOp by simp
  next
    case (Some A1)
    have indexA1: 𝒜 (compE1 Vs e1) = ⌊index Vs ‘ A1⌋
      using A-compE1[OF Some] BinOp.prems by auto
    have A ∪ A1 ⊆ set Vs using BinOp.prems A-fv[OF Some] by auto
    hence 𝒟 (compE1 Vs e2) ⌊index Vs ‘ (A1 ∪ A)⌋ using BinOp Some by(auto)
    hence 𝒟 (compE1 Vs e2) ⌊index Vs ‘ A1 ∪ index Vs ‘ A⌋
      by(simp add: image-Un)
    thus ?thesis using IH1 indexA1 by auto
  qed
next
  case (AAcc a i A Vs)
  have IH1: ⋀A Vs. ⟦A ⊆ set Vs; fv a ⊆ set Vs; 𝒟 a ⌊A⌋⟧ ⟹ 𝒟 (compE1 Vs a) ⌊index Vs ‘ A⌋ by
fact
  have IH2: ⋀A Vs. ⟦A ⊆ set Vs; fv i ⊆ set Vs; 𝒟 i ⌊A⌋⟧ ⟹ 𝒟 (compE1 Vs i) ⌊index Vs ‘ A⌋ by
fact
  from ‹𝒟 (a⌊i⌉) ⌊A⌋› have D1: 𝒟 a ⌊A⌋ and D2: 𝒟 i (⌊A⌋ ⊔ 𝒜 a) by auto
  from ‹fv (a⌊i⌉) ⊆ set Vs› have fv1: fv a ⊆ set Vs and fv2: fv i ⊆ set Vs by auto
  show ?case
  proof(cases 𝒜 a)
    case None thus ?thesis using AAcc by simp
```

```
  next
    case (Some A1)
    with fv1 have 𝒜 (compE1 Vs a) = ⌊index Vs ‘ A1⌋ by−(rule A-compE1)
    moreover from A-fv[OF Some] fv1 ‹A ⊆ set Vs› have A1 ∪ A ⊆ set Vs by auto
    from IH2[OF this fv2] Some D2 have 𝒟 (compE1 Vs i) ⌊index Vs ‘ A1 ∪ index Vs ‘ A⌋
      by(simp add: image-Un)
    ultimately show ?thesis using IH1[OF ‹A ⊆ set Vs› fv1 D1] by(simp)
  qed
next
  case (AAss a i e A Vs)
  have IH1: ⋀A Vs. ⟦A ⊆ set Vs; fv a ⊆ set Vs; 𝒟 a ⌊A⌋⟧ ⟹ 𝒟 (compE1 Vs a) ⌊index Vs ‘ A⌋ by
fact
  have IH2: ⋀A Vs. ⟦A ⊆ set Vs; fv i ⊆ set Vs; 𝒟 i ⌊A⌋⟧ ⟹ 𝒟 (compE1 Vs i) ⌊index Vs ‘ A⌋ by
fact
  have IH3: ⋀A Vs. ⟦A ⊆ set Vs; fv e ⊆ set Vs; 𝒟 e ⌊A⌋⟧ ⟹ 𝒟 (compE1 Vs e) ⌊index Vs ‘ A⌋ by
fact
  from ‹𝒟 (a⌊i⌉:=e) ⌊A⌋› have D1: 𝒟 a ⌊A⌋ and D2: 𝒟 i (⌊A⌋ ⊔ 𝒜 a)
    and D3: 𝒟 e (⌊A⌋ ⊔ 𝒜 a ⊔ 𝒜 i) by auto
  from ‹fv (a⌊i⌉ := e) ⊆ set Vs›
  have fv1: fv a ⊆ set Vs and fv2: fv i ⊆ set Vs and fv3: fv e ⊆ set Vs by auto
  show ?case
  proof(cases 𝒜 a)
    case None thus ?thesis using AAss by simp
  next
    case (Some A1)
    with fv1 have A1: 𝒜 (compE1 Vs a) = ⌊index Vs ‘ A1⌋ by−(rule A-compE1)
    from A-fv[OF Some] fv1 ‹A ⊆ set Vs› have A1 ∪ A ⊆ set Vs by auto
    from IH2[OF this fv2] Some D2 have D2′: 𝒟 (compE1 Vs i) ⌊index Vs ‘ A1 ∪ index Vs ‘ A⌋
      by(simp add: image-Un)
    show ?thesis
    proof(cases 𝒜 i)
      case None thus ?thesis using AAss D2′ Some A1 by simp
    next
      case (Some A2)
      with fv2 have A2: 𝒜 (compE1 Vs i) = ⌊index Vs ‘ A2⌋ by−(rule A-compE1)
      moreover from A-fv[OF Some] fv2 ‹A1 ∪ A ⊆ set Vs› have A1 ∪ A ∪ A2 ⊆ set Vs by auto
      from IH3[OF this fv3] Some ‹𝒜 a = ⌊A1⌋› D3
      have 𝒟 (compE1 Vs e) ⌊index Vs ‘ A1 ∪ index Vs ‘ A ∪ index Vs ‘ A2⌋
        by(simp add: image-Un Un-commute Un-assoc)
      ultimately show ?thesis using IH1[OF ‹A ⊆ set Vs› fv1 D1] D2′ A1 A2 by(simp)
    qed
  qed
next
  case (FAss e1 F D e2)
  hence IH1: 𝒟 (compE1 Vs e1) ⌊index Vs ‘ A⌋ by simp
  show ?case
  proof (cases 𝒜 e1)
    case None thus ?thesis using FAss by simp
  next
    case (Some A1)
    have indexA1: 𝒜 (compE1 Vs e1) = ⌊index Vs ‘ A1⌋
      using A-compE1[OF Some] FAss.prems by auto
    have A ∪ A1 ⊆ set Vs using FAss.prems A-fv[OF Some] by auto
    hence 𝒟 (compE1 Vs e2) ⌊index Vs ‘ (A ∪ A1)⌋ using FAss Some by auto
```

```
    hence 𝒟 (compE1 Vs e2) ⌊index Vs ‘ A ∪ index Vs ‘ A1⌋
      by(simp add: image-Un)
    thus ?thesis using IH1 indexA1 by auto
  qed
next
  case (CompareAndSwap e1 D F e2 e3 A Vs)
  have IH1: ⋀A Vs. ⟦A ⊆ set Vs; fv e1 ⊆ set Vs; 𝒟 e1 ⌊A⌋⟧ ⟹ 𝒟 (compE1 Vs e1) ⌊index Vs ‘ A⌋
by fact
  have IH2: ⋀A Vs. ⟦A ⊆ set Vs; fv e2 ⊆ set Vs; 𝒟 e2 ⌊A⌋⟧ ⟹ 𝒟 (compE1 Vs e2) ⌊index Vs ‘ A⌋
by fact
  have IH3: ⋀A Vs. ⟦A ⊆ set Vs; fv e3 ⊆ set Vs; 𝒟 e3 ⌊A⌋⟧ ⟹ 𝒟 (compE1 Vs e3) ⌊index Vs ‘ A⌋
by fact
  from ‹𝒟 (e1•compareAndSwap(D•F, e2, e3)) ⌊A⌋› have D1: 𝒟 e1 ⌊A⌋ and D2: 𝒟 e2 (⌊A⌋ ⊔ 𝒜
e1)
    and D3: 𝒟 e3 (⌊A⌋ ⊔ 𝒜 e1 ⊔ 𝒜 e2) by auto
  from ‹fv (e1•compareAndSwap(D•F, e2, e3)) ⊆ set Vs›
  have fv1: fv e1 ⊆ set Vs and fv2: fv e2 ⊆ set Vs and fv3: fv e3 ⊆ set Vs by auto
  show ?case
  proof(cases 𝒜 e1)
    case None thus ?thesis using CompareAndSwap by simp
  next
    case (Some A1)
    with fv1 have A1: 𝒜 (compE1 Vs e1) = ⌊index Vs ‘ A1⌋ by−(rule A-compE1)
    from A-fv[OF Some] fv1 ‹A ⊆ set Vs› have A1 ∪ A ⊆ set Vs by auto
    from IH2[OF this fv2] Some D2 have D2′: 𝒟 (compE1 Vs e2) ⌊index Vs ‘ A1 ∪ index Vs ‘ A⌋
      by(simp add: image-Un)
    show ?thesis
    proof(cases 𝒜 e2)
      case None thus ?thesis using CompareAndSwap D2′ Some A1 by simp
    next
      case (Some A2)
      with fv2 have A2: 𝒜 (compE1 Vs e2) = ⌊index Vs ‘ A2⌋ by−(rule A-compE1)
      moreover from A-fv[OF Some] fv2 ‹A1 ∪ A ⊆ set Vs› have A1 ∪ A ∪ A2 ⊆ set Vs by auto
      from IH3[OF this fv3] Some ‹𝒜 e1 = ⌊A1⌋› D3
      have 𝒟 (compE1 Vs e3) ⌊index Vs ‘ A1 ∪ index Vs ‘ A ∪ index Vs ‘ A2⌋
        by(simp add: image-Un Un-commute Un-assoc)
      ultimately show ?thesis using IH1[OF ‹A ⊆ set Vs› fv1 D1] D2′ A1 A2 by(simp)
    qed
  qed
next
  case (Call e1 M es)
  hence IH1: 𝒟 (compE1 Vs e1) ⌊index Vs ‘ A⌋ by simp
  show ?case
  proof (cases 𝒜 e1)
    case None thus ?thesis using Call by simp
  next
    case (Some A1)
    have indexA1: 𝒜 (compE1 Vs e1) = ⌊index Vs ‘ A1⌋
      using A-compE1[OF Some] Call.prems by auto
    have A ∪ A1 ⊆ set Vs using Call.prems A-fv[OF Some] by auto
    hence 𝒟s (compEs1 Vs es) ⌊index Vs ‘ (A ∪ A1)⌋ using Call Some by auto
    hence 𝒟s (compEs1 Vs es) ⌊index Vs ‘ A ∪ index Vs ‘ A1⌋
      by(simp add: image-Un)
    thus ?thesis using IH1 indexA1 by auto
```

```
  qed
next
  case (Synchronized V exp1 exp2 A Vs)
  have IH1: ⋀A Vs. ⟦A ⊆ set Vs; fv exp1 ⊆ set Vs; 𝒟 exp1 ⌊A⌋⟧ ⟹ 𝒟 (compE1 Vs exp1) ⌊index Vs
‘ A⌋ by fact
  have IH2: ⋀A Vs. ⟦A ⊆ set Vs; fv exp2 ⊆ set Vs; 𝒟 exp2 ⌊A⌋⟧ ⟹ 𝒟 (compE1 Vs exp2) ⌊index Vs
‘ A⌋ by fact
  from ‹𝒟 (sync V (exp1) exp2) ⌊A⌋› have D1: 𝒟 exp1 ⌊A⌋ and D2: 𝒟 exp2 (⌊A⌋ ⊔ 𝒜 exp1) by
auto
  from ‹fv (sync V (exp1) exp2) ⊆ set Vs› have fv1: fv exp1 ⊆ set Vs and fv2: fv exp2 ⊆ set Vs by
auto
  show ?case
  proof(cases 𝒜 exp1)
    case None thus ?thesis using Synchronized by(simp)
  next
    case (Some A1)
    with fv1 have A1: 𝒜 (compE1 Vs exp1) = ⌊index Vs ‘ A1⌋ by−(rule A-compE1)
    from A-fv[OF Some] fv1 ‹A ⊆ set Vs› have A ∪ A1 ⊆ set Vs by auto
    hence A ∪ A1 ⊆ set (Vs @ [fresh-var Vs]) by simp
    from IH2[OF this] fv2 Some D2
    have D2′: 𝒟 (compE1 (Vs @ [fresh-var Vs]) exp2) ⌊index (Vs @ [fresh-var Vs]) ‘ (A ∪ A1)⌋
      by(simp)
    moreover from fresh-var-fresh[of Vs] ‹A ∪ A1 ⊆ set Vs›
    have (fresh-var Vs) ∉ A ∪ A1 by auto
    with ‹A ∪ A1 ⊆ set Vs›
    have index (Vs @ [fresh-var Vs]) ‘ (A ∪ A1) = index Vs ‘ A ∪ index Vs ‘ A1
      by(simp add: image-index image-Un)
    ultimately show ?thesis using IH1[OF ‹A ⊆ set Vs› fv1 D1] D2′ A1 by(simp)
  qed
next
  case (InSynchronized V a e A Vs)
  have IH: ⋀A Vs. ⟦A ⊆ set Vs; fv e ⊆ set Vs; 𝒟 e ⌊A⌋⟧ ⟹ 𝒟 (compE1 Vs e) ⌊index Vs ‘ A⌋ by
fact
  from IH[of A Vs @ [fresh-var Vs]] ‹A ⊆ set Vs› ‹fv (insync V (a) e) ⊆ set Vs› ‹𝒟 (insync V (a) e)
⌊A⌋›
  have 𝒟 (compE1 (Vs @ [fresh-var Vs]) e) ⌊index (Vs @ [fresh-var Vs]) ‘ A⌋ by(auto)
  moreover from fresh-var-fresh[of Vs] ‹A ⊆ set Vs› have (fresh-var Vs) ∉ A by auto
  with ‹A ⊆ set Vs› have index (Vs @ [fresh-var Vs]) ‘ A = index Vs ‘ A by(simp add: image-index)
  ultimately show ?case by(simp)
next
  case (TryCatch e1 C V e2)
  have ⟦ A∪{V} ⊆ set(Vs@[V]); fv e2 ⊆ set(Vs@[V]); 𝒟 e2 ⌊A∪{V}⌋⟧ ⟹
       𝒟 (compE1 (Vs@[V]) e2) ⌊index (Vs@[V]) ‘ (A∪{V})⌋ by fact
  hence 𝒟 (compE1 (Vs@[V]) e2) ⌊index (Vs@[V]) ‘ (A∪{V})⌋
    using TryCatch.prems by(simp add:Diff-subset-conv)
  moreover have index (Vs@[V]) ‘ A ⊆ index Vs ‘ A ∪ {size Vs}
    using TryCatch.prems by(auto simp add: image-index split:if-split-asm)
  ultimately show ?case using TryCatch by(auto simp:hyperset-defs elim!:D-mono′)
next
  case (Seq e1 e2)
  hence IH1: 𝒟 (compE1 Vs e1) ⌊index Vs ‘ A⌋ by simp
  show ?case
  proof (cases 𝒜 e1)
    case None thus ?thesis using Seq by simp
```

```
  next
    case (Some A1)
    have indexA1: 𝒜 (compE1 Vs e1) = ⌊index Vs ‘ A1⌋
      using A-compE1[OF Some] Seq.prems by auto
    have A ∪ A1 ⊆ set Vs using Seq.prems A-fv[OF Some] by auto
    hence 𝒟 (compE1 Vs e2) ⌊index Vs ‘ (A ∪ A1)⌋ using Seq Some by auto
    hence 𝒟 (compE1 Vs e2) ⌊index Vs ‘ A ∪ index Vs ‘ A1⌋
      by(simp add: image-Un)
    thus ?thesis using IH1 indexA1 by auto
  qed
next
  case (Cond e e1 e2)
  hence IH1: 𝒟 (compE1 Vs e) ⌊index Vs ‘ A⌋ by simp
  show ?case
  proof (cases 𝒜 e)
    case None thus ?thesis using Cond by simp
  next
    case (Some B)
    have indexB: 𝒜 (compE1 Vs e) = ⌊index Vs ‘ B⌋
      using A-compE1[OF Some] Cond.prems by auto
    have A ∪ B ⊆ set Vs using Cond.prems A-fv[OF Some] by auto
    hence 𝒟 (compE1 Vs e1) ⌊index Vs ‘ (A ∪ B)⌋
      and 𝒟 (compE1 Vs e2) ⌊index Vs ‘ (A ∪ B)⌋
      using Cond Some by auto
    hence 𝒟 (compE1 Vs e1) ⌊index Vs ‘ A ∪ index Vs ‘ B⌋
      and 𝒟 (compE1 Vs e2) ⌊index Vs ‘ A ∪ index Vs ‘ B⌋
      by(simp add: image-Un)+
    thus ?thesis using IH1 indexB by auto
  qed
next
  case (While e1 e2)
  hence IH1: 𝒟 (compE1 Vs e1) ⌊index Vs ‘ A⌋ by simp
  show ?case
  proof (cases 𝒜 e1)
    case None thus ?thesis using While by simp
  next
    case (Some A1)
    have indexA1: 𝒜 (compE1 Vs e1) = ⌊index Vs ‘ A1⌋
      using A-compE1[OF Some] While.prems by auto
    have A ∪ A1 ⊆ set Vs using While.prems A-fv[OF Some] by auto
    hence 𝒟 (compE1 Vs e2) ⌊index Vs ‘ (A ∪ A1)⌋ using While Some by auto
    hence 𝒟 (compE1 Vs e2) ⌊index Vs ‘ A ∪ index Vs ‘ A1⌋
      by(simp add: image-Un)
    thus ?thesis using IH1 indexA1 by auto
  qed
next
  case (Block V T vo e A Vs)
  have IH: ⋀A Vs. ⟦A ⊆ set Vs; fv e ⊆ set Vs; 𝒟 e ⌊A⌋⟧ ⟹ 𝒟 (compE1 Vs e) ⌊index Vs ‘ A⌋ by
fact
  from ‹fv {V:T=vo; e} ⊆ set Vs› have fv: fv e ⊆ set (Vs @ [V]) by auto
  show ?case
  proof(cases vo)
    case None
    with ‹𝒟 {V:T=vo; e} ⌊A⌋› have D: 𝒟 e ⌊A − {V}⌋ by(auto)
```

```
    from ‹A ⊆ set Vs› have A − {V} ⊆ set (Vs @ [V]) by auto
    from IH[OF this fv D] have 𝒟 (compE1 (Vs @ [V]) e) ⌊index (Vs @ [V]) ‘ (A − {V})⌋ .
    moreover from ‹A ⊆ set Vs› have size: size Vs ∉ index Vs ‘ A by(auto simp add: image-def)
    hence ⌊index Vs ‘ (A − {V})⌋ ⊑ ⌊index Vs ‘ A⌋ by(auto simp add: hyperset-defs)
    ultimately have 𝒟 (compE1 (Vs @ [V]) e) ⌊index Vs ‘ A⌋ using ‹A − {V} ⊆ set (Vs @ [V])›
      by(simp add: image-index)(erule D-mono′, auto)
    thus ?thesis using None size by(simp)
  next
    case (Some v)
    with ‹𝒟 {V:T=vo; e} ⌊A⌋› have D: 𝒟 e ⌊insert V A⌋ by(auto)
    from ‹A ⊆ set Vs› have insert V A ⊆ set (Vs @ [V]) by auto
    from IH[OF this fv D] have 𝒟 (compE1 (Vs @ [V]) e) ⌊index (Vs @ [V]) ‘ insert V A⌋ by simp
    moreover from ‹A ⊆ set Vs› have index (Vs @ [V]) ‘ insert V A ⊆ insert (length Vs) (index Vs
‘ A)
      by(auto simp add: image-index)
    ultimately show ?thesis using Some by(auto elim!: D-mono′ simp add: hyperset-defs)
  qed
next
  case (Cons-exp e1 es)
  hence IH1: 𝒟 (compE1 Vs e1) ⌊index Vs ‘ A⌋ by simp
  show ?case
  proof (cases 𝒜 e1)
    case None thus ?thesis using Cons-exp by simp
  next
    case (Some A1)
    have indexA1: 𝒜 (compE1 Vs e1) = ⌊index Vs ‘ A1⌋
      using A-compE1[OF Some] Cons-exp.prems by auto
    have A ∪ A1 ⊆ set Vs using Cons-exp.prems A-fv[OF Some] by auto
    hence 𝒟s (compEs1 Vs es) ⌊index Vs ‘ (A ∪ A1)⌋ using Cons-exp Some by auto
    hence 𝒟s (compEs1 Vs es) ⌊index Vs ‘ A ∪ index Vs ‘ A1⌋
      by(simp add: image-Un)
    thus ?thesis using IH1 indexA1 by auto
  qed
qed (simp-all add:hyperset-defs)

declare Un-ac [simp del]

lemma index-image-set: distinct xs ⟹ index xs ‘ set xs = {..<size xs}
by(induct xs rule:rev-induct) (auto simp add: image-index)

lemma D-compE1:
  ⟦ 𝒟 e ⌊set Vs⌋; fv e ⊆ set Vs; distinct Vs ⟧ ⟹ 𝒟 (compE1 Vs e) ⌊{..<length Vs}⌋
by(fastforce dest!: D-index-compE1[OF subset-refl] simp add:index-image-set)

lemma D-compE1′:
  assumes 𝒟 e ⌊set(V#Vs)⌋ and fv e ⊆ set(V#Vs) and distinct(V#Vs)
  shows 𝒟 (compE1 (V#Vs) e) ⌊{..length Vs}⌋
proof −
  have {..size Vs} = {..<size(V#Vs)} by auto
  thus ?thesis using assms by (simp only:)(rule D-compE1)
qed

lemma compP1-pres-wf: wf-J-prog P ⟹ wf-J1-prog (compP1 P)
apply simp
```

apply(*rule wf-prog-compPI*)
prefer *2* **apply** *assumption*
apply(*case-tac m*)
apply(*simp add:wf-mdecl-def*)
apply(*clarify*)
apply(*frule WT-fv*)
apply(*fastforce intro*!: *compE1-pres-wt D-compE1'* $\mathcal{B}$ *syncvars-compE1*)
done

end
theory *Compiler* **imports** *Compiler1 Compiler2* **begin**

definition *J2JVM* :: *'addr J-prog* ⇒ *'addr jvm-prog*
where [*code del*]: *J2JVM* ≡ *compP2* ∘ *compP1*

lemma *J2JVM-code* [*code*]:
J2JVM = *compP* (λ*C M Ts T* (*pns*, *body*). *compMb2* (*compE1* (*this*#*pns*) *body*))
by(*simp add*: *J2JVM-def compP2-def o-def compP-compP split-def*)

end

7.26 Correctness of both stages

theory *Correctness*
imports
J0Bisim
J1Deadlock
../Framework/FWBisimDeadlock
Correctness2
Correctness1Threaded
Correctness1
JJ1WellForm
Compiler
begin

locale *J-JVM-heap-conf-base* =
J0-J1-heap-base
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
+
J1-JVM-heap-conf-base
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
hconf compP1 P
for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*

and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr J-prog*
begin

definition *bisimJ2JVM* ::
(*('addr,'thread-id,'addr expr×'addr locals,'heap,'addr) state*,
('addr,'thread-id,'addr option × 'addr frame list,'heap,'addr) state) *bisim*
where *bisimJ2JVM* = *red-red0.mbisim* ∘$_B$ *red0-Red1'.mbisim* ∘$_B$ *mbisim-Red1'-Red1* ∘$_B$ *Red1-execd.mbisim*

definition *tlsimJ2JVM* ::
(*'thread-id* × (*'addr*, *'thread-id*, *'heap*) *J-thread-action*,
'thread-id × (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action*) *bisim*
where *tlsimJ2JVM* = *red-red0.mta-bisim* ∘$_B$ *red0-Red1'.mta-bisim* ∘$_B$ (=) ∘$_B$ *Red1-execd.mta-bisim*

end

lemma *compP2-has-method* [*simp*]: *compP2 P* ⊢ *C has M* ⟷ *P* ⊢ *C has M*
by(*auto simp add*: *compP2-def compP-has-method*)

locale *J-JVM-conf-read* =
J1-JVM-conf-read
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
hconf compP1 P
for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr J-prog*
begin

sublocale *J-JVM-heap-conf-base* **by**(*unfold-locales*)

theorem *bisimJ2JVM-weak-bisim*:
assumes *wf*: *wf-J-prog P*
shows *delay-bisimulation-diverge-final* (*mredT P*) (*execd-mthr.redT* (*J2JVM P*)) *bisimJ2JVM tlsimJ2JVM*
(*red-mthr.m*τ*move P*) (*execd-mthr.m*τ*move* (*J2JVM P*)) *red-mthr.mfinal exec-mthr.mfinal*
unfolding *bisimJ2JVM-def tlsimJ2JVM-def J2JVM-def o-apply*
apply(*rule delay-bisimulation-diverge-final-compose*)
apply(*rule FWdelay-bisimulation-diverge.mthr-delay-bisimulation-diverge-final*)
apply(*rule red-red0-FWbisim*[*OF wf-prog-wwf-prog*[*OF wf*]])
apply(*rule delay-bisimulation-diverge-final-compose*)
apply(*rule FWdelay-bisimulation-diverge.mthr-delay-bisimulation-diverge-final*)
apply(*rule red0-Red1'-FWweak-bisim*[*OF wf*])
apply(*rule delay-bisimulation-diverge-final-compose*)
apply(*rule delay-bisimulation-diverge-final.intro*)

apply(*rule bisimulation-into-delay.delay-bisimulation*)
apply(*rule Red1'-Red1-bisim-into-weak*[*OF compP1-pres-wf*[*OF wf*]])
apply(*rule bisimulation-final.delay-bisimulation-final-base*)
apply(*rule Red1'-Red1-bisimulation-final*[*OF compP1-pres-wf*[*OF wf*]])
apply(*rule FWdelay-bisimulation-diverge.mthr-delay-bisimulation-diverge-final*)
apply(*rule Red1-exec1-FWwbisim*[*OF compP1-pres-wf*[*OF wf*]])
done

lemma *bisimJ2JVM-start*:
assumes *wf*: *wf-J-prog P*
and *start*: *wf-start-state P C M vs*
shows *bisimJ2JVM* (*J-start-state P C M vs*) (*JVM-start-state* (*J2JVM P*) *C M vs*)
using *assms*
unfolding *bisimJ2JVM-def J2JVM-def o-def*
apply(*intro bisim-composeI*)
apply(*erule* (*1*) *bisim-J-J0-start*[*OF wf-prog-wwf-prog*])
apply(*erule* (*1*) *bisim-J0-J1-start*)
apply(*erule bisim-J1-J1-start*[*OF compP1-pres-wf*])
apply *simp*
apply(*erule bisim-J1-JVM-start*[*OF compP1-pres-wf*])
apply *simp*
done

end

fun *exception* :: *'addr expr* × *'addr locals* ⇒ *'addr option* × *'addr frame list*
where *exception* (*Throw a*, *xs*) = (⌊*a*⌋, [])
| *exception* - = (*None*, [])

definition *mexception* ::
(*'addr*,*'thread-id*,*'addr expr*×*'addr locals*,*'heap*,*'addr*) *state* ⇒
(*'addr*,*'thread-id*,*'addr option*×*'addr frame list*,*'heap*,*'addr*) *state*
where
⋀*ln*. *mexception s* ≡
(*locks s*, (λ*t*. *case thr s t of* ⌊(*e*, *ln*)⌋ ⇒ ⌊(*exception e*, *ln*)⌋ | *None* ⇒ *None*, *shr s*), *wset s*, *interrupts s*)

declare *compP1-def* [*simp del*]

context *J-JVM-heap-conf-base* **begin**

lemma *bisimJ2JVM-mfinal-mexception*:
assumes *bisim*: *bisimJ2JVM s s'*
and *fin*: *exec-mthr.mfinal s'*
and *fin'*: *red-mthr.mfinal s*
and *tsNotEmpty*: *thr s t* ≠ *None*
shows *s'* = *mexception s*
proof −
obtain *ls ts m ws is* **where** *s*: *s* = (*ls*, (*ts*, *m*), *ws*, *is*) **by**(*cases s*) *fastforce*
from *bisim* **obtain** *s0 s1* **where** *bisimJ0*: *red-red0.mbisim s s0*
and *bisim01*: *red0-Red1'.mbisim s0 s1*
and *bisim1JVM*: *Red1-execd.mbisim s1 s'*
unfolding *bisimJ2JVM-def* **by**(*fastforce simp add*: *mbisim-Red1'-Red1-def*)

from *bisimJ0 s* **have** [*simp*]: *locks s0 = ls wset s0 = ws interrupts s0 = is*
and *tbisimJ0*: $\bigwedge t$. *red-red0.tbisim (ws t = None) t (ts t) m (thr s0 t) (shr s0)*
by(*auto simp add*: *red-red0.mbisim-def*)
from *bisim01* **have** [*simp*]: *locks s1 = ls wset s1 = ws interrupts s1 = is*
and *tbisim01*: $\bigwedge t$. *red0-Red1'.tbisim (ws t = None) t (thr s0 t) (shr s0) (thr s1 t) (shr s1)*
by(*auto simp add*: *red0-Red1'.mbisim-def*)
from *bisim1JVM* **have** *locks s' = ls wset s' = ws interrupts s' = is*
and *tbisim1JVM*: $\bigwedge t$. *Red1-execd.tbisim (ws t = None) t (thr s1 t) (shr s1) (thr s' t) (shr s')*
by(*auto simp add*: *Red1-execd.mbisim-def*)
then obtain *ts' m'* **where** *s'*: *s' = (ls, (ts', m'), ws, is)* **by**(*cases s'*) *fastforce*
{ fix *t e x ln*
assume *tst*: *ts t = ⌊((e, x), ln)⌋*
from *tbisimJ0*[*of t*] *tst* **obtain** *e' exs'* **where** *ts0t*: *thr s0 t = ⌊((e', exs'), ln)⌋*
and *bisimtJ0*: *bisim-red-red0 ((e, x), m) ((e', exs'), shr s0)*
by(*auto simp add*: *red-red0.tbisim-def*)
from *tbisim01*[*of t*] *ts0t* **obtain** *e'' xs'' exs''*
where *ts1t*: *thr s1 t = ⌊(((e'', xs''), exs''), ln)⌋*
and *bisimt01*: *bisim-red0-Red1 ((e', exs'), shr s0) (((e'', xs''), exs''), shr s1)*
by(*auto simp add*: *red0-Red1'.tbisim-def*)
from *tbisim1JVM*[*of t*] *ts1t s'* **obtain** *xcp frs*
where *ts't*: *ts' t = ⌊((xcp, frs), ln)⌋* **and** [*simp*]: *m' = shr s1*
and *bisimt1JVM*: *bisim1-list1 t m' (e'', xs'') exs'' xcp frs*
by(*fastforce simp add*: *Red1-execd.tbisim-def*)

from *fin ts't s s'* **have** [*simp*]: *frs = []* **by**(*auto dest*: *exec-mthr.mfinalD*)
from *bisimt1JVM* **have** [*simp*]: *exs'' = []* **by**(*auto elim*: *bisim1-list1.cases*)
from *bisimt01* **have** [*simp*]: *exs' = []*
by(*auto simp add*: *bisim-red0-Red1-def elim*!: *bisim-list1E elim*: *bisim-list.cases*)
from *tst fin' s* **have** *fine*: *final e* **by**(*auto dest*: *red-mthr.mfinalD*)
hence *exception (e, x) = (xcp, frs)*
proof(*cases*)
fix *v*
assume [*simp*]: *e = Val v*
from *bisimtJ0* **have** *e' = Val v* **by**(*auto elim*!: *bisim-red-red0.cases*)
with *bisimt01* **have** *e'' = Val v* **by**(*auto simp add*: *bisim-red0-Red1-def elim*: *bisim-list1E*)
with *bisimt1JVM* **have** *xcp = None* **by**(*auto elim*: *bisim1-list1.cases*)
thus *?thesis* **by** *simp*
next
fix *a*
assume [*simp*]: *e = Throw a*
from *bisimtJ0* **have** *e' = Throw a* **by**(*auto elim*!: *bisim-red-red0.cases*)
with *bisimt01* **have** *e'' = Throw a* **by**(*auto simp add*: *bisim-red0-Red1-def elim*: *bisim-list1E*)
with *bisimt1JVM* **have** *xcp = ⌊a⌋* **by**(*auto elim*: *bisim1-list1.cases*)
thus *?thesis* **by** *simp*
qed
moreover from *bisimtJ0* **have** *shr s0 = m* **by**(*auto elim*: *bisim-red-red0.cases*)
moreover from *bisimt01* **have** *shr s1 = shr s0* **by**(*auto simp add*: *bisim-red0-Red1-def*)
ultimately have *ts' t = ⌊(exception (e, x), ln)⌋ m' = m* **using** *ts't* **by** *simp-all* **}**
moreover {
fix *t*
assume *ts t = None*
with *red-red0.mbisim-thrNone-eq*[*OF bisimJ0*, *of t*] *s* **have** *thr s0 t = None* **by** *simp*
with *bisim01* **have** *thr s1 t = None* **by**(*auto simp add*: *red0-Red1'.mbisim-thrNone-eq*)
with *bisim1JVM s'* **have** *ts' t = None* **by**(*simp add*: *Red1-execd.mbisim-thrNone-eq*) **}**

ultimately show *?thesis* **using** *s s' tsNotEmpty* **by**(*auto simp add*: *mexception-def fun-eq-iff*)
qed

end

context *J-JVM-conf-read* **begin**

theorem *J2JVM-correct1*:
fixes *C M vs*
defines *s*: $s \equiv$ *J-start-state P C M vs*
and *comps*: $cs \equiv$ *JVM-start-state* (*J2JVM P*) *C M vs*
assumes *wf*: *wf-J-prog P*
and *wf-start*: *wf-start-state P C M vs*
and *red*: *red-mthr.mthr.*τ*Runs P s* ξ
obtains ξ'
where *execd-mthr.mthr.*τ*Runs* (*J2JVM P*) *cs* ξ' *tllist-all2 tlsimJ2JVM* (*rel-option bisimJ2JVM*) ξ ξ'
and $\bigwedge s'$. $[\![$ *tfinite* ξ; *terminal* $\xi = \lfloor s' \rfloor$; *red-mthr.mfinal* $s']\!]$
$\Longrightarrow$ *tfinite* $\xi' \wedge$ *terminal* $\xi' = \lfloor$*mexception* $s' \rfloor$
and $\bigwedge s'$. $[\![$ *tfinite* ξ; *terminal* $\xi = \lfloor s' \rfloor$; *red-mthr.deadlock P* $s']\!]$
$\Longrightarrow \exists cs'$. *tfinite* $\xi' \wedge$ *terminal* $\xi' = \lfloor cs' \rfloor \wedge$ *execd-mthr.deadlock* (*J2JVM P*) $cs' \wedge$ *bisimJ2JVM* s' cs'
and $[\![$ *tfinite* ξ; *terminal* ξ = *None* $]\!] \Longrightarrow$ *tfinite* $\xi' \wedge$ *terminal* ξ' = *None*
and $\neg$ *tfinite* $\xi \Longrightarrow \neg$ *tfinite* ξ'
proof −
from *wf wf-start* **have** *bisim*: *bisimJ2JVM s cs* **unfolding** *s comps* **by**(*rule bisimJ2JVM-start*)

note *divfin* = *delay-bisimulation-diverge-final.delay-bisimulation-diverge*[*OF bisimJ2JVM-weak-bisim*[*OF wf*]]
note *divfin2* = *delay-bisimulation-diverge-final.delay-bisimulation-final-base*[*OF bisimJ2JVM-weak-bisim*[*OF wf*]]

from *delay-bisimulation-diverge.simulation-*τ*Runs1*[*OF divfin*, *OF bisim red*] **obtain** ξ'
where *exec*: *execd-mthr.mthr.*τ*Runs* (*J2JVM P*) *cs* ξ'
and *tlsim*: *tllist-all2 tlsimJ2JVM* (*rel-option bisimJ2JVM*) ξ ξ' **by** *blast*
moreover {
fix s'
assume *fin*: *tfinite* ξ **and** s': *terminal* $\xi = \lfloor s' \rfloor$ **and** *final*: *red-mthr.mfinal* s'
from *delay-bisimulation-final-base.*τ*Runs-terminate-final1*[*OF divfin2*, *OF red exec tlsim fin s' final*]
obtain cs' **where** *fin'*: *tfinite* ξ' **and** cs': *terminal* $\xi' = \lfloor cs' \rfloor$
and *final'*: *exec-mthr.mfinal* cs' **by** *blast*
from *tlsim fin s' cs'* **have** *bisim'*: *bisimJ2JVM s' cs'* **by**(*auto dest*: *tllist-all2-tfinite1-terminalD*)
from *red-mthr.mthr.*τ*Runs-into-*τ*rtrancl3p*[*OF red fin s'*]
have *thr s' start-tid* $\neq$ *None* **unfolding** *s*
by(*rule red-mthr.*τ*rtrancl3p-redT-thread-not-disappear*)(*simp add*: *start-state-def*)
with *bisim' final final'* **have** [*simp*]: $cs' =$ *mexception* s'
by(*intro bisimJ2JVM-mfinal-mexception disjI1*)
with *fin' cs'* **have** *tfinite* $\xi' \wedge$ *terminal* $\xi' = \lfloor$*mexception* $s' \rfloor$ **by** *simp* **}**
moreover {
fix s'
assume *fin*: *tfinite* ξ **and** s': *terminal* $\xi = \lfloor s' \rfloor$ **and** *dead*: *red-mthr.deadlock P* s'
from *tlsim fin s'*
obtain cs' **where** *tfinite* ξ' **and** cs': *terminal* $\xi' = \lfloor cs' \rfloor$
and *bisim'*: *bisimJ2JVM* s' cs'

by(*cases terminal* ξ')(*fastforce dest*: *tllist-all2-tfinite1-terminalD tllist-all2-tfiniteD*)+
from *bisim'* **obtain** *s0' s1' S1'* **where** *bisim0*: *red-red0.mbisim s' s0'*
and *bisim01*: *red0-Red1'.mbisim s0' s1'*
and *bisim11*: *mbisim-Red1'-Red1 s1' S1'*
and *bisim12*: *Red1-execd.mbisim S1' cs'*
unfolding *bisimJ2JVM-def* **by** *auto*

note *b0* = *red-red0-FWbisim*[*OF wf-prog-wwf-prog*[*OF wf*]]
note *b01* = *red0-Red1'-FWweak-bisim*[*OF wf*]
note *b01mthr* = *FWdelay-bisimulation-diverge.mbisim-delay-bisimulation*[*OF b01*]
note *b11* = *Red1'-Red1-bisim-into-weak*[*OF compP1-pres-wf*[*OF wf*]]
note *b11delay* = *bisimulation-into-delay.delay-bisimulation*[*OF b11*]
note *b12* = *Red1-exec1-FWwbisim*[*OF compP1-pres-wf*[*OF wf*]]
note *b12mthr* = *FWdelay-bisimulation-diverge.mbisim-delay-bisimulation*[*OF b12*]

from *FWdelay-bisimulation-diverge.deadlock1-imp-*τ*s-deadlock2*[*OF b0*, *OF bisim0 dead*, *of convert-RA*]
obtain *s0''* **where** *red0-mthr.mthr.silent-moves P s0' s0''*
and *bisim0'*: *red-red0.mbisim s' s0''*
and *dead0*: *red0-mthr.deadlock P s0''* **by** *auto*

from *delay-bisimulation-diverge.simulation-silents1*[*OF b01mthr*, *OF bisim01* ‹*red0-mthr.mthr.silent-moves P s0' s0''*›]
obtain *s1''* **where** *Red1-mthr.mthr.silent-moves False* (*compP1 P*) *s1' s1''*
and *red0-Red1'.mbisim s0'' s1''* **by** *auto*
from *FWdelay-bisimulation-diverge.deadlock1-imp-*τ*s-deadlock2*[*OF b01*, *OF* ‹*red0-Red1'.mbisim s0'' s1''*› *dead0*, *of convert-RA*]
obtain *s1'''* **where** *Red1-mthr.mthr.silent-moves False* (*compP1 P*) *s1'' s1'''*
and *dead1*: *Red1-mthr.deadlock False* (*compP1 P*) *s1'''*
and *bisim01'*: *red0-Red1'.mbisim s0'' s1'''* **by** *auto*
from ‹*Red1-mthr.mthr.silent-moves False* (*compP1 P*) *s1' s1''*› ‹*Red1-mthr.mthr.silent-moves False* (*compP1 P*) *s1'' s1'''*›
have *Red1-mthr.mthr.silent-moves False* (*compP1 P*) *s1' s1'''* **by**(*rule rtranclp-trans*)

from *delay-bisimulation-diverge.simulation-silents1*[*OF b11delay*, *OF bisim11 this*]
obtain *S1''* **where** *Red1-mthr.mthr.silent-moves True* (*compP1 P*) *S1' S1''*
and *bisim11'*: *mbisim-Red1'-Red1 s1''' S1''* **by** *auto*
from *bisim11'* **have** *s1''' = S1''* **by**(*simp add*: *mbisim-Red1'-Red1-def*)
with *dead1* **have** *dead1'*: *Red1-mthr.deadlock True* (*compP1 P*) *S1''*
by(*simp add*: *Red1-Red1'-deadlock-inv*)

from *delay-bisimulation-diverge.simulation-silents1*[*OF b12mthr*, *OF bisim12* ‹*Red1-mthr.mthr.silent-moves True* (*compP1 P*) *S1' S1''*›]
obtain *cs''* **where** *execd-mthr.mthr.silent-moves* (*compP2* (*compP1 P*)) *cs' cs''*
and *Red1-execd.mbisim S1'' cs''* **by** *auto*
from *FWdelay-bisimulation-diverge.deadlock1-imp-*τ*s-deadlock2*[*OF b12* ‹*Red1-execd.mbisim S1'' cs''*› *dead1'*, *of convert-RA*]
obtain *cs'''* **where** *execd-mthr.mthr.silent-moves* (*compP2* (*compP1 P*)) *cs'' cs'''*
and *bisim12'*: *Red1-execd.mbisim S1'' cs'''*
and *dead'*: *execd-mthr.deadlock* (*compP2* (*compP1 P*)) *cs'''* **by** *auto*
from ‹*execd-mthr.mthr.silent-moves* (*compP2* (*compP1 P*)) *cs' cs''*› ‹*execd-mthr.mthr.silent-moves* (*compP2* (*compP1 P*)) *cs'' cs'''*›
have *execd-mthr.mthr.silent-moves* (*compP2* (*compP1 P*)) *cs' cs'''* **by**(*rule rtranclp-trans*)
hence *cs''' = cs'* **using** *execd-mthr.mthr.*τ*Runs-terminal-stuck*[*OF exec* ‹*tfinite* ξ'› ‹*terminal* ξ' =

⌊*cs'*⌋›]
by(*cases rule*: *converse-rtranclpE*)(*fastforce simp add*: *J2JVM-def*)+
with *dead'* **have** *execd-mthr.deadlock* (*J2JVM P*) *cs'* **by**(*simp add*: *J2JVM-def*)
hence ∃ *cs'*. *tfinite ξ'* ∧ *terminal ξ'* = ⌊*cs'*⌋ ∧ *execd-mthr.deadlock* (*J2JVM P*) *cs'* ∧ *bisimJ2JVM s' cs'*
using ‹*tfinite ξ'*› ‹*terminal ξ'* = ⌊*cs'*⌋› *bisim'* **by** *blast* **}**
moreover {
assume *tfinite ξ* **and** *terminal ξ* = *None*
hence *tfinite ξ'* ∧ *terminal ξ'* = *None* **using** *tlsim tllist-all2-tfiniteD*[*OF tlsim*]
by(*cases terminal ξ'*)(*auto dest*: *tllist-all2-tfinite1-terminalD*) **}**
moreover {
assume ¬ *tfinite ξ*
hence ¬ *tfinite ξ'* **using** *tlsim* **by**(*blast dest*: *tllist-all2-tfiniteD*) **}**
ultimately show *thesis* **by**(*rule that*)
qed

theorem *J2JVM-correct2*:
fixes *C M vs*
defines *s*: *s* ≡ *J-start-state P C M vs*
and *comps*: *cs* ≡ *JVM-start-state* (*J2JVM P*) *C M vs*
assumes *wf*: *wf-J-prog P*
and *wf-start*: *wf-start-state P C M vs*
and *exec*: *execd-mthr.mthr.τRuns* (*J2JVM P*) *cs ξ'*
obtains *ξ*
where *red-mthr.mthr.τRuns P s ξ tllist-all2 tlsimJ2JVM* (*rel-option bisimJ2JVM*) *ξ ξ'*
and ⋀*cs'*. ⟦ *tfinite ξ'*; *terminal ξ'* = ⌊*cs'*⌋; *exec-mthr.mfinal cs'* ⟧
⟹ ∃ *s'*. *tfinite ξ* ∧ *terminal ξ* = ⌊*s'*⌋ ∧ *cs'* = *mexception s'* ∧ *bisimJ2JVM s' cs'*
and ⋀*cs'*. ⟦ *tfinite ξ'*; *terminal ξ'* = ⌊*cs'*⌋; *execd-mthr.deadlock* (*J2JVM P*) *cs'* ⟧
⟹ ∃ *s'*. *tfinite ξ* ∧ *terminal ξ* = ⌊*s'*⌋ ∧ *red-mthr.deadlock P s'* ∧ *bisimJ2JVM s' cs'*
and ⟦ *tfinite ξ'*; *terminal ξ'* = *None* ⟧ ⟹ *tfinite ξ* ∧ *terminal ξ* = *None*
and ¬ *tfinite ξ'* ⟹ ¬ *tfinite ξ*
proof −
from *wf wf-start* **have** *bisim*: *bisimJ2JVM s cs* **unfolding** *s comps* **by**(*rule bisimJ2JVM-start*)

note *divfin* = *delay-bisimulation-diverge-final.delay-bisimulation-diverge*[*OF bisimJ2JVM-weak-bisim*[*OF wf*]]
note *divfin2* = *delay-bisimulation-diverge-final.delay-bisimulation-final-base*[*OF bisimJ2JVM-weak-bisim*[*OF wf*]]

from *delay-bisimulation-diverge.simulation-τRuns2*[*OF divfin*, *OF bisim exec*] **obtain** *ξ*
where *red*: *red-mthr.mthr.τRuns P s ξ*
and *tlsim*: *tllist-all2 tlsimJ2JVM* (*rel-option bisimJ2JVM*) *ξ ξ'* **by** *blast*
moreover {
fix *cs'*
assume *fin*: *tfinite ξ'* **and** *cs'*: *terminal ξ'* = ⌊*cs'*⌋ **and** *final*: *exec-mthr.mfinal cs'*
from *delay-bisimulation-final-base.τRuns-terminate-final2*[*OF divfin2*, *OF red exec tlsim fin cs' final*]
obtain *s'* **where** *fin'*: *tfinite ξ* **and** *s'*: *terminal ξ* = ⌊*s'*⌋
and *final'*: *red-mthr.mfinal s'* **by** *blast*
from *tlsim fin s' cs'* **have** *bisim'*: *bisimJ2JVM s' cs'* **by**(*auto dest*: *tllist-all2-tfinite2-terminalD*)
from *red-mthr.mthr.τRuns-into-τrtrancl3p*[*OF red fin' s'*]
have *thr s' start-tid* ≠ *None* **unfolding** *s*
by(*rule red-mthr.τrtrancl3p-redT-thread-not-disappear*)(*simp add*: *start-state-def*)
with *bisim' final final'* **have** [*simp*]: *cs'* = *mexception s'*

by(*intro bisimJ2JVM-mfinal-mexception*)
with *fin′ s′ bisim′* **have** ∃ *s′*. *tfinite ξ* ∧ *terminal ξ* = ⌊*s′*⌋ ∧ *cs′* = *mexception s′* ∧ *bisimJ2JVM s′ cs′* **by** *simp* **}**
moreover {
fix *cs′*
assume *fin*: *tfinite ξ′* **and** *cs′*: *terminal ξ′* = ⌊*cs′*⌋ **and** *dead′*: *execd-mthr.deadlock* (*J2JVM P*) *cs′*
from *tlsim fin cs′*
obtain *s′* **where** *tfinite ξ* **and** *s′*: *terminal ξ* = ⌊*s′*⌋
and *bisim′*: *bisimJ2JVM s′ cs′*
by(*cases terminal ξ*)(*fastforce dest*: *tllist-all2-tfinite2-terminalD tllist-all2-tfiniteD*)+
from *bisim′* **obtain** *s0′ s1′ S1′* **where** *bisim0*: *red-red0.mbisim s′ s0′*
and *bisim01*: *red0-Red1′.mbisim s0′ s1′*
and *bisim11*: *mbisim-Red1′-Red1 s1′ S1′*
and *bisim12*: *Red1-execd.mbisim S1′ cs′*
unfolding *bisimJ2JVM-def* **by** *auto*

note *b0* = *red-red0-FWbisim*[*OF wf-prog-wwf-prog*[*OF wf*]]
note *b0mthr* = *FWdelay-bisimulation-diverge.mbisim-delay-bisimulation*[*OF b0*]
note *b01* = *red0-Red1′-FWweak-bisim*[*OF wf*]
note *b01mthr* = *FWdelay-bisimulation-diverge.mbisim-delay-bisimulation*[*OF b01*]
note *b11* = *Red1′-Red1-bisim-into-weak*[*OF compP1-pres-wf*[*OF wf*]]
note *b11delay* = *bisimulation-into-delay.delay-bisimulation*[*OF b11*]
note *b12* = *Red1-exec1-FWwbisim*[*OF compP1-pres-wf*[*OF wf*]]

from *FWdelay-bisimulation-diverge.deadlock2-imp-τs-deadlock1*[*OF b12 bisim12*, *of convert-RA*] *dead′*
obtain *S1″* **where** *Red1-mthr.mthr.silent-moves True* (*compP1 P*) *S1′ S1″*
and *bisim12′*: *Red1-execd.mbisim S1″ cs′*
and *dead′*: *Red1-mthr.deadlock True* (*compP1 P*) *S1″* **by**(*auto simp add*: *J2JVM-def*)
from *delay-bisimulation-diverge.simulation-silents2*[*OF b11delay*, *OF bisim11* ‹*Red1-mthr.mthr.silent-moves True* (*compP1 P*) *S1′ S1″*›]
obtain *s1″* **where** *Red1-mthr.mthr.silent-moves False* (*compP1 P*) *s1′ s1″*
and *bisim11′*: *mbisim-Red1′-Red1 s1″ S1″* **by** *blast*
from *bisim11′* **have** *s1″* = *S1″* **by**(*simp add*: *mbisim-Red1′-Red1-def*)
with *dead′* **have** *dead1*: *Red1-mthr.deadlock False* (*compP1 P*) *s1″*
by(*simp add*: *Red1-Red1′-deadlock-inv*)
from *delay-bisimulation-diverge.simulation-silents2*[*OF b01mthr*, *OF bisim01* ‹*Red1-mthr.mthr.silent-moves False* (*compP1 P*) *s1′ s1″*›]
obtain *s0″* **where** *red0-mthr.mthr.silent-moves P s0′ s0″*
and *bisim01′*: *red0-Red1′.mbisim s0″ s1″* **by** *auto*
from *FWdelay-bisimulation-diverge.deadlock2-imp-τs-deadlock1*[*OF b01 bisim01′ dead1*, *of convert-RA*]
obtain *s0‴* **where** *red0-mthr.mthr.silent-moves P s0″ s0‴*
and *bisim01″*: *red0-Red1′.mbisim s0‴ s1″*
and *dead0*: *red0-mthr.deadlock P s0‴* **by** *auto*
from ‹*red0-mthr.mthr.silent-moves P s0′ s0″*› ‹*red0-mthr.mthr.silent-moves P s0″ s0‴*›
have *red0-mthr.mthr.silent-moves P s0′ s0‴* **by**(*rule rtranclp-trans*)
from *delay-bisimulation-diverge.simulation-silents2*[*OF b0mthr*, *OF bisim0 this*]
obtain *s″* **where** *red-mthr.mthr.silent-moves P s′ s″*
and *red-red0.mbisim s″ s0‴* **by** *blast*
from *FWdelay-bisimulation-diverge.deadlock2-imp-τs-deadlock1*[*OF b0* ‹*red-red0.mbisim s″ s0‴*› *dead0*, *of convert-RA*]
obtain *s‴* **where** *red-mthr.mthr.silent-moves P s″ s‴*
and *red-red0.mbisim s‴ s0‴*

and *dead*: *red-mthr.deadlock P s‴* **by** *blast*
from ‹*red-mthr.mthr.silent-moves P s′ s″*› ‹*red-mthr.mthr.silent-moves P s″ s‴*›
have *red-mthr.mthr.silent-moves P s′ s‴* **by**(*rule rtranclp-trans*)
hence *s‴ = s′* **using** *red-mthr.mthr.τRuns-terminal-stuck*[*OF red* ‹*tfinite ξ*› ‹*terminal ξ = ⌊s′⌋*›]
by(*cases rule*: *converse-rtranclpE*) *fastforce+*
with *dead* **have** *red-mthr.deadlock P s′* **by**(*simp*)
hence ∃ *s′*. *tfinite ξ* ∧ *terminal ξ = ⌊s′⌋* ∧ *red-mthr.deadlock P s′* ∧ *bisimJ2JVM s′ cs′*
using ‹*tfinite ξ*› ‹*terminal ξ = ⌊s′⌋*› *bisim′* **by** *blast* **}**
moreover {
assume *tfinite ξ′* **and** *terminal ξ′ = None*
hence *tfinite ξ* ∧ *terminal ξ = None* **using** *tlsim tllist-all2-tfiniteD*[*OF tlsim*]
by(*cases terminal ξ*)(*auto dest*: *tllist-all2-tfinite2-terminalD*) **}**
moreover {
assume ¬ *tfinite ξ′*
hence ¬ *tfinite ξ* **using** *tlsim* **by**(*blast dest*: *tllist-all2-tfiniteD*) **}**
ultimately show *thesis* **by**(*rule that*)
qed

end

declare *compP1-def* [*simp*]

theorem *wt-J2JVM*: *wf-J-prog P* ⟹ *wf-jvm-prog* (*J2JVM P*)
unfolding *J2JVM-def o-def*
by(*rule wt-compP2*)(*rule compP1-pres-wf*)

end
theory *Preprocessor*
imports
PCompiler
../J/Annotate
../J/JWellForm
begin

primrec *annotate-Mb* ::
'addr J-prog ⇒ *cname* ⇒ *mname* ⇒ *ty list* ⇒ *ty* ⇒ (*vname list* × *'addr expr*) ⇒ (*vname list* × *'addr expr*)
where *annotate-Mb P C M Ts T* (*pns*, *e*) = (*pns*, *annotate P* [*this # pns* [↦] *Class C # Ts*] *e*)
declare *annotate-Mb.simps* [*simp del*]

primrec *annotate-Mb-code* ::
'addr J-prog ⇒ *cname* ⇒ *mname* ⇒ *ty list* ⇒ *ty* ⇒ (*vname list* × *'addr expr*) ⇒ (*vname list* × *'addr expr*)
where *annotate-Mb-code P C M Ts T* (*pns*, *e*) = (*pns*, *annotate-code P* [*this # pns* [↦] *Class C # Ts*] *e*)
declare *annotate-Mb-code.simps* [*simp del*]

definition *annotate-prog* :: *'addr J-prog* ⇒ *'addr J-prog*
where *annotate-prog P = compP* (*annotate-Mb P*) *P*

definition *annotate-prog-code* :: *'addr J-prog* ⇒ *'addr J-prog*
where *annotate-prog-code P = compP* (*annotate-Mb-code P*) *P*

lemma fixes *is-lub*

shows *WT-compP*: *is-lub*,*P*,*E* ⊢ *e* :: *T* ⟹ *is-lub*,*compP f P*,*E* ⊢ *e* :: *T*
and *WTs-compP*: *is-lub*,*P*,*E* ⊢ *es* [::] *Ts* ⟹ *is-lub*,*compP f P*,*E* ⊢ *es* [::] *Ts*
proof(*induct rule*: *WT-WTs.inducts*)
case (*WTCall E e U C M Ts T meth D es Ts'*)
from ‹*P* ⊢ *C sees M*: *Ts*→*T* = *meth in D*›
have *compP f P* ⊢ *C sees M*: *Ts*→*T* = *map-option* (*f D M Ts T*) *meth in D*
by(*auto dest*: *sees-method-compP*[**where** *f*=*f*])
with *WTCall* **show** *?case* **by**(*auto*)
qed(*auto simp del*: *fun-upd-apply*)

lemma fixes *is-lub*
shows *Anno-compP*: *is-lub*,*P*,*E* ⊢ *e* ⇝ *e'* ⟹ *is-lub*,*compP f P*,*E* ⊢ *e* ⇝ *e'*
and *Annos-compP*: *is-lub*,*P*,*E* ⊢ *es* [⇝] *es'* ⟹ *is-lub*,*compP f P*,*E* ⊢ *es* [⇝] *es'*
apply(*induct rule*: *Anno-Annos.inducts*)
apply(*auto intro*: *Anno-Annos.intros simp del*: *fun-upd-apply dest*: *WT-compP simp add*: *compC-def*)
done

lemma *annotate-prog-code-eq-annotate-prog*:
assumes *wf*: *wf-J-prog* (*annotate-prog-code P*)
shows *annotate-prog-code P* = *annotate-prog P*
proof −
let *?wf-md* = λ- - (-,-,-,-,*body*). *set* (*block-types body*) ⊆ *types P*
from *wf* **have** *wf-prog ?wf-md* (*annotate-prog-code P*)
unfolding *annotate-prog-code-def*
by(*rule wf-prog-lift*)(*auto dest*!: *WT-block-types-is-type*[*OF wf*[*unfolded annotate-prog-code-def*]] *simp add*: *wf-J-mdecl-def*)
hence *wf'*: *wf-prog ?wf-md P*
unfolding *annotate-prog-code-def* [*abs-def*]
proof(*rule wf-prog-compPD*)
fix *C M Ts T m*
assume *compP* (*annotate-Mb-code P*) *P* ⊢ *C sees M*: *Ts*→*T* = ⌊*annotate-Mb-code P C M Ts T m*⌋ *in C*
and *wf-mdecl ?wf-md* (*compP* (*annotate-Mb-code P*) *P*) *C* (*M*, *Ts*, *T*, ⌊*annotate-Mb-code P C M Ts T m*⌋)
moreover obtain *pns body* **where** *m* = (*pns*, *body*) **by**(*cases m*)
ultimately show *wf-mdecl ?wf-md P C* (*M*, *Ts*, *T*, ⌊*m*⌋)
by(*fastforce simp add*: *annotate-Mb-code-def annotate-code-def wf-mdecl-def THE-default-def the-equality Anno-code-def split*: *if-split-asm dest*: *Anno-block-types*)
qed

{ **fix** *C D fs ms M Ts T pns body*
assume (*C*, *D*, *fs*, *ms*) ∈ *set* (*classes P*)
and (*M*, *Ts*, *T*, ⌊(*pns*, *body*)⌋) ∈ *set ms*
from ‹(*C*, *D*, *fs*, *ms*) ∈ *set* (*classes P*)› **have** *class P C* = ⌊(*D*, *fs*, *ms*)⌋ **using** *wf'*
by(*cases P*)(*auto simp add*: *wf-prog-def dest*: *map-of-SomeI*)
with *wf'* **have** *sees*: *P* ⊢ *C sees M*:*Ts*→*T* = ⌊(*pns*, *body*)⌋ *in C*
using ‹(*M*, *Ts*, *T*, ⌊(*pns*, *body*)⌋) ∈ *set ms*› **by**(*rule mdecl-visible*)

from *sees-method-compP*[*OF this*, **where** *f*=*annotate-Mb-code P*]
have *sees'*: *annotate-prog-code P* ⊢ *C sees M*:*Ts*→*T* = ⌊(*pns*, *annotate-code P* [*this* ↦ *Class C*, *pns* [↦] *Ts*] *body*)⌋ *in C*
unfolding *annotate-prog-code-def annotate-Mb-code-def* **by**(*auto*)
with *wf*
have *wf-mdecl wf-J-mdecl* (*annotate-prog-code P*) *C* (*M*, *Ts*, *T*, ⌊(*pns*, *annotate-code P* [*this* ↦

Class C, *pns* [↦] *Ts*] *body*)⌋)
by(*rule sees-wf-mdecl*)
hence *set Ts* ⊆ *types P* **by**(*auto simp add*: *wf-mdecl-def annotate-prog-code-def*)
moreover from *sees* **have** *is-class P C* **by**(*rule sees-method-is-class*)
moreover from *wf' sees* **have** *wf-mdecl ?wf-md P C* (*M*, *Ts*, *T*, ⌊(*pns*, *body*)⌋) **by**(*rule sees-wf-mdecl*)
hence *set* (*block-types body*) ⊆ *types P* **by**(*simp add*: *wf-mdecl-def*)
ultimately have *ran* [*this* ↦ *Class C*, *pns* [↦] *Ts*] ∪ *set* (*block-types body*) ⊆ *types P*
by(*auto simp add*: *ran-def wf-mdecl-def map-upds-def split*: *if-split-asm dest*!: *map-of-SomeD set-zip-rightD*)
hence *annotate-code P* [*this* ↦ *Class C*, *pns* [↦] *Ts*] *body* = *annotate P* [*this* ↦ *Class C*, *pns* [↦] *Ts*] *body*
unfolding *annotate-code-def annotate-def*
by −(*rule arg-cong*[**where** *f*=*THE-default body*], *auto intro*!: *ext intro*: *Anno-code-into-Anno*[*OF wf'*] *Anno-into-Anno-code*[*OF wf'*]) **}**
thus *?thesis* **unfolding** *annotate-prog-code-def annotate-prog-def*
by(*cases P*)(*auto simp add*: *compC-def compM-def annotate-Mb-def annotate-Mb-code-def map-option-case*)
qed

end
theory *Compiler-Main*
imports
J0
Correctness
Preprocessor
begin

end

Chapter 8

Memory Models

theory *MM*
imports
../Common/Heap
begin

type-synonym *addr* = *nat*
type-synonym *thread-id* = *addr*

abbreviation (*input*)
addr2thread-id :: *addr* ⇒ *thread-id*
where *addr2thread-id* ≡ λ*x*. *x*

abbreviation (*input*)
thread-id2addr :: *thread-id* ⇒ *addr*
where *thread-id2addr* ≡ λ*x*. *x*

instantiation *nat* :: *addr* **begin**
definition *hash-addr* ≡ *int*
definition *monitor-finfun-to-list* ≡ (*finfun-to-list* :: *nat* ⇒*f* *nat* ⇒ *nat list*)
instance
by(*intro-classes*)(*simp add*: *monitor-finfun-to-list-nat-def*)
end

definition *new-Addr* :: (*addr* ⇀ *'b*) ⇒ *addr option*
where *new-Addr h* ≡ *if* ∃ *a*. *h a* = *None then Some*(*LEAST a*. *h a* = *None*) *else None*

lemma *new-Addr-SomeD*:
new-Addr h = *Some a* ⟹ *h a* = *None*
by(*auto simp add*:*new-Addr-def split*:*if-splits intro*: *LeastI*)

lemma *new-Addr-SomeI*:
finite (*dom h*) ⟹ ∃ *a*. *new-Addr h* = *Some a*
by(*simp add*: *new-Addr-def*) (*metis finite-map-freshness infinite-UNIV-nat*)

8.0.1 Code generation

definition *gen-new-Addr* :: (*addr* ⇀ *'b*) ⇒ *addr* ⇒ *addr option*
where *gen-new-Addr h n* ≡ *if* ∃ *a*. *a* ≥ *n* ∧ *h a* = *None then Some*(*LEAST a*. *a* ≥ *n* ∧ *h a* = *None*)
else None

lemma *new-Addr-code-code* [*code*]:
new-Addr h = gen-new-Addr h 0
by(*simp add*: *new-Addr-def gen-new-Addr-def*)

lemma *gen-new-Addr-code* [*code*]:
gen-new-Addr h n = (if h n = None then Some n else gen-new-Addr h (Suc n))
apply(*simp add*: *gen-new-Addr-def*)
apply(*rule impI*)
apply(*rule conjI*)
apply *safe*[*1*]
apply(*auto intro*: *Least-equality*)[*2*]
apply(*rule arg-cong*[**where** *f*=*Least*])
apply(*rule ext*)
apply *auto*[*1*]
apply(*case-tac n = ab*)
apply *simp*
apply *simp*
apply *clarify*
apply(*subgoal-tac a = n*)
apply *simp*
apply(*rule Least-equality*)
apply *auto*[*2*]
apply(*rule ccontr*)
apply(*erule-tac x*=*a* **in** *allE*)
apply *simp*
done

end

8.1 Sequential consistency

theory *SC*
imports
../Common/Conform
MM
begin

8.1.1 Objects and Arrays

type-synonym
fields = vname × cname ⇀ addr val — field name, defining class, value

type-synonym
cells = addr val list

datatype *heapobj*
= Obj cname fields
— class instance with class name and fields

| Arr ty fields cells
— element type, fields (from object), and list of each cell's content

lemma *rec-heapobj* [*simp*]: *rec-heapobj = case-heapobj*

by(*auto intro*!: *ext split*: *heapobj.split*)

primrec *obj-ty* :: *heapobj* ⇒ *htype*
where
  obj-ty (*Obj C f*) = *Class-type C*
| *obj-ty* (*Arr T fs cs*) = *Array-type T* (*length cs*)

fun *is-Arr* :: *heapobj* ⇒ *bool* **where**
  is-Arr (*Obj C fs*) = *False*
| *is-Arr* (*Arr T f el*) = *True*

lemma *is-Arr-conv*:
  is-Arr arrobj = (∃ *T f el*. *arrobj* = *Arr T f el*)
by(*cases arrobj*, *auto*)

lemma *is-ArrE*:
  ⟦ *is-Arr arrobj*; ⋀*T f el*. *arrobj* = *Arr T f el* ⟹ *thesis* ⟧ ⟹ *thesis*
  ⟦ ¬ *is-Arr arrobj*; ⋀*C fs*. *arrobj* = *Obj C fs* ⟹ *thesis* ⟧ ⟹ *thesis*
by(*cases arrobj*, *auto*)+

definition *init-fields* :: ('*field-name* × (*ty* × *fmod*)) *list* ⇒ '*field-name* ⇀ *addr val*
where *init-fields* ≡ *map-of* ∘ *map* (λ(*FD*,(*T*, *fm*)). (*FD*,*default-val T*))

primrec
  — a new, blank object with default values in all fields:
  blank :: '*m prog* ⇒ *htype* ⇒ *heapobj*
where
  blank P (*Class-type C*) = *Obj C* (*init-fields* (*fields P C*))
| *blank P* (*Array-type T n*) = *Arr T* (*init-fields* (*fields P Object*)) (*replicate n* (*default-val T*))

lemma *obj-ty-blank* [*iff*]:
  obj-ty (*blank P hT*) = *hT*
by(*cases hT*)(*simp-all*)

8.1.2 Heap

type-synonym *heap* = *addr* ⇀ *heapobj*

translations
  (*type*) *heap* <= (*type*) *nat* ⇒ *heapobj option*

abbreviation *sc-empty* :: *heap*
where *sc-empty* ≡ *Map.empty*

fun *the-obj* :: *heapobj* ⇒ *cname* × *fields* **where**
  the-obj (*Obj C fs*) = (*C*, *fs*)

fun *the-arr* :: *heapobj* ⇒ *ty* × *fields* × *cells* **where**
  the-arr (*Arr T f el*) = (*T*, *f*, *el*)

abbreviation
  cname-of :: *heap* ⇒ *addr* ⇒ *cname* **where**
  cname-of hp a == *fst* (*the-obj* (*the* (*hp a*)))

definition *sc-allocate* :: *'m prog* ⇒ *heap* ⇒ *htype* ⇒ (*heap* × *addr*) *set*
where
sc-allocate P h hT =
(*case new-Addr h of None* ⇒ {}
| *Some a* ⇒ {(*h*(*a* ↦ *blank P hT*), *a*)})

definition *sc-typeof-addr* :: *heap* ⇒ *addr* ⇒ *htype option*
where *sc-typeof-addr h a* = *map-option obj-ty* (*h a*)

inductive *sc-heap-read* :: *heap* ⇒ *addr* ⇒ *addr-loc* ⇒ *addr val* ⇒ *bool*
for *h* :: *heap* **and** *a* :: *addr*
where
Obj: ⟦ *h a* = ⌊*Obj C fs*⌋; *fs* (*F*, *D*) = ⌊*v*⌋ ⟧ ⟹ *sc-heap-read h a* (*CField D F*) *v*
| *Arr*: ⟦ *h a* = ⌊*Arr T f el*⌋; *n* < *length el* ⟧ ⟹ *sc-heap-read h a* (*ACell n*) (*el* ! *n*)
| *ArrObj*: ⟦ *h a* = ⌊*Arr T f el*⌋; *f* (*F*, *Object*) = ⌊*v*⌋ ⟧ ⟹ *sc-heap-read h a* (*CField Object F*) *v*

hide-fact (**open**) *Obj Arr ArrObj*

inductive-cases *sc-heap-read-cases* [*elim*!]:
sc-heap-read h a (*CField C F*) *v*
sc-heap-read h a (*ACell n*) *v*

inductive *sc-heap-write* :: *heap* ⇒ *addr* ⇒ *addr-loc* ⇒ *addr val* ⇒ *heap* ⇒ *bool*
for *h* :: *heap* **and** *a* :: *addr*
where
Obj: ⟦ *h a* = ⌊*Obj C fs*⌋; *h'* = *h*(*a* ↦ *Obj C* (*fs*((*F*, *D*) ↦ *v*))) ⟧ ⟹ *sc-heap-write h a* (*CField D F*) *v h'*
| *Arr*: ⟦ *h a* = ⌊*Arr T f el*⌋; *h'* = *h*(*a* ↦ *Arr T f* (*el*[*n* := *v*])) ⟧ ⟹ *sc-heap-write h a* (*ACell n*) *v h'*
| *ArrObj*: ⟦ *h a* = ⌊*Arr T f el*⌋; *h'* = *h*(*a* ↦ *Arr T* (*f*((*F*, *Object*) ↦ *v*)) *el*) ⟧ ⟹ *sc-heap-write h a* (*CField Object F*) *v h'*

hide-fact (**open**) *Obj Arr ArrObj*

inductive-cases *sc-heap-write-cases* [*elim*!]:
sc-heap-write h a (*CField C F*) *v h'*
sc-heap-write h a (*ACell n*) *v h'*

consts *sc-spurious-wakeups* :: *bool*

interpretation *sc*:
heap-base
addr2thread-id
thread-id2addr
sc-spurious-wakeups
sc-empty
sc-allocate P
sc-typeof-addr
sc-heap-read
sc-heap-write
for *P* **.**

Translate notation from *heap-base*

abbreviation *sc-preallocated* :: *'m prog* ⇒ *heap* ⇒ *bool*
where *sc-preallocated* == *sc.preallocated TYPE*('*m*)

abbreviation *sc-start-tid* :: *'md prog* ⇒ *thread-id*
where *sc-start-tid* ≡ *sc.start-tid TYPE('md)*

abbreviation *sc-start-heap-ok* :: *'m prog* ⇒ *bool*
where *sc-start-heap-ok* ≡ *sc.start-heap-ok TYPE('m)*

abbreviation *sc-start-heap* :: *'m prog* ⇒ *heap*
where *sc-start-heap* ≡ *sc.start-heap TYPE('m)*

abbreviation *sc-start-state* ::
(*cname* ⇒ *mname* ⇒ *ty list* ⇒ *ty* ⇒ *'m* ⇒ *addr val list* ⇒ *'x*)
⇒ *'m prog* ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒ (*addr*, *thread-id*, *'x*, *heap*, *addr*) *state*
where
sc-start-state f P ≡ *sc.start-state TYPE('m) P f P*

abbreviation *sc-wf-start-state* :: *'m prog* ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒ *bool*
where *sc-wf-start-state P* ≡ *sc.wf-start-state TYPE('m) P P*

notation *sc.conf* (‹-,- ⊢*sc* - :≤ -› [*51*,*51*,*51*,*51*] *50*)
notation *sc.confs* (‹-,- ⊢*sc* - [:≤] -› [*51*,*51*,*51*,*51*,*51*] *50*)
notation *sc.hext* (‹- ≤*sc* -› [*51*,*51*] *50*)

lemma *sc-start-heap-ok*: *sc-start-heap-ok P*
apply(*simp add*: *sc.start-heap-ok-def sc.start-heap-data-def initialization-list-def sc.create-initial-object-simps sc-allocate-def sys-xcpts-list-def case-option-conv-if new-Addr-SomeI del*: *blank.simps split del*: *option.split if-split*)
done

lemma *sc-wf-start-state-iff*:
sc-wf-start-state P C M vs ⟷ (∃ *Ts T meth D*. *P* ⊢ *C sees M*:*Ts*→*T* = ⌊*meth*⌋ *in D* ∧ *P*,*sc-start-heap P* ⊢*sc vs* [:≤] *Ts*)
by(*simp add*: *sc.wf-start-state.simps sc-start-heap-ok*)

lemma *sc-heap*:
heap addr2thread-id thread-id2addr (*sc-allocate P*) *sc-typeof-addr sc-heap-write P*
proof
fix *h' a h hT*
assume (*h'*, *a*) ∈ *sc-allocate P h hT*
thus *sc-typeof-addr h' a* = ⌊*hT*⌋
by(*auto simp add*: *sc-allocate-def sc-typeof-addr-def dest*: *new-Addr-SomeD split*: *if-split-asm*)
next
fix *h' h hT a*
assume (*h'*, *a*) ∈ *sc-allocate P h hT*
from *this*[*symmetric*] **show** *h* ≤*sc h'*
by(*fastforce simp add*: *sc-allocate-def sc-typeof-addr-def sc.hext-def dest*: *new-Addr-SomeD intro*!: *map-leI*)
next
fix *h a al v h'*
assume *sc-heap-write h a al v h'*
thus *h* ≤*sc h'*
by(*cases al*)(*auto intro*!: *sc.hextI simp add*: *sc-typeof-addr-def*)
qed *simp*

interpretation *sc*:
  heap
    addr2thread-id
    thread-id2addr
    sc-spurious-wakeups
    sc-empty
    sc-allocate P
    sc-typeof-addr
    sc-heap-read
    sc-heap-write
  for *P* **by**(*rule sc-heap*)

lemma *sc-hext-new*:
  h a = *None* ⟹ *h* ⊴*sc h*(*a* ↦ *arrobj*)
by(*rule sc.hextI*)(*auto simp add*: *sc-typeof-addr-def dest*!: *new-Addr-SomeD*)

lemma *sc-hext-upd-obj*: *h a* = *Some* (*Obj C fs*) ⟹ *h* ⊴*sc h*(*a*↦(*Obj C fs'*))
by(*rule sc.hextI*)(*auto simp*:*fun-upd-apply sc-typeof-addr-def*)

lemma *sc-hext-upd-arr*: ⟦ *h a* = *Some* (*Arr T f e*); *length e* = *length e'* ⟧ ⟹ *h* ⊴*sc h*(*a*↦(*Arr T f' e'*))
by(*rule sc.hextI*)(*auto simp*:*fun-upd-apply sc-typeof-addr-def*)

8.1.3 Conformance

definition *sc-fconf* :: *'m prog* ⇒ *cname* ⇒ *heap* ⇒ *fields* ⇒ *bool* (‹-,-,- ⊢*sc* - √› [*51*,*51*,*51*,*51*] *50*)
where *P*,*C*,*h* ⊢*sc fs* √ = (∀ *F D T fm*. *P* ⊢ *C has F*:*T* (*fm*) *in D* ⟶ (∃ *v*. *fs*(*F*,*D*) = *Some v* ∧ *P*,*h* ⊢*sc v* :≤ *T*))

primrec *sc-oconf* :: *'m prog* ⇒ *heap* ⇒ *heapobj* ⇒ *bool* (‹-,- ⊢*sc* - √› [*51*,*51*,*51*] *50*)
where
  P,*h* ⊢*sc Obj C fs* √ ⟷ *is-class P C* ∧ *P*,*C*,*h* ⊢*sc fs* √
| *P*,*h* ⊢*sc Arr T fs el* √ ⟷ *is-type P* (*T*⌊⌋) ∧ *P*,*Object*,*h* ⊢*sc fs* √ ∧ (∀ *v* ∈ *set el*. *P*,*h* ⊢*sc v* :≤ *T*)

definition *sc-hconf* :: *'m prog* ⇒ *heap* ⇒ *bool* (‹- ⊢*sc* - √› [*51*,*51*] *50*)
where *P* ⊢*sc h* √ ⟷ (∀ *a obj*. *h a* = *Some obj* ⟶ *P*,*h* ⊢*sc obj* √)

interpretation *sc*: *heap-conf-base*
  addr2thread-id
  thread-id2addr
  sc-spurious-wakeups
  sc-empty
  sc-allocate P
  sc-typeof-addr
  sc-heap-read
  sc-heap-write
  sc-hconf P
  P
for *P* **.**

declare *sc.typeof-addr-thread-id2-addr-addr2thread-id* [*simp del*]

lemma *sc-conf-upd-obj*: *h a* = *Some*(*Obj C fs*) ⟹ (*P*,*h*(*a*↦(*Obj C fs'*)) ⊢*sc x* :≤ *T*) = (*P*,*h* ⊢*sc x* :≤ *T*)

apply (*unfold sc.conf-def*)
apply (*rule val.induct*)
apply (*auto simp*:*fun-upd-apply*)
apply (*auto simp add*: *sc-typeof-addr-def split*: *if-split-asm*)
done

lemma *sc-conf-upd-arr*: *h a* = *Some*(*Arr T f el*) ⟹ (*P*,*h*(*a*↦(*Arr T f′ el′*)) ⊢*sc x* :≤ *T′*) = (*P*,*h* ⊢*sc x* :≤ *T′*)
apply(*unfold sc.conf-def*)
apply (*rule val.induct*)
apply (*auto simp*:*fun-upd-apply*)
apply(*auto simp add*: *sc-typeof-addr-def split*: *if-split-asm*)
done

lemma *sc-oconf-hext*: *P*,*h* ⊢*sc obj* √ ⟹ *h* ⊴*sc h′* ⟹ *P*,*h′* ⊢*sc obj* √
by(*cases obj*)(*fastforce elim*: *sc.conf-hext simp add*: *sc-fconf-def*)+

lemma *sc-oconf-init-fields*:
assumes *P* ⊢ *C has-fields FDTs*
shows *P*,*h* ⊢*sc* (*Obj C* (*init-fields FDTs*)) √
using *assms has-fields-is-class*[*OF assms*]
by(*auto simp add*: *has-field-def init-fields-def sc-fconf-def split-def o-def map-of-map*[*simplified split-def*, **where** *f*=λ*p*. *default-val* (*fst p*)] *dest*: *has-fields-fun*)

lemma *sc-oconf-init*:
is-htype P hT ⟹ *P*,*h* ⊢*sc blank P hT* √
by(*cases hT*)(*auto simp add*: *sc-fconf-def has-field-def init-fields-def split-def o-def map-of-map*[*simplified split-def*, **where** *f*=λ*p*. *default-val* (*fst p*)] *dest*: *has-fields-fun*)

lemma *sc-oconf-fupd* [*intro?*]:
⟦ *P* ⊢ *C has F*:*T* (*fm*) *in D*; *P*,*h* ⊢*sc v* :≤ *T*; *P*,*h* ⊢*sc* (*Obj C fs*) √ ⟧
⟹ *P*,*h* ⊢*sc* (*Obj C* (*fs*((*F*,*D*)↦*v*))) √
unfolding *has-field-def*
by(*auto simp add*: *sc-fconf-def has-field-def dest*: *has-fields-fun*)

lemma *sc-oconf-fupd-arr* [*intro?*]:
⟦ *P*,*h* ⊢*sc v* :≤ *T*; *P*,*h* ⊢*sc* (*Arr T f el*) √ ⟧
⟹ *P*,*h* ⊢*sc* (*Arr T f* (*el*[*i* := *v*])) √
by(*auto dest*: *subsetD*[*OF set-update-subset-insert*])

lemma *sc-oconf-fupd-arr-fields*:
⟦ *P* ⊢ *Object has F*:*T* (*fm*) *in Object*; *P*,*h* ⊢*sc v* :≤ *T*; *P*,*h* ⊢*sc* (*Arr T′ f el*) √ ⟧
⟹ *P*,*h* ⊢*sc* (*Arr T′* (*f*((*F*, *Object*) ↦ *v*)) *el*) √
by(*auto dest*: *has-fields-fun simp add*: *sc-fconf-def has-field-def*)

lemma *sc-oconf-new*: ⟦ *P*,*h* ⊢*sc obj* √; *h a* = *None* ⟧ ⟹ *P*,*h*(*a* ↦ *arrobj*) ⊢*sc obj* √
by(*erule sc-oconf-hext*)(*rule sc-hext-new*)

lemmas *sc-oconf-upd-obj* = *sc-oconf-hext* [*OF* - *sc-hext-upd-obj*]

lemma *sc-oconf-upd-arr*:
assumes *P*,*h* ⊢*sc obj* √
and *ha*: *h a* = ⌊*Arr T f el*⌋
shows *P*,*h*(*a* ↦ *Arr T f′ el′*) ⊢*sc obj* √

using *assms*
by(*cases obj*)(*auto simp add*: *sc-conf-upd-arr*[**where** *h*=*h*, *OF ha*] *sc-fconf-def*)

lemma *sc-hconfD*: ⟦ *P* ⊢*sc h* √; *h a* = *Some obj* ⟧ ⟹ *P*,*h* ⊢*sc obj* √
unfolding *sc-hconf-def* **by** *blast*

lemmas *sc-preallocated-new* = *sc.preallocated-hext*[*OF - sc-hext-new*]
lemmas *sc-preallocated-upd-obj* = *sc.preallocated-hext* [*OF - sc-hext-upd-obj*]
lemmas *sc-preallocated-upd-arr* = *sc.preallocated-hext* [*OF - sc-hext-upd-arr*]

lemma *sc-hconf-new*: ⟦ *P* ⊢*sc h* √; *h a* = *None*; *P*,*h* ⊢*sc obj* √ ⟧ ⟹ *P* ⊢*sc h*(*a*↦*obj*) √
unfolding *sc-hconf-def*
by(*auto intro*: *sc-oconf-new*)

lemma *sc-hconf-upd-obj*: ⟦ *P* ⊢*sc h* √; *h a* = *Some* (*Obj C fs*); *P*,*h* ⊢*sc* (*Obj C fs'*) √ ⟧ ⟹ *P* ⊢*sc h*(*a*↦(*Obj C fs'*)) √
unfolding *sc-hconf-def*
by(*auto intro*: *sc-oconf-upd-obj simp del*: *sc-oconf.simps*)

lemma *sc-hconf-upd-arr*: ⟦ *P* ⊢*sc h* √; *h a* = *Some*(*Arr T f el*); *P*,*h* ⊢*sc* (*Arr T f' el'*) √ ⟧ ⟹ *P* ⊢*sc h*(*a*↦(*Arr T f' el'*)) √
unfolding *sc-hconf-def*
by(*auto intro*: *sc-oconf-upd-arr simp del*: *sc-oconf.simps*)

lemma *sc-heap-conf*:
heap-conf addr2thread-id thread-id2addr sc-empty (*sc-allocate P*) *sc-typeof-addr sc-heap-write* (*sc-hconf P*) *P*
proof
show *P* ⊢*sc sc-empty* √ **by**(*simp add*: *sc-hconf-def*)
next
fix *h a hT*
assume *sc-typeof-addr h a* = ⌊*hT*⌋ *P* ⊢*sc h* √
thus *is-htype P hT*
by(*auto simp add*: *sc-typeof-addr-def sc-oconf-def dest*!: *sc-hconfD split*: *heapobj.split-asm*)
next
fix *h h' hT a*
assume *P* ⊢*sc h* √ (*h'*, *a*) ∈ *sc-allocate P h hT is-htype P hT*
thus *P* ⊢*sc h'* √
by(*auto simp add*: *sc-allocate-def dest*!: *new-Addr-SomeD intro*: *sc-hconf-new sc-oconf-init split*: *if-split-asm*)
next
fix *h a al T v h'*
assume *P* ⊢*sc h* √
and *sc.addr-loc-type P h a al T*
and *P*,*h* ⊢*sc v* :≤ *T*
and *sc-heap-write h a al v h'*
thus *P* ⊢*sc h'* √
by(*cases al*)(*fastforce elim*!: *sc.addr-loc-type.cases simp add*: *sc-typeof-addr-def intro*: *sc-hconf-upd-obj sc-oconf-fupd sc-hconfD sc-hconf-upd-arr sc-oconf-fupd-arr sc-oconf-fupd-arr-fields*)+
qed

interpretation *sc*: *heap-conf*
addr2thread-id
thread-id2addr

```
  sc-spurious-wakeups
  sc-empty
  sc-allocate P
  sc-typeof-addr
  sc-heap-read
  sc-heap-write
  sc-hconf P
  P
for P
by(rule sc-heap-conf)

lemma sc-heap-progress:
   heap-progress addr2thread-id thread-id2addr sc-empty (sc-allocate P) sc-typeof-addr sc-heap-read
sc-heap-write (sc-hconf P) P
proof
  fix h a al T
  assume hconf: P ⊢sc h √
    and alt: sc.addr-loc-type P h a al T
  from alt obtain arrobj where arrobj: h a = ⌊arrobj⌋
    by(auto elim!: sc.addr-loc-type.cases simp add: sc-typeof-addr-def)
  from alt show ∃ v. sc-heap-read h a al v ∧ P,h ⊢sc v :≤ T
  proof(cases)
    case (addr-loc-type-field U F fm D)
    note [simp] = ‹al = CField D F›
    show ?thesis
    proof(cases arrobj)
      case (Obj C' fs)
      with ‹sc-typeof-addr h a = ⌊U⌋› arrobj
      have [simp]: C' = class-type-of U by(auto simp add: sc-typeof-addr-def)
      from hconf arrobj Obj have P,h ⊢sc Obj (class-type-of U) fs √ by(auto dest: sc-hconfD)
      with ‹P ⊢ class-type-of U has F:T (fm) in D› obtain v
        where fs (F, D) = ⌊v⌋ P,h ⊢sc v :≤ T by(fastforce simp add: sc-fconf-def)
      thus ?thesis using Obj arrobj by(auto intro: sc-heap-read.intros)
    next
      case (Arr T' f el)
      with ‹sc-typeof-addr h a = ⌊U⌋› arrobj
      have [simp]: U = Array-type T' (length el) by(auto simp add: sc-typeof-addr-def)
      from hconf arrobj Arr have P,h ⊢sc Arr T' f el √ by(auto dest: sc-hconfD)
      from ‹P ⊢ class-type-of U has F:T (fm) in D› have [simp]: D = Object
        by(auto dest: has-field-decl-above)
      with ‹P,h ⊢sc Arr T' f el √› ‹P ⊢ class-type-of U has F:T (fm) in D›
      obtain v where f (F, Object) = ⌊v⌋ P,h ⊢sc v :≤ T
        by(fastforce simp add: sc-fconf-def)
      thus ?thesis using Arr arrobj by(auto intro: sc-heap-read.intros)
    qed
  next
    case (addr-loc-type-cell n' n)
    with arrobj obtain f el
      where [simp]: arrobj = Arr T f el
      by(cases arrobj)(auto simp add: sc-typeof-addr-def)
    from addr-loc-type-cell arrobj
    have [simp]: al = ACell n n < length el by(auto simp add: sc-typeof-addr-def)
    from hconf arrobj have P,h ⊢sc Arr T f el √ by(auto dest: sc-hconfD)
    hence P,h ⊢sc el ! n :≤ T by(fastforce)
```

```
    thus ?thesis using arrobj by(fastforce intro: sc-heap-read.intros)
  qed
next
  fix h a al T v
  assume alt: sc.addr-loc-type P h a al T
  from alt obtain arrobj where arrobj: h a = ⌊arrobj⌋
    by(auto elim!: sc.addr-loc-type.cases simp add: sc-typeof-addr-def)
  thus ∃h'. sc-heap-write h a al v h' using alt
   by(cases arrobj)(fastforce intro: sc-heap-write.intros elim!: sc.addr-loc-type.cases simp add: sc-typeof-addr-def
dest: has-field-decl-above)+
qed

interpretation sc: heap-progress
  addr2thread-id
  thread-id2addr
  sc-spurious-wakeups
  sc-empty
  sc-allocate P
  sc-typeof-addr
  sc-heap-read
  sc-heap-write
  sc-hconf P
  P
for P
by(rule sc-heap-progress)

lemma sc-heap-conf-read:
   heap-conf-read addr2thread-id thread-id2addr sc-empty (sc-allocate P) sc-typeof-addr sc-heap-read
sc-heap-write (sc-hconf P) P
proof
  fix h a al v T
  assume read: sc-heap-read h a al v
    and alt: sc.addr-loc-type P h a al T
    and hconf: P ⊢sc h √
  thus P,h ⊢sc v :≤ T
   by(auto elim!: sc-heap-read.cases sc.addr-loc-type.cases simp add: sc-typeof-addr-def)(fastforce dest!:
sc-hconfD simp add: sc-fconf-def)+
qed

interpretation sc: heap-conf-read
  addr2thread-id
  thread-id2addr
  sc-spurious-wakeups
  sc-empty
  sc-allocate P
  sc-typeof-addr
  sc-heap-read
  sc-heap-write
  sc-hconf P
  P
for P
by(rule sc-heap-conf-read)

abbreviation sc-deterministic-heap-ops :: 'm prog ⇒ bool
```

where *sc-deterministic-heap-ops* ≡ *sc.deterministic-heap-ops TYPE*('*m*)

lemma *sc-deterministic-heap-ops*: ¬ *sc-spurious-wakeups* ⟹ *sc-deterministic-heap-ops P*
by(*rule sc.deterministic-heap-opsI*)(*auto elim*: *sc-heap-read.cases sc-heap-write.cases simp add*: *sc-allocate-def*)

8.1.4 Code generation

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
sc-heap-read **.**

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
sc-heap-write **.**

lemma *eval-sc-heap-read-i-i-i-o*:
Predicate.eval (*sc-heap-read-i-i-i-o h ad al*) = *sc-heap-read h ad al*
by(*auto elim*: *sc-heap-read-i-i-i-oE intro*: *sc-heap-read-i-i-i-oI intro*!: *ext*)

lemma *eval-sc-heap-write-i-i-i-i-o*:
Predicate.eval (*sc-heap-write-i-i-i-i-o h ad al v*) = *sc-heap-write h ad al v*
by(*auto elim*: *sc-heap-write-i-i-i-i-oE intro*: *sc-heap-write-i-i-i-i-oI intro*!: *ext*)

end

theory *SC-Interp*
imports
SC
../Compiler/Correctness
../J/Deadlocked
../BV/JVMDeadlocked
begin

Do not interpret these locales, it just takes too long to generate all definitions and theorems.

lemma *sc-J-typesafe*:
J-typesafe addr2thread-id thread-id2addr sc-empty (*sc-allocate P*) *sc-typeof-addr sc-heap-read sc-heap-write* (*sc-hconf P*) *P*
by *unfold-locales*

lemma *sc-JVM-typesafe*:
JVM-typesafe addr2thread-id thread-id2addr sc-empty (*sc-allocate P*) *sc-typeof-addr sc-heap-read sc-heap-write* (*sc-hconf P*) *P*
by *unfold-locales*

lemma *compP2-compP1-convs*:
is-type (*compP2* (*compP1 P*)) = *is-type P*
is-class (*compP2* (*compP1 P*)) = *is-class P*
sc.addr-loc-type (*compP2* (*compP1 P*)) = *sc.addr-loc-type P*
sc.conf (*compP2* (*compP1 P*)) = *sc.conf P*
by(*simp-all add*: *compP2-def heap-base.compP-conf heap-base.compP-addr-loc-type fun-eq-iff split*: *addr-loc.splits*)

lemma *sc-J-JVM-conf-read*:
J-JVM-conf-read addr2thread-id thread-id2addr sc-empty (*sc-allocate P*) *sc-typeof-addr sc-heap-read*

sc-heap-write (*sc-hconf P*) *P*
apply(*rule J-JVM-conf-read.intro*)
apply(*rule J1-JVM-conf-read.intro*)
apply(*rule JVM-conf-read.intro*)
 prefer *2*
 apply(*rule JVM-heap-conf.intro*)
 apply(*rule JVM-heap-conf-base'.intro*)
 apply(*unfold compP2-def compP1-def compP-heap compP-heap-conf compP-heap-conf-read*)
 apply *unfold-locales*
done

end

8.2 Sequential consistency with efficient data structures

theory *SC-Collections*
imports
  ../Common/Conform

  ../Basic/JT-ICF
  MM
begin

hide-const (**open**) *new-Addr*
hide-fact (**open**) *new-Addr-SomeD new-Addr-SomeI*

8.2.1 Objects and Arrays

type-synonym *fields* = (*char*, (*cname*, *addr val*) *lm*) *tm*
type-synonym *array-cells* = (*nat*, *addr val*) *rbt*
type-synonym *array-fields* = (*vname*, *addr val*) *lm*

datatype *heapobj*
  = *Obj cname fields* — class instance with class name and fields
  | *Arr ty nat array-fields array-cells* — element type, size, fields and cell contents

lemma *rec-heapobj* [*simp*]: *rec-heapobj* = *case-heapobj*
by(*auto intro*!: *ext split*: *heapobj.split*)

primrec *obj-ty* :: *heapobj* ⇒ *htype*
where
  obj-ty (*Obj c f*) = *Class-type c*
| *obj-ty* (*Arr t si f e*) = *Array-type t si*

fun *is-Arr* :: *heapobj* ⇒ *bool* **where**
  is-Arr (*Obj C fs*) = *False*
| *is-Arr* (*Arr T f si el*) = *True*

lemma *is-Arr-conv*:
  is-Arr arrobj = (∃ *T si f el*. *arrobj* = *Arr T si f el*)
by(*cases arrobj*, *auto*)

lemma *is-ArrE*:
  ⟦ *is-Arr arrobj*; ⋀*T si f el*. *arrobj* = *Arr T si f el* ⟹ *thesis* ⟧ ⟹ *thesis*

⟦ ¬ *is-Arr arrobj*; ⋀*C fs*. *arrobj* = *Obj C fs* ⟹ *thesis* ⟧ ⟹ *thesis*
by(*cases arrobj*, *auto*)+

definition *init-fields* :: ((*vname* × *cname*) × *ty*) *list* ⇒ *fields*
where
init-fields FDTs ≡
foldr (λ((*F*, *D*), *T*) *fields*.
let F′ = *String.explode F*
in tm-update F′ (*lm-update D* (*default-val T*)
(*case tm-lookup F′ fields of None* ⇒ *lm-empty* () | *Some lm* ⇒ *lm*))
fields)
FDTs (*tm-empty* ())

definition *init-fields-array* :: (*vname* × *ty*) *list* ⇒ *array-fields*
where
init-fields-array ≡ *lm.to-map* ∘ *map* (λ(*F*, *T*). (*F*, *default-val T*))

definition *init-cells* :: *ty* ⇒ *nat* ⇒ *array-cells*
where *init-cells T n* = *foldl* (λ*cells i*. *rm-update i* (*default-val T*) *cells*) (*rm-empty* ()) [*0*..<*n*]

primrec — a new, blank object with default values in all fields:
blank :: *′m prog* ⇒ *htype* ⇒ *heapobj*
where
blank P (*Class-type C*) = *Obj C* (*init-fields* (*map* (λ(*FD*, (*T*, *fm*)). (*FD*, *T*)) (*TypeRel.fields P C*)))
| *blank P* (*Array-type T n*) =
Arr T n (*init-fields-array* (*map* (λ((*F*, *D*), (*T*, *fm*)). (*F*, *T*)) (*TypeRel.fields P Object*))) (*init-cells T n*)

lemma *obj-ty-blank* [*iff*]: *obj-ty* (*blank P hT*) = *hT*
by(*cases hT*) *simp-all*

8.2.2 Heap

type-synonym *heap* = (*addr*, *heapobj*) *rbt*

translations
(*type*) *heap* <= (*type*) (*nat*, *heapobj*) *rbt*

abbreviation *sc-empty* :: *heap*
where *sc-empty* ≡ *rm-empty* ()

fun *the-obj* :: *heapobj* ⇒ *cname* × *fields* **where**
the-obj (*Obj C fs*) = (*C*, *fs*)

fun *the-arr* :: *heapobj* ⇒ *ty* × *nat* × *array-fields* × *array-cells* **where**
the-arr (*Arr T si f el*) = (*T*, *si*, *f*, *el*)

abbreviation
cname-of :: *heap* ⇒ *addr* ⇒ *cname* **where**
cname-of hp a == *fst* (*the-obj* (*the* (*rm-lookup a hp*)))

definition *new-Addr* :: *heap* ⇒ *addr option*
where *new-Addr h* = *Some* (*case rm-max h* (λ-. *True*) *of None* ⇒ *0* | *Some* (*a*, -) ⇒ *a* + *1*)

definition *sc-allocate* :: *'m prog* ⇒ *heap* ⇒ *htype* ⇒ (*heap* × *addr*) *set*
where
sc-allocate P h hT =
(*case new-Addr h of None* ⇒ {}
| *Some a* ⇒ {(*rm-update a* (*blank P hT*) *h*, *a*)})

definition *sc-typeof-addr* :: *heap* ⇒ *addr* ⇒ *htype option*
where *sc-typeof-addr h a* = *map-option obj-ty* (*rm-lookup a h*)

inductive *sc-heap-read* :: *heap* ⇒ *addr* ⇒ *addr-loc* ⇒ *addr val* ⇒ *bool*
for *h* :: *heap* **and** *a* :: *addr*
where
Obj: ⟦ *rm-lookup a h* = ⌊*Obj C fs*⌋; *tm-lookup* (*String.explode F*) *fs* = ⌊*fs'*⌋; *lm-lookup D fs'* = ⌊*v*⌋ ⟧
⟹ *sc-heap-read h a* (*CField D F*) *v*
| *Arr*: ⟦ *rm-lookup a h* = ⌊*Arr T si f el*⌋; *n* < *si* ⟧ ⟹ *sc-heap-read h a* (*ACell n*) (*the* (*rm-lookup n el*))
| *ArrObj*: ⟦ *rm-lookup a h* = ⌊*Arr T si f el*⌋; *lm-lookup F f* = ⌊*v*⌋ ⟧ ⟹ *sc-heap-read h a* (*CField Object F*) *v*

hide-fact (**open**) *Obj Arr ArrObj*

inductive-cases *sc-heap-read-cases* [*elim*!]:
sc-heap-read h a (*CField C F*) *v*
sc-heap-read h a (*ACell n*) *v*

inductive *sc-heap-write* :: *heap* ⇒ *addr* ⇒ *addr-loc* ⇒ *addr val* ⇒ *heap* ⇒ *bool*
for *h* :: *heap* **and** *a* :: *addr*
where
Obj:
⟦ *rm-lookup a h* = ⌊*Obj C fs*⌋; *F'* = *String.explode F*;
h' = *rm-update a* (*Obj C* (*tm-update F'* (*lm-update D v* (*case tm-lookup* (*String.explode F*) *fs of None* ⇒ *lm-empty* () | *Some fs'* ⇒ *fs'*)) *fs*)) *h* ⟧
⟹ *sc-heap-write h a* (*CField D F*) *v h'*

| *Arr*:
⟦ *rm-lookup a h* = ⌊*Arr T si f el*⌋; *h'* = *rm-update a* (*Arr T si f* (*rm-update n v el*)) *h* ⟧
⟹ *sc-heap-write h a* (*ACell n*) *v h'*

| *ArrObj*:
⟦ *rm-lookup a h* = ⌊*Arr T si f el*⌋; *h'* = *rm-update a* (*Arr T si* (*lm-update F v f*) *el*) *h* ⟧
⟹ *sc-heap-write h a* (*CField Object F*) *v h'*

hide-fact (**open**) *Obj Arr ArrObj*

inductive-cases *sc-heap-write-cases* [*elim*!]:
sc-heap-write h a (*CField C F*) *v h'*
sc-heap-write h a (*ACell n*) *v h'*

consts *sc-spurious-wakeups* :: *bool*

lemma *new-Addr-SomeD*: *new-Addr h* = ⌊*a*⌋ ⟹ *rm-lookup a h* = *None*
apply(*simp add*: *new-Addr-def*)
apply(*drule rm.max-None*[*OF rm.invar*])
apply(*simp add*: *rm.lookup-correct rel-of-def*)

apply(*clarsimp simp add*: *rm.lookup-correct*)
apply(*frule rm.max-Some*[*OF rm.invar*])
apply(*clarsimp simp add*: *rel-of-def*)
apply(*hypsubst-thin*)
apply(*rule ccontr*)
apply(*clarsimp*)
apply(*drule-tac k'=Suc a* **in** *rm.max-Some*(*2*)[*OF rm.invar*])
apply(*auto simp add*: *rel-of-def*)
done

interpretation *sc*:
heap-base
addr2thread-id
thread-id2addr
sc-spurious-wakeups
sc-empty
sc-allocate P
sc-typeof-addr
sc-heap-read
sc-heap-write
for *P* **.**

Translate notation from *heap-base*

abbreviation *sc-preallocated* :: *'m prog* ⇒ *heap* ⇒ *bool*
where *sc-preallocated* == *sc.preallocated TYPE*('*m*)

abbreviation *sc-start-tid* :: *'md prog* ⇒ *thread-id*
where *sc-start-tid* ≡ *sc.start-tid TYPE*('*md*)

abbreviation *sc-start-heap-ok* :: *'m prog* ⇒ *bool*
where *sc-start-heap-ok* ≡ *sc.start-heap-ok TYPE*('*m*)

abbreviation *sc-start-heap* :: *'m prog* ⇒ *heap*
where *sc-start-heap* ≡ *sc.start-heap TYPE*('*m*)

abbreviation *sc-start-state* ::
(*cname* ⇒ *mname* ⇒ *ty list* ⇒ *ty* ⇒ *'m* ⇒ *addr val list* ⇒ *'x*)
⇒ *'m prog* ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒ (*addr*, *thread-id*, *'x*, *heap*, *addr*) *state*
where
sc-start-state f P ≡ *sc.start-state TYPE*('*m*) *P f P*

abbreviation *sc-wf-start-state* :: *'m prog* ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒ *bool*
where *sc-wf-start-state P* ≡ *sc.wf-start-state TYPE*('*m*) *P P*

notation *sc.conf* (‹-,- ⊢*sc* - :≤ -› [*51*,*51*,*51*,*51*] *50*)
notation *sc.confs* (‹-,- ⊢*sc* - [:≤] -› [*51*,*51*,*51*,*51*] *50*)
notation *sc.hext* (‹- ⊴*sc* -› [*51*,*51*] *50*)

lemma *new-Addr-SomeI*: ∃ *a*. *new-Addr h* = *Some a*
by(*simp add*: *new-Addr-def*)

lemma *sc-start-heap-ok*: *sc-start-heap-ok P*
by(*simp add*: *sc.start-heap-ok-def sc.start-heap-data-def initialization-list-def sc.create-initial-object-simps sc-allocate-def case-option-conv-if new-Addr-SomeI sys-xcpts-list-def del*: *blank.simps split del*: *option.split*

if-split)

lemma *sc-wf-start-state-iff*:
sc-wf-start-state P C M vs $\longleftrightarrow$ (∃ *Ts T meth D*. *P* ⊢ *C sees M*:*Ts*→*T* = ⌊*meth*⌋ *in D* ∧ *P*,*sc-start-heap P* ⊢*sc vs* [:≤] *Ts*)
by(*simp add*: *sc.wf-start-state.simps sc-start-heap-ok*)

lemma *sc-heap*:
heap addr2thread-id thread-id2addr (*sc-allocate P*) *sc-typeof-addr sc-heap-write P*
proof
fix *h' a h hT*
assume (*h'*, *a*) ∈ *sc-allocate P h hT*
thus *sc-typeof-addr h' a* = ⌊*hT*⌋
by(*auto simp add*: *sc-allocate-def sc-typeof-addr-def rm.lookup-correct rm.update-correct dest*: *new-Addr-SomeD split*: *if-split-asm*)
next
fix *h h' hT a*
assume (*h'*, *a*) ∈ *sc-allocate P h hT*
from *this*[*symmetric*] **show** *h* ≤*sc h'*
by(*fastforce simp add*: *sc-allocate-def sc-typeof-addr-def sc.hext-def rm.lookup-correct rm.update-correct intro*!: *map-leI dest*: *new-Addr-SomeD*)
next
fix *h a al v h'*
assume *sc-heap-write h a al v h'*
thus *h* ≤*sc h'*
by(*cases al*)(*auto intro*!: *sc.hextI simp add*: *sc-typeof-addr-def rm.lookup-correct rm.update-correct*)
qed *simp*

interpretation *sc*:
heap
addr2thread-id
thread-id2addr
sc-spurious-wakeups
sc-empty
sc-allocate P
sc-typeof-addr
sc-heap-read
sc-heap-write
P
for *P* **by**(*rule sc-heap*)

declare *sc.typeof-addr-thread-id2-addr-addr2thread-id* [*simp del*]

lemma *sc-hext-new*:
rm-lookup a h = *None* ⟹ *h* ≤*sc rm-update a arrobj h*
by(*rule sc.hextI*)(*auto simp add*: *sc-typeof-addr-def rm.lookup-correct rm.update-correct dest*!: *new-Addr-SomeD*)

lemma *sc-hext-upd-obj*: *rm-lookup a h* = *Some* (*Obj C fs*) ⟹ *h* ≤*sc rm-update a* (*Obj C fs'*) *h*
by(*rule sc.hextI*)(*auto simp*:*fun-upd-apply sc-typeof-addr-def rm.lookup-correct rm.update-correct*)

lemma *sc-hext-upd-arr*: ⟦ *rm-lookup a h* = *Some* (*Arr T si f e*) ⟧ ⟹ *h* ≤*sc rm-update a* (*Arr T si f' e'*) *h*
by(*rule sc.hextI*)(*auto simp*:*fun-upd-apply sc-typeof-addr-def rm.lookup-correct rm.update-correct*)

8.2.3 Conformance

definition *sc-oconf* :: *'m prog* ⇒ *heap* ⇒ *heapobj* ⇒ *bool* (‹-,- ⊢*sc* - √› [*51*,*51*,*51*] *50*)
where
P,*h* ⊢*sc obj* √ ≡
(*case obj of*
Obj C fs ⇒
is-class P C ∧
(∀ *F D T fm*. *P* ⊢ *C has F*:*T* (*fm*) *in D* ⟶
(∃ *fs' v*. *tm-α fs* (*String.explode F*) = *Some fs'* ∧ *lm-α fs' D* = *Some v* ∧ *P*,*h* ⊢*sc v* :≤ *T*))
| *Arr T si f el* ⇒
is-type P (*T*⌊⌉) ∧ (∀ *n*. *n* < *si* ⟶ (∃ *v*. *rm-α el n* = *Some v* ∧ *P*,*h* ⊢*sc v* :≤ *T*)) ∧
(∀ *F T fm*. *P* ⊢ *Object has F*:*T* (*fm*) *in Object* ⟶ (∃ *v*. *lm-lookup F f* = *Some v* ∧ *P*,*h* ⊢*sc v* :≤ *T*)))

definition *sc-hconf* :: *'m prog* ⇒ *heap* ⇒ *bool* (‹- ⊢*sc* - √› [*51*,*51*] *50*)
where *P* ⊢*sc h* √ ⟷ (∀ *a obj*. *rm-α h a* = *Some obj* ⟶ *P*,*h* ⊢*sc obj* √)

interpretation *sc*:
heap-conf-base
addr2thread-id
thread-id2addr
sc-spurious-wakeups
sc-empty
sc-allocate P
sc-typeof-addr
sc-heap-read
sc-heap-write
sc-hconf P
P
for *P*
.

lemma *sc-conf-upd-obj*: *rm-lookup a h* = *Some*(*Obj C fs*) ⟹ (*P*,*rm-update a* (*Obj C fs'*) *h* ⊢*sc x* :≤ *T*) = (*P*,*h* ⊢*sc x* :≤ *T*)
apply (*unfold sc.conf-def*)
apply (*rule val.induct*)
apply (*auto simp*:*fun-upd-apply*)
apply (*auto simp add*: *sc-typeof-addr-def rm.lookup-correct rm.update-correct split*: *if-split-asm*)
done

lemma *sc-conf-upd-arr*:
rm-lookup a h = *Some*(*Arr T si f el*) ⟹ (*P*,*rm-update a* (*Arr T si f' el'*) *h* ⊢*sc x* :≤ *T'*) = (*P*,*h* ⊢*sc x* :≤ *T'*)
apply(*unfold sc.conf-def*)
apply (*rule val.induct*)
apply (*auto simp*:*fun-upd-apply*)
apply(*auto simp add*: *sc-typeof-addr-def rm.lookup-correct rm.update-correct split*: *if-split-asm*)
done

lemma *sc-oconf-hext*: *P*,*h* ⊢*sc obj* √ ⟹ *h* ⊴*sc h'* ⟹ *P*,*h'* ⊢*sc obj* √
unfolding *sc-oconf-def*
by(*fastforce split*: *heapobj.split elim*: *sc.conf-hext*)

lemma *map-of-fields-init-fields*:
assumes *map-of FDTs* (*F*, *D*) = ⌊(*T*, *fm*)⌋
shows ∃*fs′ v*. *tm-α* (*init-fields* (*map* (λ(*FD*, (*T*, *fm*)). (*FD*, *T*)) *FDTs*)) (*String.explode F*) = ⌊*fs′*⌋ ∧ *lm-α fs′ D* = ⌊*v*⌋ ∧ *sc.conf P h v T*
using *assms*
by(*induct FDTs*)(*auto simp add*: *tm.lookup-correct tm.update-correct lm.update-correct init-fields-def String.explode-inject*)

lemma *sc-oconf-init-fields*:
assumes *P* ⊢ *C has-fields FDTs*
shows *P*,*h* ⊢*sc* (*Obj C* (*init-fields* (*map* (λ(*FD*, (*T*, *fm*)). (*FD*, *T*)) *FDTs*))) √
using *assms has-fields-is-class*[*OF assms*] *map-of-fields-init-fields*[*of FDTs*]
by(*fastforce simp add*: *has-field-def sc-oconf-def dest*: *has-fields-fun*)

lemma *sc-oconf-init-arr*:
assumes *type*: *is-type P* (*T*⌊⌋)
shows *P*,*h* ⊢*sc Arr T n* (*init-fields-array* (*map* (λ((*F*, *D*), (*T*, *fm*)). (*F*, *T*)) (*TypeRel.fields P Object*))) (*init-cells T n*) √
proof −
{ fix *n′*
assume *n′* < *n*
{ fix *rm* **and** *k* :: *nat*
assume ∀*i*<*k*. ∃*v*. *rm-α rm i* = ⌊*v*⌋ ∧ *sc.conf P h v T*
with ‹*n′* < *n*› **have** ∃*v*. *rm-α* (*foldl* (λ*cells i*. *rm-update i* (*default-val T*) *cells*) *rm* [*k*..<*n*]) *n′* = ⌊*v*⌋ ∧ *sc.conf P h v T*
by(*induct m*≡*n*−*k arbitrary*: *n k rm*)(*auto simp add*: *rm.update-correct upt-conv-Cons type*)
}
from *this*[*of 0 rm-empty* ()]
have ∃*v*. *rm-α* (*foldl* (λ*cells i*. *rm-update i* (*default-val T*) *cells*) (*rm-empty* ()) [*0*..<*n*]) *n′* = ⌊*v*⌋ ∧ *sc.conf P h v T* **by** *simp*
}
moreover
{ fix *F T fm*
assume *P* ⊢ *Object has F*:*T* (*fm*) *in Object*
then obtain *FDTs* **where** *has*: *P* ⊢ *Object has-fields FDTs*
and *FDTs*: *map-of FDTs* (*F*, *Object*) = ⌊(*T*, *fm*)⌋
by(*auto simp add*: *has-field-def*)
from *has* **have** *snd* ‘ *fst* ‘ *set FDTs* ⊆ {*Object*} **by**(*rule Object-has-fields-Object*)
with *FDTs* **have** *map-of* (*map* ((λ(*F*, *T*). (*F*, *default-val T*)) ∘ (λ((*F*, *D*), *T*, *fm*). (*F*, *T*))) *FDTs*) *F* = ⌊*default-val T*⌋
by(*induct FDTs*) *auto*
with *has FDTs*
have ∃*v*. *lm-lookup F* (*init-fields-array* (*map* (λ((*F*, *D*), *T*, *fm*). (*F*, *T*)) (*TypeRel.fields P Object*))) = ⌊*v*⌋ ∧
sc.conf P h v T
by(*auto simp add*: *init-fields-array-def lm-correct has-field-def*)
}
ultimately show *?thesis* **using** *type* **by**(*auto simp add*: *sc-oconf-def init-cells-def*)
qed

lemma *sc-oconf-fupd* [*intro?*]:
⟦ *P* ⊢ *C has F*:*T* (*fm*) *in D*; *P*,*h* ⊢*sc v* :≤ *T*; *P*,*h* ⊢*sc* (*Obj C fs*) √;
fs′ = (*case tm-lookup* (*String.explode F*) *fs of None* ⇒ *lm-empty* () | *Some fs′* ⇒ *fs′*) ⟧
⟹ *P*,*h* ⊢*sc* (*Obj C* (*tm-update* (*String.explode F*) (*lm-update D v fs′*) *fs*)) √

unfolding *sc-oconf-def has-field-def*
apply(*auto dest*: *has-fields-fun simp add*: *lm.update-correct tm.update-correct tm.lookup-correct String.explode-inject*)
apply(*drule* (*1*) *has-fields-fun*, *fastforce*)
apply(*drule* (*1*) *has-fields-fun*, *fastforce*)
done

lemma *sc-oconf-fupd-arr* [*intro?*]:
⟦ *P*,*h* ⊢*sc v* :≤ *T*; *P*,*h* ⊢*sc* (*Arr T si f el*) √ ⟧
⟹ *P*,*h* ⊢*sc* (*Arr T si f* (*rm-update i v el*)) √
unfolding *sc-oconf-def*
by(*auto simp add*: *rm.update-correct*)

lemma *sc-oconf-fupd-arr-fields*:
⟦ *P* ⊢ *Object has F*:*T* (*fm*) *in Object*; *P*,*h* ⊢*sc v* :≤ *T*; *P*,*h* ⊢*sc* (*Arr T*′ *si f el*) √ ⟧
⟹ *P*,*h* ⊢*sc* (*Arr T*′ *si* (*lm-update F v f*) *el*) √
unfolding *sc-oconf-def* **by**(*auto dest*: *has-field-fun simp add*: *lm-correct*)

lemma *sc-oconf-new*: ⟦ *P*,*h* ⊢*sc obj* √; *rm-lookup a h* = *None* ⟧ ⟹ *P*,*rm-update a arrobj h* ⊢*sc obj* √
by(*erule sc-oconf-hext*)(*rule sc-hext-new*)

lemmas *sc-oconf-upd-obj* = *sc-oconf-hext* [*OF* - *sc-hext-upd-obj*]

lemma *sc-oconf-upd-arr*:
assumes *P*,*h* ⊢*sc obj* √
and *ha*: *rm-lookup a h* = ⌊*Arr T si f el*⌋
shows *P*,*rm-update a* (*Arr T si f*′ *el*′) *h* ⊢*sc obj* √
using *assms*
by(*fastforce simp add*: *sc-oconf-def sc-conf-upd-arr*[*OF ha*] *split*: *heapobj.split*)

lemma *sc-oconf-blank*: *is-htype P hT* ⟹ *P*,*h* ⊢*sc blank P hT* √
apply(*cases hT*)
apply(*fastforce dest*: *map-of-fields-init-fields simp add*: *has-field-def sc-oconf-def*)
by(*auto intro*: *sc-oconf-init-arr*)

lemma *sc-hconfD*: ⟦ *P* ⊢*sc h* √; *rm-lookup a h* = *Some obj* ⟧ ⟹ *P*,*h* ⊢*sc obj* √
unfolding *sc-hconf-def* **by**(*auto simp add*: *rm.lookup-correct*)

lemmas *sc-preallocated-new* = *sc.preallocated-hext*[*OF* - *sc-hext-new*]
lemmas *sc-preallocated-upd-obj* = *sc.preallocated-hext* [*OF* - *sc-hext-upd-obj*]
lemmas *sc-preallocated-upd-arr* = *sc.preallocated-hext* [*OF* - *sc-hext-upd-arr*]

lemma *sc-hconf-new*: ⟦ *P* ⊢*sc h* √; *rm-lookup a h* = *None*; *P*,*h* ⊢*sc obj* √ ⟧ ⟹ *P* ⊢*sc rm-update a obj h* √
unfolding *sc-hconf-def*
by(*auto intro*: *sc-oconf-new simp add*: *rm.lookup-correct rm.update-correct*)

lemma *sc-hconf-upd-obj*: ⟦ *P* ⊢*sc h* √; *rm-lookup a h* = *Some* (*Obj C fs*); *P*,*h* ⊢*sc* (*Obj C fs*′) √ ⟧
⟹ *P* ⊢*sc rm-update a* (*Obj C fs*′) *h* √
unfolding *sc-hconf-def*
by(*auto intro*: *sc-oconf-upd-obj simp add*: *rm.lookup-correct rm.update-correct*)

lemma *sc-hconf-upd-arr*: ⟦ *P* ⊢*sc h* √; *rm-lookup a h* = *Some*(*Arr T si f el*); *P*,*h* ⊢*sc* (*Arr T si f*′ *el*′) √ ⟧ ⟹ *P* ⊢*sc rm-update a* (*Arr T si f*′ *el*′) *h* √

unfolding *sc-hconf-def*
by(*auto intro*: *sc-oconf-upd-arr simp add*: *rm.lookup-correct rm.update-correct*)

lemma *sc-heap-conf*:
heap-conf addr2thread-id thread-id2addr sc-empty (*sc-allocate P*) *sc-typeof-addr sc-heap-write* (*sc-hconf P*) *P*
proof
show *P* ⊢*sc sc-empty* √ **by**(*simp add*: *sc-hconf-def rm.empty-correct*)
next
fix *h a hT*
assume *sc-typeof-addr h a* = ⌊*hT*⌋ *P* ⊢*sc h* √
thus *is-htype P hT*
by(*auto simp add*: *sc-typeof-addr-def sc-oconf-def dest*!: *sc-hconfD split*: *heapobj.split-asm*)
next
fix *h′ hT h a*
assume *P* ⊢*sc h* √ (*h′*, *a*) ∈ *sc-allocate P h hT is-htype P hT*
thus *P* ⊢*sc h′* √
by(*auto simp add*: *sc-allocate-def dest*!: *new-Addr-SomeD intro*: *sc-hconf-new sc-oconf-blank split*: *if-split-asm*)
next
fix *h a al T v h′*
assume *P* ⊢*sc h* √
and *sc.addr-loc-type P h a al T*
and *P*,*h* ⊢*sc v* :≤ *T*
and *sc-heap-write h a al v h′*
thus *P* ⊢*sc h′* √
by(*cases al*)(*fastforce elim*!: *sc.addr-loc-type.cases simp add*: *sc-typeof-addr-def intro*: *sc-hconf-upd-obj sc-oconf-fupd sc-hconfD sc-hconf-upd-arr sc-oconf-fupd-arr sc-oconf-fupd-arr-fields*)+
qed

interpretation *sc*:
heap-conf
addr2thread-id
thread-id2addr
sc-spurious-wakeups
sc-empty
sc-allocate P
sc-typeof-addr
sc-heap-read
sc-heap-write
sc-hconf P
P
for *P*
by(*rule sc-heap-conf*)

lemma *sc-heap-progress*:
heap-progress addr2thread-id thread-id2addr sc-empty (*sc-allocate P*) *sc-typeof-addr sc-heap-read sc-heap-write* (*sc-hconf P*) *P*
proof
fix *h a al T*
assume *hconf*: *P* ⊢*sc h* √
and *alt*: *sc.addr-loc-type P h a al T*
from *alt* **obtain** *arrobj* **where** *arrobj*: *rm-lookup a h* = ⌊*arrobj*⌋
by(*auto elim*!: *sc.addr-loc-type.cases simp add*: *sc-typeof-addr-def*)

```
  from alt show ∃ v. sc-heap-read h a al v ∧ P,h ⊢sc v :≤ T
  proof(cases)
    case (addr-loc-type-field U F fm D)
    note [simp] = ‹al = CField D F›
    show ?thesis
    proof(cases arrobj)
      case (Obj C' fs)
      with ‹sc-typeof-addr h a = ⌊U⌋› arrobj
      have [simp]: C' = class-type-of U by(auto simp add: sc-typeof-addr-def)
      from hconf arrobj Obj have P,h ⊢sc Obj (class-type-of U) fs √ by(auto dest: sc-hconfD)
      with ‹P ⊢ class-type-of U has F:T (fm) in D› obtain fs' v
      where tm-lookup (String.explode F) fs = ⌊fs'⌋ lm-lookup D fs' = ⌊v⌋ P,h ⊢sc v :≤ T
      by(fastforce simp add: sc-oconf-def tm.lookup-correct lm.lookup-correct)
      thus ?thesis using Obj arrobj by(auto intro: sc-heap-read.intros)
    next
      case (Arr T' si f el)
      with ‹sc-typeof-addr h a = ⌊U⌋› arrobj
      have [simp]: U = Array-type T' si by(auto simp add: sc-typeof-addr-def)
      from hconf arrobj Arr have P,h ⊢sc Arr T' si f el √ by(auto dest: sc-hconfD)
      from ‹P ⊢ class-type-of U has F:T (fm) in D› have [simp]: D = Object
        by(auto dest: has-field-decl-above)
      with ‹P,h ⊢sc Arr T' si f el √› ‹P ⊢ class-type-of U has F:T (fm) in D›
      obtain v where lm-lookup F f = ⌊v⌋ P,h ⊢sc v :≤ T
        by(fastforce simp add: sc-oconf-def)
      thus ?thesis using Arr arrobj by(auto intro: sc-heap-read.intros)
    qed
  next
    case (addr-loc-type-cell n' n)
    with arrobj obtain si f el
      where [simp]: arrobj = Arr T si f el
      by(cases arrobj)(auto simp add: sc-typeof-addr-def)
    from addr-loc-type-cell arrobj
    have [simp]: al = ACell n and n: n < si by(auto simp add: sc-typeof-addr-def)
    from hconf arrobj have P,h ⊢sc Arr T si f el √ by(auto dest: sc-hconfD)
    with n obtain v where rm-lookup n el = ⌊v⌋ P,h ⊢sc v :≤ T
      by(fastforce simp add: sc-oconf-def rm.lookup-correct)
    thus ?thesis using arrobj n by(fastforce intro: sc-heap-read.intros)
  qed
next
  fix h a al T v
  assume alt: sc.addr-loc-type P h a al T
  from alt obtain arrobj where arrobj: rm-lookup a h = ⌊arrobj⌋
    by(auto elim!: sc.addr-loc-type.cases simp add: sc-typeof-addr-def)
  thus ∃ h'. sc-heap-write h a al v h' using alt
   by(cases arrobj)(fastforce intro: sc-heap-write.intros elim!: sc.addr-loc-type.cases simp add: sc-typeof-addr-def
dest: has-field-decl-above)+
qed

interpretation sc:
  heap-progress
    addr2thread-id
    thread-id2addr
    sc-spurious-wakeups
    sc-empty
```

```
    sc-allocate P
    sc-typeof-addr
    sc-heap-read
    sc-heap-write
    sc-hconf P
    P
  for P
by(rule sc-heap-progress)

lemma sc-heap-conf-read:
   heap-conf-read addr2thread-id thread-id2addr sc-empty (sc-allocate P) sc-typeof-addr sc-heap-read
sc-heap-write (sc-hconf P) P
proof
  fix h a al v T
  assume read: sc-heap-read h a al v
    and alt: sc.addr-loc-type P h a al T
    and hconf: P ⊢sc h √
  thus P,h ⊢sc v :≤ T
    apply(auto elim!: sc-heap-read.cases sc.addr-loc-type.cases simp add: sc-typeof-addr-def)
   apply(fastforce dest!: sc-hconfD simp add: sc-oconf-def tm.lookup-correct lm.lookup-correct rm.lookup-correct)+
    done
qed

interpretation sc:
  heap-conf-read
    addr2thread-id
    thread-id2addr
    sc-spurious-wakeups
    sc-empty
    sc-allocate P
    sc-typeof-addr
    sc-heap-read
    sc-heap-write
    sc-hconf P
    P
  for P
by(rule sc-heap-conf-read)

abbreviation sc-deterministic-heap-ops :: 'm prog ⇒ bool
where sc-deterministic-heap-ops ≡ sc.deterministic-heap-ops TYPE('m)

lemma sc-deterministic-heap-ops: ¬ sc-spurious-wakeups ⟹ sc-deterministic-heap-ops P
by(rule sc.deterministic-heap-opsI)(auto elim: sc-heap-read.cases sc-heap-write.cases simp add: sc-allocate-def)
```

8.2.4 Code generation

```
code-pred
  (modes: i ⇒ i ⇒ i ⇒ i ⇒ bool, i ⇒ i ⇒ i ⇒ o ⇒ bool)
  sc-heap-read .

code-pred
  (modes: i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool, i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool)
  sc-heap-write .
```

lemma *eval-sc-heap-read-i-i-i-o*:
Predicate.eval (*sc-heap-read-i-i-i-o h ad al*) = *sc-heap-read h ad al*
by(*auto elim*: *sc-heap-read-i-i-i-oE intro*: *sc-heap-read-i-i-i-oI intro*!: *ext*)

lemma *eval-sc-heap-write-i-i-i-i-o*:
Predicate.eval (*sc-heap-write-i-i-i-i-o h ad al v*) = *sc-heap-write h ad al v*
by(*auto elim*: *sc-heap-write-i-i-i-i-oE intro*: *sc-heap-write-i-i-i-i-oI intro*!: *ext*)

end

8.3 Orders as predicates

theory *Orders*
imports
Main
begin

8.3.1 Preliminaries

transfer *refl-on* et al. from *HOL.Relation* to predicates

abbreviation *refl-onP* :: *'a set* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool*
where *refl-onP A r* ≡ *refl-on A* {(*x*, *y*). *r x y*}

abbreviation *reflP* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool*
where *reflP* == *refl-onP UNIV*

abbreviation *symP* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool*
where *symP r* ≡ *sym* {(*x*, *y*). *r x y*}

abbreviation *total-onP* :: *'a set* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool*
where *total-onP A r* ≡ *total-on A* {(*x*, *y*). *r x y*}

abbreviation *irreflP* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool*
where *irreflP r* == *irrefl* {(*x*, *y*). *r x y*}

definition *irreflclp* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a* ⇒ *'a* ⇒ *bool* (‹-$^{\neq\neq}$› [*1000*] *1000*)
where $r^{\neq\neq}$ *a b* = (*r a b* ∧ *a* ≠ *b*)

definition *porder-on* :: *'a set* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool*
where *porder-on A r* ⟷ *refl-onP A r* ∧ *transp r* ∧ *antisymp r*

definition *torder-on* :: *'a set* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool*
where *torder-on A r* ⟷ *porder-on A r* ∧ *total-onP A r*

definition *order-consistent* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool*
where *order-consistent r s* ⟷ (∀ *a a'*. *r a a'* ∧ *s a' a* ⟶ *a* = *a'*)

definition *restrictP* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a set* ⇒ *'a* ⇒ *'a* ⇒ *bool* (**infixl** ‹|`› *110*)
where (*r* |` *A*) *a b* ⟷ *r a b* ∧ *a* ∈ *A* ∧ *b* ∈ *A*

definition *inv-imageP* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'b* ⇒ *'a*) ⇒ *'b* ⇒ *'b* ⇒ *bool*
where [*iff*]: *inv-imageP r f a b* ⟷ *r* (*f a*) (*f b*)

lemma *refl-onPI*: $(\bigwedge a\ a'.\ r\ a\ a' \Longrightarrow a \in A \wedge a' \in A) \Longrightarrow (\bigwedge a.\ a : A \Longrightarrow r\ a\ a) \Longrightarrow$ *refl-onP A r*
by(*rule refl-onI*)(*auto*)

lemma *refl-onPD*: *refl-onP A r* ==> *a* : *A* ==> *r a a*
by(*drule* (*1*) *refl-onD*)(*simp*)

lemma *refl-onPD1*: *refl-onP A r* ==> *r a b* ==> *a* : *A*
by(*erule refl-onD1*)(*simp*)

lemma *refl-onPD2*: *refl-onP A r* ==> *r a b* ==> *b* : *A*
by(*erule refl-onD2*)(*simp*)

lemma *refl-onP-Int*: *refl-onP A r* ==> *refl-onP B s* ==> *refl-onP* $(A \cap B)$ $(\lambda a\ a'.\ r\ a\ a' \wedge s\ a\ a')$
by(*drule* (*1*) *refl-on-Int*)(*simp add*: *split-def inf-fun-def inf-set-def*)

lemma *refl-onP-Un*: *refl-onP A r* ==> *refl-onP B s* ==> *refl-onP* $(A \cup B)$ $(\lambda a\ a'.\ r\ a\ a' \vee s\ a\ a')$
by(*drule* (*1*) *refl-on-Un*)(*simp add*: *split-def sup-fun-def sup-set-def*)

lemma *refl-onP-empty*[*simp*]: *refl-onP* $\{\}$ $(\lambda a\ a'.\ False)$
unfolding *split-def* **by** *simp*

lemma *refl-onP-tranclp*:
  assumes *refl-onP A r*
  shows *refl-onP A r*$\hat{}$++
proof(*rule refl-onPI*)
  fix *a a'*
  assume *r*$\hat{}$++ *a a'*
  thus $a \in A \wedge a' \in A$
    by(*induct*)(*blast intro*: *refl-onPD1*[*OF assms*] *refl-onPD2*[*OF assms*])+
next
  fix *a*
  assume $a \in A$
  from *refl-onPD*[*OF assms this*] **show** *r*$\hat{}$++ *a a* **..**
qed

lemma *irreflPI*: $(\bigwedge a.\ \neg\ r\ a\ a) \Longrightarrow$ *irreflP r*
unfolding *irrefl-def* **by** *blast*

lemma *irreflPE*:
  assumes *irreflP r*
  obtains $\forall\, a.\ \neg\ r\ a\ a$
using *assms* **unfolding** *irrefl-def* **by** *blast*

lemma *irreflPD*: $[\![$ *irreflP r*; *r a a* $]\!] \Longrightarrow$ *False*
unfolding *irrefl-def* **by** *blast*

lemma *irreflclpD*:
  $r^{\neq\neq}\ a\ b \Longrightarrow r\ a\ b \wedge a \neq b$
by(*simp add*: *irreflclp-def*)

lemma *irreflclpI* [*intro*!]:
  $[\![$ *r a b*; $a \neq b$ $]\!] \Longrightarrow r^{\neq\neq}\ a\ b$
by(*simp add*: *irreflclp-def*)

lemma *irreflclpE* [*elim*!]:
assumes $r^{\neq\neq}$ *a b*
obtains *r a b a* $\neq$ *b*
using *assms* **by**(*simp add*: *irreflclp-def*)

lemma *transPI*: $(\bigwedge x\ y\ z.\ [\![\ r\ x\ y;\ r\ y\ z\ ]\!] \Longrightarrow r\ x\ z) \Longrightarrow$ *transp r*
by (*fact transpI*)

lemma *transPD*: $[\![$*transp r*; *r x y*; *r y z* $]\!] \Longrightarrow$ *r x z*
by (*fact transpD*)

lemma *transP-tranclp*: *transp r*$\hat{}$++
by (*fact trans-trancl* [*to-pred*])

lemma *antisymPI*: $(\bigwedge x\ y.\ [\![\ r\ x\ y;\ r\ y\ x\ ]\!] \Longrightarrow x = y) \Longrightarrow$ *antisymp r*
by (*fact antisympI*)

lemma *antisymPD*: $[\![$ *antisymp r*; *r a b*; *r b a* $]\!] \Longrightarrow a = b$
by (*fact antisympD*)

lemma *antisym-subset*:
$[\![$ *antisymp r*; $\bigwedge a\ a'.\ s\ a\ a' \Longrightarrow r\ a\ a'\ ]\!] \Longrightarrow$ *antisymp s*
by (*blast intro*: *antisymp-less-eq* [*of s r*])

lemma *symPI*: $(\bigwedge x\ y.\ r\ x\ y \Longrightarrow r\ y\ x) \Longrightarrow$ *symP r*
by(*rule symI*)(*blast*)

lemma *symPD*: $[\![$ *symP r*; *r x y* $]\!] \Longrightarrow$ *r y x*
by(*blast dest*: *symD*)

8.3.2 Easy properties

lemma *porder-onI*:
$[\![$ *refl-onP A r*; *antisymp r*; *transp r* $]\!] \Longrightarrow$ *porder-on A r*
unfolding *porder-on-def* **by** *blast*

lemma *porder-onE*:
assumes *porder-on A r*
obtains *refl-onP A r antisymp r transp r*
using *assms* **unfolding** *porder-on-def* **by** *blast*

lemma *torder-onI*:
$[\![$ *porder-on A r*; *total-onP A r* $]\!] \Longrightarrow$ *torder-on A r*
unfolding *torder-on-def* **by** *blast*

lemma *torder-onE*:
assumes *torder-on A r*
obtains *porder-on A r total-onP A r*
using *assms* **unfolding** *torder-on-def* **by** *blast*

lemma *total-onI*:
$(\bigwedge x\ y.\ [\![\ x \in A;\ y \in A\ ]\!] \Longrightarrow (x, y) \in r \vee x = y \vee (y, x) \in r) \Longrightarrow$ *total-on A r*
unfolding *total-on-def* **by** *blast*

lemma *total-onPI*:
(⋀*x y*. ⟦ *x* ∈ *A*; *y* ∈ *A* ⟧ ⟹ *r x y* ∨ *x* = *y* ∨ *r y x*) ⟹ *total-onP A r*
by(*rule total-onI*) *simp*

lemma *total-onD*:
⟦ *total-on A r*; *x* ∈ *A*; *y* ∈ *A* ⟧ ⟹ (*x*, *y*) ∈ *r* ∨ *x* = *y* ∨ (*y*, *x*) ∈ *r*
unfolding *total-on-def* **by** *blast*

lemma *total-onPD*:
⟦ *total-onP A r*; *x* ∈ *A*; *y* ∈ *A* ⟧ ⟹ *r x y* ∨ *x* = *y* ∨ *r y x*
by(*drule* (*2*) *total-onD*) *blast*

8.3.3 Order consistency

lemma *order-consistentI*:
(⋀*a a'*. ⟦ *r a a'*; *s a' a* ⟧ ⟹ *a* = *a'*) ⟹ *order-consistent r s*
unfolding *order-consistent-def* **by** *blast*

lemma *order-consistentD*:
⟦ *order-consistent r s*; *r a a'*; *s a' a* ⟧ ⟹ *a* = *a'*
unfolding *order-consistent-def* **by** *blast*

lemma *order-consistent-subset*:
⟦ *order-consistent r s*; ⋀*a a'*. *r' a a'* ⟹ *r a a'*; ⋀*a a'*. *s' a a'* ⟹ *s a a'* ⟧ ⟹ *order-consistent r' s'*
by(*blast intro*: *order-consistentI order-consistentD*)

lemma *order-consistent-sym*:
order-consistent r s ⟹ *order-consistent s r*
by(*blast intro*: *order-consistentI dest*: *order-consistentD*)

lemma *antisym-order-consistent-self*:
antisymp r ⟹ *order-consistent r r*
by(*rule order-consistentI*)(*erule antisympD*, *simp-all*)

lemma *total-on-refl-on-consistent-into*:
assumes *r*: *total-onP A r refl-onP A r*
and *consist*: *order-consistent r s*
and *x*: *x* ∈ *A* **and** *y*: *y* ∈ *A* **and** *s*: *s x y*
shows *r x y*
proof(*cases x* = *y*)
case *True*
with *r x y* **show** *?thesis* **by**(*blast intro*: *refl-onPD*)
next
case *False*
with *r x y* **have** *r x y* ∨ *r y x* **by**(*blast dest*: *total-onD*)
thus *?thesis*
proof
assume *r y x*
with *s consist* **have** *x* = *y* **by**(*blast dest*: *order-consistentD*)
with *False* **show** *?thesis* **by**(*contradiction*)
qed
qed

lemma *porder-torder-tranclpE* [*consumes 5*, *case-names base step*]:

```
  assumes r: porder-on A r
  and s: torder-on B s
  and consist: order-consistent r s
  and B-subset-A: B ⊆ A
  and trancl: (λa b. r a b ∨ s a b)^++ x y
  obtains r x y
      | u v where r x u s u v r v y
proof(atomize-elim)
  from r have refl-onP A r transp r by(blast elim: porder-onE)+
  from s have refl-onP B s total-onP B s transp s
    by(blast elim: torder-onE porder-onE)+

  from trancl show r x y ∨ (∃ u v. r x u ∧ s u v ∧ r v y)
  proof(induct)
    case (base y)
    thus ?case
    proof
      assume s x y
      with s have x ∈ B y ∈ B
        by(blast elim: torder-onE porder-onE dest: refl-onPD1 refl-onPD2)+
      with B-subset-A have x ∈ A y ∈ A by blast+
      with refl-onPD[OF ‹refl-onP A r›, of x] refl-onPD[OF ‹refl-onP A r›, of y] ‹s x y›
      show ?thesis by(iprover)
    next
      assume r x y
      thus ?thesis ..
    qed
  next
    case (step y z)
    note IH = ‹r x y ∨ (∃ u v. r x u ∧ s u v ∧ r v y)›
    from ‹r y z ∨ s y z› show ?case
    proof
      assume s y z
      with ‹refl-onP B s› have y ∈ B z ∈ B
        by(blast dest: refl-onPD2 refl-onPD1)+
      from IH show ?thesis
      proof
        assume r x y
        moreover from ‹z ∈ B› B-subset-A r have r z z
          by(blast elim: porder-onE dest: refl-onPD)
        ultimately show ?thesis using ‹s y z› by blast
      next
        assume ∃ u v. r x u ∧ s u v ∧ r v y
        then obtain u v where r x u s u v r v y by blast
        from ‹refl-onP B s› ‹s u v› have v ∈ B by(rule refl-onPD2)
        with ‹total-onP B s› ‹refl-onP B s› order-consistent-sym[OF consist]
        have s v y using ‹y ∈ B› ‹r v y›
          by(rule total-on-refl-on-consistent-into)
        with ‹transp s› have s v z using ‹s y z› by(rule transPD)
        with ‹transp s› ‹s u v› have s u z by(rule transPD)
        moreover from ‹z ∈ B› B-subset-A have z ∈ A ..
        with ‹refl-onP A r› have r z z by(rule refl-onPD)
        ultimately show ?thesis using ‹r x u› by blast
      qed
```

```
    next
      assume r y z
      with IH show ?thesis
        by(blast intro: transPD[OF ‹transp r›])
    qed
  qed
qed

lemma torder-on-porder-on-consistent-tranclp-antisym:
  assumes r: porder-on A r
  and s: torder-on B s
  and consist: order-consistent r s
  and B-subset-A: B ⊆ A
  shows antisymp (λx y. r x y ∨ s x y)^++
proof(rule antisymPI)
  fix x y
  let ?rs = λx y. r x y ∨ s x y
  assume ?rs^++ x y ?rs^++ y x
  from r have antisymp r transp r by(blast elim: porder-onE)+
  from s have total-onP B s refl-onP B s transp s antisymp s
    by(blast elim: torder-onE porder-onE)+

  from r s consist B-subset-A ‹?rs^++ x y›
  show x = y
  proof(cases rule: porder-torder-tranclpE)
    case base
    from r s consist B-subset-A ‹?rs^++ y x›
    show ?thesis
    proof(cases rule: porder-torder-tranclpE)
      case base
      with ‹antisymp r› ‹r x y› show ?thesis by(rule antisymPD)
    next
      case (step u v)
      from ‹r v x› ‹r x y› ‹r y u› have r v u by(blast intro: transPD[OF ‹transp r›])
      with consist have v = u using ‹s u v› by(rule order-consistentD)
      with ‹r y u› ‹r v x› have r y x by(blast intro: transPD[OF ‹transp r›])
      with ‹r x y› show ?thesis by(rule antisymPD[OF ‹antisymp r›])
    qed
  next
    case (step u v)
    from r s consist B-subset-A ‹?rs^++ y x›
    show ?thesis
    proof(cases rule: porder-torder-tranclpE)
      case base
      from ‹r v y› ‹r y x› ‹r x u› have r v u by(blast intro: transPD[OF ‹transp r›])
      with order-consistent-sym[OF consist] ‹s u v›
      have u = v by(rule order-consistentD)
      with ‹r v y› ‹r x u› have r x y by(blast intro: transPD[OF ‹transp r›])
      thus ?thesis using ‹r y x› by(rule antisymPD[OF ‹antisymp r›])
    next
      case (step u' v')
    note r-into-s = total-on-refl-on-consistent-into[OF ‹total-onP B s› ‹refl-onP B s› order-consistent-sym[OF
consist]]
      from ‹refl-onP B s› ‹s u v› ‹s u' v'›
```

```
    have u ∈ B v ∈ B u' ∈ B v' ∈ B by(blast dest: refl-onPD1 refl-onPD2)+
    from ‹r v' x› ‹r x u› have r v' u by(rule transPD[OF ‹transp r›])
    with ‹v' ∈ B› ‹u ∈ B› have s v' u by(rule r-into-s)
    also note ‹s u v›
    also (transPD[OF ‹transp s›])
    from ‹r v y› ‹r y u'› have r v u' by(rule transPD[OF ‹transp r›])
    with ‹v ∈ B› ‹u' ∈ B› have s v u' by(rule r-into-s)
    finally (transPD[OF ‹transp s›])
    have v' = u' using ‹s u' v'› by(rule antisymPD[OF ‹antisymp s›])
    moreover with ‹s v u'› ‹s v' u› have s v u by(blast intro: transPD[OF ‹transp s›])
    with ‹s u v› have u = v by(rule antisymPD[OF ‹antisymp s›])
    ultimately have r x y r y x using ‹r x u› ‹r v y› ‹r y u'› ‹r v' x›
      by(blast intro: transPD[OF ‹transp r›])+
    thus ?thesis by(rule antisymPD[OF ‹antisymp r›])
  qed
 qed
qed

lemma porder-on-torder-on-tranclp-porder-onI:
 assumes r: porder-on A r
 and s: torder-on B s
 and consist: order-consistent r s
 and subset: B ⊆ A
 shows porder-on A (λa b. r a b ∨ s a b)^++
proof(rule porder-onI)
 let ?rs = λa b. r a b ∨ s a b
 from r have refl-onP A r by(rule porder-onE)
 moreover from s have refl-onP B s by(blast elim: torder-onE porder-onE)
 ultimately have refl-onP (A ∪ B) ?rs by(rule refl-onP-Un)
 also from subset have A ∪ B = A by blast
 finally show refl-onP A ?rs^++ by(rule refl-onP-tranclp)

 show transp ?rs^++ by(rule transP-tranclp)

 from r s consist subset show antisymp ?rs^++
  by (rule torder-on-porder-on-consistent-tranclp-antisym)
qed

lemma porder-on-sub-torder-on-tranclp-porder-onI:
 assumes r: porder-on A r
 and s: torder-on B s
 and consist: order-consistent r s
 and t: ⋀x y. t x y ⟹ s x y
 and subset: B ⊆ A
 shows porder-on A (λx y. r x y ∨ t x y)^++
proof(rule porder-onI)
 let ?rt = λx y. r x y ∨ t x y
 let ?rs = λx y. r x y ∨ s x y
 from r s consist subset have antisymp ?rs^++
  by(rule torder-on-porder-on-consistent-tranclp-antisym)
 thus antisymp ?rt^++
 proof(rule antisym-subset)
  fix x y
  assume ?rt^++ x y thus ?rs^++ x y
```

```
    by(induct)(blast intro: tranclp.r-into-trancl t tranclp.trancl-into-trancl t)+
  qed

  from s have refl-onP B s by(blast elim: porder-onE torder-onE)
  { fix x y
    assume t x y
    hence s x y by(rule t)
    hence x ∈ B y ∈ B
      by(blast dest: refl-onPD1[OF ‹refl-onP B s›] refl-onPD2[OF ‹refl-onP B s›])+
    with subset have x ∈ A y ∈ A by blast+ }
  note t-reflD = this

  from r have refl-onP A r by(rule porder-onE)
  show refl-onP A ?rt^++
  proof(rule refl-onPI)
    fix a a'
    assume ?rt^++ a a'
    thus a ∈ A ∧ a' ∈ A
      by(induct)(auto dest: refl-onPD1[OF ‹refl-onP A r›] refl-onPD2[OF ‹refl-onP A r›] t-reflD)
  next
    fix a
    assume a ∈ A
    with ‹refl-onP A r› have r a a by(rule refl-onPD)
    thus ?rt^++ a a by(blast intro: tranclp.r-into-trancl)
  qed

  show transp ?rt^++ by(rule transP-tranclp)
qed
```

8.3.4 Order restrictions

```
lemma restrictPI [intro!, simp]:
  ⟦ r a b; a ∈ A; b ∈ A ⟧ ⟹ (r |‘ A) a b
unfolding restrictP-def by simp

lemma restrictPE [elim!]:
  assumes (r |‘ A) a b
  obtains r a b a ∈ A b ∈ A
using assms unfolding restrictP-def by simp

lemma restrictP-empty [simp]: R |‘ {} = (λ- -. False)
by(simp add: restrictP-def[abs-def])

lemma refl-on-restrictPI:
  refl-onP A r ⟹ refl-onP (A ∩ B) (r |‘ B)
by(rule refl-onPI)(blast dest: refl-onPD1 refl-onPD2 refl-onPD)+

lemma refl-on-restrictPI':
  ⟦ refl-onP A r; B = A ∩ C ⟧ ⟹ refl-onP B (r |‘ C)
by(simp add: refl-on-restrictPI)

lemma antisym-restrictPI:
  antisymp r ⟹ antisymp (r |‘ A)
by(rule antisymPI)(blast dest: antisymPD)
```

lemma *trans-restrictPI*:
transp r $\Longrightarrow$ *transp* (*r* |‘ *A*)
by(*rule transPI*)(*blast dest*: *transPD*)

lemma *porder-on-restrictPI*:
porder-on A r $\Longrightarrow$ *porder-on* (*A* ∩ *B*) (*r* |‘ *B*)
by(*blast elim*: *porder-onE intro*: *refl-on-restrictPI antisym-restrictPI trans-restrictPI porder-onI*)

lemma *porder-on-restrictPI'*:
⟦ *porder-on A r*; *B* = *A* ∩ *C* ⟧ $\Longrightarrow$ *porder-on B* (*r* |‘ *C*)
by(*simp add*: *porder-on-restrictPI*)

lemma *total-on-restrictPI*:
total-onP A r $\Longrightarrow$ *total-onP* (*A* ∩ *B*) (*r* |‘ *B*)
by(*blast intro*: *total-onPI dest*: *total-onPD*)

lemma *total-on-restrictPI'*:
⟦ *total-onP A r*; *B* = *A* ∩ *C* ⟧ $\Longrightarrow$ *total-onP B* (*r* |‘ *C*)
by(*simp add*: *total-on-restrictPI*)

lemma *torder-on-restrictPI*:
torder-on A r $\Longrightarrow$ *torder-on* (*A* ∩ *B*) (*r* |‘ *B*)
by(*blast elim*: *torder-onE intro*: *torder-onI porder-on-restrictPI total-on-restrictPI*)

lemma *torder-on-restrictPI'*:
⟦ *torder-on A r*; *B* = *A* ∩ *C* ⟧ $\Longrightarrow$ *torder-on B* (*r* |‘ *C*)
by(*simp add*: *torder-on-restrictPI*)

lemma *restrictP-commute*:
fixes *r* :: *'a* ⇒ *'a* ⇒ *bool*
shows *r* |‘ *A* |‘ *B* = *r* |‘ *B* |‘ *A*
by(*blast intro*!: *ext*)

lemma *restrictP-subsume1*:
fixes *r* :: *'a* ⇒ *'a* ⇒ *bool*
assumes *A* ⊆ *B*
shows *r* |‘ *A* |‘ *B* = *r* |‘ *A*
using *assms* **by**(*blast intro*!: *ext*)

lemma *restrictP-subsume2*:
fixes *r* :: *'a* ⇒ *'a* ⇒ *bool*
assumes *B* ⊆ *A*
shows *r* |‘ *A* |‘ *B* = *r* |‘ *B*
using *assms* **by**(*blast intro*!: *ext*)

lemma *restrictP-idem* [*simp*]:
fixes *r* :: *'a* ⇒ *'a* ⇒ *bool*
shows *r* |‘ *A* |‘ *A* = *r* |‘ *A*
by(*simp add*: *restrictP-subsume1*)

8.3.5 Maximal elements w.r.t. a total order

definition *max-torder* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a* ⇒ *'a* ⇒ *'a*

where *max-torder r a b* = (*if Domainp r a* ∧ *Domainp r b then if r a b then b else a*
else if a = *b then a else SOME a.* ¬ *Domainp r a*)

lemma *refl-on-DomainD*: *refl-on A r* ⟹ *A* = *Domain r*
by(*auto simp add*: *Domain-unfold dest*: *refl-onD refl-onD1*)

lemma *refl-onP-DomainPD*: *refl-onP A r* ⟹ *A* = {*a. Domainp r a*}
by(*drule refl-on-DomainD*) *auto*

lemma *semilattice-max-torder*:
assumes *tot*: *torder-on A r*
shows *semilattice* (*max-torder r*)
proof −
from *tot* **have** *as*: *antisymp r*
and *to*: *total-onP A r*
and *trans*: *transp r*
and *refl*: *refl-onP A r*
by(*auto elim*: *torder-onE porder-onE*)
from *refl* **have** {*a. Domainp r a*} = *A* **by** (*rule refl-onP-DomainPD*[*symmetric*])
from *this* [*symmetric*] **have** *domain*: ⋀*a. Domainp r a* ⟷ *a* ∈ *A* **by** *simp*
show *?thesis*
proof
fix *x y z*
show *max-torder r* (*max-torder r x y*) *z* = *max-torder r x* (*max-torder r y z*)
proof (*cases x* ≠ *y* ∧ *x* ≠ *z* ∧ *y* ≠ *z*)
case *True*
have ∗: ⋀*a b. a* ≠ *b* ⟹ *max-torder r a b* = (*if Domainp r a* ∧ *Domainp r b then*
if r a b then b else a else SOME a. ¬ *Domainp r a*)
by (*auto simp add*: *max-torder-def*)
with *True* **show** *?thesis*
apply (*simp only*: *max-torder-def domain*)
apply (*auto split*!: *if-splits*)
apply (*blast dest*: *total-onPD* [*OF to*] *transPD* [*OF trans*] *antisymPD* [*OF as*] *refl-onPD1* [*OF refl*] *refl-onPD2* [*OF refl*] *someI* [**where** *P*=λ*a. a* ∉ *A*])+
done
next
have *max-torder-idem*: ⋀*a. max-torder r a a* = *a* **by** (*simp add*: *max-torder-def*)
case *False* **then show** *?thesis*
apply (*auto simp add*: *max-torder-idem*)
apply (*auto simp add*: *max-torder-def domain dest*: *someI* [**where** *P*=λ*a. a* ∉ *A*])
done
qed
next
fix *x y*
show *max-torder r x y* = *max-torder r y x*
by (*auto simp add*: *max-torder-def domain dest*: *total-onPD* [*OF to*] *antisymPD* [*OF as*])
next
fix *x*
show *max-torder r x x* = *x*
by (*simp add*: *max-torder-def*)
qed
qed

lemma *max-torder-ge-conv-disj*:

assumes *tot*: *torder-on A r* **and** *x*: $x \in A$ **and** *y*: $y \in A$
shows *r z* (*max-torder r x y*) $\longleftrightarrow$ *r z x* $\vee$ *r z y*
proof −
from *tot* **have** *as*: *antisymp r*
and *to*: *total-onP A r*
and *trans*: *transp r*
and *refl*: *refl-onP A r*
by(*auto elim*: *torder-onE porder-onE*)
from *refl* **have** {*a*. *Domainp r a*} = *A* **by** (*rule refl-onP-DomainPD*[*symmetric*])
from *this* [*symmetric*] **have** *domain*: $\bigwedge$*a*. *Domainp r a* $\longleftrightarrow$ $a \in A$ **by** *simp*
show *?thesis* **using** *x y*
by(*simp add*: *max-torder-def domain*)(*blast dest*: *total-onPD*[*OF to*] *transPD*[*OF trans*])
qed

definition *Max-torder* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a set* ⇒ '*a*
where
Max-torder r = *semilattice-set.F* (*max-torder r*)

context
fixes *A r*
assumes *tot*: *torder-on A r*
begin

lemma *semilattice-set*:
semilattice-set (*max-torder r*)
by (*rule semilattice-set.intro*, *rule semilattice-max-torder*) (*fact tot*)

lemma *domain*:
Domainp r a $\longleftrightarrow$ $a \in A$
proof −
from *tot* **have** {*a*. *Domainp r a*} = *A*
by (*auto elim*: *torder-onE porder-onE dest*: *refl-onP-DomainPD* [*symmetric*])
from *this* [*symmetric*] **show** *?thesis* **by** *simp*
qed

lemma *Max-torder-in-Domain*:
assumes *B*: *finite B B* ≠ {} *B* ⊆ *A*
shows *Max-torder r B* ∈ *A*
proof −
interpret *Max-torder*: *semilattice-set max-torder r*
rewrites
semilattice-set.F (*max-torder r*) = *Max-torder r*
by (*fact semilattice-set Max-torder-def* [*symmetric*])+
show *?thesis* **using** *B*
by (*induct rule*: *finite-ne-induct*) (*simp-all add*: *max-torder-def domain*)
qed

lemma *Max-torder-in-set*:
assumes *B*: *finite B B* ≠ {} *B* ⊆ *A*
shows *Max-torder r B* ∈ *B*
proof −
interpret *Max-torder*: *semilattice-set max-torder r*
rewrites
semilattice-set.F (*max-torder r*) = *Max-torder r*

by (*fact semilattice-set Max-torder-def* [*symmetric*])+
show *?thesis* **using** *B*
by (*induct rule*: *finite-ne-induct*) (*auto simp add*: *max-torder-def domain*)
qed

lemma *Max-torder-above-iff*:
assumes *B*: *finite B B* ≠ {} *B* ⊆ *A*
shows *r x* (*Max-torder r B*) ⟷ (∃ *a*∈*B*. *r x a*)
proof −
interpret *Max-torder*: *semilattice-set max-torder r*
rewrites
semilattice-set.F (*max-torder r*) = *Max-torder r*
by (*fact semilattice-set Max-torder-def* [*symmetric*])+
from *B* **show** *?thesis*
by (*induct rule*: *finite-ne-induct*) (*simp-all add*: *max-torder-ge-conv-disj* [*OF tot*] *Max-torder-in-Domain*)
qed

end

lemma *Max-torder-above*:
assumes *tot*: *torder-on A r*
and *finite B a* ∈ *B B* ⊆ *A*
shows *r a* (*Max-torder r B*)
proof −
from *tot* **have** *refl*: *refl-onP A r* **by**(*auto elim*: *torder-onE porder-onE*)
with ‹*a* ∈ *B*› ‹*B* ⊆ *A*› **have** *r a a* **by**(*blast dest*: *refl-onPD*[*OF refl*])
from ‹*a* ∈ *B*› **have** *B* ≠ {} **by** *auto*
from *Max-torder-above-iff* [*OF tot* ‹*finite B*› *this* ‹*B* ⊆ *A*›, *of a*] ‹*r a a*› ‹*a* ∈ *B*›
show *?thesis* **by** *blast*
qed

lemma *inv-imageP-id* [*simp*]: *inv-imageP R id* = *R*
by(*simp add*: *fun-eq-iff*)

lemma *inv-into-id* [*simp*]: *a* ∈ *A* ⟹ *inv-into A id a* = *a*
by (*metis f-inv-into-f id-apply image-id*)

end

8.4 Axiomatic specification of the JMM

theory *JMM-Spec*
imports
Orders
../Common/Observable-Events
Coinductive.Coinductive-List
begin

8.4.1 Definitions

type-synonym *JMM-action* = *nat*
type-synonym (*'addr*, *'thread-id*) *execution* = (*'thread-id* × (*'addr*, *'thread-id*) *obs-event action*) *llist*

definition *actions* :: (*'addr*, *'thread-id*) *execution* ⇒ *JMM-action set*

where *actions E* = {*n*. *enat n* < *llength E*}

definition *action-tid* :: (*'addr*, *'thread-id*) *execution* ⇒ *JMM-action* ⇒ *'thread-id*
where *action-tid E a* = *fst* (*lnth E a*)

definition *action-obs* :: (*'addr*, *'thread-id*) *execution* ⇒ *JMM-action* ⇒ (*'addr*, *'thread-id*) *obs-event action*
where *action-obs E a* = *snd* (*lnth E a*)

definition *tactions* :: (*'addr*, *'thread-id*) *execution* ⇒ *'thread-id* ⇒ *JMM-action set*
where *tactions E t* = {*a*. *a* ∈ *actions E* ∧ *action-tid E a* = *t*}

inductive *is-new-action* :: (*'addr*, *'thread-id*) *obs-event action* ⇒ *bool*
where
NewHeapElem: *is-new-action* (*NormalAction* (*NewHeapElem a hT*))

inductive *is-write-action* :: (*'addr*, *'thread-id*) *obs-event action* ⇒ *bool*
where
NewHeapElem: *is-write-action* (*NormalAction* (*NewHeapElem ad hT*))
| *WriteMem*: *is-write-action* (*NormalAction* (*WriteMem ad al v*))

Initialisation actions are synchronisation actions iff they initialize volatile fields – cf. JMM mailing list, message no. 62 (5 Nov. 2006). However, intuitively correct programs might not be correctly synchronized:

```
     x = 0
---------------
r1 = x | r2 = x
```

Here, if x is not volatile, the initial write can belong to at most one thread. Hence, it is happens-before to either r1 = x or r2 = x, but not both. In any sequentially consistent execution, both reads must read from the initilisation action x = 0, but it is not happens-before ordered to one of them.

Moreover, if only volatile initialisations synchronize-with all thread-start actions, this breaks the proof of the DRF guarantee since it assumes that the happens-before relation <= *hb* a for an action *a* in a topologically sorted action sequence depends only on the actions before it. Counter example: (y is volatile)

[(t1, start), (t1, init x), (t2, start), (t1, init y), ...

Here, (t1, init x) <= *hb* (t2, start) via: (t1, init x) <= *po* (t1, init y) <= *sw* (t2, start), but in [(t1, start), (t1, init x), (t2, start)], not (t1, init x) <= *hb* (t2, start).

Sevcik speculated that one might add an initialisation thread which performs all initialisation actions. All normal threads' start action would then synchronize on the final action of the initialisation thread. However, this contradicts the memory chain condition in the final field extension to the JMM (threads must read addresses of objects that they have not created themselves before they can access the fields of the object at that address) – not modelled here.

Instead, we leave every initialisation action in the thread it belongs to, but order it explicitly before the thread's start action and add synchronizes-with edges from *all* initialisation actions to *all* thread start actions.

inductive *saction* :: *'m prog* ⇒ (*'addr*, *'thread-id*) *obs-event action* ⇒ *bool*
for *P* :: *'m prog*
where
NewHeapElem: *saction P* (*NormalAction* (*NewHeapElem a hT*))

| *Read*: *is-volatile P al* $\Longrightarrow$ *saction P* (*NormalAction* (*ReadMem a al v*))
| *Write*: *is-volatile P al* $\Longrightarrow$ *saction P* (*NormalAction* (*WriteMem a al v*))
| *ThreadStart*: *saction P* (*NormalAction* (*ThreadStart t*))
| *ThreadJoin*: *saction P* (*NormalAction* (*ThreadJoin t*))
| *SyncLock*: *saction P* (*NormalAction* (*SyncLock a*))
| *SyncUnlock*: *saction P* (*NormalAction* (*SyncUnlock a*))
| *ObsInterrupt*: *saction P* (*NormalAction* (*ObsInterrupt t*))
| *ObsInterrupted*: *saction P* (*NormalAction* (*ObsInterrupted t*))
| *InitialThreadAction*: *saction P InitialThreadAction*
| *ThreadFinishAction*: *saction P ThreadFinishAction*

definition *sactions* :: *'m prog* $\Rightarrow$ (*'addr*, *'thread-id*) *execution* $\Rightarrow$ *JMM-action set*
where *sactions P E* = {*a*. *a* $\in$ *actions E* $\wedge$ *saction P* (*action-obs E a*)}

inductive-set *write-actions* :: (*'addr*, *'thread-id*) *execution* $\Rightarrow$ *JMM-action set*
for *E* :: (*'addr*, *'thread-id*) *execution*
where
write-actionsI: ⟦ *a* $\in$ *actions E*; *is-write-action* (*action-obs E a*) ⟧ $\Longrightarrow$ *a* $\in$ *write-actions E*

NewObj and *NewArr* actions only initialize those fields and array cells that are in fact in the object or array. Hence, they are not a write for - reads from addresses for which no object/array is created during the whole execution - reads from fields/cells that are not part of the object/array at the specified address.

primrec *addr-locs* :: *'m prog* $\Rightarrow$ *htype* $\Rightarrow$ *addr-loc set*
where
addr-locs P (*Class-type C*) = {*CField D F*|*D F*. $\exists$*fm T*. *P* $\vdash$ *C has F*:*T* (*fm*) *in D*}
| *addr-locs P* (*Array-type T n*) = ({*ACell n'*|*n'*. *n'* < *n*} $\cup$ {*CField Object F*|*F*. $\exists$*fm T*. *P* $\vdash$ *Object has F*:*T* (*fm*) *in Object*})

action-loc-aux would naturally be an inductive set, but inductive_set does not allow to pattern match on parameters. Hence, specify it using function and derive the setup manually.

fun *action-loc-aux* :: *'m prog* $\Rightarrow$ (*'addr*, *'thread-id*) *obs-event action* $\Rightarrow$ (*'addr* $\times$ *addr-loc*) *set*
where
action-loc-aux P (*NormalAction* (*NewHeapElem ad* (*Class-type C*))) =
{(*ad*, *CField D F*)|*D F T fm*. *P* $\vdash$ *C has F*:*T* (*fm*) *in D*}
| *action-loc-aux P* (*NormalAction* (*NewHeapElem ad* (*Array-type T n'*))) =
{(*ad*, *ACell n*)|*n*. *n* < *n'*} $\cup$ {(*ad*, *CField D F*)|*D F T fm*. *P* $\vdash$ *Object has F*:*T* (*fm*) *in D*}
| *action-loc-aux P* (*NormalAction* (*WriteMem ad al v*)) = {(*ad*, *al*)}
| *action-loc-aux P* (*NormalAction* (*ReadMem ad al v*)) = {(*ad*, *al*)}
| *action-loc-aux* - - = {}

lemma *action-loc-aux-intros* [*intro?*]:
P $\vdash$ *class-type-of hT has F*:*T* (*fm*) *in D* $\Longrightarrow$ (*ad*, *CField D F*) $\in$ *action-loc-aux P* (*NormalAction* (*NewHeapElem ad hT*))
n < *n'* $\Longrightarrow$ (*ad*, *ACell n*) $\in$ *action-loc-aux P* (*NormalAction* (*NewHeapElem ad* (*Array-type T n'*)))
(*ad*, *al*) $\in$ *action-loc-aux P* (*NormalAction* (*WriteMem ad al v*))
(*ad*, *al*) $\in$ *action-loc-aux P* (*NormalAction* (*ReadMem ad al v*))
by(*cases hT*) *auto*

lemma *action-loc-aux-cases* [*elim?*, *cases set*: *action-loc-aux*]:
assumes *adal* $\in$ *action-loc-aux P obs*
obtains (*NewHeapElem*) *hT F T fm D ad* **where** *obs* = *NormalAction* (*NewHeapElem ad hT*) *adal* = (*ad*, *CField D F*) *P* $\vdash$ *class-type-of hT has F*:*T* (*fm*) *in D*

| (*NewArr*) *n n' ad T* **where** *obs* = *NormalAction* (*NewHeapElem ad* (*Array-type T n'*)) *adal* = (*ad*, *ACell n*) *n* < *n'*
| (*WriteMem*) *ad al v* **where** *obs* = *NormalAction* (*WriteMem ad al v*) *adal* = (*ad*, *al*)
| (*ReadMem*) *ad al v* **where** *obs* = *NormalAction* (*ReadMem ad al v*) *adal* = (*ad*, *al*)
using *assms* **by**(*cases* (*P*, *obs*) *rule*: *action-loc-aux.cases*) *fastforce+*

lemma *action-loc-aux-simps* [*simp*]:
(*ad'*, *al'*) ∈ *action-loc-aux P* (*NormalAction* (*NewHeapElem ad hT*)) ⟷
(∃ *D F T fm*. *ad* = *ad'* ∧ *al'* = *CField D F* ∧ *P* ⊢ *class-type-of hT has F*:*T* (*fm*) *in D*) ∨
(∃ *n T n'*. *ad* = *ad'* ∧ *al'* = *ACell n* ∧ *hT* = *Array-type T n'* ∧ *n* < *n'*)
(*ad'*, *al'*) ∈ *action-loc-aux P* (*NormalAction* (*WriteMem ad al v*)) ⟷ *ad* = *ad'* ∧ *al* = *al'*
(*ad'*, *al'*) ∈ *action-loc-aux P* (*NormalAction* (*ReadMem ad al v*)) ⟷ *ad* = *ad'* ∧ *al* = *al'*
(*ad'*, *al'*) ∉ *action-loc-aux P InitialThreadAction*
(*ad'*, *al'*) ∉ *action-loc-aux P ThreadFinishAction*
(*ad'*, *al'*) ∉ *action-loc-aux P* (*NormalAction* (*ExternalCall a m vs v*))
(*ad'*, *al'*) ∉ *action-loc-aux P* (*NormalAction* (*ThreadStart t*))
(*ad'*, *al'*) ∉ *action-loc-aux P* (*NormalAction* (*ThreadJoin t*))
(*ad'*, *al'*) ∉ *action-loc-aux P* (*NormalAction* (*SyncLock a*))
(*ad'*, *al'*) ∉ *action-loc-aux P* (*NormalAction* (*SyncUnlock a*))
(*ad'*, *al'*) ∉ *action-loc-aux P* (*NormalAction* (*ObsInterrupt t*))
(*ad'*, *al'*) ∉ *action-loc-aux P* (*NormalAction* (*ObsInterrupted t*))
by(*cases hT*) *auto*

declare *action-loc-aux.simps* [*simp del*]

abbreviation *action-loc* :: *'m prog* ⇒ (*'addr*, *'thread-id*) *execution* ⇒ *JMM-action* ⇒ (*'addr* × *addr-loc*) *set*
where *action-loc P E a* ≡ *action-loc-aux P* (*action-obs E a*)

inductive-set *read-actions* :: (*'addr*, *'thread-id*) *execution* ⇒ *JMM-action set*
for *E* :: (*'addr*, *'thread-id*) *execution*
where
ReadMem: ⟦ *a* ∈ *actions E*; *action-obs E a* = *NormalAction* (*ReadMem ad al v*) ⟧ ⟹ *a* ∈ *read-actions E*

fun *addr-loc-default* :: *'m prog* ⇒ *htype* ⇒ *addr-loc* ⇒ *'addr val*
where
addr-loc-default P (*Class-type C*) (*CField D F*) = *default-val* (*fst* (*the* (*map-of* (*fields P C*) (*F*, *D*))))
| *addr-loc-default P* (*Array-type T n*) (*ACell n'*) = *default-val T*
| *addr-loc-default-Array-CField*:
addr-loc-default P (*Array-type T n*) (*CField D F*) = *default-val* (*fst* (*the* (*map-of* (*fields P Object*) (*F*, *Object*))))
| *addr-loc-default P* - - = *undefined*

definition *new-actions-for* :: *'m prog* ⇒ (*'addr*, *'thread-id*) *execution* ⇒ (*'addr* × *addr-loc*) ⇒ *JMM-action set*
where
new-actions-for P E adal =
{*a*. *a* ∈ *actions E* ∧ *adal* ∈ *action-loc P E a* ∧ *is-new-action* (*action-obs E a*)}

inductive-set *external-actions* :: (*'addr*, *'thread-id*) *execution* ⇒ *JMM-action set*
for *E* :: (*'addr*, *'thread-id*) *execution*
where
⟦ *a* ∈ *actions E*; *action-obs E a* = *NormalAction* (*ExternalCall ad M vs v*) ⟧

$\Longrightarrow a \in \mathit{external\text{-}actions}\ E$

fun *value-written-aux* :: $'m\ \mathit{prog} \Rightarrow ('\mathit{addr}, '\mathit{thread\text{-}id})\ \mathit{obs\text{-}event\ action} \Rightarrow \mathit{addr\text{-}loc} \Rightarrow '\mathit{addr}\ \mathit{val}$
where
value-written-aux P (*NormalAction* (*NewHeapElem* $\mathit{ad}'\ hT$)) $\mathit{al} = \mathit{addr\text{-}loc\text{-}default}\ P\ hT\ \mathit{al}$
| *value-written-aux-WriteMem'*:
value-written-aux P (*NormalAction* (*WriteMem* $\mathit{ad}\ \mathit{al}'\ v$)) $\mathit{al} = (\mathit{if}\ \mathit{al} = \mathit{al}'\ \mathit{then}\ v\ \mathit{else}\ \mathit{undefined})$
| *value-written-aux-undefined*:
value-written-aux P - $\mathit{al} = \mathit{undefined}$

primrec *value-written* :: $'m\ \mathit{prog} \Rightarrow ('\mathit{addr}, '\mathit{thread\text{-}id})\ \mathit{execution} \Rightarrow \mathit{JMM\text{-}action} \Rightarrow ('\mathit{addr} \times \mathit{addr\text{-}loc}) \Rightarrow '\mathit{addr}\ \mathit{val}$
where *value-written* $P\ E\ a\ (\mathit{ad}, \mathit{al}) = \mathit{value\text{-}written\text{-}aux}\ P\ (\mathit{action\text{-}obs}\ E\ a)\ \mathit{al}$

definition *value-read* :: $('\mathit{addr}, '\mathit{thread\text{-}id})\ \mathit{execution} \Rightarrow \mathit{JMM\text{-}action} \Rightarrow '\mathit{addr}\ \mathit{val}$
where
value-read $E\ a =$
(*case action-obs* $E\ a$ *of*
NormalAction obs $\Rightarrow$
(*case obs of*
ReadMem ad al $v \Rightarrow v$
| - $\Rightarrow$ *undefined*)
| - $\Rightarrow$ *undefined*)

definition *action-order* :: $('\mathit{addr}, '\mathit{thread\text{-}id})\ \mathit{execution} \Rightarrow \mathit{JMM\text{-}action} \Rightarrow \mathit{JMM\text{-}action} \Rightarrow \mathit{bool}$ ($\langle$- $\vdash$ - $\leq a$ -$\rangle$ [51,0,50] 50)
where
$E \vdash a \leq a\ a' \longleftrightarrow$
$a \in \mathit{actions}\ E \wedge a' \in \mathit{actions}\ E \wedge$
(*if is-new-action* (*action-obs* $E\ a$)
then is-new-action (*action-obs* $E\ a'$) $\longrightarrow a \leq a'$
else $\neg$ *is-new-action* (*action-obs* $E\ a'$) $\wedge\ a \leq a'$)

definition *program-order* :: $('\mathit{addr}, '\mathit{thread\text{-}id})\ \mathit{execution} \Rightarrow \mathit{JMM\text{-}action} \Rightarrow \mathit{JMM\text{-}action} \Rightarrow \mathit{bool}$ ($\langle$- $\vdash$ - $\leq po$ -$\rangle$ [51,0,50] 50)
where
$E \vdash a \leq po\ a' \longleftrightarrow E \vdash a \leq a\ a' \wedge \mathit{action\text{-}tid}\ E\ a = \mathit{action\text{-}tid}\ E\ a'$

inductive *synchronizes-with* ::
$'m\ \mathit{prog}$
$\Rightarrow ('\mathit{thread\text{-}id} \times ('\mathit{addr}, '\mathit{thread\text{-}id})\ \mathit{obs\text{-}event\ action}) \Rightarrow ('\mathit{thread\text{-}id} \times ('\mathit{addr}, '\mathit{thread\text{-}id})\ \mathit{obs\text{-}event\ action}) \Rightarrow \mathit{bool}$
($\langle$- $\vdash$ - $\leadsto sw$ -$\rangle$ [51, 51, 51] 50)
for P :: $'m\ \mathit{prog}$
where
ThreadStart: $P \vdash (t,$ *NormalAction* (*ThreadStart* $t')) \leadsto sw\ (t',$ *InitialThreadAction*)
| *ThreadFinish*: $P \vdash (t,$ *ThreadFinishAction*$) \leadsto sw\ (t',$ *NormalAction* (*ThreadJoin* $t'))$
| *UnlockLock*: $P \vdash (t,$ *NormalAction* (*SyncUnlock* $a')) \leadsto sw\ (t',$ *NormalAction* (*SyncLock* $a'))$
| — Only volatile writes synchronize with volatile reads. We could check volatility of *al* here, but this is checked by *sactions* in *sync-with* anyway.
Volatile: $P \vdash (t,$ *NormalAction* (*WriteMem* $a\ \mathit{al}\ v')) \leadsto sw\ (t',$ *NormalAction* (*ReadMem* $a\ \mathit{al}\ v'))$
| *VolatileNew*:
$\mathit{al} \in \mathit{addr\text{-}locs}\ P\ hT$
$\Longrightarrow P \vdash (t,$ *NormalAction* (*NewHeapElem* $a\ hT')) \leadsto sw\ (t',$ *NormalAction* (*ReadMem* $a\ \mathit{al}\ v'))$

| *NewHeapElem*: $P \vdash (t, \mathit{NormalAction}\ (\mathit{NewHeapElem}\ a\ hT)) \rightsquigarrow sw\ (t', \mathit{InitialThreadAction})$
| *Interrupt*: $P \vdash (t, \mathit{NormalAction}\ (\mathit{ObsInterrupt}\ t')) \rightsquigarrow sw\ (t'', \mathit{NormalAction}\ (\mathit{ObsInterrupted}\ t'))$

definition *sync-order* ::
$'m\ prog \Rightarrow ('addr, 'thread\text{-}id)\ execution \Rightarrow JMM\text{-}action \Rightarrow JMM\text{-}action \Rightarrow bool$
$(\langle\text{-},\text{-} \vdash \text{-} \leq so \text{-}\rangle\ [51,0,0,50]\ 50)$
where
$P,E \vdash a \leq so\ a' \longleftrightarrow a \in sactions\ P\ E \wedge a' \in sactions\ P\ E \wedge E \vdash a \leq a\ a'$

definition *sync-with* ::
$'m\ prog \Rightarrow ('addr, 'thread\text{-}id)\ execution \Rightarrow JMM\text{-}action \Rightarrow JMM\text{-}action \Rightarrow bool$
$(\langle\text{-},\text{-} \vdash \text{-} \leq sw \text{-}\rangle\ [51,\ 0,\ 0,\ 50]\ 50)$
where
$P,E \vdash a \leq sw\ a' \longleftrightarrow$
$P,E \vdash a \leq so\ a' \wedge P \vdash (action\text{-}tid\ E\ a,\ action\text{-}obs\ E\ a) \rightsquigarrow sw\ (action\text{-}tid\ E\ a',\ action\text{-}obs\ E\ a')$

definition *po-sw* :: $'m\ prog \Rightarrow ('addr, 'thread\text{-}id)\ execution \Rightarrow JMM\text{-}action \Rightarrow JMM\text{-}action \Rightarrow bool$
where $po\text{-}sw\ P\ E\ a\ a' \longleftrightarrow E \vdash a \leq po\ a' \vee P,E \vdash a \leq sw\ a'$

abbreviation *happens-before* ::
$'m\ prog \Rightarrow ('addr, 'thread\text{-}id)\ execution \Rightarrow JMM\text{-}action \Rightarrow JMM\text{-}action \Rightarrow bool$
$(\langle\text{-},\text{-} \vdash \text{-} \leq hb \text{-}\rangle\ [51,\ 0,\ 0,\ 50]\ 50)$
where $happens\text{-}before\ P\ E \equiv (po\text{-}sw\ P\ E)^{\frown}{+}{+}$

type-synonym *write-seen* = $JMM\text{-}action \Rightarrow JMM\text{-}action$

definition *is-write-seen* :: $'m\ prog \Rightarrow ('addr, 'thread\text{-}id)\ execution \Rightarrow write\text{-}seen \Rightarrow bool$ **where**
$is\text{-}write\text{-}seen\ P\ E\ ws \longleftrightarrow$
$(\forall a \in read\text{-}actions\ E.\ \forall ad\ al\ v.\ action\text{-}obs\ E\ a = \mathit{NormalAction}\ (\mathit{ReadMem}\ ad\ al\ v) \longrightarrow$
$ws\ a \in write\text{-}actions\ E \wedge (ad, al) \in action\text{-}loc\ P\ E\ (ws\ a) \wedge$
$value\text{-}written\ P\ E\ (ws\ a)\ (ad, al) = v \wedge \neg\ P,E \vdash a \leq hb\ ws\ a \wedge$
$(is\text{-}volatile\ P\ al \longrightarrow \neg\ P,E \vdash a \leq so\ ws\ a) \wedge$
$(\forall w' \in write\text{-}actions\ E.\ (ad, al) \in action\text{-}loc\ P\ E\ w' \longrightarrow$
$(P,E \vdash ws\ a \leq hb\ w' \wedge P,E \vdash w' \leq hb\ a \vee is\text{-}volatile\ P\ al \wedge P,E \vdash ws\ a \leq so\ w' \wedge P,E \vdash w' \leq so\ a) \longrightarrow$
$w' = ws\ a))$

definition *thread-start-actions-ok* :: $('addr, 'thread\text{-}id)\ execution \Rightarrow bool$
where
$thread\text{-}start\text{-}actions\text{-}ok\ E \longleftrightarrow$
$(\forall a \in actions\ E.\ \neg\ is\text{-}new\text{-}action\ (action\text{-}obs\ E\ a) \longrightarrow$
$(\exists i.\ i \leq a \wedge action\text{-}obs\ E\ i = \mathit{InitialThreadAction} \wedge action\text{-}tid\ E\ i = action\text{-}tid\ E\ a))$

primrec *wf-exec* :: $'m\ prog \Rightarrow ('addr, 'thread\text{-}id)\ execution \times write\text{-}seen \Rightarrow bool$ $(\langle\text{-} \vdash \text{-} \surd\rangle\ [51,\ 50]\ 51)$
where $P \vdash (E, ws) \surd \longleftrightarrow is\text{-}write\text{-}seen\ P\ E\ ws \wedge thread\text{-}start\text{-}actions\text{-}ok\ E$

inductive *most-recent-write-for* :: $'m\ prog \Rightarrow ('addr, 'thread\text{-}id)\ execution \Rightarrow JMM\text{-}action \Rightarrow JMM\text{-}action \Rightarrow bool$
$(\langle\text{-},\text{-} \vdash \text{-} \rightsquigarrow mrw \text{-}\rangle\ [50,\ 0,\ 51]\ 51)$
for P :: $'m\ prog$ **and** E :: $('addr, 'thread\text{-}id)\ execution$ **and** ra :: $JMM\text{-}action$ **and** wa :: $JMM\text{-}action$
where
$[\![\ ra \in read\text{-}actions\ E;\ adal \in action\text{-}loc\ P\ E\ ra;\ E \vdash wa \leq a\ ra;$
$wa \in write\text{-}actions\ E;\ adal \in action\text{-}loc\ P\ E\ wa;$

$\bigwedge wa'.\ [\![\ wa' \in \textit{write-actions}\ E;\ \textit{adal} \in \textit{action-loc}\ P\ E\ wa'\ ]\!]$
$\Longrightarrow E \vdash wa' \leq_a wa \vee E \vdash ra \leq_a wa'\ ]\!]$
$\Longrightarrow P,E \vdash ra \rightsquigarrow_{mrw} wa$

primrec *sequentially-consistent* :: *'m prog* ⇒ ((*'addr*, *'thread-id*) *execution* × *write-seen*) ⇒ *bool*
where
sequentially-consistent P (*E*, *ws*) ⟷ (∀ *r* ∈ *read-actions E*. *P*,*E* ⊢ *r* ⇝*mrw ws r*)

8.4.2 Actions

inductive-cases *is-new-action-cases* [*elim*!]:
is-new-action (*NormalAction* (*ExternalCall a M vs v*))
is-new-action (*NormalAction* (*ReadMem a al v*))
is-new-action (*NormalAction* (*WriteMem a al v*))
is-new-action (*NormalAction* (*NewHeapElem a hT*))
is-new-action (*NormalAction* (*ThreadStart t*))
is-new-action (*NormalAction* (*ThreadJoin t*))
is-new-action (*NormalAction* (*SyncLock a*))
is-new-action (*NormalAction* (*SyncUnlock a*))
is-new-action (*NormalAction* (*ObsInterrupt t*))
is-new-action (*NormalAction* (*ObsInterrupted t*))
is-new-action InitialThreadAction
is-new-action ThreadFinishAction

inductive-simps *is-new-action-simps* [*simp*]:
is-new-action (*NormalAction* (*NewHeapElem a hT*))
is-new-action (*NormalAction* (*ExternalCall a M vs v*))
is-new-action (*NormalAction* (*ReadMem a al v*))
is-new-action (*NormalAction* (*WriteMem a al v*))
is-new-action (*NormalAction* (*ThreadStart t*))
is-new-action (*NormalAction* (*ThreadJoin t*))
is-new-action (*NormalAction* (*SyncLock a*))
is-new-action (*NormalAction* (*SyncUnlock a*))
is-new-action (*NormalAction* (*ObsInterrupt t*))
is-new-action (*NormalAction* (*ObsInterrupted t*))
is-new-action InitialThreadAction
is-new-action ThreadFinishAction

lemmas *is-new-action-iff* = *is-new-action.simps*

inductive-simps *is-write-action-simps* [*simp*]:
is-write-action InitialThreadAction
is-write-action ThreadFinishAction
is-write-action (*NormalAction* (*ExternalCall a m vs v*))
is-write-action (*NormalAction* (*ReadMem a al v*))
is-write-action (*NormalAction* (*WriteMem a al v*))
is-write-action (*NormalAction* (*NewHeapElem a hT*))
is-write-action (*NormalAction* (*ThreadStart t*))
is-write-action (*NormalAction* (*ThreadJoin t*))
is-write-action (*NormalAction* (*SyncLock a*))
is-write-action (*NormalAction* (*SyncUnlock a*))
is-write-action (*NormalAction* (*ObsInterrupt t*))
is-write-action (*NormalAction* (*ObsInterrupted t*))

declare *saction.intros* [*intro*!]

inductive-cases *saction-cases* [*elim*!]:
saction P (*NormalAction* (*ExternalCall a M vs v*))
saction P (*NormalAction* (*ReadMem a al v*))
saction P (*NormalAction* (*WriteMem a al v*))
saction P (*NormalAction* (*NewHeapElem a hT*))
saction P (*NormalAction* (*ThreadStart t*))
saction P (*NormalAction* (*ThreadJoin t*))
saction P (*NormalAction* (*SyncLock a*))
saction P (*NormalAction* (*SyncUnlock a*))
saction P (*NormalAction* (*ObsInterrupt t*))
saction P (*NormalAction* (*ObsInterrupted t*))
saction P InitialThreadAction
saction P ThreadFinishAction

inductive-simps *saction-simps* [*simp*]:
saction P (*NormalAction* (*ExternalCall a M vs v*))
saction P (*NormalAction* (*ReadMem a al v*))
saction P (*NormalAction* (*WriteMem a al v*))
saction P (*NormalAction* (*NewHeapElem a hT*))
saction P (*NormalAction* (*ThreadStart t*))
saction P (*NormalAction* (*ThreadJoin t*))
saction P (*NormalAction* (*SyncLock a*))
saction P (*NormalAction* (*SyncUnlock a*))
saction P (*NormalAction* (*ObsInterrupt t*))
saction P (*NormalAction* (*ObsInterrupted t*))
saction P InitialThreadAction
saction P ThreadFinishAction

lemma *new-action-saction* [*simp*, *intro*]: *is-new-action a* $\Longrightarrow$ *saction P a*
by(*blast elim*: *is-new-action.cases*)

lemmas *saction-iff* = *saction.simps*

lemma *actionsD*: *a* $\in$ *actions E* $\Longrightarrow$ *enat a* $<$ *llength E*
unfolding *actions-def* **by** *blast*

lemma *actionsE*:
assumes *a* $\in$ *actions E*
obtains *enat a* $<$ *llength E*
using *assms* **unfolding** *actions-def* **by** *blast*

lemma *actions-lappend*:
llength xs = *enat n* $\Longrightarrow$ *actions* (*lappend xs ys*) = *actions xs* $\cup$ ((+) *n*) ‘ *actions ys*
unfolding *actions-def*
apply *safe*
apply(*erule contrapos-np*)
apply(*rule-tac x*=*x* $-$ *n* **in** *image-eqI*)
apply *simp-all*
apply(*case-tac* [!] *llength ys*)
apply *simp-all*
done

lemma *tactionsE*:
assumes $a \in$ *tactions E t*
obtains *obs* **where** $a \in$ *actions E action-tid E a* $=$ *t action-obs E a* $=$ *obs*
using *assms*
by(*cases lnth E a*)(*auto simp add*: *tactions-def action-tid-def action-obs-def*)

lemma *sactionsI*:
$[\![a \in$ *actions E*; *saction P* (*action-obs E a*) $]\!] \Longrightarrow a \in$ *sactions P E*
unfolding *sactions-def* **by** *blast*

lemma *sactionsE*:
assumes $a \in$ *sactions P E*
obtains $a \in$ *actions E saction P* (*action-obs E a*)
using *assms* **unfolding** *sactions-def* **by** *blast*

lemma *sactions-actions* [*simp*]:
$a \in$ *sactions P E* $\Longrightarrow a \in$ *actions E*
by(*rule sactionsE*)

lemma *value-written-aux-WriteMem* [*simp*]:
value-written-aux P (*NormalAction* (*WriteMem ad al v*)) *al* $=$ *v*
by *simp*

declare *value-written-aux-undefined* [*simp del*]
declare *value-written-aux-WriteMem'* [*simp del*]

inductive-simps *is-write-action-iff*:
is-write-action a

inductive-simps *write-actions-iff*:
$a \in$ *write-actions E*

lemma *write-actions-actions* [*simp*]:
$a \in$ *write-actions E* $\Longrightarrow a \in$ *actions E*
by(*rule write-actions.induct*)

inductive-simps *read-actions-iff*:
$a \in$ *read-actions E*

lemma *read-actions-actions* [*simp*]:
$a \in$ *read-actions E* $\Longrightarrow a \in$ *actions E*
by(*rule read-actions.induct*)

lemma *read-action-action-locE*:
assumes $r \in$ *read-actions E*
obtains *ad al* **where** (*ad*, *al*) $\in$ *action-loc P E r*
using *assms* **by** *cases auto*

lemma *read-actions-not-write-actions*:
$[\![a \in$ *read-actions E*; $a \in$ *write-actions E* $]\!] \Longrightarrow$ *False*
by(*auto elim*!: *read-actions.cases write-actions.cases*)

lemma *read-actions-Int-write-actions* [*simp*]:
read-actions E $\cap$ *write-actions E* $= \{\}$

write-actions E ∩ read-actions E = {}
by(*blast dest*: *read-actions-not-write-actions*)+

lemma *action-loc-addr-fun*:
⟦ (*ad*, *al*) ∈ *action-loc P E a*; (*ad′*, *al′*) ∈ *action-loc P E a* ⟧ ⟹ *ad* = *ad′*
by(*auto elim*!: *action-loc-aux-cases*)

lemma *value-written-cong* [*cong*]:
⟦ *P* = *P′*; *a* = *a′*; *action-obs E a′* = *action-obs E′ a′* ⟧
⟹ *value-written P E a* = *value-written P′ E′ a′*
by(*rule ext*)(*auto split*: *action.splits*)

declare *value-written.simps* [*simp del*]

lemma *new-actionsI*:
⟦ *a* ∈ *actions E*; *adal* ∈ *action-loc P E a*; *is-new-action* (*action-obs E a*) ⟧
⟹ *a* ∈ *new-actions-for P E adal*
unfolding *new-actions-for-def* **by** *blast*

lemma *new-actionsE*:
assumes *a* ∈ *new-actions-for P E adal*
obtains *a* ∈ *actions E adal* ∈ *action-loc P E a is-new-action* (*action-obs E a*)
using *assms* **unfolding** *new-actions-for-def* **by** *blast*

lemma *action-loc-read-action-singleton*:
⟦ *r* ∈ *read-actions E*; *adal* ∈ *action-loc P E r*; *adal′* ∈ *action-loc P E r* ⟧ ⟹ *adal* = *adal′*
by(*cases adal*, *cases adal′*)(*fastforce elim*: *read-actions.cases action-loc-aux-cases*)

lemma *addr-locsI*:
P ⊢ *class-type-of hT has F*:*T* (*fm*) *in D* ⟹ *CField D F* ∈ *addr-locs P hT*
⟦ *hT* = *Array-type T n*; *n′* < *n* ⟧ ⟹ *ACell n′* ∈ *addr-locs P hT*
by(*cases hT*)(*auto dest*: *has-field-decl-above*)

8.4.3 Orders

8.4.4 Action order

lemma *action-orderI*:
assumes *a* ∈ *actions E a′* ∈ *actions E*
and ⟦ *is-new-action* (*action-obs E a*); *is-new-action* (*action-obs E a′*) ⟧ ⟹ *a* ≤ *a′*
and ¬ *is-new-action* (*action-obs E a*) ⟹ ¬ *is-new-action* (*action-obs E a′*) ∧ *a* ≤ *a′*
shows *E* ⊢ *a* ≤*a a′*
using *assms* **unfolding** *action-order-def* **by** *simp*

lemma *action-orderE*:
assumes *E* ⊢ *a* ≤*a a′*
obtains *a* ∈ *actions E a′* ∈ *actions E*
is-new-action (*action-obs E a*) *is-new-action* (*action-obs E a′*) ⟶ *a* ≤ *a′*
| *a* ∈ *actions E a′* ∈ *actions E*
¬ *is-new-action* (*action-obs E a*) ¬ *is-new-action* (*action-obs E a′*) *a* ≤ *a′*
using *assms* **unfolding** *action-order-def* **by**(*simp split*: *if-split-asm*)

lemma *refl-action-order*:
refl-onP (*actions E*) (*action-order E*)

by(*rule refl-onPI*)(*auto elim*: *action-orderE intro*: *action-orderI*)

lemma *antisym-action-order*:
antisymp (*action-order E*)
by(*rule antisympI*)(*auto elim*!: *action-orderE*)

lemma *trans-action-order*:
transp (*action-order E*)
by(*rule transpI*)(*auto elim*!: *action-orderE intro*: *action-orderI*)

lemma *porder-action-order*:
porder-on (*actions E*) (*action-order E*)
by(*blast intro*: *porder-onI refl-action-order antisym-action-order trans-action-order*)

lemma *total-action-order*:
total-onP (*actions E*) (*action-order E*)
by(*rule total-onPI*)(*auto simp add*: *action-order-def*)

lemma *torder-action-order*:
torder-on (*actions E*) (*action-order E*)
by(*blast intro*: *torder-onI total-action-order porder-action-order*)

lemma *wf-action-order*: *wfP* (*action-order E*)$^{\neq\neq}$
unfolding *wfp-eq-minimal*
proof(*intro strip*)
fix Q **and** x :: *JMM-action*
assume $x \in Q$
show $\exists\, z \in Q.\ \forall\, y.$ (*action-order E*)$^{\neq\neq}$ $y\ z \longrightarrow y \notin Q$
proof(*cases* $\exists\, a \in Q.\ a \in$ *actions E* $\wedge$ *is-new-action* (*action-obs E a*))
case *True*
then obtain a **where** a: $a \in$ *actions E* $\wedge$ *is-new-action* (*action-obs E a*) $\wedge\ a \in Q$ **by** *blast*
define a' **where** $a' = (LEAST\ a'.\ a' \in$ *actions E* $\wedge$ *is-new-action* (*action-obs E a'*) $\wedge\ a' \in Q)$
from a **have** a': $a' \in$ *actions E* $\wedge$ *is-new-action* (*action-obs E a'*) $\wedge\ a' \in Q$
unfolding *a'-def* **by**(*rule LeastI*)
{ fix y
assume *y-le-a'*: (*action-order E*)$^{\neq\neq}$ $y\ a'$
have $y \notin Q$
proof
assume $y \in Q$
with *y-le-a' a'* **have** y: $y \in$ *actions E* $\wedge$ *is-new-action* (*action-obs E y*) $\wedge\ y \in Q$
by(*auto elim*: *action-orderE*)
hence $a' \leq y$ **unfolding** *a'-def* **by**(*rule Least-le*)
with *y-le-a' a'* **show** *False* **by**(*auto elim*: *action-orderE*)
qed }
with a' **show** *?thesis* **by** *blast*
next
case *False*
hence *not-new*: $\bigwedge a.\ [\![\ a \in Q;\ a \in$ *actions E* $]\!] \Longrightarrow \neg$ *is-new-action* (*action-obs E a*) **by** *blast*
show *?thesis*
proof(*cases* $Q \cap$ *actions E* $= \{\}$)
case *True*
with ‹$x \in Q$› **show** *?thesis* **by**(*auto elim*: *action-orderE*)
next
case *False*

```
    define a' where a' = (LEAST a'. a' ∈ Q ∧ a' ∈ actions E ∧ ¬ is-new-action (action-obs E a'))
    from False obtain a where a ∈ Q a ∈ actions E by blast
    with not-new[OF this] have a ∈ Q ∧ a ∈ actions E ∧ ¬ is-new-action (action-obs E a) by blast
    hence a': a' ∈ Q ∧ a' ∈ actions E ∧ ¬ is-new-action (action-obs E a')
      unfolding a'-def by(rule LeastI)
    { fix y
      assume y-le-a': (action-order E)^≠≠ y a'
      hence y ∈ actions E by(auto elim: action-orderE)
      have y ∉ Q
      proof
        assume y ∈ Q
        hence y-not-new: ¬ is-new-action (action-obs E y)
          using ‹y ∈ actions E› by(rule not-new)
        with ‹y ∈ Q› ‹y ∈ actions E› have a' ≤ y
          unfolding a'-def by −(rule Least-le, blast)
        with y-le-a' y-not-new show False by(auto elim: action-orderE)
      qed }
    with a' show ?thesis by blast
  qed
 qed
qed

lemma action-order-is-new-actionD:
  ⟦ E ⊢ a ≤a a'; is-new-action (action-obs E a') ⟧ ⟹ is-new-action (action-obs E a)
by(auto elim: action-orderE)
```

8.4.5 Program order

```
lemma program-orderI:
  assumes E ⊢ a ≤a a' and action-tid E a = action-tid E a'
  shows E ⊢ a ≤po a'
using assms unfolding program-order-def by auto

lemma program-orderE:
  assumes E ⊢ a ≤po a'
  obtains t obs obs'
  where E ⊢ a ≤a a'
  and action-tid E a = t action-obs E a = obs
  and action-tid E a' = t action-obs E a' = obs'
using assms unfolding program-order-def
by(cases lnth E a)(cases lnth E a', auto simp add: action-obs-def action-tid-def)

lemma refl-on-program-order:
  refl-onP (actions E) (program-order E)
by(rule refl-onPI)(auto elim: action-orderE program-orderE intro: program-orderI refl-onPD[OF refl-action-order])

lemma antisym-program-order:
  antisymp (program-order E)
using antisympD[OF antisym-action-order]
by(auto intro: antisympI elim!: program-orderE)

lemma trans-program-order:
  transp (program-order E)
by(rule transpI)(auto elim!: program-orderE intro: program-orderI dest: transPD[OF trans-action-order])
```

lemma *porder-program-order*:
porder-on (*actions E*) (*program-order E*)
by(*blast intro*: *porder-onI refl-on-program-order antisym-program-order trans-program-order*)

lemma *total-program-order-on-tactions*:
total-onP (*tactions E t*) (*program-order E*)
by(*rule total-onPI*)(*auto elim*: *tactionsE simp add*: *program-order-def dest*: *total-onD*[*OF total-action-order*])

8.4.6 Synchronization order

lemma *sync-orderI*:
⟦ $E \vdash a \leq_a a'$; $a \in$ *sactions P E*; $a' \in$ *sactions P E* ⟧ ⟹ $P,E \vdash a \leq_{so} a'$
unfolding *sync-order-def* **by** *blast*

lemma *sync-orderE*:
assumes $P,E \vdash a \leq_{so} a'$
obtains $a \in$ *sactions P E* $a' \in$ *sactions P E* $E \vdash a \leq_a a'$
using *assms* **unfolding** *sync-order-def* **by** *blast*

lemma *refl-on-sync-order*:
refl-onP (*sactions P E*) (*sync-order P E*)
by(*rule refl-onPI*)(*fastforce elim*: *sync-orderE intro*: *sync-orderI refl-onPD*[*OF refl-action-order*])+

lemma *antisym-sync-order*:
antisymp (*sync-order P E*)
using *antisympD*[*OF antisym-action-order*]
by(*rule antisympI*)(*auto elim*!: *sync-orderE*)

lemma *trans-sync-order*:
transp (*sync-order P E*)
by(*rule transpI*)(*auto elim*!: *sync-orderE intro*: *sync-orderI dest*: *transPD*[*OF trans-action-order*])

lemma *porder-sync-order*:
porder-on (*sactions P E*) (*sync-order P E*)
by(*blast intro*: *porder-onI refl-on-sync-order antisym-sync-order trans-sync-order*)

lemma *total-sync-order*:
total-onP (*sactions P E*) (*sync-order P E*)
apply(*rule total-onPI*)
apply(*simp add*: *sync-order-def*)
apply(*rule total-onPD*[*OF total-action-order*])
apply *simp-all*
done

lemma *torder-sync-order*:
torder-on (*sactions P E*) (*sync-order P E*)
by(*blast intro*: *torder-onI porder-sync-order total-sync-order*)

8.4.7 Synchronizes with

lemma *sync-withI*:
⟦ $P,E \vdash a \leq_{so} a'$; $P \vdash$ (*action-tid E a*, *action-obs E a*) $\rightsquigarrow_{sw}$ (*action-tid E a'*, *action-obs E a'*) ⟧
⟹ $P,E \vdash a \leq_{sw} a'$

unfolding *sync-with-def* **by** *blast*

lemma *sync-withE*:
assumes $P,E \vdash a \leq_{sw} a'$
obtains $P,E \vdash a \leq_{so} a'$ $P \vdash$ (*action-tid E a*, *action-obs E a*) $\rightsquigarrow_{sw}$ (*action-tid E a'*, *action-obs E a'*)
using *assms* **unfolding** *sync-with-def* **by** *blast*

lemma *irrefl-synchronizes-with*:
irreflP (*synchronizes-with P*)
by(*rule irreflPI*)(*auto elim*: *synchronizes-with.cases*)

lemma *irrefl-sync-with*:
irreflP (*sync-with P E*)
by(*rule irreflPI*)(*auto elim*: *sync-withE intro*: *irreflPD*[*OF irrefl-synchronizes-with*])

lemma *anitsym-sync-with*:
antisymp (*sync-with P E*)
using *antisymPD*[*OF antisym-sync-order*, *of P E*]
by −(*rule antisymPI*, *auto elim*: *sync-withE*)

lemma *consistent-program-order-sync-order*:
order-consistent (*program-order E*) (*sync-order P E*)
apply(*rule order-consistent-subset*)
apply(*rule antisym-order-consistent-self*[*OF antisym-action-order*[*of E*]])
apply(*blast elim*: *program-orderE sync-orderE*)+
done

lemma *consistent-program-order-sync-with*:
order-consistent (*program-order E*) (*sync-with P E*)
by(*rule order-consistent-subset*[*OF consistent-program-order-sync-order*])(*blast elim*: *sync-withE*)+

8.4.8 Happens before

lemma *porder-happens-before*:
porder-on (*actions E*) (*happens-before P E*)
unfolding *po-sw-def* [*abs-def*]
by(*rule porder-on-sub-torder-on-tranclp-porder-onI*[*OF porder-program-order torder-sync-order consistent-program-order-sync-order*])(*auto elim*: *sync-withE*)

lemma *porder-tranclp-po-so*:
porder-on (*actions E*) (λ*a a'*. *program-order E a a'* ∨ *sync-order P E a a'*)^++
by(*rule porder-on-torder-on-tranclp-porder-onI*[*OF porder-program-order torder-sync-order consistent-program-order-sync-ord*
auto

lemma *happens-before-refl*:
assumes $a \in$ *actions E*
shows $P,E \vdash a \leq_{hb} a$
using *porder-happens-before*[*of E P*]
by(*rule porder-onE*)(*erule refl-onPD*[*OF - assms*])

lemma *happens-before-into-po-so-tranclp*:
assumes $P,E \vdash a \leq_{hb} a'$
shows (λ*a a'*. $E \vdash a \leq_{po} a' \vee P,E \vdash a \leq_{so} a'$)^++ *a a'*
using *assms* **unfolding** *po-sw-def* [*abs-def*]

by(*induct*)(*blast elim*: *sync-withE intro*: *tranclp.trancl-into-trancl*)+

lemma *po-sw-into-action-order*:
  po-sw P E a a′ ⟹ *E* ⊢ *a* ≤*a a′*
by(*auto elim*: *program-orderE sync-withE sync-orderE simp add*: *po-sw-def*)

lemma *happens-before-into-action-order*:
  assumes *P*,*E* ⊢ *a* ≤*hb a′*
  shows *E* ⊢ *a* ≤*a a′*
using *assms*
by *induct*(*blast intro*: *po-sw-into-action-order transPD*[*OF trans-action-order*])+

lemma *action-order-consistent-with-happens-before*:
  order-consistent (*action-order E*) (*happens-before P E*)
by(*blast intro*: *order-consistent-subset antisym-order-consistent-self antisym-action-order happens-before-into-actio*

lemma *happens-before-new-actionD*:
  assumes *hb*: *P*,*E* ⊢ *a* ≤*hb a′*
  and *new*: *is-new-action* (*action-obs E a′*)
  shows *is-new-action* (*action-obs E a*) *action-tid E a* = *action-tid E a′ a* ≤ *a′*
using *hb*
proof(*induct rule*: *converse-tranclp-induct*)
  case (*base a*)

  case *1* **from** *new base* **show** *?case*
    by(*auto dest*: *po-sw-into-action-order elim*: *action-orderE*)
  case *2* **from** *new base* **show** *?case*
    by(*auto simp add*: *po-sw-def elim*!: *sync-withE elim*: *program-orderE synchronizes-with.cases*)
  case *3* **from** *new base* **show** *?case*
    by(*auto dest*: *po-sw-into-action-order elim*: *action-orderE*)
next
  case (*step a a″*)

  note *po-sw* = ‹*po-sw P E a a″*›
    and *new* = ‹*is-new-action* (*action-obs E a″*)›
    and *tid* = ‹*action-tid E a″* = *action-tid E a′*›

  case *1* **from** *new po-sw* **show** *?case*
    by(*auto dest*: *po-sw-into-action-order elim*: *action-orderE*)
  case *2* **from** *new po-sw tid* **show** *?case*
    by(*auto simp add*: *po-sw-def elim*!: *sync-withE elim*: *program-orderE synchronizes-with.cases*)
  case *3* **from** *new po-sw* ‹*a″* ≤ *a′*› **show** *?case*
    by(*auto dest*!: *po-sw-into-action-order elim*!: *action-orderE*)
qed

lemma *external-actions-not-new*:
  ⟦ *a* ∈ *external-actions E*; *is-new-action* (*action-obs E a*) ⟧ ⟹ *False*
by(*erule external-actions.cases*)(*simp*)

8.4.9 Most recent writes and sequential consistency

lemma *most-recent-write-for-fun*:
  ⟦ *P*,*E* ⊢ *ra* ↝*mrw wa*; *P*,*E* ⊢ *ra* ↝*mrw wa′* ⟧ ⟹ *wa* = *wa′*
apply(*erule most-recent-write-for.cases*)+

apply *clarsimp*
apply(*erule meta-allE*)+
apply(*erule meta-impE*)
 apply(*rotate-tac 3*)
 apply *assumption*
apply(*erule* (*1*) *meta-impE*)
apply(*frule* (*1*) *action-loc-read-action-singleton*)
 apply(*rotate-tac 1*)
 apply *assumption*
apply(*fastforce dest*: *antisymPD*[*OF antisym-action-order*] *elim*: *write-actions.cases read-actions.cases*)
done

lemma *THE-most-recent-writeI*: $P,E \vdash r \rightsquigarrow mrw\ w \Longrightarrow (THE\ w.\ P,E \vdash r \rightsquigarrow mrw\ w) = w$
by(*blast dest*: *most-recent-write-for-fun*)+

lemma *most-recent-write-for-write-actionsD*:
 assumes $P,E \vdash ra \rightsquigarrow mrw\ wa$
 shows $wa \in write\text{-}actions\ E$
using *assms* **by** *cases*

lemma *most-recent-write-recent*:
 $[\![\ P,E \vdash r \rightsquigarrow mrw\ w;\ adal \in action\text{-}loc\ P\ E\ r;\ w' \in write\text{-}actions\ E;\ adal \in action\text{-}loc\ P\ E\ w'\ ]\!]$
 $\Longrightarrow E \vdash w' \leq a\ w \vee E \vdash r \leq a\ w'$
apply(*erule most-recent-write-for.cases*)
apply(*drule* (*1*) *action-loc-read-action-singleton*)
 apply(*rotate-tac 1*)
 apply *assumption*
apply *clarsimp*
done

lemma *is-write-seenI*:
 $[\![\ \bigwedge a\ ad\ al\ v.\ [\![\ a \in read\text{-}actions\ E;\ action\text{-}obs\ E\ a = NormalAction\ (ReadMem\ ad\ al\ v)\ ]\!]$
 $\Longrightarrow ws\ a \in write\text{-}actions\ E;$
 $\bigwedge a\ ad\ al\ v.\ [\![\ a \in read\text{-}actions\ E;\ action\text{-}obs\ E\ a = NormalAction\ (ReadMem\ ad\ al\ v)\ ]\!]$
 $\Longrightarrow (ad,\ al) \in action\text{-}loc\ P\ E\ (ws\ a);$
 $\bigwedge a\ ad\ al\ v.\ [\![\ a \in read\text{-}actions\ E;\ action\text{-}obs\ E\ a = NormalAction\ (ReadMem\ ad\ al\ v)\ ]\!]$
 $\Longrightarrow value\text{-}written\ P\ E\ (ws\ a)\ (ad,\ al) = v;$
 $\bigwedge a\ ad\ al\ v.\ [\![\ a \in read\text{-}actions\ E;\ action\text{-}obs\ E\ a = NormalAction\ (ReadMem\ ad\ al\ v)\ ]\!]$
 $\Longrightarrow \neg\ P,E \vdash a \leq hb\ ws\ a;$
 $\bigwedge a\ ad\ al\ v.\ [\![\ a \in read\text{-}actions\ E;\ action\text{-}obs\ E\ a = NormalAction\ (ReadMem\ ad\ al\ v);\ is\text{-}volatile\ P\ al\ ]\!]$
 $\Longrightarrow \neg\ P,E \vdash a \leq so\ ws\ a;$
 $\bigwedge a\ ad\ al\ v\ a'.\ [\![\ a \in read\text{-}actions\ E;\ action\text{-}obs\ E\ a = NormalAction\ (ReadMem\ ad\ al\ v);$
 $a' \in write\text{-}actions\ E;\ (ad,\ al) \in action\text{-}loc\ P\ E\ a';\ P,E \vdash ws\ a \leq hb\ a';$
 $P,E \vdash a' \leq hb\ a\ ]\!] \Longrightarrow a' = ws\ a;$
 $\bigwedge a\ ad\ al\ v\ a'.\ [\![\ a \in read\text{-}actions\ E;\ action\text{-}obs\ E\ a = NormalAction\ (ReadMem\ ad\ al\ v);$
 $a' \in write\text{-}actions\ E;\ (ad,\ al) \in action\text{-}loc\ P\ E\ a';\ is\text{-}volatile\ P\ al;\ P,E \vdash ws\ a \leq so\ a';$
 $P,E \vdash a' \leq so\ a\ ]\!] \Longrightarrow a' = ws\ a\ ]\!]$
 $\Longrightarrow is\text{-}write\text{-}seen\ P\ E\ ws$
unfolding *is-write-seen-def*
by(*blast 30*)

lemma *is-write-seenD*:
 $[\![\ is\text{-}write\text{-}seen\ P\ E\ ws;\ a \in read\text{-}actions\ E;\ action\text{-}obs\ E\ a = NormalAction\ (ReadMem\ ad\ al\ v)\ ]\!]$

$\Longrightarrow$ *ws a* $\in$ *write-actions E* $\wedge$ (*ad*, *al*) $\in$ *action-loc P E* (*ws a*) $\wedge$ *value-written P E* (*ws a*) (*ad*, *al*) = *v* $\wedge$ $\neg$ *P*,*E* $\vdash$ *a* $\leq$*hb ws a* $\wedge$ (*is-volatile P al* $\longrightarrow$ $\neg$ *P*,*E* $\vdash$ *a* $\leq$*so ws a*) $\wedge$
(∀ *a'* $\in$ *write-actions E*. (*ad*, *al*) $\in$ *action-loc P E a'* $\wedge$ (*P*,*E* $\vdash$ *ws a* $\leq$*hb a'* $\wedge$ *P*,*E* $\vdash$ *a'* $\leq$*hb a* $\vee$ *is-volatile P al* $\wedge$ *P*,*E* $\vdash$ *ws a* $\leq$*so a'* $\wedge$ *P*,*E* $\vdash$ *a'* $\leq$*so a*) $\longrightarrow$ *a'* = *ws a*)
unfolding *is-write-seen-def* **by** *blast*

lemma *thread-start-actions-okI*:
(⋀*a*. ⟦ *a* $\in$ *actions E*; $\neg$ *is-new-action* (*action-obs E a*) ⟧
$\Longrightarrow$ ∃ *i*. *i* $\leq$ *a* $\wedge$ *action-obs E i* = *InitialThreadAction* $\wedge$ *action-tid E i* = *action-tid E a*)
$\Longrightarrow$ *thread-start-actions-ok E*
unfolding *thread-start-actions-ok-def* **by** *blast*

lemma *thread-start-actions-okD*:
⟦ *thread-start-actions-ok E*; *a* $\in$ *actions E*; $\neg$ *is-new-action* (*action-obs E a*) ⟧
$\Longrightarrow$ ∃ *i*. *i* $\leq$ *a* $\wedge$ *action-obs E i* = *InitialThreadAction* $\wedge$ *action-tid E i* = *action-tid E a*
unfolding *thread-start-actions-ok-def* **by** *blast*

lemma *thread-start-actions-ok-prefix*:
⟦ *thread-start-actions-ok E'*; *lprefix E E'* ⟧ $\Longrightarrow$ *thread-start-actions-ok E*
apply(*clarsimp simp add*: *lprefix-conv-lappend*)
apply(*rule thread-start-actions-okI*)
apply(*drule-tac a*=*a* **in** *thread-start-actions-okD*)
apply(*simp add*: *actions-def*)
apply(*auto simp add*: *action-obs-def lnth-lappend1 actions-def action-tid-def le-less-trans*[**where** *y*=*enat a* **for** *a*])
apply (*metis add.right-neutral add-strict-mono not-gr-zero*)
done

lemma *wf-execI* [*intro?*]:
⟦ *is-write-seen P E ws*;
thread-start-actions-ok E ⟧
$\Longrightarrow$ *P* $\vdash$ (*E*, *ws*) $\surd$
by *simp*

lemma *wf-exec-is-write-seenD*:
P $\vdash$ (*E*, *ws*) $\surd$ $\Longrightarrow$ *is-write-seen P E ws*
by *simp*

lemma *wf-exec-thread-start-actions-okD*:
P $\vdash$ (*E*, *ws*) $\surd$ $\Longrightarrow$ *thread-start-actions-ok E*
by *simp*

lemma *sequentially-consistentI*:
(⋀*r*. *r* $\in$ *read-actions E* $\Longrightarrow$ *P*,*E* $\vdash$ *r* $\rightsquigarrow$*mrw ws r*)
$\Longrightarrow$ *sequentially-consistent P* (*E*, *ws*)
by *simp*

lemma *sequentially-consistentE*:
assumes *sequentially-consistent P* (*E*, *ws*) *a* $\in$ *read-actions E*
obtains *P*,*E* $\vdash$ *a* $\rightsquigarrow$*mrw ws a*
using *assms* **by** *simp*

declare *sequentially-consistent.simps* [*simp del*]

8.4.10 Similar actions

Similar actions differ only in the values written/read.

inductive *sim-action* ::
  (*'addr*, *'thread-id*) *obs-event action* ⇒ (*'addr*, *'thread-id*) *obs-event action* ⇒ *bool*
  (‹- ≈ -› [*50*, *50*] *51*)
where
  InitialThreadAction: *InitialThreadAction* ≈ *InitialThreadAction*
| *ThreadFinishAction*: *ThreadFinishAction* ≈ *ThreadFinishAction*
| *NewHeapElem*: *NormalAction* (*NewHeapElem a hT*) ≈ *NormalAction* (*NewHeapElem a hT*)
| *ReadMem*: *NormalAction* (*ReadMem ad al v*) ≈ *NormalAction* (*ReadMem ad al v'*)
| *WriteMem*: *NormalAction* (*WriteMem ad al v*) ≈ *NormalAction* (*WriteMem ad al v'*)
| *ThreadStart*: *NormalAction* (*ThreadStart t*) ≈ *NormalAction* (*ThreadStart t*)
| *ThreadJoin*: *NormalAction* (*ThreadJoin t*) ≈ *NormalAction* (*ThreadJoin t*)
| *SyncLock*: *NormalAction* (*SyncLock a*) ≈ *NormalAction* (*SyncLock a*)
| *SyncUnlock*: *NormalAction* (*SyncUnlock a*) ≈ *NormalAction* (*SyncUnlock a*)
| *ExternalCall*: *NormalAction* (*ExternalCall a M vs v*) ≈ *NormalAction* (*ExternalCall a M vs v*)
| *ObsInterrupt*: *NormalAction* (*ObsInterrupt t*) ≈ *NormalAction* (*ObsInterrupt t*)
| *ObsInterrupted*: *NormalAction* (*ObsInterrupted t*) ≈ *NormalAction* (*ObsInterrupted t*)

definition *sim-actions* :: (*'addr*, *'thread-id*) *execution* ⇒ (*'addr*, *'thread-id*) *execution* ⇒ *bool* (‹- [≈]
-› [*51*, *50*] *51*)
where *sim-actions* = *llist-all2* (λ(*t*, *a*) (*t'*, *a'*). *t* = *t'* ∧ *a* ≈ *a'*)

lemma *sim-action-refl* [*intro*!, *simp*]:
  obs ≈ *obs*
apply(*cases obs*)
 apply(*rename-tac obs'*)
 apply(*case-tac obs'*)
apply(*auto intro*: *sim-action.intros*)
done

inductive-cases *sim-action-cases* [*elim*!]:
  InitialThreadAction ≈ *obs*
  ThreadFinishAction ≈ *obs*
  NormalAction (*NewHeapElem a hT*) ≈ *obs*
  NormalAction (*ReadMem ad al v*) ≈ *obs*
  NormalAction (*WriteMem ad al v*) ≈ *obs*
  NormalAction (*ThreadStart t*) ≈ *obs*
  NormalAction (*ThreadJoin t*) ≈ *obs*
  NormalAction (*SyncLock a*) ≈ *obs*
  NormalAction (*SyncUnlock a*) ≈ *obs*
  NormalAction (*ObsInterrupt t*) ≈ *obs*
  NormalAction (*ObsInterrupted t*) ≈ *obs*
  NormalAction (*ExternalCall a M vs v*) ≈ *obs*

  obs ≈ *InitialThreadAction*
  obs ≈ *ThreadFinishAction*
  obs ≈ *NormalAction* (*NewHeapElem a hT*)
  obs ≈ *NormalAction* (*ReadMem ad al v'*)
  obs ≈ *NormalAction* (*WriteMem ad al v'*)
  obs ≈ *NormalAction* (*ThreadStart t*)
  obs ≈ *NormalAction* (*ThreadJoin t*)
  obs ≈ *NormalAction* (*SyncLock a*)

obs ≈ *NormalAction* (*SyncUnlock a*)
obs ≈ *NormalAction* (*ObsInterrupt t*)
obs ≈ *NormalAction* (*ObsInterrupted t*)
obs ≈ *NormalAction* (*ExternalCall a M vs v*)

inductive-simps *sim-action-simps* [*simp*]:
InitialThreadAction ≈ *obs*
ThreadFinishAction ≈ *obs*
NormalAction (*NewHeapElem a hT*) ≈ *obs*
NormalAction (*ReadMem ad al v*) ≈ *obs*
NormalAction (*WriteMem ad al v*) ≈ *obs*
NormalAction (*ThreadStart t*) ≈ *obs*
NormalAction (*ThreadJoin t*) ≈ *obs*
NormalAction (*SyncLock a*) ≈ *obs*
NormalAction (*SyncUnlock a*) ≈ *obs*
NormalAction (*ObsInterrupt t*) ≈ *obs*
NormalAction (*ObsInterrupted t*) ≈ *obs*
NormalAction (*ExternalCall a M vs v*) ≈ *obs*

obs ≈ *InitialThreadAction*
obs ≈ *ThreadFinishAction*
obs ≈ *NormalAction* (*NewHeapElem a hT*)
obs ≈ *NormalAction* (*ReadMem ad al v′*)
obs ≈ *NormalAction* (*WriteMem ad al v′*)
obs ≈ *NormalAction* (*ThreadStart t*)
obs ≈ *NormalAction* (*ThreadJoin t*)
obs ≈ *NormalAction* (*SyncLock a*)
obs ≈ *NormalAction* (*SyncUnlock a*)
obs ≈ *NormalAction* (*ObsInterrupt t*)
obs ≈ *NormalAction* (*ObsInterrupted t*)
obs ≈ *NormalAction* (*ExternalCall a M vs v*)

lemma *sim-action-trans* [*trans*]:
⟦ *obs* ≈ *obs′*; *obs′* ≈ *obs″* ⟧ ⟹ *obs* ≈ *obs″*
by(*erule sim-action.cases*) *auto*

lemma *sim-action-sym* [*sym*]:
assumes *obs* ≈ *obs′*
shows *obs′* ≈ *obs*
using *assms* **by** *cases simp-all*

lemma *sim-actions-sym* [*sym*]:
E [≈] *E′* ⟹ *E′* [≈] *E*
unfolding *sim-actions-def*
by(*auto simp add*: *llist-all2-conv-all-lnth split-beta intro*: *sim-action-sym*)

lemma *sim-actions-action-obsD*:
E [≈] *E′* ⟹ *action-obs E a* ≈ *action-obs E′ a*
unfolding *sim-actions-def action-obs-def*
by(*cases enat a* < *llength E*)(*auto dest*: *llist-all2-lnthD llist-all2-llengthD simp add*: *split-beta lnth-beyond split*: *enat.split*)

lemma *sim-actions-action-tidD*:
E [≈] *E′* ⟹ *action-tid E a* = *action-tid E′ a*

unfolding *sim-actions-def action-tid-def*
by(*cases enat a < llength E*)(*auto dest*: *llist-all2-lnthD llist-all2-llengthD simp add*: *lnth-beyond split*: *enat.split*)

lemma *action-loc-aux-sim-action*:
  $a \approx a' \Longrightarrow$ *action-loc-aux P a* = *action-loc-aux P a'*
by(*auto elim*!: *action-loc-aux-cases intro*: *action-loc-aux-intros*)

lemma *eq-into-sim-actions*:
  assumes $E = E'$
  shows E [$\approx$] E'
unfolding *sim-actions-def assms*
by(*rule llist-all2-reflI*)(*auto*)

8.4.11 Well-formedness conditions for execution sets

locale *executions-base* =
  fixes $\mathcal{E}$ :: (*'addr*, *'thread-id*) *execution set*
  and P :: *'m prog*

locale *drf* =
  executions-base $\mathcal{E}$ *P*
  for $\mathcal{E}$ :: (*'addr*, *'thread-id*) *execution set*
  and P :: *'m prog* +
  assumes $\mathcal{E}$*-new-actions-for-fun*:
  ⟦ $E \in \mathcal{E}$; $a \in$ *new-actions-for P E adal*; $a' \in$ *new-actions-for P E adal* ⟧ $\Longrightarrow a = a'$
  and $\mathcal{E}$*-sequential-completion*:
  ⟦ $E \in \mathcal{E}$; $P \vdash (E, ws)\ \surd$; $\bigwedge a.$ ⟦ $a < r$; $a \in$ *read-actions E* ⟧ $\Longrightarrow P,E \vdash a \rightsquigarrow$*mrw ws a* ⟧
  $\Longrightarrow \exists E' \in \mathcal{E}.\ \exists ws'.\ P \vdash (E', ws')\ \surd \wedge$ *ltake* (*enat r*) *E* = *ltake* (*enat r*) *E'* $\wedge$ *sequentially-consistent* *P* (*E'*, *ws'*) $\wedge$
  action-tid E r = *action-tid E' r* $\wedge$ *action-obs E r* $\approx$ *action-obs E' r* $\wedge$
  ($r \in$ *actions E* $\longrightarrow$ $r \in$ *actions E'*)

locale *executions-aux* =
  executions-base $\mathcal{E}$ *P*
  for $\mathcal{E}$ :: (*'addr*, *'thread-id*) *execution set*
  and P :: *'m prog* +
  assumes *init-before-read*:
  ⟦ $E \in \mathcal{E}$; $P \vdash (E, ws)\ \surd$; $r \in$ *read-actions E*; *adal* $\in$ *action-loc P E r*;
  $\bigwedge a.$ ⟦ $a < r$; $a \in$ *read-actions E* ⟧ $\Longrightarrow P,E \vdash a \rightsquigarrow$*mrw ws a* ⟧
  $\Longrightarrow \exists i{<}r.\ i \in$ *new-actions-for P E adal*
  and $\mathcal{E}$*-new-actions-for-fun*:
  ⟦ $E \in \mathcal{E}$; $a \in$ *new-actions-for P E adal*; $a' \in$ *new-actions-for P E adal* ⟧ $\Longrightarrow a = a'$

locale *sc-legal* =
  executions-aux $\mathcal{E}$ *P*
  for $\mathcal{E}$ :: (*'addr*, *'thread-id*) *execution set*
  and P :: *'m prog* +
  assumes $\mathcal{E}$*-hb-completion*:
  ⟦ $E \in \mathcal{E}$; $P \vdash (E, ws)\ \surd$; $\bigwedge a.$ ⟦ $a < r$; $a \in$ *read-actions E* ⟧ $\Longrightarrow P,E \vdash a \rightsquigarrow$*mrw ws a* ⟧
  $\Longrightarrow \exists E' \in \mathcal{E}.\ \exists ws'.\ P \vdash (E', ws')\ \surd \wedge$ *ltake* (*enat r*) *E* = *ltake* (*enat r*) *E'* $\wedge$
  ($\forall a \in$ *read-actions E'*. *if a < r then ws' a* = *ws a else P,E'* $\vdash$ *ws' a* $\leq$*hb a*) $\wedge$
  action-tid E' r = *action-tid E r* $\wedge$
  (*if r* $\in$ *read-actions E then sim-action else* (=)) (*action-obs E' r*) (*action-obs E r*) $\wedge$

$(r \in \mathit{actions}\ E \longrightarrow r \in \mathit{actions}\ E')$

locale *jmm-consistent* =
drf?: *drf* $\mathcal{E}$ P +
sc-legal $\mathcal{E}$ P
for $\mathcal{E}$ :: (*'addr*, *'thread-id*) *execution set*
and P :: *'m prog*

8.4.12 Legal executions

record (*'addr*, *'thread-id*) *pre-justifying-execution* =
committed :: *JMM-action set*
justifying-exec :: (*'addr*, *'thread-id*) *execution*
justifying-ws :: *write-seen*

record (*'addr*, *'thread-id*) *justifying-execution* =
(*'addr*, *'thread-id*) *pre-justifying-execution* +
action-translation :: *JMM-action* ⇒ *JMM-action*

type-synonym (*'addr*, *'thread-id*) *justification* = *nat* ⇒ (*'addr*, *'thread-id*) *justifying-execution*

definition *wf-action-translation-on* ::
(*'addr*, *'thread-id*) *execution* ⇒ (*'addr*, *'thread-id*) *execution* ⇒ *JMM-action set* ⇒ (*JMM-action* ⇒ *JMM-action*) ⇒ *bool*
where
wf-action-translation-on E E' A f ⟷
inj-on f (*actions* E) ∧
(∀ $a \in A$. *action-tid* E a = *action-tid* E' (f a) ∧ *action-obs* E a ≈ *action-obs* E' (f a))

abbreviation *wf-action-translation* :: (*'addr*, *'thread-id*) *execution* ⇒ (*'addr*, *'thread-id*) *justifying-execution* ⇒ *bool*
where
wf-action-translation E J ≡
wf-action-translation-on (*justifying-exec* J) E (*committed* J) (*action-translation* J)

context
fixes P :: *'m prog*
and E :: (*'addr*, *'thread-id*) *execution*
and *ws* :: *write-seen*
and J :: (*'addr*, *'thread-id*) *justification*
begin

This context defines the causality constraints for the JMM. The weak versions are for the fixed JMM as presented by Sevcik and Aspinall at ECOOP 2008.

Committed actions are an ascending chain with all actions of E as a limit

definition *is-commit-sequence* :: *bool* **where**
is-commit-sequence ⟷
committed (J 0) = {} ∧
(∀ n. *action-translation* (J n) ‘ *committed* (J n) ⊆ *action-translation* (J (*Suc* n)) ‘ *committed* (J (*Suc* n))) ∧
actions E = (⋃ n. *action-translation* (J n) ‘ *committed* (J n))

definition *justification-well-formed* :: *bool* **where**
justification-well-formed ⟷ (∀ n. P ⊢ (*justifying-exec* (J n), *justifying-ws* (J n)) √)

definition *committed-subset-actions* :: *bool* **where** — JMM constraint 1
committed-subset-actions $\longleftrightarrow$ ($\forall$ *n*. *committed* (*J n*) $\subseteq$ *actions* (*justifying-exec* (*J n*)))

definition *happens-before-committed* :: *bool* **where** — JMM constraint 2
happens-before-committed $\longleftrightarrow$
($\forall$ *n*. *happens-before P* (*justifying-exec* (*J n*)) |‘ *committed* (*J n*) =
inv-imageP (*happens-before P E*) (*action-translation* (*J n*)) |‘ *committed* (*J n*))

definition *happens-before-committed-weak* :: *bool* **where** — relaxed JMM constraint
happens-before-committed-weak $\longleftrightarrow$
($\forall$ *n*. $\forall$ *r* $\in$ *read-actions* (*justifying-exec* (*J n*)) $\cap$ *committed* (*J n*).
let r′ = *action-translation* (*J n*) *r*;
w′ = *ws r′*;
w = *inv-into* (*actions* (*justifying-exec* (*J n*))) (*action-translation* (*J n*)) *w′ in*
(*P*,*E* $\vdash$ *w′* $\leq_{hb}$ *r′* $\longleftrightarrow$ *P*,*justifying-exec* (*J n*) $\vdash$ *w* $\leq_{hb}$ *r*) $\wedge$
$\neg$ *P*,*justifying-exec* (*J n*) $\vdash$ *r* $\leq_{hb}$ *w*)

definition *sync-order-committed* :: *bool* **where** — JMM constraint 3
sync-order-committed $\longleftrightarrow$
($\forall$ *n*. *sync-order P* (*justifying-exec* (*J n*)) |‘ *committed* (*J n*) =
inv-imageP (*sync-order P E*) (*action-translation* (*J n*)) |‘ *committed* (*J n*))

definition *value-written-committed* :: *bool* **where** — JMM constraint 4
value-written-committed $\longleftrightarrow$
($\forall$ *n*. $\forall$ *w* $\in$ *write-actions* (*justifying-exec* (*J n*)) $\cap$ *committed* (*J n*).
let w′ = *action-translation* (*J n*) *w*
in ($\forall$ *adal* $\in$ *action-loc P E w′*. *value-written P* (*justifying-exec* (*J n*)) *w adal* = *value-written P E w′ adal*))

definition *write-seen-committed* :: *bool* **where** — JMM constraint 5
write-seen-committed $\longleftrightarrow$
($\forall$ *n*. $\forall$ *r′* $\in$ *read-actions* (*justifying-exec* (*J n*)) $\cap$ *committed* (*J n*).
let r = *action-translation* (*J n*) *r′*;
r″ = *inv-into* (*actions* (*justifying-exec* (*J* (*Suc n*)))) (*action-translation* (*J* (*Suc n*))) *r*
in action-translation (*J* (*Suc n*)) (*justifying-ws* (*J* (*Suc n*)) *r″*) = *ws r*)

uncommited reads see writes that happen before them – JMM constraint 6

definition *uncommitted-reads-see-hb* :: *bool* **where**
uncommitted-reads-see-hb $\longleftrightarrow$
($\forall$ *n*. $\forall$ *r′* $\in$ *read-actions* (*justifying-exec* (*J* (*Suc n*))).
action-translation (*J* (*Suc n*)) *r′* $\in$ *action-translation* (*J n*) ‘ *committed* (*J n*) $\vee$
P,*justifying-exec* (*J* (*Suc n*)) $\vdash$ *justifying-ws* (*J* (*Suc n*)) *r′* $\leq_{hb}$ *r′*)

newly committed reads see already committed writes and write-seen relationship must not change any more – JMM constraint 7

definition *committed-reads-see-committed-writes* :: *bool* **where**
committed-reads-see-committed-writes $\longleftrightarrow$
($\forall$ *n*. $\forall$ *r′* $\in$ *read-actions* (*justifying-exec* (*J* (*Suc n*))) $\cap$ *committed* (*J* (*Suc n*)).
let r = *action-translation* (*J* (*Suc n*)) *r′*;
committed-n = *action-translation* (*J n*) ‘ *committed* (*J n*)
in r $\in$ *committed-n* $\vee$
(*action-translation* (*J* (*Suc n*)) (*justifying-ws* (*J* (*Suc n*)) *r′*) $\in$ *committed-n* $\wedge$ *ws r* $\in$ *committed-n*))

definition *committed-reads-see-committed-writes-weak* :: *bool* **where**
committed-reads-see-committed-writes-weak ⟷
(∀ *n*. ∀ *r'* ∈ *read-actions* (*justifying-exec* (*J* (*Suc n*))) ∩ *committed* (*J* (*Suc n*)).
let r = *action-translation* (*J* (*Suc n*)) *r'*;
committed-n = *action-translation* (*J n*) ‘ *committed* (*J n*)
in r ∈ *committed-n* ∨ *ws r* ∈ *committed-n*)

external actions must be committed as soon as hb-subsequent actions are committed – JMM constraint 9

definition *external-actions-committed* :: *bool* **where**
external-actions-committed ⟷
(∀ *n*. ∀ *a* ∈ *external-actions* (*justifying-exec* (*J n*)). ∀ *a'* ∈ *committed* (*J n*).
P,*justifying-exec* (*J n*) ⊢ *a* ≤*hb a'* ⟶ *a* ∈ *committed* (*J n*))

well-formedness conditions for action translations

definition *wf-action-translations* :: *bool* **where**
wf-action-translations ⟷
(∀ *n*. *wf-action-translation-on* (*justifying-exec* (*J n*)) *E* (*committed* (*J n*)) (*action-translation* (*J n*)))

end

Rule 8 of the justification for the JMM is incorrect because there might be no transitive reduction of the happens-before relation for an infinite execution, if infinitely many initialisation actions have to be ordered before the start action of every thread. Hence, *is-justified-by* omits this constraint.

primrec *is-justified-by* ::
'm prog ⇒ (*'addr*, *'thread-id*) *execution* × *write-seen* ⇒ (*'addr*, *'thread-id*) *justification* ⇒ *bool*
(‹- ⊢ - *justified'-by* -› [*51*, *50*, *50*] *50*)
where
P ⊢ (*E*, *ws*) *justified-by J* ⟷
is-commit-sequence E J ∧
justification-well-formed P J ∧
committed-subset-actions J ∧
happens-before-committed P E J ∧
sync-order-committed P E J ∧
value-written-committed P E J ∧
write-seen-committed ws J ∧
uncommitted-reads-see-hb P J ∧
committed-reads-see-committed-writes ws J ∧
external-actions-committed P J ∧
wf-action-translations E J

Sevcik requires in the fixed JMM that external actions may only be committed when everything that happens before has already been committed. On the level of legality, this constraint is vacuous because it is always possible to delay committing external actions, so we omit it here.

primrec *is-weakly-justified-by* ::
'm prog ⇒ (*'addr*, *'thread-id*) *execution* × *write-seen* ⇒ (*'addr*, *'thread-id*) *justification* ⇒ *bool*
(‹- ⊢ - *weakly'-justified'-by* -› [*51*, *50*, *50*] *50*)
where
P ⊢ (*E*, *ws*) *weakly-justified-by J* ⟷
is-commit-sequence E J ∧
justification-well-formed P J ∧

committed-subset-actions J ∧
happens-before-committed-weak P E ws J ∧
— no *sync-order* constraint
value-written-committed P E J ∧
write-seen-committed ws J ∧
uncommitted-reads-see-hb P J ∧
committed-reads-see-committed-writes-weak ws J ∧
wf-action-translations E J

Notion of conflict is strengthened to explicitly exclude volatile locations. Otherwise, the following program is not correctly synchronised:

```
volatile x = 0;
---------------
r = x; | x = 1;
```

because in the SC execution [Init x 0, (t1, Read x 0), (t2, Write x 1)], the read and write are unrelated in hb, because synchronises-with is asymmetric for volatiles.

The JLS considers conflicting volatiles for data races, but this is only a remark on the DRF guarantee. See JMM mailing list posts #2477 to 2488.

definition *non-volatile-conflict* ::
'm prog ⇒ (*'addr*, *'thread-id*) *execution* ⇒ *JMM-action* ⇒ *JMM-action* ⇒ *bool*
(‹-,- ⊢/(-)†(-)› [*51*,*50*,*50*,*50*] *51*)
where
P,*E* ⊢ *a* † *a'* ⟷
(*a* ∈ *read-actions E* ∧ *a'* ∈ *write-actions E* ∨
a ∈ *write-actions E* ∧ *a'* ∈ *read-actions E* ∨
a ∈ *write-actions E* ∧ *a'* ∈ *write-actions E*) ∧
(∃ *ad al*. (*ad*, *al*) ∈ *action-loc P E a* ∩ *action-loc P E a'* ∧ ¬ *is-volatile P al*)

definition *correctly-synchronized* :: *'m prog* ⇒ (*'addr*, *'thread-id*) *execution set* ⇒ *bool*
where
correctly-synchronized P $\mathcal{E}$ ⟷
(∀ *E* ∈ $\mathcal{E}$. ∀ *ws*. *P* ⊢ (*E*, *ws*) √ ⟶ *sequentially-consistent P* (*E*, *ws*)
⟶ (∀ *a* ∈ *actions E*. ∀ *a'* ∈ *actions E*. *P*,*E* ⊢ *a* † *a'*
⟶ *P*,*E* ⊢ *a* ≤*hb a'* ∨ *P*,*E* ⊢ *a'* ≤*hb a*))

primrec *gen-legal-execution* ::
(*'m prog* ⇒ (*'addr*, *'thread-id*) *execution* × *write-seen* ⇒ (*'addr*, *'thread-id*) *justification* ⇒ *bool*)
⇒ *'m prog* ⇒ (*'addr*, *'thread-id*) *execution set* ⇒ (*'addr*, *'thread-id*) *execution* × *write-seen* ⇒ *bool*
where
gen-legal-execution is-justification P $\mathcal{E}$ (*E*, *ws*) ⟷
E ∈ $\mathcal{E}$ ∧ *P* ⊢ (*E*, *ws*) √ ∧
(∃ *J*. *is-justification P* (*E*, *ws*) *J* ∧ *range* (*justifying-exec* ∘ *J*) ⊆ $\mathcal{E}$)

abbreviation *legal-execution* ::
'm prog ⇒ (*'addr*, *'thread-id*) *execution set* ⇒ (*'addr*, *'thread-id*) *execution* × *write-seen* ⇒ *bool*
where
legal-execution ≡ *gen-legal-execution is-justified-by*

abbreviation *weakly-legal-execution* ::
'm prog ⇒ (*'addr*, *'thread-id*) *execution set* ⇒ (*'addr*, *'thread-id*) *execution* × *write-seen* ⇒ *bool*
where

weakly-legal-execution ≡ *gen-legal-execution is-weakly-justified-by*

declare *gen-legal-execution.simps* [*simp del*]

lemma *sym-non-volatile-conflict*:
symP (*non-volatile-conflict P E*)
unfolding *non-volatile-conflict-def*
by(*rule symPI*) *blast*

lemma *legal-executionI*:
⟦ $E \in \mathcal{E}$; $P \vdash (E, ws)\ \surd$; *is-justification P* (*E*, *ws*) *J*; *range* (*justifying-exec* ∘ *J*) ⊆ $\mathcal{E}$ ⟧
⟹ *gen-legal-execution is-justification P* $\mathcal{E}$ (*E*, *ws*)
unfolding *gen-legal-execution.simps* **by** *blast*

lemma *legal-executionE*:
assumes *gen-legal-execution is-justification P* $\mathcal{E}$ (*E*, *ws*)
obtains *J* **where** $E \in \mathcal{E}$ $P \vdash (E, ws)\ \surd$ *is-justification P* (*E*, *ws*) *J range* (*justifying-exec* ∘ *J*) ⊆ $\mathcal{E}$
using *assms* **unfolding** *gen-legal-execution.simps* **by** *blast*

lemma *legal-*$\mathcal{E}$*D*: *gen-legal-execution is-justification P* $\mathcal{E}$ (*E*, *ws*) ⟹ $E \in \mathcal{E}$
by(*erule legal-executionE*)

lemma *legal-wf-execD*:
gen-legal-execution is-justification P $\mathcal{E}$ *Ews* ⟹ $P \vdash Ews\ \surd$
by(*cases Ews*)(*auto elim*: *legal-executionE*)

lemma *correctly-synchronizedD*:
⟦ *correctly-synchronized P* $\mathcal{E}$; $E \in \mathcal{E}$; $P \vdash (E, ws)\ \surd$; *sequentially-consistent P* (*E*, *ws*) ⟧
⟹ ∀ *a a'*. *a* ∈ *actions E* ⟶ *a'* ∈ *actions E* ⟶ *P*,*E* ⊢ *a* ↑ *a'* ⟶ *P*,*E* ⊢ *a* ≤*hb a'* ∨ *P*,*E* ⊢ *a'* ≤*hb a*
unfolding *correctly-synchronized-def* **by** *blast*

lemma *wf-action-translation-on-actionD*:
⟦ *wf-action-translation-on E E' A f*; *a* ∈ *A* ⟧
⟹ *action-tid E a* = *action-tid E'* (*f a*) ∧ *action-obs E a* ≈ *action-obs E'* (*f a*)
unfolding *wf-action-translation-on-def* **by** *blast*

lemma *wf-action-translation-on-inj-onD*:
wf-action-translation-on E E' A f ⟹ *inj-on f* (*actions E*)
unfolding *wf-action-translation-on-def* **by** *simp*

lemma *wf-action-translation-on-action-locD*:
⟦ *wf-action-translation-on E E' A f*; *a* ∈ *A* ⟧
⟹ *action-loc P E a* = *action-loc P E'* (*f a*)
apply(*drule* (*1*) *wf-action-translation-on-actionD*)
apply(*cases* (*P*, *action-obs E a*) *rule*: *action-loc-aux.cases*)
apply *auto*
done

lemma *weakly-justified-write-seen-hb-read-committed*:
assumes *J*: *P* ⊢ (*E*, *ws*) *weakly-justified-by J*
and *r*: *r* ∈ *read-actions* (*justifying-exec* (*J n*)) *r* ∈ *committed* (*J n*)
shows *ws* (*action-translation* (*J n*) *r*) ∈ *action-translation* (*J n*) ‘ *committed* (*J n*)
using *r*

proof(*induct n arbitrary*: *r*)
case *0*
from *J* **have** [*simp*]: *committed* (*J 0*) = {}
by(*simp add*: *is-commit-sequence-def*)
with *0* **show** *?case* **by** *simp*
next
case (*Suc n*)
let *?E* = λ*n*. *justifying-exec* (*J n*)
and *?ws* = λ*n*. *justifying-ws* (*J n*)
and *?C* = λ*n*. *committed* (*J n*)
and *?φ* = λ*n*. *action-translation* (*J n*)

note *r* = ‹*r* ∈ *read-actions* (*?E* (*Suc n*))›
hence *r* ∈ *actions* (*?E* (*Suc n*)) **by** *simp*

from *J* **have** *wfan*: *wf-action-translation-on* (*?E n*) *E* (*?C n*) (*?φ n*)
and *wfaSn*: *wf-action-translation-on* (*?E* (*Suc n*)) *E* (*?C* (*Suc n*)) (*?φ* (*Suc n*))
by(*simp-all add*: *wf-action-translations-def*)

from *wfaSn* **have** *injSn*: *inj-on* (*?φ* (*Suc n*)) (*actions* (*?E* (*Suc n*)))
by(*rule wf-action-translation-on-inj-onD*)
from *J* **have** *C-sub-A*: *?C* (*Suc n*) ⊆ *actions* (*?E* (*Suc n*))
by(*simp add*: *committed-subset-actions-def*)
from *J* **have** *CnCSn*: *?φ n* ‘ *?C n* ⊆ *?φ* (*Suc n*) ‘ *?C* (*Suc n*)
by(*simp add*: *is-commit-sequence-def*)

from *J* **have** *wsSn*: *is-write-seen P* (*?E* (*Suc n*)) (*?ws* (*Suc n*))
by(*simp add*: *justification-well-formed-def*)
from *r* **obtain** *ad al v* **where** *action-obs* (*?E* (*Suc n*)) *r* = *NormalAction* (*ReadMem ad al v*) **by** *cases*
from *is-write-seenD*[*OF wsSn r this*]
have *wsSn*: *?ws* (*Suc n*) *r* ∈ *actions* (*?E* (*Suc n*)) **by** *simp*

show *?case*
proof(*cases ?φ* (*Suc n*) *r* ∈ *?φ n* ‘ *?C n*)
case *True*
then obtain *r′* **where** *r′*: *r′* ∈ *?C n*
and *r-r′*: *?φ* (*Suc n*) *r* = *?φ n r′* **by**(*auto*)
from *r′ wfan* **have** *action-tid* (*?E n*) *r′* = *action-tid E* (*?φ n r′*)
and *action-obs* (*?E n*) *r′* ≈ *action-obs E* (*?φ n r′*)
by(*blast dest*: *wf-action-translation-on-actionD*)+
moreover from *r′ CnCSn* **have** *?φ* (*Suc n*) *r* ∈ *?φ* (*Suc n*) ‘ *?C* (*Suc n*)
unfolding *r-r′* **by** *auto*
hence *r* ∈ *?C* (*Suc n*)
unfolding *inj-on-image-mem-iff*[*OF injSn* ‹*r* ∈ *actions* (*?E* (*Suc n*))› *C-sub-A*] .
with *wfaSn* **have** *action-tid* (*?E* (*Suc n*)) *r* = *action-tid E* (*?φ* (*Suc n*) *r*)
and *action-obs* (*?E* (*Suc n*)) *r* ≈ *action-obs E* (*?φ* (*Suc n*) *r*)
by(*blast dest*: *wf-action-translation-on-actionD*)+
ultimately have *tid*: *action-tid* (*?E n*) *r′* = *action-tid* (*?E* (*Suc n*)) *r*
and *obs*: *action-obs* (*?E n*) *r′* ≈ *action-obs* (*?E* (*Suc n*)) *r*
unfolding *r-r′* **by**(*auto intro*: *sim-action-trans sim-action-sym*)

from *J* **have** *?C n* ⊆ *actions* (*?E n*) **by**(*simp add*: *committed-subset-actions-def*)
with *r′* **have** *r′* ∈ *actions* (*?E n*) **by** *blast*

```
    with r obs have r' ∈ read-actions (?E n)
      by cases(auto intro: read-actions.intros)
    hence ws: ws (?φ n r') ∈ ?φ n ‘ ?C n using r' by(rule Suc)

    have r-conv-inv: r = inv-into (actions (?E (Suc n))) (?φ (Suc n)) (?φ n r')
      using ‹r ∈ actions (?E (Suc n))› unfolding r-r'[symmetric]
      by(simp add: inv-into-f-f[OF injSn])
    with ‹r' ∈ ?C n› r J ‹r' ∈ read-actions (?E n)›
    have ws-eq: ?φ (Suc n) (?ws (Suc n) r) = ws (?φ n r')
      by(simp add: Let-def write-seen-committed-def)
    with ws CnCSn have ?φ (Suc n) (?ws (Suc n) r) ∈ ?φ (Suc n) ‘ ?C (Suc n) by auto
    hence ?ws (Suc n) r ∈ ?C (Suc n)
      by(subst (asm) inj-on-image-mem-iff[OF injSn wsSn C-sub-A])
    moreover from ws CnCSn have ws (?φ (Suc n) r) ∈ ?φ (Suc n) ‘ ?C (Suc n)
      unfolding r-r' by auto
    ultimately show ?thesis by simp
  next
    case False
    with r ‹r ∈ ?C (Suc n)› J
    have ws (?φ (Suc n) r) ∈ ?φ n ‘ ?C n
      unfolding is-weakly-justified-by.simps Let-def committed-reads-see-committed-writes-weak-def
      by blast
    hence ws (?φ (Suc n) r) ∈ ?φ (Suc n) ‘ ?C (Suc n)
      using CnCSn by blast+
    thus ?thesis by(simp add: inj-on-image-mem-iff[OF injSn wsSn C-sub-A])
  qed
qed

lemma justified-write-seen-hb-read-committed:
  assumes J: P ⊢ (E, ws) justified-by J
  and r: r ∈ read-actions (justifying-exec (J n)) r ∈ committed (J n)
  shows justifying-ws (J n) r ∈ committed (J n) (is ?thesis1)
  and ws (action-translation (J n) r) ∈ action-translation (J n) ‘ committed (J n) (is ?thesis2)
proof −
  have ?thesis1 ∧ ?thesis2 using r
  proof(induct n arbitrary: r)
    case 0
    from J have [simp]: committed (J 0) = {}
      by(simp add: is-commit-sequence-def)
    with 0 show ?case by simp
  next
    case (Suc n)
    let ?E = λn. justifying-exec (J n)
      and ?ws = λn. justifying-ws (J n)
      and ?C = λn. committed (J n)
      and ?φ = λn. action-translation (J n)

    note r = ‹r ∈ read-actions (?E (Suc n))›
    hence r ∈ actions (?E (Suc n)) by simp

    from J have wfan: wf-action-translation-on (?E n) E (?C n) (?φ n)
      and wfaSn: wf-action-translation-on (?E (Suc n)) E (?C (Suc n)) (?φ (Suc n))
      by(simp-all add: wf-action-translations-def)
```

```
  from wfaSn have injSn: inj-on (?φ (Suc n)) (actions (?E (Suc n)))
    by(rule wf-action-translation-on-inj-onD)
  from J have C-sub-A: ?C (Suc n) ⊆ actions (?E (Suc n))
    by(simp add: committed-subset-actions-def)
  from J have CnCSn: ?φ n ‘ ?C n ⊆ ?φ (Suc n) ‘ ?C (Suc n)
    by(simp add: is-commit-sequence-def)

  from J have wsSn: is-write-seen P (?E (Suc n)) (?ws (Suc n))
    by(simp add: justification-well-formed-def)
  from r obtain ad al v where action-obs (?E (Suc n)) r = NormalAction (ReadMem ad al v) by
cases
  from is-write-seenD[OF wsSn r this]
  have wsSn: ?ws (Suc n) r ∈ actions (?E (Suc n)) by simp

  show ?case
  proof(cases ?φ (Suc n) r ∈ ?φ n ‘ ?C n)
    case True
    then obtain r' where r': r' ∈ ?C n
      and r-r': ?φ (Suc n) r = ?φ n r' by(auto)
    from r' wfan have action-tid (?E n) r' = action-tid E (?φ n r')
      and action-obs (?E n) r' ≈ action-obs E (?φ n r')
      by(blast dest: wf-action-translation-on-actionD)+
    moreover from r' CnCSn have ?φ (Suc n) r ∈ ?φ (Suc n) ‘ ?C (Suc n)
      unfolding r-r' by auto
    hence r ∈ ?C (Suc n)
      unfolding inj-on-image-mem-iff[OF injSn ‹r ∈ actions (?E (Suc n))› C-sub-A] .
    with wfaSn have action-tid (?E (Suc n)) r = action-tid E (?φ (Suc n) r)
      and action-obs (?E (Suc n)) r ≈ action-obs E (?φ (Suc n) r)
      by(blast dest: wf-action-translation-on-actionD)+
    ultimately have tid: action-tid (?E n) r' = action-tid (?E (Suc n)) r
      and obs: action-obs (?E n) r' ≈ action-obs (?E (Suc n)) r
      unfolding r-r' by(auto intro: sim-action-trans sim-action-sym)

    from J have ?C n ⊆ actions (?E n) by(simp add: committed-subset-actions-def)
    with r' have r' ∈ actions (?E n) by blast
    with r obs have r' ∈ read-actions (?E n)
      by cases(auto intro: read-actions.intros)
    hence ?ws n r' ∈ ?C n ∧ ws (?φ n r') ∈ ?φ n ‘ ?C n using r' by(rule Suc)
    then obtain ws: ws (?φ n r') ∈ ?φ n ‘ ?C n ..

    have r-conv-inv: r = inv-into (actions (?E (Suc n))) (?φ (Suc n)) (?φ n r')
      using ‹r ∈ actions (?E (Suc n))› unfolding r-r'[symmetric]
      by(simp add: inv-into-f-f[OF injSn])
    with ‹r' ∈ ?C n› r J ‹r' ∈ read-actions (?E n)›
    have ws-eq: ?φ (Suc n) (?ws (Suc n) r) = ws (?φ n r')
      by(simp add: Let-def write-seen-committed-def)
    with ws CnCSn have ?φ (Suc n) (?ws (Suc n) r) ∈ ?φ (Suc n) ‘ ?C (Suc n) by auto
    hence ?ws (Suc n) r ∈ ?C (Suc n)
      by(subst (asm) inj-on-image-mem-iff[OF injSn wsSn C-sub-A])
    moreover from ws CnCSn have ws (?φ (Suc n) r) ∈ ?φ (Suc n) ‘ ?C (Suc n)
      unfolding r-r' by auto
    ultimately show ?thesis by simp
  next
    case False
```

```
      with r ‹r ∈ ?C (Suc n)› J
      have ?φ (Suc n) (?ws (Suc n) r) ∈ ?φ n ‘ ?C n
        and ws (?φ (Suc n) r) ∈ ?φ n ‘ ?C n
        unfolding is-justified-by.simps Let-def committed-reads-see-committed-writes-def
        by blast+
      hence ?φ (Suc n) (?ws (Suc n) r) ∈ ?φ (Suc n) ‘ ?C (Suc n)
        and ws (?φ (Suc n) r) ∈ ?φ (Suc n) ‘ ?C (Suc n)
        using CnCSn by blast+
      thus ?thesis by(simp add: inj-on-image-mem-iff[OF injSn wsSn C-sub-A])
    qed
  qed
  thus ?thesis1 ?thesis2 by simp-all
qed

lemma is-justified-by-imp-is-weakly-justified-by:
  assumes justified: P ⊢ (E, ws) justified-by J
  and wf: P ⊢ (E, ws) √
  shows P ⊢ (E, ws) weakly-justified-by J
  unfolding is-weakly-justified-by.simps
proof(intro conjI)
  let ?E = λn. justifying-exec (J n)
    and ?ws = λn. justifying-ws (J n)
    and ?C = λn. committed (J n)
    and ?φ = λn. action-translation (J n)

  from justified
  show is-commit-sequence E J justification-well-formed P J committed-subset-actions J
    value-written-committed P E J write-seen-committed ws J uncommitted-reads-see-hb P J
    wf-action-translations E J by(simp-all)

  show happens-before-committed-weak P E ws J
    unfolding happens-before-committed-weak-def Let-def
  proof(intro strip conjI)
    fix n r
    assume r ∈ read-actions (?E n) ∩ ?C n
    hence read: r ∈ read-actions (?E n) and committed: r ∈ ?C n by simp-all
    with justified have committed-ws: ?ws n r ∈ ?C n
      and committed-ws': ws (?φ n r) ∈ ?φ n ‘ ?C n
      by(rule justified-write-seen-hb-read-committed)+
    from committed-ws' obtain w where w: ws (?φ n r) = ?φ n w
      and committed-w: w ∈ ?C n by blast

   from committed-w justified have w ∈ actions (?E n) by(auto simp add: committed-subset-actions-def)
    moreover from justified have inj-on (?φ n) (actions (?E n))
      by(auto simp add: wf-action-translations-def dest: wf-action-translation-on-inj-onD)
    ultimately have w-def: w = inv-into (actions (?E n)) (?φ n) (ws (?φ n r))
      by(simp-all add: w)

    from committed committed-w
    have P,?E n ⊢ w ≤hb r ⟷ (happens-before P (?E n) |‘ ?C n) w r by auto
    also have ... ⟷ (inv-imageP (happens-before P E) (?φ n) |‘ ?C n) w r
      using justified by(simp add: happens-before-committed-def)
    also have ... ⟷ P,E ⊢ ?φ n w ≤hb ?φ n r using committed committed-w by auto
    finally show P,E ⊢ ws (?φ n r) ≤hb ?φ n r ⟷ P,?E n ⊢ inv-into (actions (?E n)) (?φ n) (ws
```

```
(?φ n r)) ≤hb r
      unfolding w[symmetric] unfolding w-def ..

    have P,?E n⊢ r ≤hb w ⟷ (happens-before P (?E n) |‘ ?C n) r w
      using committed committed-w by auto
    also have ... ⟷ (inv-imageP (happens-before P E) (?φ n) |‘ ?C n) r w
      using justified by(simp add: happens-before-committed-def)
    also have ... ⟷ P,E ⊢ ?φ n r ≤hb ws (?φ n r) using w committed committed-w by auto
    also {
      from read obtain ad al v where action-obs (?E n) r = NormalAction (ReadMem ad al v) by
cases auto
     with justified committed obtain v' where obs': action-obs E (?φ n r) = NormalAction (ReadMem
ad al v')
        by(fastforce simp add: wf-action-translations-def dest!: wf-action-translation-on-actionD)
      moreover from committed justified have ?φ n r ∈ actions E
        by(auto simp add: is-commit-sequence-def)
      ultimately have read': ?φ n r ∈ read-actions E by(auto intro: read-actions.intros)
      from wf have is-write-seen P E ws by(rule wf-exec-is-write-seenD)
      from is-write-seenD[OF this read' obs']
      have ¬ P,E ⊢ ?φ n r ≤hb ws (?φ n r) by simp }
    ultimately show ¬ P,?E n ⊢ r ≤hb inv-into (actions (?E n)) (?φ n) (ws (?φ n r))
      unfolding w-def by simp
  qed

  from justified have committed-reads-see-committed-writes ws J by simp
  thus committed-reads-see-committed-writes-weak ws J
   by(auto simp add: committed-reads-see-committed-writes-def committed-reads-see-committed-writes-weak-def)
qed

corollary legal-imp-weakly-legal-execution:
  legal-execution P 𝓔 Ews ⟹ weakly-legal-execution P 𝓔 Ews
by(cases Ews)(auto 4 4 simp add: gen-legal-execution.simps simp del: is-justified-by.simps is-weakly-justified-by.simps
intro: is-justified-by-imp-is-weakly-justified-by)

lemma drop-0th-justifying-exec:
  assumes P ⊢ (E, ws) justified-by J
  and wf: P ⊢ (E', ws') √
  shows P ⊢ (E, ws) justified-by (J(0 := ⦇committed = {}, justifying-exec = E', justifying-ws = ws',
action-translation = id⦈)))
  (is - ⊢ - justified-by ?J)
using assms
unfolding is-justified-by.simps is-commit-sequence-def
  justification-well-formed-def committed-subset-actions-def happens-before-committed-def
  sync-order-committed-def value-written-committed-def write-seen-committed-def uncommitted-reads-see-hb-def
  committed-reads-see-committed-writes-def external-actions-committed-def wf-action-translations-def
proof(intro conjI strip)
  let ?E = λn. justifying-exec (?J n)
    and ?φ = λn. action-translation (?J n)
    and ?C = λn. committed (?J n)
    and ?ws = λn. justifying-ws (?J n)

  show ?C 0 = {} by simp

  from assms have C-0: committed (J 0) = {} by(simp add: is-commit-sequence-def)
```

```
  hence (⋃n. ?φ n ‘ ?C n) = (⋃n. action-translation (J n) ‘ committed (J n))
    by −(rule SUP-cong, simp-all)
  also have ... = actions E using assms by(simp add: is-commit-sequence-def)
  finally show actions E = (⋃n. ?φ n ‘ ?C n) ..

  fix n
  { fix r'
    assume r' ∈ read-actions (?E (Suc n))
    thus ?φ (Suc n) r' ∈ ?φ n ‘ ?C n ∨ P,?E (Suc n) ⊢ ?ws (Suc n) r' ≤hb r'
     using assms by(auto dest!: bspec simp add: uncommitted-reads-see-hb-def is-commit-sequence-def)
}
  { fix r'
    assume r': r' ∈ read-actions (?E (Suc n)) ∩ ?C (Suc n)

    have n ≠ 0
    proof
      assume n = 0
      hence r' ∈ read-actions (justifying-exec (J 1)) ∩ committed (J 1) using r' by simp
      hence action-translation (J 1) r' ∈ action-translation (J 0) ‘ committed (J 0) ∨
            ws (action-translation (J 1) r') ∈ action-translation (J 0) ‘ committed (J 0) using assms
        unfolding One-nat-def is-justified-by.simps Let-def committed-reads-see-committed-writes-def
        by(metis (lifting))
      thus False unfolding C-0 by simp
    qed
    thus let r = ?φ (Suc n) r'; committed-n = ?φ n ‘ ?C n
       in r ∈ committed-n ∨
          (?φ (Suc n) (?ws (Suc n) r') ∈ committed-n ∧ ws r ∈ committed-n)
      using assms r' by(simp add: committed-reads-see-committed-writes-def) }
  { fix a a'
    assume a ∈ external-actions (?E n)
      and a' ∈ ?C n P,?E n ⊢ a ≤hb a'
    moreover hence n > 0 by(simp split: if-split-asm)
    ultimately show a ∈ ?C n using assms
      by(simp add: external-actions-committed-def) blast }

  from assms have wf-action-translation E (?J 0)
    by(simp add: wf-action-translations-def wf-action-translation-on-def)
  thus wf-action-translation E (?J n) using assms by(simp add: wf-action-translations-def)
qed auto

lemma drop-0th-weakly-justifying-exec:
  assumes P ⊢ (E, ws) weakly-justified-by J
  and wf: P ⊢ (E', ws') √
  shows P ⊢ (E, ws) weakly-justified-by (J(0 := ⦇committed = {}, justifying-exec = E', justifying-ws
= ws', action-translation = id⦈))
  (is - ⊢ - weakly-justified-by ?J)
using assms
unfolding is-weakly-justified-by.simps is-commit-sequence-def
  justification-well-formed-def committed-subset-actions-def happens-before-committed-weak-def
  value-written-committed-def write-seen-committed-def uncommitted-reads-see-hb-def
  committed-reads-see-committed-writes-weak-def external-actions-committed-def wf-action-translations-def
proof(intro conjI strip)
  let ?E = λn. justifying-exec (?J n)
    and ?φ = λn. action-translation (?J n)
```

```
    and ?C = λn. committed (?J n)
    and ?ws = λn. justifying-ws (?J n)

  show ?C 0 = {} by simp

  from assms have C-0: committed (J 0) = {} by(simp add: is-commit-sequence-def)
  hence (⋃ n. ?φ n ‘ ?C n) = (⋃ n. action-translation (J n) ‘ committed (J n))
    by −(rule SUP-cong, simp-all)
  also have ... = actions E using assms by(simp add: is-commit-sequence-def)
  finally show actions E = (⋃ n. ?φ n ‘ ?C n) ..

  fix n
  { fix r'
    assume r' ∈ read-actions (?E (Suc n))
    thus ?φ (Suc n) r' ∈ ?φ n ‘ ?C n ∨ P,?E (Suc n) ⊢ ?ws (Suc n) r' ≤hb r'
     using assms by(auto dest!: bspec simp add: uncommitted-reads-see-hb-def is-commit-sequence-def)
}
  { fix r'
    assume r': r' ∈ read-actions (?E (Suc n)) ∩ ?C (Suc n)

    have n ≠ 0
    proof
      assume n = 0
      hence r' ∈ read-actions (justifying-exec (J 1)) ∩ committed (J 1) using r' by simp
      hence action-translation (J 1) r' ∈ action-translation (J 0) ‘ committed (J 0) ∨
            ws (action-translation (J 1) r') ∈ action-translation (J 0) ‘ committed (J 0) using assms
      unfolding One-nat-def is-weakly-justified-by.simps Let-def committed-reads-see-committed-writes-weak-def
        by(metis (lifting))
      thus False unfolding C-0 by simp
    qed
    thus let r = ?φ (Suc n) r'; committed-n = ?φ n ‘ ?C n
      in r ∈ committed-n ∨ ws r ∈ committed-n
      using assms r' by(simp add: committed-reads-see-committed-writes-weak-def) }

  from assms have wf-action-translation E (?J 0)
    by(simp add: wf-action-translations-def wf-action-translation-on-def)
  thus wf-action-translation E (?J n) using assms by(simp add: wf-action-translations-def)
qed auto
```

8.4.13 Executions with common prefix

```
lemma actions-change-prefix:
  assumes read: a ∈ actions E
  and prefix: ltake n E [≈] ltake n E'
  and rn: enat a < n
  shows a ∈ actions E'
using llist-all2-llengthD[OF prefix[unfolded sim-actions-def]] read rn
by(simp add: actions-def min-def split: if-split-asm)

lemma action-obs-change-prefix:
  assumes prefix: ltake n E [≈] ltake n E'
  and rn: enat a < n
  shows action-obs E a ≈ action-obs E' a
proof −
```

from *rn* **have** *action-obs E a = action-obs (ltake n E) a*
by(*simp add*: *action-obs-def lnth-ltake*)
also from *prefix* **have** *... ≈ action-obs (ltake n E′) a*
by(*rule sim-actions-action-obsD*)
also have *... = action-obs E′ a* **using** *rn*
by(*simp add*: *action-obs-def lnth-ltake*)
finally show *?thesis* **.**
qed

lemma *action-obs-change-prefix-eq*:
assumes *prefix*: *ltake n E = ltake n E′*
and *rn*: *enat a < n*
shows *action-obs E a = action-obs E′ a*
proof −
from *rn* **have** *action-obs E a = action-obs (ltake n E) a*
by(*simp add*: *action-obs-def lnth-ltake*)
also from *prefix* **have** *... = action-obs (ltake n E′) a*
by(*simp add*: *action-obs-def*)
also have *... = action-obs E′ a* **using** *rn*
by(*simp add*: *action-obs-def lnth-ltake*)
finally show *?thesis* **.**
qed

lemma *read-actions-change-prefix*:
assumes *read*: *r ∈ read-actions E*
and *prefix*: *ltake n E [≈] ltake n E′ enat r < n*
shows *r ∈ read-actions E′*
using *read action-obs-change-prefix*[*OF prefix*] *actions-change-prefix*[*OF - prefix*]
by(*cases*)(*auto intro*: *read-actions.intros*)

lemma *sim-action-is-write-action-eq*:
assumes *obs ≈ obs′*
shows *is-write-action obs ⟷ is-write-action obs′*
using *assms* **by** *cases simp-all*

lemma *write-actions-change-prefix*:
assumes *write*: *w ∈ write-actions E*
and *prefix*: *ltake n E [≈] ltake n E′ enat w < n*
shows *w ∈ write-actions E′*
using *write action-obs-change-prefix*[*OF prefix*] *actions-change-prefix*[*OF - prefix*]
by(*cases*)(*auto intro*: *write-actions.intros dest*: *sim-action-is-write-action-eq*)

lemma *action-loc-change-prefix*:
assumes *ltake n E [≈] ltake n E′ enat a < n*
shows *action-loc P E a = action-loc P E′ a*
using *action-obs-change-prefix*[*OF assms*]
by(*fastforce elim*!: *action-loc-aux-cases intro*: *action-loc-aux-intros*)

lemma *sim-action-is-new-action-eq*:
assumes *obs ≈ obs′*
shows *is-new-action obs = is-new-action obs′*
using *assms* **by** *cases auto*

lemma *action-order-change-prefix*:

assumes *ao*: $E \vdash a \leq_a a'$
and *prefix*: *ltake n E* $[\approx]$ *ltake n E'*
and *an*: *enat a* $< n$
and *a'n*: *enat a'* $< n$
shows $E' \vdash a \leq_a a'$
using *ao actions-change-prefix*[*OF - prefix an*] *actions-change-prefix*[*OF - prefix a'n*] *action-obs-change-prefix*[*OF prefix an*] *action-obs-change-prefix*[*OF prefix a'n*]
by(*auto simp add*: *action-order-def split*: *if-split-asm dest*: *sim-action-is-new-action-eq*)

lemma *value-written-change-prefix*:
assumes *eq*: *ltake n E* = *ltake n E'*
and *an*: *enat a* $< n$
shows *value-written P E a* = *value-written P E' a*
using *action-obs-change-prefix-eq*[*OF eq an*]
by(*simp add*: *value-written-def fun-eq-iff*)

lemma *action-tid-change-prefix*:
assumes *prefix*: *ltake n E* $[\approx]$ *ltake n E'*
and *an*: *enat a* $< n$
shows *action-tid E a* = *action-tid E' a*
proof −
from *an* **have** *action-tid E a* = *action-tid* (*ltake n E*) *a*
by(*simp add*: *action-tid-def lnth-ltake*)
also from *prefix* **have** ... = *action-tid* (*ltake n E'*) *a*
by(*rule sim-actions-action-tidD*)
also from *an* **have** ... = *action-tid E' a*
by(*simp add*: *action-tid-def lnth-ltake*)
finally show *?thesis* **.**
qed

lemma *program-order-change-prefix*:
assumes *po*: $E \vdash a \leq_{po} a'$
and *prefix*: *ltake n E* $[\approx]$ *ltake n E'*
and *an*: *enat a* $< n$
and *a'n*: *enat a'* $< n$
shows $E' \vdash a \leq_{po} a'$
using *po action-order-change-prefix*[*OF - prefix an a'n*]
action-tid-change-prefix[*OF prefix an*] *action-tid-change-prefix*[*OF prefix a'n*]
by(*auto elim*!: *program-orderE intro*: *program-orderI*)

lemma *sim-action-sactionD*:
assumes *obs* $\approx$ *obs'*
shows *saction P obs* $\longleftrightarrow$ *saction P obs'*
using *assms* **by** *cases simp-all*

lemma *sactions-change-prefix*:
assumes *sync*: $a \in$ *sactions P E*
and *prefix*: *ltake n E* $[\approx]$ *ltake n E'*
and *rn*: *enat a* $< n$
shows $a \in$ *sactions P E'*
using *sync action-obs-change-prefix*[*OF prefix rn*] *actions-change-prefix*[*OF - prefix rn*]
unfolding *sactions-def* **by**(*simp add*: *sim-action-sactionD*)

lemma *sync-order-change-prefix*:
assumes *so*: $P,E \vdash a \leq_{so} a'$
and *prefix*: *ltake n E* [≈] *ltake n E'*
and *an*: *enat a* < *n*
and *a'n*: *enat a'* < *n*
shows $P,E' \vdash a \leq_{so} a'$
using *so action-order-change-prefix*[*OF - prefix an a'n*] *sactions-change-prefix*[*OF - prefix an*, *of P*] *sactions-change-prefix*[*OF - prefix a'n*, *of P*]
by(*simp add*: *sync-order-def*)

lemma *sim-action-synchronizes-withD*:
assumes *obs* ≈ *obs' obs''* ≈ *obs'''*
shows $P \vdash (t, obs) \rightsquigarrow_{sw} (t', obs'') \longleftrightarrow P \vdash (t, obs') \rightsquigarrow_{sw} (t', obs''')$
using *assms*
by(*auto elim*!: *sim-action.cases synchronizes-with.cases intro*: *synchronizes-with.intros*)

lemma *sync-with-change-prefix*:
assumes *sw*: $P,E \vdash a \leq_{sw} a'$
and *prefix*: *ltake n E* [≈] *ltake n E'*
and *an*: *enat a* < *n*
and *a'n*: *enat a'* < *n*
shows $P,E' \vdash a \leq_{sw} a'$
using *sw sync-order-change-prefix*[*OF - prefix an a'n*, *of P*]
action-tid-change-prefix[*OF prefix an*] *action-tid-change-prefix*[*OF prefix a'n*]
action-obs-change-prefix[*OF prefix an*] *action-obs-change-prefix*[*OF prefix a'n*]
by(*auto simp add*: *sync-with-def dest*: *sim-action-synchronizes-withD*)

lemma *po-sw-change-prefix*:
assumes *posw*: *po-sw P E a a'*
and *prefix*: *ltake n E* [≈] *ltake n E'*
and *an*: *enat a* < *n*
and *a'n*: *enat a'* < *n*
shows *po-sw P E' a a'*
using *posw sync-with-change-prefix*[*OF - prefix an a'n*, *of P*] *program-order-change-prefix*[*OF - prefix an a'n*]
by(*auto simp add*: *po-sw-def*)

lemma *happens-before-new-not-new*:
assumes *tsa-ok*: *thread-start-actions-ok E*
and *a*: *a* ∈ *actions E*
and *a'*: *a'* ∈ *actions E*
and *new-a*: *is-new-action* (*action-obs E a*)
and *new-a'*: ¬ *is-new-action* (*action-obs E a'*)
shows $P,E \vdash a \leq_{hb} a'$
proof −
from *thread-start-actions-okD*[*OF tsa-ok a' new-a'*]
obtain *i* **where** *i* ≤ *a'*
and *obs-i*: *action-obs E i* = *InitialThreadAction*
and *action-tid E i* = *action-tid E a'* **by** *auto*
from ‹*i* ≤ *a'*› *a'* **have** *i* ∈ *actions E*
by(*auto simp add*: *actions-def le-less-trans*[**where** *y*=*enat a'*])
with ‹*i* ≤ *a'*› *obs-i a' new-a'* **have** $E \vdash i \leq_a a'$ **by**(*simp add*: *action-order-def*)

```
  hence E ⊢ i ≤po a' using ‹action-tid E i = action-tid E a'›
    by(rule program-orderI)

  moreover {
    from ‹i ∈ actions E› obs-i
    have i ∈ sactions P E by(auto intro: sactionsI)
    from a ‹i ∈ actions E› new-a obs-i have E ⊢ a ≤a i by(simp add: action-order-def)
    moreover from a new-a have a ∈ sactions P E by(auto intro: sactionsI)
    ultimately have P,E ⊢ a ≤so i using ‹i ∈ sactions P E› by(rule sync-orderI)
     moreover from new-a obs-i have P ⊢ (action-tid E a, action-obs E a) ⇝sw (action-tid E i,
action-obs E i)
      by cases(auto intro: synchronizes-with.intros)
    ultimately have P,E ⊢ a ≤sw i by(rule sync-withI) }
  ultimately show ?thesis unfolding po-sw-def [abs-def] by(blast intro: tranclp.r-into-trancl tran-
clp-trans)
qed

lemma happens-before-change-prefix:
  assumes hb: P,E ⊢ a ≤hb a'
  and tsa-ok: thread-start-actions-ok E'
  and prefix: ltake n E [≈] ltake n E'
  and an: enat a < n
  and a'n: enat a' < n
  shows P,E' ⊢ a ≤hb a'
using hb an a'n
proof induct
  case (base a')
  thus ?case by(rule tranclp.r-into-trancl[where r=po-sw P E', OF po-sw-change-prefix[OF - prefix]])
next
  case (step a' a'')
  show ?case
  proof(cases is-new-action (action-obs E a') ∧ ¬ is-new-action (action-obs E a''))
    case False
    from ‹po-sw P E a' a''› have E ⊢ a' ≤a a'' by(rule po-sw-into-action-order)
    with ‹enat a'' < n› False have enat a' < n
      by(safe elim!: action-orderE)(metis Suc-leI Suc-n-not-le-n enat-ord-simps(2) le-trans nat-neq-iff
xtrans(10))+
    with ‹enat a < n› have P,E' ⊢ a ≤hb a' by(rule step)
    moreover from ‹po-sw P E a' a''› prefix ‹enat a' < n› ‹enat a'' < n›
    have po-sw P E' a' a'' by(rule po-sw-change-prefix)
    ultimately show ?thesis ..
  next
    case True
    then obtain new-a': is-new-action (action-obs E a')
      and ¬ is-new-action (action-obs E a'') ..
    from ‹P,E ⊢ a ≤hb a'› new-a'
    have new-a: is-new-action (action-obs E a)
      and tid: action-tid E a = action-tid E a'
      and a ≤ a' by(rule happens-before-new-actionD)+

    note tsa-ok moreover
    from porder-happens-before[of E P] have a ∈ actions E
      by(rule porder-onE)(erule refl-onPD1, rule ‹P,E ⊢ a ≤hb a'›)
    hence a ∈ actions E' using an by(rule actions-change-prefix[OF - prefix])
```

```
    moreover
    from ‹po-sw P E a' a''› refl-on-program-order[of E] refl-on-sync-order[of P E]
    have a'' ∈ actions E
      unfolding po-sw-def by(auto dest: refl-onPD2 elim!: sync-withE)
    hence a'' ∈ actions E' using ‹enat a'' < n› by(rule actions-change-prefix[OF - prefix])
    moreover
    from new-a action-obs-change-prefix[OF prefix an]
    have is-new-action (action-obs E' a) by(cases) auto
    moreover
    from ‹¬ is-new-action (action-obs E a'')› action-obs-change-prefix[OF prefix ‹enat a'' < n›]
    have ¬ is-new-action (action-obs E' a'') by(auto elim: is-new-action.cases)
    ultimately show P,E' ⊢ a ≤hb a'' by(rule happens-before-new-not-new)
  qed
qed

lemma thread-start-actions-ok-change:
  assumes tsa: thread-start-actions-ok E
  and sim: E [≈] E'
  shows thread-start-actions-ok E'
proof(rule thread-start-actions-okI)
  fix a
  assume a ∈ actions E' ¬ is-new-action (action-obs E' a)
  from sim have len-eq: llength E = llength E' by(simp add: sim-actions-def)(rule llist-all2-llengthD)
  with sim have sim': ltake (llength E) E [≈] ltake (llength E) E' by(simp add: ltake-all)

  from ‹a ∈ actions E'› len-eq have enat a < llength E by(simp add: actions-def)
  with ‹a ∈ actions E'› sim'[symmetric] have a ∈ actions E by(rule actions-change-prefix)
  moreover have ¬ is-new-action (action-obs E a)
    using action-obs-change-prefix[OF sim' ‹enat a < llength E›] ‹¬ is-new-action (action-obs E' a)›
    by(auto elim!: is-new-action.cases)
  ultimately obtain i where i ≤ a action-obs E i = InitialThreadAction action-tid E i = action-tid
E a
    by(blast dest: thread-start-actions-okD[OF tsa])
  thus ∃ i ≤ a. action-obs E' i = InitialThreadAction ∧ action-tid E' i = action-tid E' a
     using action-tid-change-prefix[OF sim', of i] action-tid-change-prefix[OF sim', of a] ‹enat a <
llength E›
      action-obs-change-prefix[OF sim', of i]
    by(cases llength E)(auto intro!: exI[where x=i])
qed

context executions-aux begin

lemma E-new-same-addr-singleton:
  assumes E: E ∈ E
  shows ∃ a. new-actions-for P E adal ⊆ {a}
by(blast dest: E-new-actions-for-fun[OF E])

lemma new-action-before-read:
  assumes E: E ∈ E
  and wf: P ⊢ (E, ws) √
  and ra: ra ∈ read-actions E
  and adal: adal ∈ action-loc P E ra
  and new: wa ∈ new-actions-for P E adal
  and sc: ⋀a. ⟦ a < ra; a ∈ read-actions E ⟧ ⟹ P,E ⊢ a ↝mrw ws a
```

shows *wa* < *ra*
using *E-new-same-addr-singleton*[*OF E*, *of adal*] *init-before-read*[*OF E wf ra adal sc*] *new*
by *auto*

lemma *mrw-before*:
assumes *E*: $E \in \mathcal{E}$
and *wf*: $P \vdash (E, ws)\ \surd$
and *mrw*: $P,E \vdash r \rightsquigarrow mrw\ w$
and *sc*: $\bigwedge a.\ [\![\ a < r;\ a \in \textit{read-actions}\ E\ ]\!] \Longrightarrow P,E \vdash a \rightsquigarrow mrw\ ws\ a$
shows *w* < *r*
using *mrw read-actions-not-write-actions*[*of r E*]
apply *cases*
apply(*erule action-orderE*)
apply(*erule* (*1*) *new-action-before-read*[*OF E wf*])
apply(*simp add*: *new-actions-for-def*)
apply(*erule* (*1*) *sc*)
apply(*cases w* = *r*)
apply *auto*
done

lemma *mrw-change-prefix*:
assumes *E'*: $E' \in \mathcal{E}$
and *mrw*: $P,E \vdash r \rightsquigarrow mrw\ w$
and *tsa-ok*: *thread-start-actions-ok E'*
and *prefix*: *ltake n E* $[\approx]$ *ltake n E'*
and *an*: *enat r* < *n*
and *a'n*: *enat w* < *n*
shows $P,E' \vdash r \rightsquigarrow mrw\ w$
using *mrw*
proof *cases*
fix *adal*
assume *r*: $r \in \textit{read-actions}\ E$
and *adal-r*: $adal \in \textit{action-loc}\ P\ E\ r$
and *war*: $E \vdash w \leq a\ r$
and *w*: $w \in \textit{write-actions}\ E$
and *adal-w*: $adal \in \textit{action-loc}\ P\ E\ w$
and *mrw*: $\bigwedge wa'.\ [\![wa' \in \textit{write-actions}\ E;\ adal \in \textit{action-loc}\ P\ E\ wa']\!]$
$\Longrightarrow E \vdash wa' \leq a\ w \vee E \vdash r \leq a\ wa'$
show *?thesis*
proof(*rule most-recent-write-for.intros*)
from *r prefix an* **show** *r'*: $r \in \textit{read-actions}\ E'$
by(*rule read-actions-change-prefix*)
from *adal-r* **show** $adal \in \textit{action-loc}\ P\ E'\ r$
by(*simp add*: *action-loc-change-prefix*[*OF prefix*[*symmetric*] *an*])
from *war prefix a'n an* **show** $E' \vdash w \leq a\ r$ **by**(*rule action-order-change-prefix*)
from *w prefix a'n* **show** *w'*: $w \in \textit{write-actions}\ E'$ **by**(*rule write-actions-change-prefix*)
from *adal-w* **show** *adal-w'*: $adal \in \textit{action-loc}\ P\ E'\ w$ **by**(*simp add*: *action-loc-change-prefix*[*OF prefix*[*symmetric*] *a'n*])

fix *wa'*
assume *wa'*: $wa' \in \textit{write-actions}\ E'$
and *adal-wa'*: $adal \in \textit{action-loc}\ P\ E'\ wa'$
show $E' \vdash wa' \leq a\ w \vee E' \vdash r \leq a\ wa'$
proof(*cases enat wa'* < *n*)

```
      case True
      note wa'n = this
      with wa' prefix[symmetric] have wa' ∈ write-actions E by(rule write-actions-change-prefix)
      moreover from adal-wa' have adal ∈ action-loc P E wa'
        by(simp add: action-loc-change-prefix[OF prefix wa'n])
      ultimately have E ⊢ wa' ≤a w ∨ E ⊢ r ≤a wa' by(rule mrw)
      thus ?thesis
      proof
        assume E ⊢ wa' ≤a w
        hence E' ⊢ wa' ≤a w using prefix wa'n a'n by(rule action-order-change-prefix)
        thus ?thesis ..
      next
        assume E ⊢ r ≤a wa'
        hence E' ⊢ r ≤a wa' using prefix an wa'n by(rule action-order-change-prefix)
        thus ?thesis ..
      qed
    next
      case False note wa'n = this
      show ?thesis
      proof(cases is-new-action (action-obs E' wa'))
        case False
        hence E' ⊢ r ≤a wa' using wa'n r' wa' an
          by(auto intro!: action-orderI) (metis enat-ord-code(1) linorder-le-cases order-le-less-trans)
        thus ?thesis ..
      next
        case True
      with wa' adal-wa' have new: wa' ∈ new-actions-for P E' adal by(simp add: new-actions-for-def)
        show ?thesis
        proof(cases is-new-action (action-obs E' w))
          case True
          with adal-w' a'n w' have w ∈ new-actions-for P E' adal by(simp add: new-actions-for-def)
          with E' new have wa' = w by(rule E-new-actions-for-fun)
          thus ?thesis using w' by(auto intro: refl-onPD[OF refl-action-order])
        next
          case False
          with True wa' w' show ?thesis by(auto intro!: action-orderI)
        qed
      qed
    qed
  qed
qed

lemma action-order-read-before-write:
  assumes E: E ∈ E P ⊢ (E, ws) √
  and ao: E ⊢ w ≤a r
  and r: r ∈ read-actions E
  and w: w ∈ write-actions E
  and adal: adal ∈ action-loc P E r adal ∈ action-loc P E w
  and sc: ⋀a. ⟦ a < r; a ∈ read-actions E ⟧ ⟹ P,E ⊢ a ↝mrw ws a
  shows w < r
using ao
proof(cases rule: action-orderE)
  case 1
  from init-before-read[OF E r adal(1) sc]
```

```
  obtain i where i < r i ∈ new-actions-for P E adal by blast
  moreover from ‹is-new-action (action-obs E w)› adal(2) ‹w ∈ actions E›
  have w ∈ new-actions-for P E adal by(simp add: new-actions-for-def)
  ultimately show w < r using E by(auto dest: E-new-actions-for-fun)
next
  case 2
  with r w show ?thesis
    by(cases w = r)(auto dest: read-actions-not-write-actions)
qed

end

end
```

8.5 The data race free guarantee of the JMM

```
theory JMM-DRF
imports
  JMM-Spec
begin

context drf begin

lemma drf-lemma:
  assumes wf: P ⊢ (E, ws) √
  and E: E ∈ E
  and sync: correctly-synchronized P E
  and read-before: ⋀r. r ∈ read-actions E ⟹ P,E ⊢ ws r ≤hb r
  shows sequentially-consistent P (E, ws)
proof(rule ccontr)
  let ?Q = {r. r ∈ read-actions E ∧ ¬ P,E ⊢ r ↝mrw ws r}

  assume ¬ ?thesis
  then obtain r where r ∈ read-actions E ¬ P,E ⊢ r ↝mrw ws r
    by(auto simp add: sequentially-consistent-def)
  hence r ∈ ?Q by simp
  with wf-action-order[of E] obtain r'
    where r' ∈ ?Q
    and (action-order E)^≠≠^** r' r
    and r'-min: ⋀a. (action-order E)^≠≠ a r' ⟹ a ∉ ?Q
    by(rule wfP-minimalE) blast
  from ‹r' ∈ ?Q› have r': r' ∈ read-actions E
    and not-mrw: ¬ P,E ⊢ r' ↝mrw ws r' by blast+

  from r' obtain ad al v where obs-r': action-obs E r' = NormalAction (ReadMem ad al v)
    by(cases) auto
  from wf have ws: is-write-seen P E ws
    and tsa-ok: thread-start-actions-ok E
    by(rule wf-exec-is-write-seenD wf-exec-thread-start-actions-okD)+
  from is-write-seenD[OF ws r' obs-r']
  have ws-r: ws r' ∈ write-actions E
    and adal: (ad, al) ∈ action-loc P E (ws r')
    and v: v = value-written P E (ws r') (ad, al)
```

```
    and not-hb: ¬ P,E ⊢ r' ≤hb ws r' by auto
  from r' have P,E ⊢ ws r' ≤hb r' by(rule read-before)
  hence E ⊢ ws r' ≤a r' by(rule happens-before-into-action-order)
  from not-mrw
  have ∃ w'. w' ∈ write-actions E ∧ (ad, al) ∈ action-loc P E w' ∧
      ¬ P,E ⊢ w' ≤hb ws r' ∧ ¬ P,E ⊢ w' ≤so ws r' ∧
      ¬ P,E ⊢ r' ≤hb w' ∧ ¬ P,E ⊢ r' ≤so w' ∧ E ⊢ w' ≤a r'
  proof(rule contrapos-np)
    assume inbetween: ¬ ?thesis
    note r'
    moreover from obs-r' have (ad, al) ∈ action-loc P E r' by simp
    moreover note ‹E ⊢ ws r' ≤a r'› ws-r adal
    moreover
    { fix w'
      assume w' ∈ write-actions E (ad, al) ∈ action-loc P E w'
     with inbetween have P,E ⊢ w' ≤hb ws r' ∨ P,E ⊢ w' ≤so ws r' ∨ P,E ⊢ r' ≤hb w' ∨ P,E ⊢ r'
≤so w' ∨ ¬ E ⊢ w' ≤a r' by simp
      moreover from total-onPD[OF total-action-order, of w' E r'] ‹w' ∈ write-actions E› r'
      have E ⊢ w' ≤a r' ∨ E ⊢ r' ≤a w' by(auto dest: read-actions-not-write-actions)
      ultimately have E ⊢ w' ≤a ws r' ∨ E ⊢ r' ≤a w' unfolding sync-order-def
        by(blast intro: happens-before-into-action-order) }
    ultimately show P,E ⊢ r' ⇝mrw ws r' by(rule most-recent-write-for.intros)
  qed
  then obtain w' where w': w' ∈ write-actions E
    and adal-w': (ad, al) ∈ action-loc P E w'
    and ¬ P,E ⊢ w' ≤hb ws r' ¬ P,E ⊢ r' ≤hb w' E ⊢ w' ≤a r'
    and so: ¬ P,E ⊢ w' ≤so ws r' ¬ P,E ⊢ r' ≤so w' by blast

  have ws r' ≠ w' using ‹¬ P,E ⊢ w' ≤hb ws r'› ws-r
    by(auto intro: happens-before-refl)

  have vol: ¬ is-volatile P al
  proof
    assume vol-al: is-volatile P al
    with r' obs-r' have r' ∈ sactions P E by cases(rule sactionsI, simp-all)
    moreover from w' vol-al adal-w' have w' ∈ sactions P E
      by(cases)(auto intro: sactionsI elim!: is-write-action.cases)
    ultimately have P,E ⊢ w' ≤so r' ∨ w' = r' ∨ P,E ⊢ r' ≤so w'
      using total-sync-order[of P E] by(blast dest: total-onPD)
    moreover have w' ≠ r' using w' r' by(auto dest: read-actions-not-write-actions)
    ultimately have P,E ⊢ w' ≤so r' using ‹¬ P,E ⊢ r' ≤so w'› by simp
    moreover from ws-r vol-al adal have ws r' ∈ sactions P E
      by(cases)(auto intro: sactionsI elim!: is-write-action.cases)
    with total-sync-order[of P E] ‹w' ∈ sactions P E› ‹¬ P,E ⊢ w' ≤so ws r'› ‹ws r' ≠ w'›
    have P,E ⊢ ws r' ≤so w' by(blast dest: total-onPD)
    ultimately show False
      using is-write-seenD[OF ws r' obs-r'] w' adal-w' vol-al ‹ws r' ≠ w'› by auto
  qed

  { fix a
    assume a < r' and a ∈ read-actions E
    hence (action-order E)≠≠ a r' using r' obs-r' by(auto intro: action-orderI)
    from r'-min[OF this] ‹a ∈ read-actions E›
    have P,E ⊢ a ⇝mrw ws a by simp }
```

from $\mathcal{E}$-*sequential-completion*[*OF E wf this*, *of r'*] *r'*
obtain *E' ws'* **where** $E' \in \mathcal{E}\ P \vdash (E', ws')\ \surd$
and *eq*: *ltake* (*enat r'*) *E* = *ltake* (*enat r'*) *E'*
and *sc'*: *sequentially-consistent P* (*E'*, *ws'*)
and *r''*: *action-tid E r'* = *action-tid E' r' action-obs E r'* ≈ *action-obs E' r'*
and *r'* ∈ *actions E'*
by *auto*

from ‹*P* ⊢ (*E'*, *ws'*) $\surd$› **have** *tsa-ok'*: *thread-start-actions-ok E'*
by(*rule wf-exec-thread-start-actions-okD*)

from ‹*r'* ∈ *read-actions E*› **have** *enat r'* < *llength E* **by**(*auto elim*: *read-actions.cases actionsE*)
moreover from ‹*r'* ∈ *actions E'*› **have** *enat r'* < *llength E'* **by**(*auto elim*: *actionsE*)
ultimately have *eq'*: *ltake* (*enat* (*Suc r'*)) *E* [≈] *ltake* (*enat* (*Suc r'*)) *E'*
using *eq*[*THEN eq-into-sim-actions*] *r''*
by(*auto simp add*: *ltake-Suc-conv-snoc-lnth sim-actions-def split-beta action-tid-def action-obs-def intro*!: *llist-all2-lappendI*)
from *r'* **have** *r''*: *r'* ∈ *read-actions E'*
by(*rule read-actions-change-prefix*[*OF -eq'*]) *simp*
from *obs-r'* **have** (*ad*, *al*) ∈ *action-loc P E r'* **by** *simp*
hence *adal-r''*: (*ad*, *al*) ∈ *action-loc P E' r'*
by(*subst* (*asm*) *action-loc-change-prefix*[*OF eq'*]) *simp*

from ‹¬ *P*,*E* ⊢ *w'* ≤*hb ws r'*›
have ¬ *is-new-action* (*action-obs E w'*)
proof(*rule contrapos-nn*)
assume *new-w'*: *is-new-action* (*action-obs E w'*)
show *P*,*E* ⊢ *w'* ≤*hb ws r'*
proof(*cases is-new-action* (*action-obs E* (*ws r'*)))
case *True*
with *adal new-w' adal-w' w' ws-r*
have *ws r'* ∈ *new-actions-for P E* (*ad*, *al*) *w'* ∈ *new-actions-for P E* (*ad*, *al*)
by(*auto simp add*: *new-actions-for-def*)
with ‹*E* ∈ $\mathcal{E}$› **have** *ws r'* = *w'* **by**(*rule* $\mathcal{E}$*-new-actions-for-fun*)
thus *?thesis* **using** *w'* **by**(*auto intro*: *happens-before-refl*)
next
case *False*
with *tsa-ok w' ws-r new-w'*
show *?thesis* **by**(*auto intro*: *happens-before-new-not-new*)
qed
qed
with ‹*E* ⊢ *w'* ≤*a r'*› **have** *w'* ≤ *r'* **by**(*auto elim*!: *action-orderE*)
moreover from *w' r'* **have** *w'* ≠ *r'* **by**(*auto intro*: *read-actions-not-write-actions*)
ultimately have *w'* < *r'* **by** *simp*
with *w'* **have** *w'* ∈ *write-actions E'*
by(*auto intro*: *write-actions-change-prefix*[*OF - eq'*])
hence *w'* ∈ *actions E'* **by** *simp*

from *adal-w'* ‹*w'* < *r'*›
have (*ad*, *al*) ∈ *action-loc P E' w'*
by(*subst action-loc-change-prefix*[*symmetric*, *OF eq'*]) *simp-all*

from *vol* ‹*r'* ∈ *read-actions E'*› ‹*w'* ∈ *write-actions E'*› ‹(*ad*, *al*) ∈ *action-loc P E' w'*› *adal-r''*

```
  have P,E' ⊢ r' † w' unfolding non-volatile-conflict-def by auto
  with sync ‹E' ∈ ℰ› ‹P ⊢ (E', ws') √› sc' ‹r' ∈ actions E'› ‹w' ∈ actions E'›
  have hb'-r'-w': P,E' ⊢ r' ≤hb w' ∨ P,E' ⊢ w' ≤hb r'
    by(rule correctly-synchronizedD[rule-format])
  hence P,E ⊢ r' ≤hb w' ∨ P,E ⊢ w' ≤hb r' using ‹w' < r'›
    by(auto intro: happens-before-change-prefix[OF - tsa-ok eq'[symmetric]])
  with ‹¬ P,E ⊢ r' ≤hb w'› have P,E ⊢ w' ≤hb r' by simp

  have P,E ⊢ ws r' ≤hb w'
  proof(cases is-new-action (action-obs E (ws r')))
    case False
    with ‹E ⊢ ws r' ≤a r'› have ws r' ≤ r' by(auto elim!: action-orderE)
    moreover from ws-r r' have ws r' ≠ r' by(auto dest: read-actions-not-write-actions)
    ultimately have ws r' < r' by simp
    with ws-r have ws r' ∈ write-actions E'
      by(auto intro: write-actions-change-prefix[OF - eq'])
    hence ws r' ∈ actions E' by simp

    from adal ‹ws r' < r'›
    have (ad, al) ∈ action-loc P E' (ws r')
      by(subst action-loc-change-prefix[symmetric, OF eq']) simp-all
    hence P,E' ⊢ ws r' † w'
      using ‹ws r' ∈ write-actions E'› ‹w' ∈ write-actions E'› ‹(ad, al) ∈ action-loc P E' w'› vol
      unfolding non-volatile-conflict-def by auto
    with sync ‹E' ∈ ℰ› ‹P ⊢ (E', ws') √› sc' ‹ws r' ∈ actions E'› ‹w' ∈ actions E'›
    have P,E' ⊢ ws r' ≤hb w' ∨ P,E' ⊢ w' ≤hb ws r'
      by(rule correctly-synchronizedD[rule-format])
    thus P,E ⊢ ws r' ≤hb w' using ‹w' < r'› ‹ws r' < r'› ‹¬ P,E ⊢ w' ≤hb ws r'›
      by(auto dest: happens-before-change-prefix[OF - tsa-ok eq'[symmetric]])
  next
    case True
    with tsa-ok ws-r w' ‹¬ is-new-action (action-obs E w')›
    show P,E ⊢ ws r' ≤hb w' by(auto intro: happens-before-new-not-new)
  qed
  moreover
  from wf have is-write-seen P E ws by(rule wf-exec-is-write-seenD)
  ultimately have w' = ws r'
    using is-write-seenD[OF ‹is-write-seen P E ws› ‹r' ∈ read-actions E› obs-r']
      ‹w' ∈ write-actions E› ‹(ad, al) ∈ action-loc P E w'› ‹P,E ⊢ w' ≤hb r'›
    by auto
  with porder-happens-before[of E P] ‹¬ P,E ⊢ w' ≤hb ws r'› ws-r show False
    by(auto dest: refl-onPD[where a=ws r'] elim!: porder-onE)
qed

lemma justified-action-committedD:
  assumes justified: P ⊢ (E, ws) weakly-justified-by J
  and a: a ∈ actions E
  obtains n a' where a = action-translation (J n) a' a' ∈ committed (J n)
proof(atomize-elim)
  from justified have actions E = (⋃n. action-translation (J n) ` committed (J n))
    by(simp add: is-commit-sequence-def)
  with a show ∃ n a'. a = action-translation (J n) a' ∧ a' ∈ committed (J n) by auto
qed
```

theorem *drf-weak*:
assumes *sync*: *correctly-synchronized* P $\mathcal{E}$
and *legal*: *weakly-legal-execution* P $\mathcal{E}$ (E, ws)
shows *sequentially-consistent* P (E, ws)
using *legal-wf-execD*[*OF legal*] *legal-*$\mathcal{E}$*D*[*OF legal*] *sync*
proof(*rule drf-lemma*)
fix r
assume $r \in$ *read-actions* E

from *legal* **obtain** J **where** E: $E \in \mathcal{E}$
and *wf-exec*: $P \vdash (E, ws) \surd$
and J: $P \vdash (E, ws)$ *weakly-justified-by* J
and *range-J*: *range* (*justifying-exec* $\circ$ J) $\subseteq \mathcal{E}$
by(*rule legal-executionE*)

let *?E* $= \lambda n.$ *justifying-exec* $(J\ n)$
and *?ws* $= \lambda n.$ *justifying-ws* $(J\ n)$
and *?C* $= \lambda n.$ *committed* $(J\ n)$
and *?*φ $= \lambda n.$ *action-translation* $(J\ n)$

from ‹$r \in$ *read-actions* E› **have** $r \in$ *actions* E **by** *simp*
with J **obtain** $n\ r'$ **where** r: $r =$ *action-translation* $(J\ n)\ r'$
and r': $r' \in$ *?C* n **by**(*rule justified-action-committedD*)

note ‹$r \in$ *read-actions* E›
moreover from J **have** *wfan*: *wf-action-translation-on* (*?E* n) E (*?C* n) (*?*φ n)
by(*simp add*: *wf-action-translations-def*)
hence *action-obs* (*?E* n) $r' \approx$ *action-obs* $E\ r$ **using** r' **unfolding** r
by(*blast dest*: *wf-action-translation-on-actionD*)
moreover from $J\ r'$ **have** $r' \in$ *actions* (*?E* n)
by(*auto simp add*: *committed-subset-actions-def*)
ultimately have $r' \in$ *read-actions* (*?E* n) **unfolding** r
by *cases*(*auto intro*: *read-actions.intros*)
hence $P,E \vdash ws$ (*?*φ $n\ r'$) $\leq hb$ *?*φ $n\ r'$ **using** ‹$r' \in$ *?C* n›
proof(*induct n arbitrary*: r')
case *0*
from J **have** *?C 0* $= \{\}$ **by**(*simp add*: *is-commit-sequence-def*)
with *0* **have** *False* **by** *simp*
thus *?case* **..**
next
case (*Suc n r*)
note $r =$ ‹$r \in$ *read-actions* (*?E* (*Suc n*))›
from J **have** *wfan*: *wf-action-translation-on* (*?E* n) E (*?C* n) (*?*φ n)
and *wfaSn*: *wf-action-translation-on* (*?E* (*Suc n*)) E (*?C* (*Suc n*)) (*?*φ (*Suc n*))
by(*simp-all add*: *wf-action-translations-def*)

from *wfaSn* **have** *injSn*: *inj-on* (*?*φ (*Suc n*)) (*actions* (*?E* (*Suc n*)))
by(*rule wf-action-translation-on-inj-onD*)
from J **have** *C-sub-A*: *?C* (*Suc n*) $\subseteq$ *actions* (*?E* (*Suc n*))
by(*simp add*: *committed-subset-actions-def*)

from J **have** *wf*: $P \vdash$ (*?E* (*Suc n*), *?ws* (*Suc n*)) $\surd$ **by**(*simp add*: *justification-well-formed-def*)
moreover from *range-J* **have** *?E* (*Suc n*) $\in \mathcal{E}$ **by** *auto*
ultimately have *sc*: *sequentially-consistent* P (*?E* (*Suc n*), *?ws* (*Suc n*)) **using** *sync*

proof(*rule drf-lemma*)
fix r'
assume r': $r' \in$ *read-actions* (*?E* (*Suc n*))
hence $r' \in$ *actions* (*?E* (*Suc n*)) **by** *simp*

show *P*,*?E* (*Suc n*) ⊢ *?ws* (*Suc n*) $r' \leq_{hb} r'$
proof(*cases ?φ* (*Suc n*) $r' \in$ *?φ n* ‘ *?C n*)
case *True*
then obtain r'' **where** r'': $r'' \in$ *?C n*
and r'-r'': *?φ* (*Suc n*) r' = *?φ n* r'' **by**(*auto*)
from r'' *wfan* **have** *action-tid* (*?E n*) r'' = *action-tid E* (*?φ n* r'')
and *action-obs* (*?E n*) $r'' \approx$ *action-obs E* (*?φ n* r'')
by(*blast dest*: *wf-action-translation-on-actionD*)+
moreover from *J* **have** *?φ n* ‘ *?C n* ⊆ *?φ* (*Suc n*) ‘ *?C* (*Suc n*)
by(*simp add*: *is-commit-sequence-def*)
with r'' **have** *?φ* (*Suc n*) $r' \in$ *?φ* (*Suc n*) ‘ *?C* (*Suc n*)
unfolding r'-r'' **by** *auto*
hence $r' \in$ *?C* (*Suc n*)
unfolding *inj-on-image-mem-iff*[*OF injSn* ‹$r' \in$ *actions* (*?E* (*Suc n*))› *C-sub-A*] .
with *wfaSn* **have** *action-tid* (*?E* (*Suc n*)) r' = *action-tid E* (*?φ* (*Suc n*) r')
and *action-obs* (*?E* (*Suc n*)) $r' \approx$ *action-obs E* (*?φ* (*Suc n*) r')
by(*blast dest*: *wf-action-translation-on-actionD*)+
ultimately have *tid*: *action-tid* (*?E n*) r'' = *action-tid* (*?E* (*Suc n*)) r'
and *obs*: *action-obs* (*?E n*) $r'' \approx$ *action-obs* (*?E* (*Suc n*)) r'
unfolding r'-r'' **by**(*auto intro*: *sim-action-trans sim-action-sym*)

from *J* **have** *?C n* ⊆ *actions* (*?E n*) **by**(*simp add*: *committed-subset-actions-def*)
with r'' **have** $r'' \in$ *actions* (*?E n*) **by** *blast*
with r' *obs* **have** $r'' \in$ *read-actions* (*?E n*)
by *cases*(*auto intro*: *read-actions.intros*)
hence *hb''*: *P*,*E* ⊢ *ws* (*?φ n* r'') ≤*hb* *?φ n* r''
using ‹$r'' \in$ *?C n*› **by**(*rule Suc*)

have *r-conv-inv*: r' = *inv-into* (*actions* (*?E* (*Suc n*))) (*?φ* (*Suc n*)) (*?φ n* r'')
using ‹$r' \in$ *actions* (*?E* (*Suc n*))› **unfolding** r'-r''[*symmetric*]
by(*simp add*: *inv-into-f-f*[*OF injSn*])
with ‹$r'' \in$ *?C n*› r' *J* ‹$r'' \in$ *read-actions* (*?E n*)›
have *ws-eq*[*symmetric*]: *?φ* (*Suc n*) (*?ws* (*Suc n*) r') = *ws* (*?φ n* r'')
by(*simp add*: *write-seen-committed-def*)
with r'-r''[*symmetric*] *hb''* **have** *P*,*E* ⊢ *?φ* (*Suc n*) (*?ws* (*Suc n*) r') ≤*hb* *?φ* (*Suc n*) r' **by** *simp*

moreover

from *J* r' ‹$r' \in$ *committed* (*J* (*Suc n*))›
have *ws* (*?φ* (*Suc n*) r') ∈ *?φ* (*Suc n*) ‘ *?C* (*Suc n*)
by(*rule weakly-justified-write-seen-hb-read-committed*)
then obtain w' **where** w': *ws* (*?φ* (*Suc n*) r') = *?φ* (*Suc n*) w'
and *committed-w'*: $w' \in$ *?C* (*Suc n*) **by** *blast*
with *C-sub-A* **have** *w'-action*: $w' \in$ *actions* (*?E* (*Suc n*)) **by** *auto*

hence *w'-def*: w' = *inv-into* (*actions* (*?E* (*Suc n*))) (*?φ* (*Suc n*)) (*ws* (*?φ* (*Suc n*) r'))
using *injSn* **unfolding** w' **by** *simp*

```
      from J r' ‹r' ∈ committed (J (Suc n))›
      have hb-eq: P,E ⊢ ws (?φ (Suc n) r') ≤hb ?φ (Suc n) r' ⟷ P,?E (Suc n) ⊢ w' ≤hb r'
        unfolding w'-def by(simp add: happens-before-committed-weak-def)

      from r' obtain ad al v where action-obs (?E (Suc n)) r' = NormalAction (ReadMem ad al
v) by(cases)
      from is-write-seenD[OF wf-exec-is-write-seenD[OF wf] r' this]
      have ?ws (Suc n) r' ∈ actions (?E (Suc n)) by(auto)
      with injSn have w' = ?ws (Suc n) r'
        unfolding w'-def ws-eq[folded r'-r''] by(rule inv-into-f-f)
      thus ?thesis using hb'' hb-eq w'-action r'-r''[symmetric] w' injSn by simp
    next
      case False
      with J r' show ?thesis by(auto simp add: uncommitted-reads-see-hb-def)
    qed
  qed

  from r have r ∈ actions (?E (Suc n)) by simp
  let ?w = inv-into (actions (?E (Suc n))) (?φ (Suc n)) (ws (?φ (Suc n)) r))
  from J r ‹r ∈ ?C (Suc n)› have ws-rE-comm: ws (?φ (Suc n) r) ∈ ?φ (Suc n) ‘ ?C (Suc n)
    by(rule weakly-justified-write-seen-hb-read-committed)
  hence ?w ∈ ?C (Suc n) using C-sub-A by(auto simp add: inv-into-f-f[OF injSn])
  with C-sub-A have w: ?w ∈ actions (?E (Suc n)) by blast

  from ws-rE-comm C-sub-A have w-eq: ?φ (Suc n) ?w = ws (?φ (Suc n) r)
    by(auto simp: f-inv-into-f[where f=?φ (Suc n)])
  from r obtain ad al v
    where obsr: action-obs (?E (Suc n)) r = NormalAction (ReadMem ad al v) by cases
  hence adal-r: (ad, al) ∈ action-loc P (?E (Suc n)) r by simp
  from J wfaSn ‹r ∈ ?C (Suc n)›
  have obs-sim: action-obs (?E (Suc n)) r ≈ action-obs E (?φ (Suc n) r) ?φ (Suc n) r ∈ actions E
  by(auto dest: wf-action-translation-on-actionD simp add: committed-subset-actions-def is-commit-sequence-def)
  with obsr have rE: ?φ (Suc n) r ∈ read-actions E by(fastforce intro: read-actions.intros)
  from obs-sim obsr obtain v'
    where obsrE: action-obs E (?φ (Suc n) r) = NormalAction (ReadMem ad al v') by auto
  from wf-exec have is-write-seen P E ws by(rule wf-exec-is-write-seenD)
  from is-write-seenD[OF this rE obsrE]
  have ws (?φ (Suc n) r) ∈ write-actions E
    and (ad, al) ∈ action-loc P E (ws (?φ (Suc n) r))
    and nhb: ¬ P,E ⊢ ?φ (Suc n) r ≤hb ws (?φ (Suc n) r)
    and vol: is-volatile P al ⟹ ¬ P,E ⊢ ?φ (Suc n) r ≤so ws (?φ (Suc n) r) by simp-all

  show ?case
  proof(cases is-volatile P al)
    case False

    from wf-action-translation-on-actionD[OF wfaSn ‹?w ∈ ?C (Suc n)›]
    have action-obs (?E (Suc n)) ?w ≈ action-obs E (?φ (Suc n) ?w) by simp
    with w-eq have obs-sim-w: action-obs (?E (Suc n)) ?w ≈ action-obs E (ws (?φ (Suc n) r)) by
simp
    with ‹ws (?φ (Suc n) r) ∈ write-actions E› ‹?w ∈ actions (?E (Suc n))›
    have ?w ∈ write-actions (?E (Suc n))
      by cases(fastforce intro: write-actions.intros is-write-action.intros elim!: is-write-action.cases)
    from ‹(ad, al) ∈ action-loc P E (ws (?φ (Suc n) r))› obs-sim-w
```

have (*ad*, *al*) ∈ *action-loc P* (*?E* (*Suc n*)) *?w* **by** *cases*(*auto intro*: *action-loc-aux-intros*)
with *r adal-r* ‹*?w* ∈ *write-actions* (*?E* (*Suc n*))› *False*
have *P*,*?E* (*Suc n*) ⊢ *r* † *?w* **by**(*auto simp add*: *non-volatile-conflict-def*)
with *sc* ‹*r* ∈ *actions* (*?E* (*Suc n*))› *w*
have *P*,*?E* (*Suc n*) ⊢ *r* ≤*hb ?w* ∨ *P*,*?E* (*Suc n*) ⊢ *?w* ≤*hb r*
by(*rule correctly-synchronizedD*[*rule-format*, *OF sync* ‹*?E* (*Suc n*) ∈ $\mathcal{E}$› *wf*])
moreover from *J r* ‹*r* ∈ *?C* (*Suc n*)›
have *P*,*?E* (*Suc n*) ⊢ *?w* ≤*hb r* ⟷ *P*,*E* ⊢ *ws* (*?φ* (*Suc n*) *r*) ≤*hb ?φ* (*Suc n*) *r*
and ¬ *P*,*?E* (*Suc n*) ⊢ *r* ≤*hb ?w*
by(*simp-all add*: *happens-before-committed-weak-def*)
ultimately show *?thesis* **by** *auto*
next
case *True*
with *rE obsrE* **have** *?φ* (*Suc n*) *r* ∈ *sactions P E* **by** *cases* (*auto intro*: *sactionsI*)
moreover from ‹*ws* (*?φ* (*Suc n*) *r*) ∈ *write-actions E*› ‹(*ad*, *al*) ∈ *action-loc P E* (*ws* (*?φ* (*Suc n*) *r*))› *True*
have *ws* (*?φ* (*Suc n*) *r*) ∈ *sactions P E* **by** *cases*(*auto intro*!: *sactionsI elim*: *is-write-action.cases*)
moreover have *?φ* (*Suc n*) *r* ≠ *ws* (*?φ* (*Suc n*) *r*)
using ‹*ws* (*?φ* (*Suc n*) *r*) ∈ *write-actions E*› *rE* **by**(*auto dest*: *read-actions-not-write-actions*)
ultimately have *P*,*E* ⊢ *ws* (*?φ* (*Suc n*) *r*) ≤*so ?φ* (*Suc n*) *r*
using *total-sync-order*[*of P E*] *vol*[*OF True*] **by**(*auto dest*: *total-onPD*)
moreover from ‹*ws* (*?φ* (*Suc n*) *r*) ∈ *write-actions E*› ‹(*ad*, *al*) ∈ *action-loc P E* (*ws* (*?φ* (*Suc n*) *r*))› *True*
have *P* ⊢ (*action-tid E* (*ws* (*?φ* (*Suc n*) *r*)), *action-obs E* (*ws* (*?φ* (*Suc n*) *r*))) ⇝*sw*
(*action-tid E* (*?φ* (*Suc n*) *r*), *action-obs E* (*?φ* (*Suc n*) *r*))
by *cases*(*fastforce elim*!: *is-write-action.cases intro*: *synchronizes-with.intros addr-locsI simp add*: *obsrE*)
ultimately have *P*,*E* ⊢ *ws* (*?φ* (*Suc n*) *r*) ≤*sw ?φ* (*Suc n*) *r* **by**(*rule sync-withI*)
thus *?thesis* **unfolding** *po-sw-def* **by** *blast*
qed
qed
thus *P*,*E* ⊢ *ws r* ≤*hb r* **unfolding** *r* **.**
qed

corollary *drf*:
⟦ *correctly-synchronized P* $\mathcal{E}$; *legal-execution P* $\mathcal{E}$ (*E*, *ws*) ⟧
⟹ *sequentially-consistent P* (*E*, *ws*)
by(*erule drf-weak*)(*rule legal-imp-weakly-legal-execution*)

end

end

8.6 Sequentially consistent executions are legal

theory *SC-Legal* **imports**
JMM-Spec
begin

context *executions-base* **begin**

primrec *commit-for-sc* :: *'m prog* ⇒ (*'addr*, *'thread-id*) *execution* × *write-seen* ⇒ (*'addr*, *'thread-id*) *justification*

where
commit-for-sc P (E, ws) n =
(if enat n ≤ llength E then
let (E′, ws′) = SOME (E′, ws′). E′ ∈ ℰ ∧ P ⊢ (E′, ws′) √ ∧ enat n ≤ llength E′ ∧
ltake (enat (n − 1)) E = ltake (enat (n − 1)) E′ ∧
(n > 0 ⟶ action-tid E′ (n − 1) = action-tid E (n − 1) ∧
(if n − 1 ∈ read-actions E then sim-action else (=))
(action-obs E′ (n − 1)) (action-obs E (n − 1)) ∧
(∀ i < n − 1. i ∈ read-actions E ⟶ ws′ i = ws i)) ∧
(∀ r ∈ read-actions E′. n − 1 ≤ r ⟶ P,E′ ⊢ ws′ r ≤hb r)
in (|committed = {..<n}, justifying-exec = E′, justifying-ws = ws′, action-translation = id|)
else (|committed = actions E, justifying-exec = E, justifying-ws = ws, action-translation = id|))

end

context *sc-legal* **begin**

lemma *commit-for-sc-correct*:
assumes *E*: *E ∈ ℰ*
and *wf*: *P ⊢ (E, ws) √*
and *sc*: *sequentially-consistent P (E, ws)*
shows *wf-action-translation-commit-for-sc*:
⋀n. wf-action-translation E (commit-for-sc P (E, ws) n) (**is** *⋀n. ?thesis1 n*)
and *commit-for-sc-in-ℰ*:
⋀n. justifying-exec (commit-for-sc P (E, ws) n) ∈ ℰ (**is** *⋀n. ?thesis2 n*)
and *commit-for-sc-wf*:
⋀n. P ⊢ (justifying-exec (commit-for-sc P (E, ws) n), justifying-ws (commit-for-sc P (E, ws) n))
√
(**is** *⋀n. ?thesis3 n*)
and *commit-for-sc-justification*:
P ⊢ (E, ws) justified-by commit-for-sc P (E, ws) (**is** *?thesis4*)
proof −
let *?φ = commit-for-sc P (E, ws)*

note [*simp*] = *split-beta*
from *wf* **have** *tsok*: *thread-start-actions-ok E* **by** *simp*

let *?P = λn (E′, ws′). E′ ∈ ℰ ∧ P ⊢ (E′, ws′) √ ∧ (enat n ≤ llength E ⟶ enat n ≤ llength E′) ∧*
ltake (enat (n − 1)) E = ltake (enat (n − 1)) E′ ∧
(n > 0 ⟶ action-tid E′ (n − 1) = action-tid E (n − 1) ∧
(if n − 1 ∈ read-actions E then sim-action else (=))
(action-obs E′ (n − 1)) (action-obs E (n − 1)) ∧
(∀ i < n − 1. i ∈ read-actions E ⟶ ws′ i = ws i)) ∧
(∀ r ∈ read-actions E′. n − 1 ≤ r ⟶ P,E′ ⊢ ws′ r ≤hb r)
define *E′ ws′* **where** *E′ n = fst (Eps (?P n))* **and** *ws′ n = snd (Eps (?P n))* **for** *n*
hence [*simp*]:
⋀n. commit-for-sc P (E, ws) n =
(if enat n ≤ llength E
then (|committed = {..<n}, justifying-exec = E′ n, justifying-ws = ws′ n, action-translation = id|)
else (|committed = actions E, justifying-exec = E, justifying-ws = ws, action-translation = id|))
by *simp*
note [*simp del*] = *commit-for-sc.simps*

have *(∀ n. ?thesis1 n) ∧ (∀ n. ?thesis2 n) ∧ (∀ n. ?thesis3 n) ∧ ?thesis4*

```
  unfolding is-justified-by.simps is-commit-sequence-def justification-well-formed-def committed-subset-actions-def
      happens-before-committed-def sync-order-committed-def value-written-committed-def uncommitted-reads-see-hb-def
    committed-reads-see-committed-writes-def external-actions-committed-def wf-action-translations-def
      write-seen-committed-def
  proof(intro conjI strip LetI)

    show committed (?φ 0) = {}
      by(auto simp add: actions-def zero-enat-def[symmetric])
    show actions-E: actions E = (⋃n. action-translation (?φ n) ‘ committed (?φ n))
      by(auto simp add: actions-def less-le-trans[where y=enat n for n] split: if-split-asm)
     hence committed-subset-E: ⋀n. action-translation (?φ n) ‘ committed (?φ n) ⊆ actions E by fastforce

    { fix n
      have ?P n (Eps (?P n))
      proof(cases n)
        case 0
        from E-hb-completion[OF E wf, of 0] have ∃Ews. ?P 0 Ews
          by(fastforce simp add: zero-enat-def[symmetric])
        thus ?thesis unfolding 0 by(rule someI-ex)
      next
        case (Suc n')
        moreover
        from sc have sc': ⋀a. ⟦ a < n'; a ∈ read-actions E ⟧ ⟹ P,E ⊢ a ↝mrw ws a
          by(simp add: sequentially-consistent-def)
        from E-hb-completion[OF E wf this, of n']
        obtain E' ws' where E' ∈ ℰ and P ⊢ (E', ws') √
          and eq: ltake (enat n') E = ltake (enat n') E'
          and hb: ∀a∈read-actions E'. if a < n' then ws' a = ws a else P,E' ⊢ ws' a ≤hb a
          and n-sim: action-tid E' n' = action-tid E n'
            (if n' ∈ read-actions E then sim-action else (=)) (action-obs E' n') (action-obs E n')
          and n: n' ∈ actions E ⟹ n' ∈ actions E' by blast
        moreover {
          assume enat n ≤ llength E
          with n Suc have enat n ≤ llength E'
            by(simp add: actions-def Suc-ile-eq) }
        moreover {
          fix i
          assume i ∈ read-actions E
          moreover from eq have ltake (enat n') E [≈] ltake (enat n') E'
            by(rule eq-into-sim-actions)
          moreover assume i < n'
          hence enat i < enat n' by simp
          ultimately have i ∈ read-actions E' by(rule read-actions-change-prefix)
          with hb[rule-format, OF this] ‹i < n'›
          have ws' i = ws i by simp }
        ultimately have ?P n (E', ws') by simp
        thus ?thesis by(rule someI)
      qed }
    hence P [simplified]: ⋀n. ?P n (E' n, ws' n) by(simp add: E'-def ws'-def)
    { fix n
      assume n-E: enat n ≤ llength E
      have ltake (enat n) (E' n) [≈] ltake (enat n) E unfolding sim-actions-def
```

```
    proof(rule llist-all2-all-lnthI)
     show llength (ltake (enat n) (E' n)) = llength (ltake (enat n) E)
      using n-E P[of n] by(clarsimp simp add: min-def)
    next
     fix n'
     assume n': enat n' < llength (ltake (enat n) (E' n))
     show (λ(t, a) (t', a'). t = t' ∧ a ≈ a') (lnth (ltake (enat n) (E' n)) n') (lnth (ltake (enat n)
E) n')
     proof(cases n = Suc n')
      case True
      with P[of n] show ?thesis
       by(simp add: action-tid-def action-obs-def lnth-ltake split: if-split-asm)
     next
      case False
      with n' have n' < n − 1 by auto
      moreover from P[of n] have lnth (ltake (enat (n − 1)) (E' n)) n' = lnth (ltake (enat (n −
1)) E) n' by simp
      ultimately show ?thesis by(simp add: lnth-ltake)
     qed
    qed }
   note sim = this
   note len-eq = llist-all2-llengthD[OF this[unfolded sim-actions-def]]

   { fix n
    show wf-action-translation E (?φ n)
    proof(cases enat n ≤ llength E)
     case False thus ?thesis by(simp add: wf-action-translation-on-def)
    next
     case True
     hence {..<n} ⊆ actions E
      by(auto simp add: actions-def min-def less-le-trans[where y=enat n] split: if-split-asm)
     moreover
     from True len-eq[OF True] have {..<n} ⊆ actions (E' n)
      by(auto simp add: actions-def min-def less-le-trans[where y=enat n] split: if-split-asm)
     moreover {
      fix a assume a < n
      moreover from sim[OF True]
      have action-tid (ltake (enat n) (E' n)) a = action-tid (ltake (enat n) E) a
       action-obs (ltake (enat n) (E' n)) a ≈ action-obs (ltake (enat n) E) a
       by(rule sim-actions-action-tidD sim-actions-action-obsD)+
      ultimately have action-tid (E' n) a = action-tid E a action-obs (E' n) a ≈ action-obs E a
       by(simp-all add: action-tid-def action-obs-def lnth-ltake) }
     ultimately show ?thesis by(auto simp add: wf-action-translation-on-def del: subsetI)
    qed
    thus wf-action-translation E (?φ n) . }
   note wfa = this

   { fix n from P E show justifying-exec (?φ n) ∈ E
     by(cases enat n ≤ llength E) simp-all }
   note En = this

   { fix n from P wf show P ⊢ (justifying-exec (?φ n), justifying-ws (?φ n)) √
     by(cases enat n ≤ llength E) simp-all
    thus P ⊢ (justifying-exec (?φ n), justifying-ws (?φ n)) √ . }
```

```
  note wfn = this

  { fix n show action-translation (?φ n) ‘ committed (?φ n) ⊆ action-translation (?φ (Suc n)) ‘ committed (?φ (Suc n))
    by(auto simp add: actions-def less-le-trans[where y=enat n]) (metis Suc-ile-eq order-less-imp-le)
}
  note committed-subset = this

  { fix n
    from len-eq[of n] have enat n ≤ llength E ⟹ {..<n} ⊆ actions (E' n)
      by(auto simp add: E'-def actions-def min-def less-le-trans[where y=enat n] split: if-split-asm)
    thus committed (?φ n) ⊆ actions (justifying-exec (?φ n))
      by(simp add: actions-def E'-def) }
  note committed-actions = this

  fix n
  show happens-before P (justifying-exec (?φ n)) |‘ committed (?φ n) =
       inv-imageP (happens-before P E) (action-translation (?φ n)) |‘ committed (?φ n)
  proof(cases enat n ≤ llength E)
    case False thus ?thesis by simp
  next
    case True thus ?thesis
    proof(safe intro!: ext)
      fix a b
      assume hb: P,justifying-exec (?φ n) ⊢ a ≤hb b
        and a: a ∈ committed (?φ n)
        and b: b ∈ committed (?φ n)
      from hb True have P,E' n ⊢ a ≤hb b by(simp add: E'-def)
      moreover note tsok sim[OF True]
      moreover from a b True
      have enat a < enat n enat b < enat n by simp-all
      ultimately have P,E ⊢ a ≤hb b by(rule happens-before-change-prefix)
      thus P,E ⊢ action-translation (?φ n) a ≤hb action-translation (?φ n) b by simp
    next
      fix a b
      assume a: a ∈ committed (?φ n)
        and b: b ∈ committed (?φ n)
        and hb: P,E ⊢ action-translation (?φ n) a ≤hb action-translation (?φ n) b
      from hb True have P,E ⊢ a ≤hb b by simp
      moreover from wfn[of n] True have thread-start-actions-ok (E' n) by(simp)
      moreover from sim[OF True] have ltake (enat n) E [≈] ltake (enat n) (E' n)
        by(rule sim-actions-sym)
      moreover from a b True have enat a < enat n enat b < enat n by simp-all
      ultimately have P,E' n ⊢ a ≤hb b by(rule happens-before-change-prefix)
      thus P,justifying-exec (?φ n) ⊢ a ≤hb b using True by(simp add: E'-def)
    qed
  qed

  show sync-order P (justifying-exec (?φ n)) |‘ committed (?φ n) =
    inv-imageP (sync-order P E) (action-translation (?φ n)) |‘ committed (?φ n)
  proof(cases enat n ≤ llength E)
    case False thus ?thesis by simp
  next
    case True thus ?thesis
```

```
  proof(safe intro!: ext)
    fix a b
    assume hb: P,justifying-exec (?φ n) ⊢ a ≤so b
      and a: a ∈ committed (?φ n)
      and b: b ∈ committed (?φ n)
    from hb True have P,E' n ⊢ a ≤so b by(simp add: E'-def)
    moreover note sim[OF True]
    moreover from a b True
    have enat a < enat n enat b < enat n by simp-all
    ultimately have P,E ⊢ a ≤so b by(rule sync-order-change-prefix)
    thus P,E ⊢ action-translation (?φ n) a ≤so action-translation (?φ n) b by simp
  next
    fix a b
    assume a: a ∈ committed (?φ n)
      and b: b ∈ committed (?φ n)
      and hb: P,E ⊢ action-translation (?φ n) a ≤so action-translation (?φ n) b
    from hb True have P,E ⊢ a ≤so b by simp
    moreover from sim[OF True] have ltake (enat n) E [≈] ltake (enat n) (E' n)
      by(rule sim-actions-sym)
    moreover from a b True have enat a < enat n enat b < enat n by simp-all
    ultimately have P,E' n ⊢ a ≤so b by(rule sync-order-change-prefix)
    thus P,justifying-exec (?φ n) ⊢ a ≤so b using True by(simp add: E'-def)
  qed
qed

{ fix w w' adal
  assume w: w ∈ write-actions (justifying-exec (?φ n)) ∩ committed (?φ n)
    and w': w' = action-translation (?φ n) w
    and adal: adal ∈ action-loc P E w'
  show value-written P (justifying-exec (?φ n)) w adal = value-written P E w' adal
  proof(cases enat n ≤ llength E)
    case False thus ?thesis using w' by simp
  next
    case True
    note n-E = this
    have action-obs E w = action-obs (E' n) w
    proof(cases w < n - 1)
      case True
      with P[of n] w' n-E show ?thesis
        by(clarsimp simp add: action-obs-change-prefix-eq)
    next
      case False
      with w True have w = n - 1 n > 0 by auto
      moreover
      with True have w ∈ actions E
        by(simp add: actions-def)(metis Suc-ile-eq Suc-pred)
      with True w wf-action-translation-on-actionD[OF wfa, of w n] w'
      have w' ∈ write-actions E
        by(auto intro!: write-actions.intros elim!: write-actions.cases is-write-action.cases)
      hence w' ∉ read-actions E by(blast dest: read-actions-not-write-actions)
      ultimately show ?thesis using P[of n] w' True by clarsimp
    qed
    with True w' show ?thesis by(cases adal)(simp add: value-written.simps)
  qed }
```

```
  { fix r' r r''
    assume r': r' ∈ read-actions (justifying-exec (commit-for-sc P (E, ws) n)) ∩ committed (?φ n)
      and r: r = action-translation (?φ n) r'
     and r'': r'' = inv-into (actions (justifying-exec (?φ (Suc n)))) (action-translation (?φ (Suc n)))
r
    from r' r committed-subset[of n] have r ∈ actions E
    by(auto split: if-split-asm elim!: read-actions.cases simp add: actions-def Suc-ile-eq less-trans[where
y=enat n])
    with r' r have r-actions: r ∈ read-actions E
    by(fastforce dest: wf-action-translation-on-actionD[OF wfa] split: if-split-asm elim!: read-actions.cases
intro: read-actions.intros)
    moreover from r' committed-subset[of n] committed-actions[of Suc n]
   have r' ∈ actions (justifying-exec (?φ (Suc n))) by(auto split: if-split-asm elim: read-actions.cases)
    ultimately have r'' = r' using r' r r'' by(cases enat (Suc n) ≤ llength E) simp-all
    moreover from r' have r' < n
    by(simp add: actions-def split: if-split-asm)(metis enat-ord-code(2) linorder-linear order-less-le-trans)
    ultimately show action-translation (?φ (Suc n)) (justifying-ws (?φ (Suc n)) r'') = ws r
      using P[of Suc n] r' r r-actions by(clarsimp split: if-split-asm) }

  { fix r'
    assume r': r' ∈ read-actions (justifying-exec (?φ (Suc n)))
    show action-translation (?φ (Suc n)) r' ∈ action-translation (?φ n) ‘ committed (?φ n) ∨
         P,justifying-exec (?φ (Suc n)) ⊢ justifying-ws (?φ (Suc n)) r' ≤hb r' (is ?committed ∨ ?hb)
    proof(cases r' < n)
      case True
      hence ?committed using r'
     by(auto elim!: actionsE split: if-split-asm dest!: read-actions-actions)(metis Suc-ile-eq linorder-not-le
not-less-iff-gr-or-eq)
      thus ?thesis ..
    next
      case False
      hence r' ≥ n by simp
      hence enat (Suc n) ≤ llength E using False r'
            by(auto split: if-split-asm dest!: read-actions-actions elim!: actionsE) (metis Suc-ile-eq
enat-ord-code(2) not-le-imp-less order-less-le-trans)
      hence ?hb using P[of Suc n] r' ‹r' ≥ n› by simp
      thus ?thesis ..
    qed }

  { fix r' r C-n
    assume r': r' ∈ read-actions (justifying-exec (?φ (Suc n))) ∩ committed (?φ (Suc n))
      and r: r = action-translation (?φ (Suc n)) r'
      and C-n: C-n = action-translation (?φ n) ‘ committed (?φ n)
    show r ∈ C-n ∨ action-translation (?φ (Suc n)) (justifying-ws (?φ (Suc n)) r') ∈ C-n ∧ ws r ∈
C-n
      (is - ∨ (?C-ws-n ∧ ?C-ws))
    proof(cases r ∈ C-n)
      case True thus ?thesis ..
    next
      case False
      with r' r C-n have [simp]: r' = n
        apply(auto split: if-split-asm dest!: read-actions-actions elim!: actionsE)
        apply(metis enat-ord-code(1) less-SucI less-eq-Suc-le not-less-eq-eq order-trans)
```

```
      by (metis Suc-ile-eq enat-ord-code(1) leD leI linorder-cases)
    from r' have len-E: enat (Suc n) ≤ llength E
      by(clarsimp simp add: actions-def Suc-ile-eq split: if-split-asm)
    with r' P[of Suc n] have P,justifying-exec (?φ (Suc n)) ⊢ ws' (Suc n) r' ≤hb r' by(simp)
  hence justifying-exec (?φ (Suc n)) ⊢ ws' (Suc n) r' ≤a r' by(rule happens-before-into-action-order)
    moreover from r' have r' ∈ read-actions (justifying-exec (?φ (Suc n))) by simp
     moreover then obtain ad al v where action-obs (justifying-exec (?φ (Suc n))) r' = NormalAction (ReadMem ad al v)
      by cases auto
    with wfn[of Suc n] ‹r' ∈ read-actions -› len-E obtain adal
      where ws' (Suc n) r' ∈ write-actions (justifying-exec (?φ (Suc n)))
      and adal ∈ action-loc P (justifying-exec (?φ (Suc n))) r'
      and adal ∈ action-loc P (justifying-exec (?φ (Suc n))) (ws' (Suc n) r')
      by(clarsimp)(auto dest: is-write-seenD)
    moreover {
      from En[of Suc n] len-E have E' (Suc n) ∈ ℰ by simp
      moreover
      fix a assume a ∈ read-actions (justifying-exec (?φ (Suc n))) and a < r'
      hence a ∈ read-actions (E' (Suc n)) enat a < enat (Suc n) using len-E by simp-all
      with sim[OF len-E] have a: a ∈ read-actions E by −(rule read-actions-change-prefix)
   with ‹a < r'› have mrw: P,E ⊢ a ↝mrw ws a using sc by(simp add: sequentially-consistent-def)
      from P[of Suc n] ‹a < r'› a len-E have ws a = ws' (Suc n) a by simp
      with mrw have mrw': P,E ⊢ a ↝mrw ws' (Suc n) a by simp
      moreover from wfn[of Suc n] wf len-E have thread-start-actions-ok (E' (Suc n)) by(simp)
      moreover note sim[OF len-E, symmetric]
      moreover from E wf mrw' have ws' (Suc n) a < a
        by(rule mrw-before)(erule sequentially-consistentE[OF sc])
      with ‹a < r'› have ws' (Suc n) a < r' by simp
      ultimately have P,E' (Suc n) ⊢ a ↝mrw ws' (Suc n) a using ‹a < r'›
        by −(rule mrw-change-prefix, simp+)
     hence P,justifying-exec (?φ (Suc n)) ⊢ a ↝mrw justifying-ws (?φ (Suc n)) a using len-E by simp
    }
    ultimately have ws' (Suc n) r' < r' by(rule action-order-read-before-write[OF En wfn])
     with len-E C-n have ?C-ws-n by clarsimp (metis Suc-ile-eq linorder-le-cases order-less-irrefl order-trans)
    moreover
    from r' have r' ∈ committed (?φ (Suc n)) by blast
    with r' r len-E wf-action-translation-on-actionD[OF wfa this] committed-subset-E[of Suc n]
     have r ∈ read-actions E by(fastforce elim!: read-actions.cases intro: read-actions.intros split: if-split-asm)
    with sc obtain P,E ⊢ r ↝mrw ws r by(rule sequentially-consistentE)
    with E wf have ws r < r by(rule mrw-before)(rule sequentially-consistentE[OF sc])
    with C-n len-E r have ?C-ws by(auto simp add: Suc-ile-eq)
    ultimately show ?thesis by simp
  qed }

  { fix a a'
    assume a: a ∈ external-actions (justifying-exec (?φ n))
      and a': a' ∈ committed (?φ n)
      and hb: P,justifying-exec (?φ n) ⊢ a ≤hb a'
    from hb have justifying-exec (?φ n) ⊢ a ≤a a'
      by(rule happens-before-into-action-order)
    with a have a ≤ a' by(auto elim!: action-orderE dest: external-actions-not-new)
```

with a' a **show** $a \in$ *committed* (*?φ* n) **by**(*auto elim*: *external-actions.cases*) **}**
qed
thus $\bigwedge n$. *?thesis1* n $\bigwedge n$. *?thesis2* n $\bigwedge n$. *?thesis3* n *?thesis4*
by *blast+*
qed

theorem *SC-is-legal*:
assumes E: $E \in \mathcal{E}$
and *wf*: $P \vdash (E, ws)\ \surd$
and *sc*: *sequentially-consistent* P (E, ws)
shows *legal-execution* P $\mathcal{E}$ (E, ws)
using E *wf*
apply(*rule legal-executionI*)
apply(*rule commit-for-sc-correct*[*OF assms*])
apply *clarify*
apply(*unfold o-apply*)
apply(*rule commit-for-sc-in-*$\mathcal{E}$[*OF assms*])
done

end

context *jmm-consistent* **begin**

theorem *consistent*:
assumes $E \in \mathcal{E}$ $P \vdash (E, ws)\ \surd$
shows $\exists E \in \mathcal{E}.\ \exists ws.$ *legal-execution* P $\mathcal{E}$ (E, ws)
proof −
from $\mathcal{E}$*-sequential-completion*[*OF assms*, *of 0*]
obtain E' ws' **where** $E' \in \mathcal{E}$ $P \vdash (E', ws')\ \surd$ *sequentially-consistent* P (E', ws') **by** *auto*
moreover hence *legal-execution* P $\mathcal{E}$ (E', ws') **by**(*rule SC-is-legal*)
ultimately show *?thesis* **by** *blast*
qed

end

end

8.7 Non-speculative prefixes of executions

theory *Non-Speculative* **imports**
JMM-Spec
../Framework/FWLTS
begin

declare *addr-locsI* [*simp*]

8.7.1 Previously written values

fun *w-value* ::
$'m$ *prog* ⇒ (($'addr$ × *addr-loc*) ⇒ $'addr$ *val set*) ⇒ ($'addr$, $'thread\text{-}id$) *obs-event action*
⇒ (($'addr$ × *addr-loc*) ⇒ $'addr$ *val set*)
where
w-value P *vs* (*NormalAction* (*WriteMem ad al v*)) = *vs*((*ad*, *al*) := *insert v* (*vs* (*ad*, *al*)))
| *w-value* P *vs* (*NormalAction* (*NewHeapElem ad hT*)) =

```
   (λ(ad', al). if ad = ad' ∧ al ∈ addr-locs P hT
                then insert (addr-loc-default P hT al) (vs (ad, al))
                else vs (ad', al))
| w-value P vs - = vs

lemma w-value-cases:
  obtains ad al v where x = NormalAction (WriteMem ad al v)
  | ad hT where x = NormalAction (NewHeapElem ad hT)
  | ad M vs v where x = NormalAction (ExternalCall ad M vs v)
  | ad al v where x = NormalAction (ReadMem ad al v)
  | t where x = NormalAction (ThreadStart t)
  | t where x = NormalAction (ThreadJoin t)
  | ad where x = NormalAction (SyncLock ad)
  | ad where x = NormalAction (SyncUnlock ad)
  | t where x = NormalAction (ObsInterrupt t)
  | t where x = NormalAction (ObsInterrupted t)
  | x = InitialThreadAction
  | x = ThreadFinishAction
by pat-completeness

abbreviation w-values ::
  'm prog ⇒ (('addr × addr-loc) ⇒ 'addr val set) ⇒ ('addr, 'thread-id) obs-event action list
  ⇒ (('addr × addr-loc) ⇒ 'addr val set)
where w-values P ≡ foldl (w-value P)

lemma in-w-valuesD:
  assumes w: v ∈ w-values P vs0 obs (ad, al)
  and v: v ∉ vs0 (ad, al)
  shows ∃ obs' wa obs''. obs = obs' @ wa # obs'' ∧ is-write-action wa ∧ (ad, al) ∈ action-loc-aux P
wa ∧
          value-written-aux P wa al = v
  (is ?concl obs)
using w
proof(induction obs rule: rev-induct)
  case Nil thus ?case using v by simp
next
  case (snoc ob obs)
  from snoc.IH show ?case
  proof(cases v ∈ w-values P vs0 obs (ad, al))
    case False thus ?thesis using ‹v ∈ w-values P vs0 (obs @ [ob]) (ad, al)›
      by(cases ob rule: w-value-cases)(auto 4 4 intro: action-loc-aux-intros split: if-split-asm simp add:
addr-locs-def split: htype.split-asm)
  qed fastforce
qed

lemma w-values-WriteMemD:
  assumes NormalAction (WriteMem ad al v) ∈ set obs
  shows v ∈ w-values P vs0 obs (ad, al)
using assms
proof(induct obs rule: rev-induct)
 apply simp
apply clarsimp
proof(erule disjE)
 apply clarsimp
```

apply *clarsimp*
apply(*case-tac x rule*: *w-value-cases*)
apply *auto*
done

lemma *w-values-new-actionD*:
assumes *NormalAction* (*NewHeapElem ad hT*) ∈ *set obs* (*ad*, *al*) ∈ *action-loc-aux P* (*NormalAction* (*NewHeapElem ad hT*))
shows *addr-loc-default P hT al* ∈ *w-values P vs0 obs* (*ad*, *al*)
using *assms*
apply(*induct obs rule*: *rev-induct*)
apply *simp*
apply *clarsimp*
apply(*rename-tac w' obs*)
apply(*case-tac w' rule*: *w-value-cases*)
apply(*auto simp add*: *split-beta*)
done

lemma *w-value-mono*: *vs0 adal* ⊆ *w-value P vs0 ob adal*
by(*cases ob rule*: *w-value-cases*)(*auto split*: *if-split-asm simp add*: *split-beta*)

lemma *w-values-mono*: *vs0 adal* ⊆ *w-values P vs0 obs adal*
by(*induct obs rule*: *rev-induct*)(*auto del*: *subsetI intro*: *w-value-mono subset-trans*)

lemma *w-value-greater*: *vs0* ≤ *w-value P vs0 ob*
by(*rule le-funI*)(*rule w-value-mono*)

lemma *w-values-greater*: *vs0* ≤ *w-values P vs0 obs*
by(*rule le-funI*)(*rule w-values-mono*)

lemma *w-values-eq-emptyD*:
assumes *w-values P vs0 obs adal* = {}
and *w* ∈ *set obs* **and** *is-write-action w* **and** *adal* ∈ *action-loc-aux P w*
shows *False*
using *assms*(*4*) *assms*(*1*−*3*)
apply(*cases rule*: *action-loc-aux-cases*)
apply(*auto dest*!: *w-values-new-actionD*[**where** *?vs0.0*=*vs0* **and** *P*=*P*] *w-values-WriteMemD*[**where** *?vs0.0*=*vs0* **and** *P*=*P*])
apply *blast*
done

8.7.2 Coinductive version of non-speculative prefixes

coinductive *non-speculative* ::
'm prog ⇒ (*'addr* × *addr-loc* ⇒ *'addr val set*) ⇒ (*'addr*, *'thread-id*) *obs-event action llist* ⇒ *bool*
for *P* :: *'m prog*
where
LNil: *non-speculative P vs LNil*
| *LCons*:
⟦ *case ob of NormalAction* (*ReadMem ad al v*) ⇒ *v* ∈ *vs* (*ad*, *al*) | - ⇒ *True*;
non-speculative P (*w-value P vs ob*) *obs* ⟧
⟹ *non-speculative P vs* (*LCons ob obs*)

inductive-simps *non-speculative-simps* [*simp*]:
non-speculative P vs LNil
non-speculative P vs (*LCons ob obs*)

lemma *non-speculative-lappend*:
assumes *lfinite obs*
shows *non-speculative P vs* (*lappend obs obs′*) ⟷
non-speculative P vs obs ∧ *non-speculative P* (*w-values P vs* (*list-of obs*)) *obs′*
(**is** *?concl vs obs*)
using *assms*
proof(*induct arbitrary*: *vs*)
case *lfinite-LNil* **thus** *?case* **by** *simp*
next
case (*lfinite-LConsI obs ob*)
have *?concl* (*w-value P vs ob*) *obs* **by** *fact*
thus *?case* **using** ‹*lfinite obs*› **by** *simp*
qed

lemma
assumes *non-speculative P vs obs*
shows *non-speculative-ltake*: *non-speculative P vs* (*ltake n obs*) (**is** *?thesis1*)
and *non-speculative-ldrop*: *non-speculative P* (*w-values P vs* (*list-of* (*ltake n obs*))) (*ldrop n obs*) (**is** *?thesis2*)
proof −
note *assms*
also have *obs* = *lappend* (*ltake n obs*) (*ldrop n obs*) **by**(*simp add*: *lappend-ltake-ldrop*)
finally have *?thesis1* ∧ *?thesis2*
by(*cases n*)(*simp-all add*: *non-speculative-lappend del*: *lappend-ltake-enat-ldropn*)
thus *?thesis1 ?thesis2* **by** *blast*+
qed

lemma *non-speculative-coinduct-append* [*consumes 1*, *case-names non-speculative*, *case-conclusion non-speculative LNil lappend*]:
assumes *major*: *X vs obs*
and *step*: ⋀*vs obs*. *X vs obs*
⟹ *obs* = *LNil* ∨
(∃ *obs′ obs″*. *obs* = *lappend obs′ obs″* ∧ *obs′* ≠ *LNil* ∧ *non-speculative P vs obs′* ∧
(*lfinite obs′* ⟶ (*X* (*w-values P vs* (*list-of obs′*)) *obs″* ∨
non-speculative P (*w-values P vs* (*list-of obs′*)) *obs″*)))
(**is** ⋀*vs obs*. - ⟹ - ∨ *?step vs obs*)
shows *non-speculative P vs obs*
proof −
from *major*
have ∃ *obs′ obs″*. *obs* = *lappend* (*llist-of obs′*) *obs″* ∧ *non-speculative P vs* (*llist-of obs′*) ∧
X (*w-values P vs obs′*) *obs″*
by(*auto intro*: *exI*[**where** *x*=[]])
thus *?thesis*
proof(*coinduct*)
case (*non-speculative vs obs*)
then obtain *obs′ obs″*
where *obs*: *obs* = *lappend* (*llist-of obs′*) *obs″*
and *sc-obs′*: *non-speculative P vs* (*llist-of obs′*)
and *X*: *X* (*w-values P vs obs′*) *obs″* **by** *blast*

```
  show ?case
  proof(cases obs')
    case Nil
    with X have X vs obs'' by simp
    from step[OF this] show ?thesis
    proof
      assume obs'' = LNil
      with Nil obs show ?thesis by simp
    next
      assume ?step vs obs''
      then obtain obs''' obs'''' 
        where obs'': obs'' = lappend obs''' obs'''' and obs''' ≠ LNil
        and sc-obs''': non-speculative P vs obs'''
        and fin: lfinite obs''' ⟹ X (w-values P vs (list-of obs''')) obs'''' ∨
                                   non-speculative P (w-values P vs (list-of obs''')) obs''''
        by blast
      from ‹obs''' ≠ LNil› obtain ob obs''''' where obs''': obs''' = LCons ob obs'''''
        unfolding neq-LNil-conv by blast
      with Nil obs'' obs have concl1: obs = LCons ob (lappend obs''''' obs'''') by simp
      have concl2: case ob of NormalAction (ReadMem ad al v) ⇒ v ∈ vs (ad, al) | - ⇒ True
        using sc-obs''' obs''' by simp

      show ?thesis
      proof(cases lfinite obs''')
        case False
        hence lappend obs''''' obs'''' = obs''''' using obs''' by(simp add: lappend-inf)
        hence non-speculative P (w-value P vs ob) (lappend obs''''' obs'''')
          using sc-obs''' obs''' by simp
        with concl1 concl2 have ?LCons by blast
        thus ?thesis by simp
      next
        case True
        with obs''' obtain obs'''''' where obs''''': obs''''' = llist-of obs''''''
          by simp(auto simp add: lfinite-eq-range-llist-of)
        from fin[OF True] have ?LCons
        proof
          assume X: X (w-values P vs (list-of obs''')) obs''''
          hence X (w-values P (w-value P vs ob) obs'''''') obs''''
            using obs''''' obs''' by simp
          moreover from obs'''''
          have lappend obs''''' obs'''' = lappend (llist-of obs'''''') obs'''' by simp
          moreover have non-speculative P (w-value P vs ob) (llist-of obs'''''')
            using sc-obs''' obs''' obs''''' by simp
          ultimately show ?thesis using concl1 concl2 by blast
        next
          assume non-speculative P (w-values P vs (list-of obs''')) obs''''
          with sc-obs''' obs''''' obs'''
          have non-speculative P (w-value P vs ob) (lappend obs''''' obs'''')
            by(simp add: non-speculative-lappend)
          with concl1 concl2 show ?thesis by blast
        qed
        thus ?thesis by simp
      qed
    qed
```

```
    next
      case (Cons ob obs''')
      hence obs = LCons ob (lappend (llist-of obs''') obs'')
        using obs by simp
      moreover from sc-obs' Cons
      have case ob of NormalAction (ReadMem ad al v) ⇒ v ∈ vs (ad, al) | - ⇒ True
        and non-speculative P (w-value P vs ob) (llist-of obs''') by simp-all
      moreover from X Cons have X (w-values P (w-value P vs ob) obs''') obs'' by simp
      ultimately show ?thesis by blast
    qed
  qed
qed

lemma non-speculative-coinduct-append-wf
  [consumes 2, case-names non-speculative, case-conclusion non-speculative LNil lappend]:
  assumes major: X vs obs a
  and wf: wf R
  and step: ⋀vs obs a. X vs obs a
    ⟹ obs = LNil ∨
      (∃ obs' obs'' a'. obs = lappend obs' obs'' ∧ non-speculative P vs obs' ∧ (obs' = LNil ⟶ (a', a)
∈ R) ∧
                    (lfinite obs' ⟶ X (w-values P vs (list-of obs')) obs'' a' ∨
                                     non-speculative P (w-values P vs (list-of obs')) obs''))
    (is ⋀vs obs a. - ⟹ - ∨ ?step vs obs a)
  shows non-speculative P vs obs
proof −
  { fix vs obs a
    assume X vs obs a
    with wf
    have obs = LNil ∨ (∃ obs' obs''. obs = lappend obs' obs'' ∧ obs' ≠ LNil ∧ non-speculative P vs
obs' ∧
        (lfinite obs' ⟶ (∃ a. X (w-values P vs (list-of obs')) obs'' a) ∨
                        non-speculative P (w-values P vs (list-of obs')) obs''))
      (is - ∨ ?step-concl vs obs)
    proof(induct a arbitrary: vs obs rule: wf-induct[consumes 1, case-names wf])
      case (wf a)
      note IH = wf.hyps[rule-format]
      from step[OF ‹X vs obs a›]
      show ?case
      proof
        assume obs = LNil thus ?thesis ..
      next
        assume ?step vs obs a
        then obtain obs' obs'' a'
          where obs: obs = lappend obs' obs''
          and sc-obs': non-speculative P vs obs'
          and decr: obs' = LNil ⟹ (a', a) ∈ R
          and fin: lfinite obs' ⟹
                X (w-values P vs (list-of obs')) obs'' a' ∨
                non-speculative P (w-values P vs (list-of obs')) obs''
          by blast
        show ?case
        proof(cases obs' = LNil)
          case True
```

```
        hence lfinite obs' by simp
        from fin[OF this] show ?thesis
        proof
          assume X: X (w-values P vs (list-of obs')) obs'' a'
          from True have (a', a) ∈ R by(rule decr)
          from IH[OF this X] show ?thesis
          proof
            assume obs'' = LNil
            with True obs have obs = LNil by simp
            thus ?thesis ..
          next
            assume ?step-concl (w-values P vs (list-of obs')) obs''
            hence ?step-concl vs obs using True obs by simp
            thus ?thesis ..
          qed
        next
          assume non-speculative P (w-values P vs (list-of obs')) obs''
          thus ?thesis using obs True
           by cases(auto cong: action.case-cong obs-event.case-cong intro: exI[where x=LCons x LNil
for x])
        qed
      next
        case False
        with obs sc-obs' fin show ?thesis by auto
      qed
    qed
  qed }
  note step' = this

  from major show ?thesis
  proof(coinduction arbitrary: vs obs a rule: non-speculative-coinduct-append)
    case (non-speculative vs obs)
    thus ?case by simp(rule step')
  qed
qed

lemma non-speculative-nthI:
  (⋀i ad al v.
    ⟦ enat i < llength obs; lnth obs i = NormalAction (ReadMem ad al v);
      non-speculative P vs (ltake (enat i) obs) ⟧
    ⟹ v ∈ w-values P vs (list-of (ltake (enat i) obs)) (ad, al))
  ⟹ non-speculative P vs obs
proof(coinduction arbitrary: vs obs rule: non-speculative.coinduct)
  case (non-speculative vs obs)
  hence nth:
    ⋀i ad al v. ⟦ enat i < llength obs; lnth obs i = NormalAction (ReadMem ad al v);
                  non-speculative P vs (ltake (enat i) obs) ⟧
    ⟹ v ∈ w-values P vs (list-of (ltake (enat i) obs)) (ad, al) by blast
  show ?case
  proof(cases obs)
    case LNil thus ?thesis by simp
  next
    case (LCons ob obs')
    { fix ad al v
```

```
      assume ob = NormalAction (ReadMem ad al v)
      with nth[of 0 ad al v] LCons
      have v ∈ vs (ad, al) by(simp add: zero-enat-def[symmetric]) }
    note base = this
    moreover {
      fix i ad al v
      assume enat i < llength obs' lnth obs' i = NormalAction (ReadMem ad al v)
        and non-speculative P (w-value P vs ob) (ltake (enat i) obs')
      with LCons nth[of Suc i ad al v] base
      have v ∈ w-values P (w-value P vs ob) (list-of (ltake (enat i) obs')) (ad, al)
        by(clarsimp simp add: eSuc-enat[symmetric] split: obs-event.split action.split) }
    ultimately have ?LCons using LCons by(simp split: action.split obs-event.split)
    thus ?thesis ..
  qed
qed

locale executions-sc-hb =
  executions-base E P
  for E :: ('addr, 'thread-id) execution set
  and P :: 'm prog +
  assumes E-new-actions-for-fun:
  ⟦ E ∈ E; a ∈ new-actions-for P E adal; a' ∈ new-actions-for P E adal ⟧ ⟹ a = a'
  and E-ex-new-action:
  ⟦ E ∈ E; ra ∈ read-actions E; adal ∈ action-loc P E ra; non-speculative P (λ-. {}) (ltake (enat ra)
(lmap snd E)) ⟧
  ⟹ ∃ wa. wa ∈ new-actions-for P E adal ∧ wa < ra
begin

lemma E-new-same-addr-singleton:
  assumes E: E ∈ E
  shows ∃ a. new-actions-for P E adal ⊆ {a}
by(blast dest: E-new-actions-for-fun[OF E])

lemma new-action-before-read:
  assumes E: E ∈ E
  and ra: ra ∈ read-actions E
  and adal: adal ∈ action-loc P E ra
  and new: wa ∈ new-actions-for P E adal
  and sc: non-speculative P (λ-. {}) (ltake (enat ra) (lmap snd E))
  shows wa < ra
using E-new-same-addr-singleton[OF E, of adal] E-ex-new-action[OF E ra adal sc] new
by auto

lemma most-recent-write-exists:
  assumes E: E ∈ E
  and ra: ra ∈ read-actions E
  and sc: non-speculative P (λ-. {}) (ltake (enat ra) (lmap snd E))
  shows ∃ wa. P,E ⊢ ra ↝mrw wa
proof −
  from ra obtain ad al where
    adal: (ad, al) ∈ action-loc P E ra
    by(rule read-action-action-locE)

  define Q where Q = {a. a ∈ write-actions E ∧ (ad, al) ∈ action-loc P E a ∧ E ⊢ a ≤a ra}
```

```
  let ?A = new-actions-for P E (ad, al)
  let ?B = {a. a ∈ actions E ∧ (∃ v'. action-obs E a = NormalAction (WriteMem ad al v')) ∧ a ≤
ra}

  have Q ⊆ ?A ∪ ?B unfolding Q-def
    by(auto elim!: write-actions.cases action-loc-aux-cases simp add: new-actions-for-def elim: ac-
tion-orderE)
  moreover from E-new-same-addr-singleton[OF E, of (ad, al)]
  have finite ?A by(blast intro: finite-subset)
  moreover have finite ?B by auto
  ultimately have finQ: finite Q
    by(blast intro: finite-subset)

  from E-ex-new-action[OF E ra adal sc] ra obtain wa
    where wa: wa ∈ Q unfolding Q-def
    by(fastforce elim!: new-actionsE is-new-action.cases read-actions.cases intro: write-actionsI ac-
tion-orderI)

  define wa' where wa' = Max-torder (action-order E) Q

  from wa have Q ≠ {} Q ⊆ actions E by(auto simp add: Q-def)
  with finQ have wa' ∈ Q unfolding wa'-def
    by(rule Max-torder-in-set[OF torder-action-order])
  hence E ⊢ wa' ≤a ra wa' ∈ write-actions E
    and (ad, al) ∈ action-loc P E wa' by(simp-all add: Q-def)
  with ra adal have P,E ⊢ ra ↝mrw wa'
  proof
    fix wa''
    assume wa'': wa'' ∈ write-actions E (ad, al) ∈ action-loc P E wa''
    from ‹wa'' ∈ write-actions E› ra
    have ra ≠ wa'' by(auto dest: read-actions-not-write-actions)
    show E ⊢ wa'' ≤a wa' ∨ E ⊢ ra ≤a wa''
    proof(rule disjCI)
      assume ¬ E ⊢ ra ≤a wa''
      with total-onPD[OF total-action-order, of ra E wa'']
        ‹ra ≠ wa''› ‹ra ∈ read-actions E› ‹wa'' ∈ write-actions E›
      have E ⊢ wa'' ≤a ra by simp
      with wa'' have wa'' ∈ Q by(simp add: Q-def)
      with finQ show E ⊢ wa'' ≤a wa'
        using ‹Q ⊆ actions E› unfolding wa'-def
        by(rule Max-torder-above[OF torder-action-order])
    qed
  qed
  thus ?thesis ..
qed

lemma mrw-before:
  assumes E: E ∈ E
  and mrw: P,E ⊢ r ↝mrw w
  and sc: non-speculative P (λ-. {}) (ltake (enat r) (lmap snd E))
  shows w < r
using mrw read-actions-not-write-actions[of r E]
apply cases
apply(erule action-orderE)
```

apply(*erule* (*1*) *new-action-before-read*[*OF E*])
apply(*simp add*: *new-actions-for-def*)
apply(*rule sc*)
apply(*cases* $w = r$)
apply *auto*
done

lemma *sequentially-consistent-most-recent-write-for*:
assumes E: $E \in \mathcal{E}$
and *sc*: *non-speculative P* (λ-. {}) (*lmap snd E*)
shows *sequentially-consistent P* (*E*, λ*r*. *THE w*. *P*,*E* ⊢ *r* ⇝*mrw w*)
proof(*rule sequentially-consistentI*)
fix *r*
assume *r*: *r* ∈ *read-actions E*
from *sc* **have** *sc'*: *non-speculative P* (λ-. {}) (*ltake* (*enat r*) (*lmap snd E*))
by(*rule non-speculative-ltake*)
from *most-recent-write-exists*[*OF E r this*]
obtain *w* **where** *P*,*E* ⊢ *r* ⇝*mrw w* **..**
thus *P*,*E* ⊢ *r* ⇝*mrw THE w*. *P*,*E* ⊢ *r* ⇝*mrw w*
by(*simp add*: *THE-most-recent-writeI*)
qed

end

locale *jmm-multithreaded* = *multithreaded-base* +
constrains *final* :: *'x* ⇒ *bool*
and *r* :: (*'l*, *'thread-id*, *'x*, *'m*, *'w*, (*'addr*, *'thread-id*) *obs-event action*) *semantics*
and *convert-RA* :: *'l released-locks* ⇒ (*'addr*, *'thread-id*) *obs-event action list*
fixes *P* :: *'md prog*

end

8.8 Sequentially consistent completion of executions in the JMM

theory *SC-Completion*
imports
Non-Speculative
begin

8.8.1 Most recently written values

fun *mrw-value* ::
'm prog ⇒ ((*'addr* × *addr-loc*) ⇀ (*'addr val* × *bool*)) ⇒ (*'addr*, *'thread-id*) *obs-event action*
⇒ ((*'addr* × *addr-loc*) ⇀ (*'addr val* × *bool*))
where
mrw-value P vs (*NormalAction* (*WriteMem ad al v*)) = *vs*((*ad*, *al*) ↦ (*v*, *True*))
| *mrw-value P vs* (*NormalAction* (*NewHeapElem ad hT*)) =
(λ(*ad'*, *al*). *if ad* = *ad'* ∧ *al* ∈ *addr-locs P hT* ∧ (*case vs* (*ad*, *al*) *of None* ⇒ *True* | *Some* (*v*, *b*)
⇒ ¬ *b*)
then Some (*addr-loc-default P hT al*, *False*)
else vs (*ad'*, *al*))
| *mrw-value P vs* - = *vs*

lemma *mrw-value-cases*:
obtains *ad al v* **where** $x = NormalAction\ (WriteMem\ ad\ al\ v)$
| *ad hT* **where** $x = NormalAction\ (NewHeapElem\ ad\ hT)$
| *ad M vs v* **where** $x = NormalAction\ (ExternalCall\ ad\ M\ vs\ v)$
| *ad al v* **where** $x = NormalAction\ (ReadMem\ ad\ al\ v)$
| *t* **where** $x = NormalAction\ (ThreadStart\ t)$
| *t* **where** $x = NormalAction\ (ThreadJoin\ t)$
| *ad* **where** $x = NormalAction\ (SyncLock\ ad)$
| *ad* **where** $x = NormalAction\ (SyncUnlock\ ad)$
| *t* **where** $x = NormalAction\ (ObsInterrupt\ t)$
| *t* **where** $x = NormalAction\ (ObsInterrupted\ t)$
| $x = InitialThreadAction$
| $x = ThreadFinishAction$
by *pat-completeness*

abbreviation *mrw-values* ::
$'m\ prog \Rightarrow (('addr \times addr\text{-}loc) \rightharpoonup ('addr\ val \times bool)) \Rightarrow ('addr, 'thread\text{-}id)\ obs\text{-}event\ action\ list$
$\Rightarrow (('addr \times addr\text{-}loc) \rightharpoonup ('addr\ val \times bool))$
where $mrw\text{-}values\ P \equiv foldl\ (mrw\text{-}value\ P)$

lemma *mrw-values-eq-SomeD*:
assumes *mrw*: $mrw\text{-}values\ P\ vs0\ obs\ (ad, al) = \lfloor(v, b)\rfloor$
and $vs0\ (ad, al) = \lfloor(v, b)\rfloor \Longrightarrow \exists wa.\ wa \in set\ obs \wedge is\text{-}write\text{-}action\ wa \wedge (ad, al) \in action\text{-}loc\text{-}aux\ P\ wa \wedge (b \longrightarrow \neg\ is\text{-}new\text{-}action\ wa)$
shows $\exists obs'\ wa\ obs''.\ obs = obs' @ wa \# obs'' \wedge is\text{-}write\text{-}action\ wa \wedge (ad, al) \in action\text{-}loc\text{-}aux\ P\ wa \wedge$
$value\text{-}written\text{-}aux\ P\ wa\ al = v \wedge (is\text{-}new\text{-}action\ wa \longleftrightarrow \neg\ b) \wedge$
$(\forall ob{\in}set\ obs''.\ is\text{-}write\text{-}action\ ob \longrightarrow (ad, al) \in action\text{-}loc\text{-}aux\ P\ ob \longrightarrow is\text{-}new\text{-}action\ ob \wedge b)$
(**is** *?concl obs*)
using *assms*
proof(*induct obs rule*: *rev-induct*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*snoc ob obs*)
note *mrw* = ‹$mrw\text{-}values\ P\ vs0\ (obs @ [ob])\ (ad, al) = \lfloor(v, b)\rfloor$›
show *?case*
proof(*cases is-write-action ob* $\wedge$ $(ad, al) \in action\text{-}loc\text{-}aux\ P\ ob \wedge (is\text{-}new\text{-}action\ ob \longrightarrow \neg\ b)$)
case *True* **thus** *?thesis* **using** *mrw*
by(*fastforce elim*!: *is-write-action.cases intro*: *action-loc-aux-intros split*: *if-split-asm*)
next
case *False*
with *mrw* **have** $mrw\text{-}values\ P\ vs0\ obs\ (ad, al) = \lfloor(v, b)\rfloor$
by(*cases ob rule*: *mrw-value-cases*)(*auto split*: *if-split-asm simp add*: *addr-locs-def split*: *htype.split-asm*)
moreover
{ **assume** $vs0\ (ad, al) = \lfloor(v, b)\rfloor$
hence $\exists wa.\ wa \in set\ (obs @ [ob]) \wedge is\text{-}write\text{-}action\ wa \wedge (ad, al) \in action\text{-}loc\text{-}aux\ P\ wa \wedge (b \longrightarrow \neg\ is\text{-}new\text{-}action\ wa)$
by(*rule snoc*)
with *False* **have** $\exists wa.\ wa \in set\ obs \wedge is\text{-}write\text{-}action\ wa \wedge (ad, al) \in action\text{-}loc\text{-}aux\ P\ wa \wedge (b \longrightarrow \neg\ is\text{-}new\text{-}action\ wa)$
by *auto* }
ultimately have *?concl obs* **by**(*rule snoc*)

```
    thus ?thesis using False mrw by fastforce
  qed
qed

lemma mrw-values-WriteMemD:
  assumes NormalAction (WriteMem ad al v') ∈ set obs
  shows ∃ v. mrw-values P vs0 obs (ad, al) = Some (v, True)
using assms
apply(induct obs rule: rev-induct)
 apply simp
apply clarsimp
apply(erule disjE)
 apply clarsimp
apply clarsimp
apply(case-tac x rule: mrw-value-cases)
apply simp-all
done

lemma mrw-values-new-actionD:
  assumes w ∈ set obs is-new-action w adal ∈ action-loc-aux P w
  shows ∃ v b. mrw-values P vs0 obs adal = Some (v, b)
using assms
apply(induct obs rule: rev-induct)
 apply simp
apply clarsimp
apply(erule disjE)
 apply(fastforce simp add: split-beta elim!: action-loc-aux-cases is-new-action.cases)
apply clarsimp
apply(rename-tac w' obs' v b)
apply(case-tac w' rule: mrw-value-cases)
apply(auto simp add: split-beta)
done

lemma mrw-value-dom-mono:
  dom vs ⊆ dom (mrw-value P vs ob)
by(cases ob rule: mrw-value-cases) auto

lemma mrw-values-dom-mono:
  dom vs ⊆ dom (mrw-values P vs obs)
by(induct obs arbitrary: vs)(auto intro: subset-trans[OF mrw-value-dom-mono] del: subsetI)

lemma mrw-values-eq-NoneD:
  assumes mrw-values P vs0 obs adal = None
  and w ∈ set obs and is-write-action w and adal ∈ action-loc-aux P w
  shows False
using assms
apply −
apply(erule is-write-action.cases)
apply(fastforce dest: mrw-values-WriteMemD[where ?vs0.0=vs0 and P=P] mrw-values-new-actionD[where
?vs0.0=vs0] elim: action-loc-aux-cases)+
done

lemma mrw-values-mrw:
  assumes mrw: mrw-values P vs0 (map snd obs) (ad, al) = ⌊(v, b)⌋
```

and *initial*: *vs0* (*ad*, *al*) = ⌊(*v*, *b*)⌋ ⟹ ∃ *wa*. *wa* ∈ *set* (*map snd obs*) ∧ *is-write-action wa* ∧ (*ad*, *al*) ∈ *action-loc-aux P wa* ∧ (*b* ⟶ ¬ *is-new-action wa*)
shows ∃ *i*. *i* < *length obs* ∧ *P*,*llist-of* (*obs* @ [(*t*, *NormalAction* (*ReadMem ad al v*))]) ⊢ *length obs* ⇝*mrw i* ∧ *value-written P* (*llist-of obs*) *i* (*ad*, *al*) = *v*
proof −
from *mrw-values-eq-SomeD*[*OF mrw initial*]
obtain *obs′ wa obs″* **where** *obs*: *map snd obs* = *obs′* @ *wa* # *obs″*
and *wa*: *is-write-action wa*
and *adal*: (*ad*, *al*) ∈ *action-loc-aux P wa*
and *written*: *value-written-aux P wa al* = *v*
and *new*: *is-new-action wa* ⟷ ¬ *b*
and *last*: ⋀*ob*. ⟦ *ob* ∈ *set obs″*; *is-write-action ob*; (*ad*, *al*) ∈ *action-loc-aux P ob* ⟧ ⟹ *is-new-action ob* ∧ *b*
by *blast*
let *?i* = *length obs′*
let *?E* = *llist-of* (*obs* @ [(*t*, *NormalAction* (*ReadMem ad al v*))])

from *obs* **have** *len*: *length* (*map snd obs*) = *Suc* (*length obs′*) + *length obs″* **by** *simp*
hence *?i* < *length obs* **by** *simp*
moreover
hence *obs-i*: *action-obs ?E ?i* = *wa* **using** *len obs*
by(*auto simp add*: *action-obs-def map-eq-append-conv*)

have *P*,*?E* ⊢ *length obs* ⇝*mrw ?i*
proof(*rule most-recent-write-for.intros*)
show *length obs* ∈ *read-actions ?E*
by(*auto intro*: *read-actions.intros simp add*: *actions-def action-obs-def*)
show (*ad*, *al*) ∈ *action-loc P ?E* (*length obs*)
by(*simp add*: *action-obs-def lnth-llist-of*)
show *?E* ⊢ *length obs′* ≤*a length obs* **using** *len*
by−(*rule action-orderI*, *auto simp add*: *actions-def action-obs-def nth-append*)
show *?i* ∈ *write-actions ?E* **using** *len obs wa*
by−(*rule write-actions.intros*, *auto simp add*: *actions-def action-obs-def nth-append map-eq-append-conv*)
show (*ad*, *al*) ∈ *action-loc P ?E ?i* **using** *obs-i adal* **by** *simp*

fix *wa′*
assume *wa′*: *wa′* ∈ *write-actions ?E*
and *adal′*: (*ad*, *al*) ∈ *action-loc P ?E wa′*
from *wa′* ‹*?i* ∈ *write-actions ?E*›
have *wa′* ∈ *actions ?E ?i* ∈ *actions ?E* **by** *simp-all*
hence *?E* ⊢ *wa′* ≤*a ?i*
proof(*rule action-orderI*)
assume *new-wa′*: *is-new-action* (*action-obs ?E wa′*)
and *new-i*: *is-new-action* (*action-obs ?E ?i*)
from *new-i obs-i new* **have** *b*: ¬ *b* **by** *simp*

show *wa′* ≤ *?i*
proof(*rule ccontr*)
assume ¬ *?thesis*
hence *?i* < *wa′* **by** *simp*
hence *snd* (*obs* ! *wa′*) ∈ *set obs″* **using** *obs wa′* **unfolding** *in-set-conv-nth*
by −(*rule exI*[**where** *x*=*wa′* − *Suc* (*length obs′*)], *auto elim*!: *write-actions.cases actionsE simp add*: *action-obs-def lnth-llist-of actions-def nth-append map-eq-append-conv nth-Cons′ split*: *if-split-asm*)

```
      moreover from wa' have is-write-action (snd (obs ! wa'))
        by cases(auto simp add: action-obs-def nth-append actions-def split: if-split-asm)
      moreover from adal' wa' have (ad, al) ∈ action-loc-aux P (snd (obs ! wa'))
         by(auto simp add: action-obs-def nth-append nth-Cons' actions-def split: if-split-asm elim!:
write-actions.cases)
      ultimately show False using last[of snd (obs ! wa')] b by simp
    qed
  next
    assume new-wa': ¬ is-new-action (action-obs ?E wa')
    with wa' adal' obtain v' where NormalAction (WriteMem ad al v') ∈ set (map snd obs)
      unfolding in-set-conv-nth
     by (fastforce elim!: write-actions.cases is-write-action.cases simp add: action-obs-def actions-def
nth-append split: if-split-asm intro!: exI[where x=wa'])
    from mrw-values-WriteMemD[OF this, of P vs0] mrw have b by simp
    with new obs-i have ¬ is-new-action (action-obs ?E ?i) by simp
    moreover
    have wa' ≤ ?i
    proof(rule ccontr)
      assume ¬ ?thesis
      hence ?i < wa' by simp
      hence snd (obs ! wa') ∈ set obs'' using obs wa' unfolding in-set-conv-nth
          by −(rule exI[where x=wa' − Suc (length obs')], auto elim!: write-actions.cases ac-
tionsE simp add: action-obs-def lnth-llist-of actions-def nth-append map-eq-append-conv nth-Cons'
split: if-split-asm)
      moreover from wa' have is-write-action (snd (obs ! wa'))
        by cases(auto simp add: action-obs-def nth-append actions-def split: if-split-asm)
      moreover from adal' wa' have (ad, al) ∈ action-loc-aux P (snd (obs ! wa'))
         by(auto simp add: action-obs-def nth-append nth-Cons' actions-def split: if-split-asm elim!:
write-actions.cases)
      ultimately have is-new-action (snd (obs ! wa')) using last[of snd (obs ! wa')] by simp
      moreover from new-wa' wa' have ¬ is-new-action (snd (obs ! wa'))
          by(auto elim!: write-actions.cases simp add: action-obs-def nth-append actions-def split:
if-split-asm)
      ultimately show False by contradiction
    qed
    ultimately
    show ¬ is-new-action (action-obs ?E ?i) ∧ wa' ≤ ?i by blast
  qed
  thus ?E ⊢ wa' ≤a ?i ∨ ?E ⊢ length obs ≤a wa' ..
qed
  moreover from written ‹?i < length obs› obs-i
  have value-written P (llist-of obs) ?i (ad, al) = v
    by(simp add: value-written-def action-obs-def nth-append)
  ultimately show ?thesis by blast
qed

lemma mrw-values-no-write-unchanged:
  assumes no-write: ⋀w. ⟦ w ∈ set obs; is-write-action w; adal ∈ action-loc-aux P w ⟧
  ⟹ case vs adal of None ⇒ False | Some (v, b) ⇒ b ∧ is-new-action w
  shows mrw-values P vs obs adal = vs adal
using assms
proof(induct obs arbitrary: vs)
  case Nil show ?case by simp
next
```

case (*Cons ob obs*)
from *Cons.prems*[*of ob*]
have *mrw-value P vs ob adal* = *vs adal*
apply(*cases adal*)
apply(*cases ob rule*: *mrw-value-cases*, *fastforce*+)
apply(*auto simp add*: *addr-locs-def split*: *htype.split-asm*)
apply *blast*+
done
moreover
have *mrw-values P* (*mrw-value P vs ob*) *obs adal* = *mrw-value P vs ob adal*
proof(*rule Cons.hyps*)
fix *w*
assume *w* ∈ *set obs is-write-action w adal* ∈ *action-loc-aux P w*
with *Cons.prems*[*of w*] ‹*mrw-value P vs ob adal* = *vs adal*›
show *case mrw-value P vs ob adal of None* ⇒ *False* | ⌊(*v*, *b*)⌋ ⇒ *b* ∧ *is-new-action w* **by** *simp*
qed
ultimately show *?case* **by** *simp*
qed

8.8.2 Coinductive version of sequentially consistent prefixes

coinductive *ta-seq-consist* ::
'm prog ⇒ (*'addr* × *addr-loc* ⇀ *'addr val* × *bool*) ⇒ (*'addr*, *'thread-id*) *obs-event action llist* ⇒ *bool*
for *P* :: *'m prog*
where
LNil: *ta-seq-consist P vs LNil*
| *LCons*:
⟦ *case ob of NormalAction* (*ReadMem ad al v*) ⇒ ∃ *b*. *vs* (*ad*, *al*) = ⌊(*v*, *b*)⌋ | - ⇒ *True*;
ta-seq-consist P (*mrw-value P vs ob*) *obs* ⟧
⟹ *ta-seq-consist P vs* (*LCons ob obs*)

inductive-simps *ta-seq-consist-simps* [*simp*]:
ta-seq-consist P vs LNil
ta-seq-consist P vs (*LCons ob obs*)

lemma *ta-seq-consist-lappend*:
assumes *lfinite obs*
shows *ta-seq-consist P vs* (*lappend obs obs'*) ⟷
ta-seq-consist P vs obs ∧ *ta-seq-consist P* (*mrw-values P vs* (*list-of obs*)) *obs'*
(**is** *?concl vs obs*)
using *assms*
proof(*induct arbitrary*: *vs*)
case *lfinite-LNil* **thus** *?case* **by** *simp*
next
case (*lfinite-LConsI obs ob*)
have *?concl* (*mrw-value P vs ob*) *obs* **by** *fact*
thus *?case* **using** ‹*lfinite obs*› **by**(*simp split*: *action.split add*: *list-of-LCons*)
qed

lemma
assumes *ta-seq-consist P vs obs*
shows *ta-seq-consist-ltake*: *ta-seq-consist P vs* (*ltake n obs*) (**is** *?thesis1*)
and *ta-seq-consist-ldrop*: *ta-seq-consist P* (*mrw-values P vs* (*list-of* (*ltake n obs*))) (*ldrop n obs*) (**is** *?thesis2*)

proof −
note *assms*
also have *obs* = *lappend* (*ltake n obs*) (*ldrop n obs*) **by**(*simp add*: *lappend-ltake-ldrop*)
finally have *?thesis1* ∧ *?thesis2*
by(*cases n*)(*simp-all add*: *ta-seq-consist-lappend del*: *lappend-ltake-enat-ldropn*)
thus *?thesis1 ?thesis2* **by** *blast*+
qed

lemma *ta-seq-consist-coinduct-append* [*consumes 1*, *case-names ta-seq-consist*, *case-conclusion ta-seq-consist LNil lappend*]:
assumes *major*: *X vs obs*
and *step*: ⋀*vs obs*. *X vs obs*
⟹ *obs* = *LNil* ∨
(∃ *obs′ obs″*. *obs* = *lappend obs′ obs″* ∧ *obs′* ≠ *LNil* ∧ *ta-seq-consist P vs obs′* ∧
(*lfinite obs′* ⟶ (*X* (*mrw-values P vs* (*list-of obs′*)) *obs″* ∨
ta-seq-consist P (*mrw-values P vs* (*list-of obs′*)) *obs″*)))
(**is** ⋀*vs obs*. - ⟹ - ∨ *?step vs obs*)
shows *ta-seq-consist P vs obs*
proof −
from *major*
have ∃ *obs′ obs″*. *obs* = *lappend* (*llist-of obs′*) *obs″* ∧ *ta-seq-consist P vs* (*llist-of obs′*) ∧
X (*mrw-values P vs obs′*) *obs″*
by(*auto intro*: *exI*[**where** *x*=[]])
thus *?thesis*
proof(*coinduct*)
case (*ta-seq-consist vs obs*)
then obtain *obs′ obs″*
where *obs*: *obs* = *lappend* (*llist-of obs′*) *obs″*
and *sc-obs′*: *ta-seq-consist P vs* (*llist-of obs′*)
and *X*: *X* (*mrw-values P vs obs′*) *obs″* **by** *blast*

show *?case*
proof(*cases obs′*)
case *Nil*
with *X* **have** *X vs obs″* **by** *simp*
from *step*[*OF this*] **show** *?thesis*
proof
assume *obs″* = *LNil*
with *Nil obs* **show** *?thesis* **by** *simp*
next
assume *?step vs obs″*
then obtain *obs‴ obs⁗*
where *obs″*: *obs″* = *lappend obs‴ obs⁗* **and** *obs‴* ≠ *LNil*
and *sc-obs‴*: *ta-seq-consist P vs obs‴*
and *fin*: *lfinite obs‴* ⟹ *X* (*mrw-values P vs* (*list-of obs‴*)) *obs⁗* ∨
ta-seq-consist P (*mrw-values P vs* (*list-of obs‴*)) *obs⁗*
by *blast*
from ‹*obs‴* ≠ *LNil*› **obtain** *ob obs⁗⁗* **where** *obs‴*: *obs‴* = *LCons ob obs⁗⁗*
unfolding *neq-LNil-conv* **by** *blast*
with *Nil obs″ obs* **have** *concl1*: *obs* = *LCons ob* (*lappend obs⁗⁗ obs⁗*) **by** *simp*
have *concl2*: *case ob of NormalAction* (*ReadMem ad al v*) ⇒ ∃ *b*. *vs* (*ad*, *al*) = ⌊(*v*, *b*)⌋ | - ⇒
True
using *sc-obs‴ obs‴* **by** *simp*

```
      show ?thesis
      proof(cases lfinite obs''')
        case False
        hence lappend obs'''''' obs'''' = obs'''''' using obs''' by(simp add: lappend-inf)
        hence ta-seq-consist P (mrw-value P vs ob) (lappend obs'''''' obs'''')
          using sc-obs''' obs''' by simp
        with concl1 concl2 have ?LCons by blast
        thus ?thesis by simp
      next
        case True
        with obs''' obtain obs'''''' where obs'''''': obs'''''' = llist-of obs''''''
          by simp(auto simp add: lfinite-eq-range-llist-of)
        from fin[OF True] have ?LCons
        proof
          assume X: X (mrw-values P vs (list-of obs''')) obs''''
          hence X (mrw-values P (mrw-value P vs ob) obs'''''') obs''''
            using obs'''''' obs''' by simp
          moreover from obs''''''
          have lappend obs'''''' obs'''' = lappend (llist-of obs'''''') obs'''' by simp
          moreover have ta-seq-consist P (mrw-value P vs ob) (llist-of obs'''''')
            using sc-obs''' obs''' obs'''''' by simp
          ultimately show ?thesis using concl1 concl2 by blast
        next
          assume ta-seq-consist P (mrw-values P vs (list-of obs''')) obs''''
          with sc-obs''' obs'''''' obs'''
          have ta-seq-consist P (mrw-value P vs ob) (lappend obs'''''' obs'''')
            by(simp add: ta-seq-consist-lappend)
          with concl1 concl2 show ?thesis by blast
        qed
        thus ?thesis by simp
      qed
    qed
  next
    case (Cons ob obs''')
    hence obs = LCons ob (lappend (llist-of obs''') obs'')
      using obs by simp
    moreover from sc-obs' Cons
    have case ob of NormalAction (ReadMem ad al v) ⇒ ∃ b. vs (ad, al) = ⌊(v, b)⌋ | - ⇒ True
      and ta-seq-consist P (mrw-value P vs ob) (llist-of obs''') by simp-all
    moreover from X Cons have X (mrw-values P (mrw-value P vs ob) obs''') obs'' by simp
    ultimately show ?thesis by blast
  qed
 qed
qed

lemma ta-seq-consist-coinduct-append-wf
  [consumes 2, case-names ta-seq-consist, case-conclusion ta-seq-consist LNil lappend]:
  assumes major: X vs obs a
  and wf: wf R
  and step: ⋀vs obs a. X vs obs a
    ⟹ obs = LNil ∨
      (∃ obs' obs'' a'. obs = lappend obs' obs'' ∧ ta-seq-consist P vs obs' ∧ (obs' = LNil ⟶ (a', a) ∈
R) ∧
                    (lfinite obs' ⟶ X (mrw-values P vs (list-of obs')) obs'' a' ∨
```

```
                                    ta-seq-consist P (mrw-values P vs (list-of obs')) obs''))
    (is ⋀vs obs a. - ⟹ - ∨ ?step vs obs a)
  shows ta-seq-consist P vs obs
proof −
  { fix vs obs a
    assume X vs obs a
    with wf
    have obs = LNil ∨ (∃ obs' obs''. obs = lappend obs' obs'' ∧ obs' ≠ LNil ∧ ta-seq-consist P vs obs'
∧
          (lfinite obs' ⟶ (∃ a. X (mrw-values P vs (list-of obs')) obs'' a) ∨
                          ta-seq-consist P (mrw-values P vs (list-of obs')) obs''))
      (is - ∨ ?step-concl vs obs)
    proof(induct a arbitrary: vs obs rule: wf-induct[consumes 1, case-names wf])
      case (wf a)
      note IH = wf.hyps[rule-format]
      from step[OF ‹X vs obs a›]
      show ?case
      proof
        assume obs = LNil thus ?thesis ..
      next
        assume ?step vs obs a
        then obtain obs' obs'' a'
          where obs: obs = lappend obs' obs''
          and sc-obs': ta-seq-consist P vs obs'
          and decr: obs' = LNil ⟹ (a', a) ∈ R
          and fin: lfinite obs' ⟹
                X (mrw-values P vs (list-of obs')) obs'' a' ∨
                ta-seq-consist P (mrw-values P vs (list-of obs')) obs''
          by blast
        show ?case
        proof(cases obs' = LNil)
          case True
          hence lfinite obs' by simp
          from fin[OF this] show ?thesis
          proof
            assume X: X (mrw-values P vs (list-of obs')) obs'' a'
            from True have (a', a) ∈ R by(rule decr)
            from IH[OF this X] show ?thesis
            proof
              assume obs'' = LNil
              with True obs have obs = LNil by simp
              thus ?thesis ..
            next
              assume ?step-concl (mrw-values P vs (list-of obs')) obs''
              hence ?step-concl vs obs using True obs by simp
              thus ?thesis ..
            qed
          next
            assume ta-seq-consist P (mrw-values P vs (list-of obs')) obs''
            thus ?thesis using obs True
             by cases(auto cong: action.case-cong obs-event.case-cong intro: exI[where x=LCons x LNil
for x])
          qed
        next
```

```
        case False
        with obs sc-obs' fin show ?thesis by auto
      qed
    qed
  qed }
 note step' = this

 from major show ?thesis
 proof(coinduction arbitrary: vs obs a rule: ta-seq-consist-coinduct-append)
  case (ta-seq-consist vs obs a)
  thus ?case by simp(rule step')
 qed
qed

lemma ta-seq-consist-nthI:
 (⋀i ad al v. ⟦ enat i < llength obs; lnth obs i = NormalAction (ReadMem ad al v);
    ta-seq-consist P vs (ltake (enat i) obs) ⟧
  ⟹ ∃ b. mrw-values P vs (list-of (ltake (enat i) obs)) (ad, al) = ⌊(v, b)⌋)
 ⟹ ta-seq-consist P vs obs
proof(coinduction arbitrary: vs obs)
 case (ta-seq-consist vs obs)
 hence nth:
  ⋀i ad al v. ⟦ enat i < llength obs; lnth obs i = NormalAction (ReadMem ad al v);
          ta-seq-consist P vs (ltake (enat i) obs) ⟧
  ⟹ ∃ b. mrw-values P vs (list-of (ltake (enat i) obs)) (ad, al) = ⌊(v, b)⌋ by blast
 show ?case
 proof(cases obs)
  case LNil thus ?thesis by simp
 next
  case (LCons ob obs')
  { fix ad al v
   assume ob = NormalAction (ReadMem ad al v)
   with nth[of 0 ad al v] LCons
   have ∃ b. vs (ad, al) = ⌊(v, b)⌋
    by(simp add: zero-enat-def[symmetric]) }
  note base = this
  moreover {
   fix i ad al v
   assume enat i < llength obs' lnth obs' i = NormalAction (ReadMem ad al v)
    and ta-seq-consist P (mrw-value P vs ob) (ltake (enat i) obs')
   with LCons nth[of Suc i ad al v] base
   have ∃ b. mrw-values P (mrw-value P vs ob) (list-of (ltake (enat i) obs')) (ad, al) = ⌊(v, b)⌋
    by(clarsimp simp add: eSuc-enat[symmetric] split: obs-event.split action.split) }
  ultimately have ?LCons using LCons by(simp split: action.split obs-event.split)
  thus ?thesis ..
 qed
qed

lemma ta-seq-consist-into-non-speculative:
 ⟦ ta-seq-consist P vs obs; ∀ adal. set-option (vs adal) ⊆ vs' adal × UNIV ⟧
 ⟹ non-speculative P vs' obs
proof(coinduction arbitrary: vs' obs vs)
 case (non-speculative vs' obs vs)
 thus ?case
```

```
    apply cases
    apply(auto split: action.split-asm obs-event.split-asm)
    apply(rule exI, erule conjI, auto)+
    done
qed

lemma llist-of-list-of-append:
  lfinite xs ⟹ llist-of (list-of xs @ ys) = lappend xs (llist-of ys)
unfolding lfinite-eq-range-llist-of by(clarsimp simp add: lappend-llist-of-llist-of)

lemma ta-seq-consist-most-recent-write-for:
  assumes sc: ta-seq-consist P Map.empty (lmap snd E)
  and read: r ∈ read-actions E
  and new-actions-for-fun: ⋀adal a a'. ⟦ a ∈ new-actions-for P E adal; a' ∈ new-actions-for P E adal
⟧ ⟹ a = a'
  shows ∃ i. P,E ⊢ r ↝mrw i ∧ i < r
proof −
  from read obtain t v ad al
    where nth-r: lnth E r = (t, NormalAction (ReadMem ad al v))
    and r: enat r < llength E
    by(cases)(cases lnth E r, auto simp add: action-obs-def actions-def)
  from nth-r r
  have E-unfold: E = lappend (ltake (enat r) E) (LCons (t, NormalAction (ReadMem ad al v)) (ldropn
(Suc r) E))
    by (metis lappend-ltake-enat-ldropn ldropn-Suc-conv-ldropn)
  from sc obtain b where sc': ta-seq-consist P Map.empty (ltake (enat r) (lmap snd E))
    and mrw': mrw-values P Map.empty (map snd (list-of (ltake (enat r) E))) (ad, al) = ⌊(v, b)⌋
    by(subst (asm) (3) E-unfold)(auto simp add: ta-seq-consist-lappend lmap-lappend-distrib)

  from mrw-values-mrw[OF mrw', of t] r
  obtain E' w'
    where E': E' = llist-of (list-of (ltake (enat r) E) @ [(t, NormalAction (ReadMem ad al v))])
    and v: v = value-written P (ltake (enat r) E) w' (ad, al)
    and mrw'': P,E' ⊢ r ↝mrw w'
    and w': w' < r by(fastforce simp add: length-list-of-conv-the-enat min-def split: if-split-asm)

  from E' r have sim: ltake (enat (Suc r)) E' [≈] ltake (enat (Suc r)) E
   by(subst E-unfold)(simp add: ltake-lappend llist-of-list-of-append min-def, auto simp add: eSuc-enat[symmetric]
zero-enat-def[symmetric] eq-into-sim-actions)
  from nth-r have adal-r: (ad, al) ∈ action-loc P E r by(simp add: action-obs-def)
  from E' r have nth-r': lnth E' r = (t, NormalAction (ReadMem ad al v))
    by(auto simp add: nth-append length-list-of-conv-the-enat min-def)
  with mrw'' w' r adal-r obtain E ⊢ w' ≤a r w' ∈ write-actions E (ad, al) ∈ action-loc P E w'
    by cases(fastforce simp add: action-obs-def action-loc-change-prefix[OF sim[symmetric], simplified
action-obs-def] intro: action-order-change-prefix[OF - sim] write-actions-change-prefix[OF - sim])

  with read adal-r have P,E ⊢ r ↝mrw w'
  proof(rule most-recent-write-for.intros)
    fix wa'
    assume write': wa' ∈ write-actions E
      and adal-wa': (ad, al) ∈ action-loc P E wa'
    show E ⊢ wa' ≤a w' ∨ E ⊢ r ≤a wa'
    proof(cases r ≤ wa')
      assume r ≤ wa'
```

```
      show ?thesis
      proof(cases is-new-action (action-obs E wa'))
        case False
        with ‹r ≤ wa'› have E ⊢ r ≤a wa' using read write'
          by(auto simp add: action-order-def elim!: read-actions.cases)
        thus ?thesis ..
      next
        case True
        with write' adal-wa' have wa' ∈ new-actions-for P E (ad, al)
          by(simp add: new-actions-for-def)
        hence w' ∉ new-actions-for P E (ad, al) using r w' ‹r ≤ wa'›
          by(auto dest: new-actions-for-fun)
        with ‹w' ∈ write-actions E› ‹(ad, al) ∈ action-loc P E w'›
        have ¬ is-new-action (action-obs E w') by(simp add: new-actions-for-def)
        with write' True ‹w' ∈ write-actions E› have E ⊢ wa' ≤a w' by(simp add: action-order-def)
        thus ?thesis ..
      qed
    next
      assume ¬ r ≤ wa'
      hence wa' < r by simp
      with write' adal-wa'
      have wa' ∈ write-actions E' (ad, al) ∈ action-loc P E' wa'
      by(auto intro: write-actions-change-prefix[OF - sim[symmetric]] simp add: action-loc-change-prefix[OF
sim])
      from most-recent-write-recent[OF mrw'' - this] nth-r'
      have E' ⊢ wa' ≤a w' ∨ E' ⊢ r ≤a wa' by(simp add: action-obs-def)
      thus ?thesis using ‹wa' < r› w'
        by(auto 4 3 del: disjCI intro: disjI1 disjI2 action-order-change-prefix[OF - sim])
    qed
  qed
  with w' show ?thesis by blast
qed

lemma ta-seq-consist-mrw-before:
  assumes sc: ta-seq-consist P Map.empty (lmap snd E)
  and new-actions-for-fun: ⋀adal a a'. ⟦ a ∈ new-actions-for P E adal; a' ∈ new-actions-for P E adal
⟧ ⟹ a = a'
  and mrw: P,E ⊢ r ↝mrw w
  shows w < r
proof −
  from mrw have r ∈ read-actions E by cases
  with sc new-actions-for-fun obtain w' where P,E ⊢ r ↝mrw w' w' < r
    by(auto dest: ta-seq-consist-most-recent-write-for)
  with mrw show ?thesis by(auto dest: most-recent-write-for-fun)
qed

lemma ta-seq-consist-imp-sequentially-consistent:
  assumes tsa-ok: thread-start-actions-ok E
  and new-actions-for-fun: ⋀adal a a'. ⟦ a ∈ new-actions-for P E adal; a' ∈ new-actions-for P E adal
⟧ ⟹ a = a'
  and seq: ta-seq-consist P Map.empty (lmap snd E)
  shows ∃ ws. sequentially-consistent P (E, ws) ∧ P ⊢ (E, ws) √
proof(intro exI conjI)
  define ws where ws i = (THE w. P,E ⊢ i ↝mrw w) for i
```

```
  from seq have ns: non-speculative P (λ-. {}) (lmap snd E)
    by(rule ta-seq-consist-into-non-speculative) simp
  show sequentially-consistent P (E, ws) unfolding ws-def
  proof(rule sequentially-consistentI)
    fix r
    assume r ∈ read-actions E
    with seq new-actions-for-fun
    obtain w where P,E ⊢ r ↝mrw w by(auto dest: ta-seq-consist-most-recent-write-for)
    thus P,E ⊢ r ↝mrw THE w. P,E ⊢ r ↝mrw w by(simp add: THE-most-recent-writeI)
  qed

  show P ⊢ (E, ws) √
  proof(rule wf-execI)
    show is-write-seen P E ws
    proof(rule is-write-seenI)
      fix a ad al v
      assume a: a ∈ read-actions E
        and adal: action-obs E a = NormalAction (ReadMem ad al v)
      from ns have seq': non-speculative P (λ-. {}) (ltake (enat a) (lmap snd E)) by(rule non-speculative-ltake)
      from seq a seq new-actions-for-fun
      obtain w where mrw: P,E ⊢ a ↝mrw w
        and w < a by(auto dest: ta-seq-consist-most-recent-write-for)
      hence w: ws a = w by(simp add: ws-def THE-most-recent-writeI)
      with mrw adal

      show ws a ∈ write-actions E
        and (ad, al) ∈ action-loc P E (ws a)
        and ¬ P,E ⊢ a ≤hb ws a
          by(fastforce elim!: most-recent-write-for.cases dest: happens-before-into-action-order anti-
symPD[OF antisym-action-order] read-actions-not-write-actions)+

      let ?between = ltake (enat (a − Suc w)) (ldropn (Suc w) E)
      let ?prefix = ltake (enat w) E
      let ?vs-prefix = mrw-values P Map.empty (map snd (list-of ?prefix))

      { fix v'
        assume new: is-new-action (action-obs E w)
          and vs': ?vs-prefix (ad, al) = ⌊(v', True)⌋
        from mrw-values-eq-SomeD[OF vs']
        obtain obs' wa obs'' where split: map snd (list-of ?prefix) = obs' @ wa # obs''
          and wa: is-write-action wa
          and adal': (ad, al) ∈ action-loc-aux P wa
          and new-wa: ¬ is-new-action wa by blast
        from split have length (map snd (list-of ?prefix)) = Suc (length obs' + length obs'') by simp
      hence len-prefix: llength ?prefix = enat ... by(simp add: length-list-of-conv-the-enat min-enat1-conv-enat)
        with split have nth (map snd (list-of ?prefix)) (length obs') = wa
          and enat (length obs') < llength ?prefix by simp-all
      hence snd (lnth ?prefix (length obs')) = wa by(simp add: list-of-lmap[symmetric] del: list-of-lmap)
        hence wa': action-obs E (length obs') = wa and enat (length obs') < llength E
          using ‹enat (length obs') < llength ?prefix› by(auto simp add: action-obs-def lnth-ltake)
          with wa have length obs' ∈ write-actions E by(auto intro: write-actions.intros simp add:
actions-def)
        from most-recent-write-recent[OF mrw - this, of (ad, al)] adal adal' wa'
        have E ⊢ length obs' ≤a w ∨ E ⊢ a ≤a length obs' by simp
```

hence *False* **using** *new-wa new wa′ adal len-prefix* ‹*w* < *a*›
by(*auto elim*!: *action-orderE simp add*: *min-enat1-conv-enat split*: *enat.split-asm*)
}
hence *mrw-value-w*: *mrw-value P ?vs-prefix* (*snd* (*lnth E w*)) (*ad*, *al*) =
⌊(*value-written P E w* (*ad*, *al*), ¬ *is-new-action* (*action-obs E w*))⌋
using ‹*ws a* ∈ *write-actions E*› ‹(*ad*, *al*) ∈ *action-loc P E* (*ws a*)› *w*
by(*cases snd* (*lnth E w*) *rule*: *mrw-value-cases*)(*fastforce elim*: *write-actions.cases simp add*: *value-written-def action-obs-def*)+
have *mrw-values P* (*mrw-value P ?vs-prefix* (*snd* (*lnth E w*))) (*list-of* (*lmap snd ?between*)) (*ad*, *al*) = ⌊(*value-written P E w* (*ad*, *al*), ¬ *is-new-action* (*action-obs E w*))⌋
proof(*subst mrw-values-no-write-unchanged*)
fix *wa*
assume *wa* ∈ *set* (*list-of* (*lmap snd ?between*))
and *write-wa*: *is-write-action wa*
and *adal-wa*: (*ad*, *al*) ∈ *action-loc-aux P wa*
hence *wa*: *wa* ∈ *lset* (*lmap snd ?between*) **by** *simp*
from *wa* **obtain** *i-wa* **where** *wa* = *lnth* (*lmap snd ?between*) *i-wa*
and *i-wa*: *enat i-wa* < *llength* (*lmap snd ?between*)
unfolding *lset-conv-lnth* **by** *blast*
moreover hence *i-wa-len*: *enat* (*Suc* (*w* + *i-wa*)) < *llength E* **by**(*cases llength E*) *auto*
ultimately have *wa′*: *wa* = *action-obs E* (*Suc* (*w* + *i-wa*))
by(*simp-all add*: *lnth-ltake action-obs-def ac-simps*)
with *write-wa i-wa-len* **have** *Suc* (*w* + *i-wa*) ∈ *write-actions E*
by(*auto intro*: *write-actions.intros simp add*: *actions-def*)
from *most-recent-write-recent*[*OF mrw - this*, *of* (*ad*, *al*)] *adal adal-wa wa′*
have *E* ⊢ *Suc* (*w* + *i-wa*) ≤*a w* ∨ *E* ⊢ *a* ≤*a Suc* (*w* + *i-wa*) **by**(*simp*)
hence *is-new-action wa* ∧ ¬ *is-new-action* (*action-obs E w*)
using *adal i-wa wa′* **by**(*auto elim*: *action-orderE*)
thus *case* (*mrw-value P ?vs-prefix* (*snd* (*lnth E w*)) (*ad*, *al*)) *of None* ⇒ *False* | *Some* (*v*, *b*) ⇒ *b* ∧ *is-new-action wa*
unfolding *mrw-value-w* **by** *simp*
qed(*simp add*: *mrw-value-w*)

moreover

from *a* **have** *a* ∈ *actions E* **by** *simp*
hence *enat a* < *llength E* **by**(*rule actionsE*)
with ‹*w* < *a*› **have** *enat* (*a* − *Suc w*) < *llength E* − *enat* (*Suc w*)
by(*cases llength E*) *simp-all*
hence *E* = *lappend* (*lappend ?prefix* (*LCons* (*lnth E w*) *?between*)) (*LCons* (*lnth* (*ldropn* (*Suc w*) *E*) (*a* − *Suc w*)) (*ldropn* (*Suc* (*a* − *Suc w*)) (*ldropn* (*Suc w*) *E*)))
using ‹*w* < *a*› ‹*enat a* < *llength E*› **unfolding** *lappend-assoc lappend-code*
apply(*subst ldropn-Suc-conv-ldropn*, *simp*)
apply(*subst lappend-ltake-enat-ldropn*)
apply(*subst ldropn-Suc-conv-ldropn*, *simp add*: *less-trans*[**where** *y*=*enat a*])
by *simp*
hence *E′*: *E* = *lappend* (*lappend ?prefix* (*LCons* (*lnth E w*) *?between*)) (*LCons* (*lnth E a*) (*ldropn* (*Suc a*) *E*))
using ‹*w* < *a*› ‹*enat a* < *llength E*› **by** *simp*

from *seq* **have** *ta-seq-consist P* (*mrw-values P Map.empty* (*list-of* (*lappend* (*lmap snd ?prefix*) (*LCons* (*snd* (*lnth E w*)) (*lmap snd ?between*))))) (*lmap snd* (*LCons* (*lnth E a*) (*ldropn* (*Suc a*) *E*)))
by(*subst* (*asm*) *E′*)(*simp add*: *lmap-lappend-distrib ta-seq-consist-lappend*)
ultimately show *value-written P E* (*ws a*) (*ad*, *al*) = *v* **using** *adal w*

by(*clarsimp simp add*: *action-obs-def list-of-lappend list-of-LCons*)

show ¬ *P*,*E* ⊢ *a* ≤*so ws a* **using** ‹*w* < *a*› *w adal* **by**(*auto elim*!: *action-orderE sync-orderE*)

fix *a*′
assume *a*′: *a*′ ∈ *write-actions E* (*ad*, *al*) ∈ *action-loc P E a*′

{
presume *E* ⊢ *ws a* ≤*a a*′ *E* ⊢ *a*′ ≤*a a*
with *mrw adal a*′ **have** *a*′ = *ws a* **unfolding** *w*
by *cases*(*fastforce dest*: *antisymPD*[*OF antisym-action-order*] *read-actions-not-write-actions elim*!: *meta-allE*[**where** *x*=*a*′])
thus *a*′ = *ws a a*′ = *ws a* **by** −
next
assume *P*,*E* ⊢ *ws a* ≤*hb a*′ *P*,*E* ⊢ *a*′ ≤*hb a*
thus *E* ⊢ *ws a* ≤*a a*′ *E* ⊢ *a*′ ≤*a a* **using** *a*′ **by**(*blast intro*: *happens-before-into-action-order*)+
next
assume *is-volatile P al P*,*E* ⊢ *ws a* ≤*so a*′ *P*,*E* ⊢ *a*′ ≤*so a*
thus *E* ⊢ *ws a* ≤*a a*′ *E* ⊢ *a*′ ≤*a a* **by**(*auto elim*: *sync-orderE*)
}
qed
qed(*rule tsa-ok*)
qed

8.8.3 Cut-and-update and sequentially consistent completion

inductive *foldl-list-all2* ::
(′*b* ⇒ ′*c* ⇒ ′*a* ⇒ ′*a*) ⇒ (′*b* ⇒ ′*c* ⇒ ′*a* ⇒ *bool*) ⇒ (′*b* ⇒ ′*c* ⇒ ′*a* ⇒ *bool*) ⇒ ′*b list* ⇒ ′*c list* ⇒ ′*a* ⇒ *bool*
for *f* **and** *P* **and** *Q*
where
foldl-list-all2 f P Q [] [] *s*
| ⟦ *Q x y s*; *P x y s* ⟹ *foldl-list-all2 f P Q xs ys* (*f x y s*) ⟧ ⟹ *foldl-list-all2 f P Q* (*x* # *xs*) (*y* # *ys*) *s*

inductive-simps *foldl-list-all2-simps* [*simp*]:
foldl-list-all2 f P Q [] *ys s*
foldl-list-all2 f P Q xs [] *s*
foldl-list-all2 f P Q (*x* # *xs*) (*y* # *ys*) *s*

inductive-simps *foldl-list-all2-Cons1*:
foldl-list-all2 f P Q (*x* # *xs*) *ys s*

inductive-simps *foldl-list-all2-Cons2*:
foldl-list-all2 f P Q xs (*y* # *ys*) *s*

definition *eq-upto-seq-inconsist* ::
′*m prog* ⇒ (′*addr*, ′*thread-id*) *obs-event action list* ⇒ (′*addr*, ′*thread-id*) *obs-event action list*
⇒ (′*addr* × *addr-loc* ⇀ ′*addr val* × *bool*) ⇒ *bool*
where
eq-upto-seq-inconsist P =
foldl-list-all2 (λ*ob ob*′ *vs*. *mrw-value P vs ob*)
(λ*ob ob*′ *vs*. *case ob of NormalAction* (*ReadMem ad al v*) ⇒ ∃ *b*. *vs* (*ad*, *al*) = *Some*

```
(v, b) | - ⇒ True)
                (λob ob' vs. if (case ob of NormalAction (ReadMem ad al v) ⇒ ∃ b. vs (ad, al) = Some
(v, b) | - ⇒ True) then ob = ob' else ob ≈ ob')

lemma eq-upto-seq-inconsist-simps:
  eq-upto-seq-inconsist P [] obs' vs ⟷ obs' = []
  eq-upto-seq-inconsist P obs [] vs ⟷ obs = []
  eq-upto-seq-inconsist P (ob # obs) (ob' # obs') vs ⟷
   (case ob of NormalAction (ReadMem ad al v) ⇒
      if (∃ b. vs (ad, al) = ⌊(v, b)⌋)
      then ob = ob' ∧ eq-upto-seq-inconsist P obs obs' (mrw-value P vs ob)
      else ob ≈ ob'
    | - ⇒ ob = ob' ∧ eq-upto-seq-inconsist P obs obs' (mrw-value P vs ob))
by(auto simp add: eq-upto-seq-inconsist-def split: action.split obs-event.split)

lemma eq-upto-seq-inconsist-Cons1:
  eq-upto-seq-inconsist P (ob # obs) obs' vs ⟷
   (∃ ob' obs''. obs' = ob' # obs'' ∧
      (case ob of NormalAction (ReadMem ad al v) ⇒
         if (∃ b. vs (ad, al) = ⌊(v, b)⌋)
         then ob' = ob ∧ eq-upto-seq-inconsist P obs obs'' (mrw-value P vs ob)
         else ob ≈ ob'
       | - ⇒ ob' = ob ∧ eq-upto-seq-inconsist P obs obs'' (mrw-value P vs ob)))
unfolding eq-upto-seq-inconsist-def
by(auto split: obs-event.split action.split simp add: foldl-list-all2-Cons1)

lemma eq-upto-seq-inconsist-appendD:
  assumes eq-upto-seq-inconsist P (obs @ obs') obs'' vs
  and ta-seq-consist P vs (llist-of obs)
  shows length obs ≤ length obs'' (is ?thesis1)
  and take (length obs) obs'' = obs (is ?thesis2)
  and eq-upto-seq-inconsist P obs' (drop (length obs) obs'') (mrw-values P vs obs) (is ?thesis3)
using assms
by(induct obs arbitrary: obs'' vs)(auto split: action.split-asm obs-event.split-asm simp add: eq-upto-seq-inconsist-Co

lemma ta-seq-consist-imp-eq-upto-seq-inconsist-refl:
  ta-seq-consist P vs (llist-of obs) ⟹ eq-upto-seq-inconsist P obs obs vs
apply(induct obs arbitrary: vs)
apply(auto simp add: eq-upto-seq-inconsist-simps split: action.split obs-event.split)
done

context notes split-paired-Ex [simp del] eq-upto-seq-inconsist-simps [simp] begin

lemma eq-upto-seq-inconsist-appendI:
  ⟦ eq-upto-seq-inconsist P obs OBS vs;
    ⟦ ta-seq-consist P vs (llist-of obs) ⟧ ⟹ eq-upto-seq-inconsist P obs' OBS' (mrw-values P vs OBS)
⟧
  ⟹ eq-upto-seq-inconsist P (obs @ obs') (OBS @ OBS') vs
apply(induct obs arbitrary: vs OBS)
 apply simp
apply(auto simp add: eq-upto-seq-inconsist-Cons1)
apply(simp split: action.split obs-event.split)
apply auto
done
```

lemma *eq-upto-seq-inconsist-trans*:
⟦ *eq-upto-seq-inconsist P obs obs' vs*; *eq-upto-seq-inconsist P obs' obs'' vs* ⟧
⟹ *eq-upto-seq-inconsist P obs obs'' vs*
apply(*induction obs arbitrary*: *obs' obs'' vs*)
apply(*clarsimp simp add*: *eq-upto-seq-inconsist-Cons1*)+
apply(*auto split*!: *action.split obs-event.split if-split-asm*)
done

lemma *eq-upto-seq-inconsist-append2*:
⟦ *eq-upto-seq-inconsist P obs obs' vs*; ¬ *ta-seq-consist P vs* (*llist-of obs*) ⟧
⟹ *eq-upto-seq-inconsist P obs* (*obs'* @ *obs''*) *vs*
apply(*induction obs arbitrary*: *obs' vs*)
apply(*clarsimp simp add*: *eq-upto-seq-inconsist-Cons1*)+
apply(*auto split*!: *action.split obs-event.split if-split-asm*)
done

end

context *executions-sc-hb* **begin**

lemma *ta-seq-consist-mrwI*:
assumes *E*: $E \in \mathcal{E}$
and *wf*: $P \vdash (E, ws)\ \surd$
and *mrw*: $\bigwedge a.$ ⟦ *enat a* < *r*; *a* ∈ *read-actions E* ⟧ ⟹ $P,E \vdash a \rightsquigarrow mrw\ ws\ a$
shows *ta-seq-consist P Map.empty* (*lmap snd* (*ltake r E*))
proof(*rule ta-seq-consist-nthI*)
fix *i ad al v*
assume *i-len*: *enat i* < *llength* (*lmap snd* (*ltake r E*))
and *E-i*: *lnth* (*lmap snd* (*ltake r E*)) *i* = *NormalAction* (*ReadMem ad al v*)
and *sc*: *ta-seq-consist P Map.empty* (*ltake* (*enat i*) (*lmap snd* (*ltake r E*)))
from *i-len* **have** *enat i* < *r* **by** *simp*
with *sc* **have** *ta-seq-consist P Map.empty* (*ltake* (*enat i*) (*lmap snd E*))
by(*simp add*: *min-def split*: *if-split-asm*)
hence *ns*: *non-speculative P* (λ-. {}) (*ltake* (*enat i*) (*lmap snd E*))
by(*rule ta-seq-consist-into-non-speculative*) *simp*
from *i-len* **have** *i* ∈ *actions E* **by**(*simp add*: *actions-def*)
moreover from *E-i i-len* **have** *obs-i*: *action-obs E i* = *NormalAction* (*ReadMem ad al v*)
by(*simp add*: *action-obs-def lnth-ltake*)
ultimately have *read*: *i* ∈ *read-actions E* **..**
with *i-len* **have** *mrw-i*: $P,E \vdash i \rightsquigarrow mrw\ ws\ i$ **by**(*auto intro*: *mrw*)
with *E* **have** *ws i* < *i* **using** *ns* **by**(*rule mrw-before*)

from *mrw-i obs-i* **obtain** *adal-w*: (*ad*, *al*) ∈ *action-loc P E* (*ws i*)
and *adal-r*: (*ad*, *al*) ∈ *action-loc P E i*
and *write*: *ws i* ∈ *write-actions E* **by** *cases auto*

from *wf* **have** *is-write-seen P E ws* **by**(*rule wf-exec-is-write-seenD*)
from *is-write-seenD*[*OF this read obs-i*]
have *vw-v*: *value-written P E* (*ws i*) (*ad*, *al*) = *v* **by** *simp*

let *?vs* = *mrw-values P Map.empty* (*map snd* (*list-of* (*ltake* (*enat* (*ws i*)) *E*)))

```
  from ‹ws i < i› i-len have enat (ws i) < llength (ltake (enat i) E)
    by(simp add: less-trans[where y=enat i])
  hence ltake (enat i) E = lappend (ltake (enat (ws i)) (ltake (enat i) E)) (LCons (lnth (ltake (enat
i) E) (ws i)) (ldropn (Suc (ws i)) (ltake (enat i) E)))
    by(simp only: ldropn-Suc-conv-ldropn lappend-ltake-enat-ldropn)
  also have ... = lappend (ltake (enat (ws i)) E) (LCons (lnth E (ws i)) (ldropn (Suc (ws i)) (ltake
(enat i) E)))
    using ‹ws i < i› i-len ‹enat (ws i) < llength (ltake (enat i) E)›
    by (simp add: lnth-ltake)
  finally have r-E: ltake (enat i) E = ... .

  have mrw-values P Map.empty (list-of (ltake (enat i) (lmap snd (ltake r E)))) (ad, al)
    = mrw-values P Map.empty (map snd (list-of (ltake (enat i) E))) (ad, al)
    using ‹enat i < r› by(auto simp add: min-def)
  also have ... = mrw-values P (mrw-value P ?vs (snd (lnth E (ws i)))) (map snd (list-of (ldropn
(Suc (ws i)) (ltake (enat i) E)))) (ad, al)
    by(subst r-E)(simp add: list-of-lappend)
  also have ... = mrw-value P ?vs (snd (lnth E (ws i))) (ad, al)
  proof(rule mrw-values-no-write-unchanged)
    fix wa
    assume wa: wa ∈ set (map snd (list-of (ldropn (Suc (ws i)) (ltake (enat i) E))))
      and is-write-action wa (ad, al) ∈ action-loc-aux P wa

    from wa obtain w where w < length (map snd (list-of (ldropn (Suc (ws i)) (ltake (enat i) E))))
      and map snd (list-of (ldropn (Suc (ws i)) (ltake (enat i) E))) ! w = wa
      unfolding in-set-conv-nth by blast
    moreover hence Suc (ws i + w) < i (is ?w < -) using i-len
      by(cases llength E)(simp-all add: length-list-of-conv-the-enat)
    ultimately have obs-w': action-obs E ?w = wa using i-len
      by(simp add: action-obs-def lnth-ltake less-trans[where y=enat i] ac-simps)
    from ‹?w < i› i-len have ?w ∈ actions E
      by(simp add: actions-def less-trans[where y=enat i])
    with ‹is-write-action wa› obs-w' ‹(ad, al) ∈ action-loc-aux P wa›
    have write': ?w ∈ write-actions E
      and adal': (ad, al) ∈ action-loc P E ?w
      by(auto intro: write-actions.intros)

    from ‹?w < i› ‹i ∈ read-actions E› ‹?w ∈ actions E›
    have E ⊢ ?w ≤a i by(auto simp add: action-order-def elim: read-actions.cases)

    from mrw-i adal-r write' adal'
    have E ⊢ ?w ≤a ws i ∨ E ⊢ i ≤a ?w by(rule most-recent-write-recent)
    hence E ⊢ ?w ≤a ws i
    proof
      assume E ⊢ i ≤a ?w
      with ‹E ⊢ ?w ≤a i› have ?w = i by(rule antisymPD[OF antisym-action-order])
      with write' read have False by(auto dest: read-actions-not-write-actions)
      thus ?thesis ..
    qed

    from adal-w write have mrw-value P ?vs (snd (lnth E (ws i))) (ad, al) ≠ None
      by(cases snd (lnth E (ws i)) rule: mrw-value-cases)
        (auto simp add: action-obs-def split: if-split-asm elim: write-actions.cases)
    then obtain b v where vb: mrw-value P ?vs (snd (lnth E (ws i))) (ad, al) = Some (v, b) by auto
```

```
    moreover
    from ‹E ⊢ ?w ≤a ws i› obs-w'
    have is-new-action wa ¬ is-new-action (action-obs E (ws i)) by(auto elim!: action-orderE)
    from ‹¬ is-new-action (action-obs E (ws i))› write adal-w
    obtain v' where action-obs E (ws i) = NormalAction (WriteMem ad al v')
      by(auto elim!: write-actions.cases is-write-action.cases)
    with vb have b by(simp add: action-obs-def)
    with ‹is-new-action wa› vb
     show case mrw-value P ?vs (snd (lnth E (ws i))) (ad, al) of None ⇒ False | ⌊(v, b)⌋ ⇒ b ∧
is-new-action wa by simp
  qed
  also {
    fix v
    assume ?vs (ad, al) = Some (v, True)
      and is-new-action (action-obs E (ws i))

    from mrw-values-eq-SomeD[OF this(1)]
    obtain wa where wa ∈ set (map snd (list-of (ltake (enat (ws i)) E)))
      and is-write-action wa
      and (ad, al) ∈ action-loc-aux P wa
      and ¬ is-new-action wa by(fastforce simp del: set-map)
    moreover then obtain w where w: w < ws i and wa: wa = snd (lnth E w)
    unfolding in-set-conv-nth by(cases llength E)(auto simp add: lnth-ltake length-list-of-conv-the-enat)
    ultimately have w ∈ write-actions E action-obs E w = wa (ad, al) ∈ action-loc P E w
      using ‹ws i ∈ write-actions E›
       by(auto intro!: write-actions.intros simp add: actions-def less-trans[where y=enat (ws i)] ac-
tion-obs-def elim!: write-actions.cases)
    with mrw-i adal-r have E ⊢ w ≤a ws i ∨ E ⊢ i ≤a w by −(rule most-recent-write-recent)
    hence False
    proof
      assume E ⊢ w ≤a ws i
       moreover from ‹¬ is-new-action wa› ‹is-new-action (action-obs E (ws i))› write w wa ‹w ∈
write-actions E›
      have E ⊢ ws i ≤a w by(auto simp add: action-order-def action-obs-def)
      ultimately have w = ws i by(rule antisymPD[OF antisym-action-order])
      with ‹w < ws i› show False by simp
    next
      assume E ⊢ i ≤a w
      moreover from ‹w ∈ write-actions E› ‹w < ws i› ‹ws i < i› read
      have E ⊢ w ≤a i by(auto simp add: action-order-def elim: read-actions.cases)
      ultimately have i = w by(rule antisymPD[OF antisym-action-order])
      with ‹w < ws i› ‹ws i < i› show False by simp
    qed }
  then obtain b where ... = Some (v, b) using vw-v write adal-w
    apply(atomize-elim)
    apply(auto simp add: action-obs-def value-written-def write-actions-iff)
    apply(erule is-write-action.cases)
    apply auto
    done
  finally show ∃ b. mrw-values P Map.empty (list-of (ltake (enat i) (lmap snd (ltake r E)))) (ad, al)
= ⌊(v, b)⌋
    by blast
qed
```

end

context *jmm-multithreaded* **begin**

definition *complete-sc* :: ('*l*,'*thread-id*,'*x*,'*m*,'*w*) *state* ⇒ ('*addr* × *addr-loc* ⇀ '*addr val* × *bool*) ⇒
('*thread-id* × ('*l*, '*thread-id*, '*x*, '*m*, '*w*, ('*addr*, '*thread-id*) *obs-event action*) *thread-action*) *llist*
where
complete-sc s vs = *unfold-llist*
(λ(*s*, *vs*). ∀ *t ta s'*. ¬ *s* −*t*▷*ta*→ *s'*)
(λ(*s*, *vs*). *fst* (*SOME* ((*t*, *ta*), *s'*). *s* −*t*▷*ta*→ *s'* ∧ *ta-seq-consist P vs* (*llist-of* {|*ta*|}$_o$)))
(λ(*s*, *vs*). *let* ((*t*, *ta*), *s'*) = *SOME* ((*t*, *ta*), *s'*). *s* −*t*▷*ta*→ *s'* ∧ *ta-seq-consist P vs* (*llist-of* {|*ta*|}$_o$)
in (*s'*, *mrw-values P vs* {|*ta*|}$_o$))
(*s*, *vs*)

definition *sc-completion* :: ('*l*, '*thread-id*, '*x*, '*m*, '*w*) *state* ⇒ ('*addr* × *addr-loc* ⇀ '*addr val* × *bool*)
⇒ *bool*
where
sc-completion s vs ⟷
(∀ *ttas s' t x ta x' m'*.
s −▷*ttas*→* *s'* ⟶ *ta-seq-consist P vs* (*llist-of* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*))) ⟶
thr s' t = ⌊(*x*, *no-wait-locks*)⌋ ⟶ *t* ⊢ (*x*, *shr s'*) −*ta*→ (*x'*, *m'*) ⟶ *actions-ok s' t ta* ⟶
(∃ *ta' x'' m''*. *t* ⊢ (*x*, *shr s'*) −*ta'*→ (*x''*, *m''*) ∧ *actions-ok s' t ta'* ∧
ta-seq-consist P (*mrw-values P vs* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*))) (*llist-of*
{|*ta'*|}$_o$)))

lemma *sc-completionD*:
⟦ *sc-completion s vs*; *s* −▷*ttas*→* *s'*; *ta-seq-consist P vs* (*llist-of* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*)));

thr s' t = ⌊(*x*, *no-wait-locks*)⌋; *t* ⊢ (*x*, *shr s'*) −*ta*→ (*x'*, *m'*); *actions-ok s' t ta* ⟧
⟹ ∃ *ta' x'' m''*. *t* ⊢ (*x*, *shr s'*) −*ta'*→ (*x''*, *m''*) ∧ *actions-ok s' t ta'* ∧
ta-seq-consist P (*mrw-values P vs* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*))) (*llist-of* {|*ta'*|}$_o$)
unfolding *sc-completion-def* **by** *blast*

lemma *sc-completionI*:
(⋀*ttas s' t x ta x' m'*.
⟦ *s* −▷*ttas*→* *s'*; *ta-seq-consist P vs* (*llist-of* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*)));
thr s' t = ⌊(*x*, *no-wait-locks*)⌋; *t* ⊢ (*x*, *shr s'*) −*ta*→ (*x'*, *m'*); *actions-ok s' t ta* ⟧
⟹ ∃ *ta' x'' m''*. *t* ⊢ (*x*, *shr s'*) −*ta'*→ (*x''*, *m''*) ∧ *actions-ok s' t ta'* ∧
ta-seq-consist P (*mrw-values P vs* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*))) (*llist-of* {|*ta'*|}$_o$))
⟹ *sc-completion s vs*
unfolding *sc-completion-def* **by** *blast*

lemma *sc-completion-shift*:
assumes *sc-c*: *sc-completion s vs*
and *τRed*: *s* −▷*ttas*→* *s'*
and *sc*: *ta-seq-consist P vs* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* {|*ta*|}$_o$) (*llist-of ttas*)))
shows *sc-completion s'* (*mrw-values P vs* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*)))
proof(*rule sc-completionI*)
fix *ttas' s'' t x ta x' m'*
assume *τRed'*: *s'* −*ttas'*→* *s''*
and *sc'*: *ta-seq-consist P* (*mrw-values P vs* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*))) (*llist-of* (*concat*
(*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas'*)))
and *red*: *thr s'' t* = ⌊(*x*, *no-wait-locks*)⌋ *t* ⊢ ⟨*x*, *shr s''*⟩ −*ta*→ ⟨*x'*, *m'*⟩ *actions-ok s'' t ta*
from *τRed τRed'* **have** *s* −▷*ttas* @ *ttas'*→* *s''* **unfolding** *RedT-def* **by**(*rule rtrancl3p-trans*)

moreover from *sc sc'* **have** *ta-seq-consist P vs* (*llist-of* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) (*ttas* @ *ttas'*))))
apply(*simp add*: *lappend-llist-of-llist-of*[*symmetric*] *ta-seq-consist-lappend del*: *lappend-llist-of-llist-of*)
apply(*simp add*: *lconcat-llist-of*[*symmetric*] *lmap-llist-of*[*symmetric*] *llist.map-comp o-def split-def del*: *lmap-llist-of*)
done
ultimately
show ∃ *ta' x'' m''*. *t* ⊢ ⟨*x*, *shr s''*⟩ −*ta'*→ ⟨*x''*, *m''*⟩ ∧ *actions-ok s'' t ta'* ∧
ta-seq-consist P (*mrw-values P* (*mrw-values P vs* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*))) (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas'*))) (*llist-of* {|*ta'*|}$_o$)
using *red* **unfolding** *foldl-append*[*symmetric*] *concat-append*[*symmetric*] *map-append*[*symmetric*]
by(*rule sc-completionD*[*OF sc-c*])
qed

lemma *complete-sc-in-Runs*:
assumes *cau*: *sc-completion s vs*
and *ta-seq-consist-convert-RA*: ⋀*vs ln*. *ta-seq-consist P vs* (*llist-of* (*convert-RA ln*))
shows *mthr.Runs s* (*complete-sc s vs*)
proof −
let *?ttas'* = λ*ttas'* :: (*'thread-id* × (*'l*,*'thread-id*,*'x*,*'m*,*'w*, (*'addr*, *'thread-id*) *obs-event action*) *thread-action*) *list*.
concat (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas'*)
let *?vs ttas'* = *mrw-values P vs* (*?ttas' ttas'*)

define *s' vs'*
and *ttas* :: (*'thread-id* × (*'l*,*'thread-id*,*'x*,*'m*,*'w*, (*'addr*, *'thread-id*) *obs-event action*) *thread-action*) *list*
where *s'* = *s* **and** *vs'* = *vs* **and** *ttas* = []
hence *s* −▷*ttas*→* *s' ta-seq-consist P vs* (*llist-of* (*?ttas' ttas*)) **by** *auto*
hence *mthr.Runs s'* (*complete-sc s'* (*?vs ttas*))
proof(*coinduction arbitrary*: *s' ttas rule*: *mthr.Runs.coinduct*)
case (*Runs s' ttas'*)
note *Red* = ‹*s* −▷*ttas'*→* *s'*›
and *sc* = ‹*ta-seq-consist P vs* (*llist-of* (*?ttas' ttas'*))›
show *?case*
proof(*cases* ∃ *t' ta' s''*. *s'* −*t'*▷*ta'*→ *s''*)
case *False*
hence *?Stuck* **by**(*simp add*: *complete-sc-def*)
thus *?thesis* **..**
next
case *True*
let *?proceed* = λ((*t'*, *ta'*), *s''*). *s'* −*t'*▷*ta'*→ *s''* ∧ *ta-seq-consist P* (*?vs ttas'*) (*llist-of* {|*ta'*|}$_o$)
from *True* **obtain** *t' ta' s''* **where** *red*: *s'* −*t'*▷*ta'*→ *s''* **by**(*auto*)
then obtain *ta'' s'''* **where** *s'* −*t'*▷*ta''*→ *s'''*
and *ta-seq-consist P* (*?vs ttas'*) (*llist-of* {|*ta''*|}$_o$)
proof(*cases*)
case (*redT-normal x x' m'*)
note *red* = ‹*t'* ⊢ ⟨*x*, *shr s'*⟩ −*ta'*→ ⟨*x'*, *m'*⟩›
and *ts''t'* = ‹*thr s' t'* = ⌊(*x*, *no-wait-locks*)⌋›
and *aok* = ‹*actions-ok s' t' ta'*›
and *s''* = ‹*redT-upd s' t' ta' x' m' s''*›
from *sc-completionD*[*OF cau Red sc ts''t' red aok*]
obtain *ta'' x'' m''* **where** *red'*: *t'* ⊢ ⟨*x*, *shr s'*⟩ −*ta''*→ ⟨*x''*, *m''*⟩
and *aok'*: *actions-ok s' t' ta''*

```
      and sc': ta-seq-consist P (?vs ttas') (llist-of {|ta''|}_o) by blast
    from redT-updWs-total obtain ws' where redT-updWs t' (wset s') {|ta''|}_w ws' ..
    then obtain s''' where redT-upd s' t' ta'' x'' m'' s''' by fastforce
    with red' ts''t' aok' have s' -t'▷ta''→ s''' ..
    thus thesis using sc' by(rule that)
  next
    case redT-acquire
    thus thesis by(simp add: that[OF red] ta-seq-consist-convert-RA)
  qed
  hence ?proceed ((t', ta''), s''') using Red by(auto)
  hence *: ?proceed (Eps ?proceed) by(rule someI)
  moreover from Red * have s -▷ttas' @ [fst (Eps ?proceed)]→* snd (Eps ?proceed)
    by(auto simp add: split-beta RedT-def intro: rtrancl3p-step)
  moreover from True
  have complete-sc s' (?vs ttas') = LCons (fst (Eps ?proceed)) (complete-sc (snd (Eps ?proceed))
(?vs (ttas' @ [fst (Eps ?proceed)])))
    unfolding complete-sc-def by(simp add: split-def)
  moreover from sc ‹?proceed (Eps ?proceed)›
  have ta-seq-consist P vs (llist-of (?ttas' (ttas' @ [fst (Eps ?proceed)])))
    unfolding map-append concat-append lappend-llist-of-llist-of[symmetric]
    by(subst ta-seq-consist-lappend)(auto simp add: split-def)
  ultimately have ?Step
    by(fastforce intro: exI[where x=ttas' @ [fst (Eps ?proceed)]] simp del: split-paired-Ex)
  thus ?thesis by simp
 qed
qed
thus ?thesis by(simp add: s'-def ttas-def)
qed

lemma complete-sc-ta-seq-consist:
assumes cau: sc-completion s vs
and ta-seq-consist-convert-RA: ⋀vs ln. ta-seq-consist P vs (llist-of (convert-RA ln))
shows ta-seq-consist P vs (lconcat (lmap (λ(t, ta). llist-of {|ta|}_o) (complete-sc s vs)))
proof −
define vs' where vs' = vs
let ?obs = λttas. lconcat (lmap (λ(t, ta). llist-of {|ta|}_o) ttas)
define obs where obs = ?obs (complete-sc s vs)
define a where a = complete-sc s vs'
let ?ttas' = λttas' :: ('thread-id × ('l,'thread-id,'x,'m,'w,('addr, 'thread-id) obs-event action) thread-action)
list.
          concat (map (λ(t, ta). {|ta|}_o) ttas')
let ?vs = λttas'. mrw-values P vs (?ttas' ttas')
from vs'-def obs-def
have s -▷[]→* s ta-seq-consist P vs (llist-of (?ttas' [])) vs' = ?vs [] by(auto)
hence ∃ s' ttas'. obs = ?obs (complete-sc s' vs') ∧ s -▷ttas'→* s' ∧
                ta-seq-consist P vs (llist-of (?ttas' ttas')) ∧ vs' = ?vs ttas' ∧
                a = complete-sc s' vs'
  unfolding obs-def vs'-def a-def by metis
moreover have wf (inv-image {(m, n). m < n} (llength ∘ ltakeWhile (λ(t, ta). {|ta|}_o = [])))
  (is wf ?R) by(rule wf-inv-image)(rule wellorder-class.wf)
ultimately show ta-seq-consist P vs' obs
proof(coinduct vs' obs a rule: ta-seq-consist-coinduct-append-wf)
  case (ta-seq-consist vs' obs a)
  then obtain s' ttas' where obs-def: obs = ?obs (complete-sc s' (?vs ttas'))
```

```
  and Red: s −▷ttas′→* s′
  and sc: ta-seq-consist P vs (llist-of (?ttas′ ttas′))
  and vs′-def: vs′ = ?vs ttas′
  and a-def: a = complete-sc s′ vs′ by blast

show ?case
proof(cases ∃ t′ ta′ s″. s′ −t′▷ta′→ s″)
  case False
  hence obs = LNil unfolding obs-def complete-sc-def by simp
  hence ?LNil unfolding obs-def by auto
  thus ?thesis ..
next
  case True
  let ?proceed = λ((t′, ta′), s″). s′ −t′▷ta′→ s″ ∧ ta-seq-consist P (?vs ttas′) (llist-of ⦃ta′⦄o)
  let ?tta = fst (Eps ?proceed)
  let ?s′ = snd (Eps ?proceed)

  from True obtain t′ ta′ s″ where red: s′ −t′▷ta′→ s″ by blast
  then obtain ta″ s‴ where s′ −t′▷ta″→ s‴
    and ta-seq-consist P (?vs ttas′) (llist-of ⦃ta″⦄o)
  proof(cases)
    case (redT-normal x x′ m′)
    note red = ‹t′ ⊢ ⟨x, shr s′⟩ −ta′→ ⟨x′, m′⟩›
      and ts″t′ = ‹thr s′ t′ = ⌊(x, no-wait-locks)⌋›
      and aok = ‹actions-ok s′ t′ ta′›
      and s‴ = ‹redT-upd s′ t′ ta′ x′ m′ s″›
    from sc-completionD[OF cau Red sc ts″t′ red aok]
    obtain ta″ x″ m″ where red′: t′ ⊢ ⟨x, shr s′⟩ −ta″→ ⟨x″, m″⟩
      and aok′: actions-ok s′ t′ ta″
      and sc′: ta-seq-consist P (?vs ttas′) (llist-of ⦃ta″⦄o) by blast
    from redT-updWs-total obtain ws′ where redT-updWs t′ (wset s′) ⦃ta″⦄w ws′ ..
    then obtain s‴ where redT-upd s′ t′ ta″ x″ m″ s‴ by fastforce
    with red′ ts″t′ aok′ have s′ −t′▷ta″→ s‴ ..
    thus thesis using sc′ by(rule that)
  next
    case redT-acquire
    thus thesis by(simp add: that[OF red] ta-seq-consist-convert-RA)
  qed
  hence ?proceed ((t′, ta″), s‴) by auto
  hence ?proceed (Eps ?proceed) by(rule someI)
  show ?thesis
  proof(cases obs = LNil)
    case True thus ?thesis ..
  next
    case False
    from True
    have csc-unfold: complete-sc s′ (?vs ttas′) = LCons ?tta (complete-sc ?s′ (?vs (ttas′ @ [?tta])))
      unfolding complete-sc-def by(simp add: split-def)
    hence obs = lappend (llist-of ⦃snd ?tta⦄o) (?obs (complete-sc ?s′ (?vs (ttas′ @ [?tta]))))
      using obs-def by(simp add: split-beta)
    moreover have ta-seq-consist P vs′ (llist-of ⦃snd ?tta⦄o)
      using ‹?proceed (Eps ?proceed)› vs′-def by(clarsimp simp add: split-beta)
    moreover {
      assume llist-of ⦃snd ?tta⦄o = LNil
```

moreover from *obs-def* ‹*obs* ≠ *LNil*›
have *lfinite* (*ltakeWhile* (λ(*t*, *ta*). {|*ta*|}$_o$ = []) (*complete-sc s'* (*?vs ttas'*)))
unfolding *lfinite-ltakeWhile* **by**(*fastforce simp add*: *split-def lconcat-eq-LNil*)
ultimately have (*complete-sc ?s'* (*?vs* (*ttas'* @ [*?tta*])), *a*) ∈ *?R*
unfolding *a-def vs'-def csc-unfold*
by(*clarsimp simp add*: *split-def llist-of-eq-LNil-conv*)(*auto simp add*: *lfinite-eq-range-llist-of*)
}
moreover have *?obs* (*complete-sc ?s'* (*?vs* (*ttas'* @ [*?tta*]))) = *?obs* (*complete-sc ?s'* (*mrw-values P vs'* (*list-of* (*llist-of* {|*snd ?tta*|}$_o$))))
unfolding *vs'-def* **by**(*simp add*: *split-def*)
moreover from ‹*?proceed* (*Eps ?proceed*)› *Red*
have *s* −▷*ttas'* @ [*?tta*]→∗ *?s'* **by**(*auto simp add*: *RedT-def split-def intro*: *rtrancl3p-step*)
moreover from *sc* ‹*?proceed* (*Eps ?proceed*)›
have *ta-seq-consist P vs* (*llist-of* (*?ttas'* (*ttas'* @ [*?tta*])))
by(*clarsimp simp add*: *split-def ta-seq-consist-lappend lappend-llist-of-llist-of*[*symmetric*] *simp del*: *lappend-llist-of-llist-of*)
moreover have *mrw-values P vs'* (*list-of* (*llist-of* {|*snd ?tta*|}$_o$)) = *?vs* (*ttas'* @ [*?tta*])
unfolding *vs'-def* **by**(*simp add*: *split-def*)
moreover have *complete-sc ?s'* (*?vs* (*ttas'* @ [*?tta*])) = *complete-sc ?s'* (*mrw-values P vs'* (*list-of* (*llist-of* {|*snd ?tta*|}$_o$)))
unfolding *vs'-def* **by**(*simp add*: *split-def*)
ultimately have *?lappend* **by** *blast*
thus *?thesis* **..**
qed
qed
qed
qed

lemma *sequential-completion-Runs*:
assumes *sc-completion s vs*
and ⋀*vs ln*. *ta-seq-consist P vs* (*llist-of* (*convert-RA ln*))
shows ∃*ttas*. *mthr.Runs s ttas* ∧ *ta-seq-consist P vs* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* {|*ta*|}$_o$) *ttas*))
using *complete-sc-ta-seq-consist*[*OF assms*] *complete-sc-in-Runs*[*OF assms*]
by *blast*

definition *cut-and-update* :: ('*l*, '*thread-id*, '*x*, '*m*, '*w*) *state* ⇒ ('*addr* × *addr-loc* ⇀ '*addr val* × *bool*) ⇒ *bool*
where
cut-and-update s vs ⟷
(∀ *ttas s' t x ta x' m'*.
s −▷*ttas*→∗ *s'* ⟶ *ta-seq-consist P vs* (*llist-of* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*))) ⟶
thr s' t = ⌊(*x*, *no-wait-locks*)⌋ ⟶ *t* ⊢ (*x*, *shr s'*) −*ta*→ (*x'*, *m'*) ⟶ *actions-ok s' t ta* ⟶
(∃ *ta' x'' m''*. *t* ⊢ (*x*, *shr s'*) −*ta'*→ (*x''*, *m''*) ∧ *actions-ok s' t ta'* ∧
ta-seq-consist P (*mrw-values P vs* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*))) (*llist-of* {|*ta'*|}$_o$) ∧
eq-upto-seq-inconsist P {|*ta*|}$_o$ {|*ta'*|}$_o$ (*mrw-values P vs* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*)))))

lemma *cut-and-updateI*[*intro?*]:
(⋀*ttas s' t x ta x' m'*.
⟦ *s* −▷*ttas*→∗ *s'*; *ta-seq-consist P vs* (*llist-of* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*)));
thr s' t = ⌊(*x*, *no-wait-locks*)⌋; *t* ⊢ (*x*, *shr s'*) −*ta*→ (*x'*, *m'*); *actions-ok s' t ta* ⟧
⟹ ∃ *ta' x'' m''*. *t* ⊢ (*x*, *shr s'*) −*ta'*→ (*x''*, *m''*) ∧ *actions-ok s' t ta'* ∧

ta-seq-consist P (mrw-values P vs (concat (map (λ(t, ta). {|ta|}$_o$) ttas))) (llist-of {|ta'|}$_o$) ∧
eq-upto-seq-inconsist P {|ta|}$_o$ {|ta'|}$_o$ (mrw-values P vs (concat (map (λ(t, ta). {|ta|}$_o$) ttas))))
⟹ cut-and-update s vs
unfolding *cut-and-update-def* **by** *blast*

lemma *cut-and-updateD*:
⟦ cut-and-update s vs; s −▷ttas→ s'; ta-seq-consist P vs (llist-of (concat (map (λ(t, ta). {|ta|}$_o$) ttas))); thr s' t = ⌊(x, no-wait-locks)⌋; t ⊢ (x, shr s') −ta→ (x', m'); actions-ok s' t ta ⟧*
⟹ ∃ta' x'' m''. t ⊢ (x, shr s') −ta'→ (x'', m'') ∧ actions-ok s' t ta' ∧
ta-seq-consist P (mrw-values P vs (concat (map (λ(t, ta). {|ta|}$_o$) ttas))) (llist-of {|ta'|}$_o$) ∧
eq-upto-seq-inconsist P {|ta|}$_o$ {|ta'|}$_o$ (mrw-values P vs (concat (map (λ(t, ta). {|ta|}$_o$) ttas)))
unfolding *cut-and-update-def* **by** *blast*

lemma *cut-and-update-imp-sc-completion*:
cut-and-update s vs ⟹ sc-completion s vs
apply(*rule sc-completionI*)
apply(*drule* (5) *cut-and-updateD*)
apply *blast*
done

lemma *sequential-completion*:
assumes *cut-and-update*: *cut-and-update s vs*
and *ta-seq-consist-convert-RA*: *⋀vs ln. ta-seq-consist P vs (llist-of (convert-RA ln))*
and *Red*: *s −▷ttas→* s'*
and *sc*: *ta-seq-consist P vs (llist-of (concat (map (λ(t, ta). {|ta|}$_o$) ttas)))*
and *red*: *s' −t▷ta→ s''*
shows
∃ta' ttas'. mthr.Runs s' (LCons (t, ta') ttas') ∧
ta-seq-consist P vs (lconcat (lmap (λ(t, ta). llist-of {|ta|}$_o$) (lappend (llist-of ttas) (LCons (t, ta') ttas')))) ∧
eq-upto-seq-inconsist P {|ta|}$_o$ {|ta'|}$_o$ (mrw-values P vs (concat (map (λ(t, ta). {|ta|}$_o$) ttas)))
proof −
from *red* **obtain** *ta' s'''*
where *red'*: *redT s' (t, ta') s'''*
and *sc'*: *ta-seq-consist P vs (lconcat (lmap (λ(t, ta). llist-of {|ta|}$_o$) (lappend (llist-of ttas) (LCons (t, ta') LNil))))*
and *eq*: *eq-upto-seq-inconsist P {|ta|}$_o$ {|ta'|}$_o$ (mrw-values P vs (concat (map (λ(t, ta). {|ta|}$_o$) ttas)))*
proof *cases*
case (*redT-normal x x' m'*)
note *ts't* = ‹*thr s' t = ⌊(x, no-wait-locks)⌋*›
and *red* = ‹*t ⊢ ⟨x, shr s'⟩ −ta→ ⟨x', m'⟩*›
and *aok* = ‹*actions-ok s' t ta*›
and *s''* = ‹*redT-upd s' t ta x' m' s''*›
from *cut-and-updateD*[*OF cut-and-update, OF Red sc ts't red aok*]
obtain *ta' x'' m''* **where** *red*: *t ⊢ ⟨x, shr s'⟩ −ta'→ ⟨x'', m''⟩*
and *sc'*: *ta-seq-consist P (mrw-values P vs (concat (map (λ(t, y). {|y|}$_o$) ttas))) (llist-of {|ta'|}$_o$)*
and *eq*: *eq-upto-seq-inconsist P {|ta|}$_o$ {|ta'|}$_o$ (mrw-values P vs (concat (map (λ(t, ta). {|ta|}$_o$) ttas)))*

and *aok*: *actions-ok s′ t ta′* **by** *blast*
obtain *ws‴* **where** *redT-updWs t* (*wset s′*) $\{\!|ta'|\!\}_w$ *ws‴*
using *redT-updWs-total* **..**
then obtain *s‴* **where** *s‴*: *redT-upd s′ t ta′ x″ m″ s‴* **by** *fastforce*
with *red* ‹*thr s′ t* = ⌊(*x*, *no-wait-locks*)⌋› *aok* **have** *s′* −*t*▷*ta′*→ *s‴* **by**(*rule redT.redT-normal*)
moreover from *sc sc′*
have *ta-seq-consist P vs* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* $\{\!|ta|\!\}_o$) (*lappend* (*llist-of ttas*) (*LCons* (*t*, *ta′*) *LNil*))))
by(*auto simp add*: *lmap-lappend-distrib ta-seq-consist-lappend split-def lconcat-llist-of*[*symmetric*] *o-def list-of-lconcat*)
ultimately show *thesis* **using** *eq* **by**(*rule that*)
next
case (*redT-acquire x ln n*)
hence *ta-seq-consist P vs* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* $\{\!|ta|\!\}_o$) (*lappend* (*llist-of ttas*) (*LCons* (*t*, *ta*) *LNil*))))
and *eq-upto-seq-inconsist P* $\{\!|ta|\!\}_o$ $\{\!|ta|\!\}_o$ (*mrw-values P vs* (*concat* (*map* (λ(*t*, *ta*). $\{\!|ta|\!\}_o$) *ttas*)))
using *sc*
by(*simp-all add*: *lmap-lappend-distrib ta-seq-consist-lappend split-def lconcat-llist-of*[*symmetric*] *o-def list-of-lconcat ta-seq-consist-convert-RA ta-seq-consist-imp-eq-upto-seq-inconsist-refl*)
with *red* **show** *thesis* **by**(*rule that*)
qed

Now, find a sequentially consistent completion from *s‴* onwards.

from *Red red′* **have** *Red′*: *s* −▷*ttas* @ [(*t*, *ta′*)]→∗ *s‴*
unfolding *RedT-def* **by**(*auto intro*: *rtrancl3p-step*)

from *sc sc′*
have *ta-seq-consist P vs* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* $\{\!|ta|\!\}_o$) (*llist-of* (*ttas* @ [(*t*, *ta′*)]))))
by(*simp add*: *o-def split-def lappend-llist-of-llist-of*[*symmetric*])
with *cut-and-update-imp-sc-completion*[*OF cut-and-update*] *Red′*
have *sc-completion s‴* (*mrw-values P vs* (*concat* (*map* (λ(*t*, *ta*). $\{\!|ta|\!\}_o$) (*ttas* @ [(*t*, *ta′*)]))))
by(*rule sc-completion-shift*)
from *sequential-completion-Runs*[*OF this ta-seq-consist-convert-RA*]
obtain *ttas′* **where** *τRuns*: *mthr.Runs s‴ ttas′*
and *sc″*: *ta-seq-consist P* (*mrw-values P vs* (*concat* (*map* (λ(*t*, *ta*). $\{\!|ta|\!\}_o$) (*ttas* @ [(*t*, *ta′*)]))))
(*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* $\{\!|ta|\!\}_o$) *ttas′*))
by *blast*
from *red′ τRuns* **have** *mthr.Runs s′* (*LCons* (*t*, *ta′*) *ttas′*) **..**
moreover from *sc sc′ sc″*
have *ta-seq-consist P vs* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* $\{\!|ta|\!\}_o$) (*lappend* (*llist-of ttas*) (*LCons* (*t*, *ta′*) *ttas′*))))
unfolding *lmap-lappend-distrib lconcat-lappend* **by**(*simp add*: *o-def ta-seq-consist-lappend split-def list-of-lconcat*)
ultimately show *?thesis* **using** *eq* **by** *blast*
qed

end

end

8.9 Happens-before consistent completion of executions in the JMM

theory *HB-Completion* **imports**
Non-Speculative
begin

coinductive *ta-hb-consistent* :: *'m prog* ⇒ (*'thread-id* × (*'addr*, *'thread-id*) *obs-event action*) *list* ⇒ (*'thread-id* × (*'addr*, *'thread-id*) *obs-event action*) *llist* ⇒ *bool*
for *P* :: *'m prog*
where
LNil: *ta-hb-consistent P obs LNil*
| *LCons*:
⟦ *ta-hb-consistent P* (*obs* @ [*ob*]) *obs'*;
case ob of (*t*, *NormalAction* (*ReadMem ad al v*))
⇒ (∃ *w*. *w* ∈ *write-actions* (*llist-of* (*obs* @ [*ob*])) ∧ (*ad*, *al*) ∈ *action-loc P* (*llist-of* (*obs* @ [*ob*])) *w* ∧
value-written P (*llist-of* (*obs* @ [*ob*])) *w* (*ad*, *al*) = *v* ∧
P,*llist-of* (*obs* @ [*ob*]) ⊢ *w* ≤*hb length obs* ∧
(∀ *w'*∈*write-actions* (*llist-of* (*obs* @ [*ob*])). (*ad*, *al*) ∈ *action-loc P* (*llist-of* (*obs* @ [*ob*])) *w'* ⟶
(*P*,*llist-of* (*obs* @ [*ob*]) ⊢ *w* ≤*hb w'* ∧ *P*,*llist-of* (*obs* @ [*ob*]) ⊢ *w'* ≤*hb length obs* ∨
is-volatile P al ∧ *P*,*llist-of* (*obs* @ [*ob*]) ⊢ *w* ≤*so w'* ∧ *P*,*llist-of* (*obs* @ [*ob*]) ⊢ *w'* ≤*so length obs*) ⟶
w' = *w*))
| - ⇒ *True* ⟧
⟹ *ta-hb-consistent P obs* (*LCons ob obs'*)

inductive-simps *ta-hb-consistent-LNil* [*simp*]:
ta-hb-consistent P obs LNil

inductive-simps *ta-hb-consistent-LCons*:
ta-hb-consistent P obs (*LCons ob obs'*)

lemma *ta-hb-consistent-into-non-speculative*:
ta-hb-consistent P obs0 obs
⟹ *non-speculative P* (*w-values P* (λ-. {}) (*map snd obs0*)) (*lmap snd obs*)
proof(*coinduction arbitrary*: *obs0 obs*)
case (*non-speculative obs0 obs*)
let *?vs* = *w-values P* (λ-. {}) (*map snd obs0*)
let *?CH* = λ*vs obs'*. ∃ *obs0 obs*. *vs* = *w-values P* (λ-. {}) (*map snd obs0*) ∧ *obs'* = *lmap snd obs* ∧ *ta-hb-consistent P obs0 obs*
from *non-speculative* **show** *?case*
proof(*cases*)
case *LNil* **hence** *?LNil* **by** *simp*
thus *?thesis* **..**
next
case (*LCons tob obs''*)
note *obs* = ‹*obs* = *LCons tob obs''*›
obtain *t ob* **where** *tob*: *tob* = (*t*, *ob*) **by**(*cases tob*)
from ‹*ta-hb-consistent P* (*obs0* @ [*tob*]) *obs''*› *tob obs*
have *?CH* (*w-value P ?vs ob*) (*lmap snd obs''*) **by**(*auto intro*!: *exI*)
moreover {
fix *ad al v*

```
    assume ob: ob = NormalAction (ReadMem ad al v)
    with LCons tob obtain w where w: w ∈ write-actions (llist-of (obs0 @ [tob]))
     and adal: (ad, al) ∈ action-loc P (llist-of (obs0 @ [tob])) w
     and v: value-written P (llist-of (obs0 @ [tob])) w (ad, al) = v by auto
    from w obtain is-write-action (action-obs (llist-of (obs0 @ [tob])) w)
     and w-actions: w ∈ actions (llist-of (obs0 @ [tob])) by cases
    hence v ∈ ?vs (ad, al)
    proof(cases)
     case (WriteMem ad' al' v')
     hence NormalAction (WriteMem ad al v) ∈ set (map snd obs0)
      using adal ob tob v w-actions unfolding in-set-conv-nth
      by(auto simp add: action-obs-def nth-append value-written.simps actions-def cong: conj-cong
split: if-split-asm)
     thus ?thesis by(rule w-values-WriteMemD)
    next
     case (NewHeapElem ad' hT)
     hence NormalAction (NewHeapElem ad hT) ∈ set (map snd obs0)
      using adal ob tob v w-actions unfolding in-set-conv-nth
      by(auto simp add: action-obs-def nth-append value-written.simps actions-def cong: conj-cong
split: if-split-asm)
     thus ?thesis using NewHeapElem adal unfolding v[symmetric]
      by(fastforce simp add: value-written.simps intro!: w-values-new-actionD intro: rev-image-eqI)
    qed }
   hence case ob of NormalAction (ReadMem ad al v) ⇒ v ∈ ?vs (ad, al) | - ⇒ True
    by(simp split: action.split obs-event.split)
   ultimately have ?LCons using obs tob by simp
   thus ?thesis ..
  qed
qed

lemma ta-hb-consistent-lappendI:
  assumes hb1: ta-hb-consistent P E E'
  and hb2: ta-hb-consistent P (E @ list-of E') E''
  and fin: lfinite E'
  shows ta-hb-consistent P E (lappend E' E'')
using fin hb1 hb2
proof(induction arbitrary: E)
  case lfinite-LNil thus ?case by simp
next
  case (lfinite-LConsI E' tob)
  from ‹ta-hb-consistent P E (LCons tob E')›
  have ta-hb-consistent P (E @ [tob]) E' by cases
  moreover from ‹ta-hb-consistent P (E @ list-of (LCons tob E')) E''› ‹lfinite E'›
  have ta-hb-consistent P ((E @ [tob]) @ list-of E') E'' by simp
  ultimately have ta-hb-consistent P (E @ [tob]) (lappend E' E'') by(rule lfinite-LConsI.IH)
  thus ?case unfolding lappend-code apply(rule ta-hb-consistent.LCons)
   using ‹ta-hb-consistent P E (LCons tob E')›
   by cases (simp split: prod.split-asm action.split-asm obs-event.split-asm)
qed

lemma ta-hb-consistent-coinduct-append
  [consumes 1, case-names ta-hb-consistent, case-conclusion ta-hb-consistent LNil lappend]:
  assumes major: X E tobs
  and step: ⋀E tobs. X E tobs
```

```
    ⟹ tobs = LNil ∨
      (∃ tobs' tobs''. tobs = lappend tobs' tobs'' ∧ tobs' ≠ LNil ∧ ta-hb-consistent P E tobs' ∧
                  (lfinite tobs' ⟶ (X (E @ list-of tobs') tobs''∨
                                    ta-hb-consistent P (E @ list-of tobs') tobs'')))
    (is ⋀E tobs. - ⟹ - ∨ ?step E tobs)
  shows ta-hb-consistent P E tobs
proof −
  from major
  have ∃ tobs' tobs''. tobs = lappend (llist-of tobs') tobs'' ∧ ta-hb-consistent P E (llist-of tobs') ∧
                  X (E @ tobs') tobs''
    by(auto intro: exI[where x=[]])
  thus ?thesis
  proof(coinduct)
    case (ta-hb-consistent E tobs)
    then obtain tobs' tobs''
      where tobs: tobs = lappend (llist-of tobs') tobs''
      and hb-tobs': ta-hb-consistent P E (llist-of tobs')
      and X: X (E @ tobs') tobs'' by blast

    show ?case
    proof(cases tobs')
      case Nil
      with X have X E tobs'' by simp
      from step[OF this] show ?thesis
      proof
        assume tobs'' = LNil
        with Nil tobs show ?thesis by simp
      next
        assume ?step E tobs''
        then obtain tobs''' tobs''''
          where tobs'': tobs'' = lappend tobs''' tobs'''' and tobs''' ≠ LNil
          and sc-obs''': ta-hb-consistent P E tobs'''
          and fin: lfinite tobs''' ⟹ X (E @ list-of tobs''') tobs'''' ∨
                                    ta-hb-consistent P (E @ list-of tobs''') tobs''''
          by blast
        from ‹tobs''' ≠ LNil› obtain t ob tobs''''' where tobs''': tobs''' = LCons (t, ob) tobs'''''
          unfolding neq-LNil-conv by auto
        with Nil tobs'' tobs have concl1: tobs = LCons (t, ob) (lappend tobs''''' tobs'''') by simp

        have ?LCons
        proof(cases lfinite tobs''')
          case False
          hence lappend tobs''''' tobs'''' = tobs''''' using tobs''' by(simp add: lappend-inf)
          hence ta-hb-consistent P (E @ [(t, ob)]) (lappend tobs''''' tobs'''')
            using sc-obs''' tobs''' by(simp add: ta-hb-consistent-LCons)
          with concl1 show ?LCons apply(simp)
            using sc-obs'''[unfolded tobs'''] by cases simp
        next
          case True
          with tobs''' obtain tobs'''''' where tobs''''': tobs''''' = llist-of tobs''''''
            by simp(auto simp add: lfinite-eq-range-llist-of)
          from fin[OF True]
         have ta-hb-consistent P (E @ [(t, ob)]) (llist-of tobs'''''') ∧ X (E @ (t, ob) # tobs'''''') tobs''''
∨
```

```
            ta-hb-consistent P (E @ [(t, ob)]) (lappend (llist-of tobs''''''') tobs'''')
        proof
          assume X: X (E @ list-of tobs''') tobs''''
          hence X (E @ (t, ob) # tobs''''') tobs'''' using tobs''''' tobs''' by simp
          moreover have ta-hb-consistent P (E @ [(t, ob)]) (llist-of tobs''''''')
            using sc-obs''' tobs''' tobs''''' by(simp add: ta-hb-consistent-LCons)
          ultimately show ?thesis by simp
        next
          assume ta-hb-consistent P (E @ list-of tobs''') tobs''''
          with sc-obs''' tobs''''' tobs'''
          have ta-hb-consistent P (E @ [(t, ob)]) (lappend (llist-of tobs''''''') tobs'''')
            by(simp add: ta-hb-consistent-LCons ta-hb-consistent-lappendI)
          thus ?thesis ..
        qed
        hence ((∃ tobs' tobs''. lappend (llist-of tobs''''''') tobs'''' = lappend (llist-of tobs') tobs'' ∧
                              ta-hb-consistent P (E @ [(t, ob)]) (llist-of tobs') ∧ X (E @ (t, ob) # tobs')
tobs'') ∨
              ta-hb-consistent P (E @ [(t, ob)]) (lappend (llist-of tobs''''''') tobs''''))
          by auto
        thus ?LCons using concl1 tobs''''' apply(simp)
          using sc-obs'''[unfolded tobs'''] by cases simp
      qed
      thus ?thesis ..
    qed
  next
    case (Cons tob tobs''')
    with X tobs hb-tobs' show ?thesis by(auto simp add: ta-hb-consistent-LCons)
  qed
 qed
qed

lemma ta-hb-consistent-coinduct-append-wf
  [consumes 2, case-names ta-hb-consistent, case-conclusion ta-hb-consistent LNil lappend]:
  assumes major: X E obs a
  and wf: wf R
  and step: ⋀E obs a. X E obs a
    ⟹ obs = LNil ∨
      (∃ obs' obs'' a'. obs = lappend obs' obs'' ∧ ta-hb-consistent P E obs' ∧ (obs' = LNil ⟶ (a', a)
∈ R) ∧
                    (lfinite obs' ⟶ X (E @ list-of obs') obs'' a' ∨
                                     ta-hb-consistent P (E @ list-of obs') obs''))
    (is ⋀E obs a. - ⟹ - ∨ ?step E obs a)
  shows ta-hb-consistent P E obs
proof −
  { fix E obs a
    assume X E obs a
    with wf
    have obs = LNil ∨ (∃ obs' obs''. obs = lappend obs' obs'' ∧ obs' ≠ LNil ∧ ta-hb-consistent P E
obs' ∧
          (lfinite obs' ⟶ (∃ a. X (E @ list-of obs') obs'' a) ∨
                          ta-hb-consistent P (E @ list-of obs') obs''))
      (is - ∨ ?step-concl E obs)
    proof(induction a arbitrary: E obs rule: wf-induct[consumes 1, case-names wf])
      case (wf a)
```

```
      note IH = wf.IH[rule-format]
      from step[OF ‹X E obs a›]
      show ?case
      proof
        assume obs = LNil thus ?thesis ..
      next
        assume ?step E obs a
        then obtain obs' obs'' a'
          where obs: obs = lappend obs' obs''
          and sc-obs': ta-hb-consistent P E obs'
          and decr: obs' = LNil ⟹ (a', a) ∈ R
          and fin: lfinite obs' ⟹
                X (E @ list-of obs') obs'' a' ∨
                ta-hb-consistent P (E @ list-of obs') obs''
          by blast
        show ?case
        proof(cases obs' = LNil)
          case True
          hence lfinite obs' by simp
          from fin[OF this] show ?thesis
          proof
            assume X: X (E @ list-of obs') obs'' a'
            from True have (a', a) ∈ R by(rule decr)
            from IH[OF this X] show ?thesis
            proof
              assume obs'' = LNil
              with True obs have obs = LNil by simp
              thus ?thesis ..
            next
              assume ?step-concl (E @ list-of obs') obs''
              hence ?step-concl E obs using True obs by simp
              thus ?thesis ..
            qed
          next
            assume ta-hb-consistent P (E @ list-of obs') obs''
            thus ?thesis using obs True
              by cases (auto 4 3 cong: action.case-cong obs-event.case-cong intro: exI[where x=LCons
x LNil for x] simp add: ta-hb-consistent-LCons)
          qed
        next
          case False
          with obs sc-obs' fin show ?thesis by auto
        qed
      qed
    qed }
  note step' = this

  from major show ?thesis
  proof(coinduction arbitrary: E obs a rule: ta-hb-consistent-coinduct-append)
    case (ta-hb-consistent E obs)
    thus ?case by simp(rule step')
  qed
qed
```

lemma *ta-hb-consistent-lappendD2*:
assumes *hb*: *ta-hb-consistent P E* (*lappend E′ E″*)
and *fin*: *lfinite E′*
shows *ta-hb-consistent P* (*E* @ *list-of E′*) *E″*
using *fin hb*
by(*induct arbitrary*: *E*)(*fastforce simp add*: *ta-hb-consistent-LCons*)+

lemma *ta-hb-consistent-Read-hb*:
fixes *E E′* **defines** *E″* ≡ *lappend* (*llist-of E′*) *E*
assumes *hb*: *ta-hb-consistent P E′ E*
and *tsa*: *thread-start-actions-ok E″*
and *E″*: *is-write-seen P* (*llist-of E′*) *ws′*
and *new-actions-for-fun*:
⋀*w w′ adal*. ⟦ *w* ∈ *new-actions-for P E″ adal*;
w′ ∈ *new-actions-for P E″ adal* ⟧ ⟹ *w* = *w′*
shows ∃ *ws*. *P* ⊢ (*E″*, *ws*) √ ∧ (∀ *n*. *n* ∈ *read-actions E″* ⟶ *length E′* ≤ *n* ⟶ *P*,*E″* ⊢ *ws n* ≤*hb* *n*) ∧
(∀ *n*. *n* < *length E′* ⟶ *ws n* = *ws′ n*)
proof(*intro exI conjI strip*)
let *?P* =
λ*n w*. *case lnth E″ n of*
(*t*, *NormalAction* (*ReadMem ad al v*)) ⇒
(*w* ∈ *write-actions E″* ∧ (*ad*, *al*) ∈ *action-loc P E″ w* ∧ *value-written P E″ w* (*ad*, *al*) = *v* ∧
P,*E″* ⊢ *w* ≤*hb n* ∧
(∀ *w′*∈*write-actions E″*. (*ad*, *al*) ∈ *action-loc P E″ w′* ⟶
(*P*,*E″* ⊢ *w* ≤*hb w′* ∧ *P*,*E″* ⊢ *w′* ≤*hb n* ∨
is-volatile P al ∧ *P*,*E″* ⊢ *w* ≤*so w′* ∧ *P*,*E″* ⊢ *w′* ≤*so n*) ⟶
w′ = *w*))
let *?ws* = λ*n*. *if n* < *length E′ then ws′ n else Eps* (*?P n*)

have ⋀*n*. *n* < *length E′* ⟹ *?ws n* = *ws′ n* **by** *simp*
moreover
have *P* ⊢ (*E″*, *?ws*) √ ∧
(∀ *n ad al v*. *n* ∈ *read-actions E″* ⟶ *length E′* ≤ *n* ⟶ *action-obs E″ n* = *NormalAction* (*ReadMem ad al v*) ⟶ *P*,*E″* ⊢ *?ws n* ≤*hb n*)
proof(*intro conjI wf-execI strip is-write-seenI*)
fix *a′ ad al v*
assume *read*: *a′* ∈ *read-actions E″*
and *aobs*: *action-obs E″ a′* = *NormalAction* (*ReadMem ad al v*)
then obtain *t* **where** *a′*: *enat a′* < *llength E″*
and *lnth″*: *lnth E″ a′* = (*t*, *NormalAction* (*ReadMem ad al v*))
by(*cases*)(*cases lnth E″ a′*, *clarsimp simp add*: *actions-def action-obs-def*)

have *?ws a′* ∈ *write-actions E″* ∧
(*ad*, *al*) ∈ *action-loc P E″* (*?ws a′*) ∧
value-written P E″ (*?ws a′*) (*ad*, *al*) = *v* ∧
(*length E′* ≤ *a′* ⟶ *P*,*E″* ⊢ *?ws a′* ≤*hb a′*) ∧
¬ *P*,*E″* ⊢ *a′* ≤*hb ?ws a′* ∧
(*is-volatile P al* ⟶ ¬ *P*,*E″* ⊢ *a′* ≤*so ?ws a′*) ∧
(∀ *a″*. *a″* ∈ *write-actions E″* ⟶ (*ad*, *al*) ∈ *action-loc P E″ a″* ⟶
(*P*,*E″* ⊢ *?ws a′* ≤*hb a″* ∧ *P*,*E″* ⊢ *a″* ≤*hb a′* ∨ *is-volatile P al* ∧ *P*,*E″* ⊢ *?ws a′* ≤*so a″* ∧ *P*,*E″* ⊢ *a″* ≤*so a′*)
⟶ *a″* = *?ws a′*)
proof(*cases a′* < *length E′*, *safe del*: *notI disjE conjE*)

assume *a′-E′*: *a′ < length E′*
with *read aobs* **have** *a′*: *a′ ∈ read-actions* (*llist-of E′*)
and *aobs′*: *action-obs* (*llist-of E′*) *a′ = NormalAction* (*ReadMem ad al v*)
by(*auto simp add*: *E′′-def action-obs-def lnth-lappend1 actions-def elim*: *read-actions.cases intro*: *read-actions.intros*)
have *sim*: *ltake* (*enat* (*length E′*)) (*llist-of E′*) [≈] *ltake* (*enat* (*length E′*)) (*lappend* (*llist-of E′*) *E*)
by(*rule eq-into-sim-actions*)(*simp add*: *ltake-all ltake-lappend1*)
from *tsa* **have** *tsa′*: *thread-start-actions-ok* (*llist-of E′*)
by(*rule thread-start-actions-ok-prefix*)(*simp add*: *E′′-def lprefix-lappend*)

from *is-write-seenD*[*OF E′′ a′ aobs′*] *a′-E′*
show *?ws a′ ∈ write-actions E′′*
and (*ad*, *al*) ∈ *action-loc P E′′* (*?ws a′*)
and *value-written P E′′* (*?ws a′*) (*ad*, *al*) = *v*
and ¬ *P*,*E′′* ⊢ *a′* ≤*hb ?ws a′*
and *is-volatile P al* ⟹ ¬ *P*,*E′′* ⊢ *a′* ≤*so ?ws a′*
by(*auto elim*!: *write-actions.cases intro*!: *write-actions.intros simp add*: *E′′-def lnth-lappend1 actions-def action-obs-def value-written-def enat-less-enat-plusI dest*: *happens-before-change-prefix*[*OF - tsa′ sim*[*symmetric*]] *sync-order-change-prefix*[*OF - sim*[*symmetric*]])

{ **assume** *length E′* ≤ *a′*
thus *P*,*E′′* ⊢ *?ws a′* ≤*hb a′* **using** *a′-E′* **by** *simp* }

{ **fix** *w*
assume *w*: *w ∈ write-actions E′′* (*ad*, *al*) ∈ *action-loc P E′′ w*
and *hbso*: *P*,*E′′* ⊢ *?ws a′* ≤*hb w* ∧ *P*,*E′′* ⊢ *w* ≤*hb a′* ∨ *is-volatile P al* ∧ *P*,*E′′* ⊢ *?ws a′* ≤*so w* ∧ *P*,*E′′* ⊢ *w* ≤*so a′*
show *w = ?ws a′*
proof(*cases w < length E′*)
case *True*
with *is-write-seenD*[*OF E′′ a′ aobs′*] *a′-E′ w hbso* **show** *?thesis*
by(*auto 4 3 elim*!: *write-actions.cases intro*!: *write-actions.intros simp add*: *E′′-def lnth-lappend1 actions-def action-obs-def value-written-def enat-less-enat-plusI dest*: *happens-before-change-prefix*[*OF - tsa*[*unfolded E′′-def*] *sim*] *happens-before-change-prefix*[*OF - tsa′ sim*[*symmetric*]] *sync-order-change-prefix*[*OF - sim*, *simplified*] *sync-order-change-prefix*[*OF - sim*[*symmetric*], *simplified*] *bspec*[**where** *x*=*w*])
next
case *False*
from *hbso* **have** *E′′* ⊢ *w* ≤*a a′* **by**(*auto intro*: *happens-before-into-action-order elim*: *sync-orderE*)
moreover from *w*(*1*) *read* **have** *w* ≠ *a′* **by**(*auto dest*: *read-actions-not-write-actions*)
ultimately have *new-w*: *is-new-action* (*action-obs E′′ w*) **using** *False aobs a′-E′*
by(*cases rule*: *action-orderE*) *auto*
moreover from *hbso a′-E′* **have** *E′′* ⊢ *ws′ a′* ≤*a w*
by(*auto intro*: *happens-before-into-action-order elim*: *sync-orderE*)
hence *new-a′*: *is-new-action* (*action-obs E′′* (*?ws a′*)) **using** *new-w a′-E′*
by(*cases rule*: *action-orderE*) *auto*
ultimately have *w ∈ new-actions-for P E′′* (*ad*, *al*) *?ws a′ ∈ new-actions-for P E′′* (*ad*, *al*)
using *w is-write-seenD*[*OF E′′ a′ aobs′*] *a′-E′*
by(*auto simp add*: *new-actions-for-def actions-def action-obs-def lnth-lappend1 E′′-def enat-less-enat-plusI elim*!: *write-actions.cases*)
thus *?thesis* **by**(*rule new-actions-for-fun*)
qed }
next

assume ¬ $a' <$ *length* E'
hence a'-E': *length* $E' \leq a'$ **by** *simp*
define a **where** $a = a' -$ *length* E'
with a' a'-E' **have** a: *enat* $a <$ *llength* E
by(*simp add*: E''-*def*) (*metis enat-add-mono le-add-diff-inverse plus-enat-simps*(*1*))

from *a-def aobs lnth''* a'-E'
have *aobs*: *action-obs* E $a =$ *NormalAction* (*ReadMem ad al v*)
and *lnth*: *lnth* E $a = (t,$ *NormalAction* (*ReadMem ad al v*))
by(*simp-all add*: E''-*def lnth-lappend2 action-obs-def*)

define E''' **where** $E''' =$ *lappend* (*llist-of* E') (*ltake* (*enat a*) E)
let *?E''* = *lappend* E''' (*LCons* (t, *NormalAction* (*ReadMem ad al v*)) *LNil*)

note *hb* **also**
have *E-unfold1*: $E =$ *lappend* (*ltake* (*enat a*) E) (*ldropn a* E) **by** *simp*
also have *E-unfold2*: *ldropn a* $E =$ *LCons* (t, *NormalAction* (*ReadMem ad al v*)) (*ldropn* (*Suc a*) E)
using *a lnth* **by** (*metis ldropn-Suc-conv-ldropn*)
finally
have *ta-hb-consistent* P (E' @ *list-of* (*ltake* (*enat a*) E))
(*LCons* (t, *NormalAction* (*ReadMem ad al v*)) (*ldropn* (*Suc a*) E))
by(*rule ta-hb-consistent-lappendD2*) *simp*
with *a* a'-E' *a-def* **obtain** w **where** w: $w \in$ *write-actions ?E''*
and *adal-w*: $(ad,\ al) \in$ *action-loc* P *?E''* w
and *written*: *value-written* P *?E''* w $(ad,\ al) = v$
and *hb*: P,*?E''* $\vdash w \leq_{hb} a'$
and *in-between-so*:
$\bigwedge w'$. ⟦ $w' \in$ *write-actions ?E''*; $(ad,\ al) \in$ *action-loc* P *?E''* w';
is-volatile P *al*; P,*?E''* $\vdash w \leq_{so} w'$; P,*?E''* $\vdash w' \leq_{so} a'$ ⟧
$\Longrightarrow w' = w$
and *in-between-hb*:
$\bigwedge w'$. ⟦ $w' \in$ *write-actions ?E''*; $(ad,\ al) \in$ *action-loc* P *?E''* w';
P,*?E''* $\vdash w \leq_{hb} w'$; P,*?E''* $\vdash w' \leq_{hb} a'$ ⟧
$\Longrightarrow w' = w$
by(*auto simp add*: *ta-hb-consistent-LCons length-list-of-conv-the-enat min-def lnth-ltake lappend-llist-of-llist-of*[*symmetric*] E'''-*def lappend-assoc simp del*: *lappend-llist-of-llist-of nth-list-of split*: *if-splits*)

from a' a'-E' *a*
have *eq*: *ltake* (*enat* (*Suc* a')) *?E''* = *ltake* (*enat* (*Suc* a')) E'' (**is** *?lhs* = *?rhs*)
unfolding E''-*def* E'''-*def lappend-assoc*
apply(*subst* (*2*) *E-unfold1*)
apply(*subst E-unfold2*)
apply(*subst* (*1 2*) *ltake-lappend2*)
apply(*simp*)
apply(*rule arg-cong*) **back**
apply(*subst* (*1 2*) *ltake-lappend2*)
apply(*simp add*: *min-def*)
apply (*metis Suc-diff-le a-def le-Suc-eq order-le-less*)
apply(*rule arg-cong*) **back**
apply(*auto simp add*: *min-def a-def*)
apply(*auto simp add*: *eSuc-enat*[*symmetric*] *zero-enat-def*[*symmetric*])
done

hence *sim*: *?lhs* [≈] *?rhs* **by**(*rule eq-into-sim-actions*)
from *tsa* **have** *tsa'*: *thread-start-actions-ok ?E''* **unfolding** *E''-def E'''-def lappend-assoc*
by(*rule thread-start-actions-ok-prefix*)(*subst* (*2*) *E-unfold1*, *simp add*: *E-unfold2*)

from *w a a' a-def a'-E'* **have** *w-a'*: *w* < *Suc a'*
by *cases*(*simp add*: *actions-def E'''-def min-def zero-enat-def eSuc-enat split*: *if-split-asm*)

from *w sim* **have** *w* ∈ *write-actions E''* **by**(*rule write-actions-change-prefix*)(*simp add*: *w-a'*)
moreover
from *adal-w action-loc-change-prefix*[*OF sim*, *of w P*] *w-a'*
have (*ad*, *al*) ∈ *action-loc P E'' w* **by** *simp*
moreover
from *written value-written-change-prefix*[*OF eq*, *of w P*] *w-a'*
have *value-written P E'' w* (*ad*, *al*) = *v* **by** *simp*
moreover
from *hb tsa sim* **have** *P*,*E''* ⊢ *w* ≤*hb a'* **by**(*rule happens-before-change-prefix*)(*simp-all add*: *w-a'*)
moreover {
fix *w'*
assume *w'*: *w'* ∈ *write-actions E''*
and *adal*: (*ad*, *al*) ∈ *action-loc P E'' w'*
and *hbso*: *P*,*E''* ⊢ *w* ≤*hb w'* ∧ *P*,*E''* ⊢ *w'* ≤*hb a'* ∨ *is-volatile P al* ∧ *P*,*E''* ⊢ *w* ≤*so w'* ∧ *P*,*E''* ⊢ *w'* ≤*so a'*
(**is** *?hbso E''*)
from *hbso* **have** *ao*: *E''* ⊢ *w* ≤*a w' E''* ⊢ *w'* ≤*a a'*
by(*auto dest*: *happens-before-into-action-order elim*: *sync-orderE*)
have *w'* = *w*
proof(*cases is-new-action* (*action-obs E'' w'*))
case *True*
hence *w'* ∈ *new-actions-for P E''* (*ad*, *al*) **using** *w' adal* **by**(*simp add*: *new-actions-for-def*)
moreover from *ao True* **have** *is-new-action* (*action-obs E'' w*) **by**(*cases rule*: *action-orderE*) *simp-all*
with ‹*w* ∈ *write-actions E''*› ‹(*ad*, *al*) ∈ *action-loc P E'' w*›
have *w* ∈ *new-actions-for P E''* (*ad*, *al*) **by**(*simp add*: *new-actions-for-def*)
ultimately show *w'* = *w* **by**(*rule new-actions-for-fun*)
next
case *False*
with *ao* **have** *w'* ≤ *a'* **by**(*auto elim*: *action-orderE*)
hence *w'-a*: *enat w'* < *enat* (*Suc a'*) **by** *simp*
with *hbso w-a'* **have** *?hbso ?E''*
by(*auto 4 3 elim*: *happens-before-change-prefix*[*OF - tsa' sim*[*symmetric*]] *sync-order-change-prefix*[*OF - sim*[*symmetric*]] *del*: *disjCI intro*: *disjI1 disjI2*)
moreover from *w'* ‹*w'* ≤ *a'*› *a' a lnth a'-E'* **have** *w'* ∈ *write-actions ?E''*
by(*cases*)(*cases w'* < *a'*, *auto intro*!: *write-actions.intros simp add*: *E'''-def actions-def action-obs-def lnth-lappend min-def zero-enat-def eSuc-enat lnth-ltake a-def E''-def not-le not-less*)
moreover from *adal* ‹*w'* ≤ *a'*› *a a' lnth w' a'-E'* **have** (*ad*, *al*) ∈ *action-loc P ?E'' w'*
by(*cases w'* < *a'*)(*cases w'* < *length E'*, *auto simp add*: *E'''-def action-obs-def lnth-lappend lappend-assoc*[*symmetric*] *min-def lnth-ltake less-trans*[**where** *y*=*enat a*] *a-def E''-def lnth-ltake elim*: *write-actions.cases*)
ultimately show *w'* = *w* **by**(*blast dest*: *in-between-so in-between-hb*)
qed }
ultimately have *?P a' w* **using** *a'-E' lnth* **unfolding** *E''-def a-def* **by**(*simp add*: *lnth-lappend*)
hence *P*: *?P a'* (*Eps* (*?P a'*)) **by**(*rule someI*[**where** *P*=*?P a'*])

```
    from P lnth'' a'-E'
    show ?ws a' ∈ write-actions E''
      and (ad, al) ∈ action-loc P E'' (?ws a')
      and value-written P E'' (?ws a') (ad, al) = v
      and P,E'' ⊢ ?ws a' ≤hb a' by simp-all

    show ¬ P,E'' ⊢ a' ≤hb ?ws a'
    proof
      assume P,E'' ⊢ a' ≤hb ?ws a'
      with ‹P,E'' ⊢ ?ws a' ≤hb a'› have a' = ?ws a'
        by(blast dest: antisymPD[OF antisym-action-order] happens-before-into-action-order)
      with read ‹?ws a' ∈ write-actions E''› show False
        by(auto dest: read-actions-not-write-actions)
    qed

    show ¬ P,E'' ⊢ a' ≤so ?ws a'
    proof
      assume P,E'' ⊢ a' ≤so ?ws a'
      hence E'' ⊢ a' ≤a ?ws a' by(blast elim: sync-orderE)
      with ‹P,E'' ⊢ ?ws a' ≤hb a'› have a' = ?ws a'
        by(blast dest: antisymPD[OF antisym-action-order] happens-before-into-action-order)
      with read ‹?ws a' ∈ write-actions E''› show False
        by(auto dest: read-actions-not-write-actions)
    qed

    fix a''
    assume a'' ∈ write-actions E'' (ad, al) ∈ action-loc P E'' a''
      and P,E'' ⊢ ?ws a' ≤hb a'' ∧ P,E'' ⊢ a'' ≤hb a' ∨
          is-volatile P al ∧ P,E'' ⊢ ?ws a' ≤so a'' ∧ P,E'' ⊢ a'' ≤so a'
    thus a'' = ?ws a' using lnth'' P a'-E' by −(erule disjE, clarsimp+)
  qed
  thus ?ws a' ∈ write-actions E''
    and (ad, al) ∈ action-loc P E'' (?ws a')
    and value-written P E'' (?ws a') (ad, al) = v
    and length E' ≤ a' ⟹ P,E'' ⊢ ?ws a' ≤hb a'
    and ¬ P,E'' ⊢ a' ≤hb ?ws a'
    and is-volatile P al ⟹ ¬ P,E'' ⊢ a' ≤so ?ws a'
    and ⋀a''. ⟦ a'' ∈ write-actions E''; (ad, al) ∈ action-loc P E'' a''; P,E'' ⊢ ?ws a' ≤hb a''; P,E''
⊢ a'' ≤hb a' ⟧ ⟹ a'' = ?ws a'
    and ⋀a''. ⟦ a'' ∈ write-actions E''; (ad, al) ∈ action-loc P E'' a''; is-volatile P al; P,E'' ⊢ ?ws
a' ≤so a''; P,E'' ⊢ a'' ≤so a' ⟧ ⟹ a'' = ?ws a'
    by blast+
  qed(assumption|rule tsa)+
  thus P ⊢ (E'', ?ws) √
    and ⋀n. ⟦ n ∈ read-actions E''; length E' ≤ n ⟧ ⟹ P,E'' ⊢ ?ws n ≤hb n
    by(blast elim: read-actions.cases intro: read-actions.intros)+

  fix n
  assume n < length E'
  thus ?ws n = ws' n by simp
qed

lemma ta-hb-consistent-not-ReadI:
  (⋀t ad al v. (t, NormalAction (ReadMem ad al v)) ∉ lset E) ⟹ ta-hb-consistent P E' E
```

proof(*coinduction arbitrary*: *E' E*)
case (*ta-hb-consistent E' E*)
thus *?case* **by**(*cases E*)(*auto split*: *action.split obs-event.split*, *blast*)
qed

context *jmm-multithreaded* **begin**

definition *complete-hb* :: ('*l*,'*thread-id*,'*x*,'*m*,'*w*) *state* ⇒ ('*thread-id* × ('*addr*, '*thread-id*) *obs-event action*) *list*
⇒ ('*thread-id* × ('*l*, '*thread-id*, '*x*, '*m*, '*w*, ('*addr*, '*thread-id*) *obs-event action*) *thread-action*) *llist*
where
complete-hb s E = *unfold-llist*
(λ(*s*, *E*). ∀ *t ta s'*. ¬ *s* −*t*▷*ta*→ *s'*)
(λ(*s*, *E*). *fst* (*SOME* ((*t*, *ta*), *s'*). *s* −*t*▷*ta*→ *s'* ∧ *ta-hb-consistent P E* (*llist-of* (*map* (*Pair t*) ⦃*ta*⦄$_o$))))
(λ(*s*, *E*). *let* ((*t*, *ta*), *s'*) = *SOME* ((*t*, *ta*), *s'*). *s* −*t*▷*ta*→ *s'* ∧ *ta-hb-consistent P E* (*llist-of* (*map* (*Pair t*) ⦃*ta*⦄$_o$))
in (*s'*, *E* @ *map* (*Pair t*) ⦃*ta*⦄$_o$))
(*s*, *E*)

definition *hb-completion* ::
('*l*, '*thread-id*, '*x*, '*m*, '*w*) *state* ⇒ ('*thread-id* × ('*addr*, '*thread-id*) *obs-event action*) *list* ⇒ *bool*
where
hb-completion s E ⟷
(∀ *ttas s' t x ta x' m' i*.
s −▷*ttas*→* *s'* ⟶
non-speculative P (*w-values P* (λ-. {}) (*map snd E*)) (*llist-of* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas*))) ⟶
thr s' t = ⌊(*x*, *no-wait-locks*)⌋ ⟶ *t* ⊢ (*x*, *shr s'*) −*ta*→ (*x'*, *m'*) ⟶ *actions-ok s' t ta* ⟶
non-speculative P (*w-values P* (*w-values P* (λ-. {}) (*map snd E*)) (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas*))) (*llist-of* (*take i* ⦃*ta*⦄$_o$)) ⟶
(∃ *ta' x'' m''*. *t* ⊢ (*x*, *shr s'*) −*ta'*→ (*x''*, *m''*) ∧ *actions-ok s' t ta'* ∧
take i ⦃*ta'*⦄$_o$ = *take i* ⦃*ta*⦄$_o$ ∧
ta-hb-consistent P
(*E* @ *concat* (*map* (λ(*t*, *ta*). *map* (*Pair t*) ⦃*ta*⦄$_o$) *ttas*) @ *map* (*Pair t*) (*take i* ⦃*ta*⦄$_o$))
(*llist-of* (*map* (*Pair t*) (*drop i* ⦃*ta'*⦄$_o$))) ∧
(*i* < *length* ⦃*ta*⦄$_o$ ⟶ *i* < *length* ⦃*ta'*⦄$_o$) ∧
(*if* ∃ *ad al v*. ⦃*ta*⦄$_o$! *i* = *NormalAction* (*ReadMem ad al v*) *then sim-action else* (=)) (⦃*ta*⦄$_o$! *i*) (⦃*ta'*⦄$_o$! *i*)))

lemma *hb-completionD*:
⟦ *hb-completion s E*; *s* −▷*ttas*→* *s'*;
non-speculative P (*w-values P* (λ-. {}) (*map snd E*)) (*llist-of* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas*)));
thr s' t = ⌊(*x*, *no-wait-locks*)⌋; *t* ⊢ (*x*, *shr s'*) −*ta*→ (*x'*, *m'*); *actions-ok s' t ta*;
non-speculative P (*w-values P* (*w-values P* (λ-. {}) (*map snd E*)) (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas*))) (*llist-of* (*take i* ⦃*ta*⦄$_o$)) ⟧
⟹ ∃ *ta' x'' m''*. *t* ⊢ (*x*, *shr s'*) −*ta'*→ (*x''*, *m''*) ∧ *actions-ok s' t ta'* ∧
take i ⦃*ta'*⦄$_o$ = *take i* ⦃*ta*⦄$_o$ ∧
ta-hb-consistent P (*E* @ *concat* (*map* (λ(*t*, *ta*). *map* (*Pair t*) ⦃*ta*⦄$_o$) *ttas*) @ *map* (*Pair t*) (*take i* ⦃*ta*⦄$_o$))
(*llist-of* (*map* (*Pair t*) (*drop i* ⦃*ta'*⦄$_o$))) ∧
(*i* < *length* ⦃*ta*⦄$_o$ ⟶ *i* < *length* ⦃*ta'*⦄$_o$) ∧

(*if* ∃ *ad al v*. ⦃*ta*⦄$_o$! *i* = *NormalAction* (*ReadMem ad al v*) *then sim-action else* (=)) (⦃*ta*⦄$_o$! *i*) (⦃*ta'*⦄$_o$! *i*)
unfolding *hb-completion-def* **by** *blast*

lemma *hb-completionI* [*intro?*]:
(⋀*ttas s' t x ta x' m' i*.
⟦ *s* −▷*ttas*→∗ *s'*; *non-speculative P* (*w-values P* (λ-. {}) (*map snd E*)) (*llist-of* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas*)));
thr s' t = ⌊(*x*, *no-wait-locks*)⌋; *t* ⊢ (*x*, *shr s'*) −*ta*→ (*x'*, *m'*); *actions-ok s' t ta*;
non-speculative P (*w-values P* (*w-values P* (λ-. {}) (*map snd E*)) (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas*))) (*llist-of* (*take i* ⦃*ta*⦄$_o$)) ⟧
⟹ ∃ *ta' x'' m''*. *t* ⊢ (*x*, *shr s'*) −*ta'*→ (*x''*, *m''*) ∧ *actions-ok s' t ta'* ∧ *take i* ⦃*ta'*⦄$_o$ = *take i* ⦃*ta*⦄$_o$ ∧
ta-hb-consistent P (*E* @ *concat* (*map* (λ(*t*, *ta*). *map* (*Pair t*) ⦃*ta*⦄$_o$) *ttas*) @ *map* (*Pair t*) (*take i* ⦃*ta*⦄$_o$)) (*llist-of* (*map* (*Pair t*) (*drop i* ⦃*ta'*⦄$_o$))) ∧
(*i* < *length* ⦃*ta*⦄$_o$ ⟶ *i* < *length* ⦃*ta'*⦄$_o$) ∧
(*if* ∃ *ad al v*. ⦃*ta*⦄$_o$! *i* = *NormalAction* (*ReadMem ad al v*) *then sim-action else* (=)) (⦃*ta*⦄$_o$! *i*) (⦃*ta'*⦄$_o$! *i*))
⟹ *hb-completion s E*
unfolding *hb-completion-def* **by** *blast*

lemma *hb-completion-shift*:
assumes *hb-c*: *hb-completion s E*
and *τRed*: *s* −▷*ttas*→∗ *s'*
and *sc*: *non-speculative P* (*w-values P* (λ-. {}) (*map snd E*)) (*llist-of* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas*)))
(**is** *non-speculative* - *?vs* -)
shows *hb-completion s'* (*E* @ (*concat* (*map* (λ(*t*, *ta*). *map* (*Pair t*) ⦃*ta*⦄$_o$) *ttas*)))
(**is** *hb-completion* - *?E*)
proof(*rule hb-completionI*)
fix *ttas' s'' t x ta x' m' i*
assume *τRed'*: *s'* −▷*ttas'*→∗ *s''*
and *sc'*: *non-speculative P* (*w-values P* (λ-. {}) (*map snd ?E*)) (*llist-of* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas'*)))
and *red*: *thr s'' t* = ⌊(*x*, *no-wait-locks*)⌋ *t* ⊢ ⟨*x*, *shr s''*⟩ −*ta*→ ⟨*x'*, *m'*⟩ *actions-ok s'' t ta*
and *ns*: *non-speculative P* (*w-values P* (*w-values P* (λ-. {}) (*map snd ?E*)) (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas'*))) (*llist-of* (*take i* ⦃*ta*⦄$_o$))
from *τRed τRed'* **have** *s* −▷*ttas* @ *ttas'*→∗ *s''* **unfolding** *RedT-def* **by**(*rule rtrancl3p-trans*)
moreover from *sc sc'* **have** *non-speculative P ?vs* (*llist-of* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) (*ttas* @ *ttas'*))))
unfolding *map-append concat-append lappend-llist-of-llist-of*[*symmetric*] *map-concat*
by(*simp add*: *non-speculative-lappend o-def split-def del*: *lappend-llist-of-llist-of*)
ultimately
show ∃ *ta' x'' m''*. *t* ⊢ ⟨*x*, *shr s''*⟩ −*ta'*→ ⟨*x''*, *m''*⟩ ∧ *actions-ok s'' t ta'* ∧ *take i* ⦃*ta'*⦄$_o$ = *take i* ⦃*ta*⦄$_o$ ∧
ta-hb-consistent P (*?E* @ *concat* (*map* (λ(*t*, *ta*). *map* (*Pair t*) ⦃*ta*⦄$_o$) *ttas'*) @ *map* (*Pair t*) (*take i* ⦃*ta*⦄$_o$)) (*llist-of* (*map* (*Pair t*) (*drop i* ⦃*ta'*⦄$_o$))) ∧
(*i* < *length* ⦃*ta*⦄$_o$ ⟶ *i* < *length* ⦃*ta'*⦄$_o$) ∧
(*if* ∃ *ad al v*. ⦃*ta*⦄$_o$! *i* = *NormalAction* (*ReadMem ad al v*) *then sim-action else* (=)) (⦃*ta*⦄$_o$! *i*) (⦃*ta'*⦄$_o$! *i*)
using *red ns* **unfolding** *append-assoc*
apply(*subst* (*2*) *append-assoc*[*symmetric*])
unfolding *concat-append*[*symmetric*] *map-append*[*symmetric*] *foldr-append*[*symmetric*]

```
    by(rule hb-completionD[OF hb-c])(simp-all add: map-concat o-def split-def)
qed

lemma hb-completion-shift1:
  assumes hb-c: hb-completion s E
  and Red: s −t▷ta→ s′
  and sc: non-speculative P (w-values P (λ-. {}) (map snd E)) (llist-of {|ta|}_o)
  shows hb-completion s′ (E @ map (Pair t) {|ta|}_o)
using hb-completion-shift[OF hb-c, of [(t, ta)] s′] Red sc
by(simp add: RedT-def rtrancl3p-Cons rtrancl3p-Nil del: split-paired-Ex)

lemma complete-hb-in-Runs:
  assumes hb-c: hb-completion s E
  and ta-hb-consistent-convert-RA: ⋀t E ln. ta-hb-consistent P E (llist-of (map (Pair t) (convert-RA
ln)))
  shows mthr.Runs s (complete-hb s E)
using hb-c
proof(coinduction arbitrary: s E)
  case (Runs s E)
  let ?P = λ((t, ta), s′). s −t▷ta→ s′ ∧ ta-hb-consistent P E (llist-of (map (Pair t) {|ta|}_o))
  show ?case
  proof(cases ∃ t ta s′. s −t▷ta→ s′)
    case False
    then have ?Stuck by(simp add: complete-hb-def)
    thus ?thesis ..
  next
    case True
    let ?t = fst (fst (Eps ?P)) and ?ta = snd (fst (Eps ?P)) and ?s′ = snd (Eps ?P)
    from True obtain t ta s′ where red: s −t▷ta→ s′ by blast
    hence ∃ x. ?P x
    proof(cases)
      case (redT-normal x x′ m′)
      from hb-completionD[OF Runs - - ‹thr s t = ⌊(x, no-wait-locks)⌋› ‹t ⊢ ⟨x, shr s⟩ −ta→ ⟨x′, m′⟩›
‹actions-ok s t ta›, of [] 0]
      obtain ta′ x″ m″ where t ⊢ ⟨x, shr s⟩ −ta′→ ⟨x″, m″⟩
        and actions-ok s t ta′ ta-hb-consistent P E (llist-of (map (Pair t) {|ta′|}_o))
        by fastforce
      moreover obtain ws′ where redT-updWs t (wset s) {|ta′|}_w ws′ by (metis redT-updWs-total)
      ultimately show ?thesis using ‹thr s t = ⌊(x, no-wait-locks)⌋›
        by(cases ta′)(auto intro!: exI redT.redT-normal)
    next
      case (redT-acquire x n ln)
      thus ?thesis using ta-hb-consistent-convert-RA[of E t ln]
        by(auto intro!: exI redT.redT-acquire)
    qed
    hence ?P (Eps ?P) by(rule someI-ex)
    hence red: s −?t▷?ta→ ?s′
      and hb: ta-hb-consistent P E (llist-of (map (Pair ?t) {|?ta|}_o))
      by(simp-all add: split-beta)
    moreover
    from ta-hb-consistent-into-non-speculative[OF hb]
    have non-speculative P (w-values P (λ-. {}) (map snd E)) (llist-of {|?ta|}_o) by(simp add: o-def)
    with Runs red have hb-completion ?s′ (E @ map (Pair ?t) {|?ta|}_o) by(rule hb-completion-shift1)
    ultimately have ?Step using True
```

```
    unfolding complete-hb-def by(fastforce simp del: split-paired-Ex simp add: split-def)
   thus ?thesis ..
 qed
qed

lemma complete-hb-ta-hb-consistent:
 assumes hb-completion s E
 and ta-hb-consistent-convert-RA: ⋀E t ln. ta-hb-consistent P E (llist-of (map (Pair t) (convert-RA ln)))
 shows ta-hb-consistent P E (lconcat (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) (complete-hb s E)))
 (is ta-hb-consistent - - (?obs (complete-hb s E)))
proof −
 define obs a where obs = ?obs (complete-hb s E) and a = complete-hb s E
  with ‹hb-completion s E› have ∃s. hb-completion s E ∧ obs = ?obs (complete-hb s E) ∧ a = complete-hb s E by blast
 moreover have wf (inv-image {(m, n). m < n} (llength ∘ ltakeWhile (λ(t, ta). ⦃ta⦄o = [])))
  (is wf ?R) by(rule wf-inv-image)(rule wellorder-class.wf)
 ultimately show ta-hb-consistent P E obs
 proof(coinduct E obs a rule: ta-hb-consistent-coinduct-append-wf)
  case (ta-hb-consistent E obs a)
  then obtain s where hb-c: hb-completion s E
   and obs: obs = lconcat (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) (complete-hb s E))
   and a: a = complete-hb s E
   by blast
  let ?P = λ((t, ta), s'). s −t▷ta→ s' ∧ ta-hb-consistent P E (llist-of (map (Pair t) ⦃ta⦄o))
  show ?case
  proof(cases ∃t ta s'. s −t▷ta→ s')
   case False
   with obs have ?LNil by(simp add: complete-hb-def)
   thus ?thesis ..
  next
   case True
   let ?t = fst (fst (Eps ?P)) and ?ta = snd (fst (Eps ?P)) and ?s' = snd (Eps ?P)
   from True obtain t ta s' where red: s −t▷ta→ s' by blast
   hence ∃x. ?P x
   proof(cases)
    case (redT-normal x x' m')
    from hb-completionD[OF hb-c - - ‹thr s t = ⌊(x, no-wait-locks)⌋› ‹t ⊢ ⟨x, shr s⟩ −ta→ ⟨x', m'⟩›
‹actions-ok s t ta›, of [] 0]
     obtain ta' x'' m'' where t ⊢ ⟨x, shr s⟩ −ta'→ ⟨x'', m''⟩
      and actions-ok s t ta' ta-hb-consistent P E (llist-of (map (Pair t) ⦃ta'⦄o))
      by fastforce
     moreover obtain ws' where redT-updWs t (wset s) ⦃ta'⦄w ws' by (metis redT-updWs-total)
     ultimately show ?thesis using ‹thr s t = ⌊(x, no-wait-locks)⌋›
      by(cases ta')(auto intro!: exI redT.redT-normal)
   next
    case (redT-acquire x n ln)
    thus ?thesis using ta-hb-consistent-convert-RA[of E t ln]
     by(auto intro!: exI redT.redT-acquire)
   qed
   hence ?P (Eps ?P) by(rule someI-ex)
   hence red': s −?t▷?ta→ ?s'
    and hb: ta-hb-consistent P E (llist-of (map (Pair ?t) ⦃?ta⦄o))
```

```
      by(simp-all add: split-beta)
    moreover
    from ta-hb-consistent-into-non-speculative[OF hb]
    have non-speculative P (w-values P (λ-. {}) (map snd E)) (llist-of {|?ta|}_o) by(simp add: o-def)
    with hb-c red' have hb-c': hb-completion ?s' (E @ map (Pair ?t) {|?ta|}_o)
      by(rule hb-completion-shift1)
    show ?thesis
    proof(cases lnull obs)
      case True thus ?thesis unfolding lnull-def by simp
    next
      case False
      have eq: (∀ t ta s'. ¬ s -t▷ta→ s') = False using True by auto
      { assume {|?ta|}_o = []
        moreover from obs False
        have lfinite (ltakeWhile (λ(t, ta). {|ta|}_o = []) (complete-hb s E))
          unfolding lfinite-ltakeWhile by(fastforce simp add: split-def lconcat-eq-LNil)
        ultimately have (complete-hb ?s' (E @ map (Pair ?t) {|?ta|}_o), a) ∈ ?R
          using red unfolding a complete-hb-def
          apply(subst (2) unfold-llist.code)
          apply(subst (asm) unfold-llist.code)
          apply(auto simp add: split-beta simp del: split-paired-Ex split-paired-All split: if-split-asm)
          apply(auto simp add: lfinite-eq-range-llist-of)
          done }
      hence ?lappend using red hb hb-c' unfolding obs complete-hb-def
        apply(subst unfold-llist.code)
        apply(simp add: split-beta eq del: split-paired-Ex split-paired-All split del: if-split)
       apply(intro exI conjI impI refl disjI1|rule refl|assumption|simp-all add: llist-of-eq-LNil-conv)+
        done
      thus ?thesis ..
    qed
  qed
 qed
qed

lemma hb-completion-Runs:
  assumes hb-completion s E
  and ⋀E t ln. ta-hb-consistent P E (llist-of (map (Pair t) (convert-RA ln)))
  shows ∃ttas. mthr.Runs s ttas ∧ ta-hb-consistent P E (lconcat (lmap (λ(t, ta). llist-of (map (Pair
t) {|ta|}_o)) ttas))
using complete-hb-in-Runs[OF assms] complete-hb-ta-hb-consistent[OF assms]
by blast

end

end
```

8.10 Locales for heap operations with set of allocated addresses

```
theory JMM-Heap
imports
  ../Common/WellForm
  SC-Completion
  HB-Completion
```

begin

definition *w-addrs* :: (*'addr* × *addr-loc* ⇒ *'addr val set*) ⇒ *'addr set*
where *w-addrs vs* = {*a*. ∃ *adal*. *Addr a* ∈ *vs adal*}

lemma *w-addrs-empty* [*simp*]: *w-addrs* (λ-. {}) = {}
by(*simp add*: *w-addrs-def*)

locale *allocated-heap-base* = *heap-base* +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
fixes *allocated* :: *'heap* ⇒ *'addr set*

locale *allocated-heap* =
allocated-heap-base +
heap +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *allocated* :: *'heap* ⇒ *'addr set*
and *P* :: *'m prog*

assumes *allocated-empty*: *allocated empty-heap* = {}
and *allocate-allocatedD*:
(*h'*, *a*) ∈ *allocate h hT* ⟹ *allocated h'* = *insert a* (*allocated h*) ∧ *a* ∉ *allocated h*
and *heap-write-allocated-same*:
heap-write h a al v h' ⟹ *allocated h'* = *allocated h*
begin

lemma *allocate-allocated-mono*: (*h'*, *a*) ∈ *allocate h C* ⟹ *allocated h* ⊆ *allocated h'*
by(*simp-all add*: *allocate-allocatedD*)

lemma
shows *start-addrs-allocated*: *allocated start-heap* = *set start-addrs*
and *distinct-start-addrs'*: *distinct start-addrs*
proof −
{ fix *h ads b* **and** *xs* :: *cname list*
let *?start-addrs h ads b xs* = *fst* (*snd* (*foldl create-initial-object* (*h*, *ads*, *b*) *xs*))
let *?start-heap h ads b xs* = *fst* (*foldl create-initial-object* (*h*, *ads*, *b*) *xs*)
assume *allocated h* = *set ads*
hence *allocated* (*?start-heap h ads b xs*) = *set* (*?start-addrs h ads b xs*) ∧
(*distinct ads* ⟶ *distinct* (*?start-addrs h ads b xs*))
(**is** *?concl xs h ads b*)

```
    proof(induct xs arbitrary: h ads b)
      case Nil thus ?case by auto
    next
      case (Cons x xs)
      note ads = ‹allocated h = set ads›
      show ?case
      proof(cases b ∧ allocate h (Class-type x) ≠ {})
        case False thus ?thesis using ads
          by(simp add: create-initial-object-simps zip-append1)
      next
        case [simp]: True
        then obtain h' a'
          where h'a': (SOME ha. ha ∈ allocate h (Class-type x)) = (h', a')
          and new-obj: (h', a') ∈ allocate h (Class-type x)
          by(cases (SOME ha. ha ∈ allocate h (Class-type x)))(auto simp del: True dest: allocate-Eps)

        from new-obj have allocated h' = insert a' (allocated h) a' ∉ allocated h
          by(auto dest: allocate-allocatedD)
        with ads have allocated h' = set (ads @ [a']) by auto
        hence ?concl xs h' (ads @ [a']) True by(rule Cons)
        moreover have a' ∉ set ads using ‹a' ∉ allocated h› ads by blast
        ultimately show ?thesis by(simp add: create-initial-object-simps new-obj h'a')
      qed
    qed }
  from this[of empty-heap [] True initialization-list]
  show allocated start-heap = set start-addrs
    and distinct-start-addrs: distinct start-addrs
    unfolding start-heap-def start-addrs-def start-heap-data-def
    by(auto simp add: allocated-empty)
qed

lemma w-addrs-start-heap-obs: w-addrs (w-values P vs (map NormalAction start-heap-obs)) ⊆ w-addrs
vs
proof −
  { fix xs
    let ?NewObj = λa C. NewHeapElem a (Class-type C) :: ('addr, 'thread-id) obs-event
    let ?start-heap-obs xs = map (λ(C, a). ?NewObj a C) xs
    have w-addrs (w-values P vs (map NormalAction (?start-heap-obs xs))) ⊆ w-addrs vs
      (is ?concl xs)
    proof(induct xs arbitrary: vs)
      case Nil thus ?case by simp
    next
      case (Cons x xs)
      have w-addrs (w-values P vs (map NormalAction (map (λ(C, a). ?NewObj a C) (x # xs))))
        = w-addrs (w-values P (w-value P vs (NormalAction (?NewObj (snd x) (fst x)))) (map Nor-
malAction (map (λ(C, a). ?NewObj a C) xs)))
        by(simp add: split-beta)
      also have ... ⊆ w-addrs (w-value P vs (NormalAction (?NewObj (snd x) (fst x)))) by(rule Cons)
      also have ... ⊆ w-addrs vs
        by(auto simp add: w-addrs-def default-val-not-Addr Addr-not-default-val)
      finally show ?case .
    qed }
  thus ?thesis by(simp add: start-heap-obs-def)
qed
```

end

context *heap-base* **begin**

lemma *addr-loc-default-conf*:
  $P \vdash$ *class-type-of CTn has F*:*T* (*fm*) *in C*
  $\Longrightarrow$ *P*,*h* $\vdash$ *addr-loc-default P CTn* (*CField C F*) :$\leq$ *T*
apply(*cases CTn*)
 apply *simp*
apply(*frule has-field-decl-above*)
apply *simp*
done

definition *vs-conf* :: *'m prog* $\Rightarrow$ *'heap* $\Rightarrow$ (*'addr* $\times$ *addr-loc* $\Rightarrow$ *'addr val set*) $\Rightarrow$ *bool*
where *vs-conf P h vs* $\longleftrightarrow$ ($\forall$ *ad al v*. *v* $\in$ *vs* (*ad*, *al*) $\longrightarrow$ ($\exists$ *T*. *P*,*h* $\vdash$ *ad*@*al* : *T* $\wedge$ *P*,*h* $\vdash$ *v* :$\leq$ *T*))

lemma *vs-confI*:
  ($\bigwedge$*ad al v*. *v* $\in$ *vs* (*ad*, *al*) $\Longrightarrow$ $\exists$ *T*. *P*,*h* $\vdash$ *ad*@*al* : *T* $\wedge$ *P*,*h* $\vdash$ *v* :$\leq$ *T*) $\Longrightarrow$ *vs-conf P h vs*
unfolding *vs-conf-def* **by** *blast*

lemma *vs-confD*:
  $[\![$ *vs-conf P h vs*; *v* $\in$ *vs* (*ad*, *al*) $]\!]$ $\Longrightarrow$ $\exists$ *T*. *P*,*h* $\vdash$ *ad*@*al* : *T* $\wedge$ *P*,*h* $\vdash$ *v* :$\leq$ *T*
unfolding *vs-conf-def* **by** *blast*

lemma *vs-conf-insert-iff*:
  vs-conf P h (*vs*((*ad*, *al*) := *insert v* (*vs* (*ad*, *al*))))
  $\longleftrightarrow$ *vs-conf P h vs* $\wedge$ ($\exists$ *T*. *P*,*h* $\vdash$ *ad*@*al* : *T* $\wedge$ *P*,*h* $\vdash$ *v* :$\leq$ *T*)
by(*auto 4 3 elim*: *vs-confD intro*: *vs-confI split*: *if-split-asm*)

end

context *heap* **begin**

lemma *vs-conf-hext*: $[\![$ *vs-conf P h vs*; *h* $\trianglelefteq$ *h'* $]\!]$ $\Longrightarrow$ *vs-conf P h' vs*
by(*blast intro*!: *vs-confI intro*: *conf-hext addr-loc-type-hext-mono dest*: *vs-confD*)

lemma *vs-conf-allocate*:
  $[\![$ *vs-conf P h vs*; (*h'*, *a*) $\in$ *allocate h hT*; *is-htype P hT* $]\!]$
  $\Longrightarrow$ *vs-conf P h'* (*w-value P vs* (*NormalAction* (*NewHeapElem a hT*)))
apply(*drule vs-conf-hext*)
 apply(*erule hext-allocate*)
apply(*auto intro*!: *vs-confI simp add*: *addr-locs-def split*: *if-split-asm htype.split-asm*)
apply(*auto 3 3 intro*: *addr-loc-type.intros defval-conf dest*: *allocate-SomeD elim*: *has-field-is-class vs-confD*)
apply(*rule exI conjI addr-loc-type.intros*|*drule allocate-SomeD*|*erule has-field-is-class*|*simp*)+
done

end

heap-read-typeable must not be defined in *heap-conf-base* (where it should be) because this would lead to duplicate definitions of *heap-read-typeable* in contexts where *heap-conf-base* is imported twice with different parameters, e.g., *P* and *J2JVM P* in *J-JVM-heap-conf-read*.

context *heap-base* **begin**

definition *heap-read-typeable* :: (*'heap* ⇒ *bool*) ⇒ *'m prog* ⇒ *bool*
where *heap-read-typeable hconf P* ⟷ (∀ *h ad al v T*. *hconf h* ⟶ *P,h* ⊢ *ad@al* : *T* ⟶ *P,h* ⊢ *v* :≤ *T* ⟶ *heap-read h ad al v*)

lemma *heap-read-typeableI*:
(⋀*h ad al v T*. ⟦ *P,h* ⊢ *ad@al* : *T*; *P,h* ⊢ *v* :≤ *T*; *hconf h* ⟧ ⟹ *heap-read h ad al v*) ⟹ *heap-read-typeable hconf P*
unfolding *heap-read-typeable-def* **by** *blast*

lemma *heap-read-typeableD*:
⟦ *heap-read-typeable hconf P*; *P,h* ⊢ *ad@al* : *T*; *P,h* ⊢ *v* :≤ *T*; *hconf h* ⟧ ⟹ *heap-read h ad al v*
unfolding *heap-read-typeable-def* **by** *blast*

end

context *heap-base* **begin**

definition *heap-read-typed* :: *'m prog* ⇒ *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
where *heap-read-typed P h ad al v* ⟷ *heap-read h ad al v* ∧ (∀ *T*. *P,h* ⊢ *ad@al* : *T* ⟶ *P,h* ⊢ *v* :≤ *T*)

lemma *heap-read-typedI*:
⟦ *heap-read h ad al v*; ⋀*T*. *P,h* ⊢ *ad@al* : *T* ⟹ *P,h* ⊢ *v* :≤ *T* ⟧ ⟹ *heap-read-typed P h ad al v*
unfolding *heap-read-typed-def* **by** *blast*

lemma *heap-read-typed-into-heap-read*:
heap-read-typed P h ad al v ⟹ *heap-read h ad al v*
unfolding *heap-read-typed-def* **by** *blast*

lemma *heap-read-typed-typed*:
⟦ *heap-read-typed P h ad al v*; *P,h* ⊢ *ad@al* : *T* ⟧ ⟹ *P,h* ⊢ *v* :≤ *T*
unfolding *heap-read-typed-def* **by** *blast*

end

context *heap-conf* **begin**

lemma *heap-conf-read-heap-read-typed*:
heap-conf-read addr2thread-id thread-id2addr empty-heap allocate typeof-addr (*heap-read-typed P*) *heap-write hconf P*
proof
fix *h a al v T*
assume *heap-read-typed P h a al v P,h* ⊢ *a@al* : *T*
thus *P,h* ⊢ *v* :≤ *T* **by**(*rule heap-read-typed-typed*)
qed

end

context *heap* **begin**

lemma *start-addrs-dom-w-values*:
assumes *wf*: *wf-syscls P*
and *a*: *a* ∈ *set start-addrs*
and *adal*: *P,start-heap* ⊢ *a@al* : *T*

shows *w-values P* (λ-. {}) (*map NormalAction start-heap-obs*) (*a*, *al*) ≠ {}
proof −
from *a* **obtain** *CTn* **where** *CTn*: *NewHeapElem a CTn* ∈ *set start-heap-obs*
unfolding *in-set-start-addrs-conv-NewHeapElem* **..**
then obtain *obs obs'* **where** *obs*: *start-heap-obs* = *obs* @ *NewHeapElem a CTn* # *obs'* **by**(*auto dest*: *split-list*)
have *w-value P* (*w-values P* (λ-. {}) (*map NormalAction obs*)) (*NormalAction* (*NewHeapElem a CTn*)) (*a*, *al*) ≠ {}
proof(*cases CTn*)
case [*simp*]: (*Class-type C*)
with *wf CTn* **have** *typeof-addr start-heap a* = ⌊*Class-type C*⌋
by(*auto intro*: *NewHeapElem-start-heap-obsD*)
with *adal* **show** *?thesis* **by** *cases auto*
next
case [*simp*]: (*Array-type T n*)
with *wf CTn* **have** *typeof-addr start-heap a* = ⌊*Array-type T n*⌋
by(*auto dest*: *NewHeapElem-start-heap-obsD*)
with *adal* **show** *?thesis* **by** *cases*(*auto dest*: *has-field-decl-above*)
qed
moreover have *w-value P* (*w-values P* (λ-. {}) (*map NormalAction obs*)) (*NormalAction* (*NewHeapElem a CTn* :: (*'addr*, *'thread-id*) *obs-event*))
(*a*, *al*) ⊆ *w-values P* (λ-. {}) (*map NormalAction start-heap-obs*) (*a*, *al*)
by(*simp add*: *obs del*: *w-value.simps*)(*rule w-values-mono*)
ultimately show *?thesis* **by** *blast*
qed

end

end

8.11 Combination of locales for heap operations and interleaving

theory *JMM-Framework*
imports
JMM-Heap
../Framework/FWInitFinLift
../Common/WellForm
begin

lemma *enat-plus-eq-enat-conv*: — Move to Extended_Nat
enat m + *n* = *enat k* ⟷ *k* ≥ *m* ∧ *n* = *enat* (*k* − *m*)
by(*cases n*) *auto*

declare *convert-new-thread-action-id* [*simp*]

context *heap* **begin**

lemma *init-fin-lift-state-start-state*:
init-fin-lift-state s (*start-state f P C M vs*) = *start-state* (λ*C M Ts T meth vs*. (*s*, *f C M Ts T meth vs*)) *P C M vs*
by(*simp add*: *start-state-def init-fin-lift-state-def split-beta fun-eq-iff*)

lemma *non-speculative-start-heap-obs*:
non-speculative P vs (*llist-of* (*map snd* (*lift-start-obs start-tid start-heap-obs*)))
apply(*rule non-speculative-nthI*)
using *start-heap-obs-not-Read*
by(*clarsimp simp add*: *lift-start-obs-def lnth-LCons o-def eSuc-enat*[*symmetric*] *in-set-conv-nth split*: *nat.split-asm*)

lemma *ta-seq-consist-start-heap-obs*:
ta-seq-consist P Map.empty (*llist-of* (*map snd* (*lift-start-obs start-tid start-heap-obs*)))
using *start-heap-obs-not-Read*
by(*auto intro*: *ta-seq-consist-nthI simp add*: *lift-start-obs-def o-def lnth-LCons in-set-conv-nth split*: *nat.split-asm*)

end

context *allocated-heap* **begin**

lemma *w-addrs-lift-start-heap-obs*:
w-addrs (*w-values P vs* (*map snd* (*lift-start-obs start-tid start-heap-obs*))) ⊆ *w-addrs vs*
by(*simp add*: *lift-start-obs-def o-def w-addrs-start-heap-obs*)

end

context *heap* **begin**

lemma *w-values-start-heap-obs-typeable*:
assumes *wf*: *wf-syscls P*
and *mrws*: *v* ∈ *w-values P* (λ-. {}) (*map snd* (*lift-start-obs start-tid start-heap-obs*)) (*ad*, *al*)
shows ∃ *T*. *P*,*start-heap* ⊢ *ad*@*al* : *T* ∧ *P*,*start-heap* ⊢ *v* :≤ *T*
proof −
from *in-w-valuesD*[*OF mrws*]
obtain *obs*′ *wa obs*″
where *eq*: *map snd* (*lift-start-obs start-tid start-heap-obs*) = *obs*′ @ *wa* # *obs*″
and *is-write-action wa*
and *adal*: (*ad*, *al*) ∈ *action-loc-aux P wa*
and *vwa*: *value-written-aux P wa al* = *v*
by *blast*
from ‹*is-write-action wa*› **show** *?thesis*
proof *cases*
case (*WriteMem ad*′ *al*′ *v*′)
with *vwa adal eq* **have** *WriteMem ad al v* ∈ *set start-heap-obs*
by(*auto simp add*: *map-eq-append-conv Cons-eq-append-conv lift-start-obs-def*)
thus *?thesis* **by**(*rule start-heap-write-typeable*)
next
case (*NewHeapElem ad*′ *hT*)
with *vwa adal eq* **have** *NewHeapElem ad hT* ∈ *set start-heap-obs*
by(*auto simp add*: *map-eq-append-conv Cons-eq-append-conv lift-start-obs-def*)
hence *typeof-addr start-heap ad* = ⌊*hT*⌋
by(*rule NewHeapElem-start-heap-obsD*[*OF wf*])
thus *?thesis* **using** *adal vwa NewHeapElem*
apply(*cases hT*)
apply(*auto intro*!: *addr-loc-type.intros dest*: *has-field-decl-above*)
apply(*frule has-field-decl-above*)
apply(*auto intro*!: *addr-loc-type.intros dest*: *has-field-decl-above*)

```
    done
  qed
qed

lemma start-state-vs-conf:
  wf-syscls P ⟹ vs-conf P start-heap (w-values P (λ-. {}) (map snd (lift-start-obs start-tid start-heap-obs)))
by(rule vs-confI)(rule w-values-start-heap-obs-typeable)

end
```

8.11.1 JMM traces for Jinja semantics

```
context multithreaded-base begin

inductive-set ℰ :: ('l,'t,'x,'m,'w) state ⇒ ('t × 'o) llist set
  for σ :: ('l,'t,'x,'m,'w) state
where
  mthr.Runs σ E'
  ⟹ lconcat (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) E') ∈ ℰ σ

lemma actions-ℰE-aux:
  fixes σ E'
  defines E == lconcat (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) E')
  assumes mthr: mthr.Runs σ E'
  and a: enat a < llength E
  obtains m n t ta
  where lnth E a = (t, ⦃ta⦄o ! n)
  and n < length ⦃ta⦄o and enat m < llength E'
  and a = (∑ i<m. length ⦃snd (lnth E' i)⦄o) + n
  and lnth E' m = (t, ta)
proof −
  from lnth-lconcat-conv[OF a[unfolded E-def], folded E-def]
  obtain m n
    where lnth E a = lnth (lnth (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) E') m) n
    and enat n < llength (lnth (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) E') m)
    and enat m < llength (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) E')
    and enat a = (∑ i<m. llength (lnth (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) E') i)) + enat
n
    by blast
  moreover
  obtain t ta where lnth E' m = (t, ta) by(cases lnth E' m)
  ultimately have E-a: lnth E a = (t, ⦃ta⦄o ! n)
    and n: n < length ⦃ta⦄o
    and m: enat m < llength E'
    and a: enat a = (∑ i<m. llength (lnth (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) E') i)) +
enat n
    by(simp-all)
  note a
  also have (∑ i<m. llength (lnth (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) E') i)) =
            sum (enat ∘ (λi. length ⦃snd (lnth E' i)⦄o)) {..<m}
    using m by(simp add: less-trans[where y=enat m] split-beta)
  also have ... = enat (∑ i<m. length ⦃snd (lnth E' i)⦄o)
    by(subst sum-comp-morphism)(simp-all add: zero-enat-def)
  finally have a: a = (∑ i<m. length ⦃snd (lnth E' i)⦄o) + n by simp
```

```
  with E-a n m show thesis using ‹lnth E' m = (t, ta)› by(rule that)
qed

lemma actions-ℰE:
  assumes E: E ∈ ℰ σ
  and a: enat a < llength E
  obtains E' m n t ta
  where E = lconcat (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) E')
  and mthr.Runs σ E'
  and lnth E a = (t, ⦃ta⦄o ! n)
  and n < length ⦃ta⦄o and enat m < llength E'
  and a = (∑ i<m. length ⦃snd (lnth E' i)⦄o) + n
  and lnth E' m = (t, ta)
proof −
  from E obtain E' ws
    where E: E = lconcat (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) E')
    and mthr.Runs σ E' by(rule ℰ.cases) blast
  from ‹mthr.Runs σ E'› a[unfolded E]
  show ?thesis
    by(rule actions-ℰE-aux)(fold E, rule that[OF E ‹mthr.Runs σ E'›])
qed

end

context τmultithreaded-wf begin

    Alternative characterisation for ℰ

lemma ℰ-conv-Runs:
  ℰ σ = lconcat ‘ lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) ‘ llist-of-tllist ‘ {E. mthr.τRuns σ E}
  (is ?lhs = ?rhs)
proof(intro equalityI subsetI)
  fix E
  assume E ∈ ?rhs
  then obtain E' where E: E = lconcat (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) (llist-of-tllist
E'))
    and τRuns: mthr.τRuns σ E' by(blast)
  obtain E'' where E': E' = tmap (λ(tls, s', tl, s''). tl) (case-sum (λ(tls, s'). ⌊s'⌋) Map.empty) E''
    and τRuns': mthr.τRuns-table2 σ E''
    using τRuns by(rule mthr.τRuns-into-τRuns-table2)
  have mthr.Runs σ (lconcat (lappend (lmap (λ(tls, s, tl, s'). llist-of (tls @ [tl])) (llist-of-tllist E''))
                                     (LCons (case terminal E'' of Inl (tls, s') ⇒ llist-of tls | Inr tls ⇒ tls)
LNil)))
    (is mthr.Runs - ?E''')
    using τRuns' by(rule mthr.τRuns-table2-into-Runs)
  moreover
  let ?tail = λE''. case terminal E'' of Inl (tls, s') ⇒ llist-of tls | Inr tls ⇒ tls
  {
   have E = lconcat (lfilter (λxs. ¬ lnull xs) (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) (llist-of-tllist
E')))
      unfolding E by(simp add: lconcat-lfilter-neq-LNil)
    also have ... = lconcat (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) (lmap (λ(tls, s', tta, s'').
tta) (lfilter (λ(tls, s', (t, ta), s''). ⦃ta⦄o ≠ []) (llist-of-tllist E'))))
      by(simp add: E' lfilter-lmap llist.map-comp o-def split-def)
    also
```

from ‹*mthr.τRuns-table2 σ E''*›
have *lmap* (λ(*tls*, *s'*, *tta*, *s''*). *tta*) (*lfilter* (λ(*tls*, *s'*, (*t*, *ta*), *s''*). {|*ta*|}$_o$ ≠ []) (*llist-of-tllist E''*)) =
lfilter (λ(*t*, *ta*). {|*ta*|}$_o$ ≠ []) (*lconcat* (*lappend* (*lmap* (λ(*tls*, *s*, *tl*, *s'*). *llist-of* (*tls* @ [*tl*])) (*llist-of-tllist E''*)) (*LCons* (*?tail E''*) *LNil*)))
(**is** *?lhs σ E'' = ?rhs σ E''*)
proof(*coinduction arbitrary*: *σ E'' rule*: *llist.coinduct-strong*)
case (*Eq-llist σ E''*)
have *?lnull*
by(*cases lfinite* (*llist-of-tllist E''*))(*fastforce split*: *sum.split-asm simp add*: *split-beta lset-lconcat-lfinite lappend-inf mthr.silent-move2-def dest*: *mthr.τRuns-table2-silentsD*[*OF Eq-llist*] *mthr.τRuns-table2-terminal-silent Eq-llist*] *mthr.τRuns-table2-terminal-inf-stepD*[*OF Eq-llist*] *mτmove-silentD inf-step-silentD silent-moves2-silentD split*: *sum.split-asm*)+
moreover
have *?LCons*
proof(*intro impI conjI*)
assume *lhs'*: ¬ *lnull* (*lmap* (λ(*tls*, *s'*, *tta*, *s''*). *tta*) (*lfilter* (λ(*tls*, *s'*, (*t*, *ta*), *s''*). {|*ta*|}$_o$ ≠ []) (*llist-of-tllist E''*)))
(**is** ¬ *lnull ?lhs'*)
and ¬ *lnull* (*lfilter* (λ(*t*, *ta*). {|*ta*|}$_o$ ≠ []) (*lconcat* (*lappend* (*lmap* (λ(*tls*, *s*, *tl*, *s'*). *llist-of* (*tls* @ [*tl*])) (*llist-of-tllist E''*)) (*LCons* (*case terminal E'' of Inl* (*tls*, *s'*) ⇒ *llist-of tls* | *Inr tls* ⇒ *tls*) *LNil*))))
(**is** ¬ *lnull ?rhs'*)

note *τRuns'* = ‹*mthr.τRuns-table2 σ E''*›
from *lhs'* **obtain** *tl tls'* **where** *?lhs σ E'' = LCons tl tls'*
by(*auto simp only*: *not-lnull-conv*)
then obtain *tls s' s''* *tlsstlss'*
where *tls'*: *tls' = lmap* (λ(*tls*, *s'*, *tta*, *s''*). *tta*) *tlsstlss'*
and *filter*: *lfilter* (λ(*tls*, *s'*, (*t*, *ta*), *s''*). *obs-a ta* ≠ []) (*llist-of-tllist E''*) = *LCons* (*tls*, *s'*, *tl*, *s''*) *tlsstlss'*
using *lhs'* **by**(*fastforce simp add*: *lmap-eq-LCons-conv*)
from *lfilter-eq-LConsD*[*OF filter*]
obtain *us vs* **where** *eq*: *llist-of-tllist E'' = lappend us* (*LCons* (*tls*, *s'*, *tl*, *s''*) *vs*)
and *fin*: *lfinite us*
and *empty*: ∀(*tls*, *s'*, (*t*, *ta*), *s''*)∈*lset us*. *obs-a ta* = []
and *neq-empty*: *obs-a* (*snd tl*) ≠ []
and *tlsstlss'*: *tlsstlss' = lfilter* (λ(*tls*, *s'*, (*t*, *ta*), *s''*). *obs-a ta* ≠ []) *vs*
by(*auto simp add*: *split-beta*)
from *eq* **obtain** *E'''* **where** *E''*: *E'' = lappendt us E'''*
and *eq'*: *llist-of-tllist E''' = LCons* (*tls*, *s'*, *tl*, *s''*) *vs*
and *terminal*: *terminal E''' = terminal E''*
unfolding *llist-of-tllist-eq-lappend-conv* **by** *auto*
from *τRuns' fin E''* **obtain** *σ'* **where** *τRuns''*: *mthr.τRuns-table2 σ' E'''*
by(*auto dest*: *mthr.τRuns-table2-lappendtD*)
then obtain *σ'' E''''* **where** *mthr.τRuns-table2 σ'' E'''' E''' = TCons* (*tls*, *s'*, *tl*, *s''*) *E''''*
using *eq'* **by** *cases auto*
moreover from *τRuns' E'' fin*
have ∀(*tls*, *s*, *tl*, *s'*)∈*lset us*. ∀(*t*, *ta*)∈*set tls*. *ta* = *ε*
by(*fastforce dest*: *mthr.τRuns-table2-silentsD mτmove-silentD simp add*: *mthr.silent-move2-def*)
hence *lfilter* (λ(*t*, *ta*). *obs-a ta* ≠ []) (*lconcat* (*lmap* (λ(*tls*, *s*, *tl*, *s'*). *llist-of* (*tls* @ [*tl*])) *us*)) = *LNil*
using *empty* **by**(*auto simp add*: *lfilter-empty-conv lset-lconcat-lfinite split-beta*)
moreover from *τRuns'' eq'* **have** *snd ' set tls* ⊆ {*ε*}
by(*cases*)(*fastforce dest*: *silent-moves2-silentD*)+
hence [(*t*, *ta*)←*tls* . *obs-a ta* ≠ []] = []

by(*auto simp add*: *filter-empty-conv split-beta*)
ultimately
show *lhd ?lhs′ = lhd ?rhs′*
and (∃ σ *E″*. *ltl ?lhs′ = lmap* (λ(*tls*, *s′*, *tta*, *s″*). *tta*) (*lfilter* (λ(*tls*, *s′*, (*t*, *ta*), *s″*). ⦃*ta*⦄$_o$ ≠ []) (*llist-of-tllist E″*)) ∧
ltl ?rhs′ = lfilter (λ(*t*, *ta*). ⦃*ta*⦄$_o$ ≠ []) (*lconcat* (*lappend* (*lmap* (λ(*tls*, *s*, *tl*, *s′*). *llist-of* (*tls* @ [*tl*])) (*llist-of-tllist E″*)) (*LCons* (*case terminal E″ of Inl* (*tls*, *s′*) ⇒ *llist-of tls* | *Inr tls* ⇒ *tls*) *LNil*))) ∧
τtrsys.τRuns-table2 redT mτmove σ E″) ∨
ltl ?lhs′ = ltl ?rhs′
using *lhs′ E″ fin tls′ tlsstlss′ filter eq′ neq-empty*
by(*auto simp add*: *lmap-lappend-distrib lappend-assoc split-beta filter-empty-conv simp del*: *split-paired-Ex*)
qed
ultimately show *?case* **..**
qed
also have *lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) (*obs-a ta*))) ... = *lfilter* (λ*obs*. ¬ *lnull obs*) (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) (*obs-a ta*))) (*lconcat* (*lappend* (*lmap* (λ(*tls*, *s*, *tl*, *s′*). *llist-of* (*tls* @ [*tl*])) (*llist-of-tllist E″*)) (*LCons* (*?tail E″*) *LNil*))))
unfolding *lfilter-lmap* **by**(*simp add*: *o-def split-def llist-of-eq-LNil-conv*)
finally have *E = lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) ⦃*ta*⦄$_o$)) *?E‴*)
by(*simp add*: *lconcat-lfilter-neq-LNil*) **}**
ultimately show *E* ∈ *?lhs* **by**(*blast intro*: *E.intros*)
next
fix *E*
assume *E* ∈ *?lhs*
then obtain *E′* **where** *E*: *E = lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) (*obs-a ta*))) *E′*)
and *Runs*: *mthr.Runs σ E′* **by**(*blast elim*: *E.cases*)
from *Runs* **obtain** *E″* **where** *E′*: *E′ = lmap* (λ(*s*, *tl*, *s′*). *tl*) *E″*
and *Runs′*: *mthr.Runs-table σ E″* **by**(*rule mthr.Runs-into-Runs-table*)
have *mthr.τRuns σ* (*tmap* (λ(*s*, *tl*, *s′*). *tl*) *id* (*tfilter None* (λ(*s*, *tl*, *s′*). ¬ *mτmove s tl s′*) (*tllist-of-llist* (*Some* (*llast* (*LCons σ* (*lmap* (λ(*s*, *tl*, *s′*). *s′*) *E″*)))) *E″*)))
(**is** *mthr.τRuns - ?E‴*)
using *Runs′* **by**(*rule mthr.Runs-table-into-τRuns*)
moreover
have (λ(*s*, (*t*, *ta*), *s′*). *obs-a ta* ≠ []) = (λ(*s*, (*t*, *ta*), *s′*). *obs-a ta* ≠ [] ∧ ¬ *mτmove s* (*t*, *ta*) *s′*)
by(*rule ext*)(*auto dest*: *mτmove-silentD*)
hence *E = lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) (*obs-a ta*))) (*llist-of-tllist ?E‴*))
unfolding *E E′*
by(*subst* (*1 2*) *lconcat-lfilter-neq-LNil*[*symmetric*])(*simp add*: *lfilter-lmap lfilter-lfilter o-def split-def*)
ultimately show *E* ∈ *?rhs* **by**(*blast*)
qed

end

Running threads have been started before

definition *Status-no-wait-locks* :: (*'l*,*'t*,*status* × *'x*) *thread-info* ⇒ *bool*
where
Status-no-wait-locks ts ⟷
(∀ *t status x ln*. *ts t* = ⌊((*status*, *x*), *ln*)⌋ ⟶ *status* ≠ *Running* ⟶ *ln* = *no-wait-locks*)

lemma *Status-no-wait-locks-PreStartD*:
⋀*ln*. ⟦ *Status-no-wait-locks ts*; *ts t* = ⌊((*PreStart*, *x*), *ln*)⌋ ⟧ ⟹ *ln* = *no-wait-locks*
unfolding *Status-no-wait-locks-def* **by** *blast*

lemma *Status-no-wait-locks-FinishedD*:
⋀*ln*. ⟦ *Status-no-wait-locks ts*; *ts t* = ⌊((*Finished*, *x*), *ln*)⌋ ⟧ ⟹ *ln* = *no-wait-locks*
unfolding *Status-no-wait-locks-def* **by** *blast*

lemma *Status-no-wait-locksI*:
(⋀*t status x ln*. ⟦ *ts t* = ⌊((*status*, *x*), *ln*)⌋; *status* = *PreStart* ∨ *status* = *Finished* ⟧ ⟹ *ln* = *no-wait-locks*)
⟹ *Status-no-wait-locks ts*
unfolding *Status-no-wait-locks-def*
apply *clarify*
apply(*case-tac status*)
apply *auto*
done

context *heap-base* **begin**

lemma *Status-no-wait-locks-start-state*:
Status-no-wait-locks (*thr* (*init-fin-lift-state status* (*start-state f P C M vs*)))
by(*clarsimp simp add*: *Status-no-wait-locks-def init-fin-lift-state-def start-state-def split-beta*)

end

context *multithreaded-base* **begin**

lemma *init-fin-preserve-Status-no-wait-locks*:
assumes *ok*: *Status-no-wait-locks* (*thr s*)
and *redT*: *multithreaded-base.redT init-fin-final init-fin* (*map NormalAction* ∘ *convert-RA*) *s tta s′*
shows *Status-no-wait-locks* (*thr s′*)
using *redT*
proof(*cases rule*: *multithreaded-base.redT.cases*[*consumes 1*, *case-names redT-normal redT-acquire*])
case *redT-acquire*
with *ok* **show** *?thesis*
by(*auto intro*!: *Status-no-wait-locksI dest*: *Status-no-wait-locks-PreStartD Status-no-wait-locks-FinishedD split*: *if-split-asm*)
next
case *redT-normal*
show *?thesis*
proof(*rule Status-no-wait-locksI*)
fix *t′ status′ x′ ln′*
assume *tst′*: *thr s′ t′* = ⌊((*status′*, *x′*), *ln′*)⌋
and *status*: *status′* = *PreStart* ∨ *status′* = *Finished*
show *ln′* = *no-wait-locks*
proof(*cases thr s t′*)
case *None*
with *redT-normal tst′* **show** *?thesis*
by(*fastforce elim*!: *init-fin.cases dest*: *redT-updTs-new-thread simp add*: *final-thread.actions-ok-iff split*: *if-split-asm*)
next
case (*Some sxln*)
obtain *status″ x″ ln″*
where [*simp*]: *sxln* = ((*status″*, *x″*), *ln″*) **by**(*cases sxln*) *auto*
show *?thesis*
proof(*cases fst tta* = *t′*)

```
      case True
      with redT-normal tst' status show ?thesis by(auto simp add: expand-finfun-eq fun-eq-iff)
    next
      case False
      with tst' redT-normal Some status have status'' = status' ln'' = ln'
        by(force dest: redT-updTs-Some simp add: final-thread.actions-ok-iff)+
      with ok Some status show ?thesis
        by(auto dest: Status-no-wait-locks-PreStartD Status-no-wait-locks-FinishedD)
    qed
  qed
 qed
qed

lemma init-fin-Running-InitialThreadAction:
 assumes redT: multithreaded-base.redT init-fin-final init-fin (map NormalAction ∘ convert-RA) s tta
s'
 and not-running: ⋀x ln. thr s t ≠ ⌊((Running, x), ln)⌋
 and running: thr s' t = ⌊((Running, x'), ln')⌋
 shows tta = (t, {⌊InitialThreadAction⌋})
using redT
proof(cases rule: multithreaded-base.redT.cases[consumes 1, case-names redT-normal redT-acquire])
 case redT-acquire
 with running not-running show ?thesis by(auto split: if-split-asm)
next
 case redT-normal
 show ?thesis
 proof(cases thr s t)
  case None
  with redT-normal running not-running show ?thesis
   by(fastforce simp add: final-thread.actions-ok-iff elim: init-fin.cases dest: redT-updTs-new-thread
split: if-split-asm)
 next
  case (Some a)
  with redT-normal running not-running show ?thesis
   apply(cases a)
   apply(auto simp add: final-thread.actions-ok-iff split: if-split-asm elim: init-fin.cases)
   apply((drule (1) redT-updTs-Some)?, fastforce)+
   done
 qed
qed

end

context if-multithreaded begin

lemma init-fin-Trsys-preserve-Status-no-wait-locks:
 assumes ok: Status-no-wait-locks (thr s)
 and Trsys: if.mthr.Trsys s ttas s'
 shows Status-no-wait-locks (thr s')
using Trsys ok
by(induct)(blast dest: init-fin-preserve-Status-no-wait-locks)+

lemma init-fin-Trsys-Running-InitialThreadAction:
 assumes redT: if.mthr.Trsys s ttas s'
```

and *not-running*: $\bigwedge x\ ln.\ thr\ s\ t \neq \lfloor((Running,\ x),\ ln)\rfloor$
and *running*: $thr\ s'\ t = \lfloor((Running,\ x'),\ ln')\rfloor$
shows $(t,\ \{\!|InitialThreadAction|\!\}) \in set\ ttas$
using *redT not-running running*
proof(*induct arbitrary*: $x'\ ln'$)
case *rtrancl3p-refl* **thus** *?case* **by**(*fastforce*)
next
case (*rtrancl3p-step s ttas s' tta s''*) **thus** *?case*
by(*cases* $\exists x\ ln.\ thr\ s'\ t = \lfloor((Running,\ x),\ ln)\rfloor$)(*fastforce dest*: *init-fin-Running-InitialThreadAction*)+
qed

end

locale *heap-multithreaded-base* =
heap-base
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
+
mthr: *multithreaded-base final r convert-RA*
for *addr2thread-id* :: $('addr :: addr) \Rightarrow 'thread\text{-}id$
and *thread-id2addr* :: $'thread\text{-}id \Rightarrow 'addr$
and *spurious-wakeups* :: *bool*
and *empty-heap* :: $'heap$
and *allocate* :: $'heap \Rightarrow htype \Rightarrow ('heap \times 'addr)\ set$
and *typeof-addr* :: $'heap \Rightarrow 'addr \rightharpoonup htype$
and *heap-read* :: $'heap \Rightarrow 'addr \Rightarrow addr\text{-}loc \Rightarrow 'addr\ val \Rightarrow bool$
and *heap-write* :: $'heap \Rightarrow 'addr \Rightarrow addr\text{-}loc \Rightarrow 'addr\ val \Rightarrow 'heap \Rightarrow bool$
and *final* :: $'x \Rightarrow bool$
and *r* :: $('addr,\ 'thread\text{-}id,\ 'x,\ 'heap,\ 'addr,\ ('addr,\ 'thread\text{-}id)\ obs\text{-}event)\ semantics$ (‹- ⊢ - –-→ -› [*50,0,0,50*] *80*)
and *convert-RA* :: $'addr\ released\text{-}locks \Rightarrow ('addr,\ 'thread\text{-}id)\ obs\text{-}event\ list$

sublocale *heap-multithreaded-base* < *mthr*: *if-multithreaded-base final r convert-RA*
.

context *heap-multithreaded-base* **begin**

abbreviation $\mathcal{E}$*-start* ::
$(cname \Rightarrow mname \Rightarrow ty\ list \Rightarrow ty \Rightarrow 'md \Rightarrow 'addr\ val\ list \Rightarrow 'x)$
$\Rightarrow 'md\ prog \Rightarrow cname \Rightarrow mname \Rightarrow 'addr\ val\ list \Rightarrow status$
$\Rightarrow ('thread\text{-}id \times ('addr,\ 'thread\text{-}id)\ obs\text{-}event\ action)\ llist\ set$
where
$\mathcal{E}$*-start f P C M vs status* ≡
lappend (*llist-of* (*lift-start-obs start-tid start-heap-obs*)) ‘
mthr.if.$\mathcal{E}$ (*init-fin-lift-state status* (*start-state f P C M vs*))

end

locale *heap-multithreaded* =
heap-multithreaded-base
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write

```
    final r convert-RA
  +
  heap
    addr2thread-id thread-id2addr
    spurious-wakeups
    empty-heap allocate typeof-addr heap-read heap-write
    P
  +
  mthr: multithreaded final r convert-RA

  for addr2thread-id :: ('addr :: addr) ⇒ 'thread-id
  and thread-id2addr :: 'thread-id ⇒ 'addr
  and spurious-wakeups :: bool
  and empty-heap :: 'heap
  and allocate :: 'heap ⇒ htype ⇒ ('heap × 'addr) set
  and typeof-addr :: 'heap ⇒ 'addr ⇀ htype
  and heap-read :: 'heap ⇒ 'addr ⇒ addr-loc ⇒ 'addr val ⇒ bool
  and heap-write :: 'heap ⇒ 'addr ⇒ addr-loc ⇒ 'addr val ⇒ 'heap ⇒ bool
  and final :: 'x ⇒ bool
  and r :: ('addr, 'thread-id, 'x, 'heap, 'addr, ('addr, 'thread-id) obs-event) semantics (‹- ⊢ - -–→ -›
[50,0,0,50] 80)
  and convert-RA :: 'addr released-locks ⇒ ('addr, 'thread-id) obs-event list
  and P :: 'md prog

sublocale heap-multithreaded < mthr: if-multithreaded final r convert-RA
by(unfold-locales)

sublocale heap-multithreaded < if: jmm-multithreaded
  mthr.init-fin-final mthr.init-fin map NormalAction ∘ convert-RA P
.

context heap-multithreaded begin

lemma thread-start-actions-ok-init-fin-RedT:
  assumes Red: mthr.if.RedT (init-fin-lift-state status (start-state f P C M vs)) ttas s'
          (is mthr.if.RedT ?start-state - -)
  shows thread-start-actions-ok (llist-of (lift-start-obs start-tid start-heap-obs @ concat (map (λ(t, ta).
map (Pair t) {|ta|}_o) ttas)))
   (is thread-start-actions-ok (llist-of (?obs-prefix @ ?E')))
proof(rule thread-start-actions-okI)
  let ?E = llist-of (?obs-prefix @ ?E')
  fix a
  assume a: a ∈ actions ?E
    and new: ¬ is-new-action (action-obs ?E a)
  show ∃ i ≤ a. action-obs ?E i = InitialThreadAction ∧ action-tid ?E i = action-tid ?E a
  proof(cases action-tid ?E a = start-tid)
    case True thus ?thesis
      by(auto simp add: lift-start-obs-def action-tid-def action-obs-def)
  next
    case False
    let ?a = a − length ?obs-prefix

    from False have a-len: a ≥ length ?obs-prefix
      by(rule contrapos-np)(auto simp add: lift-start-obs-def action-tid-def lnth-LCons nth-append split:
```

```
nat.split)
  hence [simp]: action-tid ?E a = action-tid (llist-of ?E') ?a action-obs ?E a = action-obs (llist-of ?E') ?a
    by(simp-all add: action-tid-def nth-append action-obs-def)

  from False have not-running: ⋀x ln. thr ?start-state (action-tid (llist-of ?E') ?a) ≠ ⌊((Running, x), ln)⌋
    by(auto simp add: start-state-def split-beta init-fin-lift-state-def split: if-split-asm)

  from a a-len have ?a < length ?E' by(simp add: actions-def)
  from nth-concat-conv[OF this]
  obtain m n where E'-a: ?E' ! ?a = (λ(t, ta). (t, {|ta|}_o ! n)) (ttas ! m)
    and n: n < length {|snd (ttas ! m)|}_o
    and m: m < length ttas
    and a-conv: ?a = (∑ i<m. length (map (λ(t, ta). map (Pair t) {|ta|}_o) ttas ! i)) + n
    by(clarsimp simp add: split-def)

  from Red obtain s'' s''' where Red1: mthr.if.RedT ?start-state (take m ttas) s''
    and red: mthr.if.redT s'' (ttas ! m) s'''
    and Red2: mthr.if.RedT s''' (drop (Suc m) ttas) s'
    unfolding mthr.if.RedT-def
    by(subst (asm) (4) id-take-nth-drop[OF m])(blast elim: rtrancl3p-appendE rtrancl3p-converseE)

  from E'-a m n have [simp]: action-tid (llist-of ?E') ?a = fst (ttas ! m)
    by(simp add: action-tid-def split-def)

  from red obtain status x ln where tst: thr s'' (fst (ttas ! m)) = ⌊((status, x), ln)⌋ by cases auto
  show ?thesis
  proof(cases status = PreStart ∨ status = Finished)
    case True
    from Red1 have Status-no-wait-locks (thr s'')
      unfolding mthr.if.RedT-def
      by(rule mthr.init-fin-Trsys-preserve-Status-no-wait-locks[OF Status-no-wait-locks-start-state])
    with True tst have ln = no-wait-locks
      by(auto dest: Status-no-wait-locks-PreStartD Status-no-wait-locks-FinishedD)
    with red tst True have {|snd (ttas ! m)|}_o = [InitialThreadAction] by(cases) auto
    hence action-obs ?E a = InitialThreadAction using a-conv n a-len E'-a
      by(simp add: action-obs-def nth-append split-beta)
    thus ?thesis by(auto)
  next
    case False
    hence status = Running by(cases status) auto
    with tst mthr.init-fin-Trsys-Running-InitialThreadAction[OF Red1[unfolded mthr.if.RedT-def] not-running]
    have (fst (ttas ! m), {|InitialThreadAction|}) ∈ set (take m ttas)
      using E'-a by(auto simp add: action-tid-def split-beta)
    then obtain i where i: i < m
      and nth-i: ttas ! i = (fst (ttas ! m), {|InitialThreadAction|})
      unfolding in-set-conv-nth by auto

    let ?i' = length (concat (map (λ(t, ta). map (Pair t) {|ta|}_o) (take i ttas)))
    let ?i = length ?obs-prefix + ?i'

    from i m nth-i
```

```
    have ?i' < length (concat (map (λ(t, ta). map (Pair t) ⦃ta⦄o) (take m ttas)))
      apply(simp add: length-concat o-def split-beta)
      apply(subst (6) id-take-nth-drop[where i=i])
      apply(simp-all add: take-map[symmetric] min-def)
      done
    also from m have ... ≤ ?a unfolding a-conv
      by(simp add: length-concat sum-list-sum-nth min-def split-def atLeast0LessThan)
    finally have ?i < a using a-len by simp
    moreover
    from i m nth-i have ?i' < length ?E'
      apply(simp add: length-concat o-def split-def)
      apply(subst (7) id-take-nth-drop[where i=i])
      apply(simp-all add: take-map[symmetric])
      done
    from nth-i i E'-a a-conv m
    have lnth ?E ?i = (fst (ttas ! m), InitialThreadAction)
    by(simp add: lift-start-obs-def nth-append length-concat o-def split-def)(rule nth-concat-eqI[where
k=0 and i=i], simp-all add: take-map o-def split-def)
    ultimately show ?thesis using E'-a
      by(cases ttas ! m)(auto simp add: action-obs-def action-tid-def nth-append intro!: exI[where
x=?i])
  qed
 qed
qed



lemma thread-start-actions-ok-init-fin:
 assumes E: E ∈ mthr.if.ℰ (init-fin-lift-state status (start-state f P C M vs))
 shows thread-start-actions-ok (lappend (llist-of (lift-start-obs start-tid start-heap-obs)) E)
 (is thread-start-actions-ok ?E)
proof(rule thread-start-actions-okI)
 let ?start-heap-obs = lift-start-obs start-tid start-heap-obs
 let ?start-state = init-fin-lift-state status (start-state f P C M vs)
 fix a
 assume a: a ∈ actions ?E
  and a-new: ¬ is-new-action (action-obs ?E a)
 show ∃i. i ≤ a ∧ action-obs ?E i = InitialThreadAction ∧ action-tid ?E i = action-tid ?E a
 proof(cases action-tid ?E a = start-tid)
  case True thus ?thesis
   by(auto simp add: lift-start-obs-def action-tid-def action-obs-def)
 next
  case False

  let ?a = a − length ?start-heap-obs

  from False have a ≥ length ?start-heap-obs
   by(rule contrapos-np)(auto simp add: lift-start-obs-def action-tid-def lnth-LCons lnth-lappend1
split: nat.split)
  hence [simp]: action-tid ?E a = action-tid E ?a action-obs ?E a = action-obs E ?a
   by(simp-all add: action-tid-def lnth-lappend2 action-obs-def)

  from False have not-running: ⋀x ln. thr ?start-state (action-tid E ?a) ≠ ⌊((Running, x), ln)⌋
   by(auto simp add: start-state-def split-beta init-fin-lift-state-def split: if-split-asm)
```

from *E* **obtain** *E'* **where** *E'*: *E* = *lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) *E'*)
and *τRuns*: *mthr.if.mthr.Runs ?start-state E'* **by**(*rule mthr.if.*$\mathcal{E}$*.cases*)
from *a E'* ‹*a* ≥ *length ?start-heap-obs*›
have *enat-a*: *enat ?a* < *llength* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) *E'*))
by(*cases llength* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) *E'*)))(*auto simp add*: *actions-def*)
with *τRuns* **obtain** *m n t ta*
where *a-obs*: *lnth* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) *E'*)) (*a* − *length ?start-heap-obs*) = (*t*, {|*ta*|}$_o$! *n*)
and *n*: *n* < *length* {|*ta*|}$_o$
and *m*: *enat m* < *llength E'*
and *a-conv*: *?a* = (∑ *i*<*m*. *length* {|*snd* (*lnth E' i*)|}$_o$) + *n*
and *E'-m*: *lnth E' m* = (*t*, *ta*)
by(*rule mthr.if.actions-*$\mathcal{E}$*E-aux*)
from *a-obs* **have** [*simp*]: *action-tid E ?a* = *t action-obs E ?a* = {|*ta*|}$_o$! *n*
by(*simp-all add*: *E' action-tid-def action-obs-def*)

let *?E'* = *ldropn* (*Suc m*) *E'*
let *?m-E'* = *ltake* (*enat m*) *E'*
have *E'-unfold*: *E'* = *lappend* (*ltake* (*enat m*) *E'*) (*LCons* (*lnth E' m*) *?E'*)
unfolding *ldropn-Suc-conv-ldropn*[*OF m*] **by** *simp*
hence *mthr.if.mthr.Runs ?start-state* (*lappend ?m-E'* (*LCons* (*lnth E' m*) *?E'*))
using *τRuns* **by** *simp*
then obtain *σ'* **where** *σ-σ'*: *mthr.if.mthr.Trsys ?start-state* (*list-of ?m-E'*) *σ'*
and *τRuns'*: *mthr.if.mthr.Runs σ'* (*LCons* (*lnth E' m*) *?E'*)
by(*rule mthr.if.mthr.Runs-lappendE*) *simp*
from *τRuns'* **obtain** *σ'''* **where** *red-a*: *mthr.if.redT σ'* (*t*, *ta*) *σ'''*
and *τRuns''*: *mthr.if.mthr.Runs σ'''* *?E'*
unfolding *E'-m* **by** *cases*
from *red-a* **obtain** *status x ln* **where** *tst*: *thr σ' t* = ⌊((*status*, *x*), *ln*)⌋ **by** *cases auto*
show *?thesis*
proof(*cases status* = *PreStart* ∨ *status* = *Finished*)
case *True*
have *Status-no-wait-locks* (*thr σ'*)
by(*rule mthr.init-fin-Trsys-preserve-Status-no-wait-locks*[*OF - σ-σ'*])(*rule Status-no-wait-locks-start-state*)
with *True tst* **have** *ln* = *no-wait-locks*
by(*auto dest*: *Status-no-wait-locks-PreStartD Status-no-wait-locks-FinishedD*)
with *red-a tst True* **have** {|*ta*|}$_o$ = [*InitialThreadAction*] **by**(*cases*) *auto*
hence *action-obs E ?a* = *InitialThreadAction* **using** *a-obs n* **unfolding** *E'*
by(*simp add*: *action-obs-def*)
thus *?thesis* **by**(*auto*)
next
case *False*
hence *status* = *Running* **by**(*cases status*) *auto*
with *tst mthr.init-fin-Trsys-Running-InitialThreadAction*[*OF σ-σ' not-running*]
have (*action-tid E ?a*, {|*InitialThreadAction*|}) ∈ *set* (*list-of* (*ltake* (*enat m*) *E'*))
using *a-obs E'* **by**(*auto simp add*: *action-tid-def*)
then obtain *i* **where** *i* < *m enat i* < *llength E'*
and *nth-i*: *lnth E' i* = (*action-tid E ?a*, {|*InitialThreadAction*|})
unfolding *in-set-conv-nth*
by(*cases llength E'*)(*auto simp add*: *length-list-of-conv-the-enat lnth-ltake*)

let *?i'* = ∑ *i*<*i*. *length* {|*snd* (*lnth E' i*)|}$_o$

```
    let ?i = length ?start-heap-obs + ?i'

    from ‹i < m› have (∑ i<m. length ⦇snd (lnth E' i)⦈o) = ?i' + (∑ i=i..<m. length ⦇snd (lnth
E' i)⦈o)
      unfolding atLeast0LessThan[symmetric] by(subst sum.atLeastLessThan-concat) simp-all
    hence ?i' ≤ ?a unfolding a-conv by simp
    hence ?i ≤ a using ‹a ≥ length ?start-heap-obs› by arith


    from ‹?i' ≤ ?a› have enat ?i' < llength E using enat-a E'
      by(simp add: le-less-trans[where y=enat ?a])
    from lnth-lconcat-conv[OF this[unfolded E'], folded E']
    obtain k l
      where nth-i': lnth E ?i' = lnth (lnth (lmap (λ(t, ta). llist-of (map (Pair t) ⦇ta⦈o)) E') k) l
      and l: l < length ⦇snd (lnth E' k)⦈o
      and k: enat k < llength E'
      and i-conv: enat ?i' = (∑ i<k. llength (lnth (lmap (λ(t, ta). llist-of (map (Pair t) ⦇ta⦈o)) E')
i)) + enat l
      by(fastforce simp add: split-beta)

    have (∑ i<k. llength (lnth (lmap (λ(t, ta). llist-of (map (Pair t) ⦇ta⦈o)) E') i)) =
        (∑ i<k. (enat ∘ (λi. length ⦇snd (lnth E' i)⦈o)) i)
      by(rule sum.cong)(simp-all add: less-trans[where y=enat k] split-beta k)
    also have ... = enat (∑ i<k. length ⦇snd (lnth E' i)⦈o)
      by(rule sum-comp-morphism)(simp-all add: zero-enat-def)
    finally have i-conv: ?i' = (∑ i<k. length ⦇snd (lnth E' i)⦈o) + l using i-conv by simp

    have [simp]: i = k
    proof(rule ccontr)
      assume i ≠ k
      thus False unfolding neq-iff
      proof
        assume i < k
        hence (∑ i<k. length ⦇snd (lnth E' i)⦈o) =
            (∑ i<i. length ⦇snd (lnth E' i)⦈o) + (∑ i=i..<k. length ⦇snd (lnth E' i)⦈o)
          unfolding atLeast0LessThan[symmetric] by(subst sum.atLeastLessThan-concat) simp-all
        with i-conv have (∑ i=i..<k. length ⦇snd (lnth E' i)⦈o) = l l = 0 by simp-all
        moreover have (∑ i=i..<k. length ⦇snd (lnth E' i)⦈o) ≥ length ⦇snd (lnth E' i)⦈o
          by(subst sum.atLeast-Suc-lessThan[OF ‹i < k›]) simp
        ultimately show False using nth-i by simp
      next
        assume k < i
        hence ?i' = (∑ i<k. length ⦇snd (lnth E' i)⦈o) + (∑ i=k..<i. length ⦇snd (lnth E' i)⦈o)
          unfolding atLeast0LessThan[symmetric] by(subst sum.atLeastLessThan-concat) simp-all
        with i-conv have (∑ i=k..<i. length ⦇snd (lnth E' i)⦈o) = l by simp
        moreover have (∑ i=k..<i. length ⦇snd (lnth E' i)⦈o) ≥ length ⦇snd (lnth E' k)⦈o
          by(subst sum.atLeast-Suc-lessThan[OF ‹k < i›]) simp
        ultimately show False using l by simp
      qed
    qed
    with l nth-i have [simp]: l = 0 by simp

    hence lnth E ?i' = (action-tid E ?a, InitialThreadAction)
      using nth-i nth-i' k by simp
```

with ‹*?i* ≤ *a*› **show** *?thesis*
by(*auto simp add*: *action-tid-def action-obs-def lnth-lappend2*)
qed
qed
qed

end

In the subsequent locales, *convert-RA* refers to *convert-RA* and is no longer a parameter!

lemma *convert-RA-not-write*:
⋀*ln*. *ob* ∈ *set* (*convert-RA ln*) ⟹ ¬ *is-write-action* (*NormalAction ob*)
by(*auto simp add*: *convert-RA-def*)

lemma *ta-seq-consist-convert-RA*:
fixes *ln* **shows**
ta-seq-consist P vs (*llist-of* ((*map NormalAction* ∘ *convert-RA*) *ln*))
proof(*rule ta-seq-consist-nthI*)
fix *i ad al v*
assume *enat i* < *llength* (*llist-of* ((*map NormalAction* ∘ *convert-RA*) *ln* :: (*'b*, *'c*) *obs-event action list*))
and *lnth* (*llist-of* ((*map NormalAction* ∘ *convert-RA*) *ln* :: (*'b*, *'c*) *obs-event action list*)) *i* = *NormalAction* (*ReadMem ad al v*)
hence *ReadMem ad al v* ∈ *set* (*convert-RA ln* :: (*'b*, *'c*) *obs-event list*)
by(*auto simp add*: *in-set-conv-nth*)
hence *False* **by**(*auto simp add*: *convert-RA-def*)
thus ∃ *b*. *mrw-values P vs* (*list-of* (*ltake* (*enat i*) (*llist-of* ((*map NormalAction* ∘ *convert-RA*) *ln*)))) (*ad*, *al*) = ⌊(*v*, *b*)⌋ **..**
qed

lemma *ta-hb-consistent-convert-RA*:
⋀*ln*. *ta-hb-consistent P E* (*llist-of* (*map* (*Pair t*) ((*map NormalAction* ∘ *convert-RA*) *ln*)))
by(*rule ta-hb-consistent-not-ReadI*)(*auto simp add*: *convert-RA-def*)

locale *allocated-multithreaded* =
allocated-heap
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
allocated
P
+
mthr: *multithreaded final r convert-RA*

for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *allocated* :: *'heap* ⇒ *'addr set*

and *final* :: *'x* ⇒ *bool*
and *r* :: (*'addr*, *'thread-id*, *'x*, *'heap*, *'addr*, (*'addr*, *'thread-id*) *obs-event*) *semantics* (‹- ⊢ - –-→ -› [*50*,*0*,*0*,*50*] *80*)
and *P* :: *'md prog*
+
assumes *red-allocated-mono*: *t* ⊢ (*x*, *m*) –*ta*→ (*x'*, *m'*) ⟹ *allocated m* ⊆ *allocated m'*
and *red-New-allocatedD*:
⟦ *t* ⊢ (*x*, *m*) –*ta*→ (*x'*, *m'*); *NewHeapElem ad CTn* ∈ *set* {|*ta*|}*o* ⟧
⟹ *ad* ∈ *allocated m'* ∧ *ad* ∉ *allocated m*
and *red-allocated-NewD*:
⟦ *t* ⊢ (*x*, *m*) –*ta*→ (*x'*, *m'*); *ad* ∈ *allocated m'*; *ad* ∉ *allocated m* ⟧
⟹ ∃ *CTn*. *NewHeapElem ad CTn* ∈ *set* {|*ta*|}*o*
and *red-New-same-addr-same*:
⟦ *t* ⊢ (*x*, *m*) –*ta*→ (*x'*, *m'*);
{|*ta*|}*o* ! *i* = *NewHeapElem a CTn*; *i* < *length* {|*ta*|}*o*;
{|*ta*|}*o* ! *j* = *NewHeapElem a CTn'*; *j* < *length* {|*ta*|}*o* ⟧
⟹ *i* = *j*

sublocale *allocated-multithreaded* < *heap-multithreaded*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
final r convert-RA P
by(*unfold-locales*)

context *allocated-multithreaded* **begin**

lemma *redT-allocated-mono*:
assumes *mthr.redT* *σ* (*t*, *ta*) *σ'*
shows *allocated* (*shr σ*) ⊆ *allocated* (*shr σ'*)
using *assms*
by *cases*(*auto dest*: *red-allocated-mono del*: *subsetI*)

lemma *RedT-allocated-mono*:
assumes *mthr.RedT* *σ ttas σ'*
shows *allocated* (*shr σ*) ⊆ *allocated* (*shr σ'*)
using *assms* **unfolding** *mthr.RedT-def*
by *induct*(*auto dest*!: *redT-allocated-mono intro*: *subset-trans del*: *subsetI*)

lemma *init-fin-allocated-mono*:
t ⊢ (*x*, *m*) –*ta*→*i* (*x'*, *m'*) ⟹ *allocated m* ⊆ *allocated m'*
by(*cases rule*: *mthr.init-fin.cases*)(*auto dest*: *red-allocated-mono*)

lemma *init-fin-redT-allocated-mono*:
assumes *mthr.if.redT* *σ* (*t*, *ta*) *σ'*
shows *allocated* (*shr σ*) ⊆ *allocated* (*shr σ'*)
using *assms*
by *cases*(*auto dest*: *init-fin-allocated-mono del*: *subsetI*)

lemma *init-fin-RedT-allocated-mono*:
assumes *mthr.if.RedT* *σ ttas σ'*
shows *allocated* (*shr σ*) ⊆ *allocated* (*shr σ'*)
using *assms* **unfolding** *mthr.if.RedT-def*

by *induct*(*auto dest*!: *init-fin-redT-allocated-mono intro*: *subset-trans del*: *subsetI*)

lemma *init-fin-red-New-allocatedD*:
assumes $t \vdash (x, m)\ -ta\rightarrow i\ (x', m')$ *NormalAction* (*NewHeapElem ad CTn*) $\in$ *set* $\{\!|ta|\!\}_o$
shows $ad \in$ *allocated* $m' \wedge ad \notin$ *allocated* m
using *assms*
by *cases*(*auto dest*: *red-New-allocatedD*)

lemma *init-fin-red-allocated-NewD*:
assumes $t \vdash (x, m)\ -ta\rightarrow i\ (x', m')$ $ad \in$ *allocated* m' $ad \notin$ *allocated* m
shows $\exists$ *CTn*. *NormalAction* (*NewHeapElem ad CTn*) $\in$ *set* $\{\!|ta|\!\}_o$
using *assms*
by(*cases*)(*auto dest*!: *red-allocated-NewD*)

lemma *init-fin-red-New-same-addr-same*:
assumes $t \vdash (x, m)\ -ta\rightarrow i\ (x', m')$
and $\{\!|ta|\!\}_o\ !\ i =$ *NormalAction* (*NewHeapElem a CTn*) $i <$ *length* $\{\!|ta|\!\}_o$
and $\{\!|ta|\!\}_o\ !\ j =$ *NormalAction* (*NewHeapElem a CTn'*) $j <$ *length* $\{\!|ta|\!\}_o$
shows $i = j$
using *assms*
by *cases*(*auto dest*: *red-New-same-addr-same*)

lemma *init-fin-redT-allocated-NewHeapElemD*:
assumes *mthr.if.redT s* (*t*, *ta*) *s'*
and *ad* $\in$ *allocated* (*shr s'*)
and *ad* $\notin$ *allocated* (*shr s*)
shows $\exists$ *CTn*. *NormalAction* (*NewHeapElem ad CTn*) $\in$ *set* $\{\!|ta|\!\}_o$
using *assms*
by(*cases*)(*auto dest*: *init-fin-red-allocated-NewD*)

lemma *init-fin-RedT-allocated-NewHeapElemD*:
assumes *mthr.if.RedT s ttas s'*
and *ad* $\in$ *allocated* (*shr s'*)
and *ad* $\notin$ *allocated* (*shr s*)
shows $\exists$ *t ta CTn*. (*t*, *ta*) $\in$ *set ttas* $\wedge$ *NormalAction* (*NewHeapElem ad CTn*) $\in$ *set* $\{\!|ta|\!\}_o$
using *assms*
proof(*induct rule*: *mthr.if.RedT-induct'*)
case *refl* **thus** *?case* **by** *simp*
next
case (*step ttas s' t ta s''*) **thus** *?case*
by(*cases ad* $\in$ *allocated* (*shr s'*))(*fastforce simp del*: *split-paired-Ex dest*: *init-fin-redT-allocated-NewHeapElemD*)+
qed

lemma $\mathcal{E}$*-new-actions-for-unique*:
assumes *E*: $E \in \mathcal{E}$*-start f P C M vs status*
and *a*: $a \in$ *new-actions-for P E adal*
and *a'*: $a' \in$ *new-actions-for P E adal*
shows $a = a'$
using *a a'*
proof(*induct a a' rule*: *wlog-linorder-le*)
case *symmetry* **thus** *?case* **by** *simp*
next
case (*le a a'*)
note $a =$ ‹$a \in$ *new-actions-for P E adal*›

```
    and a' = ‹a' ∈ new-actions-for P E adal›
    and a-a' = ‹a ≤ a'›
  obtain ad al where adal: adal = (ad, al) by(cases adal)

  let ?init-obs = lift-start-obs start-tid start-heap-obs
  let ?start-state = init-fin-lift-state status (start-state f P C M vs)

  have distinct: distinct (filter (λobs. ∃ a CTn. obs = NormalAction (NewHeapElem a CTn)) (map
snd ?init-obs))
    unfolding start-heap-obs-def
   by(fastforce intro: inj-onI intro!: distinct-filter simp add: distinct-map distinct-zipI1 distinct-initialization-list)

  from start-addrs-allocated
  have dom-start-state: {a. ∃ CTn. NormalAction (NewHeapElem a CTn) ∈ snd ` set ?init-obs} ⊆
allocated (shr ?start-state)
   by(fastforce simp add: init-fin-lift-state-conv-simps shr-start-state dest: NewHeapElem-start-heap-obs-start-addrsD
subsetD)

  show ?case
  proof(cases a' < length ?init-obs)
    case True
    with a' adal E obtain t-a' CTn-a'
      where CTn-a': ?init-obs ! a' = (t-a', NormalAction (NewHeapElem ad CTn-a'))
       by(cases ?init-obs ! a')(fastforce elim!: is-new-action.cases action-loc-aux-cases simp add: ac-
tion-obs-def lnth-lappend1 new-actions-for-def )+
    from True a-a' have len-a: a < length ?init-obs by simp
    with a adal E obtain t-a CTn-a
      where CTn-a: ?init-obs ! a = (t-a, NormalAction (NewHeapElem ad CTn-a))
       by(cases ?init-obs ! a)(fastforce elim!: is-new-action.cases action-loc-aux-cases simp add: ac-
tion-obs-def lnth-lappend1 new-actions-for-def )+
    from CTn-a CTn-a' True len-a
    have NormalAction (NewHeapElem ad CTn-a') ∈ snd ` set ?init-obs
      and NormalAction (NewHeapElem ad CTn-a) ∈ snd ` set ?init-obs unfolding set-conv-nth
      by(fastforce intro: rev-image-eqI)+
    hence [simp]: CTn-a' = CTn-a using distinct-start-addrs'
      by(auto simp add: in-set-conv-nth distinct-conv-nth start-heap-obs-def start-addrs-def) blast
     from distinct-filterD[OF distinct, of a' a NormalAction (NewHeapElem ad CTn-a)] len-a True
CTn-a CTn-a'
    show a = a' by simp
  next
    case False
    obtain n where n: length ?init-obs = n by blast
    with False have n ≤ a' by simp

    from E obtain E'' where E: E = lappend (llist-of ?init-obs) E''
      and E'': E'' ∈ mthr.if.ℰ ?start-state by auto
    from E'' obtain E' where E': E'' = lconcat (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) E')
      and τRuns: mthr.if.mthr.Runs ?start-state E' by(rule mthr.if.ℰ.cases)

    from E E'' a' n ‹n ≤ a'› adal have a': a' − n ∈ new-actions-for P E'' adal
     by(auto simp add: new-actions-for-def lnth-lappend2 action-obs-def actions-lappend elim: actionsE)

    from a' have a' − n ∈ actions E'' by(auto elim: new-actionsE)
    hence enat (a' − n) < llength E'' by(rule actionsE)
```

with $\tau Runs$ **obtain** a'-m a'-n t-a' ta-a'
where E-a': $lnth\ E''\ (a' - n) = (t\text{-}a',\ \{\!|ta\text{-}a'|\!\}_o\ !\ a'\text{-}n)$
and a'-n: $a'\text{-}n < length\ \{\!|ta\text{-}a'|\!\}_o$ **and** a'-m: $enat\ a'\text{-}m < llength\ E'$
and a'-$conv$: $a' - n = (\sum i{<}a'\text{-}m.\ length\ \{\!|snd\ (lnth\ E'\ i)|\!\}_o) + a'\text{-}n$
and E'-a'-m: $lnth\ E'\ a'\text{-}m = (t\text{-}a',\ ta\text{-}a')$
unfolding E' **by**($rule\ mthr.if.actions\text{-}\mathcal{E}E\text{-}aux$)

from a' **have** $is\text{-}new\text{-}action\ (action\text{-}obs\ E''\ (a' - n))$
and $(ad,\ al) \in action\text{-}loc\ P\ E''\ (a' - n)$
unfolding $adal$ **by**($auto\ elim$: $new\text{-}actionsE$)
then obtain CTn'
where $action\text{-}obs\ E''\ (a' - n) = NormalAction\ (NewHeapElem\ ad\ CTn')$
by $cases(fastforce)+$
hence $New\text{-}ta\text{-}a'$: $\{\!|ta\text{-}a'|\!\}_o\ !\ a'\text{-}n = NormalAction\ (NewHeapElem\ ad\ CTn')$
using E-a' a'-n **unfolding** $action\text{-}obs\text{-}def$ **by** $simp$

show $?thesis$
proof($cases\ a < n$)
case $True$
with $a\ adal\ E\ n$ **obtain** t-$a\ CTn$-a **where** $?init\text{-}obs\ !\ a = (t\text{-}a,\ NormalAction\ (NewHeapElem\ ad\ CTn\text{-}a))$
by($cases\ ?init\text{-}obs\ !\ a$)($fastforce\ elim!$: $is\text{-}new\text{-}action.cases\ simp\ add$: $action\text{-}obs\text{-}def\ lnth\text{-}lappend1\ new\text{-}actions\text{-}for\text{-}def$)+

with $subsetD[OF\ dom\text{-}start\text{-}state,\ of\ ad]\ n\ True$
have $a\text{-}shr\text{-}\sigma$: $ad \in allocated\ (shr\ ?start\text{-}state)$
by($fastforce\ simp\ add$: $set\text{-}conv\text{-}nth\ intro$: $rev\text{-}image\text{-}eqI$)

have $E'\text{-}unfold'$: $E' = lappend\ (ltake\ (enat\ a'\text{-}m)\ E')\ (LCons\ (lnth\ E'\ a'\text{-}m)\ (ldropn\ (Suc\ a'\text{-}m)\ E'))$
unfolding $ldropn\text{-}Suc\text{-}conv\text{-}ldropn[OF\ a'\text{-}m]$ **by** $simp$
hence $mthr.if.mthr.Runs\ ?start\text{-}state\ (lappend\ (ltake\ (enat\ a'\text{-}m)\ E')\ (LCons\ (lnth\ E'\ a'\text{-}m)\ (ldropn\ (Suc\ a'\text{-}m)\ E')))$
using $\tau Runs$ **by** $simp$

then obtain σ'
where σ-σ': $mthr.if.mthr.Trsys\ ?start\text{-}state\ (list\text{-}of\ (ltake\ (enat\ a'\text{-}m)\ E'))\ \sigma'$
and $\tau Runs'$: $mthr.if.mthr.Runs\ \sigma'\ (LCons\ (lnth\ E'\ a'\text{-}m)\ (ldropn\ (Suc\ a'\text{-}m)\ E'))$
by($rule\ mthr.if.mthr.Runs\text{-}lappendE$) $simp$
from $\tau Runs'$ **obtain** σ''
where $red\text{-}a'$: $mthr.if.redT\ \sigma'\ (t\text{-}a',\ ta\text{-}a')\ \sigma''$
and $\tau Runs''$: $mthr.if.mthr.Runs\ \sigma''\ (ldropn\ (Suc\ a'\text{-}m)\ E')$
unfolding E'-a'-m **by** $cases$
from $New\text{-}ta\text{-}a'\ a'\text{-}n$ **have** $NormalAction\ (NewHeapElem\ ad\ CTn') \in set\ \{\!|ta\text{-}a'|\!\}_o$
unfolding $in\text{-}set\text{-}conv\text{-}nth$ **by** $blast$
with $red\text{-}a'$ **obtain** x-$a'\ x'$-$a'\ m'$-a'
where red'-a': $mthr.init\text{-}fin\ t\text{-}a'\ (x\text{-}a',\ shr\ \sigma')\ ta\text{-}a'\ (x'\text{-}a',\ m'\text{-}a')$
and σ''': $redT\text{-}upd\ \sigma'\ t\text{-}a'\ ta\text{-}a'\ x'\text{-}a'\ m'\text{-}a'\ \sigma''$
and $ts\text{-}t\text{-}a'$: $thr\ \sigma'\ t\text{-}a' = \lfloor(x\text{-}a',\ no\text{-}wait\text{-}locks)\rfloor$
by $cases\ auto$
from red'-a' ‹$NormalAction\ (NewHeapElem\ ad\ CTn') \in set\ \{\!|ta\text{-}a'|\!\}_o$›
obtain ta'-$a'\ X$-$a'\ X'$-a'
where x-a': $x\text{-}a' = (Running,\ X\text{-}a')$
and x'-a': $x'\text{-}a' = (Running,\ X'\text{-}a')$

```
    and ta-a': ta-a' = convert-TA-initial (convert-obs-initial ta'-a')
    and red''-a': t-a' ⊢ ⟨X-a', shr σ'⟩ −ta'-a'→ ⟨X'-a', m'-a'⟩
    by cases fastforce+

  from ta-a' New-ta-a' a'-n have New-ta'-a': {|ta'-a'|}_o ! a'-n = NewHeapElem ad CTn'
    and a'-n': a'-n < length {|ta'-a'|}_o by auto
  hence NewHeapElem ad CTn' ∈ set {|ta'-a'|}_o unfolding in-set-conv-nth by blast
  with red''-a' have allocated-ad': ad ∉ allocated (shr σ')
    by(auto dest: red-New-allocatedD)

  have allocated (shr ?start-state) ⊆ allocated (shr σ')
    using σ-σ' unfolding mthr.if.RedT-def[symmetric] by(rule init-fin-RedT-allocated-mono)
  hence False using allocated-ad' a-shr-σ by blast
  thus ?thesis ..
next
  case False
  hence n ≤ a by simp

  from E E'' a n ‹n ≤ a› adal have a: a − n ∈ new-actions-for P E'' adal
    by(auto simp add: new-actions-for-def lnth-lappend2 action-obs-def actions-lappend elim: actionsE)

  from a have a − n ∈ actions E'' by(auto elim: new-actionsE)
  hence enat (a − n) < llength E'' by(rule actionsE)

  with τRuns obtain a-m a-n t-a ta-a
    where E-a: lnth E'' (a − n) = (t-a, {|ta-a|}_o ! a-n)
    and a-n: a-n < length {|ta-a|}_o and a-m: enat a-m < llength E'
    and a-conv: a − n = (∑ i<a-m. length {|snd (lnth E' i)|}_o) + a-n
    and E'-a-m: lnth E' a-m = (t-a, ta-a)
    unfolding E' by(rule mthr.if.actions-ℰE-aux)

  from a have is-new-action (action-obs E'' (a − n))
    and (ad, al) ∈ action-loc P E'' (a − n)
    unfolding adal by(auto elim: new-actionsE)
  then obtain CTn where action-obs E'' (a − n) = NormalAction (NewHeapElem ad CTn)
    by cases(fastforce)+
  hence New-ta-a: {|ta-a|}_o ! a-n = NormalAction (NewHeapElem ad CTn)
    using E-a a-n unfolding action-obs-def by simp

  let ?E' = ldropn (Suc a-m) E'

  have E'-unfold: E' = lappend (ltake (enat a-m) E') (LCons (lnth E' a-m) ?E')
    unfolding ldropn-Suc-conv-ldropn[OF a-m] by simp
  hence mthr.if.mthr.Runs ?start-state (lappend (ltake (enat a-m) E') (LCons (lnth E' a-m) ?E'))
    using τRuns by simp
  then obtain σ' where σ-σ': mthr.if.mthr.Trsys ?start-state (list-of (ltake (enat a-m) E')) σ'
    and τRuns': mthr.if.mthr.Runs σ' (LCons (lnth E' a-m) ?E')
    by(rule mthr.if.mthr.Runs-lappendE) simp
  from τRuns' obtain σ''
    where red-a: mthr.if.redT σ' (t-a, ta-a) σ''
    and τRuns'': mthr.if.mthr.Runs σ'' ?E'
    unfolding E'-a-m by cases
  from New-ta-a a-n have NormalAction (NewHeapElem ad CTn) ∈ set {|ta-a|}_o
```

unfolding *in-set-conv-nth* **by** *blast*
with *red-a* **obtain** *x-a x'-a m'-a*
where *red'-a*: *mthr.init-fin t-a* (*x-a*, *shr σ'*) *ta-a* (*x'-a*, *m'-a*)
and *σ'''*: *redT-upd σ' t-a ta-a x'-a m'-a σ''*
and *ts-t-a*: *thr σ' t-a* = ⌊(*x-a*, *no-wait-locks*)⌋
by *cases auto*
from *red'-a* ‹*NormalAction* (*NewHeapElem ad CTn*) ∈ *set* {|*ta-a*|}$_o$›
obtain *ta'-a X-a X'-a*
where *x-a*: *x-a* = (*Running*, *X-a*)
and *x'-a*: *x'-a* = (*Running*, *X'-a*)
and *ta-a*: *ta-a* = *convert-TA-initial* (*convert-obs-initial ta'-a*)
and *red''-a*: *t-a* ⊢ (*X-a*, *shr σ'*) −*ta'-a*→ (*X'-a*, *m'-a*)
by *cases fastforce+*
from *ta-a New-ta-a a-n* **have** *New-ta'-a*: {|*ta'-a*|}$_o$! *a-n* = *NewHeapElem ad CTn*
and *a-n'*: *a-n* < *length* {|*ta'-a*|}$_o$ **by** *auto*
hence *NewHeapElem ad CTn* ∈ *set* {|*ta'-a*|}$_o$ **unfolding** *in-set-conv-nth* **by** *blast*
with *red''-a* **have** *allocated-m'-a-ad*: *ad* ∈ *allocated m'-a*
by(*auto dest*: *red-New-allocatedD*)

have *a-m* ≤ *a'-m*
proof(*rule ccontr*)
assume ¬ *?thesis*
hence *a'-m* < *a-m* **by** *simp*
hence (∑ *i*<*a-m*. *length* {|*snd* (*lnth E' i*)|}$_o$) = (∑ *i*<*a'-m*. *length* {|*snd* (*lnth E' i*)|}$_o$) + (∑ *i* = *a'-m*..<*a-m*. *length* {|*snd* (*lnth E' i*)|}$_o$)
by(*simp add*: *sum-upto-add-nat*)
hence *a'* − *n* < *a* − *n* **using** ‹*a'-m* < *a-m*› *a'-n E'-a'-m* **unfolding** *a-conv a'-conv*
by(*subst* (*asm*) *sum.atLeast-Suc-lessThan*) *simp-all*
with *a-a'* **show** *False* **by** *simp*
qed

have *a'-less*: *a'* − *n* < (*a* − *n*) − *a-n* + *length* {|*ta-a*|}$_o$
proof(*rule ccontr*)
assume ¬ *?thesis*
hence *a'-greater*: (*a* − *n*) − *a-n* + *length* {|*ta-a*|}$_o$ ≤ *a'* − *n* **by** *simp*

have *a-m* < *a'-m*
proof(*rule ccontr*)
assume ¬ *?thesis*
with ‹*a-m* ≤ *a'-m*› **have** *a-m* = *a'-m* **by** *simp*
with *a'-greater a-n a'-n E'-a'-m E'-a-m* **show** *False*
unfolding *a-conv a'-conv* **by** *simp*
qed
hence *a'-m-a-m*: *enat* (*a'-m* − *Suc a-m*) < *llength ?E'* **using** *a'-m*
by(*cases llength E'*) *simp-all*
from ‹*a-m* < *a'-m*› *a'-m E'-a'-m*
have *E'-a'-m'*: *lnth ?E'* (*a'-m* − *Suc a-m*) = (*t-a'*, *ta-a'*) **by** *simp*

have *E'-unfold'*: *?E'* = *lappend* (*ltake* (*enat* (*a'-m* − *Suc a-m*)) *?E'*) (*LCons* (*lnth ?E'* (*a'-m* − *Suc a-m*)) (*ldropn* (*Suc* (*a'-m* − *Suc a-m*)) *?E'*))
unfolding *ldropn-Suc-conv-ldropn*[*OF a'-m-a-m*] *lappend-ltake-enat-ldropn* **..**
hence *mthr.if.mthr.Runs σ''* (*lappend* (*ltake* (*enat* (*a'-m* − *Suc a-m*)) *?E'*) (*LCons* (*lnth ?E'* (*a'-m* − *Suc a-m*)) (*ldropn* (*Suc* (*a'-m* − *Suc a-m*)) *?E'*)))
using *τRuns''* **by** *simp*

```
      then obtain σ'''
        where σ''-σ''': mthr.if.mthr.Trsys σ'' (list-of (ltake (enat (a'-m − Suc a-m)) ?E')) σ'''
       and τRuns''': mthr.if.mthr.Runs σ''' (LCons (lnth ?E' (a'-m − Suc a-m)) (ldropn (Suc (a'-m
− Suc a-m)) ?E'))
        by(rule mthr.if.mthr.Runs-lappendE) simp
      from τRuns''' obtain σ''''
        where red-a': mthr.if.redT σ''' (t-a', ta-a') σ''''
        and τRuns'''': mthr.if.mthr.Runs σ'''' (ldropn (Suc (a'-m − Suc a-m)) ?E')
        unfolding E'-a'-m' by cases
      from New-ta-a' a'-n have NormalAction (NewHeapElem ad CTn') ∈ set ⦃ta-a'⦄o
        unfolding in-set-conv-nth by blast
      with red-a' obtain x-a' x'-a' m'-a'
        where red'-a': mthr.init-fin t-a' (x-a', shr σ''') ta-a' (x'-a', m'-a')
        and σ''''': redT-upd σ''' t-a' ta-a' x'-a' m'-a' σ''''
        and ts-t-a': thr σ''' t-a' = ⌊(x-a', no-wait-locks)⌋
        by cases auto
      from red'-a' ‹NormalAction (NewHeapElem ad CTn') ∈ set ⦃ta-a'⦄o›
      obtain ta'-a' X-a' X'-a'
        where x-a': x-a' = (Running, X-a')
        and x'-a': x'-a' = (Running, X'-a')
        and ta-a': ta-a' = convert-TA-initial (convert-obs-initial ta'-a')
        and red''-a': t-a' ⊢ (X-a', shr σ''') −ta'-a'→ (X'-a', m'-a')
        by cases fastforce+
      from ta-a' New-ta-a' a'-n have New-ta'-a': ⦃ta'-a'⦄o ! a'-n = NewHeapElem ad CTn'
        and a'-n': a'-n < length ⦃ta'-a'⦄o by auto
      hence NewHeapElem ad CTn' ∈ set ⦃ta'-a'⦄o unfolding in-set-conv-nth by blast
      with red''-a' have allocated-ad': ad ∉ allocated (shr σ''')
        by(auto dest: red-New-allocatedD)

      have allocated m'-a = allocated (shr σ'') using σ''' by auto
      also have ... ⊆ allocated (shr σ''')
        using σ''-σ''' unfolding mthr.if.RedT-def[symmetric] by(rule init-fin-RedT-allocated-mono)
      finally have ad ∈ allocated (shr σ''') using allocated-m'-a-ad by blast
      with allocated-ad' show False by contradiction
    qed

    from ‹a-m ≤ a'-m› have [simp]: a-m = a'-m
    proof(rule le-antisym)
      show a'-m ≤ a-m
      proof(rule ccontr)
        assume ¬ ?thesis
        hence a-m < a'-m by simp
       hence (∑ i<a'-m. length ⦃snd (lnth E' i)⦄o) = (∑ i<a-m. length ⦃snd (lnth E' i)⦄o) + (∑ i
= a-m..<a'-m. length ⦃snd (lnth E' i)⦄o)
          by(simp add: sum-upto-add-nat)
        with a'-less ‹a-m < a'-m› E'-a-m a-n a'-n show False
          unfolding a'-conv a-conv by(subst (asm) sum.atLeast-Suc-lessThan) simp-all
      qed
    qed
    with E'-a-m E'-a'-m have [simp]: t-a' = t-a ta-a' = ta-a by simp-all
    from New-ta-a' a'-n ta-a have a'-n': a'-n < length ⦃ta'-a⦄o
      and New-ta'-a': ⦃ta'-a⦄o ! a'-n = NewHeapElem ad CTn' by auto
    with red''-a New-ta'-a a-n' have a'-n = a-n
      by(auto dest: red-New-same-addr-same)
```

```
    with ‹a-m = a'-m› have a − n = a' − n unfolding a-conv a'-conv by simp
    thus ?thesis using ‹n ≤ a› ‹n ≤ a'› by simp
   qed
  qed
qed

end
```

Knowledge of addresses of a multithreaded state

```
fun ka-Val :: 'addr val ⇒ 'addr set
where
  ka-Val (Addr a) = {a}
| ka-Val - = {}

fun new-obs-addr :: ('addr, 'thread-id) obs-event ⇒ 'addr set
where
  new-obs-addr (ReadMem ad al (Addr ad')) = {ad'}
| new-obs-addr (NewHeapElem ad hT) = {ad}
| new-obs-addr - = {}

lemma new-obs-addr-cases[consumes 1, case-names ReadMem NewHeapElem, cases set]:
  assumes ad ∈ new-obs-addr ob
  obtains ad' al where ob = ReadMem ad' al (Addr ad)
  | CTn where ob = NewHeapElem ad CTn
using assms
by(cases ob rule: new-obs-addr.cases) auto

definition new-obs-addrs :: ('addr, 'thread-id) obs-event list ⇒ 'addr set
where
  new-obs-addrs obs = ⋃(new-obs-addr ‘ set obs)

fun new-obs-addr-if :: ('addr, 'thread-id) obs-event action ⇒ 'addr set
where
  new-obs-addr-if (NormalAction a) = new-obs-addr a
| new-obs-addr-if - = {}

definition new-obs-addrs-if :: ('addr, 'thread-id) obs-event action list ⇒ 'addr set
where
  new-obs-addrs-if obs = ⋃(new-obs-addr-if ‘ set obs)

lemma ka-Val-subset-new-obs-Addr-ReadMem:
  ka-Val v ⊆ new-obs-addr (ReadMem ad al v)
by(cases v) simp-all

lemma typeof-ka: typeof v ≠ None ⟹ ka-Val v = {}
by(cases v) simp-all

lemma ka-Val-undefined-value [simp]:
  ka-Val undefined-value = {}
apply(cases undefined-value :: 'a val)
apply(bestsimp simp add: undefined-value-not-Addr dest: subst)+
done

locale known-addrs-base =
```

fixes *known-addrs* :: *'t* ⇒ *'x* ⇒ *'addr set*
begin

definition *known-addrs-thr* :: (*'l*, *'t*, *'x*) *thread-info* ⇒ *'addr set*
where *known-addrs-thr ts* = (⋃ *t* ∈ *dom ts*. *known-addrs t* (*fst* (*the* (*ts t*))))

definition *known-addrs-state* :: (*'l*,*'t*,*'x*,*'m*,*'w*) *state* ⇒ *'addr set*
where *known-addrs-state s* = *known-addrs-thr* (*thr s*)

lemma *known-addrs-state-simps* [*simp*]:
known-addrs-state (*ls*, (*ts*, *m*), *ws*) = *known-addrs-thr ts*
by(*simp add*: *known-addrs-state-def*)

lemma *known-addrs-thr-cases*[*consumes 1*, *case-names known-addrs*, *cases set*: *known-addrs-thr*]:
assumes *ad* ∈ *known-addrs-thr ts*
and ⋀*t x ln*. ⟦ *ts t* = ⌊(*x*, *ln*)⌋; *ad* ∈ *known-addrs t x* ⟧ ⟹ *thesis*
shows *thesis*
using *assms*
by(*auto simp add*: *known-addrs-thr-def ran-def*)

lemma *known-addrs-stateI*:
⋀*ln*. ⟦ *ad* ∈ *known-addrs t x*; *thr s t* = ⌊(*x*, *ln*)⌋ ⟧ ⟹ *ad* ∈ *known-addrs-state s*
by(*fastforce simp add*: *known-addrs-state-def known-addrs-thr-def intro*: *rev-bexI*)

fun *known-addrs-if* :: *'t* ⇒ *status* × *'x* ⇒ *'addr set*
where *known-addrs-if t* (*s*, *x*) = *known-addrs t x*

end

locale *if-known-addrs-base* =
known-addrs-base known-addrs
+
multithreaded-base final r convert-RA
for *known-addrs* :: *'t* ⇒ *'x* ⇒ *'addr set*
and *final* :: *'x* ⇒ *bool*
and *r* :: (*'addr*, *'t*, *'x*, *'heap*, *'addr*, *'obs*) *semantics* (‹- ⊢ - –-→ -› [*50*,*0*,*0*,*50*] *80*)
and *convert-RA* :: *'addr released-locks* ⇒ *'obs list*

sublocale *if-known-addrs-base* < *if*: *known-addrs-base known-addrs-if* .

locale *known-addrs* =
allocated-multithreaded
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
allocated
final r
P
+
if-known-addrs-base known-addrs final r convert-RA

for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*

and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *allocated* :: *'heap* ⇒ *'addr set*
and *known-addrs* :: *'thread-id* ⇒ *'x* ⇒ *'addr set*
and *final* :: *'x* ⇒ *bool*
and *r* :: (*'addr*, *'thread-id*, *'x*, *'heap*, *'addr*, (*'addr*, *'thread-id*) *obs-event*) *semantics* (‹- ⊢ - –-→ -› [*50*,*0*,*0*,*50*] *80*)
and *P* :: *'md prog*
+
assumes *red-known-addrs-new*:
t ⊢ ⟨*x*, *m*⟩ –*ta*→ ⟨*x'*, *m'*⟩
⟹ *known-addrs t x'* ⊆ *known-addrs t x* ∪ *new-obs-addrs* {|*ta*|}*o*
and *red-known-addrs-new-thread*:
⟦ *t* ⊢ ⟨*x*, *m*⟩ –*ta*→ ⟨*x'*, *m'*⟩; *NewThread t' x'' m''* ∈ *set* {|*ta*|}*t* ⟧
⟹ *known-addrs t' x''* ⊆ *known-addrs t x*
and *red-read-knows-addr*:
⟦ *t* ⊢ ⟨*x*, *m*⟩ –*ta*→ ⟨*x'*, *m'*⟩; *ReadMem ad al v* ∈ *set* {|*ta*|}*o* ⟧
⟹ *ad* ∈ *known-addrs t x*
and *red-write-knows-addr*:
⟦ *t* ⊢ ⟨*x*, *m*⟩ –*ta*→ ⟨*x'*, *m'*⟩; {|*ta*|}*o* ! *n* = *WriteMem ad al* (*Addr ad'*); *n* < *length* {|*ta*|}*o* ⟧
⟹ *ad'* ∈ *known-addrs t x* ∨ *ad'* ∈ *new-obs-addrs* (*take n* {|*ta*|}*o*)
— second possibility necessary for *heap-clone*
begin

notation *mthr.redT-syntax1* (‹- –▷-→ -› [*50*,*0*,*0*,*50*] *80*)

lemma *if-red-known-addrs-new*:
assumes *t* ⊢ (*x*, *m*) –*ta*→*i* (*x'*, *m'*)
shows *known-addrs-if t x'* ⊆ *known-addrs-if t x* ∪ *new-obs-addrs-if* {|*ta*|}*o*
using *assms*
by *cases*(*auto dest*!: *red-known-addrs-new simp add*: *new-obs-addrs-if-def new-obs-addrs-def*)

lemma *if-red-known-addrs-new-thread*:
assumes *t* ⊢ (*x*, *m*) –*ta*→*i* (*x'*, *m'*) *NewThread t' x'' m''* ∈ *set* {|*ta*|}*t*
shows *known-addrs-if t' x''* ⊆ *known-addrs-if t x*
using *assms*
by *cases*(*fastforce dest*: *red-known-addrs-new-thread*)+

lemma *if-red-read-knows-addr*:
assumes *t* ⊢ (*x*, *m*) –*ta*→*i* (*x'*, *m'*) *NormalAction* (*ReadMem ad al v*) ∈ *set* {|*ta*|}*o*
shows *ad* ∈ *known-addrs-if t x*
using *assms*
by *cases*(*fastforce dest*: *red-read-knows-addr*)+

lemma *if-red-write-knows-addr*:
assumes *t* ⊢ (*x*, *m*) –*ta*→*i* (*x'*, *m'*)
and {|*ta*|}*o* ! *n* = *NormalAction* (*WriteMem ad al* (*Addr ad'*)) *n* < *length* {|*ta*|}*o*
shows *ad'* ∈ *known-addrs-if t x* ∨ *ad'* ∈ *new-obs-addrs-if* (*take n* {|*ta*|}*o*)
using *assms*
by *cases*(*auto dest*: *red-write-knows-addr simp add*: *new-obs-addrs-if-def new-obs-addrs-def take-map*)

```
lemma if-redT-known-addrs-new:
  assumes redT: mthr.if.redT s (t, ta) s'
  shows if.known-addrs-state s' ⊆ if.known-addrs-state s ∪ new-obs-addrs-if {|ta|}_o
using redT
proof(cases)
  case redT-acquire thus ?thesis
    by(cases s)(fastforce simp add: if.known-addrs-thr-def split: if-split-asm intro: rev-bexI)
next
  case (redT-normal x x' m)
  note red = ‹t ⊢ (x, shr s) −ta→i (x', m)›
  show ?thesis
  proof
    fix ad
    assume ad ∈ if.known-addrs-state s'
    hence ad ∈ if.known-addrs-thr (thr s') by(simp add: if.known-addrs-state-def)
    then obtain t' x'' ln'' where ts't': thr s' t' = ⌊(x'', ln'')⌋
      and ad: ad ∈ known-addrs-if t' x''
      by(rule if.known-addrs-thr-cases)
    show ad ∈ if.known-addrs-state s ∪ new-obs-addrs-if {|ta|}_o
    proof(cases thr s t')
      case None
      with redT-normal ‹thr s' t' = ⌊(x'', ln'')⌋›
      obtain m'' where NewThread t' x'' m'' ∈ set {|ta|}_t
        by(fastforce dest: redT-updTs-new-thread split: if-split-asm)
      with red have known-addrs-if t' x'' ⊆ known-addrs-if t x by(rule if-red-known-addrs-new-thread)
      also have ... ⊆ known-addrs-if t x ∪ new-obs-addrs-if {|ta|}_o by simp
      finally have ad ∈ known-addrs-if t x ∪ new-obs-addrs-if {|ta|}_o using ad by blast
      thus ?thesis using ‹thr s t = ⌊(x, no-wait-locks)⌋› by(blast intro: if.known-addrs-stateI)
    next
      case (Some xln)
      show ?thesis
      proof(cases t = t')
        case True
        with redT-normal ts't' if-red-known-addrs-new[OF red] ad
        have ad ∈ known-addrs-if t x ∪ new-obs-addrs-if {|ta|}_o by auto
        thus ?thesis using ‹thr s t = ⌊(x, no-wait-locks)⌋› by(blast intro: if.known-addrs-stateI)
      next
        case False
        with ts't' redT-normal ad Some show ?thesis
          by(fastforce dest: redT-updTs-Some[where ts=thr s and t=t'] intro: if.known-addrs-stateI)
      qed
    qed
  qed
qed

lemma if-redT-read-knows-addr:
  assumes redT: mthr.if.redT s (t, ta) s'
  and read: NormalAction (ReadMem ad al v) ∈ set {|ta|}_o
  shows ad ∈ if.known-addrs-state s
using redT
proof(cases)
  case redT-acquire thus ?thesis using read by auto
next
  case (redT-normal x x' m')
```

```
  with if-red-read-knows-addr[OF ‹t ⊢ (x, shr s) −ta→i (x', m')› read]
  show ?thesis
    by(auto simp add: if.known-addrs-state-def if.known-addrs-thr-def intro: bexI[where x=t])
qed

lemma init-fin-redT-known-addrs-subset:
  assumes mthr.if.redT s (t, ta) s'
  shows if.known-addrs-state s' ⊆ if.known-addrs-state s ∪ known-addrs-if t (fst (the (thr s' t)))
using assms
apply(cases)
 apply(rule subsetI)
 apply(clarsimp simp add: if.known-addrs-thr-def split: if-split-asm)
 apply(rename-tac status x status' x' m' a ws' t'' status'' x'' ln'')
 apply(case-tac thr s t'')
  apply(drule (2) redT-updTs-new-thread)
  apply clarsimp
  apply(drule (1) if-red-known-addrs-new-thread)
  apply simp
  apply(drule (1) subsetD)
  apply(rule-tac x=(status, x) in if.known-addrs-stateI)
   apply(simp)
  apply simp
 apply(frule-tac t=t'' in redT-updTs-Some, assumption)
 apply clarsimp
 apply(rule-tac x=(status'', x'') in if.known-addrs-stateI)
  apply simp
 apply simp
apply(auto simp add: if.known-addrs-state-def if.known-addrs-thr-def split: if-split-asm)
done

lemma w-values-no-write-unchanged:
  assumes no-write: ⋀w. ⟦ w ∈ set obs; is-write-action w; adal ∈ action-loc-aux P w ⟧ ⟹ False
  shows w-values P vs obs adal = vs adal
using assms
proof(induct obs arbitrary: vs)
  case Nil show ?case by simp
next
  case (Cons ob obs)
  from Cons.prems[of ob]
  have w-value P vs ob adal = vs adal
    by(cases adal)(cases ob rule: w-value-cases, auto simp add: addr-locs-def split: htype.split-asm,
blast+)
  moreover
  have w-values P (w-value P vs ob) obs adal = w-value P vs ob adal
  proof(rule Cons.hyps)
    fix w
    assume w ∈ set obs is-write-action w adal ∈ action-loc-aux P w
    with Cons.prems[of w] ‹w-value P vs ob adal = vs adal›
    show False by simp
  qed
  ultimately show ?case by simp
qed

lemma redT-non-speculative-known-addrs-allocated:
```

```
  assumes red: mthr.if.redT s (t, ta) s'
  and tasc: non-speculative P vs (llist-of {|ta|}_o)
  and ka: if.known-addrs-state s ⊆ allocated (shr s)
  and vs: w-addrs vs ⊆ allocated (shr s)
  shows if.known-addrs-state s' ⊆ allocated (shr s') (is ?thesis1)
  and w-addrs (w-values P vs {|ta|}_o) ⊆ allocated (shr s') (is ?thesis2)
proof −
  have ?thesis1 ∧ ?thesis2 using red
  proof(cases)
    case (redT-acquire x ln n)
    hence if.known-addrs-state s' = if.known-addrs-state s
      by(auto 4 4 simp add: if.known-addrs-state-def if.known-addrs-thr-def split: if-split-asm dest:
bspec)
    also note ka
    also from redT-acquire have shr s = shr s' by simp
    finally have if.known-addrs-state s' ⊆ allocated (shr s') .
    moreover have w-values P vs {|ta|}_o = vs using redT-acquire
      by(fastforce intro!: w-values-no-write-unchanged del: equalityI dest: convert-RA-not-write)
    ultimately show ?thesis using vs by(simp add: ‹shr s = shr s'›)
  next
    case (redT-normal x x' m')
    note red = ‹t ⊢ (x, shr s) −ta→i (x', m')›
      and tst = ‹thr s t = ⌊(x, no-wait-locks)⌋›
    have allocated-subset: allocated (shr s) ⊆ allocated (shr s')
      using ‹mthr.if.redT s (t, ta) s'› by(rule init-fin-redT-allocated-mono)
    with vs have vs': w-addrs vs ⊆ allocated (shr s') by blast
    { fix obs obs'
      assume {|ta|}_o = obs @ obs'
      moreover with tasc have non-speculative P vs (llist-of obs)
      by(simp add: lappend-llist-of-llist-of[symmetric] non-speculative-lappend del: lappend-llist-of-llist-of)
      ultimately have w-addrs (w-values P vs obs) ∪ new-obs-addrs-if obs ⊆ allocated (shr s')
        (is ?concl obs)
      proof(induct obs arbitrary: obs' rule: rev-induct)
        case Nil thus ?case using vs' by(simp add: new-obs-addrs-if-def)
      next
        case (snoc ob obs)
        note ta = ‹{|ta|}_o = (obs @ [ob]) @ obs'›
        note tasc = ‹non-speculative P vs (llist-of (obs @ [ob]))›
        from snoc have IH: ?concl obs
       by(simp add: lappend-llist-of-llist-of[symmetric] non-speculative-lappend del: lappend-llist-of-llist-of)
        hence ?concl (obs @ [ob])
        proof(cases ob rule: mrw-value-cases)
          case (1 ad' al v)
          note ob = ‹ob = NormalAction (WriteMem ad' al v)›
          with ta have Write: {|ta|}_o ! length obs = NormalAction (WriteMem ad' al v) by simp
          show ?thesis
          proof
            fix ad''
            assume ad'' ∈ w-addrs (w-values P vs (obs @ [ob])) ∪ new-obs-addrs-if (obs @ [ob])
            hence ad'' ∈ w-addrs (w-values P vs obs) ∪ new-obs-addrs-if obs ∨ v = Addr ad''
              by(auto simp add: ob w-addrs-def ran-def new-obs-addrs-if-def split: if-split-asm)
            thus ad'' ∈ allocated (shr s')
            proof
              assume ad'' ∈ w-addrs (w-values P vs obs) ∪ new-obs-addrs-if obs
```

```
            also note IH finally show ?thesis .
          next
            assume v: v = Addr ad''
            with Write have ⦃ta⦄o ! length obs = NormalAction (WriteMem ad' al (Addr ad'')) by
simp
              with red have ad'' ∈ known-addrs-if t x ∨ ad'' ∈ new-obs-addrs-if (take (length obs)
⦃ta⦄o)
              by(rule if-red-write-knows-addr)(simp add: ta)
            thus ?thesis
            proof
              assume ad'' ∈ known-addrs-if t x
              hence ad'' ∈ if.known-addrs-state s using tst by(rule if.known-addrs-stateI)
              with ka allocated-subset show ?thesis by blast
            next
              assume ad'' ∈ new-obs-addrs-if (take (length obs) ⦃ta⦄o)
              with ta have ad'' ∈ new-obs-addrs-if obs by simp
              with IH show ?thesis by blast
            qed
          qed
        qed
      next
        case (2 ad hT)

        hence ob: ob = NormalAction (NewHeapElem ad hT) by simp
        hence w-addrs (w-values P vs (obs @ [ob])) ⊆ w-addrs (w-values P vs obs)
          by(cases hT)(auto simp add: w-addrs-def default-val-not-Addr Addr-not-default-val)
        moreover from ob ta have NormalAction (NewHeapElem ad hT) ∈ set ⦃ta⦄o by simp
        from init-fin-red-New-allocatedD[OF red this] have ad ∈ allocated m' ..
        with redT-normal have ad ∈ allocated (shr s') by auto
        ultimately show ?thesis using IH ob by(auto simp add: new-obs-addrs-if-def)
      next
        case (4 ad al v)
        note ob = ‹ob = NormalAction (ReadMem ad al v)›
        { fix ad'
          assume v: v = Addr ad'
          with tasc ob have mrw: Addr ad' ∈ w-values P vs obs (ad, al)
              by(auto simp add: lappend-llist-of-llist-of[symmetric] non-speculative-lappend simp del:
lappend-llist-of-llist-of)
          hence ad' ∈ w-addrs (w-values P vs obs)
            by(auto simp add: w-addrs-def)
          with IH have ad' ∈ allocated (shr s') by blast }
        with ob IH show ?thesis by(cases v)(simp-all add: new-obs-addrs-if-def)
      qed(simp-all add: new-obs-addrs-if-def)
      thus ?case by simp
    qed }
  note this[of ⦃ta⦄o []]
  moreover have if.known-addrs-state s' ⊆ if.known-addrs-state s ∪ new-obs-addrs-if ⦃ta⦄o
    using ‹mthr.if.redT s (t, ta) s'› by(rule if-redT-known-addrs-new)
  ultimately show ?thesis using ka allocated-subset by blast
 qed
 thus ?thesis1 ?thesis2 by simp-all
qed
```

lemma *RedT-non-speculative-known-addrs-allocated*:
assumes *red*: *mthr.if.RedT s ttas s′*
and *tasc*: *non-speculative P vs* (*llist-of* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*)))
and *ka*: *if.known-addrs-state s* ⊆ *allocated* (*shr s*)
and *vs*: *w-addrs vs* ⊆ *allocated* (*shr s*)
shows *if.known-addrs-state s′* ⊆ *allocated* (*shr s′*) (**is** *?thesis1 s′*)
and *w-addrs* (*w-values P vs* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*))) ⊆ *allocated* (*shr s′*) (**is** *?thesis2 s′ ttas*)
proof −
from *red tasc* **have** *?thesis1 s′* ∧ *?thesis2 s′ ttas*
proof(*induct rule*: *mthr.if.RedT-induct′*)
case *refl* **thus** *?case* **using** *ka vs* **by** *simp*
next
case (*step ttas s′ t ta s″*)
hence *non-speculative P* (*w-values P vs* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*))) (*llist-of* {|*ta*|}$_o$)
and *?thesis1 s′ ?thesis2 s′ ttas*
by(*simp-all add*: *lappend-llist-of-llist-of*[*symmetric*] *non-speculative-lappend del*: *lappend-llist-of-llist-of*)
from *redT-non-speculative-known-addrs-allocated*[*OF* ‹*mthr.if.redT s′* (*t*, *ta*) *s″*› *this*]
show *?case* **by** *simp*
qed
thus *?thesis1 s′ ?thesis2 s′ ttas* **by** *simp-all*
qed

lemma *read-ex-NewHeapElem* [*consumes 5*, *case-names start Red*]:
assumes *RedT*: *mthr.if.RedT* (*init-fin-lift-state status* (*start-state f P C M vs*)) *ttas s*
and *red*: *mthr.if.redT s* (*t*, *ta*) *s′*
and *read*: *NormalAction* (*ReadMem ad al v*) ∈ *set* {|*ta*|}$_o$
and *sc*: *non-speculative P* (λ-. {}) (*llist-of* (*map snd* (*lift-start-obs start-tid start-heap-obs*) @ *concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*)))
and *known*: *known-addrs start-tid* (*f* (*fst* (*method P C M*)) *M* (*fst* (*snd* (*method P C M*))) (*fst* (*snd* (*snd* (*method P C M*)))) (*the* (*snd* (*snd* (*snd* (*method P C M*))))) *vs*) ⊆ *allocated start-heap*
obtains (*start*) *CTn* **where** *NewHeapElem ad CTn* ∈ *set start-heap-obs*
| (*Red*) *ttas′ s″ t′ ta′ s‴ ttas″ CTn*
where *mthr.if.RedT* (*init-fin-lift-state status* (*start-state f P C M vs*)) *ttas′ s″*
and *mthr.if.redT s″* (*t′*, *ta′*) *s‴*
and *mthr.if.RedT s‴ ttas″ s*
and *ttas* = *ttas′* @ (*t′*, *ta′*) # *ttas″*
and *NormalAction* (*NewHeapElem ad CTn*) ∈ *set* {|*ta′*|}$_o$
proof −
let *?start-state* = *init-fin-lift-state status* (*start-state f P C M vs*)
let *?obs-prefix* = *lift-start-obs start-tid start-heap-obs*
let *?vs-start* = *w-values P* (λ-. {}) (*map snd ?obs-prefix*)

from *sc* **have** *non-speculative P* (*w-values P* (λ-. {}) (*map snd* (*lift-start-obs start-tid start-heap-obs*))) (*llist-of* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*)))
by(*simp add*: *non-speculative-lappend lappend-llist-of-llist-of*[*symmetric*] *del*: *lappend-llist-of-llist-of*)
with *RedT* **have** *if.known-addrs-state s* ⊆ *allocated* (*shr s*)
proof(*rule RedT-non-speculative-known-addrs-allocated*)
show *if.known-addrs-state ?start-state* ⊆ *allocated* (*shr ?start-state*)
using *known*
by(*auto simp add*: *if.known-addrs-state-def if.known-addrs-thr-def start-state-def init-fin-lift-state-def split-beta split*: *if-split-asm*)

have *w-addrs ?vs-start* ⊆ *w-addrs* (λ-. {}) **by**(*rule w-addrs-lift-start-heap-obs*)

```
    thus w-addrs ?vs-start ⊆ allocated (shr ?start-state) by simp
  qed
  also from red read obtain x-ra x'-ra m'-ra
    where red'-ra: t ⊢ (x-ra, shr s) −ta→i (x'-ra, m'-ra)
    and s': redT-upd s t ta x'-ra m'-ra s'
    and ts-t: thr s t = ⌊(x-ra, no-wait-locks)⌋
    by cases auto
  from red'-ra read
  have ad ∈ known-addrs-if t x-ra by(rule if-red-read-knows-addr)
  hence ad ∈ if.known-addrs-state s using ts-t by(rule if.known-addrs-stateI)
  finally have ad ∈ allocated (shr s) .

  show ?thesis
  proof(cases ad ∈ allocated start-heap)
    case True
    then obtain CTn where NewHeapElem ad CTn ∈ set start-heap-obs
      unfolding start-addrs-allocated by(blast dest: start-addrs-NewHeapElem-start-heap-obsD)
    thus ?thesis by(rule start)
  next
    case False
   hence ad ∉ allocated (shr ?start-state) by(simp add: start-state-def split-beta shr-init-fin-lift-state)
    with RedT ‹ad ∈ allocated (shr s)› obtain t' ta' CTn
      where tta: (t', ta') ∈ set ttas
      and new: NormalAction (NewHeapElem ad CTn) ∈ set {|ta'|}_o
      by(blast dest: init-fin-RedT-allocated-NewHeapElemD)
    from tta obtain ttas' ttas'' where ttas: ttas = ttas' @ (t', ta') # ttas'' by(auto dest: split-list)
    with RedT obtain s'' s'''
      where mthr.if.RedT ?start-state ttas' s''
      and mthr.if.redT s'' (t', ta') s'''
      and mthr.if.RedT s''' ttas'' s
      unfolding mthr.if.RedT-def by(auto elim!: rtrancl3p-appendE dest!: converse-rtrancl3p-step)
    thus thesis using ttas new by(rule Red)
  qed
qed

end

locale known-addrs-typing =
  known-addrs
    addr2thread-id thread-id2addr
    spurious-wakeups
    empty-heap allocate typeof-addr heap-read heap-write
    allocated known-addrs
    final r P
  for addr2thread-id :: ('addr :: addr) ⇒ 'thread-id
  and thread-id2addr :: 'thread-id ⇒ 'addr
  and spurious-wakeups :: bool
  and empty-heap :: 'heap
  and allocate :: 'heap ⇒ htype ⇒ ('heap × 'addr) set
  and typeof-addr :: 'heap ⇒ 'addr ⇀ htype
  and heap-read :: 'heap ⇒ 'addr ⇒ addr-loc ⇒ 'addr val ⇒ bool
  and heap-write :: 'heap ⇒ 'addr ⇒ addr-loc ⇒ 'addr val ⇒ 'heap ⇒ bool
  and allocated :: 'heap ⇒ 'addr set
  and known-addrs :: 'thread-id ⇒ 'x ⇒ 'addr set
```

and *final* :: *'x* ⇒ *bool*
and *r* :: (*'addr*, *'thread-id*, *'x*, *'heap*, *'addr*, (*'addr*, *'thread-id*) *obs-event*) *semantics* (‹- ⊢ - –-→ -› [*50*,*0*,*0*,*50*] *80*)
and *wfx* :: *'thread-id* ⇒ *'x* ⇒ *'heap* ⇒ *bool*
and *P* :: *'md prog*
+
assumes *wfs-non-speculative-invar*:
⟦ *t* ⊢ (*x*, *m*) –*ta*→ (*x'*, *m'*); *wfx t x m*;
vs-conf P m vs; *non-speculative P vs* (*llist-of* (*map NormalAction* {|*ta*|}$_o$)) ⟧
⟹ *wfx t x' m'*
and *wfs-non-speculative-spawn*:
⟦ *t* ⊢ (*x*, *m*) –*ta*→ (*x'*, *m'*); *wfx t x m*;
vs-conf P m vs; *non-speculative P vs* (*llist-of* (*map NormalAction* {|*ta*|}$_o$));
NewThread t'' x'' m'' ∈ *set* {|*ta*|}$_t$ ⟧
⟹ *wfx t'' x'' m''*
and *wfs-non-speculative-other*:
⟦ *t* ⊢ (*x*, *m*) –*ta*→ (*x'*, *m'*); *wfx t x m*;
vs-conf P m vs; *non-speculative P vs* (*llist-of* (*map NormalAction* {|*ta*|}$_o$));
wfx t'' x'' m ⟧
⟹ *wfx t'' x'' m'*
and *wfs-non-speculative-vs-conf*:
⟦ *t* ⊢ (*x*, *m*) –*ta*→ (*x'*, *m'*); *wfx t x m*;
vs-conf P m vs; *non-speculative P vs* (*llist-of* (*take n* (*map NormalAction* {|*ta*|}$_o$))) ⟧
⟹ *vs-conf P m'* (*w-values P vs* (*take n* (*map NormalAction* {|*ta*|}$_o$)))
and *red-read-typeable*:
⟦ *t* ⊢ (*x*, *m*) –*ta*→ (*x'*, *m'*); *wfx t x m*; *ReadMem ad al v* ∈ *set* {|*ta*|}$_o$ ⟧
⟹ ∃ *T*. *P*,*m* ⊢ *ad*@*al* : *T*
and *red-NewHeapElemD*:
⟦ *t* ⊢ (*x*, *m*) –*ta*→ (*x'*, *m'*); *wfx t x m*; *NewHeapElem ad hT* ∈ *set* {|*ta*|}$_o$ ⟧
⟹ *typeof-addr m' ad* = ⌊*hT*⌋
and *red-hext-incr*:
⟦ *t* ⊢ (*x*, *m*) –*ta*→ (*x'*, *m'*); *wfx t x m*;
vs-conf P m vs; *non-speculative P vs* (*llist-of* (*map NormalAction* {|*ta*|}$_o$)) ⟧
⟹ *m* ≤ *m'*
begin

lemma *redT-wfs-non-speculative-invar*:
assumes *redT*: *mthr.redT s* (*t*, *ta*) *s'*
and *wfx*: *ts-ok wfx* (*thr s*) (*shr s*)
and *vs*: *vs-conf P* (*shr s*) *vs*
and *ns*: *non-speculative P vs* (*llist-of* (*map NormalAction* {|*ta*|}$_o$))
shows *ts-ok wfx* (*thr s'*) (*shr s'*)
using *redT*
proof(*cases*)
case (*redT-normal x x' m'*)
with *vs wfx ns* **show** *?thesis*
apply(*clarsimp intro*!: *ts-okI split*: *if-split-asm*)
apply(*erule wfs-non-speculative-invar*, *auto dest*: *ts-okD*)
apply(*rename-tac t' x' ln ws'*)
apply(*case-tac thr s t'*)
apply(*frule* (*2*) *redT-updTs-new-thread*, *clarify*)
apply(*frule* (*1*) *mthr.new-thread-memory*)
apply(*auto intro*: *wfs-non-speculative-other wfs-non-speculative-spawn dest*: *ts-okD simp add*: *redT-updTs-Some*)
done

next
case (*redT-acquire x ln n*)
thus *?thesis* **using** *wfx* **by**(*auto intro*!: *ts-okI dest*: *ts-okD split*: *if-split-asm*)
qed

lemma *redT-wfs-non-speculative-vs-conf*:
assumes *redT*: *mthr.redT s* (*t*, *ta*) *s′*
and *wfx*: *ts-ok wfx* (*thr s*) (*shr s*)
and *conf*: *vs-conf P* (*shr s*) *vs*
and *ns*: *non-speculative P vs* (*llist-of* (*take n* (*map NormalAction* {|*ta*|}$_o$)))
shows *vs-conf P* (*shr s′*) (*w-values P vs* (*take n* (*map NormalAction* {|*ta*|}$_o$)))
using *redT*
proof(*cases*)
case (*redT-normal x x′ m′*)
thus *?thesis* **using** *ns conf wfx* **by**(*auto dest*: *wfs-non-speculative-vs-conf ts-okD*)
next
case (*redT-acquire x l ln*)
have *w-values P vs* (*take n* (*map NormalAction* (*convert-RA ln* :: (*′addr*, *′thread-id*) *obs-event list*)))
= *vs*
by(*fastforce dest*: *in-set-takeD simp add*: *convert-RA-not-write intro*!: *w-values-no-write-unchanged del*: *equalityI*)
thus *?thesis* **using** *conf redT-acquire* **by**(*auto*)
qed

lemma *if-redT-non-speculative-invar*:
assumes *red*: *mthr.if.redT s* (*t*, *ta*) *s′*
and *ts-ok*: *ts-ok* (*init-fin-lift wfx*) (*thr s*) (*shr s*)
and *sc*: *non-speculative P vs* (*llist-of* {|*ta*|}$_o$)
and *vs*: *vs-conf P* (*shr s*) *vs*
shows *ts-ok* (*init-fin-lift wfx*) (*thr s′*) (*shr s′*)
proof −
let *?s* = *λs*. (*locks s*, (*λt*. *map-option* (*λ*((*status*, *x*), *ln*). (*x*, *ln*)) (*thr s t*), *shr s*), *wset s*, *interrupts s*)

from *ts-ok* **have** *ts-ok′*: *ts-ok wfx* (*thr* (*?s s*)) (*shr* (*?s s*)) **by**(*auto intro*!: *ts-okI dest*: *ts-okD*)
from *vs* **have** *vs′*: *vs-conf P* (*shr* (*?s s*)) *vs* **by** *simp*

from *red* **show** *?thesis*
proof(*cases*)
case (*redT-normal x x′ m*)
note *tst* = ‹*thr s t* = ⌊(*x*, *no-wait-locks*)⌋›
from ‹*t* ⊢ (*x*, *shr s*) −*ta*→*i* (*x′*, *m*)›
show *?thesis*
proof(*cases*)
case (*NormalAction X TA X′*)
from ‹*ta* = *convert-TA-initial* (*convert-obs-initial TA*)› ‹*mthr.if.actions-ok s t ta*›
have *mthr.actions-ok* (*?s s*) *t TA*
by(*auto elim*: *rev-iffD1*[*OF - thread-oks-ts-change*] *cond-action-oks-final-change*)

with *tst NormalAction* ‹*redT-upd s t ta x′ m s′*› **have** *mthr.redT* (*?s s*) (*t*, *TA*) (*?s s′*)
using *map-redT-updTs*[*of snd thr s* {|*ta*|}$_t$]
by(*auto intro*!: *mthr.redT.intros simp add*: *split-def map-prod-def o-def fun-eq-iff*)
moreover note *ts-ok′ vs′*
moreover from ‹*ta* = *convert-TA-initial* (*convert-obs-initial TA*)› **have** {|*ta*|}$_o$ = *map NormalAc-*

```
tion {|TA|}_o by(auto)
      with sc have non-speculative P vs (llist-of (map NormalAction {|TA|}_o)) by simp
      ultimately have ts-ok wfx (thr (?s s')) (shr (?s s'))
        by(auto dest: redT-wfs-non-speculative-invar)
      thus ?thesis using ‹{|ta|}_o = map NormalAction {|TA|}_o› by(auto intro!: ts-okI dest: ts-okD)
    next
      case InitialThreadAction
      with redT-normal ts-ok' vs show ?thesis
        by(auto 4 3 intro!: ts-okI dest: ts-okD split: if-split-asm)
    next
      case ThreadFinishAction
      with redT-normal ts-ok' vs show ?thesis
        by(auto 4 3 intro!: ts-okI dest: ts-okD split: if-split-asm)
    qed
  next
    case (redT-acquire x ln l)
    thus ?thesis using vs ts-ok by(auto 4 3 intro!: ts-okI dest: ts-okD split: if-split-asm)
  qed
qed

lemma if-redT-non-speculative-vs-conf:
  assumes red: mthr.if.redT s (t, ta) s'
  and ts-ok: ts-ok (init-fin-lift wfx) (thr s) (shr s)
  and sc: non-speculative P vs (llist-of (take n {|ta|}_o))
  and vs: vs-conf P (shr s) vs
  shows vs-conf P (shr s') (w-values P vs (take n {|ta|}_o))
proof −
  let ?s = λs. (locks s, (λt. map-option (λ((status, x), ln). (x, ln)) (thr s t), shr s), wset s, interrupts
s)

  from ts-ok have ts-ok': ts-ok wfx (thr (?s s)) (shr (?s s)) by(auto intro!: ts-okI dest: ts-okD)
  from vs have vs': vs-conf P (shr (?s s)) vs by simp

  from red show ?thesis
  proof(cases)
    case (redT-normal x x' m)
    note tst = ‹thr s t = ⌊(x, no-wait-locks)⌋›
    from ‹t ⊢ (x, shr s) −ta→i (x', m)›
    show ?thesis
    proof(cases)
      case (NormalAction X TA X')
      from ‹ta = convert-TA-initial (convert-obs-initial TA)› ‹mthr.if.actions-ok s t ta›
      have mthr.actions-ok (?s s) t TA
        by(auto elim: rev-iffD1[OF - thread-oks-ts-change] cond-action-oks-final-change)

      with tst NormalAction ‹redT-upd s t ta x' m s'› have mthr.redT (?s s) (t, TA) (?s s')
        using map-redT-updTs[of snd thr s {|ta|}_t]
        by(auto intro!: mthr.redT.intros simp add: split-def map-prod-def o-def fun-eq-iff)
      moreover note ts-ok' vs'
    moreover from ‹ta = convert-TA-initial (convert-obs-initial TA)› have {|ta|}_o = map NormalAc-
tion {|TA|}_o by(auto)
      with sc have non-speculative P vs (llist-of (take n (map NormalAction {|TA|}_o))) by simp
      ultimately have vs-conf P (shr (?s s')) (w-values P vs (take n (map NormalAction {|TA|}_o)))
        by(auto dest: redT-wfs-non-speculative-vs-conf)
```

```
    thus ?thesis using ‹{|ta|}_o = map NormalAction {|TA|}_o› by(auto)
   next
    case InitialThreadAction
    with redT-normal vs show ?thesis by(auto simp add: take-Cons')
   next
    case ThreadFinishAction
    with redT-normal vs show ?thesis by(auto simp add: take-Cons')
   qed
  next
   case (redT-acquire x l ln)
   have w-values P vs (take n (map NormalAction (convert-RA ln :: ('addr, 'thread-id) obs-event
list))) = vs
   by(fastforce simp add: convert-RA-not-write take-Cons' dest: in-set-takeD intro!: w-values-no-write-unchanged
del: equalityI)
   thus ?thesis using vs redT-acquire by auto
  qed
qed

lemma if-RedT-non-speculative-invar:
  assumes red: mthr.if.RedT s ttas s'
  and tsok: ts-ok (init-fin-lift wfx) (thr s) (shr s)
  and sc: non-speculative P vs (llist-of (concat (map (λ(t, ta). {|ta|}_o) ttas)))
  and vs: vs-conf P (shr s) vs
  shows ts-ok (init-fin-lift wfx) (thr s') (shr s') (is ?thesis1)
  and vs-conf P (shr s') (w-values P vs (concat (map (λ(t, ta). {|ta|}_o) ttas))) (is ?thesis2)
using red tsok sc vs unfolding mthr.if.RedT-def
proof(induct arbitrary: vs rule: rtrancl3p-converse-induct')
  case refl
  case 1 thus ?case by −
  case 2 thus ?case by simp
next
  case (step s tta s' ttas)
  obtain t ta where tta: tta = (t, ta) by(cases tta)

  case 1
  hence sc1: non-speculative P vs (llist-of {|ta|}_o)
   and sc2: non-speculative P (w-values P vs {|ta|}_o) (llist-of (concat (map (λ(t, ta). {|ta|}_o) ttas)))
   unfolding lconcat-llist-of[symmetric] lmap-llist-of[symmetric] llist.map-comp o-def llist-of.simps
llist.map(2) lconcat-LCons tta
   by(simp-all add: non-speculative-lappend list-of-lconcat o-def)
  from if-redT-non-speculative-invar[OF step(2)[unfolded tta] - sc1] if-redT-non-speculative-vs-conf[OF
step(2)[unfolded tta], where vs = vs and n=length {|ta|}_o] 1 step.hyps(3)[of w-values P vs {|ta|}_o] sc2
sc1
  show ?case by simp

  case 2
  hence sc1: non-speculative P vs (llist-of {|ta|}_o)
   and sc2: non-speculative P (w-values P vs {|ta|}_o) (llist-of (concat (map (λ(t, ta). {|ta|}_o) ttas)))
   unfolding lconcat-llist-of[symmetric] lmap-llist-of[symmetric] llist.map-comp o-def llist-of.simps
llist.map(2) lconcat-LCons tta
   by(simp-all add: non-speculative-lappend list-of-lconcat o-def)
  from if-redT-non-speculative-invar[OF step(2)[unfolded tta] - sc1] if-redT-non-speculative-vs-conf[OF
step(2)[unfolded tta], where vs = vs and n=length {|ta|}_o] 2 step.hyps(4)[of w-values P vs {|ta|}_o] sc2
sc1
```

```
  show ?case by(simp add: tta o-def)
qed

lemma init-fin-hext-incr:
  assumes t ⊢ (x, m) −ta→i (x', m')
  and init-fin-lift wfx t x m
  and non-speculative P vs (llist-of ⦃ta⦄o)
  and vs-conf P m vs
  shows m ≤ m'
using assms
by(cases)(auto intro: red-hext-incr)

lemma init-fin-redT-hext-incr:
  assumes mthr.if.redT s (t, ta) s'
  and ts-ok (init-fin-lift wfx) (thr s) (shr s)
  and non-speculative P vs (llist-of ⦃ta⦄o)
  and vs-conf P (shr s) vs
  shows shr s ≤ shr s'
using assms
by(cases)(auto dest: init-fin-hext-incr ts-okD)

lemma init-fin-RedT-hext-incr:
  assumes mthr.if.RedT s ttas s'
  and ts-ok (init-fin-lift wfx) (thr s) (shr s)
  and sc: non-speculative P vs (llist-of (concat (map (λ(t, ta). ⦃ta⦄o) ttas)))
  and vs: vs-conf P (shr s) vs
  shows shr s ≤ shr s'
using assms
proof(induction rule: mthr.if.RedT-induct')
  case refl thus ?case by simp
next
  case (step ttas s' t ta s'')
  note ts-ok = ‹ts-ok (init-fin-lift wfx) (thr s) (shr s)›
  from ‹non-speculative P vs (llist-of (concat (map (λ(t, ta). ⦃ta⦄o) (ttas @ [(t, ta)]))))›
  have ns: non-speculative P vs (llist-of (concat (map (λ(t, ta). ⦃ta⦄o) ttas)))
    and ns': non-speculative P (w-values P vs (concat (map (λ(t, ta). ⦃ta⦄o) ttas))) (llist-of ⦃ta⦄o)
   by(simp-all add: lappend-llist-of-llist-of[symmetric] non-speculative-lappend del: lappend-llist-of-llist-of)
  from ts-ok ns have shr s ≤ shr s'
    using ‹vs-conf P (shr s) vs› by(rule step.IH)
  also have ts-ok (init-fin-lift wfx) (thr s') (shr s')
    using ‹mthr.if.RedT s ttas s'› ts-ok ns ‹vs-conf P (shr s) vs›
    by(rule if-RedT-non-speculative-invar)
  with ‹mthr.if.redT s' (t, ta) s''›
  have ... ≤ shr s'' using ns'
  proof(rule init-fin-redT-hext-incr)
    from ‹mthr.if.RedT s ttas s'› ts-ok ns ‹vs-conf P (shr s) vs›
    show vs-conf P (shr s') (w-values P vs (concat (map (λ(t, ta). ⦃ta⦄o) ttas)))
      by(rule if-RedT-non-speculative-invar)
  qed
  finally show ?case .
qed

lemma init-fin-red-read-typeable:
  assumes t ⊢ (x, m) −ta→i (x', m')
```

and *init-fin-lift wfx t x m NormalAction* (*ReadMem ad al v*) ∈ *set* ⦃*ta*⦄$_o$
shows ∃ *T*. *P*,*m* ⊢ *ad*@*al* : *T*
using *assms*
by *cases*(*auto dest*: *red-read-typeable*)

lemma *Ex-new-action-for*:
assumes *wf*: *wf-syscls P*
and *wfx-start*: *ts-ok wfx* (*thr* (*start-state f P C M vs*)) *start-heap*
and *ka*: *known-addrs start-tid* (*f* (*fst* (*method P C M*)) *M* (*fst* (*snd* (*method P C M*))) (*fst* (*snd* (*snd* (*method P C M*)))) (*the* (*snd* (*snd* (*snd* (*method P C M*))))) *vs*) ⊆ *allocated start-heap*
and *E*: *E* ∈ $\mathcal{E}$-*start f P C M vs status*
and *read*: *ra* ∈ *read-actions E*
and *aloc*: *adal* ∈ *action-loc P E ra*
and *sc*: *non-speculative P* (λ-. {}) (*ltake* (*enat ra*) (*lmap snd E*))
shows ∃ *wa*. *wa* ∈ *new-actions-for P E adal* ∧ *wa* < *ra*
proof −
let *?obs-prefix* = *lift-start-obs start-tid start-heap-obs*
let *?start-state* = *init-fin-lift-state status* (*start-state f P C M vs*)

from *start-state-vs-conf*[*OF wf*]
have *vs-conf-start*: *vs-conf P start-heap* (*w-values P* (λ-. {}) (*map NormalAction start-heap-obs*))
by(*simp add*: *lift-start-obs-def o-def*)

obtain *ad al* **where** *adal*: *adal* = (*ad*, *al*) **by**(*cases adal*)
with *read aloc* **obtain** *v* **where** *ra*: *action-obs E ra* = *NormalAction* (*ReadMem ad al v*)
and *ra-len*: *enat ra* < *llength E*
by(*cases lnth E ra*)(*auto elim*!: *read-actions.cases actionsE*)

from *E* **obtain** *E*′′ **where** *E*: *E* = *lappend* (*llist-of ?obs-prefix*) *E*′′
and *E*′′: *E*′′ ∈ *mthr.if*.$\mathcal{E}$ *?start-state* **by**(*auto*)
from *E*′′ **obtain** *E*′ **where** *E*′: *E*′′ = *lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) ⦃*ta*⦄$_o$)) *E*′)
and *τRuns*: *mthr.if.mthr.Runs ?start-state E*′ **by**(*rule mthr.if*.$\mathcal{E}$.*cases*)

have *ra-len*′: *length ?obs-prefix* ≤ *ra*
proof(*rule ccontr*)
assume ¬ *?thesis*
hence *ra* < *length ?obs-prefix* **by** *simp*
moreover with *ra ra-len E* **obtain** *ra*′ *ad al v*
where *start-heap-obs* ! *ra*′ = *ReadMem ad al v ra*′ < *length start-heap-obs*
by(*cases ra*)(*auto simp add*: *lnth-LCons lnth-lappend1 action-obs-def lift-start-obs-def*)
ultimately have *ReadMem ad al v* ∈ *set start-heap-obs* **unfolding** *in-set-conv-nth* **by** *blast*
thus *False* **by**(*simp add*: *start-heap-obs-not-Read*)
qed
let *?n* = *length ?obs-prefix*
from *ra ra-len ra-len*′ *E* **have** *enat* (*ra* − *?n*) < *llength E*′′
and *ra-obs*: *action-obs E*′′ (*ra* − *?n*) = *NormalAction* (*ReadMem ad al v*)
by(*cases llength E*′′, *auto simp add*: *action-obs-def lnth-lappend2*)

from *τRuns* ‹*enat* (*ra* − *?n*) < *llength E*′′› **obtain** *ra-m ra-n t-ra ta-ra*
where *E-ra*: *lnth E*′′ (*ra* − *?n*) = (*t-ra*, ⦃*ta-ra*⦄$_o$! *ra-n*)
and *ra-n*: *ra-n* < *length* ⦃*ta-ra*⦄$_o$ **and** *ra-m*: *enat ra-m* < *llength E*′
and *ra-conv*: *ra* − *?n* = (∑ *i*<*ra-m*. *length* ⦃*snd* (*lnth E*′ *i*)⦄$_o$) + *ra-n*
and *E*′*-ra-m*: *lnth E*′ *ra-m* = (*t-ra*, *ta-ra*)
unfolding *E*′ **by**(*rule mthr.if.actions-*$\mathcal{E}$*E-aux*)

let *?E′ = ldropn (Suc ra-m) E′*

have *E′-unfold*: *E′ = lappend (ltake (enat ra-m) E′) (LCons (lnth E′ ra-m) ?E′)*
unfolding *ldropn-Suc-conv-ldropn*[*OF ra-m*] **by** *simp*
hence *mthr.if.mthr.Runs ?start-state (lappend (ltake (enat ra-m) E′) (LCons (lnth E′ ra-m) ?E′))*
using *τRuns* **by** *simp*
then obtain *σ′* **where** *σ-σ′*: *mthr.if.mthr.Trsys ?start-state (list-of (ltake (enat ra-m) E′)) σ′*
and *τRuns′*: *mthr.if.mthr.Runs σ′ (LCons (lnth E′ ra-m) ?E′)*
by(*rule mthr.if.mthr.Runs-lappendE*) *simp*
from *τRuns′* **obtain** *σ″* **where** *red-ra*: *mthr.if.redT σ′ (t-ra, ta-ra) σ″*
and *τRuns″*: *mthr.if.mthr.Runs σ″ ?E′*
unfolding *E′-ra-m* **by** *cases*

from *E-ra ra-n ra-obs* **have** *NormalAction (ReadMem ad al v) ∈ set ⦃ta-ra⦄o*
by(*auto simp add*: *action-obs-def in-set-conv-nth*)
with *red-ra* **obtain** *x-ra x′-ra m′-ra*
where *red′-ra*: *mthr.init-fin t-ra (x-ra, shr σ′) ta-ra (x′-ra, m′-ra)*
and *σ″*: *redT-upd σ′ t-ra ta-ra x′-ra m′-ra σ″*
and *ts-t-a*: *thr σ′ t-ra = ⌊(x-ra, no-wait-locks)⌋*
by *cases auto*
from *red′-ra ‹NormalAction (ReadMem ad al v) ∈ set ⦃ta-ra⦄o›*
obtain *ta′-ra X-ra X′-ra*
where *x-ra*: *x-ra = (Running, X-ra)*
and *x′-ra*: *x′-ra = (Running, X′-ra)*
and *ta-ra*: *ta-ra = convert-TA-initial (convert-obs-initial ta′-ra)*
and *red″-ra*: *t-ra ⊢ (X-ra, shr σ′) −ta′-ra→ (X′-ra, m′-ra)*
by *cases fastforce+*

from *‹NormalAction (ReadMem ad al v) ∈ set ⦃ta-ra⦄o› ta-ra*
have *ReadMem ad al v ∈ set ⦃ta′-ra⦄o* **by** *auto*

from *wfx-start* **have** *wfx-start*: *ts-ok (init-fin-lift wfx) (thr ?start-state) (shr ?start-state)*
by(*simp add*: *start-state-def split-beta*)

from *sc ra-len′*
have *non-speculative P (w-values P (λ-. {}) (map snd ?obs-prefix))*
(lmap snd (ltake (enat (ra − ?n)) (lconcat (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) E′))))
unfolding *E E′* **by**(*simp add*: *ltake-lappend2 lmap-lappend-distrib non-speculative-lappend*)
also note *ra-conv* **also note** *plus-enat-simps*(*1*)[*symmetric*]
also have *enat (∑ i<ra-m. length ⦃snd (lnth E′ i)⦄o) = (∑ i<ra-m. enat (length ⦃snd (lnth E′ i)⦄o))*
by(*subst sum-comp-morphism*[*symmetric*])(*simp-all add*: *zero-enat-def*)
also have *. . . = (∑ i<ra-m. llength (lnth (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) E′) i))*
using *ra-m* **by**−(*rule sum.cong*[*OF refl*], *simp add*: *le-less-trans*[**where** *y*=*enat ra-m*] *split-beta*)
also note *ltake-plus-conv-lappend* **also note** *lconcat-ltake*[*symmetric*]
also note *lmap-lappend-distrib*
also note *non-speculative-lappend*
finally have *non-speculative P (w-values P (λ-. {}) (map snd ?obs-prefix)) (lmap snd (lconcat (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) (llist-of (list-of (ltake (enat ra-m) E′))))))*
by(*simp add*: *split-def*)
hence *sc′*: *non-speculative P (w-values P (λ-. {}) (map snd ?obs-prefix)) (llist-of (concat (map (λ(t, ta). ⦃ta⦄o) (list-of (ltake (enat ra-m) E′)))))*
unfolding *lmap-lconcat llist.map-comp o-def lconcat-llist-of*[*symmetric*] *lmap-llist-of*[*symmetric*]

by(*simp add*: *split-beta o-def*)

from *vs-conf-start* **have** *vs-conf-start*: *vs-conf P* (*shr ?start-state*) (*w-values P* (λ-. {}) (*map snd ?obs-prefix*))
by(*simp add*:*init-fin-lift-state-conv-simps start-state-def split-beta lift-start-obs-def o-def*)
with *σ-σ′ wfx-start sc′* **have** *ts-ok* (*init-fin-lift wfx*) (*thr σ′*) (*shr σ′*)
unfolding *mthr.if.RedT-def*[*symmetric*] **by**(*rule if-RedT-non-speculative-invar*)
with *ts-t-a* **have** *wfx t-ra X-ra* (*shr σ′*) **unfolding** *x-ra* **by**(*auto dest*: *ts-okD*)

with *red″-ra* ‹*ReadMem ad al v* ∈ *set* ⦃*ta′-ra*⦄$_o$›
obtain *T′* **where** *type-adal*: *P*,*shr σ′* ⊢ *ad*@*al* : *T′* **by**(*auto dest*: *red-read-typeable*)

from *sc ra-len′* **have** *non-speculative P* (λ-. {}) (*llist-of* (*map snd ?obs-prefix*))
unfolding *E* **by**(*simp add*: *ltake-lappend2 lmap-lappend-distrib non-speculative-lappend*)
with *sc′* **have** *sc″*: *non-speculative P* (λ-. {}) (*llist-of* (*map snd* (*lift-start-obs start-tid start-heap-obs*) @ *concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) (*list-of* (*ltake* (*enat ra-m*) *E′*)))))
by(*simp add*: *lappend-llist-of-llist-of*[*symmetric*] *non-speculative-lappend del*: *lappend-llist-of-llist-of*)

from *σ-σ′ red-ra* ‹*NormalAction* (*ReadMem ad al v*) ∈ *set* ⦃*ta-ra*⦄$_o$› *sc″ ka*
show ∃ *wa*. *wa* ∈ *new-actions-for P E adal* ∧ *wa* < *ra*
unfolding *mthr.if.RedT-def*[*symmetric*]
proof(*cases rule*: *read-ex-NewHeapElem*)
case (*start CTn*)
then obtain *n* **where** *n*: *start-heap-obs* ! *n* = *NewHeapElem ad CTn*
and *len*: *n* < *length start-heap-obs*
unfolding *in-set-conv-nth* **by** *blast*
from *len* **have** *Suc n* ∈ *actions E* **unfolding** *E* **by**(*simp add*: *actions-def enat-less-enat-plusI*)
moreover
from *σ-σ′* **have** *hext*: *start-heap* ⊴ *shr σ′* **unfolding** *mthr.if.RedT-def*[*symmetric*]
using *wfx-start sc′ vs-conf-start*
by(*auto dest*!: *init-fin-RedT-hext-incr simp add*: *start-state-def split-beta init-fin-lift-state-conv-simps*)

from *start* **have** *typeof-addr start-heap ad* = ⌊*CTn*⌋
by(*auto dest*: *NewHeapElem-start-heap-obsD*[*OF wf*])
with *hext* **have** *typeof-addr* (*shr σ′*) *ad* = ⌊*CTn*⌋ **by**(*rule typeof-addr-hext-mono*)
with *type-adal* **have** *adal* ∈ *action-loc P E* (*Suc n*) **using** *n len* **unfolding** *E adal*
by *cases*(*auto simp add*: *action-obs-def lnth-lappend1 lift-start-obs-def*)
moreover have *is-new-action* (*action-obs E* (*Suc n*)) **using** *n len* **unfolding** *E*
by(*simp add*: *action-obs-def lnth-lappend1 lift-start-obs-def*)
ultimately have *Suc n* ∈ *new-actions-for P E adal* **by**(*rule new-actionsI*)
moreover have *Suc n* < *ra* **using** *ra-len′ len* **by**(*simp*)
ultimately show *?thesis* **by** *blast*
next
case (*Red ttas′ s″ t′ ta′ s‴ ttas″ CTn*)

from ‹*NormalAction* (*NewHeapElem ad CTn*) ∈ *set* ⦃*ta′*⦄$_o$›
obtain *obs obs′* **where** *obs*: ⦃*ta′*⦄$_o$ = *obs* @ *NormalAction* (*NewHeapElem ad CTn*) # *obs′*
by(*auto dest*: *split-list*)

let *?wa* = *?n* + *length* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas′*)) + *length obs*
have *enat* (*length* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas′*)) + *length obs*) < *enat* (*length* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) (*ttas′* @ [(*t′*, *ta′*)]))))
using *obs* **by** *simp*
also have ... = *llength* (*lconcat* (*lmap llist-of* (*lmap* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) (*llist-of* (*ttas′* @ [(*t′*, *ta′*)])))))

by(*simp del*: *map-map map-append add*: *lconcat-llist-of*)
also have ... ≤ *llength* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* ⦃*ta*⦄$_o$) (*llist-of* (*ttas'* @ (*t'*, *ta'*) # *ttas''*))))
by(*auto simp add*: *o-def split-def intro*: *lprefix-llist-ofI intro*!: *lprefix-lconcatI lprefix-llength-le*)
also note *len-less* = *calculation*
have ... ≤ (∑ *i*<*ra-m*. *llength* (*lnth* (*lmap* (λ(*t*, *ta*). *llist-of* ⦃*ta*⦄$_o$) *E'*) *i*))
unfolding ‹*list-of* (*ltake* (*enat ra-m*) *E'*) = *ttas'* @ (*t'*, *ta'*) # *ttas''*›[*symmetric*]
by(*simp add*: *ltake-lmap*[*symmetric*] *lconcat-ltake del*: *ltake-lmap*)
also have ... = *enat* (∑ *i*<*ra-m*. *length* ⦃*snd* (*lnth E' i*)⦄$_o$) **using** *ra-m*
by(*subst sum-comp-morphism*[*symmetric*, **where** *h*=*enat*])(*auto intro*: *sum.cong simp add*: *zero-enat-def less-trans*[**where** *y*=*enat ra-m*] *split-beta*)
also have ... ≤ *enat* (*ra* − *?n*) **unfolding** *ra-conv* **by** *simp*
finally have *enat-length*: *enat* (*length* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas'*)) + *length obs*) < *enat* (*ra* − *length* (*lift-start-obs start-tid start-heap-obs*)) **.**
then have *wa-ra*: *?wa* < *ra* **by** *simp*
with *ra-len* **have** *?wa* ∈ *actions E* **by**(*cases llength E*)(*simp-all add*: *actions-def*)
moreover
from ‹*mthr.if.redT s''* (*t'*, *ta'*) *s'''*› ‹*NormalAction* (*NewHeapElem ad CTn*) ∈ *set* ⦃*ta'*⦄$_o$›
obtain *x-wa x-wa'* **where** *ts''t'*: *thr s''* *t'* = ⌊(*x-wa*, *no-wait-locks*)⌋
and *red-wa*: *mthr.init-fin t'* (*x-wa*, *shr s''*) *ta'* (*x-wa'*, *shr s'''*)
by(*cases*) *fastforce*+

from *sc'*
have *ns*: *non-speculative P* (*w-values P* (λ-. {}) (*map snd ?obs-prefix*)) (*llist-of* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas'*)))
and *ns'*: *non-speculative P* (*w-values P* (*w-values P* (λ-. {}) (*map snd ?obs-prefix*)) (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas'*))) (*llist-of* ⦃*ta'*⦄$_o$)
and *ns''*: *non-speculative P* (*w-values P* (*w-values P* (*w-values P* (λ-. {}) (*map snd ?obs-prefix*)) (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas'*))) ⦃*ta'*⦄$_o$) (*llist-of* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas''*)))
unfolding ‹*list-of* (*ltake* (*enat ra-m*) *E'*) = *ttas'* @ (*t'*, *ta'*) # *ttas''*›
by(*simp-all add*: *lappend-llist-of-llist-of*[*symmetric*] *lmap-lappend-distrib non-speculative-lappend del*: *lappend-llist-of-llist-of*)
from ‹*mthr.if.RedT ?start-state ttas' s''*› *wfx-start ns*
have *ts-ok''*: *ts-ok* (*init-fin-lift wfx*) (*thr s''*) (*shr s''*)
using *vs-conf-start* **by**(*rule if-RedT-non-speculative-invar*)
with *ts''t'* **have** *wfxt'*: *wfx t'* (*snd x-wa*) (*shr s''*) **by**(*cases x-wa*)(*auto dest*: *ts-okD*)

{
have *action-obs E ?wa* =
snd (*lnth* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) ⦃*ta*⦄$_o$)) *E'*)) (*length* (*concat* (*map* (λ(*t*, *y*). ⦃*y*⦄$_o$) *ttas'*)) + *length obs*))
unfolding *E E'* **by**(*simp add*: *action-obs-def lnth-lappend2*)
also from *enat-length* ‹*enat* (*ra* − *?n*) < *llength E''*›
have ... = *lnth* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* ⦃*ta*⦄$_o$) *E'*)) (*length* (*concat* (*map* (λ(*t*, *y*). ⦃*y*⦄$_o$) *ttas'*)) + *length obs*)
unfolding *E'*
by(*subst lnth-lmap*[*symmetric*, **where** *f*=*snd*])(*erule* (*1*) *less-trans*, *simp add*: *lmap-lconcat llist.map-comp split-def o-def*)
also from *len-less*
have *enat* (*length* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas'*)) + *length obs*) < *llength* (*lconcat* (*ltake* (*enat ra-m*) (*lmap* (λ(*t*, *ta*). *llist-of* ⦃*ta*⦄$_o$) *E'*)))
unfolding ‹*list-of* (*ltake* (*enat ra-m*) *E'*) = *ttas'* @ (*t'*, *ta'*) # *ttas''*›[*symmetric*]
by(*simp add*: *ltake-lmap*[*symmetric*] *del*: *ltake-lmap*)
note *lnth-lconcat-ltake*[*OF this*, *symmetric*]
also note *ltake-lmap*

```
    also have ltake (enat ra-m) E' = llist-of (list-of (ltake (enat ra-m) E')) by(simp)
    also note ‹list-of (ltake (enat ra-m) E') = ttas' @ (t', ta') # ttas''›
    also note lmap-llist-of also have (λ(t, ta). llist-of ⦃ta⦄o) = llist-of ∘ (λ(t, ta). ⦃ta⦄o)
      by(simp add: o-def split-def)
    also note map-map[symmetric] also note lconcat-llist-of
    also note lnth-llist-of
    also have concat (map (λ(t, ta). ⦃ta⦄o) (ttas' @ (t', ta') # ttas'')) ! (length (concat (map (λ(t,
ta). ⦃ta⦄o) ttas')) + length obs) = NormalAction (NewHeapElem ad CTn)
      by(simp add: nth-append obs)
    finally have action-obs E ?wa = NormalAction (NewHeapElem ad CTn) .
  }
  note wa-obs = this

  from ‹mthr.if.RedT ?start-state ttas' s''› wfx-start ns vs-conf-start
  have vs'': vs-conf P (shr s'') (w-values P (w-values P (λ-. {}) (map snd ?obs-prefix)) (concat (map
(λ(t, ta). ⦃ta⦄o) ttas')))
    by(rule if-RedT-non-speculative-invar)
   from if-redT-non-speculative-vs-conf[OF ‹mthr.if.redT s'' (t', ta') s'''› ts-ok'' - vs'', of length
⦃ta'⦄o] ns'
  have vs''': vs-conf P (shr s''') (w-values P (w-values P (w-values P (λ-. {}) (map snd ?obs-prefix))
(concat (map (λ(t, ta). ⦃ta⦄o) ttas'))) ⦃ta'⦄o)
    by simp

  from ‹mthr.if.redT s'' (t', ta') s'''› ts-ok'' ns' vs''
  have ts-ok (init-fin-lift wfx) (thr s''') (shr s''')
    by(rule if-redT-non-speculative-invar)
  with ‹mthr.if.RedT s''' ttas'' σ'›
  have hext: shr s''' ≤ shr σ' using ns'' vs'''
    by(rule init-fin-RedT-hext-incr)

  from red-wa have typeof-addr (shr s''') ad = ⌊CTn⌋
  using wfxt' ‹NormalAction (NewHeapElem ad CTn) ∈ set ⦃ta'⦄o› by cases(auto dest: red-NewHeapElemD)
  with hext have typeof-addr (shr σ') ad = ⌊CTn⌋ by(rule typeof-addr-hext-mono)
  with type-adal have adal ∈ action-loc P E ?wa using wa-obs unfolding E adal
    by cases (auto simp add: action-obs-def lnth-lappend1 lift-start-obs-def)
  moreover have is-new-action (action-obs E ?wa) using wa-obs by simp
  ultimately have ?wa ∈ new-actions-for P E adal by(rule new-actionsI)
  thus ?thesis using wa-ra by blast
 qed
qed

lemma executions-sc-hb:
 assumes wf-syscls P
 and ts-ok wfx (thr (start-state f P C M vs)) start-heap
 and known-addrs start-tid (f (fst (method P C M)) M (fst (snd (method P C M))) (fst (snd (snd
(method P C M)))) (the (snd (snd (snd (method P C M))))) vs) ⊆ allocated start-heap
 shows
 executions-sc-hb (E-start f P C M vs status) P
 (is executions-sc-hb ?E P)
proof
 fix E a adal a'
 assume E ∈ ?E a ∈ new-actions-for P E adal a' ∈ new-actions-for P E adal
 thus a = a' by(rule E-new-actions-for-unique)
next
```

fix *E ra adal*
assume *E* ∈ *?E ra* ∈ *read-actions E adal* ∈ *action-loc P E ra*
and *non-speculative P* (λ-. {}) (*ltake* (*enat ra*) (*lmap snd E*))
with *assms* **show** ∃ *wa*. *wa* ∈ *new-actions-for P E adal* ∧ *wa* < *ra*
by(*rule Ex-new-action-for*)
qed

lemma *executions-aux*:
assumes *wf*: *wf-syscls P*
and *wfx-start*: *ts-ok wfx* (*thr* (*start-state f P C M vs*)) *start-heap* (**is** *ts-ok wfx* (*thr ?start-state*) -)
and *ka*: *known-addrs start-tid* (*f* (*fst* (*method P C M*)) *M* (*fst* (*snd* (*method P C M*))) (*fst* (*snd* (*snd* (*method P C M*)))) (*the* (*snd* (*snd* (*snd* (*method P C M*))))) *vs*) ⊆ *allocated start-heap*
shows *executions-aux* (*E-start f P C M vs status*) *P*
(**is** *executions-aux ?E P*)
proof
fix *E a adal a′*
assume *E* ∈ *?E a* ∈ *new-actions-for P E adal a′* ∈ *new-actions-for P E adal*
thus *a* = *a′* **by**(*rule E-new-actions-for-unique*)
next
fix *E ws r adal*
assume *E*: *E* ∈ *?E*
and *wf-exec*: *P* ⊢ (*E*, *ws*) √
and *read*: *r* ∈ *read-actions E adal* ∈ *action-loc P E r*
and *sc*: ⋀*a*. ⟦*a* < *r*; *a* ∈ *read-actions E*⟧ ⟹ *P*,*E* ⊢ *a* ↝*mrw ws a*

interpret *jmm*: *executions-sc-hb ?E P*
using *wf wfx-start ka* **by**(*rule executions-sc-hb*)

from *E wf-exec sc*
have *ta-seq-consist P Map.empty* (*ltake* (*enat r*) (*lmap snd E*))
unfolding *ltake-lmap* **by**(*rule jmm.ta-seq-consist-mrwI*) *simp*
hence *non-speculative P* (λ-. {}) (*ltake* (*enat r*) (*lmap snd E*))
by(*rule ta-seq-consist-into-non-speculative*) *simp*
with *wf wfx-start ka E read*
have ∃ *i*. *i* ∈ *new-actions-for P E adal* ∧ *i* < *r*
by(*rule Ex-new-action-for*)
thus ∃ *i*<*r*. *i* ∈ *new-actions-for P E adal* **by** *blast*
qed

lemma *drf*:
assumes *cut-and-update*:
if.cut-and-update
(*init-fin-lift-state status* (*start-state f P C M vs*))
(*mrw-values P Map.empty* (*map snd* (*lift-start-obs start-tid start-heap-obs*)))
(**is** *if.cut-and-update ?start-state* (*mrw-values - -* (*map - ?start-heap-obs*)))
and *wf*: *wf-syscls P*
and *wfx-start*: *ts-ok wfx* (*thr* (*start-state f P C M vs*)) *start-heap*
and *ka*: *known-addrs start-tid* (*f* (*fst* (*method P C M*)) *M* (*fst* (*snd* (*method P C M*))) (*fst* (*snd* (*snd* (*method P C M*)))) (*the* (*snd* (*snd* (*snd* (*method P C M*))))) *vs*) ⊆ *allocated start-heap*
shows *drf* (*E-start f P C M vs status*) *P* (**is** *drf ?E* -)
proof −
interpret *jmm*: *executions-sc-hb ?E P*
using *wf wfx-start ka* **by**(*rule executions-sc-hb*)

let *?n = length ?start-heap-obs*
let *?$\mathcal{E}'$ = lappend (llist-of ?start-heap-obs) ‘ mthr.if.$\mathcal{E}$?start-state*

show *?thesis*
proof
  fix *E ws r*
  assume *E*: *E ∈ ?$\mathcal{E}'$*
    and *wf*: *P ⊢ (E, ws) √*
    and *mrw*: *⋀a. ⟦ a < r; a ∈ read-actions E ⟧ ⟹ P,E ⊢ a ↝mrw ws a*
  show *∃ E'∈?$\mathcal{E}'$. ∃ ws'. P ⊢ (E', ws') √ ∧ ltake (enat r) E = ltake (enat r) E' ∧*
    sequentially-consistent P (E', ws') ∧
    action-tid E r = action-tid E' r ∧ action-obs E r ≈ action-obs E' r ∧
    (r ∈ actions E ⟶ r ∈ actions E')
  proof(*cases ∃ r'. r' ∈ read-actions E ∧ r ≤ r'*)
    case *False*
    have *sequentially-consistent P (E, ws)*
    proof(*rule sequentially-consistentI*)
      fix *a*
      assume *a ∈ read-actions E*
      with *False* **have** *a < r* **by** *auto*
      thus *P,E ⊢ a ↝mrw ws a* **using** *‹a ∈ read-actions E›* **by**(*rule mrw*)
    qed
    moreover have *action-obs E r ≈ action-obs E r* **by**(*rule sim-action-refl*)
    ultimately show *?thesis* **using** *wf E* **by** *blast*
  next
    case *True*
    let *?P = λr'. r' ∈ read-actions E ∧ r ≤ r'*
    let *?r = Least ?P*
    from *True* **obtain** *r'* **where** *r'*: *?P r'* **by** *blast*
    hence *r*: *?P ?r* **by**(*rule LeastI*)
    {
      fix *a*
      assume *a < ?r a ∈ read-actions E*
      have *P,E ⊢ a ↝mrw ws a*
      proof(*cases a < r*)
        case *True*
        thus *?thesis* **using** *‹a ∈ read-actions E›* **by**(*rule mrw*)
      next
        case *False*
        with *‹a ∈ read-actions E›* **have** *?P a* **by** *simp*
        hence *?r ≤ a* **by**(*rule Least-le*)
        with *‹a < ?r›* **have** *False* **by** *simp*
        thus *?thesis* **..**
      qed **}**
    note *mrw' = this*

    from *E* **obtain** *E''* **where** *E*: *E = lappend (llist-of ?start-heap-obs) E''*
      and *E''*: *E'' ∈ mthr.if.$\mathcal{E}$?start-state* **by** *auto*

    from *E''* **obtain** *E'* **where** *E'*: *E'' = lconcat (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄$_o$)) E')*
      and *τRuns*: *mthr.if.mthr.Runs ?start-state E'*
      by(*rule mthr.if.$\mathcal{E}$.cases*)

    have *r-len*: *length ?start-heap-obs ≤ ?r*

proof(*rule ccontr*)
assume ¬ *?thesis*
hence *?r* < *length ?start-heap-obs* **by** *simp*
moreover with *r E* **obtain** *t ad al v* **where** *?start-heap-obs* ! *?r* = (*t*, *NormalAction* (*ReadMem ad al v*))
by(*cases ?start-heap-obs* ! *?r*)(*fastforce elim*!: *read-actions.cases simp add*: *actions-def action-obs-def lnth-lappend1*)
ultimately have (*t*, *NormalAction* (*ReadMem ad al v*)) ∈ *set ?start-heap-obs* **unfolding** *in-set-conv-nth* **by** *blast*
thus *False* **by**(*auto simp add*: *start-heap-obs-not-Read*)
qed
let *?n* = *length ?start-heap-obs*
from *r r-len E* **have** *r*: *?r* − *?n* ∈ *read-actions E''*
by(*fastforce elim*!: *read-actions.cases simp add*: *actions-lappend action-obs-def lnth-lappend2 elim*: *actionsE intro*: *read-actions.intros*)

from *r* **have** *?r* − *?n* ∈ *actions E''* **by**(*auto*)
hence *enat* (*?r* − *?n*) < *llength E''* **by**(*rule actionsE*)
with *τRuns* **obtain** *r-m r-n t-r ta-r*
where *E-r*: *lnth E''* (*?r* − *?n*) = (*t-r*, ⦃*ta-r*⦄$_o$! *r-n*)
and *r-n*: *r-n* < *length* ⦃*ta-r*⦄$_o$ **and** *r-m*: *enat r-m* < *llength E'*
and *r-conv*: *?r* − *?n* = (∑ *i*<*r-m*. *length* ⦃*snd* (*lnth E' i*)⦄$_o$) + *r-n*
and *E'-r-m*: *lnth E' r-m* = (*t-r*, *ta-r*)
unfolding *E'* **by**(*rule mthr.if.actions-*$\mathcal{E}$*E-aux*)

let *?E'* = *ldropn* (*Suc r-m*) *E'*
let *?r-m-E'* = *ltake* (*enat r-m*) *E'*
have *E'-unfold*: *E'* = *lappend* (*ltake* (*enat r-m*) *E'*) (*LCons* (*lnth E' r-m*) *?E'*)
unfolding *ldropn-Suc-conv-ldropn*[*OF r-m*] **by** *simp*
hence *mthr.if.mthr.Runs ?start-state* (*lappend ?r-m-E'* (*LCons* (*lnth E' r-m*) *?E'*))
using *τRuns* **by** *simp*
then obtain *σ'* **where** *σ-σ'*: *mthr.if.mthr.Trsys ?start-state* (*list-of ?r-m-E'*) *σ'*
and *τRuns'*: *mthr.if.mthr.Runs σ'* (*LCons* (*lnth E' r-m*) *?E'*)
by(*rule mthr.if.mthr.Runs-lappendE*) *simp*
from *τRuns'* **obtain** *σ'''* **where** *red-ra*: *mthr.if.redT σ'* (*t-r*, *ta-r*) *σ'''*
and *τRuns''*: *mthr.if.mthr.Runs σ'''* *?E'*
unfolding *E'-r-m* **by** *cases*

let *?vs* = *mrw-values P Map.empty* (*map snd ?start-heap-obs*)
{ fix *a*
assume *enat a* < *enat ?r*
and *a* ∈ *read-actions E*
have *a* < *r*
proof(*rule ccontr*)
assume ¬ *a* < *r*
with ‹*a* ∈ *read-actions E*› **have** *?P a* **by** *simp*
hence *?r* ≤ *a* **by**(*rule Least-le*)
with ‹*enat a* < *enat ?r*› **show** *False* **by** *simp*
qed
hence *P*,*E* ⊢ *a* ↝*mrw ws a* **using** ‹*a* ∈ *read-actions E*› **by**(*rule mrw*) **}**
with ‹*E* ∈ *?*$\mathcal{E}$'› *wf* **have** *ta-seq-consist P Map.empty* (*lmap snd* (*ltake* (*enat ?r*) *E*))
by(*rule jmm.ta-seq-consist-mrwI*)

hence *start-sc*: *ta-seq-consist P Map.empty* (*llist-of* (*map snd ?start-heap-obs*))

and *ta-seq-consist P ?vs* (*lmap snd* (*ltake* (*enat* (*?r* − *?n*)) *E''*))
using ‹*?n* ≤ *?r*› **unfolding** *E ltake-lappend lmap-lappend-distrib*
by(*simp-all add*: *ta-seq-consist-lappend o-def*)

note *this*(*2*) **also from** *r-m*
have *r-m-sum-len-eq*: (∑ *i*<*r-m*. *llength* (*lnth* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) ⦃*ta*⦄$_O$)) *E'*) *i*)) = *enat* (∑ *i*<*r-m*. *length* ⦃*snd* (*lnth E' i*)⦄$_O$)
by(*subst sum-comp-morphism*[*symmetric*, **where** *h*=*enat*])(*auto simp add*: *zero-enat-def split-def less-trans*[**where** *y*=*enat r-m*] *intro*: *sum.cong*)
hence *ltake* (*enat* (*?r* − *?n*)) *E''* =
lappend (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) ⦃*ta*⦄$_O$)) *?r-m-E'*))
(*ltake* (*enat r-n*) (*ldrop* (*enat* (∑ *i*<*r-m*. *length* ⦃*snd* (*lnth E' i*)⦄$_O$)) *E''*))
unfolding *ltake-lmap*[*symmetric*] *lconcat-ltake r-conv plus-enat-simps*(*1*)[*symmetric*] *ltake-plus-conv-lappend*
unfolding *E'* **by** *simp*
finally have *ta-seq-consist P ?vs* (*lmap snd* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) ⦃*ta*⦄$_O$)) *?r-m-E'*)))
and *sc-ta-r*: *ta-seq-consist P* (*mrw-values P ?vs* (*map snd* (*list-of* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) ⦃*ta*⦄$_O$)) *?r-m-E'*))))) (*lmap snd* (*ltake* (*enat r-n*) (*ldropn* (∑ *i*<*r-m*. *length* ⦃*snd* (*lnth E' i*)⦄$_O$) *E''*)))
unfolding *lmap-lappend-distrib* **by**(*simp-all add*: *ta-seq-consist-lappend split-def ldrop-enat*)
note *this*(*1*) **also**
have *lmap snd* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) ⦃*ta*⦄$_O$)) (*ltake* (*enat r-m*) *E'*)))
= *llist-of* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_O$) (*list-of ?r-m-E'*)))
unfolding *lmap-lconcat llist.map-comp o-def split-def lconcat-llist-of*[*symmetric*] *map-map lmap-llist-of*[*symmetric*]
by *simp*
finally have *ta-seq-consist P ?vs* (*llist-of* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_O$) (*list-of ?r-m-E'*)))) .
from *if.sequential-completion*[*OF cut-and-update ta-seq-consist-convert-RA σ-σ'*[*folded mthr.if.RedT-def*] *this red-ra*]
obtain *ta' ttas'*
where *mthr.if.mthr.Runs σ'* (*LCons* (*t-r*, *ta'*) *ttas'*)
and *sc*: *ta-seq-consist P* (*mrw-values P Map.empty* (*map snd ?start-heap-obs*))
(*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* ⦃*ta*⦄$_O$) (*lappend* (*llist-of* (*list-of ?r-m-E'*)) (*LCons* (*t-r*, *ta'*) *ttas'*))))
and *eq-ta*: *eq-upto-seq-inconsist P* ⦃*ta-r*⦄$_O$ ⦃*ta'*⦄$_O$ (*mrw-values P ?vs* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_O$) (*list-of ?r-m-E'*))))
by *blast*

let *?E-sc'* = *lappend* (*llist-of* (*list-of ?r-m-E'*)) (*LCons* (*t-r*, *ta'*) *ttas'*)
let *?E-sc''* = *lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) ⦃*ta*⦄$_O$)) *?E-sc'*)
let *?E-sc* = *lappend* (*llist-of ?start-heap-obs*) *?E-sc''*

from *σ-σ'* ‹*mthr.if.mthr.Runs σ'* (*LCons* (*t-r*, *ta'*) *ttas'*)›
have *mthr.if.mthr.Runs ?start-state ?E-sc'* **by**(*rule mthr.if.mthr.Trsys-into-Runs*)
hence *?E-sc''* ∈ *mthr.if.*$\mathcal{E}$ *?start-state* **by**(*rule mthr.if.*$\mathcal{E}$*.intros*)
hence *?E-sc* ∈ *?*$\mathcal{E}$ **by**(*rule imageI*)
moreover from ‹*?E-sc''* ∈ *mthr.if.*$\mathcal{E}$ *?start-state*›
have *tsa-ok*: *thread-start-actions-ok ?E-sc* **by**(*rule thread-start-actions-ok-init-fin*)

from *sc* **have** *ta-seq-consist P Map.empty* (*lmap snd ?E-sc*)
by(*simp add*: *lmap-lappend-distrib o-def lmap-lconcat llist.map-comp split-def ta-seq-consist-lappend start-sc*)
from *ta-seq-consist-imp-sequentially-consistent*[*OF tsa-ok jmm.*$\mathcal{E}$*-new-actions-for-fun*[*OF* ‹*?E-sc* ∈ *?*$\mathcal{E}$›] *this*]

obtain *ws-sc* **where** *sequentially-consistent P* (*?E-sc*, *ws-sc*)
and $P \vdash$ (*?E-sc*, *ws-sc*) $\surd$ **unfolding** *start-heap-obs-def*[*symmetric*] **by** *iprover*
moreover {
have *enat-sum-r-m-eq*: *enat* ($\sum i$<*r-m*. *length* $\{\!|$*snd* (*lnth E' i*)$|\!\}_o$) = *llength* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) $\{\!|$*ta*$|\!\}_o$)) *?r-m-E'*))
by(*auto intro*: *sum.cong simp add*: *less-trans*[*OF - r-m*] *lnth-ltake llength-lconcat-lfinite-conv-sum sum-comp-morphism*[*symmetric*, **where** *h*=*enat*] *zero-enat-def*[*symmetric*] *split-beta*)
also have ... $\leq$ *llength E''* **unfolding** *E'*
by(*blast intro*: *lprefix-llength-le lprefix-lconcatI lmap-lprefix*)
finally have *r-m-E*: *ltake* (*enat* (*?n* + ($\sum i$<*r-m*. *length* $\{\!|$*snd* (*lnth E' i*)$|\!\}_o$))) *E* = *ltake* (*enat* (*?n* + ($\sum i$<*r-m*. *length* $\{\!|$*snd* (*lnth E' i*)$|\!\}_o$))) *?E-sc*
by(*simp add*: *ltake-lappend lappend-eq-lappend-conv lmap-lappend-distrib r-m-sum-len-eq ltake-lmap*[*symmetric*] *min-def zero-enat-def*[*symmetric*] *E E' lconcat-ltake ltake-all del*: *ltake-lmap*)

have *drop-r-m-E*: *ldropn* (*?n* + ($\sum i$<*r-m*. *length* $\{\!|$*snd* (*lnth E' i*)$|\!\}_o$)) *E* = *lappend* (*llist-of* (*map* (*Pair t-r*) $\{\!|$*ta-r*$|\!\}_o$)) (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) $\{\!|$*ta*$|\!\}_o$)) (*ldropn* (*Suc r-m*) *E'*)))
(**is** - = *?drop-r-m-E*) **using** *E'-r-m* **unfolding** *E E'*
by(*subst* (*2*) *E'-unfold*)(*simp add*: *ldropn-lappend2 lmap-lappend-distrib enat-sum-r-m-eq*[*symmetric*])

have *drop-r-m-E-sc*: *ldropn* (*?n* + ($\sum i$<*r-m*. *length* $\{\!|$*snd* (*lnth E' i*)$|\!\}_o$)) *?E-sc* =
lappend (*llist-of* (*map* (*Pair t-r*) $\{\!|$*ta'*$|\!\}_o$)) (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) $\{\!|$*ta*$|\!\}_o$)) *ttas'*))
by(*simp add*: *ldropn-lappend2 lmap-lappend-distrib enat-sum-r-m-eq*[*symmetric*])

let *?vs-r-m* = *mrw-values P ?vs* (*map snd* (*list-of* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) $\{\!|$*ta*$|\!\}_o$)) *?r-m-E'*))))
note *sc-ta-r* **also**
from *drop-r-m-E* **have** *ldropn* ($\sum i$<*r-m*. *length* $\{\!|$*snd* (*lnth E' i*)$|\!\}_o$) *E''* = *?drop-r-m-E*
unfolding *E* **by**(*simp add*: *ldropn-lappend2*)
also have *lmap snd* (*ltake* (*enat r-n*) ...) = *llist-of* (*take r-n* $\{\!|$*ta-r*$|\!\}_o$) **using** *r-n*
by(*simp add*: *ltake-lappend lmap-lappend-distrib ltake-lmap*[*symmetric*] *take-map o-def zero-enat-def*[*symmetric*] *del*: *ltake-lmap*)
finally have *sc-ta-r*: *ta-seq-consist P ?vs-r-m* (*llist-of* (*take r-n* $\{\!|$*ta-r*$|\!\}_o$)) .
note *eq-ta*
also have $\{\!|$*ta-r*$|\!\}_o$ = *take r-n* $\{\!|$*ta-r*$|\!\}_o$ @ *drop r-n* $\{\!|$*ta-r*$|\!\}_o$ **by** *simp*
finally have *eq-upto-seq-inconsist P* (*take r-n* $\{\!|$*ta-r*$|\!\}_o$ @ *drop r-n* $\{\!|$*ta-r*$|\!\}_o$) $\{\!|$*ta'*$|\!\}_o$ *?vs-r-m*
by(*simp add*: *list-of-lconcat split-def o-def map-concat*)
from *eq-upto-seq-inconsist-appendD*[*OF this sc-ta-r*]
have *r-n'*: *r-n* $\leq$ *length* $\{\!|$*ta'*$|\!\}_o$
and *take-r-n-eq*: *take r-n* $\{\!|$*ta'*$|\!\}_o$ = *take r-n* $\{\!|$*ta-r*$|\!\}_o$
and *eq-r-n*: *eq-upto-seq-inconsist P* (*drop r-n* $\{\!|$*ta-r*$|\!\}_o$) (*drop r-n* $\{\!|$*ta'*$|\!\}_o$) (*mrw-values P ?vs-r-m* (*take r-n* $\{\!|$*ta-r*$|\!\}_o$))
using *r-n* **by**(*simp-all add*: *min-def*)
from *r-conv* ‹*?n* $\leq$ *?r*› **have** *r-conv'*: *?r* = (*?n* + ($\sum i$<*r-m*. *length* $\{\!|$*snd* (*lnth E' i*)$|\!\}_o$)) + *r-n* **by** *simp*
from *r-n' r-n take-r-n-eq r-m-E drop-r-m-E drop-r-m-E-sc*
have *take-r'-eq*: *ltake* (*enat ?r*) *E* = *ltake* (*enat ?r*) *?E-sc* **unfolding** *r-conv'*
apply(*subst* (*1 2*) *plus-enat-simps*(*1*)[*symmetric*])
apply(*subst* (*1 2*) *ltake-plus-conv-lappend*)
apply(*simp add*: *lappend-eq-lappend-conv ltake-lappend1 ldrop-enat take-map*)
done
hence *take-r-eq*: *ltake* (*enat r*) *E* = *ltake* (*enat r*) *?E-sc*
by(*rule ltake-eq-ltake-antimono*)(*simp add*: ‹*?P ?r*›)

```
      from eq-r-n Cons-nth-drop-Suc[OF r-n, symmetric]
      have drop r-n ⦃ta'⦄o ≠ [] by(auto simp add: eq-upto-seq-inconsist-simps)
      hence r-n': r-n < length ⦃ta'⦄o by simp
      hence eq-r-n: ⦃ta-r⦄o ! r-n ≈ ⦃ta'⦄o ! r-n
        using eq-r-n Cons-nth-drop-Suc[OF r-n, symmetric] Cons-nth-drop-Suc[OF r-n', symmetric]
      by(simp add: eq-upto-seq-inconsist-simps split: action.split-asm obs-event.split-asm if-split-asm)
      obtain tid-eq: action-tid E r = action-tid ?E-sc r
        and obs-eq: action-obs E r ≈ action-obs ?E-sc r
      proof(cases r < ?r)
        case True
        { from True have action-tid E r = action-tid (ltake (enat ?r) E) r
            by(simp add: action-tid-def lnth-ltake)
          also note take-r'-eq
          also have action-tid (ltake (enat ?r) ?E-sc) r = action-tid ?E-sc r
            using True by(simp add: action-tid-def lnth-ltake)
          finally have action-tid E r = action-tid ?E-sc r . }
        moreover
        { from True have action-obs E r = action-obs (ltake (enat ?r) E) r
            by(simp add: action-obs-def lnth-ltake)
          also note take-r'-eq
          also have action-obs (ltake (enat ?r) ?E-sc) r = action-obs ?E-sc r
            using True by(simp add: action-obs-def lnth-ltake)
          finally have action-obs E r ≈ action-obs ?E-sc r by simp }
        ultimately show thesis by(rule that)
      next
        case False
        with ‹?P ?r› have r-eq: r = ?r by simp
        hence lnth E r = (t-r, ⦃ta-r⦄o ! r-n) using E-r r-conv' E by(simp add: lnth-lappend2)
        moreover have lnth ?E-sc r = (t-r, ⦃ta'⦄o ! r-n) using ‹?n ≤ ?r› r-n'
       by(subst r-eq)(simp add: r-conv lnth-lappend2 lmap-lappend-distrib enat-sum-r-m-eq[symmetric]
lnth-lappend1 del: length-lift-start-obs)
        ultimately have action-tid E r = action-tid ?E-sc r action-obs E r ≈ action-obs ?E-sc r
          using eq-r-n by(simp-all add: action-tid-def action-obs-def)
        thus thesis by(rule that)
      qed

     have enat r < enat ?n + llength (lconcat (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) (lappend
?r-m-E' (LCons (t-r, ta') LNil))))
        using ‹?P ?r› r-n' unfolding lmap-lappend-distrib
        by(simp add: enat-sum-r-m-eq[symmetric] r-conv')
       also have llength (lconcat (lmap (λ(t, ta). llist-of (map (Pair t) ⦃ta⦄o)) (lappend ?r-m-E'
(LCons (t-r, ta') LNil)))) ≤ llength ?E-sc''
        by(rule lprefix-llength-le[OF lprefix-lconcatI])(simp add: lmap-lprefix)
      finally have r ∈ actions ?E-sc by(simp add: actions-def add-left-mono)
      note this tid-eq obs-eq take-r-eq }
    ultimately show ?thesis by blast
  qed
 qed(rule E-new-actions-for-unique)
qed

lemma sc-legal:
 assumes hb-completion:
  if.hb-completion (init-fin-lift-state status (start-state f P C M vs)) (lift-start-obs start-tid start-heap-obs)
```

```
    (is if.hb-completion ?start-state ?start-heap-obs)
  and wf: wf-syscls P
  and wfx-start: ts-ok wfx (thr (start-state f P C M vs)) start-heap
  and ka: known-addrs start-tid (f (fst (method P C M)) M (fst (snd (method P C M))) (fst (snd
(snd (method P C M)))) (the (snd (snd (snd (method P C M))))) vs) ⊆ allocated start-heap
  shows sc-legal (E-start f P C M vs status) P
  (is sc-legal ?E P)
proof −
  interpret jmm: executions-sc-hb ?E P
    using wf wfx-start ka by(rule executions-sc-hb)

  interpret jmm: executions-aux ?E P
    using wf wfx-start ka by(rule executions-aux)

  show ?thesis
  proof
    fix E ws r
    assume E: E ∈ ?E and wf-exec: P ⊢ (E, ws) √
      and mrw: ⋀a. ⟦a < r; a ∈ read-actions E⟧ ⟹ P,E ⊢ a ⇝mrw ws a


    from E obtain E'' where E: E = lappend (llist-of ?start-heap-obs) E''
      and E'': E'' ∈ mthr.if.E ?start-state by auto

    from E'' obtain E' where E': E'' = lconcat (lmap (λ(t, ta). llist-of (map (Pair t) {|ta|}o)) E')
      and τRuns: mthr.if.mthr.Runs ?start-state E'
      by(rule mthr.if.E.cases)

    show ∃E'∈?E. ∃ws'. P ⊢ (E', ws') √ ∧ ltake (enat r) E = ltake (enat r) E' ∧
                       (∀ a∈read-actions E'. if a < r then ws' a = ws a else P,E' ⊢ ws' a ≤hb a) ∧
                       action-tid E' r = action-tid E r ∧
                        (if r ∈ read-actions E then sim-action else (=)) (action-obs E' r) (action-obs E
r) ∧
                       (r ∈ actions E ⟶ r ∈ actions E')
      (is ∃E'∈?E. ∃ws'. - ∧ ?same E' ∧ ?read E' ws' ∧ ?tid E' ∧ ?obs E' ∧ ?actions E')
    proof(cases r < length ?start-heap-obs)
      case True

      from if.hb-completion-Runs[OF hb-completion ta-hb-consistent-convert-RA]
      obtain ttas where Runs: mthr.if.mthr.Runs ?start-state ttas
       and hb: ta-hb-consistent P ?start-heap-obs (lconcat (lmap (λ(t, ta). llist-of (map (Pair t) {|ta|}o))
ttas))
        by blast

       from Runs have E: lconcat (lmap (λ(t, ta). llist-of (map (Pair t) {|ta|}o)) ttas) ∈ mthr.if.E
?start-state
        by(rule mthr.if.E.intros)

      let ?E = lappend (llist-of ?start-heap-obs) (lconcat (lmap (λ(t, ta). llist-of (map (Pair t) {|ta|}o))
ttas))
      from E have E': ?E ∈ ?E by blast

      from E have tsa: thread-start-actions-ok ?E by(rule thread-start-actions-ok-init-fin)
```

from *start-heap-obs-not-Read*
have *ws*: *is-write-seen P* (*llist-of* (*lift-start-obs start-tid start-heap-obs*)) *ws*
by(*unfold in-set-conv-nth*)(*rule is-write-seenI*, *auto simp add*: *action-obs-def actions-def lift-start-obs-def lnth-LCons elim*!: *read-actions.cases split*: *nat.split-asm*)

with *hb tsa*
have ∃ *ws'*. *P* ⊢ (*?E*, *ws'*) √ ∧
(∀ *n*. *n* ∈ *read-actions ?E* ⟶ *length ?start-heap-obs* ≤ *n* ⟶ *P*,*?E* ⊢ *ws' n* ≤*hb n*) ∧
(∀ *n*<*length ?start-heap-obs*. *ws' n* = *ws n*)
by(*rule ta-hb-consistent-Read-hb*)(*rule jmm.*$\mathcal{E}$*-new-actions-for-fun*[*OF E'*])
then obtain *ws'* **where** *wf-exec'*: *P* ⊢ (*?E*, *ws'*) √
and *read-hb*: ⋀*n*. ⟦ *n* ∈ *read-actions ?E*; *length ?start-heap-obs* ≤ *n* ⟧ ⟹ *P*,*?E* ⊢ *ws' n* ≤*hb n*
and *same*: ⋀*n*. *n*<*length ?start-heap-obs* ⟹ *ws' n* = *ws n* **by** *blast*

from *True* **have** *?same ?E* **unfolding** *E* **by**(*simp add*: *ltake-lappend1*)
moreover {
fix *a*
assume *a*: *a* ∈ *read-actions ?E*
have *if a* < *r then ws' a* = *ws a else P*,*?E* ⊢ *ws' a* ≤*hb a*
proof(*cases a* < *length ?start-heap-obs*)
case *True*
with *a* **have** *False* **using** *start-heap-obs-not-Read*
by *cases*(*auto simp add*: *action-obs-def actions-def lnth-lappend1 lift-start-obs-def lnth-LCons in-set-conv-nth split*: *nat.split-asm*)
thus *?thesis* **..**
next
case *False*
with *read-hb*[*of a*] *True a* **show** *?thesis* **by** *auto*
qed }
hence *?read ?E ws'* **by** *blast*
moreover from *True E* **have** *?tid ?E* **by**(*simp add*: *action-tid-def lnth-lappend1*)
moreover from *True E* **have** *?obs ?E* **by**(*simp add*: *action-obs-def lnth-lappend1*)
moreover from *True* **have** *?actions ?E* **by**(*simp add*: *actions-def enat-less-enat-plusI*)
ultimately show *?thesis* **using** *E' wf-exec'* **by** *blast*
next
case *False*
hence *r*: *length ?start-heap-obs* ≤ *r* **by** *simp*

show *?thesis*
proof(*cases enat r* < *llength E*)
case *False*
then obtain *?same E ?read E ws ?tid E ?obs E ?actions E*
by(*cases llength E*)(*fastforce elim*!: *read-actions.cases simp add*: *actions-def split*: *if-split-asm*)+
with *wf-exec* ‹*E* ∈ *?*$\mathcal{E}$› **show** *?thesis* **by** *blast*
next
case *True*
note *r'* = *this*

let *?r* = *r* − *length ?start-heap-obs*
from *E r r'* **have** *enat ?r* < *llength E''* **by**(*cases llength E''*)(*auto*)
with *τRuns* **obtain** *r-m r-n t-r ta-r*
where *E-r*: *lnth E'' ?r* = (*t-r*, {|*ta-r*|}$_o$! *r-n*)
and *r-n*: *r-n* < *length* {|*ta-r*|}$_o$ **and** *r-m*: *enat r-m* < *llength E'*
and *r-conv*: *?r* = (∑ *i*<*r-m*. *length* {|*snd* (*lnth E' i*)|}$_o$) + *r-n*

and *E′-r-m*: *lnth E′ r-m* = (*t-r*, *ta-r*)
unfolding *E′* **by**(*rule mthr.if.actions-ℰE-aux*)

let *?E′* = *ldropn* (*Suc r-m*) *E′*
let *?r-m-E′* = *ltake* (*enat r-m*) *E′*
have *E′-unfold*: *E′* = *lappend* (*ltake* (*enat r-m*) *E′*) (*LCons* (*lnth E′ r-m*) *?E′*)
unfolding *ldropn-Suc-conv-ldropn*[*OF r-m*] **by** *simp*
hence *mthr.if.mthr.Runs ?start-state* (*lappend ?r-m-E′* (*LCons* (*lnth E′ r-m*) *?E′*))
using *τRuns* **by** *simp*
then obtain *σ′* **where** *σ-σ′*: *mthr.if.mthr.Trsys ?start-state* (*list-of ?r-m-E′*) *σ′*
and *τRuns′*: *mthr.if.mthr.Runs σ′* (*LCons* (*lnth E′ r-m*) *?E′*)
by(*rule mthr.if.mthr.Runs-lappendE*) *simp*
from *τRuns′* **obtain** *σ‴* **where** *red-ra*: *mthr.if.redT σ′* (*t-r*, *ta-r*) *σ‴*
and *τRuns″*: *mthr.if.mthr.Runs σ‴ ?E′*
unfolding *E′-r-m* **by** *cases*

let *?vs* = *mrw-values P Map.empty* (*map snd ?start-heap-obs*)
from ‹*E* ∈ *?ℰ*› *wf-exec* **have** *ta-seq-consist P Map.empty* (*lmap snd* (*ltake* (*enat r*) *E*))
by(*rule jmm.ta-seq-consist-mrwI*)(*simp add*: *mrw*)
hence *ns*: *non-speculative P* (λ-. {}) (*lmap snd* (*ltake* (*enat r*) *E*))
by(*rule ta-seq-consist-into-non-speculative*) *simp*
also note *E* **also note** *ltake-lappend2* **also note** *E′*
also note *E′-unfold* **also note** *lmap-lappend-distrib* **also note** *lmap-lappend-distrib*
also note *lconcat-lappend* **also note** *llist.map*(*2*) **also note** *E′-r-m* **also note** *prod.simps*(*2*)
also note *ltake-lappend2* **also note** *lconcat-LCons* **also note** *ltake-lappend1*
also note *non-speculative-lappend* **also note** *lmap-lappend-distrib* **also note** *non-speculative-lappend*
also have *lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) (*ltake* (*enat r-m*) *E′*)) =
llist-of (*concat* (*map* (λ(*t*, *ta*). *map* (*Pair t*) {|*ta*|}$_o$) (*list-of* (*ltake* (*enat r-m*) *E′*))))
by(*simp add*: *lconcat-llist-of*[*symmetric*] *lmap-llist-of*[*symmetric*] *llist.map-comp o-def split-def del*: *lmap-llist-of*)
ultimately
have *non-speculative P* (λ-. {}) (*lmap snd* (*llist-of ?start-heap-obs*))
and *non-speculative P* (*w-values P* (λ-. {}) (*map snd ?start-heap-obs*))
(*lmap snd* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) (*ltake* (*enat r-m*) *E′*))))
and *ns′*: *non-speculative P* (*w-values P* (*w-values P* (λ-. {}) (*map snd ?start-heap-obs*)) (*map snd* (*concat* (*map* (λ(*t*, *ta*). *map* (*Pair t*) {|*ta*|}$_o$) (*list-of* (*ltake* (*enat r-m*) *E′*))))))
(*lmap snd* (*ltake* (*enat r-n*) (*llist-of* (*map* (*Pair t-r*) {|*ta-r*|}$_o$))))
using *r r-conv r-m r-n*
by(*simp-all add*: *length-concat o-def split-def sum-list-sum-nth length-list-of-conv-the-enat less-min-eq1 atLeast0LessThan lnth-ltake split*: *if-split-asm cong*: *sum.cong-simp*)
hence *ns*: *non-speculative P* (*w-values P* (λ-. {}) (*map snd ?start-heap-obs*))
(*llist-of* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) (*list-of* (*ltake* (*enat r-m*) *E′*)))))
unfolding *lconcat-llist-of*[*symmetric*] *lmap-lconcat lmap-llist-of*[*symmetric*] *llist.map-comp o-def split-def*
by(*simp*)

from *ns′*
have *ns′*: *non-speculative P* (*w-values P* (*w-values P* (λ-. {}) (*map snd ?start-heap-obs*)) (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) (*list-of* (*ltake* (*enat r-m*) *E′*))))) (*llist-of* (*take r-n* {|*ta-r*|}$_o$))
unfolding *map-concat map-map* **by**(*simp add*: *take-map*[*symmetric*] *o-def split-def*)

let *?hb* = λ*ta′-r* :: (*'addr*, *'thread-id*, *status* × *'x*, *'heap*, *'addr*, (*'addr*, *'thread-id*) *obs-event action*) *thread-action*.
ta-hb-consistent P (*?start-heap-obs* @ *concat* (*map* (λ(*t*, *ta*). *map* (*Pair t*) {|*ta*|}$_o$) (*list-of* (*ltake*

(*enat r-m*) *E′*))) @ *map* (*Pair t-r*) (*take r-n* ⦃*ta-r*⦄$_o$)) (*llist-of* (*map* (*Pair t-r*) (*drop r-n* ⦃*ta′-r*⦄$_o$)))
let *?sim* = λ*ta′-r*. (*if* ∃ *ad al v*. ⦃*ta-r*⦄$_o$! *r-n* = *NormalAction* (*ReadMem ad al v*) *then sim-action else* (=)) (⦃*ta-r*⦄$_o$! *r-n*) (⦃*ta′-r*⦄$_o$! *r-n*)

from *red-ra* **obtain** *ta′-r σ′′′′*
where *red-ra′*: *mthr.if.redT σ′* (*t-r*, *ta′-r*) *σ′′′′*
and *eq*: *take r-n* ⦃*ta′-r*⦄$_o$ = *take r-n* ⦃*ta-r*⦄$_o$
and *hb*: *?hb ta′-r*
and *r-n′*: *r-n* < *length* ⦃*ta′-r*⦄$_o$
and *sim*: *?sim ta′-r*
proof(*cases*)
case (*redT-normal x x′ m′*)
note *tst* = ‹*thr σ′ t-r* = ⌊(*x*, *no-wait-locks*)⌋›
and *red* = ‹*t-r* ⊢ (*x*, *shr σ′*) −*ta-r*→*i* (*x′*, *m′*)›
and *aok* = ‹*mthr.if.actions-ok σ′ t-r ta-r*›
and *σ′′′* = ‹*redT-upd σ′ t-r ta-r x′ m′ σ′′′*›
from *if.hb-completionD*[*OF hb-completion σ-σ′*[*folded mthr.if.RedT-def*] *ns tst red aok ns′*] *r-n*
obtain *ta′-r x′′ m′′*
where *red′*: *t-r* ⊢ (*x*, *shr σ′*) −*ta′-r*→*i* (*x′′*, *m′′*)
and *aok′*: *mthr.if.actions-ok σ′ t-r ta′-r*
and *eq′*: *take r-n* ⦃*ta′-r*⦄$_o$ = *take r-n* ⦃*ta-r*⦄$_o$
and *hb*: *?hb ta′-r*
and *r-n′*: *r-n* < *length* ⦃*ta′-r*⦄$_o$
and *sim*: *?sim ta′-r* **by** *blast*
from *redT-updWs-total*[*of t-r wset σ′* ⦃*ta′-r*⦄$_w$]
obtain *σ′′′′* **where** *redT-upd σ′ t-r ta′-r x′′ m′′ σ′′′′* **by** *fastforce*
with *red′ tst aok′* **have** *mthr.if.redT σ′* (*t-r*, *ta′-r*) *σ′′′′* **..**
thus *thesis* **using** *eq′ hb r-n′ sim* **by**(*rule that*)
next
case (*redT-acquire x n ln*)
hence *?hb ta-r* **using** *set-convert-RA-not-Read*[**where** *ln*=*ln*]
by −(*rule ta-hb-consistent-not-ReadI*, *fastforce simp del*: *set-convert-RA-not-Read dest*!: *in-set-dropD*)
with *red-ra r-n* **show** *?thesis* **by**(*auto intro*: *that*)
qed
from *hb*
have *non-speculative P* (*w-values P* (λ-. {}) (*map snd* (*?start-heap-obs* @ *concat* (*map* (λ(*t*, *ta*). *map* (*Pair t*) ⦃*ta*⦄$_o$) (*list-of* (*ltake* (*enat r-m*) *E′*))) @ *map* (*Pair t-r*) (*take r-n* ⦃*ta-r*⦄$_o$)))) (*lmap snd* (*llist-of* (*map* (*Pair t-r*) (*drop r-n* ⦃*ta′-r*⦄$_o$))))
by(*rule ta-hb-consistent-into-non-speculative*)
with *ns′ eq*[*symmetric*] **have** *non-speculative P* (*w-values P* (λ-. {}) (*map snd* (*?start-heap-obs* @ *concat* (*map* (λ(*t*, *ta*). *map* (*Pair t*) ⦃*ta*⦄$_o$) (*list-of* (*ltake* (*enat r-m*) *E′*)))))) (*llist-of* (*map snd* (*map* (*Pair t-r*) ⦃*ta′-r*⦄$_o$)))
by(*subst append-take-drop-id*[**where** *xs*=⦃*ta′-r*⦄$_o$ **and** *n*=*r-n*, *symmetric*])(*simp add*: *o-def map-concat split-def lappend-llist-of-llist-of*[*symmetric*] *non-speculative-lappend del*: *append-take-drop-id lappend-llist-of-llist-of*)
with *ns* **have** *ns′′*: *non-speculative P* (*w-values P* (λ-. {}) (*map snd ?start-heap-obs*)) (*llist-of* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) (*list-of* (*ltake* (*enat r-m*) *E′*) @ [(*t-r*, *ta′-r*)])))
unfolding *lconcat-llist-of*[*symmetric*] *map-append lappend-llist-of-llist-of*[*symmetric*] *lmap-llist-of*[*symmetric*] *llist.map-comp*
by(*simp add*: *o-def split-def non-speculative-lappend list-of-lconcat map-concat*)
from *σ-σ′ red-ra′* **have** *mthr.if.RedT ?start-state* (*list-of ?r-m-E′* @ [(*t-r*, *ta′-r*)]) *σ′′′′*
unfolding *mthr.if.RedT-def* **..**

with *hb-completion*
have *hb-completion'*: *if.hb-completion σ'''' (?start-heap-obs @ concat (map (λ(t, ta). map (Pair t) {|ta|}$_o$) (list-of (ltake (enat r-m) E')) @ [(t-r, ta'-r)]))*
using *ns''* **by**(*rule if.hb-completion-shift*)
from *if.hb-completion-Runs*[*OF hb-completion' ta-hb-consistent-convert-RA*]
obtain *ttas'* **where** *Runs'*: *mthr.if.mthr.Runs σ'''' ttas'*
and *hb'*: *ta-hb-consistent P (?start-heap-obs @ concat (map (λ(t, ta). map (Pair t) {|ta|}$_o$) (list-of (ltake (enat r-m) E') @ [(t-r, ta'-r)])))* (*lconcat* (*lmap* (λ(t, ta). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) *ttas'*))
by *blast*

let *?E* = *lappend* (*llist-of ?start-heap-obs*) (*lconcat* (*lmap* (λ(t, ta). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) (*lappend* (*ltake* (*enat r-m*) *E'*) (*LCons* (*t-r*, *ta'-r*) *ttas'*))))

have *E*: *lconcat* (*lmap* (λ(t, ta). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) (*lappend* (*ltake* (*enat r-m*) *E'*) (*LCons* (*t-r*, *ta'-r*) *ttas'*))) ∈ *mthr.if.E ?start-state*
by(*subst* (*4*) *llist-of-list-of*[*symmetric*])(*simp*, *blast intro*: *mthr.if.E.intros mthr.if.mthr.Trsys-into-Runs σ-σ' mthr.if.mthr.Runs.Step red-ra' Runs'*)
hence *E'*: *?E* ∈ *?E* **by** *blast*

from *E* **have** *tsa*: *thread-start-actions-ok ?E* **by**(*rule thread-start-actions-ok-init-fin*)
also let *?E'* = *lappend* (*llist-of* (*lift-start-obs start-tid start-heap-obs* @ *concat* (*map* (λ(t, ta). *map* (*Pair t*) {|*ta*|}$_o$) (*list-of* (*ltake* (*enat r-m*) *E'*))) @ *map* (*Pair t-r*) (*take r-n* {|*ta-r*|}$_o$))) (*lappend* (*llist-of* (*map* (*Pair t-r*) (*drop r-n* {|*ta'-r*|}$_o$))) (*lconcat* (*lmap* (λ(t, ta). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) *ttas'*)))
have *?E* = *?E'*
using *eq*[*symmetric*]
by(*simp add*: *lmap-lappend-distrib lappend-assoc lappend-llist-of-llist-of*[*symmetric*] *lconcat-llist-of*[*symmetric*] *lmap-llist-of*[*symmetric*] *llist.map-comp o-def split-def del*: *lmap-llist-of*)(*simp add*: *lappend-assoc*[*symmetric*] *lmap-lappend-distrib*[*symmetric*] *map-append*[*symmetric*] *lappend-llist-of-llist-of del*: *map-append*)
finally have *tsa'*: *thread-start-actions-ok ?E'* **.**

from *hb hb' eq*[*symmetric*]
have *HB*: *ta-hb-consistent P (?start-heap-obs @ concat (map (λ(t, ta). map (Pair t) {|ta|}$_o$) (list-of (ltake (enat r-m) E'))) @ map (Pair t-r) (take r-n {|ta-r|}$_o$)) (lappend (llist-of (map (Pair t-r) (drop r-n {|ta'-r|}$_o$))) (lconcat (lmap (λ(t, ta). llist-of (map (Pair t) {|ta|}$_o$)) ttas')))*
by −(*rule ta-hb-consistent-lappendI*, *simp-all add*: *take-map*[*symmetric*] *drop-map*[*symmetric*])

define *EE* **where** *EE* = *llist-of* (*?start-heap-obs* @ *concat* (*map* (λ(t, ta). *map* (*Pair t*) {|*ta*|}$_o$) (*list-of* (*ltake* (*enat r-m*) *E'*))) @ *map* (*Pair t-r*) (*take r-n* {|*ta-r*|}$_o$))

from *r r-conv* **have** *r-conv'*: *r* = (∑*i*<*r-m*. *length* {|*snd* (*lnth E' i*)|}$_o$) + *r-n* + *length ?start-heap-obs* **by** *auto*
hence *len-EE*: *llength EE* = *enat r* **using** *r-m r-n*
by(*auto simp add*: *EE-def length-concat sum-list-sum-nth atLeast0LessThan lnth-ltake less-min-eq1 split-def min-def length-list-of-conv-the-enat cong*: *sum.cong-simp*)

from *r-conv r-m*
have *r-conv3*: *llength* (*lconcat* (*lmap* (λ*x*. *llist-of* (*map* (*Pair* (*fst x*)) {|*snd x*|}$_o$)) (*ltake* (*enat r-m*) *E'*))) = *enat* (*r* − *Suc* (*length start-heap-obs*) − *r-n*)
apply(*simp add*: *llength-lconcat-lfinite-conv-sum lnth-ltake cong*: *sum.cong-simp conj-cong*)
apply(*auto simp add*: *sum-comp-morphism*[**where** *h*=*enat*, *symmetric*] *zero-enat-def less-trans*[**where** *y*=*enat r-m*] *intro*: *sum.cong*)

done

have *is-ws*: *is-write-seen P EE ws*
proof(*rule is-write-seenI*)
fix *a ad al v*
assume *a*: *a* ∈ *read-actions EE*
and *a-obs*: *action-obs EE a* = *NormalAction* (*ReadMem ad al v*)
from *a* **have** *a-r*: *a* < *r* **by** *cases*(*simp add*: *len-EE actions-def*)

from *r E′-r-m r-m r-n r-conv3*
have *eq*: *ltake* (*enat r*) *EE* = *ltake* (*enat r*) *E*
unfolding *E E′ EE-def*
apply(*subst* (*2*) *E′-unfold*)
apply(*simp add*: *ltake-lappend2 lappend-llist-of-llist-of*[*symmetric*] *lappend-eq-lappend-conv lmap-lappend-distrib lconcat-llist-of*[*symmetric*] *o-def split-def lmap-llist-of*[*symmetric*] *del*: *lappend-llist-of-llist-of lmap-llist-of*)
apply(*subst ltake-lappend1*)
defer
apply(*simp add*: *ltake-lmap*[*symmetric*] *take-map*[*symmetric*] *ltake-llist-of*[*symmetric*] *del*: *ltake-lmap ltake-llist-of*)
apply(*auto simp add*: *min-def*)
done
hence *sim*: *ltake* (*enat r*) *EE* [≈] *ltake* (*enat r*) *E* **by**(*rule eq-into-sim-actions*)

from *a sim* **have** *a′*: *a* ∈ *read-actions E*
by(*rule read-actions-change-prefix*)(*simp add*: *a-r*)
from *action-obs-change-prefix-eq*[*OF eq*, *of a*] *a-r a-obs*
have *a-obs′*: *action-obs E a* = *NormalAction* (*ReadMem ad al v*) **by** *simp*

have *a-mrw*: *P*,*E* ⊢ *a* ⇝*mrw ws a* **using** *a-r a′* **by**(*rule mrw*)
with ‹*E* ∈ *?E*› *wf-exec* **have** *ws-a-a*: *ws a* < *a*
by(*rule jmm.mrw-before*)(*auto intro*: *a-r less-trans mrw*)
hence [*simp*]: *ws a* < *r* **using** *a-r* **by** *simp*

from *wf-exec* **have** *ws*: *is-write-seen P E ws* **by**(*rule wf-exec-is-write-seenD*)
from *is-write-seenD*[*OF this a′ a-obs′*]
have *ws a* ∈ *write-actions E*
and (*ad*, *al*) ∈ *action-loc P E* (*ws a*)
and *value-written P E* (*ws a*) (*ad*, *al*) = *v*
and ¬ *P*,*E* ⊢ *a* ≤*hb ws a*
and *is-volatile P al* ⟹ ¬ *P*,*E* ⊢ *a* ≤*so ws a*
and *between*: ⋀*a′*. ⟦ *a′* ∈ *write-actions E*; (*ad*, *al*) ∈ *action-loc P E a′*;
P,*E* ⊢ *ws a* ≤*hb a′* ∧ *P*,*E* ⊢ *a′* ≤*hb a* ∨ *is-volatile P al* ∧ *P*,*E* ⊢ *ws a* ≤*so a′* ∧ *P*,*E* ⊢ *a′* ≤*so a* ⟧
⟹ *a′* = *ws a* **by** *simp-all*

from ‹*ws a* ∈ *write-actions E*› *sim*[*symmetric*]
show *ws a* ∈ *write-actions EE* **by**(*rule write-actions-change-prefix*) *simp*

from *action-loc-change-prefix*[*OF sim*, *of ws a P*] ‹(*ad*, *al*) ∈ *action-loc P E* (*ws a*)›
show (*ad*, *al*) ∈ *action-loc P EE* (*ws a*) **by**(*simp*)

from *value-written-change-prefix*[*OF eq*, *of ws a P*] ‹*value-written P E* (*ws a*) (*ad*, *al*) = *v*›
show *value-written P EE* (*ws a*) (*ad*, *al*) = *v* **by** *simp*

```
      from wf-exec have tsa-E: thread-start-actions-ok E
        by(rule wf-exec-thread-start-actions-okD)

      from ‹¬ P,E ⊢ a ≤hb ws a› show ¬ P,EE ⊢ a ≤hb ws a
      proof(rule contrapos-nn)
        assume P,EE ⊢ a ≤hb ws a
        thus P,E ⊢ a ≤hb ws a using tsa-E sim
          by(rule happens-before-change-prefix)(simp-all add: a-r)
      qed

      { assume is-volatile P al
        hence ¬ P,E ⊢ a ≤so ws a by fact
        thus ¬ P,EE ⊢ a ≤so ws a
          by(rule contrapos-nn)(rule sync-order-change-prefix[OF - sim], simp-all add: a-r) }

      fix a'
      assume a' ∈ write-actions EE (ad, al) ∈ action-loc P EE a'
      moreover
      hence [simp]: a' < r by cases(simp add: actions-def len-EE)
      ultimately have a': a' ∈ write-actions E (ad, al) ∈ action-loc P E a'
        using sim action-loc-change-prefix[OF sim, of a' P]
        by(auto intro: write-actions-change-prefix)
      { assume P,EE ⊢ ws a ≤hb a' P,EE ⊢ a' ≤hb a
        hence P,E ⊢ ws a ≤hb a' P,E ⊢ a' ≤hb a
          using tsa-E sim a-r by(auto elim!: happens-before-change-prefix)
        with between[OF a'] show a' = ws a by simp }
      { assume is-volatile P al  P,EE ⊢ ws a ≤so a' P,EE ⊢ a' ≤so a
        with sim a-r between[OF a'] show a' = ws a
          by(fastforce elim: sync-order-change-prefix intro!: disjI2 del: disjCI) }
    qed

    with HB tsa'
    have ∃ ws'. P ⊢ (?E', ws') √ ∧
            (∀ n. n ∈ read-actions ?E' ⟶ length (?start-heap-obs @ concat (map (λ(t, ta). map
(Pair t) ⦃ta⦄o) (list-of (ltake (enat r-m) E'))) @ map (Pair t-r) (take r-n ⦃ta-r⦄o)) ≤ n ⟶ P,?E'
⊢ ws' n ≤hb n) ∧
            (∀ n<length (lift-start-obs start-tid start-heap-obs @ concat (map (λ(t, ta). map (Pair
t) ⦃ta⦄o) (list-of (ltake (enat r-m) E'))) @ map (Pair t-r) (take r-n ⦃ta-r⦄o)). ws' n = ws n)
      unfolding EE-def
        by(rule ta-hb-consistent-Read-hb)(rule jmm.ℰ-new-actions-for-fun[OF ℰ'[unfolded ‹?E =
?E'›]])
    also have r-conv'': length (?start-heap-obs @ concat (map (λ(t, ta). map (Pair t) ⦃ta⦄o) (list-of
(ltake (enat r-m) E'))) @ map (Pair t-r) (take r-n ⦃ta-r⦄o)) = r
      using r-n r-m unfolding r-conv'
      by(auto simp add: length-concat sum-list-sum-nth atLeast0LessThan lnth-ltake split-def o-def
less-min-eq1 min-def length-list-of-conv-the-enat cong: sum.cong-simp)
    finally obtain ws' where wf-exec': P ⊢ (?E', ws') √
      and read-hb: ⋀n. ⟦ n ∈ read-actions ?E'; r ≤ n ⟧ ⟹ P,?E' ⊢ ws' n ≤hb n
      and read-same: ⋀n. n < r ⟹ ws' n = ws n by blast

    have ?same ?E'
      apply(subst ltake-lappend1, simp add: r-conv''[symmetric] length-list-of-conv-the-enat)
      unfolding E E' lappend-llist-of-llist-of[symmetric]
```

```
          apply(subst (1 2) ltake-lappend2, simp add: r[simplified])
          apply(subst lappend-eq-lappend-conv, simp)
          apply safe
          apply(subst E'-unfold)
          unfolding lmap-lappend-distrib
          apply(subst lconcat-lappend, simp)
          apply(subst lconcat-llist-of[symmetric])
          apply(subst (3) lmap-llist-of[symmetric])
          apply(subst (3) lmap-llist-of[symmetric])
          apply(subst llist.map-comp)
          apply(simp only: split-def o-def)
          apply(subst llist-of-list-of, simp)
          apply(subst (1 2) ltake-lappend2, simp add: r-conv3)
          apply(subst lappend-eq-lappend-conv, simp)
          apply safe
          unfolding llist.map(2) lconcat-LCons E'-r-m snd-conv fst-conv take-map
          apply(subst ltake-lappend1)
           defer
           apply(subst append-take-drop-id[where xs=⦃ta-r⦄o and n=r-n, symmetric])
           unfolding map-append lappend-llist-of-llist-of[symmetric]
           apply(subst ltake-lappend1)
            using r-n
            apply(simp add: min-def r-conv3)
           apply(rule refl)
          apply(simp add: r-conv3)
          using r-n by arith

        moreover {
          fix a
          assume a ∈ read-actions ?E'
          with read-hb[of a] read-same[of a]
          have if a < r then ws' a = ws a else P,?E' ⊢ ws' a ≤hb a by simp }
        hence ?read ?E' ws' by blast
        moreover from r-m r-n r-n'
        have E'-r: lnth ?E' r = (t-r, ⦃ta'-r⦄o ! r-n) unfolding r-conv'
          by(auto simp add: lnth-lappend nth-append length-concat sum-list-sum-nth atLeast0LessThan
split-beta lnth-ltake less-min-eq1 length-list-of-conv-the-enat cong: sum.cong-simp)
        from E-r r have E-r: lnth E r = (t-r, ⦃ta-r⦄o ! r-n)
          unfolding E by(simp add: lnth-lappend)
         have r ∈ read-actions E ⟷ (∃ ad al v. ⦃ta-r⦄o ! r-n = NormalAction (ReadMem ad al v))
using True
        by(auto elim!: read-actions.cases simp add: action-obs-def E-r actions-def intro!: read-actions.intros)
        with sim E'-r E-r have ?tid ?E' ?obs ?E'
          by(auto simp add: action-tid-def action-obs-def)
        moreover have ?actions ?E' using r-n r-m r-n' unfolding r-conv'
          by(cases llength ?E')(auto simp add: actions-def less-min-eq2 length-concat sum-list-sum-nth
atLeast0LessThan split-beta lnth-ltake less-min-eq1 length-list-of-conv-the-enat enat-plus-eq-enat-conv
cong: sum.cong-simp)
        ultimately show ?thesis using wf-exec' 𝓔'
          unfolding ‹?E = ?E'› by blast
      qed
    qed
  qed
qed
```

end

lemma *w-value-mrw-value-conf*:
assumes *set-option* (*vs′ adal*) ⊆ *vs adal* × *UNIV*
shows *set-option* (*mrw-value P vs′ ob adal*) ⊆ *w-value P vs ob adal* × *UNIV*
using *assms* **by**(*cases adal*)(*cases ob rule*: *w-value-cases*, *auto*)

lemma *w-values-mrw-values-conf*:
assumes *set-option* (*vs′ adal*) ⊆ *vs adal* × *UNIV*
shows *set-option* (*mrw-values P vs′ obs adal*) ⊆ *w-values P vs obs adal* × *UNIV*
using *assms*
by(*induct obs arbitrary*: *vs′ vs*)(*auto del*: *subsetI intro*: *w-value-mrw-value-conf*)

lemma *w-value-mrw-value-dom-eq-preserve*:
assumes *dom vs′* = {*adal*. *vs adal* ≠ {}}
shows *dom* (*mrw-value P vs′ ob*) = {*adal*. *w-value P vs ob adal* ≠ {}}
using *assms*
apply(*cases ob rule*: *w-value-cases*)
apply(*simp-all add*: *dom-def split-beta del*: *not-None-eq*)
apply(*blast elim*: *equalityE dest*: *subsetD*)+
done

lemma *w-values-mrw-values-dom-eq-preserve*:
assumes *dom vs′* = {*adal*. *vs adal* ≠ {}}
shows *dom* (*mrw-values P vs′ obs*) = {*adal*. *w-values P vs obs adal* ≠ {}}
using *assms*
by(*induct obs arbitrary*: *vs vs′*)(*auto del*: *equalityI intro*: *w-value-mrw-value-dom-eq-preserve*)

context *jmm-multithreaded* **begin**

definition *non-speculative-read* ::
nat ⇒ (*'l*, *'thread-id*, *'x*, *'m*, *'w*) *state* ⇒ (*'addr* × *addr-loc* ⇒ *'addr val set*) ⇒ *bool*
where
non-speculative-read n s vs ⟷
(∀ *ttas s′ t x ta x′ m′ i ad al v v′*.
s −▷*ttas*→* *s′* ⟶ *non-speculative P vs* (*llist-of* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄*o*) *ttas*))) ⟶
thr s′ t = ⌊(*x*, *no-wait-locks*)⌋ ⟶ *t* ⊢ (*x*, *shr s′*) −*ta*→ (*x′*, *m′*) ⟶ *actions-ok s′ t ta* ⟶
i < *length* ⦃*ta*⦄*o* ⟶
non-speculative P (*w-values P vs* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄*o*) *ttas*))) (*llist-of* (*take i* ⦃*ta*⦄*o*))
⟶
⦃*ta*⦄*o* ! *i* = *NormalAction* (*ReadMem ad al v*) ⟶
v′ ∈ *w-values P vs* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄*o*) *ttas*) @ *take i* ⦃*ta*⦄*o*) (*ad*, *al*) ⟶
(∃ *ta′ x″ m″*. *t* ⊢ (*x*, *shr s′*) −*ta′*→ (*x″*, *m″*) ∧ *actions-ok s′ t ta′* ∧
i < *length* ⦃*ta′*⦄*o* ∧ *take i* ⦃*ta′*⦄*o* = *take i* ⦃*ta*⦄*o* ∧ ⦃*ta′*⦄*o* ! *i* = *NormalAction*
(*ReadMem ad al v′*) ∧
length ⦃*ta′*⦄*o* ≤ *max n* (*length* ⦃*ta*⦄*o*)))

lemma *non-speculative-readI* [*intro?*]:
(⋀*ttas s′ t x ta x′ m′ i ad al v v′*.
⟦ *s* −▷*ttas*→* *s′*; *non-speculative P vs* (*llist-of* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄*o*) *ttas*)));
thr s′ t = ⌊(*x*, *no-wait-locks*)⌋; *t* ⊢ (*x*, *shr s′*) −*ta*→ (*x′*, *m′*); *actions-ok s′ t ta*;
i < *length* ⦃*ta*⦄*o*; *non-speculative P* (*w-values P vs* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄*o*) *ttas*))) (*llist-of*
(*take i* ⦃*ta*⦄*o*));

$\{\!|ta|\!\}_o$! i = *NormalAction* (*ReadMem ad al v*);
$v' \in$ *w-values P vs* (*concat* (*map* (λ(*t*, *ta*). $\{\!|ta|\!\}_o$) *ttas*) @ *take i* $\{\!|ta|\!\}_o$) (*ad*, *al*) ⟧
⟹ $\exists$ *ta' x'' m''*. *t* ⊢ (*x*, *shr s'*) −*ta'*→ (*x''*, *m''*) ∧ *actions-ok s' t ta'* ∧
i < *length* $\{\!|ta'|\!\}_o$ ∧ *take i* $\{\!|ta'|\!\}_o$ = *take i* $\{\!|ta|\!\}_o$ ∧ $\{\!|ta'|\!\}_o$! *i* = *NormalAction* (*ReadMem ad al v'*) ∧
length $\{\!|ta'|\!\}_o$ ≤ *max n* (*length* $\{\!|ta|\!\}_o$))
⟹ *non-speculative-read n s vs*
unfolding *non-speculative-read-def* **by** *blast*

lemma *non-speculative-readD*:
⟦ *non-speculative-read n s vs*; *s* −▷*ttas*→* *s'*; *non-speculative P vs* (*llist-of* (*concat* (*map* (λ(*t*, *ta*). $\{\!|ta|\!\}_o$) *ttas*)));
thr s' t = ⌊(*x*, *no-wait-locks*)⌋; *t* ⊢ (*x*, *shr s'*) −*ta*→ (*x'*, *m'*); *actions-ok s' t ta*;
i < *length* $\{\!|ta|\!\}_o$; *non-speculative P* (*w-values P vs* (*concat* (*map* (λ(*t*, *ta*). $\{\!|ta|\!\}_o$) *ttas*))) (*llist-of* (*take i* $\{\!|ta|\!\}_o$));
$\{\!|ta|\!\}_o$! *i* = *NormalAction* (*ReadMem ad al v*);
$v' \in$ *w-values P vs* (*concat* (*map* (λ(*t*, *ta*). $\{\!|ta|\!\}_o$) *ttas*) @ *take i* $\{\!|ta|\!\}_o$) (*ad*, *al*) ⟧
⟹ $\exists$ *ta' x'' m''*. *t* ⊢ (*x*, *shr s'*) −*ta'*→ (*x''*, *m''*) ∧ *actions-ok s' t ta'* ∧
i < *length* $\{\!|ta'|\!\}_o$ ∧ *take i* $\{\!|ta'|\!\}_o$ = *take i* $\{\!|ta|\!\}_o$ ∧ $\{\!|ta'|\!\}_o$! *i* = *NormalAction* (*ReadMem ad al v'*) ∧
length $\{\!|ta'|\!\}_o$ ≤ *max n* (*length* $\{\!|ta|\!\}_o$)
unfolding *non-speculative-read-def* **by** *blast*

end

8.11.2 *non-speculative* generalises *cut-and-update* and *ta-hb-consistent*

context *known-addrs-typing* **begin**

lemma *read-non-speculative-new-actions-for*:
fixes *status f C M params E*
defines *E* ≡ *lift-start-obs start-tid start-heap-obs*
and *vs* ≡ *w-values P* (λ-. {}) (*map snd E*)
and *s* ≡ *init-fin-lift-state status* (*start-state f P C M params*)
assumes *wf*: *wf-syscls P*
and *RedT*: *mthr.if.RedT s ttas s'*
and *redT*: *mthr.if.redT s'* (*t*, *ta'*) *s''*
and *read*: *NormalAction* (*ReadMem ad al v*) ∈ *set* $\{\!|ta'|\!\}_o$
and *ns*: *non-speculative P* (λ-. {}) (*llist-of* (*map snd E* @ *concat* (*map* (λ(*t*, *ta*). $\{\!|ta|\!\}_o$) *ttas*)))
and *ka*: *known-addrs start-tid* (*f* (*fst* (*method P C M*)) *M* (*fst* (*snd* (*method P C M*))) (*fst* (*snd* (*snd* (*method P C M*)))) (*the* (*snd* (*snd* (*snd* (*method P C M*))))) *params*) ⊆ *allocated start-heap*
and *wt*: *ts-ok* (*init-fin-lift wfx*) (*thr s*) (*shr s*)
and *type-adal*: *P*,*shr s'* ⊢ *ad*@*al* : *T*
shows $\exists$ *w*. *w* ∈ *new-actions-for P* (*llist-of* (*E* @ *concat* (*map* (λ(*t*, *ta*). *map* (*Pair t*) $\{\!|ta|\!\}_o$) *ttas*))) (*ad*, *al*)
(**is** $\exists$ *w*. *?new-w w*)
using *RedT redT read ns*[*unfolded E-def*] *ka* **unfolding** *s-def*
proof(*cases rule*: *read-ex-NewHeapElem*)
case (*start CTn*)
then obtain *n* **where** *n*: *start-heap-obs* ! *n* = *NewHeapElem ad CTn*
and *len*: *n* < *length start-heap-obs*
unfolding *in-set-conv-nth* **by** *blast*
from *ns* **have** *non-speculative P* (*w-values P* (λ-. {}) (*map snd E*)) (*llist-of* (*concat* (*map* (λ(*t*, *ta*). $\{\!|ta|\!\}_o$) *ttas*)))

```
    unfolding lappend-llist-of-llist-of[symmetric]
    by(simp add: non-speculative-lappend del: lappend-llist-of-llist-of)
  with RedT wt have hext: start-heap ⊴ shr s'
    unfolding s-def E-def using start-state-vs-conf[OF wf]
   by(auto dest!: init-fin-RedT-hext-incr simp add: start-state-def split-beta init-fin-lift-state-conv-simps)

  from start have typeof-addr start-heap ad = ⌊CTn⌋
    by(auto dest: NewHeapElem-start-heap-obsD[OF wf])
  with hext have typeof-addr (shr s') ad = ⌊CTn⌋ by(rule typeof-addr-hext-mono)
  with type-adal have (ad, al) ∈ action-loc-aux P (NormalAction (NewHeapElem ad CTn)) using n
len
    by cases (auto simp add: action-obs-def lnth-lappend1 lift-start-obs-def)
  with n len have ?new-w (Suc n)
    by(simp add: new-actions-for-def actions-def E-def action-obs-def lift-start-obs-def nth-append)
  thus ?thesis ..
next
  case (Red ttas' s'' t' ta' s''' ttas'' CTn)
  note ttas = ‹ttas = ttas' @ (t', ta') # ttas''›

  from ‹NormalAction (NewHeapElem ad CTn) ∈ set {|ta'|}_o›
  obtain obs obs' where obs: {|ta'|}_o = obs @ NormalAction (NewHeapElem ad CTn) # obs'
    by(auto dest: split-list)

  let ?n = length (lift-start-obs start-tid start-heap-obs)
  let ?wa = ?n + length (concat (map (λ(t, ta). {|ta|}_o) ttas'')) + length obs

  have ?wa = ?n + length (concat (map (λ(t, ta). map (Pair t) {|ta|}_o) ttas')) + length obs
    by(simp add: length-concat o-def split-def)
  also have ... < length (E @ concat (map (λ(t, ta). map (Pair t) {|ta|}_o) ttas))
    using obs ttas by(simp add: E-def)
  also
  from ttas obs
  have (E @ concat (map (λ(t, ta). map (Pair t) {|ta|}_o) ttas)) ! ?wa = (t', NormalAction (NewHeapElem
ad CTn))
    by(auto simp add: E-def lift-start-obs-def nth-append o-def split-def length-concat)
  moreover
  from ‹mthr.if.redT s'' (t', ta') s'''› ‹NormalAction (NewHeapElem ad CTn) ∈ set {|ta'|}_o›
  obtain x-wa x-wa' where ts''t': thr s'' t' = ⌊(x-wa, no-wait-locks)⌋
    and red-wa: mthr.init-fin t' (x-wa, shr s'') ta' (x-wa', shr s''')
    by(cases) fastforce+

  from start-state-vs-conf[OF wf]
  have vs: vs-conf P (shr s) vs unfolding vs-def E-def s-def
    by(simp add: init-fin-lift-state-conv-simps start-state-def split-def)

  from ns
  have ns: non-speculative P vs (llist-of (concat (map (λ(t, ta). {|ta|}_o) ttas')))
    and ns': non-speculative P (w-values P vs (concat (map (λ(t, ta). {|ta|}_o) ttas'))) (llist-of {|ta'|}_o)
     and ns'': non-speculative P (w-values P (w-values P vs (concat (map (λ(t, ta). {|ta|}_o) ttas')))
{|ta'|}_o) (llist-of (concat (map (λ(t, ta). {|ta|}_o) ttas'')))
    unfolding ttas vs-def
   by(simp-all add: lappend-llist-of-llist-of[symmetric] non-speculative-lappend del: lappend-llist-of-llist-of)
  from ‹mthr.if.RedT (init-fin-lift-state status (start-state f P C M params)) ttas' s''› wt ns
  have ts-ok'': ts-ok (init-fin-lift wfx) (thr s'') (shr s'') using vs unfolding vs-def s-def
```

by(*rule if-RedT-non-speculative-invar*)
with *ts''t'* **have** *wfxt'*: *wfx t'* (*snd x-wa*) (*shr s''*) **by**(*cases x-wa*)(*auto dest*: *ts-okD*)

from ‹*mthr.if.RedT* (*init-fin-lift-state status* (*start-state f P C M params*)) *ttas' s''*› *wt ns*
have *vs''*: *vs-conf P* (*shr s''*) (*w-values P* (*w-values P* (λ-. {}) (*map snd E*)) (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas'*)))
unfolding *s-def E-def vs-def*
by(*rule if-RedT-non-speculative-invar*)(*simp add*: *start-state-def split-beta init-fin-lift-state-conv-simps start-state-vs-conf*[*OF wf*])
from *if-redT-non-speculative-vs-conf*[*OF* ‹*mthr.if.redT s''* (*t'*, *ta'*) *s'''*› *ts-ok''* - *vs''*, *of length* ⦃*ta'*⦄$_o$] *ns'*
have *vs'''*: *vs-conf P* (*shr s'''*) (*w-values P* (*w-values P vs* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas'*))) ⦃*ta'*⦄$_o$)
by(*simp add*: *vs-def*)

from ‹*mthr.if.redT s''* (*t'*, *ta'*) *s'''*› *ts-ok'' ns' vs''*
have *ts-ok* (*init-fin-lift wfx*) (*thr s'''*) (*shr s'''*)
unfolding *vs-def* **by**(*rule if-redT-non-speculative-invar*)
with ‹*mthr.if.RedT s''' ttas'' s'*›
have *hext*: *shr s'''* ⊴ *shr s'* **using** *ns'' vs'''*
by(*rule init-fin-RedT-hext-incr*)

from *red-wa* **have** *typeof-addr* (*shr s'''*) *ad* = ⌊*CTn*⌋
using *wfxt'* ‹*NormalAction* (*NewHeapElem ad CTn*) ∈ *set* ⦃*ta'*⦄$_o$› **by** *cases*(*auto dest*: *red-NewHeapElemD*)
with *hext* **have** *typeof-addr* (*shr s'*) *ad* = ⌊*CTn*⌋ **by**(*rule typeof-addr-hext-mono*)
with *type-adal* **have** (*ad*, *al*) ∈ *action-loc-aux P* (*NormalAction* (*NewHeapElem ad CTn*)) **by** *cases auto*
ultimately have *?new-w ?wa*
by(*simp add*: *new-actions-for-def actions-def action-obs-def*)
thus *?thesis* **..**
qed

lemma *non-speculative-read-into-cut-and-update*:
fixes *status f C M params E*
defines *E* ≡ *lift-start-obs start-tid start-heap-obs*
and *vs* ≡ *w-values P* (λ-. {}) (*map snd E*)
and *s* ≡ *init-fin-lift-state status* (*start-state f P C M params*)
and *vs'* ≡ *mrw-values P Map.empty* (*map snd E*)
assumes *wf*: *wf-syscls P*
and *nsr*: *if.non-speculative-read n s vs*
and *wt*: *ts-ok* (*init-fin-lift wfx*) (*thr s*) (*shr s*)
and *ka*: *known-addrs start-tid* (*f* (*fst* (*method P C M*)) *M* (*fst* (*snd* (*method P C M*))) (*fst* (*snd* (*snd* (*method P C M*)))) (*the* (*snd* (*snd* (*snd* (*method P C M*))))) *params*) ⊆ *allocated start-heap*
shows *if.cut-and-update s vs'*
proof(*rule if.cut-and-updateI*)
fix *ttas s' t x ta x' m'*
assume *Red*: *mthr.if.RedT s ttas s'*
and *sc*: *ta-seq-consist P vs'* (*llist-of* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas*)))
and *tst*: *thr s' t* = ⌊(*x*, *no-wait-locks*)⌋
and *red*: *t* ⊢ (*x*, *shr s'*) −*ta*→*i* (*x'*, *m'*)
and *aok*: *mthr.if.actions-ok s' t ta*
let *?vs* = *w-values P vs* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas*))
let *?vs'* = *mrw-values P vs'* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas*))

```
  from start-state-vs-conf[OF wf]
  have vs: vs-conf P (shr s) vs unfolding vs-def E-def s-def
    by(simp add: init-fin-lift-state-conv-simps start-state-def split-def)

  from sc have ns: non-speculative P vs (llist-of (concat (map (λ(t, ta). {|ta|}_o) ttas)))
   by(rule ta-seq-consist-into-non-speculative)(auto simp add: vs'-def vs-def del: subsetI intro: w-values-mrw-values-conf)

  from ns have ns': non-speculative P (λ-. {}) (llist-of (map snd (lift-start-obs start-tid start-heap-obs)
@ concat (map (λ(t, ta). {|ta|}_o) ttas)))
    unfolding lappend-llist-of-llist-of[symmetric] vs-def
   by(simp add: non-speculative-lappend E-def non-speculative-start-heap-obs del: lappend-llist-of-llist-of)

  have vs-vs'': ⋀adal. set-option (?vs' adal) ⊆ ?vs adal × UNIV
   by(rule w-values-mrw-values-conf)(auto simp add: vs'-def vs-def del: subsetI intro: w-values-mrw-values-conf)
  from Red wt ns vs
  have wt': ts-ok (init-fin-lift wfx) (thr s') (shr s')
    by(rule if-RedT-non-speculative-invar)
  hence wtt: init-fin-lift wfx t x (shr s') using tst by(rule ts-okD)

  { fix i
    have ∃ ta' x'' m''. t ⊢ (x, shr s') −ta'→i (x'', m'') ∧ mthr.if.actions-ok s' t ta' ∧
                        length {|ta'|}_o ≤ max n (length {|ta|}_o) ∧
                        ta-seq-consist P ?vs' (llist-of (take i {|ta'|}_o)) ∧
                        eq-upto-seq-inconsist P (take i {|ta|}_o) (take i {|ta'|}_o) ?vs' ∧
                        (ta-seq-consist P ?vs' (llist-of (take i {|ta|}_o)) ⟶ ta' = ta)
    proof(induct i)
      case 0
      show ?case using red aok
        by(auto simp del: split-paired-Ex simp add: eq-upto-seq-inconsist-simps)
    next
      case (Suc i)
      then obtain ta' x'' m''
        where red': t ⊢ (x, shr s') −ta'→i (x'', m'')
        and aok': mthr.if.actions-ok s' t ta'
        and len: length {|ta'|}_o ≤ max n (length {|ta|}_o)
        and sc': ta-seq-consist P ?vs' (llist-of (take i {|ta'|}_o))
        and eusi: eq-upto-seq-inconsist P (take i {|ta|}_o) (take i {|ta'|}_o) ?vs'
        and ta'-ta: ta-seq-consist P ?vs' (llist-of (take i {|ta|}_o)) ⟹ ta' = ta
        by blast
      let ?vs'' = mrw-values P ?vs' (take i {|ta'|}_o)
      show ?case
       proof(cases i < length {|ta'|}_o ∧ ¬ ta-seq-consist P ?vs' (llist-of (take (Suc i) {|ta'|}_o)) ∧ ¬
ta-seq-consist P ?vs' (llist-of (take (Suc i) {|ta|}_o)))
        case True
        hence i: i < length {|ta'|}_o and ¬ ta-seq-consist P ?vs'' (LCons ({|ta'|}_o ! i) LNil) using sc'
       by(auto simp add: take-Suc-conv-app-nth lappend-llist-of-llist-of[symmetric] ta-seq-consist-lappend
simp del: lappend-llist-of-llist-of)
        then obtain ad al v where ta'-i: {|ta'|}_o ! i = NormalAction (ReadMem ad al v)
          by(auto split: action.split-asm obs-event.split-asm)
        from ta'-i True have read: NormalAction (ReadMem ad al v) ∈ set {|ta'|}_o by(auto simp add:
in-set-conv-nth)
        with red' have ad ∈ known-addrs-if t x by(rule if-red-read-knows-addr)
        hence ad ∈ if.known-addrs-state s' using tst by(rule if.known-addrs-stateI)
        moreover from init-fin-red-read-typeable[OF red' wtt read]
```

obtain T **where** *type-adal*: P,*shr* $s' \vdash ad@al : T$ **..**

from *redT-updWs-total*[*of* t *wset* s' $\{\!|ta'|\!\}_w$] *red'* *tst* *aok'*
obtain s'' **where** *redT'*: *mthr.if.redT* s' (t, ta') s'' **by**(*auto dest*!: *mthr.if.redT.redT-normal*)
with *wf Red*
have $\exists w.\ w \in$ *new-actions-for* P (*llist-of* (E @ *concat* (*map* ($\lambda(t, ta).$ *map* (*Pair* t) $\{\!|ta|\!\}_o$) *ttas*))) (ad, al)
(**is** $\exists w.$ *?new-w* w)
using *read ns' ka wt type-adal* **unfolding** *s-def E-def* **by**(*rule read-non-speculative-new-actions-for*)
then obtain w **where** w: *?new-w* w **..**
have (ad, al) $\in$ *dom ?vs'*
proof(*cases* $w <$ *length* E)
case *True*
with w **have** (ad, al) $\in$ *dom* vs' **unfolding** *vs'-def new-actions-for-def*
by(*clarsimp*)(*erule mrw-values-new-actionD*[*rotated 1*], *auto simp del*: *split-paired-Ex simp add*: *set-conv-nth action-obs-def nth-append intro*!: *exI*[**where** x=w])
also have *dom* $vs' \subseteq$ *dom ?vs'* **by**(*rule mrw-values-dom-mono*)
finally show *?thesis* **.**
next
case *False*
with w **show** *?thesis* **unfolding** *new-actions-for-def*
apply(*clarsimp*)
apply(*erule mrw-values-new-actionD*[*rotated 1*])
apply(*simp-all add*: *set-conv-nth action-obs-def nth-append actions-def*)
apply(*rule exI*[**where** x=w − *length* E])
apply(*subst nth-map*[**where** f=*snd*, *symmetric*])
apply(*simp-all add*: *length-concat o-def split-def map-concat*)
done
qed
hence (ad, al) $\in$ *dom* (*mrw-values* P *?vs'* (*take* i $\{\!|ta'|\!\}_o$))
by(*rule subsetD*[*OF mrw-values-dom-mono*])
then obtain v' b **where** v': *mrw-values* P *?vs'* (*take* i $\{\!|ta'|\!\}_o$) (ad, al) = $\lfloor(v', b)\rfloor$ **by** *auto*
moreover from *vs-vs''*[*of* (ad, al)]
have *set-option* (*mrw-values* P *?vs'* (*take* i $\{\!|ta'|\!\}_o$) (ad, al)) $\subseteq$ *w-values* P *?vs* (*take* i $\{\!|ta'|\!\}_o$) (ad, al) $\times$ *UNIV*
by(*rule w-values-mrw-values-conf*)
ultimately have $v' \in$ *w-values* P *?vs* (*take* i $\{\!|ta'|\!\}_o$) (ad, al) **by** *simp*
moreover from *sc'*
have *non-speculative* P (*w-values* P *vs* (*concat* (*map* ($\lambda(t, ta).$ $\{\!|ta|\!\}_o$) *ttas*))) (*llist-of* (*take* i $\{\!|ta'|\!\}_o$))
by(*blast intro*: *ta-seq-consist-into-non-speculative vs-vs''* *del*: *subsetI*)
ultimately obtain *ta''* *x''* *m''*
where *red''*: $t \vdash (x,$ *shr* $s') -ta''\!\!\rightarrow i\ (x'', m'')$
and *aok''*: *mthr.if.actions-ok* s' t *ta''*
and *i'*: $i <$ *length* $\{\!|ta''|\!\}_o$
and *eq*: *take* i $\{\!|ta''|\!\}_o$ = *take* i $\{\!|ta'|\!\}_o$
and *ta''-i*: $\{\!|ta''|\!\}_o$! i = *NormalAction* (*ReadMem ad al* v')
and *len'*: *length* $\{\!|ta''|\!\}_o \leq$ *max* n (*length* $\{\!|ta'|\!\}_o$)
using *if.non-speculative-readD*[*OF nsr Red ns tst red' aok' i - ta'-i, of v'*] **by** *auto*
from *len' len* **have** *length* $\{\!|ta''|\!\}_o \leq$ *max* n (*length* $\{\!|ta|\!\}_o$) **by** *simp*
moreover have *ta-seq-consist* P *?vs'* (*llist-of* (*take* (*Suc* i) $\{\!|ta''|\!\}_o$))
using *eq sc' i' ta''-i v'*
by(*simp add*: *take-Suc-conv-app-nth lappend-llist-of-llist-of*[*symmetric*] *ta-seq-consist-lappend del*: *lappend-llist-of-llist-of*)

```
      moreover
      have eusi': eq-upto-seq-inconsist P (take (Suc i) ⦃ta⦄o) (take (Suc i) ⦃ta''⦄o) ?vs'
      proof(cases i < length ⦃ta⦄o)
        case True
        with i' i len eq eusi ta'-i ta''-i v' show ?thesis
       by(auto simp add: take-Suc-conv-app-nth ta'-ta eq-upto-seq-inconsist-simps intro: eq-upto-seq-inconsist-appendI)
      next
        case False
        with i ta'-ta have ¬ ta-seq-consist P ?vs' (llist-of (take i ⦃ta⦄o)) by auto
        then show ?thesis using False i' eq eusi
          by(simp add: take-Suc-conv-app-nth eq-upto-seq-inconsist-append2)
      qed
      moreover {
        assume ta-seq-consist P ?vs' (llist-of (take (Suc i) ⦃ta⦄o))
        with True have ta'' = ta by simp }
      ultimately show ?thesis using red'' aok'' True by blast
    next
      case False
      hence ta-seq-consist P ?vs' (llist-of (take (Suc i) ⦃ta⦄o)) ∨
            length ⦃ta'⦄o ≤ i ∨
            ta-seq-consist P ?vs' (llist-of (take (Suc i) ⦃ta'⦄o))
        (is ?case1 ∨ ?case2 ∨ ?case3) by auto
      thus ?thesis
      proof(elim disjCE)
        assume ?case1
        moreover
        hence eq-upto-seq-inconsist P (take (Suc i) ⦃ta⦄o) (take (Suc i) ⦃ta⦄o) ?vs'
          by(rule ta-seq-consist-imp-eq-upto-seq-inconsist-refl)
        ultimately show ?thesis using red aok by fastforce
      next
        assume ?case2 and ¬ ?case1
        have eq-upto-seq-inconsist P (take (Suc i) ⦃ta⦄o) (take (Suc i) ⦃ta'⦄o) ?vs'
        proof(cases i < length ⦃ta⦄o)
          case True
            from ‹?case2› ‹¬ ?case1› have ¬ ta-seq-consist P ?vs' (llist-of (take i ⦃ta⦄o)) by(auto
simp add: ta'-ta)
          hence eq-upto-seq-inconsist P (take i ⦃ta⦄o @ [⦃ta⦄o ! i]) (take i ⦃ta'⦄o @ []) ?vs'
            by(blast intro: eq-upto-seq-inconsist-appendI[OF eusi])
          thus ?thesis using True ‹?case2› by(simp add: take-Suc-conv-app-nth)
        next
          case False with eusi ‹?case2› show ?thesis by simp
        qed
        with red' aok' len sc' eusi ‹?case2› ‹¬ ?case1›show ?thesis
          by (fastforce simp add: take-all simp del: split-paired-Ex)
      next
        assume ?case3 and ¬ ?case1 and ¬ ?case2
        with len eusi ta'-ta
        have eq-upto-seq-inconsist P (take (Suc i) ⦃ta⦄o) (take (Suc i) ⦃ta'⦄o) ?vs'
       by(cases i < length ⦃ta⦄o)(auto simp add: take-Suc-conv-app-nth lappend-llist-of-llist-of[symmetric]
ta-seq-consist-lappend intro: eq-upto-seq-inconsist-appendI eq-upto-seq-inconsist-append2 cong: action.case-cong
obs-event.case-cong)
        with red' aok' ‹?case3› len ‹¬ ?case1› show ?thesis by blast
      qed
    qed
```

```
    qed }
  from this[of max n (length ⦃ta⦄o)]
  show ∃ ta' x'' m''. t ⊢ (x, shr s') −ta'→i (x'', m'') ∧ mthr.if.actions-ok s' t ta' ∧ ta-seq-consist P
?vs' (llist-of ⦃ta'⦄o) ∧ eq-upto-seq-inconsist P ⦃ta⦄o ⦃ta'⦄o ?vs'
    by(auto simp del: split-paired-Ex cong: conj-cong)
qed

lemma non-speculative-read-into-hb-completion:
  fixes status f C M params E
  defines E ≡ lift-start-obs start-tid start-heap-obs
  and vs ≡ w-values P (λ-. {}) (map snd E)
  and s ≡ init-fin-lift-state status (start-state f P C M params)
  assumes wf: wf-syscls P
  and nsr: if.non-speculative-read n s vs
  and wt: ts-ok (init-fin-lift wfx) (thr s) (shr s)
  and ka: known-addrs start-tid (f (fst (method P C M)) M (fst (snd (method P C M))) (fst (snd
(snd (method P C M)))) (the (snd (snd (snd (method P C M))))) params) ⊆ allocated start-heap
  shows if.hb-completion s E
proof
  fix ttas s' t x ta x' m' i
  assume Red: mthr.if.RedT s ttas s'
    and ns: non-speculative P (w-values P (λ-. {}) (map snd E)) (llist-of (concat (map (λ(t, ta).
⦃ta⦄o) ttas)))
    and tst: thr s' t = ⌊(x, no-wait-locks)⌋
    and red: t ⊢ (x, shr s') −ta→i (x', m')
    and aok: mthr.if.actions-ok s' t ta
    and nsi: non-speculative P (w-values P (w-values P (λ-. {}) (map snd E)) (concat (map (λ(t, ta).
⦃ta⦄o) ttas))) (llist-of (take i ⦃ta⦄o))

  let ?E = E @ concat (map (λ(t, ta). map (Pair t) ⦃ta⦄o) ttas) @ map (Pair t) (take i ⦃ta⦄o)
  let ?vs = w-values P vs (concat (map (λ(t, ta). ⦃ta⦄o) ttas))

  from ns have ns': non-speculative P (λ-. {}) (llist-of (map snd (lift-start-obs start-tid start-heap-obs)
@ concat (map (λ(t, ta). ⦃ta⦄o) ttas)))
    unfolding lappend-llist-of-llist-of[symmetric]
   by(simp add: non-speculative-lappend E-def non-speculative-start-heap-obs del: lappend-llist-of-llist-of)

  from start-state-vs-conf[OF wf]
  have vs: vs-conf P (shr s) vs unfolding vs-def E-def s-def
    by(simp add: init-fin-lift-state-conv-simps start-state-def split-def)

  from Red wt ns vs
  have wt': ts-ok (init-fin-lift wfx) (thr s') (shr s')
    unfolding vs-def by(rule if-RedT-non-speculative-invar)
  hence wtt: init-fin-lift wfx t x (shr s') using tst by(rule ts-okD)

  { fix j
    have ∃ ta' x'' m''. t ⊢ (x, shr s') −ta'→i (x'', m'') ∧ mthr.if.actions-ok s' t ta' ∧ length ⦃ta'⦄o ≤
max n (length ⦃ta⦄o) ∧
                        take i ⦃ta'⦄o = take i ⦃ta⦄o ∧
                        ta-hb-consistent P ?E (llist-of (map (Pair t) (take j (drop i ⦃ta'⦄o)))) ∧
                        (i < length ⦃ta⦄o ⟶ i < length ⦃ta'⦄o) ∧
                         (if ∃ ad al v. ⦃ta⦄o ! i = NormalAction (ReadMem ad al v) then sim-action else
(=)) (⦃ta⦄o ! i) (⦃ta'⦄o ! i)
```

proof(*induct j*)
case *0* **from** *red aok* **show** *?case* **by**(*fastforce simp del*: *split-paired-Ex*)
next
case (*Suc j*)
then obtain *ta' x'' m''*
where *red'*: $t \vdash (x, shr\ s') -ta'\rightarrow i\ (x'', m'')$
and *aok'*: *mthr.if.actions-ok s' t ta'*
and *len*: *length* $\{\!|ta'|\!\}_o \leq max\ n$ (*length* $\{\!|ta|\!\}_o$)
and *eq*: *take i* $\{\!|ta'|\!\}_o$ = *take i* $\{\!|ta|\!\}_o$
and *hb*: *ta-hb-consistent P ?E* (*llist-of* (*map* (*Pair t*) (*take j* (*drop i* $\{\!|ta'|\!\}_o$))))
and *len-i*: $i <$ *length* $\{\!|ta|\!\}_o \longrightarrow i <$ *length* $\{\!|ta'|\!\}_o$
and *sim-i*: (*if* $\exists$ *ad al v.* $\{\!|ta|\!\}_o$! *i* = *NormalAction* (*ReadMem ad al v*) *then sim-action else* (=)) ($\{\!|ta|\!\}_o$! *i*) ($\{\!|ta'|\!\}_o$! *i*)
by *blast*
show *?case*
proof(*cases i* + *j* < *length* $\{\!|ta'|\!\}_o$)
case *False*
with *red' aok' len eq hb len-i sim-i* **show** *?thesis* **by**(*fastforce simp del*: *split-paired-Ex*)
next
case *True*
note *j* = *this*
show *?thesis*
proof(*cases* $\exists$ *ad al v.* $\{\!|ta'|\!\}_o$! (*i* + *j*) = *NormalAction* (*ReadMem ad al v*))
case *True*
then obtain *ad al v* **where** *ta'-j*: $\{\!|ta'|\!\}_o$! (*i* + *j*) = *NormalAction* (*ReadMem ad al v*) **by** *blast*
hence *read*: *NormalAction* (*ReadMem ad al v*) ∈ *set* $\{\!|ta'|\!\}_o$ **using** *j* **by**(*auto simp add*: *in-set-conv-nth*)
with *red'* **have** *ad* ∈ *known-addrs-if t x* **by**(*rule if-red-read-knows-addr*)
hence *ad* ∈ *if.known-addrs-state s'* **using** *tst* **by**(*rule if.known-addrs-stateI*)
from *init-fin-red-read-typeable*[*OF red' wtt read*] **obtain** *T*
where *type-adal*: *P*,*shr s'* ⊢ *ad*@*al* : *T* **..**

from *redT-updWs-total*[*of t wset s'* $\{\!|ta'|\!\}_w$] *red' tst aok'*
obtain *s''* **where** *redT'*: *mthr.if.redT s'* (*t*, *ta'*) *s''* **by**(*auto dest*!: *mthr.if.redT.redT-normal*)
with *wf Red*
have $\exists$ *w. w* ∈ *new-actions-for P* (*llist-of* (*E* @ *concat* (*map* (λ(*t*, *ta*). *map* (*Pair t*) $\{\!|ta|\!\}_o$) *ttas*))) (*ad*, *al*)
(**is** $\exists$ *w. ?new-w w*)
using *read ns' ka wt type-adal* **unfolding** *s-def E-def*
by(*rule read-non-speculative-new-actions-for*)
then obtain *w* **where** *w*: *?new-w w* **..**

define *E''* **where** *E''* = *?E* @ *map* (*Pair t*) (*take* (*Suc j*) (*drop i* $\{\!|ta'|\!\}_o$))

from *Red redT'* **have** *mthr.if.RedT s* (*ttas* @ [(*t*, *ta'*)]) *s''* **unfolding** *mthr.if.RedT-def* **..**
hence *tsa*: *thread-start-actions-ok* (*llist-of* (*lift-start-obs start-tid start-heap-obs* @ *concat* (*map* (λ(*t*, *ta*). *map* (*Pair t*) $\{\!|ta|\!\}_o$) (*ttas* @ [(*t*, *ta'*)])))))
unfolding *s-def* **by**(*rule thread-start-actions-ok-init-fin-RedT*)
hence *thread-start-actions-ok* (*llist-of E''*) **unfolding** *E-def*[*symmetric*] *E''-def*
by(*rule thread-start-actions-ok-prefix*)(*rule lprefix-llist-ofI*, *simp*, *metis append-take-drop-id eq map-append*)
moreover from *w* **have** *w* ∈ *actions* (*llist-of E''*)
unfolding *E''-def* **by**(*auto simp add*: *new-actions-for-def actions-def*)

moreover have *length ?E + j ∈ actions (llist-of E'')* **using** *j* **by**(*auto simp add*: *E''-def actions-def*)
moreover from *w* **have** *is-new-action (action-obs (llist-of E'') w)*
by(*auto simp add*: *new-actions-for-def action-obs-def actions-def nth-append E''-def*)
moreover have ¬ *is-new-action (action-obs (llist-of E'') (length ?E + j))*
using *j ta'-j* **by**(*auto simp add*: *action-obs-def nth-append min-def E''-def*)(*subst (asm) nth-map*, *simp-all*)
ultimately have *hb-w*: *P,llist-of E'' ⊢ w ≤hb length ?E + j*
by(*rule happens-before-new-not-new*)

define *writes* **where**
writes = {*w*. *P,llist-of E'' ⊢ w ≤hb length ?E + j ∧ w ∈ write-actions (llist-of E'') ∧*
(ad, al) ∈ action-loc P (llist-of E'') w}

define *w'* **where** *w'* = *Max-torder (action-order (llist-of E'')) writes*

have *writes-actions*: *writes ⊆ actions (llist-of E'')* **unfolding** *writes-def actions-def*
by(*auto dest*!: *happens-before-into-action-order elim*!: *action-orderE simp add*: *actions-def*)
also have *finite ...* **by**(*simp add*: *actions-def*)
finally (*finite-subset*) **have** *finite writes* **.**
moreover from *hb-w w* **have** *w-writes*: *w ∈ writes*
by(*auto 4 3 simp add*: *writes-def new-actions-for-def action-obs-def actions-def nth-append E''-def intro*!: *write-actions.intros elim*!: *is-new-action.cases*)
hence *writes ≠ {}* **by** *auto*

with *torder-action-order ‹finite writes›*
have *w'-writes*: *w' ∈ writes* **using** *writes-actions* **unfolding** *w'-def* **by**(*rule Max-torder-in-set*)
moreover
{ fix *w''*
assume *w'' ∈ writes*
with *torder-action-order ‹finite writes›*
have *llist-of E'' ⊢ w'' ≤a w'* **using** *writes-actions* **unfolding** *w'-def* **by**(*rule Max-torder-above*)
}
note *w'-maximal* = *this*

define *v'* **where** *v'* = *value-written P (llist-of E'') w' (ad, al)*

from *nsi ta-hb-consistent-into-non-speculative*[*OF hb*]
have *nsi'*: *non-speculative P (w-values P vs (concat (map (λ(t, ta). {|ta|}_o) ttas))) (llist-of (take (i + j) {|ta'|}_o))*
unfolding *take-add lappend-llist-of-llist-of*[*symmetric*] *non-speculative-lappend vs-def eq*
by(*simp add*: *non-speculative-lappend o-def map-concat split-def del*: *lappend-llist-of-llist-of*)

from *w'-writes* **have** *adal-w'*: *(ad, al) ∈ action-loc P (llist-of E'') w'* **by**(*simp add*: *writes-def*)
from *w'-writes* **have** *w' ∈ write-actions (llist-of E'')*
unfolding *writes-def* **by** *blast*
then obtain *is-write-action (action-obs (llist-of E'') w')*
and *w'-actions*: *w' ∈ actions (llist-of E'')* **by** *cases*
hence *v' ∈ w-values P (λ-. {}) (map snd E'') (ad, al)*
proof *cases*
case (*NewHeapElem ad' CTn*)
hence *NormalAction (NewHeapElem ad' CTn) ∈ set (map snd E'')*
using *w'-actions* **unfolding** *in-set-conv-nth*
by(*auto simp add*: *actions-def action-obs-def cong*: *conj-cong*)

```
          moreover have ad' = ad
            and (ad, al) ∈ action-loc-aux P (NormalAction (NewHeapElem ad CTn))
            using adal-w' NewHeapElem by auto
          ultimately show ?thesis using NewHeapElem unfolding v'-def
            by(simp add: value-written.simps w-values-new-actionD)
        next
          case (WriteMem ad' al' v'')
          hence NormalAction (WriteMem ad' al' v'') ∈ set (map snd E'')
            using w'-actions unfolding in-set-conv-nth
            by(auto simp add: actions-def action-obs-def cong: conj-cong)
          moreover have ad' = ad al' = al using adal-w' WriteMem by auto
          ultimately show ?thesis using WriteMem unfolding v'-def
            by(simp add: value-written.simps w-values-WriteMemD)
        qed
        hence v' ∈ w-values P vs (concat (map (λ(t, ta). ⦃ta⦄o) ttas) @ take (i + j) ⦃ta'⦄o) (ad, al)
          using j ta'-j eq unfolding E''-def vs-def
          by(simp add: o-def split-def map-concat take-add take-Suc-conv-app-nth)
        from if.non-speculative-readD[OF nsr Red ns[folded vs-def] tst red' aok' j nsi' ta'-j this]
        obtain ta'' x'' m''
          where red'': t ⊢ (x, shr s') −ta''→i (x'', m'')
          and aok'': mthr.if.actions-ok s' t ta''
          and j': i + j < length ⦃ta''⦄o
          and eq': take (i + j) ⦃ta''⦄o = take (i + j) ⦃ta'⦄o
          and ta''-j: ⦃ta''⦄o ! (i + j) = NormalAction (ReadMem ad al v')
          and len': length ⦃ta''⦄o ≤ max n (length ⦃ta'⦄o) by blast

        define EE where EE = ?E @ map (Pair t) (take j (drop i ⦃ta''⦄o))
          define E' where E' = ?E @ map (Pair t) (take j (drop i ⦃ta''⦄o)) @ [(t, NormalAction
(ReadMem ad al v'))]

        from len' len have length ⦃ta''⦄o ≤ max n (length ⦃ta⦄o) by simp
        moreover from eq' eq j j' have take i ⦃ta''⦄o = take i ⦃ta⦄o
          by(auto simp add: take-add min-def)
        moreover {
          note hb
          also have eq'': take j (drop i ⦃ta'⦄o) = take j (drop i ⦃ta''⦄o)
            using eq' j j' by(simp add: take-add min-def)
         also have ta-hb-consistent P (?E @ list-of (llist-of (map (Pair t) (take j (drop i ⦃ta''⦄o)))))
(llist-of [(t, ⦃ta''⦄o ! (i + j))])
            unfolding llist-of.simps ta-hb-consistent-LCons ta-hb-consistent-LNil ta''-j prod.simps
action.simps obs-event.simps list-of-llist-of append-assoc E'-def[symmetric, unfolded append-assoc]
            unfolding EE-def[symmetric, unfolded append-assoc]
          proof(intro conjI TrueI exI[where x=w'] strip)
            have llist-of E'' [≈] llist-of E' using j len eq'' ta'-j unfolding E''-def E'-def
          by(auto simp add: sim-actions-def list-all2-append List.list-all2-refl split-beta take-Suc-conv-app-nth
take-map[symmetric])
            moreover have length E'' = length E' using j j' by(simp add: E''-def E'-def)
             ultimately have sim: ltake (enat (length E')) (llist-of E'') [≈] ltake (enat (length E'))
(llist-of E') by simp

            from w'-actions ‹length E'' = length E'›
            have w'-len: w' < length E' by(simp add: actions-def)

            from ‹w' ∈ write-actions (llist-of E'')› sim
```

```
          show w' ∈ write-actions (llist-of E') by(rule write-actions-change-prefix)(simp add: w'-len)
           from adal-w' action-loc-change-prefix[OF sim, of w' P]
           show (ad, al) ∈ action-loc P (llist-of E') w' by(simp add: w'-len)

           from ta'-j j have length ?E + j ∈ read-actions (llist-of E'')
             by(auto intro!: read-actions.intros simp add: action-obs-def actions-def E''-def min-def
nth-append)(auto)
           hence w' ≠ length ?E + j using ‹w' ∈ write-actions (llist-of E'')›
            by(auto dest: read-actions-not-write-actions)
           with w'-len have w' < length ?E + j by(simp add: E'-def)
           from j j' len' eq''
          have ltake (enat (length ?E + j)) (llist-of E'') = ltake (enat (length ?E + j)) (llist-of E')
            by(auto simp add: E''-def E'-def min-def take-Suc-conv-app-nth)
           from value-written-change-prefix[OF this, of w' P] ‹w' < length ?E + j›
           show value-written P (llist-of E') w' (ad, al) = v' unfolding v'-def by simp

           from ‹thread-start-actions-ok (llist-of E'')› ‹llist-of E'' [≈] llist-of E'›
           have tsa'': thread-start-actions-ok (llist-of E')
            by(rule thread-start-actions-ok-change)

           from w'-writes j j' len len' have P,llist-of E'' ⊢ w' ≤hb length EE
            by(auto simp add: EE-def writes-def min-def ac-simps)
           thus P,llist-of E' ⊢ w' ≤hb length EE using tsa'' sim
            by(rule happens-before-change-prefix)(simp add: w'-len, simp add: EE-def E'-def)

           fix w''
           assume w'': w'' ∈ write-actions (llist-of E')
            and adal-w'': (ad, al) ∈ action-loc P (llist-of E') w''

           from w'' have w''-len: w'' < length E' by(cases)(simp add: actions-def)

           from w'' sim[symmetric] have w'': w'' ∈ write-actions (llist-of E'')
            by(rule write-actions-change-prefix)(simp add: w''-len)
           from adal-w'' action-loc-change-prefix[OF sim[symmetric], of w'' P] w''-len
           have adal-w'': (ad, al) ∈ action-loc P (llist-of E'') w'' by simp
           {
            presume w'-w'': llist-of E' ⊢ w' ≤a w''
              and w''-hb: P,llist-of E' ⊢ w'' ≤hb length EE
            from w''-hb ‹thread-start-actions-ok (llist-of E'')› sim[symmetric]
            have P,llist-of E'' ⊢ w'' ≤hb length EE
              by(rule happens-before-change-prefix)(simp add: w''-len, simp add: E'-def EE-def)
            with w'' adal-w'' j j' len len' have w'' ∈ writes
              by(auto simp add: writes-def EE-def min-def ac-simps split: if-split-asm)
            hence llist-of E'' ⊢ w'' ≤a w' by(rule w'-maximal)
            hence llist-of E' ⊢ w'' ≤a w' using sim
              by(rule action-order-change-prefix)(simp-all add: w'-len w''-len)
            thus w'' = w' w'' = w' using w'-w'' by(rule antisymPD[OF antisym-action-order])+
           }

           { assume P,llist-of E' ⊢ w' ≤hb w'' ∧ P,llist-of E' ⊢ w'' ≤hb length EE
            thus llist-of E' ⊢ w' ≤a w'' P,llist-of E' ⊢ w'' ≤hb length EE
              by(blast dest: happens-before-into-action-order)+ }
           { assume is-volatile P al ∧ P,llist-of E' ⊢ w' ≤so w'' ∧ P,llist-of E' ⊢ w'' ≤so length EE
            then obtain vol: is-volatile P al
```

```
            and so: P,llist-of E' ⊢ w' ≤so w''
            and so': P,llist-of E' ⊢ w'' ≤so length EE by blast
          from so show llist-of E' ⊢ w' ≤a w'' by(blast elim: sync-orderE)

          show P,llist-of E' ⊢ w'' ≤hb length EE
          proof(cases is-new-action (action-obs (llist-of E') w''))
            case True
            with ‹w'' ∈ write-actions (llist-of E')› ta''-j show ?thesis
             by cases(rule happens-before-new-not-new[OF tsa''], auto simp add: actions-def EE-def
E'-def action-obs-def min-def nth-append)
          next
            case False
            with ‹w'' ∈ write-actions (llist-of E')› ‹(ad, al) ∈ action-loc P (llist-of E') w''›
            obtain v'' where action-obs (llist-of E') w'' = NormalAction (WriteMem ad al v'')
              by cases(auto elim: is-write-action.cases)
            with ta''-j w'' j j' len len'
              have P ⊢ (action-tid (llist-of E') w'', action-obs (llist-of E') w'') ↝sw (action-tid
(llist-of E') (length EE), action-obs (llist-of E') (length EE))
              by(auto simp add: E'-def EE-def action-obs-def min-def nth-append Volatile)
            with so' have P,llist-of E' ⊢ w'' ≤sw length EE by(rule sync-withI)
            thus ?thesis unfolding po-sw-def [abs-def] by(blast intro: tranclp.r-into-trancl)
          qed }
        qed
          ultimately have ta-hb-consistent P ?E (lappend (llist-of (map (Pair t) (take j (drop i
{|ta''|}_o)))) (llist-of ([(t, {|ta''|}_o ! (i + j))])))
          by(rule ta-hb-consistent-lappendI) simp
        hence ta-hb-consistent P ?E (llist-of (map (Pair t) (take (Suc j) (drop i {|ta''|}_o))))
          using j' unfolding lappend-llist-of-llist-of by(simp add: take-Suc-conv-app-nth) }
      moreover from len-i have i < length {|ta|}_o ⟶ i < length {|ta''|}_o using eq' j' by auto
      moreover from sim-i eq' ta''-j ta'-j
       have (if ∃ad al v. {|ta|}_o ! i = NormalAction (ReadMem ad al v) then sim-action else (=))
({|ta|}_o ! i) ({|ta''|}_o ! i)
        by(cases j = 0)(auto split: if-split-asm, (metis add-strict-left-mono add-0-right nth-take)+)
      ultimately show ?thesis using red'' aok'' by blast
    next
      case False
        hence ta-hb-consistent P (?E @ list-of (llist-of (map (Pair t) (take j (drop i {|ta'|}_o)))))
(llist-of [(t, {|ta'|}_o ! (i + j))])
        by(simp add: ta-hb-consistent-LCons split: action.split obs-event.split)
      with hb
      have ta-hb-consistent P ?E (lappend (llist-of (map (Pair t) (take j (drop i {|ta'|}_o)))) (llist-of
([(t, {|ta'|}_o ! (i + j))])))
        by(rule ta-hb-consistent-lappendI) simp
      hence ta-hb-consistent P ?E (llist-of (map (Pair t) (take (Suc j) (drop i {|ta'|}_o))))
        using j unfolding lappend-llist-of-llist-of by(simp add: take-Suc-conv-app-nth)
      with red' aok' len eq len-i sim-i show ?thesis by blast
    qed
  qed
 qed }
from this[of max n (length {|ta|}_o)]
show ∃ta' x'' m''. t ⊢ (x, shr s') −ta'→i (x'', m'') ∧ mthr.if.actions-ok s' t ta' ∧
                   take i {|ta'|}_o = take i {|ta|}_o ∧
                   ta-hb-consistent P ?E (llist-of (map (Pair t) (drop i {|ta'|}_o))) ∧
                   (i < length {|ta|}_o ⟶ i < length {|ta'|}_o) ∧
```

(*if* ∃ *ad al v*. ⦃*ta*⦄$_o$! *i* = *NormalAction* (*ReadMem ad al v*) *then sim-action else* (=)) (⦃*ta*⦄$_o$! *i*) (⦃*ta'*⦄$_o$! *i*)
by(*simp del*: *split-paired-Ex cong*: *conj-cong split del*: *if-split*) *blast*
qed

end

end

8.12 Type-safety proof for the Java memory model

theory *JMM-Typesafe*
imports
JMM-Framework
begin

Create a dynamic list *heap-independent* of theorems for replacing heap-dependent constants by heap-independent ones.

ML ‹
structure Heap-Independent-Rules = *Named-Thms*
(
val name = @{*binding heap-independent*}
val description = *Simplification rules for heap−independent constants*
)
›
setup ‹*Heap-Independent-Rules.setup*›

locale *heap-base'* =
h: *heap-base*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate λ-. *typeof-addr heap-read heap-write*
for *addr2thread-id* :: ('*addr* :: *addr*) ⇒ '*thread-id*
and *thread-id2addr* :: '*thread-id* ⇒ '*addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: '*heap*
and *allocate* :: '*heap* ⇒ *htype* ⇒ ('*heap* × '*addr*) *set*
and *typeof-addr* :: '*addr* ⇀ *htype*
and *heap-read* :: '*heap* ⇒ '*addr* ⇒ *addr-loc* ⇒ '*addr val* ⇒ *bool*
and *heap-write* :: '*heap* ⇒ '*addr* ⇒ *addr-loc* ⇒ '*addr val* ⇒ '*heap* ⇒ *bool*
begin

definition *typeof-h* :: '*addr val* ⇒ *ty option*
where *typeof-h* = *h.typeof-h undefined*
lemma *typeof-h-conv-typeof-h* [*heap-independent*, *iff*]: *h.typeof-h h* = *typeof-h*
by(*rule ext*)(*case-tac x*, *simp-all add*: *typeof-h-def*)
lemmas *typeof-h-simps* [*simp*] = *h.typeof-h.simps* [*unfolded heap-independent*]

definition *cname-of* :: '*addr* ⇒ *cname*
where *cname-of* = *h.cname-of undefined*
lemma *cname-of-conv-cname-of* [*heap-independent*, *iff*]: *h.cname-of h* = *cname-of*
by(*simp add*: *cname-of-def h.cname-of-def*[*abs-def*])

definition *addr-loc-type* :: *'m prog* ⇒ *'addr* ⇒ *addr-loc* ⇒ *ty* ⇒ *bool*
where *addr-loc-type P* = *h.addr-loc-type P undefined*
notation *addr-loc-type* (‹- ⊢ -@- : -› [*50*, *50*, *50*, *50*] *51*)
lemma *addr-loc-type-conv-addr-loc-type* [*heap-independent*, *iff*]:
h.addr-loc-type P h = *addr-loc-type P*
by(*simp add*: *addr-loc-type-def h.addr-loc-type-def*)
lemmas *addr-loc-type-cases* [*cases pred*: *addr-loc-type*] =
h.addr-loc-type.cases[*unfolded heap-independent*]
lemmas *addr-loc-type-intros* = *h.addr-loc-type.intros*[*unfolded heap-independent*]

definition *typeof-addr-loc* :: *'m prog* ⇒ *'addr* ⇒ *addr-loc* ⇒ *ty*
where *typeof-addr-loc P* = *h.typeof-addr-loc P undefined*
lemma *typeof-addr-loc-conv-typeof-addr-loc* [*heap-independent*, *iff*]:
h.typeof-addr-loc P h = *typeof-addr-loc P*
by(*simp add*: *typeof-addr-loc-def h.typeof-addr-loc-def*[*abs-def*])

definition *conf* :: *'a prog* ⇒ *'addr val* ⇒ *ty* ⇒ *bool*
where *conf P* ≡ *h.conf P undefined*
notation *conf* (‹- ⊢ - :≤ -› [*51*,*51*,*51*] *50*)
lemma *conf-conv-conf* [*heap-independent*, *iff*]: *h.conf P h* = *conf P*
by(*simp add*: *conf-def heap-base.conf-def*[*abs-def*])
lemmas *defval-conf* [*simp*] = *h.defval-conf*[*unfolded heap-independent*]

definition *lconf* :: *'m prog* ⇒ (*vname* ⇀ *'addr val*) ⇒ (*vname* ⇀ *ty*) ⇒ *bool*
where *lconf P* = *h.lconf P undefined*
notation *lconf* (‹- ⊢ - *'*(:≤*'*) -› [*51*,*51*,*51*] *50*)
lemma *lconf-conv-lconf* [*heap-independent*, *iff*]: *h.lconf P h* = *lconf P*
by(*simp add*: *lconf-def h.lconf-def*[*abs-def*])

definition *confs* :: *'m prog* ⇒ *'addr val list* ⇒ *ty list* ⇒ *bool*
where *confs P* = *h.confs P undefined*
notation *confs* (‹- ⊢ - [:≤] -› [*51*,*51*,*51*] *50*)
lemma *confs-conv-confs* [*heap-independent*, *iff*]: *h.confs P h* = *confs P*
by(*simp add*: *confs-def*)

definition *tconf* :: *'m prog* ⇒ *'thread-id* ⇒ *bool*
where *tconf P* = *h.tconf P undefined*
notation *tconf* (‹- ⊢ - √*t*› [*51*,*51*] *50*)
lemma *tconf-conv-tconf* [*heap-independent*, *iff*]: *h.tconf P h* = *tconf P*
by(*simp add*: *tconf-def h.tconf-def*[*abs-def*])

definition *vs-conf* :: *'m prog* ⇒ (*'addr* × *addr-loc* ⇒ *'addr val set*) ⇒ *bool*
where *vs-conf P* = *h.vs-conf P undefined*
lemma *vs-conf-conv-vs-conf* [*heap-independent*, *iff*]: *h.vs-conf P h* = *vs-conf P*
by(*simp add*: *vs-conf-def h.vs-conf-def*[*abs-def*])

lemmas *vs-confI* = *h.vs-confI*[*unfolded heap-independent*]
lemmas *vs-confD* = *h.vs-confD*[*unfolded heap-independent*]

use non-speculativity to express that only type-correct values are read

primrec *vs-type-all* :: *'m prog* ⇒ *'addr* × *addr-loc* ⇒ *'addr val set*
where *vs-type-all P* (*ad*, *al*) = {*v*. ∃ *T*. *P* ⊢ *ad*@*al* : *T* ∧ *P* ⊢ *v* :≤ *T*}

lemma *vs-conf-vs-type-all* [*simp*]: *vs-conf P* (*vs-type-all P*)

by(*rule h.vs-confI*[*unfolded heap-independent*])(*simp*)

lemma *w-addrs-vs-type-all*: *w-addrs* (*vs-type-all P*) ⊆ *dom typeof-addr*
by(*auto simp add*: *w-addrs-def h.conf-def*[*unfolded heap-independent*])

lemma *w-addrs-vs-type-all-in-vs-type-all*:
(⋃ *ad* ∈ *w-addrs* (*vs-type-all P*). {(*ad*, *al*)|*al*. ∃ *T*. *P* ⊢ *ad*@*al* : *T*}) ⊆ {*adal*. *vs-type-all P adal* ≠ {}}
by(*auto simp add*: *w-addrs-def vs-type-all-def intro*: *defval-conf*)

declare *vs-type-all.simps* [*simp del*]

lemmas *vs-conf-insert-iff* = *h.vs-conf-insert-iff*[*unfolded heap-independent*]

end

locale *heap′* =
h: *heap*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate λ-. *typeof-addr heap-read heap-write*
P
for *addr2thread-id* :: (*′addr* :: *addr*) ⇒ *′thread-id*
and *thread-id2addr* :: *′thread-id* ⇒ *′addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *′heap*
and *allocate* :: *′heap* ⇒ *htype* ⇒ (*′heap* × *′addr*) *set*
and *typeof-addr* :: *′addr* ⇀ *htype*
and *heap-read* :: *′heap* ⇒ *′addr* ⇒ *addr-loc* ⇒ *′addr val* ⇒ *bool*
and *heap-write* :: *′heap* ⇒ *′addr* ⇒ *addr-loc* ⇒ *′addr val* ⇒ *′heap* ⇒ *bool*
and *P* :: *′m prog*

sublocale *heap′* < *heap-base′* .

context *heap′* **begin**

lemma *vs-conf-w-value-WriteMemD*:
⟦ *vs-conf P* (*w-value P vs ob*); *ob* = *NormalAction* (*WriteMem ad al v*) ⟧
⟹ ∃ *T*. *P* ⊢ *ad*@*al* : *T* ∧ *P* ⊢ *v* :≤ *T*
by(*auto elim*: *vs-confD*)

lemma *vs-conf-w-values-WriteMemD*:
⟦ *vs-conf P* (*w-values P vs obs*); *NormalAction* (*WriteMem ad al v*) ∈ *set obs* ⟧
⟹ ∃ *T*. *P* ⊢ *ad*@*al* : *T* ∧ *P* ⊢ *v* :≤ *T*
apply(*induct obs arbitrary*: *vs*)
apply(*auto 4 3 elim*: *vs-confD intro*: *w-values-mono*[*THEN subsetD*])
done

lemma *w-values-vs-type-all-start-heap-obs*:
assumes *wf*: *wf-syscls P*
shows *w-values P* (*vs-type-all P*) (*map snd* (*lift-start-obs h.start-tid h.start-heap-obs*)) = *vs-type-all P*
(**is** *?lhs* = *?rhs*)

proof(*rule antisym*, *rule le-funI*, *rule subsetI*)
fix *adal v*
assume *v*: *v* ∈ *?lhs adal*
obtain *ad al* **where** *adal*: *adal* = (*ad*, *al*) **by**(*cases adal*)
show *v* ∈ *?rhs adal*
proof(*rule ccontr*)
assume *v'*: ¬ *?thesis*
from *in-w-valuesD*[*OF v*[*unfolded adal*] *this*[*unfolded adal*]]
obtain *obs' wa obs''*
where *eq*: *map snd* (*lift-start-obs h.start-tid h.start-heap-obs*) = *obs'* @ *wa* # *obs''*
and *write*: *is-write-action wa*
and *loc*: (*ad*, *al*) ∈ *action-loc-aux P wa*
and *vwa*: *value-written-aux P wa al* = *v*
by *blast+*
from *write* **show** *False*
proof *cases*
case (*WriteMem ad' al' v'*)
with *vwa loc eq* **have** *WriteMem ad al v* ∈ *set h.start-heap-obs*
by(*auto simp add*: *map-eq-append-conv Cons-eq-append-conv lift-start-obs-def*)
from *h.start-heap-write-typeable*[*OF this*] *v' adal*
show *?thesis* **by**(*auto simp add*: *vs-type-all-def*)
next
case (*NewHeapElem ad' hT*)
with *vwa loc eq* **have** *NewHeapElem ad hT* ∈ *set h.start-heap-obs*
by(*auto simp add*: *map-eq-append-conv Cons-eq-append-conv lift-start-obs-def*)
hence *typeof-addr ad* = ⌊*hT*⌋
by(*rule h.NewHeapElem-start-heap-obsD*[*OF wf*])
with *v' adal loc vwa NewHeapElem* **show** *?thesis*
by(*auto simp add*: *vs-type-all-def intro*: *addr-loc-type-intros h.addr-loc-default-conf*[*unfolded heap-independent*])
qed
qed
qed(*rule w-values-greater*)

end

lemma *lprefix-lappend2I*: *lprefix xs ys* ⟹ *lprefix xs* (*lappend ys zs*)
by(*auto simp add*: *lappend-assoc lprefix-conv-lappend*)

locale *known-addrs-typing'* =
h: *known-addrs-typing*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate λ-. *typeof-addr heap-read heap-write*
allocated known-addrs
final r wfx
P
for *addr2thread-id* :: ('*addr* :: *addr*) ⇒ '*thread-id*
and *thread-id2addr* :: '*thread-id* ⇒ '*addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: '*heap*
and *allocate* :: '*heap* ⇒ *htype* ⇒ ('*heap* × '*addr*) *set*
and *typeof-addr* :: '*addr* ⇀ *htype*

and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *allocated* :: *'heap* ⇒ *'addr set*
and *known-addrs* :: *'thread-id* ⇒ *'x* ⇒ *'addr set*
and *final* :: *'x* ⇒ *bool*
and *r* :: (*'addr*, *'thread-id*, *'x*, *'heap*, *'addr*, (*'addr*, *'thread-id*) *obs-event*) *semantics* (‹- ⊢ - –-→ -› [*50*,*0*,*0*,*50*] *80*)
and *wfx* :: *'thread-id* ⇒ *'x* ⇒ *'heap* ⇒ *bool*
and *P* :: *'md prog*
+
assumes *NewHeapElem-typed*: — Should this be moved to known_addrs_typing?
⟦ *t* ⊢ (*x*, *h*) −*ta*→ (*x'*, *h'*); *NewHeapElem ad CTn* ∈ *set* {|*ta*|}$_o$; *typeof-addr ad* ≠ *None* ⟧
⟹ *typeof-addr ad* = ⌊*CTn*⌋

sublocale *known-addrs-typing'* < *heap'* **by** *unfold-locales*

context *known-addrs-typing'* **begin**

lemma *known-addrs-typeable-in-vs-type-all*:
h.if.known-addrs-state s ⊆ *dom typeof-addr*
⟹ (⋃ *a* ∈ *h.if.known-addrs-state s*. {(*a*, *al*)|*al*. ∃ *T*. *P* ⊢ *a*@*al* : *T*}) ⊆ {*adal*. *vs-type-all P adal* ≠ {}}
by(*auto 4 4 dest*: *subsetD simp add*: *vs-type-all.simps intro*: *defval-conf*)

lemma *if-NewHeapElem-typed*:
⟦ *t* ⊢ *xh* −*ta*→*i x'h'*; *NormalAction* (*NewHeapElem ad CTn*) ∈ *set* {|*ta*|}$_o$; *typeof-addr ad* ≠ *None* ⟧
⟹ *typeof-addr ad* = ⌊*CTn*⌋
by(*cases rule*: *h.mthr.init-fin.cases*)(*auto dest*: *NewHeapElem-typed*)

lemma *if-redT-NewHeapElem-typed*:
⟦ *h.mthr.if.redT s* (*t*, *ta*) *s'*; *NormalAction* (*NewHeapElem ad CTn*) ∈ *set* {|*ta*|}$_o$; *typeof-addr ad* ≠ *None* ⟧
⟹ *typeof-addr ad* = ⌊*CTn*⌋
by(*cases rule*: *h.mthr.if.redT.cases*)(*auto dest*: *if-NewHeapElem-typed*)

lemma *non-speculative-written-value-typeable*:
assumes *wfx-start*: *ts-ok wfx* (*thr* (*h.start-state f P C M vs*)) *h.start-heap*
and *wfP*: *wf-syscls P*
and *E*: *E* ∈ *h.*$\mathcal{E}$*-start f P C M vs status*
and *write*: *w* ∈ *write-actions E*
and *adal*: (*ad*, *al*) ∈ *action-loc P E w*
and *ns*: *non-speculative P* (*vs-type-all P*) (*lmap snd* (*ltake* (*enat w*) *E*))
shows ∃ *T*. *P* ⊢ *ad*@*al* : *T* ∧ *P* ⊢ *value-written P E w* (*ad*, *al*) :≤ *T*
proof −
let *?start-state* = *init-fin-lift-state status* (*h.start-state f P C M vs*)
and *?start-obs* = *lift-start-obs h.start-tid h.start-heap-obs*
and *?v* = *value-written P E w* (*ad*, *al*)

from *write* **have** *iwa*: *is-write-action* (*action-obs E w*) **by** *cases*

from *E* **obtain** *E'* **where** *E'*: *E* = *lappend* (*llist-of ?start-obs*) *E'*
and $\mathcal{E}$: *E'* ∈ *h.mthr.if.*$\mathcal{E}$ *?start-state* **by** *blast*
from $\mathcal{E}$ **obtain** *E''* **where** *E''*: *E'* = *lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) *E''*)
and *Runs*: *h.mthr.if.mthr.Runs ?start-state E''*

```
    by−(rule h.mthr.if.ℰ.cases[OF ℰ])

  have wfx': ts-ok (init-fin-lift wfx) (thr ?start-state) (shr ?start-state)
    using wfx-start by(simp add: h.shr-start-state)

  from ns E'
  have ns: non-speculative P (vs-type-all P) (lmap snd (ldropn (length (lift-start-obs h.start-tid h.start-heap-obs))
(ltake (enat w) E)))
    by(subst (asm) lappend-ltake-ldrop[where n=enat (length (lift-start-obs h.start-tid h.start-heap-obs)),
symmetric])(simp add: non-speculative-lappend min-def ltake-lappend1 w-values-vs-type-all-start-heap-obs[OF
wfP] ldrop-enat split: if-split-asm)

  show ?thesis
  proof(cases w < length ?start-obs)
    case True
    hence in-start: action-obs E w ∈ set (map snd ?start-obs)
     unfolding in-set-conv-nth E' by(simp add: lnth-lappend action-obs-def map-nth exI[where x=w])

    from iwa show ?thesis
    proof(cases)
      case (WriteMem ad' al' v')
      with adal have ad' = ad al' = al ?v = v' by(simp-all add: value-written.simps)
      with WriteMem in-start have WriteMem ad al ?v ∈ set h.start-heap-obs by auto
      thus ?thesis by(rule h.start-heap-write-typeable[unfolded heap-independent])
    next
      case (NewHeapElem ad' CTn)
      with adal have [simp]: ad' = ad by auto
      with NewHeapElem in-start have NewHeapElem ad CTn ∈ set h.start-heap-obs by auto
      with wfP have typeof-addr ad = ⌊CTn⌋ by(rule h.NewHeapElem-start-heap-obsD)
      with adal NewHeapElem show ?thesis
     by(cases al)(auto simp add: value-written.simps intro: addr-loc-type-intros h.addr-loc-default-conf[unfolded
heap-independent])
    qed
  next
    case False
    define w' where w' = w − length ?start-obs
    with write False have w'-len: enat w' < llength E'
      by(cases llength E')(auto simp add: actions-def E' elim: write-actions.cases)
    with Runs obtain m-w n-w t-w ta-w
      where E'-w: lnth E' w' = (t-w, {|ta-w|}_o ! n-w)
      and n-w: n-w < length {|ta-w|}_o
      and m-w: enat m-w < llength E''
      and w-sum: w' = (∑ i<m-w. length {|snd (lnth E'' i)|}_o) + n-w
      and E''-m-w: lnth E'' m-w = (t-w, ta-w)
      unfolding E'' by(rule h.mthr.if.actions-ℰE-aux)

    from E'-w have obs-w: action-obs E w = {|ta-w|}_o ! n-w
      using False E' w'-def by(simp add: action-obs-def lnth-lappend)

    let ?E'' = ldropn (Suc m-w) E''
    let ?m-E'' = ltake (enat m-w) E''
    have E'-unfold: E'' = lappend ?m-E'' (LCons (lnth E'' m-w) ?E'')
      unfolding ldropn-Suc-conv-ldropn[OF m-w] by simp
    hence h.mthr.if.mthr.Runs ?start-state (lappend ?m-E'' (LCons (lnth E'' m-w) ?E''))
```

using *Runs* **by** *simp*
then obtain σ' **where** σ-σ': *h.mthr.if.mthr.Trsys ?start-state* (*list-of ?m-E''*) σ'
and *Runs'*: *h.mthr.if.mthr.Runs* σ' (*LCons* (*lnth E'' m-w*) *?E''*)
by(*rule h.mthr.if.mthr.Runs-lappendE*) *simp*
from *Runs'* **obtain** σ''' **where** *red-w*: *h.mthr.if.redT* σ' (*t-w*, *ta-w*) σ'''
and *Runs''*: *h.mthr.if.mthr.Runs* σ''' *?E''*
unfolding *E''-m-w* **by** *cases*

let *?EE''* = *lmap snd* (*lappend* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) *?m-E''*)) (*llist-of* (*map* (*Pair t-w*) (*take* (*n-w* + *1*) {|*ta-w*|}$_o$))))
have *len-EE''*: *llength ?EE''* = *enat* (*w'* + *1*) **using** *n-w m-w*
apply(*simp add*: *w-sum*)
apply(*subst llength-lconcat-lfinite-conv-sum*)
apply(*simp-all add*: *split-beta plus-enat-simps*(*1*)[*symmetric*] *add-Suc-right*[*symmetric*] *del*: *plus-enat-simps*(*1*) *add-Suc-right*)
apply(*subst sum-comp-morphism*[*symmetric*, **where** *h*=*enat*])
apply(*simp-all add*: *zero-enat-def min-def le-Suc-eq*)
apply(*rule sum.cong*)
apply(*auto simp add*: *lnth-ltake less-trans*[**where** *y*=*enat m-w*])
done
have *prefix*: *lprefix ?EE''* (*lmap snd E'*) **unfolding** *E''*
by(*subst* (*2*) *E'-unfold*)(*rule lmap-lprefix*, *clarsimp simp add*: *lmap-lappend-distrib E''-m-w lprefix-lappend2I*[*OF lprefix-llist-ofI*[*OF exI*[**where** *x*=*map* (*Pair t-w*) (*drop* (*n-w* + *1*) {|*ta-w*|}$_o$)]]] *map-append*[*symmetric*])

from *iwa False* **have** *iwa'*: *is-write-action* (*action-obs E' w'*) **by**(*simp add*: *E' action-obs-def lnth-lappend w'-def*)
from *ns False*
have *non-speculative P* (*vs-type-all P*) (*lmap snd* (*ltake* (*enat w'*) *E'*))
by(*simp add*: *E' ltake-lappend lmap-lappend-distrib non-speculative-lappend ldropn-lappend2 w'-def*)
with *iwa'*
have *non-speculative P* (*vs-type-all P*) (*lappend* (*lmap snd* (*ltake* (*enat w'*) *E'*)) (*LCons* (*action-obs E' w'*) *LNil*))
by *cases*(*simp-all add*: *non-speculative-lappend*)
also have *lappend* (*lmap snd* (*ltake* (*enat w'*) *E'*)) (*LCons* (*action-obs E' w'*) *LNil*) = *lmap snd* (*ltake* (*enat* (*w'* + *1*)) *E'*)
using *w'-len* **by**(*simp add*: *ltake-Suc-conv-snoc-lnth lmap-lappend-distrib action-obs-def*)
also {
have *lprefix* (*lmap snd* (*ltake* (*enat* (*w'* + *1*)) *E'*)) (*lmap snd E'*) **by**(*rule lmap-lprefix*) *simp*
with *prefix* **have** *lprefix ?EE''* (*lmap snd* (*ltake* (*enat* (*w'* + *1*)) *E'*)) ∨
lprefix (*lmap snd* (*ltake* (*enat* (*w'* + *1*)) *E'*)) *?EE''*
by(*rule lprefix-down-linear*)
moreover have *llength* (*lmap snd* (*ltake* (*enat* (*w'* + *1*)) *E'*)) = *enat* (*w'* + *1*)
using *w'-len* **by**(*cases llength E'*) *simp-all*
ultimately have *lmap snd* (*ltake* (*enat* (*w'* + *1*)) *E'*) = *?EE''*
using *len-EE''* **by**(*auto dest*: *lprefix-llength-eq-imp-eq*) }
finally
have *ns1*: *non-speculative P* (*vs-type-all P*) (*llist-of* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) (*list-of ?m-E''*))))
and *ns2*: *non-speculative P* (*w-values P* (*vs-type-all P*) (*map snd* (*list-of* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) *?m-E''*))))) (*llist-of* (*take* (*Suc n-w*) {|*ta-w*|}$_o$))
by(*simp-all add*: *lmap-lappend-distrib non-speculative-lappend split-beta lconcat-llist-of*[*symmetric*] *lmap-lconcat llist.map-comp o-def split-def list-of-lmap*[*symmetric*] *del*: *list-of-lmap*)

```
    have vs-conf P (vs-type-all P) by simp
    with σ-σ' wfx' ns1
    have wfx': ts-ok (init-fin-lift wfx) (thr σ') (shr σ')
     and vs-conf: vs-conf P (w-values P (vs-type-all P) (concat (map (λ(t, ta). {|ta|}o) (list-of ?m-E'')))))
      by(rule h.if-RedT-non-speculative-invar[unfolded h.mthr.if.RedT-def heap-independent])+

    have concat (map (λ(t, ta). {|ta|}o) (list-of ?m-E'')) = map snd (list-of (lconcat (lmap (λ(t, ta).
llist-of (map (Pair t) {|ta|}o)) ?m-E'')))
      by(simp add: split-def lmap-lconcat llist.map-comp o-def list-of-lconcat map-concat)
    with vs-conf have vs-conf P (w-values P (vs-type-all P) ...) by simp
    with red-w wfx' ns2
    have vs-conf': vs-conf P (w-values P (w-values P (vs-type-all P) (map snd (list-of (lconcat (lmap
(λ(t, ta). llist-of (map (Pair t) {|ta|}o)) ?m-E'')))))) (take (Suc n-w) {|ta-w|}o))
      (is vs-conf - ?vs')
      by(rule h.if-redT-non-speculative-vs-conf[unfolded heap-independent])

    from len-EE'' have enat w' < llength ?EE'' by simp
    from w'-len have lnth ?EE'' w' = action-obs E' w'
      using lprefix-lnthD[OF prefix ‹enat w' < llength ?EE''›] by(simp add: action-obs-def)
    hence ... ∈ lset ?EE'' using ‹enat w' < llength ?EE''› unfolding lset-conv-lnth by(auto intro!:
exI)
     also have ... ⊆ set (map snd (list-of (lconcat (lmap (λ(t, ta). llist-of (map (Pair t) {|ta|}o))
?m-E''))) @ take (Suc n-w) {|ta-w|}o)
    by(auto 4 4 intro: rev-image-eqI rev-bexI simp add: split-beta lset-lconcat-lfinite dest: lset-lappend[THEN
subsetD])
    also have action-obs E' w' = action-obs E w
      using False by(simp add: E' w'-def lnth-lappend action-obs-def)
    also note obs-w-in-set = calculation and calculation = nothing

    from iwa have ?v ∈ w-values P (vs-type-all P) (map snd (list-of (lconcat (lmap (λ(t, ta). llist-of
(map (Pair t) {|ta|}o)) ?m-E''))) @ take (Suc n-w) {|ta-w|}o) (ad, al)
    proof(cases)
      case (WriteMem ad' al' v')
      with adal have ad' = ad al' = al ?v = v' by(simp-all add: value-written.simps)
      with obs-w-in-set WriteMem show ?thesis
        by −(rule w-values-WriteMemD, simp)
    next
      case (NewHeapElem ad' CTn)
      with adal have [simp]: ad' = ad and v: ?v = addr-loc-default P CTn al
        by(auto simp add: value-written.simps)
      with obs-w-in-set NewHeapElem adal show ?thesis
        by(unfold v)(rule w-values-new-actionD, simp-all)
    qed
    hence ?v ∈ ?vs' (ad, al) by simp
    with vs-conf' show ∃ T. P ⊢ ad@al : T ∧ P ⊢ ?v :≤ T
      by(rule h.vs-confD[unfolded heap-independent])
  qed
qed

lemma hb-read-value-typeable:
  assumes wfx-start: ts-ok wfx (thr (h.start-state f P C M vs)) h.start-heap
    (is ts-ok wfx (thr ?start-state) -)
  and wfP: wf-syscls P
  and E: E ∈ h.ℰ-start f P C M vs status
```

```
  and wf: P ⊢ (E, ws) √
  and races: ⋀a ad al v. ⟦ enat a < llength E; action-obs E a = NormalAction (ReadMem ad al v); ¬
P,E ⊢ ws a ≤hb a ⟧
           ⟹ ∃ T. P ⊢ ad@al : T ∧ P ⊢ v :≤ T
  and r: enat a < llength E
  and read: action-obs E a = NormalAction (ReadMem ad al v)
  shows ∃ T. P ⊢ ad@al : T ∧ P ⊢ v :≤ T
using r read
proof(induction a arbitrary: ad al v rule: less-induct)
  case (less a)
  note r = ‹enat a < llength E›
    and read = ‹action-obs E a = NormalAction (ReadMem ad al v)›
  show ?case
  proof(cases P,E ⊢ ws a ≤hb a)
    case False with r read show ?thesis by(rule races)
  next
    case True
    note hb = this
    hence ao: E ⊢ ws a ≤a a by(rule happens-before-into-action-order)

    from wf have ws: is-write-seen P E ws by(rule wf-exec-is-write-seenD)
    from r have a ∈ actions E by(simp add: actions-def)
    hence a ∈ read-actions E using read ..
    from is-write-seenD[OF ws this read]
    have write: ws a ∈ write-actions E
      and adal-w: (ad, al) ∈ action-loc P E (ws a)
      and written: value-written P E (ws a) (ad, al) = v by simp-all
    from write have iwa: is-write-action (action-obs E (ws a)) by cases

    let ?start-state = init-fin-lift-state status (h.start-state f P C M vs)
      and ?start-obs = lift-start-obs h.start-tid h.start-heap-obs

    show ?thesis
    proof(cases ws a < a)
      case True
      let ?EE'' = lmap snd (ltake (enat (ws a)) E)

      have non-speculative P (vs-type-all P) ?EE''
      proof(rule non-speculative-nthI)
        fix i ad' al' v'
        assume i: enat i < llength ?EE''
          and nth-i: lnth ?EE'' i = NormalAction (ReadMem ad' al' v')

        from i have i < ws a by simp
        hence i': i < a using True by(simp)
        moreover
        with r have enat i < llength E by(metis enat-ord-code(2) order-less-trans)
        moreover
        with nth-i i ‹i < ws a›
        have action-obs E i = NormalAction (ReadMem ad' al' v')
          by(simp add: action-obs-def lnth-ltake ac-simps)
        ultimately have ∃ T. P ⊢ ad'@al' : T ∧ P ⊢ v' :≤ T by(rule less.IH)
        hence v' ∈ vs-type-all P (ad', al') by(simp add: vs-type-all.simps)
        thus v' ∈ w-values P (vs-type-all P) (list-of (ltake (enat i) ?EE'')) (ad', al')
```

```
      by(rule w-values-mono[THEN subsetD])
    qed
    with wfx-start wfP E write adal-w
    show ?thesis unfolding written[symmetric] by(rule non-speculative-written-value-typeable)
  next
    case False

    from E obtain E' where E': E = lappend (llist-of ?start-obs) E'
      and ℰ: E' ∈ h.mthr.if.ℰ ?start-state by blast
    from ℰ obtain E'' where E'': E' = lconcat (lmap (λ(t, ta). llist-of (map (Pair t) {|ta|}_o)) E'')
      and Runs: h.mthr.if.mthr.Runs ?start-state E''
      by−(rule h.mthr.if.ℰ.cases[OF ℰ])

    have wfx': ts-ok (init-fin-lift wfx) (thr ?start-state) (shr ?start-state)
      using wfx-start by(simp add: h.shr-start-state)

    have a-start: ¬ a < length ?start-obs
    proof
      assume a < length ?start-obs
      with read have NormalAction (ReadMem ad al v) ∈ snd ` set ?start-obs
        unfolding set-map[symmetric] in-set-conv-nth
        by(auto simp add: E' lnth-lappend action-obs-def)
      hence ReadMem ad al v ∈ set h.start-heap-obs by auto
      thus False by(simp add: h.start-heap-obs-not-Read)
    qed
    hence ws-a-not-le: ¬ ws a < length ?start-obs using False by simp

    define w where w = ws a − length ?start-obs
    from write ws-a-not-le w-def
    have enat w < llength (lconcat (lmap (λ(t, ta). llist-of (map (Pair t) {|ta|}_o)) E''))
      by(cases llength (lconcat (lmap (λ(t, ta). llist-of (map (Pair t) {|ta|}_o)) E'')))(auto simp add:
actions-def E' E'' elim: write-actions.cases)
    with Runs obtain m-w n-w t-w ta-w
      where E'-w: lnth E' w = (t-w, {|ta-w|}_o ! n-w)
      and n-w: n-w < length {|ta-w|}_o
      and m-w: enat m-w < llength E''
      and w-sum: w = (∑ i<m-w. length {|snd (lnth E'' i)|}_o) + n-w
      and E''-m-w: lnth E'' m-w = (t-w, ta-w)
      unfolding E'' by(rule h.mthr.if.actions-ℰE-aux)

    from E'-w have obs-w: action-obs E (ws a) = {|ta-w|}_o ! n-w
      using ws-a-not-le E' w-def by(simp add: action-obs-def lnth-lappend)

    let ?E'' = ldropn (Suc m-w) E''
    let ?m-E'' = ltake (enat m-w) E''
    have E'-unfold: E'' = lappend ?m-E'' (LCons (lnth E'' m-w) ?E'')
      unfolding ldropn-Suc-conv-ldropn[OF m-w] by simp
    hence h.mthr.if.mthr.Runs ?start-state (lappend ?m-E'' (LCons (lnth E'' m-w) ?E''))
      using Runs by simp
    then obtain σ' where σ-σ': h.mthr.if.mthr.Trsys ?start-state (list-of ?m-E'') σ'
      and Runs': h.mthr.if.mthr.Runs σ' (LCons (lnth E'' m-w) ?E'')
      by(rule h.mthr.if.mthr.Runs-lappendE) simp
    from Runs' obtain σ''' where red-w: h.mthr.if.redT σ' (t-w, ta-w) σ'''
      and Runs'': h.mthr.if.mthr.Runs σ''' ?E''
```

unfolding *E''-m-w* **by** *cases*

from *write* ‹*a* ∈ *read-actions E*› **have** *ws a* ≠ *a* **by**(*auto dest*: *read-actions-not-write-actions*)
with *False* **have** *ws a* > *a* **by** *simp*
with *ao* **have** *new*: *is-new-action* (*action-obs E* (*ws a*))
by(*simp add*: *action-order-def split*: *if-split-asm*)
then obtain *CTn* **where** *obs-w'*: *action-obs E* (*ws a*) = *NormalAction* (*NewHeapElem ad CTn*)
using *adal-w* **by** *cases auto*

define *a'* **where** *a'* = *a* − *length ?start-obs*
with *False w-def*
have *enat a'* < *llength* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) ⦃*ta*⦄$_o$)) *E''*))
by(*simp add*: *le-less-trans*[*OF* - ‹*enat w* < *llength* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) ⦃*ta*⦄$_o$)) *E''*))›])
with *Runs* **obtain** *m-a n-a t-a ta-a*
where *E'-a*: *lnth E' a'* = (*t-a*, ⦃*ta-a*⦄$_o$! *n-a*)
and *n-a*: *n-a* < *length* ⦃*ta-a*⦄$_o$
and *m-a*: *enat m-a* < *llength E''*
and *a-sum*: *a'* = (∑ *i*<*m-a*. *length* ⦃*snd* (*lnth E'' i*)⦄$_o$) + *n-a*
and *E''-m-a*: *lnth E'' m-a* = (*t-a*, *ta-a*)
unfolding *E''* **by**(*rule h.mthr.if.actions-*$\mathcal{E}$*E-aux*)

from *a-start E'-a read* **have** *obs-a*: ⦃*ta-a*⦄$_o$! *n-a* = *NormalAction* (*ReadMem ad al v*)
using *E' w-def* **by**(*simp add*: *action-obs-def lnth-lappend a'-def*)

let *?E''* = *ldropn* (*Suc m-a*) *E''*
let *?m-E''* = *ltake* (*enat m-a*) *E''*
have *E'-unfold*: *E''* = *lappend ?m-E''* (*LCons* (*lnth E'' m-a*) *?E''*)
unfolding *ldropn-Suc-conv-ldropn*[*OF m-a*] **by** *simp*
hence *h.mthr.if.mthr.Runs ?start-state* (*lappend ?m-E''* (*LCons* (*lnth E'' m-a*) *?E''*))
using *Runs* **by** *simp*
then obtain *σ''* **where** *σ-σ''*: *h.mthr.if.mthr.Trsys ?start-state* (*list-of ?m-E''*) *σ''*
and *Runs''*: *h.mthr.if.mthr.Runs σ''* (*LCons* (*lnth E'' m-a*) *?E''*)
by(*rule h.mthr.if.mthr.Runs-lappendE*) *simp*
from *Runs''* **obtain** *σ'''* **where** *red-a*: *h.mthr.if.redT σ''* (*t-a*, *ta-a*) *σ'''*
and *Runs''*: *h.mthr.if.mthr.Runs σ'''* *?E''*
unfolding *E''-m-a* **by** *cases*

let *?EE''* = *llist-of* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) (*list-of ?m-E''*)))
from *m-a* **have** *enat m-a* ≤ *llength E''* **by** *simp*
hence *len-EE''*: *llength ?EE''* = *enat* (*a'* − *n-a*)
by(*simp add*: *a-sum length-concat sum-list-sum-nth atLeast0LessThan length-list-of-conv-the-enat min-def split-beta lnth-ltake*)
have *prefix*: *lprefix ?EE''* (*lmap snd E'*) **unfolding** *E''*
by(*subst* (2) *E'-unfold*)(*simp add*: *lmap-lappend-distrib lmap-lconcat llist.map-comp o-def split-def lconcat-llist-of*[*symmetric*] *lmap-llist-of*[*symmetric*] *lprefix-lappend2I del*: *lmap-llist-of*)

have *ns*: *non-speculative P* (*vs-type-all P*) *?EE''*
proof(*rule non-speculative-nthI*)
fix *i ad' al' v'*
assume *i*: *enat i* < *llength ?EE''*
and *lnth-i*: *lnth ?EE'' i* = *NormalAction* (*ReadMem ad' al' v'*)
and *non-speculative P* (*vs-type-all P*) (*ltake* (*enat i*) *?EE''*)

```
      let ?i = i + length ?start-obs

      from i len-EE'' have i < a' by simp
      hence i': ?i < a by(simp add: a'-def)
      moreover
      hence enat ?i < llength E using ‹enat a < llength E› by(simp add: less-trans[where y=enat
a])
      moreover have enat i < llength E' using i
        by -(rule less-le-trans[OF - lprefix-llength-le[OF prefix], simplified], simp)
      from lprefix-lnthD[OF prefix i] lnth-i
      have lnth (lmap snd E') i = NormalAction (ReadMem ad' al' v') by simp
      hence action-obs E ?i = NormalAction (ReadMem ad' al' v') using ‹enat i < llength E'›
        by(simp add: E' action-obs-def lnth-lappend E'')
      ultimately have ∃ T. P ⊢ ad'@al' : T ∧ P ⊢ v' :≤ T by(rule less.IH)
      hence v' ∈ vs-type-all P (ad', al') by(simp add: vs-type-all.simps)
      thus v' ∈ w-values P (vs-type-all P) (list-of (ltake (enat i) ?EE'')) (ad', al')
        by(rule w-values-mono[THEN subsetD])
    qed

    have vs-conf P (vs-type-all P) by simp
    with σ-σ'' wfx' ns
    have wfx'': ts-ok (init-fin-lift wfx) (thr σ'') (shr σ'')
     and vs'': vs-conf P (w-values P (vs-type-all P) (concat (map (λ(t, ta). {|ta|}_o) (list-of ?m-E'')))
      by(rule h.if-RedT-non-speculative-invar[unfolded heap-independent h.mthr.if.RedT-def])+

    note red-w moreover
    from n-w obs-w obs-w' have NormalAction (NewHeapElem ad CTn) ∈ set {|ta-w|}_o
      unfolding in-set-conv-nth by auto
    moreover
    have ta-a-read: NormalAction (ReadMem ad al v) ∈ set {|ta-a|}_o
      using n-a obs-a unfolding in-set-conv-nth by blast
    from red-a have ∃ T. P ⊢ ad@al : T
    proof(cases)
      case (redT-normal x x' h')
      from wfx'' ‹thr σ'' t-a = ⌊(x, no-wait-locks)⌋›
      have init-fin-lift wfx t-a x (shr σ'') by(rule ts-okD)
      with ‹t-a ⊢ (x, shr σ'') -ta-a→i (x', h')›
      show ?thesis using ta-a-read
        by(rule h.init-fin-red-read-typeable[unfolded heap-independent])
    next
      case redT-acquire thus ?thesis using n-a obs-a ta-a-read by auto
    qed
    hence typeof-addr ad ≠ None by(auto elim: addr-loc-type-cases)
    ultimately have typeof-addr ad = ⌊CTn⌋ by(rule if-redT-NewHeapElem-typed)
    with written adal-w obs-w' show ?thesis
    by(cases al)(auto simp add: value-written.simps intro: addr-loc-type-intros h.addr-loc-default-conf[unfolded
heap-independent])
  qed
 qed
qed

theorem
 assumes wfx-start: ts-ok wfx (thr (h.start-state f P C M vs)) h.start-heap
 and wfP: wf-syscls P
```

```
  and justified: P ⊢ (E, ws) weakly-justified-by J
  and J: range (justifying-exec ∘ J) ⊆ h.ℰ-start f P C M vs status
  shows read-value-typeable-justifying:
    ⟦ 0 < n; enat a < llength (justifying-exec (J n));
      action-obs (justifying-exec (J n)) a = NormalAction (ReadMem ad al v) ⟧
    ⟹ ∃ T. P ⊢ ad@al : T ∧ P ⊢ v :≤ T
  and read-value-typeable-justifed:
    ⟦ E ∈ h.ℰ-start f P C M vs status; P ⊢ (E, ws) √;
       enat a < llength E; action-obs E a = NormalAction (ReadMem ad al v) ⟧
    ⟹ ∃ T. P ⊢ ad@al : T ∧ P ⊢ v :≤ T
proof −
  let ?E = λn. justifying-exec (J n)
    and ?φ = λn. action-translation (J n)
    and ?C = λn. committed (J n)
    and ?ws = λn. justifying-ws (J n)
  let ?ℰ = h.ℰ-start f P C M vs status
    and ?start-obs = lift-start-obs h.start-tid h.start-heap-obs
  { fix a n
    assume enat a < llength (justifying-exec (J n))
      and action-obs (justifying-exec (J n)) a = NormalAction (ReadMem ad al v)
      and n > 0
    thus ∃ T. P ⊢ ad@al : T ∧ P ⊢ v :≤ T
    proof(induction n arbitrary: a ad al v)
      case 0 thus ?case by simp
    next
      case (Suc n')
      define n where n = Suc n'
      with Suc have n: 0 < n and a: enat a < llength (?E n)
        and a-obs: action-obs (?E n) a = NormalAction (ReadMem ad al v)
        by simp-all
      have wf-n: P ⊢ (?E n, ?ws n) √
        using justified by(simp add: justification-well-formed-def)
      from J have E: ?E n ∈ ?ℰ
        and E': ?E n' ∈ ?ℰ by auto
      from a a-obs wfx-start wfP E wf-n show ?case
      proof(rule hb-read-value-typeable[rotated −2])
        fix a' ad' al' v'
        assume a': enat a' < llength (?E n)
          and a'-obs: action-obs (?E n) a' = NormalAction (ReadMem ad' al' v')
          and nhb: ¬ P,?E n ⊢ ?ws n a' ≤hb a'
        from a' have a' ∈ actions (?E n) by(simp add: actions-def)
        hence read-a': a' ∈ read-actions (?E n) using a'-obs ..
        with justified nhb have committed': ?φ n a' ∈ ?φ n' ‘ ?C n'
          unfolding is-weakly-justified-by.simps n-def uncommitted-reads-see-hb-def by blast

        from justified have wfa-n: wf-action-translation E (J n)
          and wfa-n': wf-action-translation E (J n') by(simp-all add: wf-action-translations-def)
        hence inj-n: inj-on (?φ n) (actions (?E n))
          and inj-n': inj-on (?φ n') (actions (?E n'))
          by(blast dest: wf-action-translation-on-inj-onD)+
        from justified have C-n: ?C n ⊆ actions (?E n)
          and C-n': ?C n' ⊆ actions (?E n')
          and wf-n': P ⊢ (?E n', ?ws n') √
          by(simp-all add: committed-subset-actions-def justification-well-formed-def)
```

from *justified* **have** *?φ n' ‘ ?C n' ⊆ ?φ n ‘ ?C n*
unfolding *n-def* **by**(*simp add*: *is-commit-sequence-def*)
with *n-def committed'* **have** *?φ n a' ∈ ?φ n ‘ ?C n* **by** *auto*
with *inj-n C-n* **have** *committed*: *a' ∈ ?C n*
using ‹*a' ∈ actions* (*?E n*)› **by**(*auto dest*: *inj-onD*)
with *justified read-a'* **have** *ws-committed*: *ws* (*?φ n a'*) *∈ ?φ n ‘ ?C n*
by(*rule weakly-justified-write-seen-hb-read-committed*)

from *wf-n* **have** *ws-n*: *is-write-seen P* (*?E n*) (*?ws n*) **by**(*rule wf-exec-is-write-seenD*)
from *is-write-seenD*[*OF this read-a' a'-obs*]
have *ws-write*: *?ws n a' ∈ write-actions* (*?E n*)
and *adal*: (*ad'*, *al'*) *∈ action-loc P* (*?E n*) (*?ws n a'*)
and *written*: *value-written P* (*?E n*) (*?ws n a'*) (*ad'*, *al'*) = *v'* **by** *simp-all*

define *a''* **where** *a'' = inv-into* (*actions* (*?E n'*)) (*?φ n'*) (*?φ n a'*)
from *C-n' n committed'* **have** *?φ n a' ∈ ?φ n' ‘ actions* (*?E n'*) **by** *auto*
hence *a''*: *?φ n' a'' = ?φ n a'*
and *a''-action*: *a'' ∈ actions* (*?E n'*) **using** *inj-n' committed' n*
by(*simp-all add*: *a''-def f-inv-into-f inv-into-into*)
hence *committed''*: *a'' ∈ ?C n'* **using** *committed' n inj-n' C-n'* **by**(*fastforce dest*: *inj-onD*)

from *committed committed'' wfa-n wfa-n' a''* **have** *action-obs* (*?E n'*) *a'' ≈ action-obs* (*?E n*) *a'*
by(*auto dest*!: *wf-action-translation-on-actionD intro*: *sim-action-trans sim-action-sym*)
with *a'-obs committed'' C-n'* **have** *read-a''*: *a'' ∈ read-actions* (*?E n'*)
by(*auto intro*: *read-actions.intros*)

then obtain *ad'' al'' v''*
where *a''-obs*: *action-obs* (*?E n'*) *a'' = NormalAction* (*ReadMem ad'' al'' v''*) **by** *cases*

from *committed''* **have** *n' > 0* **using** *justified*
by(*cases n'*)(*simp-all add*: *is-commit-sequence-def*)
then obtain *n''* **where** *n''*: *n' = Suc n''* **by**(*cases n'*) *simp-all*

from *justified* **have** *wfa-n''*: *wf-action-translation E* (*J n''*) **by**(*simp add*: *wf-action-translations-def*)
hence *inj-n''*: *inj-on* (*?φ n''*) (*actions* (*?E n''*)) **by**(*blast dest*: *wf-action-translation-on-inj-onD*)+
from *justified* **have** *C-n''*: *?C n'' ⊆ actions* (*?E n''*) **by**(*simp add*: *committed-subset-actions-def*)

from *justified committed' committed'' n-def read-a' read-a'' n*
have *?φ n* (*?ws n* (*inv-into* (*actions* (*?E n*)) (*?φ n*) (*?φ n' a''*))) = *ws* (*?φ n' a''*)
by(*simp add*: *write-seen-committed-def*)
hence *?φ n* (*?ws n a'*) = *ws* (*?φ n a'*) **using** *inj-n* ‹*a' ∈ actions* (*?E n*)› **by**(*simp add*: *a''*)

from *ws-committed* **obtain** *w* **where** *w*: *ws* (*?φ n a'*) = *?φ n w*
and *committed-w*: *w ∈ ?C n* **by** *blast*
from *committed-w C-n* **have** *w ∈ actions* (*?E n*) **by** *blast*
hence *w-def*: *w = ?ws n a'* **using** ‹*?φ n* (*?ws n a'*) = *ws* (*?φ n a'*)› *inj-n ws-write*
unfolding *w* **by**(*auto dest*: *inj-onD*)
have *committed-ws*: *?ws n a' ∈ ?C n* **using** *committed-w* **by**(*simp add*: *w-def*)

with *wfa-n* **have** *sim-ws*: *action-obs* (*?E n*) (*?ws n a'*) *≈ action-obs E* (*?φ n* (*?ws n a'*))
by(*blast dest*: *wf-action-translation-on-actionD*)

from *wfa-n committed-ws* **have** *sim-ws*: *action-obs* (*?E n*) (*?ws n a′*) ≈ *action-obs E* (*?φ n* (*?ws n a′*))
by(*blast dest*: *wf-action-translation-on-actionD*)
with *adal* **have** *adal-E*: (*ad′*, *al′*) ∈ *action-loc P E* (*?φ n* (*?ws n a′*))
by(*simp add*: *action-loc-aux-sim-action*)

have ∃ *w* ∈ *write-actions* (*?E n′*). (*ad′*, *al′*) ∈ *action-loc P* (*?E n′*) *w* ∧ *value-written P* (*?E n′*) *w* (*ad′*, *al′*) = *v′*
proof(*cases ?φ n′ a″* ∈ *?φ n″* ‘ *?C n″*)
case *True*
then obtain *a‴* **where** *a‴*: *?φ n″ a‴* = *?φ n′ a″*
and *committed‴*: *a‴* ∈ *?C n″* **by** *auto*
from *committed‴ C-n″* **have** *a‴-action*: *a‴* ∈ *actions* (*?E n″*) **by** *auto*

from *committed″ committed‴ wfa-n′ wfa-n″ a‴* **have** *action-obs* (*?E n″*) *a‴* ≈ *action-obs* (*?E n′*) *a″*
by(*auto dest*!: *wf-action-translation-on-actionD intro*: *sim-action-trans sim-action-sym*)
with *read-a″ committed‴ C-n″* **have** *read-a‴*: *a‴* ∈ *read-actions* (*?E n″*)
by *cases*(*auto intro*: *read-actions.intros*)

hence *?φ n′* (*?ws n′* (*inv-into* (*actions* (*?E n′*)) (*?φ n′*) (*?φ n″ a‴*))) = *ws* (*?φ n″ a‴*)
using *justified committed‴*
unfolding *is-weakly-justified-by.simps n″ Let-def write-seen-committed-def* **by** *blast*
also have *inv-into* (*actions* (*?E n′*)) (*?φ n′*) (*?φ n″ a‴*) = *a″*
using *a‴ inj-n′ a″-action* **by**(*simp*)
also note *a‴* **also note** *a″*
finally have *φ-n′*: *?φ n′* (*?ws n′ a″*) = *ws* (*?φ n a′*) **.**
then have *ws* (*?φ n a′*) = *?φ n′* (*?ws n′ a″*) **..**
with ‹*?φ n* (*?ws n a′*) = *ws* (*?φ n a′*)›[*symmetric*]
have *eq-ws*: *?φ n′* (*?ws n′ a″*) = *?φ n* (*?ws n a′*) **by** *simp*

from *wf-n′*[*THEN wf-exec-is-write-seenD*, *THEN is-write-seenD*, *OF read-a″ a″-obs*]
have *ws-write′*: *?ws n′ a″* ∈ *write-actions* (*?E n′*) **by** *simp*

from *justified read-a″ committed″*
have *ws* (*?φ n′ a″*) ∈ *?φ n′* ‘ *?C n′* **by**(*rule weakly-justified-write-seen-hb-read-committed*)
then obtain *w′* **where** *w′*: *ws* (*?φ n′ a″*) = *?φ n′ w′*
and *committed-w′*: *w′* ∈ *?C n′* **by** *blast*
from *committed-w′ C-n′* **have** *w′* ∈ *actions* (*?E n′*) **by** *blast*
hence *w′-def*: *w′* = *?ws n′ a″* **using** *φ-n′ inj-n′ ws-write′*
unfolding *w′ a″*[*symmetric*] **by**(*auto dest*: *inj-onD*)
with *committed-w′* **have** *committed-ws″*: *?ws n′ a″* ∈ *committed* (*J n′*) **by** *simp*
with *committed-ws wfa-n wfa-n′ eq-ws*
have *action-obs* (*?E n′*) (*?ws n′ a″*) ≈ *action-obs* (*?E n*) (*?ws n a′*)
by(*auto dest*!: *wf-action-translation-on-actionD intro*: *sim-action-trans sim-action-sym*)
hence *adal-eq*: *action-loc P* (*?E n′*) (*?ws n′ a″*) = *action-loc P* (*?E n*) (*?ws n a′*)
by(*simp add*: *action-loc-aux-sim-action*)
with *adal* **have** *adal′*: (*ad′*, *al′*) ∈ *action-loc P* (*?E n′*) (*?ws n′ a″*) **by**(*simp add*: *action-loc-aux-sim-action*)

from *committed-ws″* **have** *?ws n′ a″* ∈ *actions* (*?E n′*) **using** *C-n′* **by** *blast*
with *ws-write* ‹*action-obs* (*?E n′*) (*?ws n′ a″*) ≈ *action-obs* (*?E n*) (*?ws n a′*)›
have *ws-write″*: *?ws n′ a″* ∈ *write-actions* (*?E n′*)
by(*cases*)(*auto intro*: *write-actions.intros simp add*: *sim-action-is-write-action-eq*)

from *wfa-n′ committed-ws″*
have *sim-ws′*: *action-obs* (*?E n′*) (*?ws n′ a″*) ≈ *action-obs E* (*?φ n′* (*?ws n′ a″*))
by(*blast dest*: *wf-action-translation-on-actionD*)
with *adal′* **have** *adal′-E*: (*ad′*, *al′*) ∈ *action-loc P E* (*?φ n′* (*?ws n′ a″*))
by(*simp add*: *action-loc-aux-sim-action*)

from *justified committed-ws ws-write adal-E*
have *value-written P* (*?E n*) (*?ws n a′*) (*ad′*, *al′*) = *value-written P E* (*?φ n* (*?ws n a′*)) (*ad′*, *al′*)
unfolding *is-weakly-justified-by.simps Let-def value-written-committed-def* **by** *blast*
also note *eq-ws*[*symmetric*]
also from *justified committed-ws″ ws-write″ adal′-E*
have *value-written P E* (*?φ n′* (*?ws n′ a″*)) (*ad′*, *al′*) = *value-written P* (*?E n′*) (*?ws n′ a″*) (*ad′*, *al′*)
unfolding *is-weakly-justified-by.simps Let-def value-written-committed-def* **by**(*blast dest*: *sym*)
finally show *?thesis* **using** *written ws-write″ adal′* **by** *auto*
next
case *False*
with *justified read-a″ committed″*
have *ws* (*?φ n′ a″*) ∈ *?φ n″* ‘ *?C n″*
unfolding *is-weakly-justified-by.simps Let-def n″ committed-reads-see-committed-writes-weak-def* **by** *blast*
with *a″* **obtain** *w* **where** *w*: *?φ n″ w* = *ws* (*?φ n a′*)
and *committed-w*: *w* ∈ *?C n″* **by** *auto*
from *justified* **have** *?φ n″* ‘ *?C n″* ⊆ *?φ n′* ‘ *?C n′* **by**(*simp add*: *is-commit-sequence-def n″*)
with *committed-w w*[*symmetric*] **have** *ws* (*?φ n a′*) ∈ *?φ n′* ‘ *?C n′* **by**(*auto*)
then obtain *w′* **where** *w′*: *ws* (*?φ n a′*) = *?φ n′ w′* **and** *committed-w′*: *w′* ∈ *?C n′* **by** *blast*
from *wfa-n′ committed-w′* **have** *action-obs* (*?E n′*) *w′* ≈ *action-obs E* (*?φ n′ w′*)
by(*blast dest*: *wf-action-translation-on-actionD*)
from *this*[*folded w′*, *folded* ‹*?φ n* (*?ws n a′*) = *ws* (*?φ n a′*)›] *sim-ws*[*symmetric*]
have *sim-w′*: *action-obs* (*?E n′*) *w′* ≈ *action-obs* (*?E n*) (*?ws n a′*) **by**(*rule sim-action-trans*)
with *ws-write committed-w′ C-n′* **have** *write-w′*: *w′* ∈ *write-actions* (*?E n′*)
by(*cases*)(*auto intro*!: *write-actions.intros simp add*: *sim-action-is-write-action-eq*)
hence *value-written P* (*?E n′*) *w′* (*ad′*, *al′*) = *value-written P E* (*?φ n′ w′*) (*ad′*, *al′*)
using *adal-E committed-w′ justified*
unfolding ‹*?φ n* (*?ws n a′*) = *ws* (*?φ n a′*)› *w′ is-weakly-justified-by.simps Let-def value-written-committed-def* **by** *blast*
also note *w′*[*symmetric*]
also note ‹*?φ n* (*?ws n a′*) = *ws* (*?φ n a′*)›[*symmetric*]
also have *value-written P E* (*?φ n* (*?ws n a′*)) (*ad′*, *al′*) = *value-written P* (*?E n*) (*?ws n a′*) (*ad′*, *al′*)
using *justified committed-ws ws-write adal-E*
unfolding *is-weakly-justified-by.simps Let-def value-written-committed-def* **by**(*blast dest*: *sym*)
also have (*ad′*, *al′*) ∈ *action-loc P* (*?E n′*) *w′* **using** *sim-w′ adal* **by**(*simp add*: *action-loc-aux-sim-action*)
ultimately show *?thesis* **using** *written write-w′* **by** *auto*
qed
then obtain *w* **where** *w*: *w* ∈ *write-actions* (*?E n′*)
and *adal*: (*ad′*, *al′*) ∈ *action-loc P* (*?E n′*) *w*
and *written*: *value-written P* (*?E n′*) *w* (*ad′*, *al′*) = *v′* **by** *blast*
from *w* **have** *w-len*: *enat w* < *llength* (*?E n′*)
by(*cases*)(*simp add*: *actions-def*)

```
      let ?EE'' = lmap snd (ltake (enat w) (?E n'))
      have non-speculative P (vs-type-all P) ?EE''
      proof(rule non-speculative-nthI)
        fix i ad al v
        assume i: enat i < llength ?EE''
          and i-nth: lnth ?EE'' i = NormalAction (ReadMem ad al v)
          and ns: non-speculative P (vs-type-all P) (ltake (enat i) ?EE'')

        from i w-len have i < w by(simp add: min-def not-le split: if-split-asm)
        with w-len have enat i < llength (?E n') by(simp add: less-trans[where y=enat w])
        moreover
        from i-nth i ‹i < w› w-len
        have action-obs (?E n') i = NormalAction (ReadMem ad al v)
          by(simp add: action-obs-def ac-simps less-trans[where y=enat w] lnth-ltake)
        moreover from n'' have 0 < n' by simp
        ultimately have ∃ T. P ⊢ ad@al : T ∧ P ⊢ v :≤ T by(rule Suc.IH)
        hence v ∈ vs-type-all P (ad, al) by(simp add: vs-type-all.simps)
        thus v ∈ w-values P (vs-type-all P) (list-of (ltake (enat i) ?EE'')) (ad, al)
          by(rule w-values-mono[THEN subsetD])
      qed
      with wfx-start wfP E' w adal
      show ∃ T. P ⊢ ad'@al' : T ∧ P ⊢ v' :≤ T
        unfolding written[symmetric] by(rule non-speculative-written-value-typeable)
    qed
  qed
}
note justifying = this

assume a: enat a < llength E
  and read: action-obs E a = NormalAction (ReadMem ad al v)
  and E: E ∈ h.ℰ-start f P C M vs status
  and wf: P ⊢ (E, ws) √
from a have action: a ∈ actions E by(auto simp add: actions-def action-obs-def)
with justified obtain n a' where a': a = ?φ n a'
  and committed': a' ∈ ?C n by(auto simp add: is-commit-sequence-def)
from justified have C-n: ?C n ⊆ actions (?E n)
  and C-Sn: ?C (Suc n) ⊆ actions (?E (Suc n))
  and wf-tr: wf-action-translation E (J n)
  and wf-tr': wf-action-translation E (J (Suc n))
  by(auto simp add: committed-subset-actions-def wf-action-translations-def)
from C-n committed' have action': a' ∈ actions (?E n) by blast
from wf-tr committed' a'
have action-tid E a = action-tid (?E n) a' action-obs E a ≈ action-obs (?E n) a'
  by(auto simp add: wf-action-translation-on-def intro: sim-action-sym)
with read obtain v'
  where action-obs (?E n) a' = NormalAction (ReadMem ad al v')
  by(clarsimp simp add: action-obs-def)
with action' have read': a' ∈ read-actions (?E n) ..

from justified have ?φ n ‘ ?C n ⊆ ?φ (Suc n) ‘ ?C (Suc n)
  by(simp add: is-commit-sequence-def)
with committed' a' have a ∈ ... by auto
then obtain a'' where a'': a = ?φ (Suc n) a''
```

```
    and committed'': a'' ∈ ?C (Suc n) by auto
  from committed'' C-Sn have action'': a'' ∈ actions (?E (Suc n)) by blast

  with wf-tr' have a'' = inv-into (actions (?E (Suc n))) (?φ (Suc n)) a
    by(simp add: a'' wf-action-translation-on-def)
  with justified read' committed' a' have ws-a: ws a = ?φ (Suc n) (?ws (Suc n) a'')
    by(simp add: write-seen-committed-def)

  from wf-tr' committed'' a''
  have action-tid E a = action-tid (?E (Suc n)) a''
    and action-obs E a ≈ action-obs (?E (Suc n)) a''
    by(auto simp add: wf-action-translation-on-def intro: sim-action-sym)
  with read obtain v''
    where a-obs'': action-obs (?E (Suc n)) a'' = NormalAction (ReadMem ad al v'')
    by(clarsimp simp add: action-obs-def)
  with action'' have read'': a'' ∈ read-actions (?E (Suc n))
    by(auto intro: read-actions.intros simp add: action-obs-def)

  have a ∈ read-actions E action-obs E a = NormalAction (ReadMem ad al v)
    using action read by(auto intro: read-actions.intros simp add: action-obs-def read)
  from is-write-seenD[OF wf-exec-is-write-seenD[OF wf] this]
  have v-eq: v = value-written P E (ws a) (ad, al)
    and adal: (ad, al) ∈ action-loc P E (ws a) by simp-all

  from justified have P ⊢ (?E (Suc n), ?ws (Suc n)) √ by(simp add: justification-well-formed-def)
  from is-write-seenD[OF wf-exec-is-write-seenD[OF this] read'' a-obs'']
  have write'': ?ws (Suc n) a'' ∈ write-actions (?E (Suc n))
    and written'': value-written P (?E (Suc n)) (?ws (Suc n) a'') (ad, al) = v''
    by simp-all

  from justified read'' committed''
  have ws (?φ (Suc n) a'') ∈ ?φ (Suc n) ‘ ?C (Suc n)
    by(rule weakly-justified-write-seen-hb-read-committed)
  then obtain w where w: ws (?φ (Suc n) a'') = ?φ (Suc n) w
    and committed-w: w ∈ ?C (Suc n) by blast
  with C-Sn have w ∈ actions (?E (Suc n)) by blast
  moreover have ws (?φ (Suc n) a'') = ?φ (Suc n) (?ws (Suc n) a'')
    using ws-a a'' by simp
  ultimately have w-def: w = ?ws (Suc n) a''
    using wf-action-translation-on-inj-onD[OF wf-tr'] write''
    unfolding w by(auto dest: inj-onD)
  with committed-w have ?ws (Suc n) a'' ∈ ?C (Suc n) by simp
  hence value-written P E (ws a) (ad, al) = value-written P (?E (Suc n)) (?ws (Suc n) a'') (ad, al)
    using adal justified write'' by(simp add: value-written-committed-def ws-a)
  with v-eq written'' have v = v'' by simp

  from read'' have enat a'' < llength (?E (Suc n)) by(cases)(simp add: actions-def)
  thus ∃ T. P ⊢ ad@al : T ∧ P ⊢ v :≤ T
    by(rule justifying)(simp-all add: a-obs'' ‹v = v''›)
qed

corollary weakly-legal-read-value-typeable:
  assumes wfx-start: ts-ok wfx (thr (h.start-state f P C M vs)) h.start-heap
  and wfP: wf-syscls P
```

and *legal*: *weakly-legal-execution P* (*h.*$\mathcal{E}$*-start f P C M vs status*) (*E*, *ws*)
and *a*: *enat a* < *llength E*
and *read*: *action-obs E a* = *NormalAction* (*ReadMem ad al v*)
shows ∃ *T*. *P* ⊢ *ad@al* : *T* ∧ *P* ⊢ *v* :≤ *T*
proof −
from *legal* **obtain** *J*
where *P* ⊢ (*E*, *ws*) *weakly-justified-by J*
and *range* (*justifying-exec* ∘ *J*) ⊆ *h.*$\mathcal{E}$*-start f P C M vs status*
and *E* ∈ *h.*$\mathcal{E}$*-start f P C M vs status*
and *P* ⊢ (*E*, *ws*) √ **by**(*rule legal-executionE*)
with *wfx-start wfP* **show** *?thesis* **using** *a read* **by**(*rule read-value-typeable-justifed*)
qed

corollary *legal-read-value-typeable*:
⟦ *ts-ok wfx* (*thr* (*h.start-state f P C M vs*)) *h.start-heap*; *wf-syscls P*;
legal-execution P (*h.*$\mathcal{E}$*-start f P C M vs status*) (*E*, *ws*);
enat a < *llength E*; *action-obs E a* = *NormalAction* (*ReadMem ad al v*) ⟧
⟹ ∃ *T*. *P* ⊢ *ad@al* : *T* ∧ *P* ⊢ *v* :≤ *T*
by(*erule* (*1*) *weakly-legal-read-value-typeable*)(*rule legal-imp-weakly-legal-execution*)

end

end

8.13 JMM Instantiation with Jinja – common parts

theory *JMM-Common*
imports
JMM-Framework
JMM-Typesafe
../Common/BinOp
../Common/ExternalCallWF
begin

context *heap* **begin**

lemma *heap-copy-loc-not-New*: **assumes** *heap-copy-loc a a′ al h ob h′*
shows *NewHeapElem a″ x* ∈ *set ob* ⟹ *False*
using *assms*
by(*auto elim*: *heap-copy-loc.cases*)

lemma *heap-copies-not-New*:
assumes *heap-copies a a′ als h obs h′*
and *NewHeapElem a″ x* ∈ *set obs*
shows *False*
using *assms*
by *induct*(*auto dest*: *heap-copy-loc-not-New*)

lemma *heap-clone-New-same-addr-same*:
assumes *heap-clone P h a h′* ⌊(*obs*, *a′*)⌋
and *obs* ! *i* = *NewHeapElem a″ x i* < *length obs*
and *obs* ! *j* = *NewHeapElem a″ x′ j* < *length obs*
shows *i* = *j*

using *assms*
apply *cases*
apply(*fastforce simp add*: *nth-Cons' gr0-conv-Suc in-set-conv-nth split*: *if-split-asm dest*: *heap-copies-not-New*)+
done

lemma *red-external-New-same-addr-same*:
  ⟦ *P*,*t* ⊢ ⟨*a*·*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h'*⟩;
    {|*ta*|}$_o$! *i* = *NewHeapElem a' x*; *i* < *length* {|*ta*|}$_o$;
    {|*ta*|}$_o$! *j* = *NewHeapElem a' x'*; *j* < *length* {|*ta*|}$_o$ ⟧
  ⟹ *i* = *j*
by(*auto elim*!: *red-external.cases simp add*: *nth-Cons' split*: *if-split-asm dest*: *heap-clone-New-same-addr-same*)

lemma *red-external-aggr-New-same-addr-same*:
  ⟦ (*ta*, *va*, *h'*) ∈ *red-external-aggr P t a M vs h*;
    {|*ta*|}$_o$! *i* = *NewHeapElem a' x*; *i* < *length* {|*ta*|}$_o$;
    {|*ta*|}$_o$! *j* = *NewHeapElem a' x'*; *j* < *length* {|*ta*|}$_o$ ⟧
  ⟹ *i* = *j*
by(*auto simp add*: *external-WT-defs.simps red-external-aggr-def nth-Cons' split*: *if-split-asm if-split-asm dest*: *heap-clone-New-same-addr-same*)

lemma *heap-copy-loc-read-typeable*:
  assumes *heap-copy-loc a a' al h obs h'*
  and *ReadMem ad al' v* ∈ *set obs*
  and *P*,*h* ⊢ *a*@*al* : *T*
  shows *ad* = *a* ∧ *al'*= *al*
using *assms* **by** *cases auto*

lemma *heap-copies-read-typeable*:
  assumes *heap-copies a a' als h obs h'*
  and *ReadMem ad al' v* ∈ *set obs*
  and *list-all2* (λ*al T*. *P*,*h* ⊢ *a*@*al* : *T*) *als Ts*
  shows *ad* = *a* ∧ *al'* ∈ *set als*
using *assms*
proof(*induct arbitrary*: *Ts*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons al h ob h' als obs h''*)
  from ‹*list-all2* (λ*al T*. *P*,*h* ⊢ *a*@*al* : *T*) (*al* # *als*) *Ts*›
  obtain *T Ts'* **where** *Ts* [*simp*]: *Ts* = *T* # *Ts'*
    and *P*,*h* ⊢ *a*@*al* : *T*
    and *list-all2* (λ*al T*. *P*,*h* ⊢ *a*@*al* : *T*) *als Ts'*
    by(*auto simp add*: *list-all2-Cons1*)
  from ‹*ReadMem ad al' v* ∈ *set* (*ob* @ *obs*)›
  show *?case* **unfolding** *set-append Un-iff*
  proof
    assume *ReadMem ad al' v* ∈ *set ob*
    with ‹*heap-copy-loc a a' al h ob h'*›
    have *ad* = *a* ∧ *al'*= *al* **using** ‹*P*,*h* ⊢ *a*@*al* : *T*›
      by(*rule heap-copy-loc-read-typeable*)
    thus *?thesis* **by** *simp*
  next
    assume *ReadMem ad al' v* ∈ *set obs*
    moreover from ‹*heap-copy-loc a a' al h ob h'*›
    have *h* ◁ *h'* **by**(*rule hext-heap-copy-loc*)

```
    from ‹list-all2 (λal T. P,h ⊢ a@al : T) als Ts'›
    have list-all2 (λal T. P,h' ⊢ a@al : T) als Ts'
      by(rule List.list-all2-mono)(rule addr-loc-type-hext-mono[OF - ‹h ⊴ h'›])
    ultimately have ad = a ∧ al' ∈ set als by(rule Cons)
    thus ?thesis by simp
  qed
qed

lemma heap-clone-read-typeable:
  assumes clone: heap-clone P h a h' ⌊(obs, a')⌋
  and read: ReadMem ad al v ∈ set obs
  shows ad = a ∧ (∃ T'. P,h ⊢ ad@al : T')
using clone
proof cases
  case (ObjClone C H' FDTs obs')
  let ?als = map (λ((F, D), Tm). CField D F) FDTs
  let ?Ts = map (λ(FD, T). fst (the (map-of FDTs FD))) FDTs
  note ‹heap-copies a a' ?als H' obs' h'›
  moreover
  from ‹obs = NewHeapElem a' (Class-type C) # obs'› read
  have ReadMem ad al v ∈ set obs' by simp
  moreover
  from ‹(H', a') ∈ allocate h (Class-type C)› have h ⊴ H' by(rule hext-allocate)
  hence typeof-addr H' a = ⌊Class-type C⌋ using ‹typeof-addr h a = ⌊Class-type C⌋›
    by(rule typeof-addr-hext-mono)
  hence type: list-all2 (λal T. P,H' ⊢ a@al : T) ?als ?Ts
    using ‹P ⊢ C has-fields FDTs›
    unfolding list-all2-map1 list-all2-map2 list-all2-same
    by(fastforce intro: addr-loc-type.intros simp add: has-field-def dest: weak-map-of-SomeI)
  ultimately have ad = a ∧ al ∈ set ?als by(rule heap-copies-read-typeable)
  hence [simp]: ad = a and al ∈ set ?als by simp-all
  then obtain F D T where [simp]: al = CField D F and ((F, D), T) ∈ set FDTs by auto
  with type ‹h ⊴ H'› ‹typeof-addr h a = ⌊Class-type C⌋› show ?thesis
    unfolding list-all2-map1 list-all2-map2 list-all2-same
    by(fastforce elim!: ballE[where x=((F, D), T)] addr-loc-type.cases dest: typeof-addr-hext-mono
intro: addr-loc-type.intros)
next
  case (ArrClone T n H' FDTs obs')
  let ?als = map (λ((F, D), Tfm). CField D F) FDTs @ map ACell [0..<n]
  let ?Ts = map (λ(FD, T). fst (the (map-of FDTs FD))) FDTs @ replicate n T
  note FDTs = ‹P ⊢ Object has-fields FDTs›
  note ‹heap-copies a a' ?als H' obs' h'›
  moreover from ‹obs = NewHeapElem a' (Array-type T n) # obs'› read
  have ReadMem ad al v ∈ set obs' by simp
  moreover from ‹(H', a') ∈ allocate h (Array-type T n)›
  have h ⊴ H' by(rule hext-allocate)
  with ‹typeof-addr h a = ⌊Array-type T n⌋›
  have type': typeof-addr H' a = ⌊Array-type T n⌋
    by(auto dest: typeof-addr-hext-mono hext-arrD)
  hence type: list-all2 (λal T. P,H' ⊢ a@al : T) ?als ?Ts using FDTs
    by(fastforce intro: list-all2-all-nthI addr-loc-type.intros simp add: has-field-def list-all2-append
list-all2-map1 list-all2-map2 list-all2-same dest: weak-map-of-SomeI)
  ultimately have ad = a ∧ al ∈ set ?als by(rule heap-copies-read-typeable)
  hence [simp]: ad = a and al ∈ set ?als by simp-all
```

hence *al* ∈ *set* (*map* (λ((*F*, *D*), *Tfm*). *CField D F*) *FDTs*) ∨ *al* ∈ *set* (*map ACell* [*0*..<*n*]) **by** *simp*
thus *?thesis*
proof
assume *al* ∈ *set* (*map* (λ((*F*, *D*), *Tfm*). *CField D F*) *FDTs*)
then obtain *F D Tfm* **where** [*simp*]: *al* = *CField D F* **and** ((*F*, *D*), *Tfm*) ∈ *set FDTs* **by** *auto*
with *type type'* ‹*h* ⊴ *H'*› ‹*typeof-addr h a* = ⌊*Array-type T n*⌋› **show** *?thesis*
by(*fastforce elim*!: *ballE*[**where** *x*=((*F*, *D*), *Tfm*)] *addr-loc-type.cases intro*: *addr-loc-type.intros simp add*: *list-all2-append list-all2-map1 list-all2-map2 list-all2-same*)
next
assume *al* ∈ *set* (*map ACell* [*0*..<*n*])
then obtain *n'* **where** [*simp*]: *al* = *ACell n'* **and** *n'* < *n* **by** *auto*
with *type type'* ‹*h* ⊴ *H'*› ‹*typeof-addr h a* = ⌊*Array-type T n*⌋›
show *?thesis* **by**(*fastforce dest*: *list-all2-nthD*[**where** *p*=*n'*] *elim*: *addr-loc-type.cases intro*: *addr-loc-type.intros*)
qed
qed

lemma *red-external-read-mem-typeable*:
assumes *red*: *P*,*t* ⊢ ⟨*a*·*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h'*⟩
and *read*: *ReadMem ad al v* ∈ *set* {|*ta*|}*o*
shows ∃ *T'*. *P*,*h* ⊢ *ad*@*al* : *T'*
using *red read*
by *cases*(*fastforce dest*: *heap-clone-read-typeable intro*: *addr-loc-type.intros*)+

end

context *heap-conf* **begin**

lemma *heap-clone-typeof-addrD*:
assumes *heap-clone P h a h'* ⌊(*obs*, *a'*)⌋
and *hconf h*
shows *NewHeapElem a'' x* ∈ *set obs* ⟹ *a''* = *a'* ∧ *typeof-addr h' a'* = *Some x*
using *assms*
by(*fastforce elim*!: *heap-clone.cases dest*: *allocate-SomeD hext-heap-copies heap-copies-not-New typeof-addr-is-type elim*: *hext-objD hext-arrD*)

lemma *red-external-New-typeof-addrD*:
⟦ *P*,*t* ⊢ ⟨*a*·*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h'*⟩; *NewHeapElem a' x* ∈ *set* {|*ta*|}*o*; *hconf h* ⟧
⟹ *typeof-addr h' a'* = *Some x*
by(*erule red-external.cases*)(*auto dest*: *heap-clone-typeof-addrD*)

lemma *red-external-aggr-New-typeof-addrD*:
⟦ (*ta*, *va*, *h'*) ∈ *red-external-aggr P t a M vs h*; *NewHeapElem a' x* ∈ *set* {|*ta*|}*o*;
is-native P (*the* (*typeof-addr h a*)) *M*; *hconf h* ⟧
⟹ *typeof-addr h' a'* = *Some x*
apply(*auto simp add*: *is-native.simps external-WT-defs.simps red-external-aggr-def split*: *if-split-asm*)
apply(*blast dest*: *heap-clone-typeof-addrD*)+
done

end

context *heap-conf* **begin**

lemma *heap-copy-loc-non-speculative-typeable*:
assumes *copy*: *heap-copy-loc ad ad' al h obs h'*

```
  and sc: non-speculative P vs (llist-of (map NormalAction obs))
  and vs: vs-conf P h vs
  and hconf: hconf h
  and wt: P,h ⊢ ad@al : T P,h ⊢ ad'@al : T
  shows heap-base.heap-copy-loc (heap-read-typed P) heap-write ad ad' al h obs h'
proof −
  from copy obtain v where obs: obs = [ReadMem ad al v, WriteMem ad' al v]
    and read: heap-read h ad al v and write: heap-write h ad' al v h' by cases
  from obs sc have v ∈ vs (ad, al) by auto
  with vs wt have v: P,h ⊢ v :≤ T by(blast dest: vs-confD addr-loc-type-fun)+
  with read wt have heap-read-typed P h ad al v
    by(auto intro: heap-read-typedI dest: addr-loc-type-fun)
  thus ?thesis using write unfolding obs by(rule heap-base.heap-copy-loc.intros)
qed

lemma heap-copy-loc-non-speculative-vs-conf:
  assumes copy: heap-copy-loc ad ad' al h obs h'
  and sc: non-speculative P vs (llist-of (take n (map NormalAction obs)))
  and vs: vs-conf P h vs
  and hconf: hconf h
  and wt: P,h ⊢ ad@al : T P,h ⊢ ad'@al : T
  shows vs-conf P h' (w-values P vs (take n (map NormalAction obs)))
proof −
  from copy obtain v where obs: obs = [ReadMem ad al v, WriteMem ad' al v]
    and read: heap-read h ad al v and write: heap-write h ad' al v h' by cases
  from write have hext: h ⊴ h' by(rule hext-heap-write)
  with vs have vs': vs-conf P h' vs by(rule vs-conf-hext)
  show ?thesis
  proof(cases n > 0)
    case True
    with obs sc have v ∈ vs (ad, al) by(auto simp add: take-Cons')
    with vs wt have v: P,h ⊢ v :≤ T by(blast dest: vs-confD addr-loc-type-fun)+
    with hext wt have P,h' ⊢ ad'@al : T P,h' ⊢ v :≤ T
      by(blast intro: addr-loc-type-hext-mono conf-hext)+
    thus ?thesis using vs' obs
      by(auto simp add: take-Cons' intro!: vs-confI split: if-split-asm dest: vs-confD)
  next
    case False thus ?thesis using vs' by simp
  qed
qed

lemma heap-copies-non-speculative-typeable:
  assumes heap-copies ad ad' als h obs h'
  and non-speculative P vs (llist-of (map NormalAction obs))
  and vs-conf P h vs
  and hconf h
  and list-all2 (λal T. P,h ⊢ ad@al : T) als Ts list-all2 (λal T. P,h ⊢ ad'@al : T) als Ts
  shows heap-base.heap-copies (heap-read-typed P) heap-write ad ad' als h obs h'
using assms
proof(induct arbitrary: Ts vs)
  case Nil show ?case by(auto intro: heap-base.heap-copies.intros)
next
  case (Cons al h ob h' als obs h'')
  note sc = ‹non-speculative P vs (llist-of (map NormalAction (ob @ obs)))›
```

```
    and vs = ‹vs-conf P h vs›
    and hconf = ‹hconf h›
    and wt = ‹list-all2 (λal T. P,h ⊢ ad@al : T) (al # als) Ts› ‹list-all2 (λal T. P,h ⊢ ad'@al : T)
(al # als) Ts›

  have sc1: non-speculative P vs (llist-of (map NormalAction ob))
    and sc2: non-speculative P (w-values P vs (map NormalAction ob)) (llist-of (map NormalAction
obs))
     using sc by(simp-all add: non-speculative-lappend lappend-llist-of-llist-of[symmetric] del: lap-
pend-llist-of-llist-of)
  from wt obtain T Ts' where Ts: Ts = T # Ts'
    and wt1: P,h ⊢ ad@al : T P,h ⊢ ad'@al : T
    and wt2: list-all2 (λal T. P,h ⊢ ad@al : T) als Ts' list-all2 (λal T. P,h ⊢ ad'@al : T) als Ts'
    by(auto simp add: list-all2-Cons1)
  from ‹heap-copy-loc ad ad' al h ob h'› sc1 vs hconf wt1
  have copy: heap-base.heap-copy-loc (heap-read-typed P) heap-write ad ad' al h ob h'
    by(rule heap-copy-loc-non-speculative-typeable)+
  from heap-copy-loc-non-speculative-vs-conf[OF ‹heap-copy-loc ad ad' al h ob h'› - vs hconf wt1, of
length ob] sc1
  have vs': vs-conf P h' (w-values P vs (map NormalAction ob)) by simp

  from ‹heap-copy-loc ad ad' al h ob h'›
  have h ≤ h' by(rule hext-heap-copy-loc)
  with wt2 have wt2': list-all2 (λal T. P,h' ⊢ ad@al : T) als Ts' list-all2 (λal T. P,h' ⊢ ad'@al :
T) als Ts'
    by −(erule List.list-all2-mono[OF - addr-loc-type-hext-mono], assumption+)+

  from copy hconf wt1 have hconf': hconf h'
    by(rule heap-conf-read.hconf-heap-copy-loc-mono[OF heap-conf-read-heap-read-typed])

   from sc2 vs' hconf' wt2' have heap-base.heap-copies (heap-read-typed P) heap-write ad ad' als h'
obs h'' by(rule Cons)
  with copy show ?case by(rule heap-base.heap-copies.Cons)
qed

lemma heap-copies-non-speculative-vs-conf:
  assumes heap-copies ad ad' als h obs h'
  and non-speculative P vs (llist-of (take n (map NormalAction obs)))
  and vs-conf P h vs
  and hconf h
  and list-all2 (λal T. P,h ⊢ ad@al : T) als Ts list-all2 (λal T. P,h ⊢ ad'@al : T) als Ts
  shows vs-conf P h' (w-values P vs (take n (map NormalAction obs)))
using assms
proof(induction arbitrary: Ts vs n)
  case Nil thus ?case by simp
next
  case (Cons al h ob h' als obs h'')
  note sc = ‹non-speculative P vs (llist-of (take n (map NormalAction (ob @ obs))))›
    and hcl = ‹heap-copy-loc ad ad' al h ob h'›
    and vs = ‹vs-conf P h vs›
    and hconf = ‹hconf h›
    and wt = ‹list-all2 (λal T. P,h ⊢ ad@al : T) (al # als) Ts› ‹list-all2 (λal T. P,h ⊢ ad'@al : T)
(al # als) Ts›
  let ?vs' = w-values P vs (take n (map NormalAction ob))
```

```
  from sc have sc1: non-speculative P vs (llist-of (take n (map NormalAction ob)))
    and sc2: non-speculative P ?vs' (llist-of (take (n − length ob) (map NormalAction obs)))
   by(simp-all add: lappend-llist-of-llist-of[symmetric] non-speculative-lappend del: lappend-llist-of-llist-of)

  from wt obtain T Ts' where Ts: Ts = T # Ts'
    and wt1: P,h ⊢ ad@al : T P,h ⊢ ad'@al : T
    and wt2: list-all2 (λal T. P,h ⊢ ad@al : T) als Ts' list-all2 (λal T. P,h ⊢ ad'@al : T) als Ts'
    by(auto simp add: list-all2-Cons1)

  from hcl sc1 vs hconf wt1 have vs': vs-conf P h' ?vs' by(rule heap-copy-loc-non-speculative-vs-conf)

  show ?case
  proof(cases n < length ob)
    case True
    from ‹heap-copies ad ad' als h' obs h''› have h' ≤ h'' by(rule hext-heap-copies)
    with vs' have vs-conf P h'' ?vs' by(rule vs-conf-hext)
    thus ?thesis using True by simp
  next
    case False
    note sc2 vs'
    moreover from False sc1 have sc1': non-speculative P vs (llist-of (map NormalAction ob)) by
simp
    with hcl have heap-base.heap-copy-loc (heap-read-typed P) heap-write ad ad' al h ob h'
      using vs hconf wt1 by(rule heap-copy-loc-non-speculative-typeable)
    hence hconf h' using hconf wt1
      by(rule heap-conf-read.hconf-heap-copy-loc-mono[OF heap-conf-read-heap-read-typed])
    moreover
    from hcl have h ≤ h' by(rule hext-heap-copy-loc)
    with wt2 have wt2': list-all2 (λal T. P,h' ⊢ ad@al : T) als Ts' list-all2 (λal T. P,h' ⊢ ad'@al :
T) als Ts'
      by −(erule List.list-all2-mono[OF - addr-loc-type-hext-mono], assumption+)+
    ultimately have vs-conf P h'' (w-values P ?vs' (take (n − length ob) (map NormalAction obs)))
      by(rule Cons.IH)
    with False show ?thesis by simp
  qed
qed

lemma heap-clone-non-speculative-typeable-Some:
  assumes clone: heap-clone P h ad h' ⌊(obs, ad')⌋
  and sc: non-speculative P vs (llist-of (map NormalAction obs))
  and vs: vs-conf P h vs
  and hconf: hconf h
 shows heap-base.heap-clone allocate typeof-addr (heap-read-typed P) heap-write P h ad h' ⌊(obs, ad')⌋
using clone
proof(cases)
  case (ObjClone C h'' FDTs obs')
  note FDTs = ‹P ⊢ C has-fields FDTs›
    and obs = ‹obs = NewHeapElem ad' (Class-type C) # obs'›
  let ?als = map (λ((F, D), Tfm). CField D F) FDTs
  let ?Ts = map (λ(FD, T). fst (the (map-of FDTs FD))) FDTs
   let ?vs = w-value P vs (NormalAction (NewHeapElem ad' (Class-type C) :: ('addr, 'thread-id)
obs-event))
  from ‹(h'', ad') ∈ allocate h (Class-type C)› have hext: h ≤ h'' by(rule hext-heap-ops)
```

hence *type*: *typeof-addr h'' ad* = ⌊*Class-type C*⌋ **using** ‹*typeof-addr h ad* = ⌊*Class-type C*⌋›
by(*rule typeof-addr-hext-mono*)

note ‹*heap-copies ad ad' ?als h'' obs' h'*›
moreover from *sc* **have** *non-speculative P ?vs* (*llist-of* (*map NormalAction obs'*))
by(*simp add*: *obs*)
moreover from ‹*P* ⊢ *C has-fields FDTs*›
have *is-class P C* **by**(*rule has-fields-is-class*)
hence *is-htype P* (*Class-type C*) **by** *simp*
with *vs* ‹(*h''*, *ad'*) ∈ *allocate h* (*Class-type C*)›
have *vs-conf P h'' ?vs* **by**(*rule vs-conf-allocate*)
moreover from ‹(*h''*, *ad'*) ∈ *allocate h* (*Class-type C*)› *hconf* ‹*is-htype P* (*Class-type C*)›
have *hconf h''* **by**(*rule hconf-allocate-mono*)
moreover from *type FDTs* **have** *list-all2* (λ*al T*. *P*,*h''* ⊢ *ad*@*al* : *T*) *?als ?Ts*
unfolding *list-all2-map1 list-all2-map2 list-all2-same*
by(*fastforce intro*: *addr-loc-type.intros simp add*: *has-field-def dest*: *weak-map-of-SomeI*)
moreover from ‹(*h''*, *ad'*) ∈ *allocate h* (*Class-type C*)› ‹*is-htype P* (*Class-type C*)›
have *typeof-addr h'' ad'* = ⌊*Class-type C*⌋ **by**(*auto dest*: *allocate-SomeD*)
with *FDTs* **have** *list-all2* (λ*al T*. *P*,*h''* ⊢ *ad'*@*al* : *T*) *?als ?Ts*
unfolding *list-all2-map1 list-all2-map2 list-all2-same*
by(*fastforce intro*: *addr-loc-type.intros simp add*: *has-field-def dest*: *weak-map-of-SomeI*)
ultimately
have *copy*: *heap-base.heap-copies* (*heap-read-typed P*) *heap-write ad ad'* (*map* (λ((*F*, *D*), *Tfm*). *CField D F*) *FDTs*) *h'' obs' h'*
by(*rule heap-copies-non-speculative-typeable*)+
from ‹*typeof-addr h ad* = ⌊*Class-type C*⌋› ‹(*h''*, *ad'*) ∈ *allocate h* (*Class-type C*)› *FDTs copy*
show *?thesis* **unfolding** *obs* **by**(*rule heap-base.heap-clone.intros*)
next
case (*ArrClone T n h'' FDTs obs'*)
note *obs* = ‹*obs* = *NewHeapElem ad'* (*Array-type T n*) # *obs'*›
and *new* = ‹(*h''*, *ad'*) ∈ *allocate h* (*Array-type T n*)›
and *FDTs* = ‹*P* ⊢ *Object has-fields FDTs*›
let *?als* = *map* (λ((*F*, *D*), *Tfm*). *CField D F*) *FDTs* @ *map ACell* [*0*..<*n*]
let *?Ts* = *map* (λ(*FD*, *T*). *fst* (*the* (*map-of FDTs FD*))) *FDTs* @ *replicate n T*
let *?vs* = *w-value P vs* (*NormalAction* (*NewHeapElem ad'* (*Array-type T n*) :: ('*addr*, '*thread-id*) *obs-event*))
from *new* **have** *hext*: *h* ⊴ *h''* **by**(*rule hext-heap-ops*)
hence *type*: *typeof-addr h'' ad* = ⌊*Array-type T n*⌋ **using** ‹*typeof-addr h ad* = ⌊*Array-type T n*⌋›
by(*rule typeof-addr-hext-mono*)

note ‹*heap-copies ad ad' ?als h'' obs' h'*›
moreover from *sc* **have** *non-speculative P ?vs* (*llist-of* (*map NormalAction obs'*)) **by**(*simp add*: *obs*)
moreover from ‹*typeof-addr h ad* = ⌊*Array-type T n*⌋› ‹*hconf h*› **have** *is-htype P* (*Array-type T n*)
by(*auto dest*: *typeof-addr-is-type*)
with *vs new* **have** *vs-conf P h'' ?vs* **by**(*rule vs-conf-allocate*)
moreover from *new hconf* ‹*is-htype P* (*Array-type T n*)› **have** *hconf h''* **by**(*rule hconf-allocate-mono*)
moreover
from *type FDTs* **have** *list-all2* (λ*al T*. *P*,*h''* ⊢ *ad*@*al* : *T*) *?als ?Ts*
by(*fastforce intro*: *list-all2-all-nthI addr-loc-type.intros simp add*: *has-field-def list-all2-append list-all2-map1 list-all2-map2 list-all2-same dest*: *weak-map-of-SomeI*)
moreover from *new* ‹*is-htype P* (*Array-type T n*)›
have *typeof-addr h'' ad'* = ⌊*Array-type T n*⌋
by(*auto dest*: *allocate-SomeD*)

hence *list-all2* (λ*al T*. *P*,*h''* ⊢ *ad'*@*al* : *T*) *?als ?Ts* **using** *FDTs*
by(*fastforce intro*: *list-all2-all-nthI addr-loc-type.intros simp add*: *has-field-def list-all2-append list-all2-map1 list-all2-map2 list-all2-same dest*: *weak-map-of-SomeI*)
ultimately have *copy*: *heap-base.heap-copies* (*heap-read-typed P*) *heap-write ad ad'* (*map* (λ((*F*, *D*), *Tfm*). *CField D F*) *FDTs* @ *map ACell* [*0*..<*n*]) *h''* *obs'* *h'*
by(*rule heap-copies-non-speculative-typeable*)+
from ‹*typeof-addr h ad* = ⌊*Array-type T n*⌋› *new FDTs copy* **show** *?thesis*
unfolding *obs* **by**(*rule heap-base.heap-clone.ArrClone*)
qed

lemma *heap-clone-non-speculative-vs-conf-Some*:
assumes *clone*: *heap-clone P h ad h'* ⌊(*obs*, *ad'*)⌋
and *sc*: *non-speculative P vs* (*llist-of* (*take n* (*map NormalAction obs*)))
and *vs*: *vs-conf P h vs*
and *hconf*: *hconf h*
shows *vs-conf P h'* (*w-values P vs* (*take n* (*map NormalAction obs*)))
using *clone*
proof(*cases*)
case (*ObjClone C h'' FDTs obs'*)
note *FDTs* = ‹*P* ⊢ *C has-fields FDTs*›
and *obs* = ‹*obs* = *NewHeapElem ad'* (*Class-type C*) # *obs'*›
let *?als* = *map* (λ((*F*, *D*), *Tfm*). *CField D F*) *FDTs*
let *?Ts* = *map* (λ(*FD*, *T*). *fst* (*the* (*map-of FDTs FD*))) *FDTs*
let *?vs* = *w-value P vs* (*NormalAction* (*NewHeapElem ad'* (*Class-type C*) :: ('*addr*, '*thread-id*) *obs-event*))
from ‹(*h''*, *ad'*) ∈ *allocate h* (*Class-type C*)› **have** *hext*: *h* ⊴ *h''* **by**(*rule hext-heap-ops*)
hence *type*: *typeof-addr h'' ad* = ⌊*Class-type C*⌋ **using** ‹*typeof-addr h ad* = ⌊*Class-type C*⌋›
by(*rule typeof-addr-hext-mono*)

note ‹*heap-copies ad ad' ?als h'' obs' h'*›
moreover from *sc* **have** *non-speculative P ?vs* (*llist-of* (*take* (*n* − *1*) (*map NormalAction obs'*)))
by(*simp add*: *obs take-Cons' split*: *if-split-asm*)
moreover from ‹*P* ⊢ *C has-fields FDTs*›
have *is-class P C* **by**(*rule has-fields-is-class*)
hence *is-htype P* (*Class-type C*) **by** *simp*
with *vs* ‹(*h''*, *ad'*) ∈ *allocate h* (*Class-type C*)›
have *vs-conf P h'' ?vs* **by**(*rule vs-conf-allocate*)
moreover from ‹(*h''*, *ad'*) ∈ *allocate h* (*Class-type C*)› *hconf* ‹*is-htype P* (*Class-type C*)›
have *hconf h''* **by**(*rule hconf-allocate-mono*)
moreover from *type FDTs* **have** *list-all2* (λ*al T*. *P*,*h''* ⊢ *ad*@*al* : *T*) *?als ?Ts*
unfolding *list-all2-map1 list-all2-map2 list-all2-same*
by(*fastforce intro*: *addr-loc-type.intros simp add*: *has-field-def dest*: *weak-map-of-SomeI*)
moreover from ‹(*h''*, *ad'*) ∈ *allocate h* (*Class-type C*)› ‹*is-htype P* (*Class-type C*)›
have *typeof-addr h'' ad'* = ⌊*Class-type C*⌋ **by**(*auto dest*: *allocate-SomeD*)
with *FDTs* **have** *list-all2* (λ*al T*. *P*,*h''* ⊢ *ad'*@*al* : *T*) *?als ?Ts*
unfolding *list-all2-map1 list-all2-map2 list-all2-same*
by(*fastforce intro*: *addr-loc-type.intros simp add*: *has-field-def dest*: *weak-map-of-SomeI*)
ultimately
have *vs'*: *vs-conf P h'* (*w-values P ?vs* (*take* (*n* − *1*) (*map NormalAction obs'*)))
by(*rule heap-copies-non-speculative-vs-conf*)
show *?thesis*
proof(*cases n* > *0*)
case *True*
with *obs vs'* **show** *?thesis* **by**(*simp add*: *take-Cons'*)

```
  next
    case False
    from ‹heap-copies ad ad' ?als h'' obs' h'› have h'' ⊴ h' by(rule hext-heap-copies)
    with ‹h ⊴ h''› have h ⊴ h' by(rule hext-trans)
    with vs have vs-conf P h' vs by(rule vs-conf-hext)
    thus ?thesis using False by simp
  qed
next
  case (ArrClone T N h'' FDTs obs')
  note obs = ‹obs = NewHeapElem ad' (Array-type T N) # obs'›
    and new = ‹(h'', ad') ∈ allocate h (Array-type T N)›
    and FDTs = ‹P ⊢ Object has-fields FDTs›
  let ?als = map (λ((F, D), Tfm). CField D F) FDTs @ map ACell [0..<N]
  let ?Ts = map (λ(FD, T). fst (the (map-of FDTs FD))) FDTs @ replicate N T
  let ?vs = w-value P vs (NormalAction (NewHeapElem ad' (Array-type T N)) :: ('addr, 'thread-id)
obs-event))
  from new have hext: h ⊴ h'' by(rule hext-heap-ops)
  hence type: typeof-addr h'' ad = ⌊Array-type T N⌋ using ‹typeof-addr h ad = ⌊Array-type T N⌋›
    by(rule typeof-addr-hext-mono)

  note ‹heap-copies ad ad' ?als h'' obs' h'›
  moreover from sc have non-speculative P ?vs (llist-of (take (n − 1) (map NormalAction obs')))
    by(simp add: obs take-Cons' split: if-split-asm)
  moreover from ‹typeof-addr h ad = ⌊Array-type T N⌋› ‹hconf h› have is-htype P (Array-type T
N)
    by(auto dest: typeof-addr-is-type)
  with vs new have vs-conf P h'' ?vs by(rule vs-conf-allocate)
  moreover from new hconf ‹is-htype P (Array-type T N)› have hconf h'' by(rule hconf-allocate-mono)
  moreover
  from type FDTs have list-all2 (λal T. P,h'' ⊢ ad@al : T) ?als ?Ts
      by(fastforce intro: list-all2-all-nthI addr-loc-type.intros simp add: has-field-def list-all2-append
list-all2-map1 list-all2-map2 list-all2-same dest: weak-map-of-SomeI)
  moreover from new ‹is-htype P (Array-type T N)›
  have typeof-addr h'' ad' = ⌊Array-type T N⌋
    by(auto dest: allocate-SomeD)
  hence list-all2 (λal T. P,h'' ⊢ ad'@al : T) ?als ?Ts using FDTs
      by(fastforce intro: list-all2-all-nthI addr-loc-type.intros simp add: has-field-def list-all2-append
list-all2-map1 list-all2-map2 list-all2-same dest: weak-map-of-SomeI)
  ultimately have vs': vs-conf P h' (w-values P ?vs (take (n − 1) (map NormalAction obs')))
    by(rule heap-copies-non-speculative-vs-conf)
  show ?thesis
  proof(cases n > 0)
    case True
    with obs vs' show ?thesis by(simp add: take-Cons')
  next
    case False
    from ‹heap-copies ad ad' ?als h'' obs' h'› have h'' ⊴ h' by(rule hext-heap-copies)
    with ‹h ⊴ h''› have h ⊴ h' by(rule hext-trans)
    with vs have vs-conf P h' vs by(rule vs-conf-hext)
    thus ?thesis using False by simp
  qed
qed

lemma heap-clone-non-speculative-typeable-None:
```

assumes *heap-clone P h ad h′ None*
shows *heap-base.heap-clone allocate typeof-addr* (*heap-read-typed P*) *heap-write P h ad h′ None*
using *assms*
by(*cases*)(*blast intro*: *heap-base.heap-clone.intros*)+

lemma *red-external-non-speculative-typeable*:
assumes *red*: *P*,*t* ⊢ ⟨*a*·*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h′*⟩
and *sc*: *non-speculative P Vs* (*llist-of* (*map NormalAction* {|*ta*|}$_o$))
and *vs*: *vs-conf P h Vs*
and *hconf*: *hconf h*
shows *heap-base.red-external addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr* (*heap-read-typed P*) *heap-write P t h a M vs ta va h′*
using *assms*
by(*cases*)(*auto intro*: *heap-base.red-external.intros heap-clone-non-speculative-typeable-None heap-clone-non-specula dest*: *hext-heap-clone elim*: *vs-conf-hext*)

lemma *red-external-non-speculative-vs-conf*:
assumes *red*: *P*,*t* ⊢ ⟨*a*·*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h′*⟩
and *sc*: *non-speculative P Vs* (*llist-of* (*take n* (*map NormalAction* {|*ta*|}$_o$)))
and *vs*: *vs-conf P h Vs*
and *hconf*: *hconf h*
shows *vs-conf P h′* (*w-values P Vs* (*take n* (*map NormalAction* {|*ta*|}$_o$)))
using *assms*
by(*cases*)(*auto intro*: *heap-base.red-external.intros heap-clone-non-speculative-vs-conf-Some dest*: *hext-heap-clone elim*: *vs-conf-hext simp add*: *take-Cons′*)

lemma *red-external-aggr-non-speculative-typeable*:
assumes *red*: (*ta*, *va*, *h′*) ∈ *red-external-aggr P t a M vs h*
and *sc*: *non-speculative P Vs* (*llist-of* (*map NormalAction* {|*ta*|}$_o$))
and *vs*: *vs-conf P h Vs*
and *hconf*: *hconf h*
and *native*: *is-native P* (*the* (*typeof-addr h a*)) *M*
shows (*ta*, *va*, *h′*) ∈ *heap-base.red-external-aggr addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr* (*heap-read-typed P*) *heap-write P t a M vs h*
using *assms*
by(*cases the* (*typeof-addr h a*))(*auto 4 3 simp add*: *is-native.simps external-WT-defs.simps red-external-aggr-def heap-base.red-external-aggr-def split*: *if-split-asm split del*: *if-split del*: *disjCI intro*: *heap-clone-non-speculative-typeab heap-clone-non-speculative-typeable-Some dest*: *sees-method-decl-above*)

lemma *red-external-aggr-non-speculative-vs-conf*:
assumes *red*: (*ta*, *va*, *h′*) ∈ *red-external-aggr P t a M vs h*
and *sc*: *non-speculative P Vs* (*llist-of* (*take n* (*map NormalAction* {|*ta*|}$_o$)))
and *vs*: *vs-conf P h Vs*
and *hconf*: *hconf h*
and *native*: *is-native P* (*the* (*typeof-addr h a*)) *M*
shows *vs-conf P h′* (*w-values P Vs* (*take n* (*map NormalAction* {|*ta*|}$_o$)))
using *assms*
by(*cases the* (*typeof-addr h a*))(*auto 4 3 simp add*: *is-native.simps external-WT-defs.simps red-external-aggr-def heap-base.red-external-aggr-def take-Cons′ split*: *if-split-asm split del*: *if-split del*: *disjCI intro*: *heap-clone-non-specul dest*: *hext-heap-clone elim*: *vs-conf-hext dest*: *sees-method-decl-above*)

end

declare *split-paired-Ex* [*simp del*]

declare *eq-upto-seq-inconsist-simps* [*simp*]

context *heap-progress* **begin**

lemma *heap-copy-loc-non-speculative-read*:
  assumes *hrt*: *heap-read-typeable hconf P*
  and *vs*: *vs-conf P h vs*
  and *type*: *P,h* ⊢ *a@al* : *T P,h* ⊢ *a′@al* : *T*
  and *hconf*: *hconf h*
  and *copy*: *heap-copy-loc a a′ al h obs h′*
  and *i*: *i* < *length obs*
  and *read*: *obs* ! *i* = *ReadMem a″ al″ v*
  and *v*: *v′* ∈ *w-values P vs* (*map NormalAction* (*take i obs*)) (*a″*, *al″*)
  shows ∃ *obs′ h″*. *heap-copy-loc a a′ al h obs′ h″* ∧ *i* < *length obs′* ∧ *take i obs′* = *take i obs* ∧
                obs′ ! *i* = *ReadMem a″ al″ v′* ∧ *length obs′* ≤ *length obs* ∧
                non-speculative P vs (*llist-of* (*map NormalAction obs′*))
using *copy*
proof *cases*
  case (*1 v″*)
  with *read i* **have** [*simp*]: *i* = *0 v″* = *v a″* = *a al″* = *al*
    by(*simp-all add*: *nth-Cons split*: *nat.split-asm*)
  from *v* **have** *v′* ∈ *vs* (*a*, *al*) **by** *simp*
  with *vs type* **have** *conf*: *P,h* ⊢ *v′* :≤ *T* **by**(*auto dest*: *addr-loc-type-fun vs-confD*)
  let *?obs″* = [*ReadMem a al v′*, *WriteMem a′ al v″*]
  from *hrt type*(*1*) *conf hconf* **have** *heap-read h a al v′* **by**(*rule heap-read-typeableD*)
  moreover from *heap-write-total*[*OF hconf type*(*2*) *conf*]
  obtain *h″* **where** *heap-write h a′ al v′ h″* **..**
  ultimately have *heap-copy-loc a a′ al h ?obs″ h″* **..**
  thus *?thesis* **using** *1* ‹*v′* ∈ *vs* (*a*, *al*)› **by**(*auto*)
qed

lemma *heap-copies-non-speculative-read*:
  assumes *hrt*: *heap-read-typeable hconf P*
  and *copies*: *heap-copies a a′ als h obs h′*
  and *vs*: *vs-conf P h vs*
  and *type1*: *list-all2* (λ*al T*. *P,h* ⊢ *a@al* : *T*) *als Ts*
  and *type2*: *list-all2* (λ*al T*. *P,h* ⊢ *a′@al* : *T*) *als Ts*
  and *hconf*: *hconf h*
  and *i*: *i* < *length obs*
  and *read*: *obs* ! *i* = *ReadMem a″ al″ v*
  and *v*: *v′* ∈ *w-values P vs* (*map NormalAction* (*take i obs*)) (*a″*, *al″*)
  and *ns*: *non-speculative P vs* (*llist-of* (*map NormalAction* (*take i obs*)))
  shows ∃ *obs′ h″*. *heap-copies a a′ als h obs′ h″* ∧ *i* < *length obs′* ∧ *take i obs′* = *take i obs* ∧
                obs′ ! *i* = *ReadMem a″ al″ v′* ∧ *length obs′* ≤ *length obs*
  (**is** *?concl als h obs vs i*)
using *copies vs type1 type2 hconf i read v ns*
proof(*induction arbitrary*: *Ts vs i*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons al h ob h′ als obs h″ Ts vs*)
  note *copy* = ‹*heap-copy-loc a a′ al h ob h′*›
  note *vs* = ‹*vs-conf P h vs*›
  note *type1* = ‹*list-all2* (λ*al T*. *P,h* ⊢ *a@al* : *T*) (*al* # *als*) *Ts*›
    and *type2* = ‹*list-all2* (λ*al T*. *P,h* ⊢ *a′@al* : *T*) (*al* # *als*) *Ts*›

note *hconf* = ‹*hconf h*›
note *i* = ‹*i* < *length* (*ob* @ *obs*)›
note *read* = ‹(*ob* @ *obs*) ! *i* = *ReadMem a'' al'' v›
note *v* = ‹*v'* ∈ *w-values P vs* (*map NormalAction* (*take i* (*ob* @ *obs*))) (*a''*, *al''*)›
note *ns* = ‹*non-speculative P vs* (*llist-of* (*map NormalAction* (*take i* (*ob* @ *obs*))))›

from *type1* **obtain** *T Ts'* **where** *Ts*: *Ts* = *T* # *Ts'*
and *type1'*: *P*,*h* ⊢ *a*@*al* : *T*
and *type1''*: *list-all2* (λ*al T*. *P*,*h* ⊢ *a*@*al* : *T*) *als Ts'*
by(*auto simp add*: *list-all2-Cons1*)
from *type2 Ts* **have** *type2'*: *P*,*h* ⊢ *a'*@*al* : *T*
and *type2''*: *list-all2* (λ*al T*. *P*,*h* ⊢ *a'*@*al* : *T*) *als Ts'*
by *simp-all*
show *?case*
proof(*cases i* < *length ob*)
case *True*
with *read v*
have *ob* ! *i* = *ReadMem a'' al'' v*
and *v'* ∈ *w-values P vs* (*map NormalAction* (*take i ob*)) (*a''*, *al''*) **by**(*simp-all add*: *nth-append*)
from *heap-copy-loc-non-speculative-read*[*OF hrt vs type1' type2' hconf copy True this*]
obtain *ob' H''* **where** *copy'*: *heap-copy-loc a a' al h ob' H''*
and *i'*: *i* < *length ob'* **and** *take i ob'* = *take i ob*
and *ob'* ! *i* = *ReadMem a'' al'' v'*
and *length ob'* ≤ *length ob*
and *ns*: *non-speculative P vs* (*llist-of* (*map NormalAction ob'*)) **by** *blast*
moreover {
from *copy'* **have** *hext*: *h* ⊴ *H''* **by**(*rule hext-heap-copy-loc*)
have *hconf H''*
by(*rule heap-conf-read.hconf-heap-copy-loc-mono*[*OF heap-conf-read-heap-read-typed*])(*rule heap-copy-loc-non-copy' ns vs hconf type1' type2'*], *fact*+)
moreover
from *type1''* **have** *list-all2* (λ*al T*. *P*,*H''* ⊢ *a*@*al* : *T*) *als Ts'*
by(*rule List.list-all2-mono*)(*rule addr-loc-type-hext-mono*[*OF* - *hext*])
moreover from *type2''* **have** *list-all2* (λ*al T*. *P*,*H''* ⊢ *a'*@*al* : *T*) *als Ts'*
by(*rule List.list-all2-mono*)(*rule addr-loc-type-hext-mono*[*OF* - *hext*])
moreover note *calculation* }
from *heap-copies-progress*[*OF this*]
obtain *obs' h'''* **where** ∗: *heap-copies a a' als H'' obs' h'''* **by** *blast*
moreover note *heap-copies-length*[*OF* ∗]
moreover note *heap-copy-loc-length*[*OF copy'*]
moreover note *heap-copies-length*[*OF* ‹*heap-copies a a' als h' obs h''*›]
ultimately show *?thesis* **using** *True* **by**(*auto intro*!: *heap-copies.Cons exI simp add*: *nth-append*)
next
case *False*
let *?vs'* = *w-values P vs* (*map NormalAction ob*)
let *?i'* = *i* − *length ob*

from *ns False* **obtain** *ns'*: *non-speculative P vs* (*llist-of* (*map NormalAction ob*))
and *ns''*: *non-speculative P ?vs'* (*llist-of* (*map NormalAction* (*take ?i' obs*)))
by(*simp add*: *lappend-llist-of-llist-of*[*symmetric*] *non-speculative-lappend del*: *lappend-llist-of-llist-of*)

from *heap-copy-loc-non-speculative-vs-conf*[*OF copy* - *vs hconf type1' type2'*, **where** *n*=*length ob*] *ns'*
have *vs-conf P h' ?vs'* **by** *simp*

```
    moreover
    from copy have hext: h ⊴ h' by(rule hext-heap-copy-loc)
    from type1'' have list-all2 (λal T. P,h' ⊢ a@al : T) als Ts'
      by(rule List.list-all2-mono)(rule addr-loc-type-hext-mono[OF - hext])
    moreover from type2'' have list-all2 (λal T. P,h' ⊢ a'@al : T) als Ts'
      by(rule List.list-all2-mono)(rule addr-loc-type-hext-mono[OF - hext])
    moreover have hconf h'
    by(rule heap-conf-read.hconf-heap-copy-loc-mono[OF heap-conf-read-heap-read-typed])(rule heap-copy-loc-non-speculative-t
copy ns' vs hconf type1' type2'], fact+)
    moreover from i False have ?i' < length obs by simp
    moreover from read False have obs ! ?i' = ReadMem a'' al'' v by(simp add: nth-append)
    moreover from v False have v' ∈ w-values P ?vs' (map NormalAction (take ?i' obs)) (a'', al'')
by(simp)
    ultimately have ?concl als h' obs ?vs' ?i' using ns'' by(rule Cons.IH)
    thus ?thesis using False copy by safe(auto intro!: heap-copies.Cons exI simp add: nth-append)
  qed
qed

lemma heap-clone-non-speculative-read:
  assumes hrt: heap-read-typeable hconf P
  and clone: heap-clone P h a h' ⌊(obs, a')⌋
  and vs: vs-conf P h vs
  and hconf: hconf h
  and i: i < length obs
  and read: obs ! i = ReadMem a'' al'' v
  and v: v' ∈ w-values P vs (map NormalAction (take i obs)) (a'', al'')
  and ns: non-speculative P vs (llist-of (map NormalAction (take i obs)))
  shows ∃ obs' h''. heap-clone P h a h'' ⌊(obs', a')⌋ ∧ i < length obs' ∧ take i obs' = take i obs ∧
                    obs' ! i = ReadMem a'' al'' v' ∧ length obs' ≤ length obs
using clone
proof cases
  case (ObjClone C h'' FDTs obs')

  note obs = ‹obs = NewHeapElem a' (Class-type C) # obs'›
  note FDTs = ‹P ⊢ C has-fields FDTs›
  let ?als = map (λ((F, D), Tm). CField D F) FDTs
  let ?Ts = map (λ(FD, T). fst (the (map-of FDTs FD))) FDTs
   let ?vs = w-value P vs (NormalAction (NewHeapElem a' (Class-type C)) :: ('addr, 'thread-id)
obs-event action)
  let ?i = i - 1
  from i read obs have i-0: i > 0 by(simp add: nth-Cons' split: if-split-asm)

  from ‹P ⊢ C has-fields FDTs› have is-class P C by(rule has-fields-is-class)
  with ‹(h'', a') ∈ allocate h (Class-type C)›
  have type-a': typeof-addr h'' a' = ⌊Class-type C⌋ and hext: h ⊴ h''
    by(auto dest: allocate-SomeD hext-allocate)

  note ‹heap-copies a a' ?als h'' obs' h'›
  moreover from ‹typeof-addr h a = ⌊Class-type C⌋› hconf have is-htype P (Class-type C)
    by(rule typeof-addr-is-type)
  with vs ‹(h'', a') ∈ allocate h (Class-type C)›
  have vs-conf P h'' ?vs by(rule vs-conf-allocate)
  moreover
  from hext ‹typeof-addr h a = ⌊Class-type C⌋›
```

have *typeof-addr h'' a = ⌊Class-type C⌋* **by**(*rule typeof-addr-hext-mono*)
hence *list-all2 (λal T. P,h'' ⊢ a@al : T) ?als ?Ts* **using** *FDTs*
unfolding *list-all2-map1 list-all2-map2 list-all2-same*
by(*fastforce intro*: *addr-loc-type.intros simp add*: *has-field-def dest*: *weak-map-of-SomeI*)
moreover from *FDTs type-a'*
have *list-all2 (λal T. P,h'' ⊢ a'@al : T) ?als ?Ts*
unfolding *list-all2-map1 list-all2-map2 list-all2-same*
by(*fastforce intro*: *addr-loc-type.intros simp add*: *has-field-def dest*: *weak-map-of-SomeI*)
moreover from ‹*(h'', a') ∈ allocate h (Class-type C)*› *hconf* ‹*is-htype P (Class-type C)*›
have *hconf h''* **by**(*rule hconf-allocate-mono*)
moreover from *i read i-0 obs* **have** *?i < length obs' obs' ! ?i = ReadMem a'' al'' v* **by** *simp-all*
moreover from *v i-0 obs*
have *v' ∈ w-values P ?vs (map NormalAction (take ?i obs')) (a'', al'')* **by**(*simp add*: *take-Cons'*)
moreover from *ns i-0 obs*
have *non-speculative P ?vs (llist-of (map NormalAction (take ?i obs')))* **by**(*simp add*: *take-Cons'*)
ultimately have *∃ obs'' h'''. heap-copies a a' ?als h'' obs'' h''' ∧*
?i < length obs'' ∧ take ?i obs'' = take ?i obs' ∧ obs'' ! ?i = ReadMem a'' al'' v' ∧
length obs'' ≤ length obs'
by(*rule heap-copies-non-speculative-read*[*OF hrt*])
thus *?thesis* **using** ‹*typeof-addr h a = ⌊Class-type C⌋*› ‹*(h'', a') ∈ allocate h (Class-type C)*› *FDTs obs i-0*
by(*auto 4 4 intro*: *heap-clone.ObjClone simp add*: *take-Cons'*)
next
case (*ArrClone T n h'' FDTs obs'*)

note *obs = ‹obs = NewHeapElem a' (Array-type T n) # obs'›*
note *FDTs = ‹P ⊢ Object has-fields FDTs›*
let *?als = map (λ((F, D), Tfm). CField D F) FDTs @ map ACell [0..<n]*
let *?Ts = map (λ(FD, T). fst (the (map-of FDTs FD))) FDTs @ replicate n T*
let *?vs = w-value P vs (NormalAction (NewHeapElem a' (Array-type T n)) :: ('addr, 'thread-id) obs-event action)*
let *?i = i − 1*
from *i read obs* **have** *i-0*: *i > 0* **by**(*simp add*: *nth-Cons' split*: *if-split-asm*)

from ‹*typeof-addr h a = ⌊Array-type T n⌋*› *hconf*
have *is-htype P (Array-type T n)* **by**(*rule typeof-addr-is-type*)
with ‹*(h'', a') ∈ allocate h (Array-type T n)*›
have *type-a'*: *typeof-addr h'' a' = ⌊Array-type T n⌋*
and *hext*: *h ⊴ h''*
by(*auto dest*: *allocate-SomeD hext-allocate*)

note ‹*heap-copies a a' ?als h'' obs' h'*›
moreover from *vs* ‹*(h'', a') ∈ allocate h (Array-type T n)*› ‹*is-htype P (Array-type T n)*›
have *vs-conf P h'' ?vs* **by**(*rule vs-conf-allocate*)
moreover from *hext* ‹*typeof-addr h a = ⌊Array-type T n⌋*›
have *type'a*: *typeof-addr h'' a = ⌊Array-type T n⌋*
by(*auto intro*: *hext-arrD*)
from *type'a FDTs* **have** *list-all2 (λal T. P,h'' ⊢ a@al : T) ?als ?Ts*
by(*fastforce intro*: *list-all2-all-nthI addr-loc-type.intros simp add*: *has-field-def list-all2-append list-all2-map1 list-all2-map2 list-all2-same dest*: *weak-map-of-SomeI*)
moreover from *type-a' FDTs*
have *list-all2 (λal T. P,h'' ⊢ a'@al : T) ?als ?Ts*
by(*fastforce intro*: *list-all2-all-nthI addr-loc-type.intros simp add*: *has-field-def list-all2-append*

list-all2-map1 list-all2-map2 list-all2-same dest: *weak-map-of-SomeI*)
moreover from ‹$(h'', a') \in allocate\ h\ (Array\text{-}type\ T\ n)$› *hconf* ‹*is-htype P* (*Array-type T n*)›
have *hconf h''* **by**(*rule hconf-allocate-mono*)
moreover from *i read i-0 obs* **have** *?i* < *length obs' obs'* ! *?i* = *ReadMem a'' al'' v* **by** *simp-all*
moreover from *v i-0 obs*
have $v' \in$ *w-values P ?vs* (*map NormalAction* (*take ?i obs'*)) (*a''*, *al''*) **by**(*simp add*: *take-Cons'*)
moreover from *ns i-0 obs*
have *non-speculative P ?vs* (*llist-of* (*map NormalAction* (*take ?i obs'*))) **by**(*simp add*: *take-Cons'*)
ultimately have $\exists$ *obs'' h'''*. *heap-copies a a' ?als h'' obs'' h'''* $\wedge$
?i < *length obs''* $\wedge$ *take ?i obs''* = *take ?i obs'* $\wedge$ *obs''* ! *?i* = *ReadMem a'' al'' v'* $\wedge$
length obs'' $\leq$ *length obs'*
by(*rule heap-copies-non-speculative-read*[*OF hrt*])
thus *?thesis* **using** ‹*typeof-addr h a* = $\lfloor$*Array-type T n*$\rfloor$› ‹$(h'', a') \in allocate\ h\ (Array\text{-}type\ T\ n)$› *FDTs obs i-0*
by(*auto 4 4 intro*: *heap-clone.ArrClone simp add*: *take-Cons'*)
qed

lemma *red-external-non-speculative-read*:
assumes *hrt*: *heap-read-typeable hconf P*
and *vs*: *vs-conf P* (*shr s*) *vs*
and *red*: *P*,*t* $\vdash$ $\langle a\cdot M(vs'),\ shr\ s\rangle$ $-ta\rightarrow ext$ $\langle va,h'\rangle$
and *aok*: *final-thread.actions-ok final s t ta*
and *hconf*: *hconf* (*shr s*)
and *i*: *i* < *length* $\{\!|ta|\!\}_o$
and *read*: $\{\!|ta|\!\}_o$! *i* = *ReadMem a'' al'' v*
and *v*: $v' \in$ *w-values P vs* (*map NormalAction* (*take i* $\{\!|ta|\!\}_o$)) (*a''*, *al''*)
and *ns*: *non-speculative P vs* (*llist-of* (*map NormalAction* (*take i* $\{\!|ta|\!\}_o$)))
shows $\exists$ *ta'' va'' h''*. *P*,*t* $\vdash$ $\langle a\cdot M(vs'),\ shr\ s\rangle$ $-ta''\rightarrow ext$ $\langle va'',\ h''\rangle$ $\wedge$ *final-thread.actions-ok final s t ta''* $\wedge$
i < *length* $\{\!|ta''|\!\}_o$ $\wedge$ *take i* $\{\!|ta''|\!\}_o$ = *take i* $\{\!|ta|\!\}_o$ $\wedge$
$\{\!|ta''|\!\}_o$! *i* = *ReadMem a'' al'' v'* $\wedge$ *length* $\{\!|ta''|\!\}_o \leq$ *length* $\{\!|ta|\!\}_o$
using *red i read*
proof *cases*
case [*simp*]: (*RedClone obs a'*)
from *heap-clone-non-speculative-read*[*OF hrt* ‹*heap-clone P* (*shr s*) *a h'* $\lfloor$(*obs*, *a'*)$\rfloor$› *vs hconf*, *of i a'' al'' v v'*] *i read v ns*
show *?thesis* **using** *aok*
by(*fastforce intro*: *red-external.RedClone simp add*: *final-thread.actions-ok-iff*)
qed(*auto simp add*: *nth-Cons*)

lemma *red-external-aggr-non-speculative-read*:
assumes *hrt*: *heap-read-typeable hconf P*
and *vs*: *vs-conf P* (*shr s*) *vs*
and *red*: (*ta*, *va*, *h'*) $\in$ *red-external-aggr P t a M vs'* (*shr s*)
and *native*: *is-native P* (*the* (*typeof-addr* (*shr s*) *a*)) *M*
and *aok*: *final-thread.actions-ok final s t ta*
and *hconf*: *hconf* (*shr s*)
and *i*: *i* < *length* $\{\!|ta|\!\}_o$
and *read*: $\{\!|ta|\!\}_o$! *i* = *ReadMem a'' al'' v*
and *v*: $v' \in$ *w-values P vs* (*map NormalAction* (*take i* $\{\!|ta|\!\}_o$)) (*a''*, *al''*)
and *ns*: *non-speculative P vs* (*llist-of* (*map NormalAction* (*take i* $\{\!|ta|\!\}_o$)))
shows $\exists$ *ta'' va'' h''*. (*ta''*, *va''*, *h''*) $\in$ *red-external-aggr P t a M vs'* (*shr s*) $\wedge$ *final-thread.actions-ok final s t ta''* $\wedge$

$i < length\ \{\!|ta''|\!\}_o \wedge take\ i\ \{\!|ta''|\!\}_o = take\ i\ \{\!|ta|\!\}_o \wedge$
$\{\!|ta''|\!\}_o\ !\ i = ReadMem\ a''\ al''\ v' \wedge length\ \{\!|ta''|\!\}_o \leq length\ \{\!|ta|\!\}_o$
using *red native aok hconf i read v ns*
apply(*simp add*: *red-external-aggr-def final-thread.actions-ok-iff ex-disj-distrib conj-disj-distribR split nth-Cons' del*: *if-split split*: *if-split-asm disj-split-asm*)
apply(*drule heap-clone-non-speculative-read*[*OF hrt - vs hconf*, *of - - - - i a'' al'' v v'*])
apply *simp-all*
apply(*fastforce*)
done

end

declare *split-paired-Ex* [*simp*]
declare *eq-upto-seq-inconsist-simps* [*simp del*]

context *allocated-heap* **begin**

lemma *heap-copy-loc-allocated-same*:
assumes *heap-copy-loc a a' al h obs h'*
shows *allocated h' = allocated h*
using *assms*
by *cases*(*auto del*: *subsetI simp*: *heap-write-allocated-same*)

lemma *heap-copy-loc-allocated-mono*:
heap-copy-loc a a' al h obs h' $\Longrightarrow$ *allocated h* $\subseteq$ *allocated h'*
by(*simp add*: *heap-copy-loc-allocated-same*)

lemma *heap-copies-allocated-same*:
assumes *heap-copies a a' al h obs h'*
shows *allocated h' = allocated h*
using *assms*
by(*induct*)(*auto simp add*: *heap-copy-loc-allocated-same*)

lemma *heap-copies-allocated-mono*:
heap-copies a a' al h obs h' $\Longrightarrow$ *allocated h* $\subseteq$ *allocated h'*
by(*simp add*: *heap-copies-allocated-same*)

lemma *heap-clone-allocated-mono*:
assumes *heap-clone P h a h' aobs*
shows *allocated h* $\subseteq$ *allocated h'*
using *assms*
by *cases*(*blast del*: *subsetI intro*: *heap-copies-allocated-mono allocate-allocated-mono intro*: *subset-trans*)+

lemma *red-external-allocated-mono*:
assumes $P,t \vdash \langle a\bullet M(vs), h\rangle\ -ta\rightarrow ext\ \langle va, h'\rangle$
shows *allocated h* $\subseteq$ *allocated h'*
using *assms*
by(*cases*)(*blast del*: *subsetI intro*: *heap-clone-allocated-mono heap-write-allocated-same*)+

lemma *red-external-aggr-allocated-mono*:
$[\![\ (ta, va, h') \in red\text{-}external\text{-}aggr\ P\ t\ a\ M\ vs\ h;\ is\text{-}native\ P\ (the\ (typeof\text{-}addr\ h\ a))\ M\ ]\!]$
$\Longrightarrow$ *allocated h* $\subseteq$ *allocated h'*
by(*cases the* (*typeof-addr h a*))(*auto simp add*: *is-native.simps external-WT-defs.simps red-external-aggr-def*

split: *if-split-asm dest*: *heap-clone-allocated-mono sees-method-decl-above*)

lemma *heap-clone-allocatedD*:
assumes *heap-clone P h a h' ⌊(obs, a')⌋*
and *NewHeapElem a'' x ∈ set obs*
shows *a'' ∈ allocated h' ∧ a'' ∉ allocated h*
using *assms*
by *cases*(*auto dest*: *allocate-allocatedD heap-copies-allocated-mono heap-copies-not-New*)

lemma *red-external-allocatedD*:
⟦ *P,t ⊢ ⟨a•M(vs), h⟩ −ta→ext ⟨va, h'⟩; NewHeapElem a' x ∈ set {|ta|}$_o$* ⟧
⟹ *a' ∈ allocated h' ∧ a' ∉ allocated h*
by(*erule red-external.cases*)(*auto dest*: *heap-clone-allocatedD*)

lemma *red-external-aggr-allocatedD*:
⟦ *(ta, va, h') ∈ red-external-aggr P t a M vs h; NewHeapElem a' x ∈ set {|ta|}$_o$;*
is-native P (the (typeof-addr h a)) M ⟧
⟹ *a' ∈ allocated h' ∧ a' ∉ allocated h*
by(*auto simp add*: *is-native.simps external-WT-defs.simps red-external-aggr-def split*: *if-split-asm dest*: *heap-clone-allocatedD sees-method-decl-above*)

lemma *heap-clone-NewHeapElemD*:
assumes *heap-clone P h a h' ⌊(obs, a')⌋*
and *ad ∈ allocated h'*
and *ad ∉ allocated h*
shows *∃ CTn. NewHeapElem ad CTn ∈ set obs*
using *assms*
by *cases*(*auto dest*!: *allocate-allocatedD heap-copies-allocated-same*)

lemma *heap-clone-fail-allocated-same*:
assumes *heap-clone P h a h' None*
shows *allocated h' = allocated h*
using *assms*
by(*cases*)(*auto*)

lemma *red-external-NewHeapElemD*:
⟦ *P,t ⊢ ⟨a•M(vs), h⟩ −ta→ext ⟨va, h'⟩; a' ∈ allocated h'; a' ∉ allocated h* ⟧
⟹ *∃ CTn. NewHeapElem a' CTn ∈ set {|ta|}$_o$*
by(*erule red-external.cases*)(*auto dest*: *heap-clone-NewHeapElemD heap-clone-fail-allocated-same*)

lemma *red-external-aggr-NewHeapElemD*:
⟦ *(ta, va, h') ∈ red-external-aggr P t a M vs h; a' ∈ allocated h'; a' ∉ allocated h;*
is-native P (the (typeof-addr h a)) M ⟧
⟹ *∃ CTn. NewHeapElem a' CTn ∈ set {|ta|}$_o$*
by(*cases the* (*typeof-addr h a*))(*auto simp add*: *is-native.simps external-WT-defs.simps red-external-aggr-def split*: *if-split-asm dest*: *heap-clone-fail-allocated-same heap-clone-NewHeapElemD sees-method-decl-above*)

end

context *heap-base* **begin**

lemma *binop-known-addrs*:
assumes *ok*: *start-heap-ok*
shows *binop bop v1 v2 = ⌊Inl v⌋ ⟹ ka-Val v ⊆ ka-Val v1 ∪ ka-Val v2 ∪ set start-addrs*

and *binop bop v1 v2* = ⌊*Inr a*⌋ ⟹ *a* ∈ *ka-Val v1* ∪ *ka-Val v2* ∪ *set start-addrs*
apply(*cases bop*, *auto split*: *if-split-asm*)[*1*]
apply(*cases bop*, *auto split*: *if-split-asm simp add*: *addr-of-sys-xcpt-start-addr*[*OF ok*])
done

lemma *heap-copy-loc-known-addrs-ReadMem*:
assumes *heap-copy-loc a a′ al h ob h′*
and *ReadMem ad al′ v* ∈ *set ob*
shows *ad* = *a*
using *assms* **by** *cases simp*

lemma *heap-copies-known-addrs-ReadMem*:
assumes *heap-copies a a′ als h obs h′*
and *ReadMem ad al v* ∈ *set obs*
shows *ad* = *a*
using *assms*
by(*induct*)(*auto dest*: *heap-copy-loc-known-addrs-ReadMem*)

lemma *heap-clone-known-addrs-ReadMem*:
assumes *heap-clone P h a h′* ⌊(*obs*, *a′*)⌋
and *ReadMem ad al v* ∈ *set obs*
shows *ad* = *a*
using *assms*
by *cases*(*auto dest*: *heap-copies-known-addrs-ReadMem*)

lemma *red-external-known-addrs-ReadMem*:
⟦ *P*,*t* ⊢ ⟨*a*•*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*,*h*′⟩; *ReadMem ad al v* ∈ *set* {|*ta*|}*o* ⟧
⟹ *ad* ∈ {*thread-id2addr t*, *a*} ∪ (⋃(*ka-Val* ‘ *set vs*)) ∪ *set start-addrs*
by(*erule red-external.cases*)(*simp-all add*: *heap-clone-known-addrs-ReadMem*)

lemma *red-external-aggr-known-addrs-ReadMem*:
⟦ (*ta*, *va*, *h′*) ∈ *red-external-aggr P t a M vs h*; *ReadMem ad al v* ∈ *set* {|*ta*|}*o* ⟧
⟹ *ad* ∈ {*thread-id2addr t*, *a*} ∪ (⋃(*ka-Val* ‘ *set vs*)) ∪ *set start-addrs*
apply(*auto simp add*: *red-external-aggr-def split*: *if-split-asm dest*: *heap-clone-known-addrs-ReadMem*)
done

lemma *heap-copy-loc-known-addrs-WriteMem*:
assumes *heap-copy-loc a a′ al h ob h′*
and *ob* ! *n* = *WriteMem ad al′* (*Addr a″*) *n* < *length ob*
shows *a″* ∈ *new-obs-addrs* (*take n ob*)
using *assms*
by *cases*(*auto simp add*: *nth-Cons new-obs-addrs-def split*: *nat.split-asm*)

lemma *heap-copies-known-addrs-WriteMem*:
assumes *heap-copies a a′ als h obs h′*
and *obs* ! *n* = *WriteMem ad al* (*Addr a″*) *n* < *length obs*
shows *a″* ∈ *new-obs-addrs* (*take n obs*)
using *assms*
by(*induct arbitrary*: *n*)(*auto simp add*: *nth-append new-obs-addrs-def dest*: *heap-copy-loc-known-addrs-WriteMem split*: *if-split-asm*)

lemma *heap-clone-known-addrs-WriteMem*:
assumes *heap-clone P h a h′* ⌊(*obs*, *a′*)⌋
and *obs* ! *n* = *WriteMem ad al* (*Addr a″*) *n* < *length obs*

shows $a'' \in$ *new-obs-addrs* (*take n obs*)
using *assms*
by *cases*(*auto simp add*: *nth-Cons new-obs-addrs-def split*: *nat.split-asm dest*: *heap-copies-known-addrs-WriteMem*)

lemma *red-external-known-addrs-WriteMem*:
$[\![P,t \vdash \langle a\bullet M(vs), h\rangle -ta\rightarrow ext\ \langle va,h'\rangle; \{\!|ta|\!\}_o\ !\ n = WriteMem\ ad\ al\ (Addr\ a'); n < length\ \{\!|ta|\!\}_o]\!]$
$\Longrightarrow a' \in \{thread\text{-}id2addr\ t, a\} \cup (\bigcup(ka\text{-}Val\ `\ set\ vs)) \cup set\ start\text{-}addrs \cup new\text{-}obs\text{-}addrs\ (take\ n\ \{\!|ta|\!\}_o)$
by(*erule red-external.cases*)(*auto dest*: *heap-clone-known-addrs-WriteMem*)

lemma *red-external-aggr-known-addrs-WriteMem*:
$[\![(ta, va, h') \in red\text{-}external\text{-}aggr\ P\ t\ a\ M\ vs\ h;$
$\{\!|ta|\!\}_o\ !\ n = WriteMem\ ad\ al\ (Addr\ a'); n < length\ \{\!|ta|\!\}_o]\!]$
$\Longrightarrow a' \in \{thread\text{-}id2addr\ t, a\} \cup (\bigcup(ka\text{-}Val\ `\ set\ vs)) \cup set\ start\text{-}addrs \cup new\text{-}obs\text{-}addrs\ (take\ n\ \{\!|ta|\!\}_o)$
apply(*auto simp add*: *red-external-aggr-def split*: *if-split-asm dest*: *heap-clone-known-addrs-WriteMem*)
done

lemma *red-external-known-addrs-mono*:
assumes *ok*: *start-heap-ok*
and *red*: $P,t \vdash \langle a\bullet M(vs), h\rangle -ta\rightarrow ext\ \langle va, h'\rangle$
shows (*case va of RetVal v* $\Rightarrow$ *ka-Val v* | *RetExc a* $\Rightarrow \{a\}$ | *RetStaySame* $\Rightarrow \{\}$) $\subseteq \{thread\text{-}id2addr\ t, a\} \cup (\bigcup(ka\text{-}Val\ `\ set\ vs)) \cup set\ start\text{-}addrs \cup new\text{-}obs\text{-}addrs\ \{\!|ta|\!\}_o$
using *red*
by *cases*(*auto simp add*: *addr-of-sys-xcpt-start-addr*[*OF ok*] *new-obs-addrs-def heap-clone.simps*)

lemma *red-external-aggr-known-addrs-mono*:
assumes *ok*: *start-heap-ok*
and *red*: $(ta, va, h') \in red\text{-}external\text{-}aggr\ P\ t\ a\ M\ vs\ h\ is\text{-}native\ P\ (the\ (typeof\text{-}addr\ h\ a))\ M$
shows (*case va of RetVal v* $\Rightarrow$ *ka-Val v* | *RetExc a* $\Rightarrow \{a\}$ | *RetStaySame* $\Rightarrow \{\}$) $\subseteq \{thread\text{-}id2addr\ t, a\} \cup (\bigcup(ka\text{-}Val\ `\ set\ vs)) \cup set\ start\text{-}addrs \cup new\text{-}obs\text{-}addrs\ \{\!|ta|\!\}_o$
using *red*
apply(*cases the* (*typeof-addr h a*))
apply(*auto simp add*: *red-external-aggr-def addr-of-sys-xcpt-start-addr*[*OF ok*] *new-obs-addrs-def heap-clone.simps split*: *if-split-asm*)
apply(*auto simp add*: *is-native.simps elim*!: *external-WT-defs.cases dest*: *sees-method-decl-above*)
done

lemma *red-external-NewThread-idD*:
$[\![P,t \vdash \langle a\bullet M(vs), h\rangle -ta\rightarrow ext\ \langle va, h'\rangle; NewThread\ t'\ (C, M', a')\ h'' \in set\ \{\!|ta|\!\}_t]\!]$
$\Longrightarrow t' = addr2thread\text{-}id\ a \wedge a' = a$
by(*erule red-external.cases*) *simp-all*

lemma *red-external-aggr-NewThread-idD*:
$[\![(ta, va, h') \in red\text{-}external\text{-}aggr\ P\ t\ a\ M\ vs\ h;$
$NewThread\ t'\ (C, M', a')\ h'' \in set\ \{\!|ta|\!\}_t]\!]$
$\Longrightarrow t' = addr2thread\text{-}id\ a \wedge a' = a$
apply(*auto simp add*: *red-external-aggr-def split*: *if-split-asm*)
done

end

locale *heap''* =
heap'

addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
P
for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *P* :: *'m prog*
+
assumes *allocate-typeof-addr-SomeD*: ⟦ (*h'*, *a*) ∈ *allocate h hT*; *typeof-addr a* ≠ *None* ⟧ ⟹ *typeof-addr a* = ⌊*hT*⌋
begin

lemma *heap-copy-loc-New-type-match*:
⟦ *h.heap-copy-loc a a' al h obs h'*; *NewHeapElem ad CTn* ∈ *set obs*; *typeof-addr ad* ≠ *None* ⟧
⟹ *typeof-addr ad* = ⌊*CTn*⌋
by(*erule h.heap-copy-loc.cases*) *simp*

lemma *heap-copies-New-type-match*:
⟦ *h.heap-copies a a' als h obs h'*; *NewHeapElem ad CTn* ∈ *set obs*; *typeof-addr ad* ≠ *None* ⟧
⟹ *typeof-addr ad* = ⌊*CTn*⌋
by(*induct rule*: *h.heap-copies.induct*)(*auto dest*: *heap-copy-loc-New-type-match*)

lemma *heap-clone-New-type-match*:
⟦ *h.heap-clone P h a h'* ⌊(*obs*, *a'*)⌋; *NewHeapElem ad CTn* ∈ *set obs*; *typeof-addr ad* ≠ *None* ⟧
⟹ *typeof-addr ad* = ⌊*CTn*⌋
by(*erule h.heap-clone.cases*)(*auto dest*: *allocate-typeof-addr-SomeD heap-copies-New-type-match*)

lemma *red-external-New-type-match*:
⟦ *h.red-external P t a M vs h ta va h'*; *NewHeapElem ad CTn* ∈ *set* {|*ta*|}$_o$; *typeof-addr ad* ≠ *None* ⟧
⟹ *typeof-addr ad* = ⌊*CTn*⌋
by(*erule h.red-external.cases*)(*auto dest*: *heap-clone-New-type-match*)

lemma *red-external-aggr-New-type-match*:
⟦ (*ta*, *va*, *h'*) ∈ *h.red-external-aggr P t a M vs h*; *NewHeapElem ad CTn* ∈ *set* {|*ta*|}$_o$; *typeof-addr ad* ≠ *None* ⟧
⟹ *typeof-addr ad* = ⌊*CTn*⌋
by(*auto simp add*: *h.red-external-aggr-def split*: *if-split-asm dest*: *heap-clone-New-type-match*)

end

end
theory *JMM-J*
imports
JMM-Framework
../J/Threaded
begin

sublocale *J-heap-base* < *red-mthr*:

heap-multithreaded-base
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
final-expr mred P convert-RA
for *P*
.

context *J-heap-base* **begin**

abbreviation *J-*$\mathcal{E}$ ::
'addr J-prog ⇒ *cname* ⇒ *mname* ⇒ *'addr val list* ⇒ *status*
⇒ (*'thread-id* × (*'addr*, *'thread-id*) *obs-event action*) *llist set*
where
J-$\mathcal{E}$ *P* ≡ *red-mthr.*$\mathcal{E}$*-start P J-local-start P*

end

end

8.14 JMM Instantiation for J

theory *DRF-J*
imports
JMM-Common
JMM-J
../J/ProgressThreaded
SC-Legal
begin

primrec *ka* :: *'addr expr* ⇒ *'addr set*
and *kas* :: *'addr expr list* ⇒ *'addr set*
where
ka (*new C*) = {}
| *ka* (*newA T*⌊*e*⌉) = *ka e*
| *ka* (*Cast T e*) = *ka e*
| *ka* (*e instanceof T*) = *ka e*
| *ka* (*Val v*) = *ka-Val v*
| *ka* (*Var V*) = {}
| *ka* (*e1* «*bop*» *e2*) = *ka e1* ∪ *ka e2*
| *ka* (*V* := *e*) = *ka e*
| *ka* (*a*⌊*e*⌉) = *ka a* ∪ *ka e*
| *ka* (*a*⌊*e*⌉ := *e'*) = *ka a* ∪ *ka e* ∪ *ka e'*
| *ka* (*a*·*length*) = *ka a*
| *ka* (*e*·*F*{*D*}) = *ka e*
| *ka* (*e*·*F*{*D*} := *e'*) = *ka e* ∪ *ka e'*
| *ka* (*e*·*compareAndSwap*(*D*·*F*, *e'*, *e''*)) = *ka e* ∪ *ka e'* ∪ *ka e''*
| *ka* (*e*·*M*(*es*)) = *ka e* ∪ *kas es*
| *ka* {*V*:*T*=*vo*; *e*} = *ka e* ∪ (*case vo of None* ⇒ {} | *Some v* ⇒ *ka-Val v*)
| *ka* (*Synchronized x e e'*) = *ka e* ∪ *ka e'*
| *ka* (*InSynchronized x a e*) = *insert a* (*ka e*)
| *ka* (*e*;; *e'*) = *ka e* ∪ *ka e'*
| *ka* (*if* (*e*) *e1 else e2*) = *ka e* ∪ *ka e1* ∪ *ka e2*

| *ka* (*while* (*b*) *e*) = *ka b* ∪ *ka e*
| *ka* (*throw e*) = *ka e*
| *ka* (*try e catch*(*C V*) *e'*) = *ka e* ∪ *ka e'*

| *kas* [] = {}
| *kas* (*e* # *es*) = *ka e* ∪ *kas es*

definition *ka-locals* :: *'addr locals* ⇒ *'addr set*
where *ka-locals xs* = {*a*. *Addr a* ∈ *ran xs*}

lemma *ka-Val-subset-ka-locals*:
xs V = ⌊*v*⌋ ⟹ *ka-Val v* ⊆ *ka-locals xs*
by(*cases v*)(*auto simp add*: *ka-locals-def ran-def*)

lemma *ka-locals-update-subset*:
ka-locals (*xs*(*V* := *None*)) ⊆ *ka-locals xs*
ka-locals (*xs*(*V* ↦ *v*)) ⊆ *ka-Val v* ∪ *ka-locals xs*
by(*auto simp add*: *ka-locals-def ran-def*)

lemma *ka-locals-empty* [*simp*]: *ka-locals Map.empty* = {}
by(*simp add*: *ka-locals-def*)

lemma *kas-append* [*simp*]: *kas* (*es* @ *es'*) = *kas es* ∪ *kas es'*
by(*induct es*) *auto*

lemma *kas-map-Val* [*simp*]: *kas* (*map Val vs*) = ⋃(*ka-Val* ‘ *set vs*)
by(*induct vs*) *auto*

lemma *ka-blocks*:
⟦ *length pns* = *length Ts*; *length vs* = *length Ts* ⟧
⟹ *ka* (*blocks pns Ts vs body*) = ⋃(*ka-Val* ‘ *set vs*) ∪ *ka body*
by(*induct pns Ts vs body rule*: *blocks.induct*)(*auto*)

lemma *WT-ka*: *P*,*E* ⊢ *e* :: *T* ⟹ *ka e* = {}
and *WTs-kas*: *P*,*E* ⊢ *es* [::] *Ts* ⟹ *kas es* = {}
by(*induct rule*: *WT-WTs.inducts*)(*auto simp add*: *typeof-ka*)

context *J-heap-base* **begin**

primrec *J-known-addrs* :: *'thread-id* ⇒ *'addr expr* × *'addr locals* ⇒ *'addr set*
where *J-known-addrs t* (*e*, *xs*) = *insert* (*thread-id2addr t*) (*ka e* ∪ *ka-locals xs* ∪ *set start-addrs*)

lemma assumes *wf*: *wf-J-prog P*
and *ok*: *start-heap-ok*
shows *red-known-addrs-mono*:
P,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩ ⟹ *J-known-addrs t* (*e'*, *lcl s'*) ⊆ *J-known-addrs t* (*e*, *lcl s*) ∪ *new-obs-addrs* {|*ta*|}*o*
and *reds-known-addrs-mono*:
P,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩ ⟹ *kas es'* ∪ *ka-locals* (*lcl s'*) ⊆ *insert* (*thread-id2addr t*) (*kas es* ∪ *ka-locals* (*lcl s*)) ∪ *new-obs-addrs* {|*ta*|}*o* ∪ *set start-addrs*
proof(*induct rule*: *red-reds.inducts*)
case *RedVar* **thus** *?case* **by**(*auto dest*: *ka-Val-subset-ka-locals*)
next
case *RedLAss* **thus** *?case* **by**(*auto simp add*: *ka-locals-def ran-def*)

next
case *RedBinOp* **thus** *?case* **by**(*auto dest*: *binop-known-addrs*[*OF ok*])
next
case *RedBinOpFail* **thus** *?case* **by**(*auto dest*: *binop-known-addrs*[*OF ok*])
next
case *RedCall* **thus** *?case*
by(*auto simp add*: *ka-blocks new-obs-addrs-def wf-mdecl-def dest*!: *sees-wf-mdecl*[*OF wf*] *WT-ka*)
next
case (*RedCallExternal s a T M Ts T D vs ta va h'*) **thus** *?case*
by(*cases va*)(*auto dest*!: *red-external-known-addrs-mono*[*OF ok*])
next
case (*BlockRed e h l V vo ta e' h' l'*)
thus *?case* **using** *ka-locals-update-subset*[**where** *xs* = *l* **and** *V*=*V*] *ka-locals-update-subset*[**where** *xs* = *l'* **and** *V*=*V*]
apply(*cases l V*)
apply(*auto simp del*: *fun-upd-apply del*: *subsetI*)
apply(*blast dest*: *ka-Val-subset-ka-locals*)+
done
qed(*simp-all add*: *new-obs-addrs-def addr-of-sys-xcpt-start-addr*[*OF ok*] *subset-Un1 subset-Un2 subset-insert ka-Val-subset-new-obs-Addr-ReadMem ka-blocks del*: *fun-upd-apply*, *blast*+)

lemma *red-known-addrs-ReadMem*:
⟦ *P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; *ReadMem ad al v* ∈ *set* {|*ta*|}$_o$ ⟧ ⟹ *ad* ∈ *J-known-addrs t* (*e*, *lcl s*)
and *reds-known-addrss-ReadMem*:
⟦ *P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *ReadMem ad al v* ∈ *set* {|*ta*|}$_o$ ⟧
⟹ *ad* ∈ *insert* (*thread-id2addr t*) (*kas es* ∪ *ka-locals* (*lcl s*)) ∪ *set start-addrs*
proof(*induct rule*: *red-reds.inducts*)
case *RedCallExternal* **thus** *?case* **by** *simp* (*blast dest*: *red-external-known-addrs-ReadMem*)
next
case (*BlockRed e h l V vo ta e' h' l'*)
thus *?case* **using** *ka-locals-update-subset*[**where** *xs* = *l* **and** *V*=*V*] *ka-locals-update-subset*[**where** *xs* = *l'* **and** *V*=*V*]
by(*auto simp del*: *fun-upd-apply*)
qed(*simp-all*, *blast*+)

lemma *red-known-addrs-WriteMem*:
⟦ *P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; {|*ta*|}$_o$! *n* = *WriteMem ad al* (*Addr a*); *n* < *length* {|*ta*|}$_o$ ⟧
⟹ *a* ∈ *J-known-addrs t* (*e*, *lcl s*) ∨ *a* ∈ *new-obs-addrs* (*take n* {|*ta*|}$_o$)
and *reds-known-addrss-WriteMem*:
⟦ *P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; {|*ta*|}$_o$! *n* = *WriteMem ad al* (*Addr a*); *n* < *length* {|*ta*|}$_o$ ⟧
⟹ *a* ∈ *insert* (*thread-id2addr t*) (*kas es* ∪ *ka-locals* (*lcl s*)) ∪ *set start-addrs* ∪ *new-obs-addrs* (*take n* {|*ta*|}$_o$)
proof(*induct rule*: *red-reds.inducts*)
case *RedCASSucceed* **thus** *?case* **by**(*auto simp add*: *nth-Cons split*: *nat.split-asm*)
next
case *RedCallExternal* **thus** *?case* **by** *simp* (*blast dest*: *red-external-known-addrs-WriteMem*)
next
case (*BlockRed e h l V vo ta e' h' l'*)
thus *?case* **using** *ka-locals-update-subset*[**where** *xs* = *l* **and** *V*=*V*] *ka-locals-update-subset*[**where** *xs* = *l'* **and** *V*=*V*]
by(*auto simp del*: *fun-upd-apply*)
qed(*simp-all*, *blast*+)

end

context *J-heap* **begin**

lemma
assumes *wf*: *wf-J-prog P*
and *ok*: *start-heap-ok*
shows *red-known-addrs-new-thread*:
⟦ *P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; *NewThread t' x' h'* ∈ *set* {|*ta*|}$_t$ ⟧
⟹ *J-known-addrs t' x'* ⊆ *J-known-addrs t* (*e*, *lcl s*)
and *reds-known-addrss-new-thread*:
⟦ *P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *NewThread t' x' h'* ∈ *set* {|*ta*|}$_t$ ⟧
⟹ *J-known-addrs t' x'* ⊆ *insert* (*thread-id2addr t*) (*kas es* ∪ *ka-locals* (*lcl s*) ∪ *set start-addrs*)
proof(*induct rule*: *red-reds.inducts*)
case *RedCallExternal* **thus** *?case*
apply *clarsimp*
apply(*frule* (*1*) *red-external-new-thread-sub-thread*)
apply(*frule* (*1*) *red-external-NewThread-idD*)
apply *clarsimp*

apply(*drule* (*1*) *addr2thread-id-inverse*)
apply *simp*
apply(*drule sub-Thread-sees-run*[*OF wf*])
apply *clarsimp*
apply(*auto 4 4 dest*: *sees-wf-mdecl*[*OF wf*] *WT-ka simp add*: *wf-mdecl-def*)
done
next
case (*BlockRed e h l V vo ta e' h' l'*)
thus *?case* **using** *ka-locals-update-subset*[**where** *xs* = *l* **and** *V*=*V*] *ka-locals-update-subset*[**where** *xs* = *l'* **and** *V*=*V*]
by(*cases l V*)(*auto simp del*: *fun-upd-apply*)
qed(*simp-all*, *blast*+)

lemma *red-New-same-addr-same*:
⟦ *convert-extTA extTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩;
{|*ta*|}$_o$! *i* = *NewHeapElem a x*; *i* < *length* {|*ta*|}$_o$;
{|*ta*|}$_o$! *j* = *NewHeapElem a x'*; *j* < *length* {|*ta*|}$_o$ ⟧
⟹ *i* = *j*
and *reds-New-same-addr-same*:
⟦ *convert-extTA extTA*,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩;
{|*ta*|}$_o$! *i* = *NewHeapElem a x*; *i* < *length* {|*ta*|}$_o$;
{|*ta*|}$_o$! *j* = *NewHeapElem a x'*; *j* < *length* {|*ta*|}$_o$ ⟧
⟹ *i* = *j*
apply(*induct rule*: *red-reds.inducts*)
apply(*auto dest*: *red-external-New-same-addr-same simp add*: *nth-Cons split*: *nat.split-asm*)
done

end

locale *J-allocated-heap* = *allocated-heap* +
constrains *addr2thread-id* :: ('*addr* :: *addr*) ⇒ '*thread-id*
and *thread-id2addr* :: '*thread-id* ⇒ '*addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: '*heap*
and *allocate* :: '*heap* ⇒ *htype* ⇒ ('*heap* × '*addr*) *set*

and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *P* :: *'addr J-prog*

sublocale *J-allocated-heap* < *J-heap*
by(*unfold-locales*)

context *J-allocated-heap* **begin**

lemma *red-allocated-mono*: *P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩ ⟹ *allocated* (*hp s*) ⊆ *allocated* (*hp s'*)
and *reds-allocated-mono*: *P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩ ⟹ *allocated* (*hp s*) ⊆ *allocated* (*hp s'*)
by(*induct rule*: *red-reds.inducts*)(*auto dest*: *allocate-allocatedD heap-write-allocated-same red-external-allocated-mono del*: *subsetI*)

lemma *red-allocatedD*:
⟦ *P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; *NewHeapElem ad CTn* ∈ *set* {|*ta*|}*o* ⟧ ⟹ *ad* ∈ *allocated* (*hp s'*) ∧ *ad* ∉ *allocated* (*hp s*)
and *reds-allocatedD*:
⟦ *P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *NewHeapElem ad CTn* ∈ *set* {|*ta*|}*o* ⟧ ⟹ *ad* ∈ *allocated* (*hp s'*) ∧ *ad* ∉ *allocated* (*hp s*)
by(*induct rule*: *red-reds.inducts*)(*auto dest*: *allocate-allocatedD heap-write-allocated-same red-external-allocatedD*)

lemma *red-allocated-NewHeapElemD*:
⟦ *P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; *ad* ∈ *allocated* (*hp s'*); *ad* ∉ *allocated* (*hp s*) ⟧ ⟹ ∃ *CTn*. *NewHeapElem ad CTn* ∈ *set* {|*ta*|}*o*
and *reds-allocated-NewHeapElemD*:
⟦ *P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *ad* ∈ *allocated* (*hp s'*); *ad* ∉ *allocated* (*hp s*) ⟧ ⟹ ∃ *CTn*. *NewHeapElem ad CTn* ∈ *set* {|*ta*|}*o*
by(*induct rule*: *red-reds.inducts*)(*auto dest*: *allocate-allocatedD heap-write-allocated-same red-external-NewHeapElemD*)

lemma *mred-allocated-multithreaded*:
allocated-multithreaded addr2thread-id thread-id2addr empty-heap allocate typeof-addr heap-write allocated final-expr (*mred P*) *P*
proof
fix *t x m ta x' m'*
assume *mred P t* (*x*, *m*) *ta* (*x'*, *m'*)
thus *allocated m* ⊆ *allocated m'*
by(*auto dest*: *red-allocated-mono del*: *subsetI simp add*: *split-beta*)
next
fix *x t m ta x' m' ad CTn*
assume *mred P t* (*x*, *m*) *ta* (*x'*, *m'*)
and *NewHeapElem ad CTn* ∈ *set* {|*ta*|}*o*
thus *ad* ∈ *allocated m'* ∧ *ad* ∉ *allocated m*
by(*auto dest*: *red-allocatedD simp add*: *split-beta*)
next
fix *t x m ta x' m' ad*
assume *mred P t* (*x*, *m*) *ta* (*x'*, *m'*)
and *ad* ∈ *allocated m' ad* ∉ *allocated m*
thus ∃ *CTn*. *NewHeapElem ad CTn* ∈ *set* {|*ta*|}*o*
by(*auto dest*: *red-allocated-NewHeapElemD simp add*: *split-beta*)
next
fix *t x m ta x' m' i a CTn j CTn'*
assume *mred P t* (*x*, *m*) *ta* (*x'*, *m'*)

and $\{\!|ta|\!\}_o$! i = *NewHeapElem a CTn i* < *length* $\{\!|ta|\!\}_o$
and $\{\!|ta|\!\}_o$! j = *NewHeapElem a CTn' j* < *length* $\{\!|ta|\!\}_o$
thus $i = j$ **by**(*auto dest*: *red-New-same-addr-same simp add*: *split-beta*)
qed

end

sublocale *J-allocated-heap* < *red-mthr*: *allocated-multithreaded*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write allocated
final-expr mred P
P
by(*rule mred-allocated-multithreaded*)

context *J-allocated-heap* **begin**

lemma *mred-known-addrs*:
assumes *wf*: *wf-J-prog P*
and *ok*: *start-heap-ok*
shows *known-addrs addr2thread-id thread-id2addr empty-heap allocate typeof-addr heap-write allocated J-known-addrs final-expr* (*mred P*) *P*
proof
fix *t x m ta x' m'*
assume *mred P t* (*x*, *m*) *ta* (*x'*, *m'*)
thus *J-known-addrs t x'* ⊆ *J-known-addrs t x* ∪ *new-obs-addrs* $\{\!|ta|\!\}_o$
by(*auto del*: *subsetI simp add*: *split-beta dest*: *red-known-addrs-mono*[*OF wf ok*])
next
fix *t x m ta x' m' t' x'' m''*
assume *mred P t* (*x*, *m*) *ta* (*x'*, *m'*)
and *NewThread t' x'' m''* ∈ *set* $\{\!|ta|\!\}_t$
thus *J-known-addrs t' x''* ⊆ *J-known-addrs t x*
by(*auto del*: *subsetI simp add*: *split-beta dest*: *red-known-addrs-new-thread*[*OF wf ok*])
next
fix *t x m ta x' m' ad al v*
assume *mred P t* (*x*, *m*) *ta* (*x'*, *m'*)
and *ReadMem ad al v* ∈ *set* $\{\!|ta|\!\}_o$
thus *ad* ∈ *J-known-addrs t x*
by(*auto simp add*: *split-beta dest*: *red-known-addrs-ReadMem*)
next
fix *t x m ta x' m' n ad al ad'*
assume *mred P t* (*x*, *m*) *ta* (*x'*, *m'*)
and $\{\!|ta|\!\}_o$! *n* = *WriteMem ad al* (*Addr ad'*) *n* < *length* $\{\!|ta|\!\}_o$
thus *ad'* ∈ *J-known-addrs t x* ∨ *ad'* ∈ *new-obs-addrs* (*take n* $\{\!|ta|\!\}_o$)
by(*auto simp add*: *split-beta dest*: *red-known-addrs-WriteMem*)
qed

end

context *J-heap* **begin**

lemma *red-read-typeable*:
⟦ *convert-extTA extTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; *P*,*E*,*hp s* ⊢ *e* : *T*; *ReadMem ad al v* ∈ *set* $\{\!|ta|\!\}_o$ ⟧

$\Longrightarrow \exists T'.\ P,hp\ s \vdash ad@al : T'$
and *reds-read-typeable*:
$[\![\ convert\text{-}extTA\ extTA,P,t \vdash \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle;\ P,E,hp\ s \vdash es\ [:]\ Ts;\ ReadMem\ ad\ al\ v \in set\ \{\!|ta|\!\}_o\ ]\!]$
$\Longrightarrow \exists T'.\ P,hp\ s \vdash ad@al : T'$
proof(*induct arbitrary*: *E T* **and** *E Ts rule*: *red-reds.inducts*)
case *RedAAcc* **thus** *?case*
by(*fastforce intro*: *addr-loc-type.intros simp add*: *nat-less-iff word-sle-eq*)
next
case *RedFAcc* **thus** *?case*
by(*fastforce intro*: *addr-loc-type.intros*)
next
case *RedCASSucceed* **thus** *?case*
by(*fastforce intro*: *addr-loc-type.intros*)
next
case *RedCASFail* **thus** *?case*
by(*fastforce intro*: *addr-loc-type.intros*)
next
case *RedCallExternal* **thus** *?case*
by(*auto intro*: *red-external-read-mem-typeable*)
qed *auto*

end

primrec *new-types* :: $('a,\ 'b,\ 'addr)\ exp \Rightarrow ty\ set$
and *new-typess* :: $('a,\ 'b,\ 'addr)\ exp\ list \Rightarrow ty\ set$
where
$new\text{-}types\ (new\ C) = \{Class\ C\}$
$| new\text{-}types\ (newA\ T\lfloor e\rceil) = insert\ (T\lfloor\rceil)\ (new\text{-}types\ e)$
$| new\text{-}types\ (Cast\ T\ e) = new\text{-}types\ e$
$| new\text{-}types\ (e\ instanceof\ T) = new\text{-}types\ e$
$| new\text{-}types\ (Val\ v) = \{\}$
$| new\text{-}types\ (Var\ V) = \{\}$
$| new\text{-}types\ (e1\ «bop»\ e2) = new\text{-}types\ e1 \cup new\text{-}types\ e2$
$| new\text{-}types\ (V := e) = new\text{-}types\ e$
$| new\text{-}types\ (a\lfloor e\rceil) = new\text{-}types\ a \cup new\text{-}types\ e$
$| new\text{-}types\ (a\lfloor e\rceil := e') = new\text{-}types\ a \cup new\text{-}types\ e \cup new\text{-}types\ e'$
$| new\text{-}types\ (a{\cdot}length) = new\text{-}types\ a$
$| new\text{-}types\ (e{\cdot}F\{D\}) = new\text{-}types\ e$
$| new\text{-}types\ (e{\cdot}F\{D\} := e') = new\text{-}types\ e \cup new\text{-}types\ e'$
$| new\text{-}types\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e'')) = new\text{-}types\ e \cup new\text{-}types\ e' \cup new\text{-}types\ e''$
$| new\text{-}types\ (e{\cdot}M(es)) = new\text{-}types\ e \cup new\text{-}typess\ es$
$| new\text{-}types\ \{V{:}T{=}vo;\ e\} = new\text{-}types\ e$
$| new\text{-}types\ (Synchronized\ x\ e\ e') = new\text{-}types\ e \cup new\text{-}types\ e'$
$| new\text{-}types\ (InSynchronized\ x\ a\ e) = new\text{-}types\ e$
$| new\text{-}types\ (e;;\ e') = new\text{-}types\ e \cup new\text{-}types\ e'$
$| new\text{-}types\ (if\ (e)\ e1\ else\ e2) = new\text{-}types\ e \cup new\text{-}types\ e1 \cup new\text{-}types\ e2$
$| new\text{-}types\ (while\ (b)\ e) = new\text{-}types\ b \cup new\text{-}types\ e$
$| new\text{-}types\ (throw\ e) = new\text{-}types\ e$
$| new\text{-}types\ (try\ e\ catch(C\ V)\ e') = new\text{-}types\ e \cup new\text{-}types\ e'$

$| new\text{-}typess\ [] = \{\}$
$| new\text{-}typess\ (e\ \#\ es) = new\text{-}types\ e \cup new\text{-}typess\ es$

lemma *new-types-blocks*:
⟦ *length pns* = *length Ts*; *length vs* = *length Ts* ⟧ ⟹ *new-types* (*blocks pns vs Ts e*) = *new-types e*
apply(*induct rule*: *blocks.induct*)
apply(*simp-all*)
done

context *J-heap-base* **begin**

lemma *WTrt-new-types-types*: *P*,*E*,*h* ⊢ *e* : *T* ⟹ *new-types e* ⊆ *types P*
and *WTrts-new-typess-types*: *P*,*E*,*h* ⊢ *es* [:] *Ts* ⟹ *new-typess es* ⊆ *types P*
by(*induct rule*: *WTrt-WTrts.inducts*) *simp-all*

end

lemma *WT-new-types-types*: *P*,*E* ⊢ *e* :: *T* ⟹ *new-types e* ⊆ *types P*
and *WTs-new-typess-types*: *P*,*E* ⊢ *es* [::] *Ts* ⟹ *new-typess es* ⊆ *types P*
by(*induct rule*: *WT-WTs.inducts*) *simp-all*

context *J-heap-conf* **begin**

lemma *red-New-typeof-addrD*:
⟦ *convert-extTA extTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩; *new-types e* ⊆ *types P*; *hconf* (*hp s*); *NewHeapElem a x* ∈ *set* {|*ta*|}$_o$ ⟧
⟹ *typeof-addr* (*hp s*′) *a* = *Some x*
and *reds-New-typeof-addrD*:
⟦ *convert-extTA extTA*,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩; *new-typess es* ⊆ *types P*; *hconf* (*hp s*); *NewHeapElem a x* ∈ *set* {|*ta*|}$_o$ ⟧
⟹ *typeof-addr* (*hp s*′) *a* = *Some x*
apply(*induct rule*: *red-reds.inducts*)
apply(*auto dest*: *allocate-SomeD red-external-New-typeof-addrD*)
done

lemma *J-conf-read-heap-read-typed*:
J-conf-read addr2thread-id thread-id2addr empty-heap allocate typeof-addr (*heap-read-typed P*) *heap-write hconf P*
proof −
interpret *conf*: *heap-conf-read*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read-typed P heap-write hconf
P
by(*rule heap-conf-read-heap-read-typed*)
show *?thesis* **by**(*unfold-locales*)
qed

lemma *red-non-speculative-vs-conf*:
⟦ *convert-extTA extTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩; *P*,*E*,*hp s* ⊢ *e* : *T*;
non-speculative P vs (*llist-of* (*take n* (*map NormalAction* {|*ta*|}$_o$))); *vs-conf P* (*hp s*) *vs*; *hconf* (*hp s*) ⟧
⟹ *vs-conf P* (*hp s*′) (*w-values P vs* (*take n* (*map NormalAction* {|*ta*|}$_o$)))
and *reds-non-speculative-vs-conf*:
⟦ *convert-extTA extTA*,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩; *P*,*E*,*hp s* ⊢ *es* [:] *Ts*;
non-speculative P vs (*llist-of* (*take n* (*map NormalAction* {|*ta*|}$_o$))); *vs-conf P* (*hp s*) *vs*; *hconf* (*hp s*) ⟧

$\Longrightarrow$ *vs-conf P* (*hp s'*) (*w-values P vs* (*take n* (*map NormalAction* $\{\!|ta|\!\}_o$)))
proof(*induct arbitrary*: *E T* **and** *E Ts rule*: *red-reds.inducts*)
case (*RedAAss h a U n i w V h' xs*)
from ‹*sint i* < *int n*› ‹*0* <=*s i*› **have** *nat* (*sint i*) < *n*
by (*simp add*: *word-sle-eq nat-less-iff*)
with ‹*typeof-addr h a* = ⌊*Array-type U n*⌋› **have** *P*,*h* ⊢ *a*@*ACell* (*nat* (*sint i*)) : *U*
by(*auto intro*: *addr-loc-type.intros*)
moreover from ‹*heap-write h a* (*ACell* (*nat* (*sint i*))) *w h'*› **have** *h* ⊴ *h'* **by**(*rule hext-heap-write*)
ultimately have *P*,*h'* ⊢ *a*@*ACell* (*nat* (*sint i*)) : *U* **by**(*rule addr-loc-type-hext-mono*)
moreover from ‹*typeof*$_h$ *w* = ⌊*V*⌋› ‹*P* ⊢ *V* ≤ *U*› **have** *P*,*h* ⊢ *w* :≤ *U* **by**(*simp add*: *conf-def*)
with ‹*h* ⊴ *h'*› **have** *P*,*h'* ⊢ *w* :≤ *U* **by**(*rule conf-hext*)
ultimately have ∃ *T*. *P*,*h'* ⊢ *a*@*ACell* (*nat* (*sint i*)) : *T* ∧ *P*,*h'* ⊢ *w* :≤ *T* **by** *blast*
thus *?case* **using** *RedAAss*
by(*auto intro*!: *vs-confI split*: *if-split-asm dest*: *vs-confD simp add*: *take-Cons'*)(*blast dest*: *vs-confD hext-heap-write intro*: *addr-loc-type-hext-mono conf-hext*)+
next
case (*RedFAss h e D F v h' xs*)
hence ∃ *T*. *P*,*h'* ⊢ *e*@*CField D F* : *T* ∧ *P*,*h'* ⊢ *v* :≤ *T*
by(*force dest*!: *hext-heap-write intro*!: *addr-loc-type.intros intro*: *typeof-addr-hext-mono type-of-hext-type-of simp add*: *conf-def*)
thus *?case* **using** *RedFAss*
by(*auto intro*!: *vs-confI simp add*: *take-Cons' split*: *if-split-asm dest*: *vs-confD*)(*blast dest*: *vs-confD hext-heap-write intro*: *addr-loc-type-hext-mono conf-hext*)+
next
case (*RedCASSucceed h a D F v v' h' l*)
hence ∃ *T*. *P*,*h'* ⊢ *a*@*CField D F* : *T* ∧ *P*,*h'* ⊢ *v'* :≤ *T*
by(*force dest*!: *hext-heap-write intro*!: *addr-loc-type.intros intro*: *typeof-addr-hext-mono type-of-hext-type-of simp add*: *conf-def take-Cons'*)
thus *?case* **using** *RedCASSucceed*
by(*auto simp add*: *take-Cons' split*: *if-split-asm dest*: *vs-confD intro*!: *vs-confI*)
(*blast dest*: *vs-confD hext-heap-write intro*: *addr-loc-type-hext-mono conf-hext*)+
next
case *RedCallExternal* **thus** *?case* **by**(*auto intro*: *red-external-non-speculative-vs-conf*)
qed(*auto dest*: *vs-conf-allocate hext-allocate intro*: *vs-conf-hext simp add*: *take-Cons'*)

lemma *red-non-speculative-typeable*:
⟦ *convert-extTA extTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; *P*,*E*,*hp s* ⊢ *e* : *T*;
non-speculative P vs (*llist-of* (*map NormalAction* $\{\!|ta|\!\}_o$)); *vs-conf P* (*hp s*) *vs*; *hconf* (*hp s*) ⟧
$\Longrightarrow$ *J-heap-base.red addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr* (*heap-read-typed P*) *heap-write* (*convert-extTA extTA*) *P t e s ta e' s'*
and *reds-non-speculative-typeable*:
⟦ *convert-extTA extTA*,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *P*,*E*,*hp s* ⊢ *es* [:] *Ts*;
non-speculative P vs (*llist-of* (*map NormalAction* $\{\!|ta|\!\}_o$)); *vs-conf P* (*hp s*) *vs*; *hconf* (*hp s*) ⟧
$\Longrightarrow$ *J-heap-base.reds addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr* (*heap-read-typed P*) *heap-write* (*convert-extTA extTA*) *P t es s ta es' s'*
proof(*induct arbitrary*: *E T* **and** *E Ts rule*: *red-reds.inducts*)
case *RedCall* **thus** *?case* **by**(*blast intro*: *J-heap-base.red-reds.RedCall*)
next
case *RedCallExternal* **thus** *?case*
by(*auto intro*: *J-heap-base.red-reds.RedCallExternal red-external-non-speculative-typeable*)
qed(*auto intro*: *J-heap-base.red-reds.intros intro*!: *heap-read-typedI dest*: *vs-confD addr-loc-type-fun*)

end

sublocale *J-heap-base* < *red-mthr*:
if-multithreaded
final-expr
mred P
convert-RA
for *P*
by(*unfold-locales*)

locale *J-allocated-heap-conf* =
J-heap-conf
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write hconf
P
+
J-allocated-heap
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
allocated
P
for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *allocated* :: *'heap* ⇒ *'addr set*
and *P* :: *'addr J-prog*
begin

lemma *mred-known-addrs-typing*:
assumes *wf*: *wf-J-prog P*
and *ok*: *start-heap-ok*
shows *known-addrs-typing addr2thread-id thread-id2addr empty-heap allocate typeof-addr heap-write allocated J-known-addrs final-expr* (*mred P*) (λ*t x h*. ∃ *ET*. *sconf-type-ok ET t x h*) *P*
proof −
interpret *known-addrs*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
allocated J-known-addrs
final-expr mred P P
using *wf ok* **by**(*rule mred-known-addrs*)

show *?thesis*
proof
fix *t x m ta x' m'*
assume *mred P t* (*x*, *m*) *ta* (*x'*, *m'*)
thus *m* ⊴ *m'* **by**(*auto dest*: *red-hext-incr simp add*: *split-beta*)

next
fix *t x m ta x′ m′ vs*
assume *red*: *mred P t* (*x*, *m*) *ta* (*x′*, *m′*)
and *ts-ok*: ∃ *ET*. *sconf-type-ok ET t x m*
and *vs*: *vs-conf P m vs*
and *ns*: *non-speculative P vs* (*llist-of* (*map NormalAction* {|*ta*|}$_o$))

let *?mred* = *J-heap-base.mred addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr* (*heap-read-typed P*) *heap-write P*

have *lift*: *lifting-inv final-expr ?mred sconf-type-ok*
by(*intro J-conf-read.lifting-inv-sconf-subject-ok J-conf-read-heap-read-typed wf*)
moreover
from *ts-ok* **obtain** *ET* **where** *type*: *sconf-type-ok ET t x m* **..**
with *red vs ns* **have** *red′*: *?mred t* (*x*, *m*) *ta* (*x′*, *m′*)
by(*auto simp add*: *split-beta sconf-type-ok-def sconf-def type-ok-def dest*: *red-non-speculative-typeable*)
ultimately have *sconf-type-ok ET t x′ m′* **using** *type*
by(*rule lifting-inv.invariant-red*[**where** *r*=*?mred*])
thus ∃ *ET*. *sconf-type-ok ET t x′ m′* **..**
{ fix *t″ x″ m″*
assume *New*: *NewThread t″ x″ m″* ∈ *set* {|*ta*|}$_t$
with *red* **have** *m″* = *snd* (*x′*, *m′*) **by**(*rule red-mthr.new-thread-memory*)
with *lift red′ type New*
show ∃ *ET*. *sconf-type-ok ET t″ x″ m″*
by−(*rule lifting-inv.invariant-NewThread*[**where** *r*=*?mred*], *simp-all*) **}**
{ fix *t″ x″*
assume ∃ *ET*. *sconf-type-ok ET t″ x″ m*
with *lifting-inv.invariant-other*[**where** *r*=*?mred*, *OF lift red′ type*]
show ∃ *ET*. *sconf-type-ok ET t″ x″ m′* **by** *blast* **}**
next
fix *t x m ta x′ m′ vs n*
assume *red*: *mred P t* (*x*, *m*) *ta* (*x′*, *m′*)
and *ts-ok*: ∃ *ET*. *sconf-type-ok ET t x m*
and *vs*: *vs-conf P m vs*
and *ns*: *non-speculative P vs* (*llist-of* (*take n* (*map NormalAction* {|*ta*|}$_o$)))
thus *vs-conf P m′* (*w-values P vs* (*take n* (*map NormalAction* {|*ta*|}$_o$)))
by(*cases x*)(*auto dest*: *red-non-speculative-vs-conf simp add*: *sconf-type-ok-def type-ok-def sconf-def*)
next
fix *t x m ta x′ m′ ad al v*
assume *mred P t* (*x*, *m*) *ta* (*x′*, *m′*)
and ∃ *ET*. *sconf-type-ok ET t x m*
and *ReadMem ad al v* ∈ *set* {|*ta*|}$_o$
thus ∃ *T*. *P*,*m* ⊢ *ad*@*al* : *T*
by(*fastforce simp add*: *sconf-type-ok-def type-ok-def sconf-def split-beta dest*: *red-read-typeable*)
next
fix *t x m ta x′ m′ ad hT*
assume *mred P t* (*x*, *m*) *ta* (*x′*, *m′*)
and ∃ *ET*. *sconf-type-ok ET t x m*
and *NewHeapElem ad hT* ∈ *set* {|*ta*|}$_o$
thus *typeof-addr m′ ad* = ⌊*hT*⌋
by(*auto dest*: *red-New-typeof-addrD*[**where** *x*=*hT*] *dest*!: *WTrt-new-types-types simp add*: *split-beta sconf-type-ok-def sconf-def type-ok-def*)
qed
qed

end

context *J-allocated-heap-conf* **begin**

lemma *executions-sc*:
assumes *wf*: *wf-J-prog P*
and *wf-start*: *wf-start-state P C M vs*
and *vs2*: $\bigcup$(*ka-Val* ' *set vs*) $\subseteq$ *set start-addrs*
shows *executions-sc-hb* (*J-*$\mathcal{E}$ *P C M vs status*) *P*
(**is** *executions-sc-hb ?E P*)
proof −
from *wf-start* **obtain** *Ts T pns body D* **where** *ok*: *start-heap-ok*
and *sees*: *P* ⊢ *C sees M*:*Ts*→*T*=⌊(*pns*, *body*)⌋ *in D*
and *vs1*: *P*,*start-heap* ⊢ *vs* [:≤] *Ts* **by** *cases auto*

interpret *known-addrs-typing*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
allocated J-known-addrs
final-expr mred P λ*t x h*. ∃*ET*. *sconf-type-ok ET t x h P*
using *wf ok* **by**(*rule mred-known-addrs-typing*)

from *wf-prog-wf-syscls*[*OF wf*] *J-start-state-sconf-type-ok*[*OF wf wf-start*]
show *?thesis*
proof(*rule executions-sc-hb*)
from *wf sees* **have** *wf-mdecl wf-J-mdecl P D* (*M*, *Ts*, *T*, ⌊(*pns*, *body*)⌋) **by**(*rule sees-wf-mdecl*)
then obtain *T*′ **where** *len1*: *length pns* = *length Ts* **and** *wt*: *P*,[*this*↦*Class D*,*pns* [↦] *Ts*] ⊢ *body* :: *T*′
by(*auto simp add*: *wf-mdecl-def*)
from *vs1* **have** *len2*: *length vs* = *length Ts* **by**(*rule list-all2-lengthD*)
show *J-known-addrs start-tid* ((λ(*pns*, *body*) *vs*. (*blocks* (*this* # *pns*) (*Class* (*fst* (*method P C M*)) # *fst* (*snd* (*method P C M*))) (*Null* # *vs*) *body*, *Map.empty*)) (*the* (*snd* (*snd* (*snd* (*method P C M*))))) *vs*) ⊆ *allocated start-heap*
using *sees vs2 len1 len2 WT-ka*[*OF wt*]
by(*auto simp add*: *split-beta start-addrs-allocated ka-blocks intro*: *start-tid-start-addrs*[*OF wf-prog-wf-syscls*[*OF wf*] *ok*])
qed
qed

end

declare *split-paired-Ex* [*simp del*]

context *J-progress* **begin**

lemma *ex-WTrt-simps*:
P,*E*,*h* ⊢ *e* : *T* ⟹ ∃*E T*. *P*,*E*,*h* ⊢ *e* : *T*
by *blast*

abbreviation (*input*) *J-non-speculative-read-bound* :: *nat*
where *J-non-speculative-read-bound* ≡ *2*

lemma assumes *hrt*: *heap-read-typeable hconf P*
and *vs*: *vs-conf P* (*shr s*) *vs*
and *hconf*: *hconf* (*shr s*)
shows *red-non-speculative-read*:
⟦ *P*,*t* ⊢ ⟨*e*, (*shr s*, *xs*)⟩ −*ta*→ ⟨*e'*, (*h'*, *xs'*)⟩; ∃*E T*. *P*,*E*,*shr s* ⊢ *e* : *T*;
red-mthr.mthr.if.actions-ok s t ta;
I < *length* ⦃*ta*⦄*o*; ⦃*ta*⦄*o* ! *I* = *ReadMem a'' al'' v*; *v'* ∈ *w-values P vs* (*map NormalAction* (*take I* ⦃*ta*⦄*o*)) (*a''*, *al''*);
non-speculative P vs (*llist-of* (*map NormalAction* (*take I* ⦃*ta*⦄*o*))) ⟧
⟹ ∃*ta' e'' xs'' h''*. *P*,*t* ⊢ ⟨*e*, (*shr s*, *xs*)⟩ −*ta'*→ ⟨*e''*, (*h''*, *xs''*)⟩ ∧
red-mthr.mthr.if.actions-ok s t ta' ∧
I < *length* ⦃*ta'*⦄*o* ∧ *take I* ⦃*ta'*⦄*o* = *take I* ⦃*ta*⦄*o* ∧
⦃*ta'*⦄*o* ! *I* = *ReadMem a'' al'' v'* ∧ *length* ⦃*ta'*⦄*o* ≤ *max J-non-speculative-read-bound* (*length* ⦃*ta*⦄*o*)
and *reds-non-speculative-read*:
⟦ *P*,*t* ⊢ ⟨*es*, (*shr s*, *xs*)⟩ [−*ta*→] ⟨*es'*, (*h'*, *xs'*)⟩; ∃*E Ts*. *P*,*E*,*shr s* ⊢ *es* [:] *Ts*;
red-mthr.mthr.if.actions-ok s t ta;
I < *length* ⦃*ta*⦄*o*; ⦃*ta*⦄*o* ! *I* = *ReadMem a'' al'' v*; *v'* ∈ *w-values P vs* (*map NormalAction* (*take I* ⦃*ta*⦄*o*)) (*a''*, *al''*);
non-speculative P vs (*llist-of* (*map NormalAction* (*take I* ⦃*ta*⦄*o*))) ⟧
⟹ ∃*ta' es'' xs'' h''*. *P*,*t* ⊢ ⟨*es*, (*shr s*, *xs*)⟩ [−*ta'*→] ⟨*es''*, (*h''*, *xs''*)⟩ ∧
red-mthr.mthr.if.actions-ok s t ta' ∧
I < *length* ⦃*ta'*⦄*o* ∧ *take I* ⦃*ta'*⦄*o* = *take I* ⦃*ta*⦄*o* ∧
⦃*ta'*⦄*o* ! *I* = *ReadMem a'' al'' v'* ∧ *length* ⦃*ta'*⦄*o* ≤ *max J-non-speculative-read-bound* (*length* ⦃*ta*⦄*o*)
proof(*induct e hxs*≡(*shr s*, *xs*) *ta e' hxs'*≡(*h'*, *xs'*)
and *es hxs*≡(*shr s*, *xs*) *ta es' hxs'*≡(*h'*, *xs'*)
arbitrary: *xs xs'* **and** *xs xs' rule*: *red-reds.inducts*)
case (*RedAAcc a U n i v e*)
hence [*simp*]: *I* = *0 al''* = *ACell* (*nat* (*sint i*)) *a''* = *a*
and *v'*: *v'* ∈ *vs* (*a*, *ACell* (*nat* (*sint i*))) **by** *simp-all*
from *RedAAcc* **have** *adal*: *P*,*shr s* ⊢ *a*@*ACell* (*nat* (*sint i*)) : *U*
by(*auto intro*: *addr-loc-type.intros simp add*: *nat-less-iff word-sle-eq*)
from *v' vs adal* **have** *P*,*shr s* ⊢ *v'* :≤ *U* **by**(*auto dest*!: *vs-confD dest*: *addr-loc-type-fun*)
with *hrt adal* **have** *heap-read* (*shr s*) *a* (*ACell* (*nat* (*sint i*))) *v'* **using** *hconf* **by**(*rule heap-read-typeableD*)
with ‹*typeof-addr* (*shr s*) *a* = ⌊*Array-type U n*⌋› ‹*0* <=*s i*› ‹*sint i* < *int n*›
‹*red-mthr.mthr.if.actions-ok s t* ⦃*ReadMem a* (*ACell* (*nat* (*sint i*))) *v*⦄›
show *?case* **by**(*fastforce intro*: *red-reds.RedAAcc*)
next
case (*RedFAcc a D F v*)
hence [*simp*]: *I* = *0 al''* = *CField D F a''* = *a*
and *v'*: *v'* ∈ *vs* (*a*, *CField D F*) **by** *simp-all*
from *RedFAcc* **obtain** *E T* **where** *P*,*E*,*shr s* ⊢ *addr a*·*F*{*D*} : *T* **by** *blast*
with *RedFAcc* **have** *adal*: *P*,*shr s* ⊢ *a*@*CField D F* : *T* **by**(*auto 4 4 intro*: *addr-loc-type.intros*)
from *v' vs adal* **have** *P*,*shr s* ⊢ *v'* :≤ *T* **by**(*auto dest*!: *vs-confD dest*: *addr-loc-type-fun*)
with *hrt adal* **have** *heap-read* (*shr s*) *a* (*CField D F*) *v'* **using** *hconf* **by**(*rule heap-read-typeableD*)
with ‹*red-mthr.mthr.if.actions-ok s t* ⦃*ReadMem a* (*CField D F*) *v*⦄›
show *?case* **by**(*fastforce intro*: *red-reds.RedFAcc*)
next
case (*RedCASSucceed a D F v'' v'''*)
hence [*simp*]: *I* = *0 al''* = *CField D F a''* = *a v''* = *v*
and *v'*: *v'* ∈ *vs* (*a*, *CField D F*) **by**(*auto simp add*: *take-Cons' split*: *if-split-asm*)
from *RedCASSucceed.prems*(*1*) **obtain** *E T* **where**
P,*E*,*shr s* ⊢ *addr a*·*compareAndSwap*(*D*·*F*, *Val v''*, *Val v'''*) : *T* **by** *clarify*

then obtain T **where** *adal*: $P,shr\ s \vdash a@CField\ D\ F : T$
and v'': $P,shr\ s \vdash v'' :\leq T$ **and** v''': $P,shr\ s \vdash v''' :\leq T$
by(*fastforce intro*: *addr-loc-type.intros simp add*: *conf-def*)
from v' *vs adal* **have** $P,shr\ s \vdash v' :\leq T$ **by**(*auto dest*!: *vs-confD dest*: *addr-loc-type-fun*)
from *hrt adal this hconf* **have** *read*: *heap-read* (*shr s*) *a* (*CField D F*) v' **by**(*rule heap-read-typeableD*)
show *?case*
proof(*cases* $v' = v''$)
case *True*
then show *?thesis* **using** *RedCASSucceed*
by(*fastforce intro*: *red-reds.RedCASSucceed*)
next
case *False*
then show *?thesis* **using** *read RedCASSucceed*
by(*fastforce intro*: *RedCASFail*)
qed
next
case (*RedCASFail a D F* v'' v''' v'''')
hence [*simp*]: $I = 0\ al'' = CField\ D\ F\ a'' = a\ v'' = v$
and v': $v' \in vs\ (a,\ CField\ D\ F)$ **by**(*auto simp add*: *take-Cons' split*: *if-split-asm*)
from *RedCASFail.prems*(*1*) **obtain** *E T* **where**
$P,E,shr\ s \vdash addr\ a{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v''',\ Val\ v'''') : T$ **by**(*iprover*)
then obtain T **where** *adal*: $P,shr\ s \vdash a@CField\ D\ F : T$
and v''': $P,shr\ s \vdash v''' :\leq T$ **and** v'''': $P,shr\ s \vdash v'''' :\leq T$
by(*fastforce intro*: *addr-loc-type.intros simp add*: *conf-def*)
from v' *vs adal* **have** $P,shr\ s \vdash v' :\leq T$ **by**(*auto dest*!: *vs-confD dest*: *addr-loc-type-fun*)
from *hrt adal this hconf* **have** *read*: *heap-read* (*shr s*) *a* (*CField D F*) v' **by**(*rule heap-read-typeableD*)
show *?case*
proof(*cases* $v' = v'''$)
case *True*
from *heap-write-total*[*OF hconf adal* v''''] **obtain** h' **where**
heap-write (*shr s*) *a* (*CField D F*) v'''' h' **..**
with *read RedCASFail True* **show** *?thesis*
by(*fastforce intro*: *RedCASSucceed*)
next
case *False*
with *read RedCASFail* **show** *?thesis* **by**(*fastforce intro*: *red-reds.RedCASFail*)
qed
next
case (*RedCallExternal a U M Ts Tr D ps ta'* *va h' ta e'*)
from $\langle P,t \vdash \langle a{\cdot}M(ps),hp\ (shr\ s,\ xs)\rangle\ -ta'\!\!\rightarrow ext\ \langle va,h'\rangle\rangle$
have *red*: $P,t \vdash \langle a{\cdot}M(ps),shr\ s\rangle\ -ta'\!\!\rightarrow ext\ \langle va,h'\rangle$ **by** *simp*
from *RedCallExternal* **have** *aok*: *red-mthr.mthr.if.actions-ok s t ta'* **by** *simp*
from *RedCallExternal* **have** $I < length\ \{\!|ta'|\!\}_o$
and $\{\!|ta'|\!\}_o\ !\ I = ReadMem\ a''\ al''\ v$
and $v' \in w\text{-}values\ P\ vs\ (map\ NormalAction\ (take\ I\ \{\!|ta'|\!\}_o))\ (a'',\ al'')$
and *non-speculative P vs* (*llist-of* (*map NormalAction* (*take I* $\{\!|ta'|\!\}_o$))) **by** *simp-all*
from *red-external-non-speculative-read*[*OF hrt vs red aok hconf this*]
$\langle typeof\text{-}addr\ (hp\ (shr\ s,\ xs))\ a = \lfloor U \rfloor\rangle$
$\langle P \vdash class\text{-}type\text{-}of\ U\ sees\ M: Ts\!\rightarrow\!Tr = Native\ in\ D\rangle\ \langle ta = extTA2J\ P\ ta'\rangle$
$\langle I < length\ \{\!|ta|\!\}_o\rangle$
show *?case* **by**(*fastforce intro*: *red-reds.RedCallExternal*)
next
case *NewArrayRed* **thus** *?case* **by**(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.NewArrayRed*)

next
case *CastRed* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.CastRed*)
next
case *InstanceOfRed* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.InstanceOfRed*)
next
case *BinOpRed1* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.BinOpRed1*)
next
case *BinOpRed2* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.BinOpRed2*)
next
case *LAssRed* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.LAssRed*)
next
case *AAccRed1* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.AAccRed1*)
next
case *AAccRed2* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.AAccRed2*)
next
case *AAssRed1* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.AAssRed1*)
next
case *AAssRed2* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.AAssRed2*)
next
case *AAssRed3* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.AAssRed3*)+
next
case *ALengthRed* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.ALengthRed*)
next
case *FAccRed* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.FAccRed*)
next
case *FAssRed1* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.FAssRed1*)
next
case *FAssRed2* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.FAssRed2*)
next
case *CASRed1* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.CASRed1*)
next
case *CASRed2* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.CASRed2*)
next
case *CASRed3* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.CASRed3*)
next
case *CallObj* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.CallObj*)

next
case *CallParams* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.CallParams*)
next
case *BlockRed* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*fastforce intro*: *red-reds.BlockRed*)+
next
case *SynchronizedRed1* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.SynchronizedRed1*)
next
case *SynchronizedRed2* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.SynchronizedRed2*)
next
case *SeqRed* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.SeqRed*)
next
case *CondRed* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.CondRed*)
next
case *ThrowRed* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.ThrowRed*)
next
case *TryRed* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.TryRed*)
next
case *ListRed1* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.ListRed1*)
next
case *ListRed2* **thus** *?case*
by(*clarsimp simp add*: *split-paired-Ex ex-WTrt-simps*)(*blast intro*: *red-reds.ListRed2*)
qed(*simp-all*)

end

sublocale *J-allocated-heap-conf* < *if-known-addrs-base*
J-known-addrs
final-expr mred P convert-RA
.

declare *split-paired-Ex* [*simp*]
declare *eq-upto-seq-inconsist-simps* [*simp del*]

locale *J-allocated-progress* =
J-progress
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write hconf
P
+
J-allocated-heap-conf
addr2thread-id thread-id2addr
spurious-wakeups

empty-heap allocate typeof-addr heap-read heap-write hconf
allocated
P
for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *allocated* :: *'heap* ⇒ *'addr set*
and *P* :: *'addr J-prog*
begin

lemma *non-speculative-read*:
assumes *wf*: *wf-J-prog P*
and *hrt*: *heap-read-typeable hconf P*
and *wf-start*: *wf-start-state P C M vs*
and *ka*: ⋃(*ka-Val* ‘ *set vs*) ⊆ *set start-addrs*
shows *red-mthr.if.non-speculative-read J-non-speculative-read-bound*
(*init-fin-lift-state status* (*J-start-state P C M vs*))
(*w-values P* (λ-. {}) (*map snd* (*lift-start-obs start-tid start-heap-obs*)))
(**is** *red-mthr.if.non-speculative-read - ?start-state ?start-vs*)
proof(*rule red-mthr.if.non-speculative-readI*)
fix *ttas s′ t x ta x′ m′ i ad al v v′*
assume *τRed*: *red-mthr.mthr.if.RedT P ?start-state ttas s′*
and *sc*: *non-speculative P ?start-vs* (*llist-of* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*)))
and *ts′t*: *thr s′ t* = ⌊(*x*, *no-wait-locks*)⌋
and *red*: *red-mthr.init-fin P t* (*x*, *shr s′*) *ta* (*x′*, *m′*)
and *aok*: *red-mthr.mthr.if.actions-ok s′ t ta*
and *i*: *i* < *length* {|*ta*|}$_o$
and *ns′*: *non-speculative P* (*w-values P ?start-vs* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*))) (*llist-of* (*take i* {|*ta*|}$_o$))
and *read*: {|*ta*|}$_o$! *i* = *NormalAction* (*ReadMem ad al v*)
and *v′*: *v′* ∈ *w-values P ?start-vs* (*concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*) @ *take i* {|*ta*|}$_o$) (*ad*, *al*)

from *wf-start* **obtain** *Ts T pns body D* **where** *ok*: *start-heap-ok*
and *sees*: *P* ⊢ *C sees M*:*Ts*→*T* = ⌊(*pns*, *body*)⌋ *in D*
and *conf*: *P*,*start-heap* ⊢ *vs* [:≤] *Ts* **by** *cases auto*

let *?conv* = λ*ttas*. *concat* (*map* (λ(*t*, *ta*). {|*ta*|}$_o$) *ttas*)
let *?vs′* = *w-values P ?start-vs* (*?conv ttas*)
let *?wt-ok* = *init-fin-lift-inv sconf-type-ok*
let *?ET-start* = *J-sconf-type-ET-start P C M*
let *?start-obs* = *map snd* (*lift-start-obs start-tid start-heap-obs*)
let *?start-state* = *init-fin-lift-state status* (*J-start-state P C M vs*)

interpret *known-addrs-typing*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
allocated J-known-addrs

final-expr mred P $\lambda t\ x\ h.\ \exists ET.$ *sconf-type-ok ET t x h P*
using *wf ok* **by**(*rule mred-known-addrs-typing*)

from *wf sees* **have** *wf-mdecl wf-J-mdecl P D* (*M*, *Ts*, *T*, ⌊(*pns*, *body*)⌋) **by**(*rule sees-wf-mdecl*)
then obtain *T′* **where** *len1*: *length pns* = *length Ts* **and** *wt*: *P*,[*this*↦*Class D*,*pns* [↦] *Ts*] ⊢ *body* :: *T′*
by(*auto simp add*: *wf-mdecl-def*)
from *conf* **have** *len2*: *length vs* = *length Ts* **by**(*rule list-all2-lengthD*)

from *wf wf-start* **have** *ts-ok-start*: *ts-ok* (*init-fin-lift* ($\lambda t\ x\ h.\ \exists ET.$ *sconf-type-ok ET t x h*)) (*thr ?start-state*) (*shr ?start-state*)
unfolding *ts-ok-init-fin-lift-init-fin-lift-state shr-start-state* **by**(*rule J-start-state-sconf-type-ok*)
have *sc′*: *non-speculative P ?start-vs* (*lmap snd* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) (*llist-of ttas*))))
using *sc* **by**(*simp add*: *lmap-lconcat llist.map-comp o-def split-def lconcat-llist-of*[*symmetric*])

from *start-state-vs-conf*[*OF wf-prog-wf-syscls*[*OF wf*]]
have *vs-conf-start*: *vs-conf P* (*shr ?start-state*) *?start-vs*
by(*simp add*:*init-fin-lift-state-conv-simps start-state-def split-beta*)
with *τRed ts-ok-start sc*
have *wt′*: *ts-ok* (*init-fin-lift* ($\lambda t\ x\ h.\ \exists ET.$ *sconf-type-ok ET t x h*)) (*thr s′*) (*shr s′*)
and *vs′*: *vs-conf P* (*shr s′*) *?vs′* **by**(*rule if-RedT-non-speculative-invar*)+

from *red i read* **obtain** *e xs e′ xs′ ta′*
where *x*: *x* = (*Running*, *e*, *xs*) **and** *x′*: *x′* = (*Running*, *e′*, *xs′*)
and *ta*: *ta* = *convert-TA-initial* (*convert-obs-initial ta′*)
and *red′*: *P*,*t* ⊢ ⟨*e*, (*shr s′*, *xs*)⟩ −*ta′*→ ⟨*e′*, (*m′*, *xs′*)⟩
by *cases fastforce*+

from *ts′t wt′ x* **obtain** *E T* **where** *wte*: *P*,*E*,*shr s′* ⊢ *e* : *T*
and *hconf*: *hconf* (*shr s′*)
by(*auto dest*!: *ts-okD simp add*: *sconf-type-ok-def sconf-def type-ok-def*)

have *aok′*: *red-mthr.mthr.if.actions-ok s′ t ta′* **using** *aok* **unfolding** *ta* **by** *simp*

from *i read v′ ta ns′* **have** *i* < *length* {|*ta′*|}$_o$ **and** {|*ta′*|}$_o$! *i* = *ReadMem ad al v*
and *v′* ∈ *w-values P ?vs′* (*map NormalAction* (*take i* {|*ta′*|}$_o$)) (*ad*, *al*)
and *non-speculative P ?vs′* (*llist-of* (*map NormalAction* (*take i* {|*ta′*|}$_o$)))
by(*simp-all add*: *take-map*)

from *red-non-speculative-read*[*OF hrt vs′ hconf red′ - aok′ this*] *wte*
obtain *ta″ e″ xs″ h″*
where *red″*: *P*,*t* ⊢ ⟨*e*, (*shr s′*, *xs*)⟩ −*ta″*→ ⟨*e″*, (*h″*, *xs″*)⟩
and *aok″*: *red-mthr.mthr.if.actions-ok s′ t ta″*
and *i″*: *i* < *length* {|*ta″*|}$_o$
and *eq″*: *take i* {|*ta″*|}$_o$ = *take i* {|*ta′*|}$_o$
and *read″*: {|*ta″*|}$_o$! *i* = *ReadMem ad al v′*
and *len″*: *length* {|*ta″*|}$_o$ ≤ *max J-non-speculative-read-bound* (*length* {|*ta′*|}$_o$) **by** *blast*

let *?x′* = (*Running*, *e″*, *xs″*)
let *?ta′* = *convert-TA-initial* (*convert-obs-initial ta″*)
from *red″* **have** *red-mthr.init-fin P t* (*x*, *shr s′*) *?ta′* (*?x′*, *h″*)
unfolding *x* **by** −(*rule red-mthr.init-fin.NormalAction*, *simp*)
moreover from *aok″* **have** *red-mthr.mthr.if.actions-ok s′ t ?ta′* **by** *simp*

```
  moreover from i'' have i < length ⦃?ta'⦄_o by simp
  moreover from eq'' have take i ⦃?ta'⦄_o = take i ⦃ta⦄_o unfolding ta by(simp add: take-map)
  moreover from read'' i'' have ⦃?ta'⦄_o ! i = NormalAction (ReadMem ad al v') by(simp add:
nth-map)
  moreover from len'' have length ⦃?ta'⦄_o ≤ max J-non-speculative-read-bound (length ⦃ta⦄_o)
    unfolding ta by simp
  ultimately
  show ∃ta' x'' m''. red-mthr.init-fin P t (x, shr s') ta' (x'', m'') ∧
                     red-mthr.mthr.if.actions-ok s' t ta' ∧
                     i < length ⦃ta'⦄_o ∧ take i ⦃ta'⦄_o = take i ⦃ta⦄_o ∧
                     ⦃ta'⦄_o ! i = NormalAction (ReadMem ad al v') ∧
                     length ⦃ta'⦄_o ≤ max J-non-speculative-read-bound (length ⦃ta⦄_o)
    by blast
qed

lemma J-cut-and-update:
  assumes wf: wf-J-prog P
  and hrt: heap-read-typeable hconf P
  and wf-start: wf-start-state P C M vs
  and ka: ⋃(ka-Val ' set vs) ⊆ set start-addrs
  shows red-mthr.if.cut-and-update (init-fin-lift-state status (J-start-state P C M vs))
        (mrw-values P Map.empty (map snd (lift-start-obs start-tid start-heap-obs)))
proof −
  from wf-start obtain Ts T pns body D where ok: start-heap-ok
    and sees: P ⊢ C sees M: Ts→T = ⌊(pns, body)⌋ in D
    and conf: P,start-heap ⊢ vs [:≤] Ts by cases auto

  interpret known-addrs-typing
    addr2thread-id thread-id2addr
    spurious-wakeups
    empty-heap allocate typeof-addr heap-read heap-write
    allocated J-known-addrs
    final-expr mred P λt x h. ∃ET. sconf-type-ok ET t x h P
    using wf ok by(rule mred-known-addrs-typing)

  let ?start-vs = w-values P (λ-. {}) (map snd (lift-start-obs start-tid start-heap-obs))
  let ?wt-ok = init-fin-lift-inv sconf-type-ok
  let ?ET-start = J-sconf-type-ET-start P C M
  let ?start-obs = map snd (lift-start-obs start-tid start-heap-obs)
  let ?start-state = init-fin-lift-state status (J-start-state P C M vs)

  from wf sees have wf-mdecl wf-J-mdecl P D (M, Ts, T, ⌊(pns, body)⌋) by(rule sees-wf-mdecl)
  then obtain T' where len1: length pns = length Ts and wt: P,[this↦Class D,pns [↦] Ts] ⊢ body
:: T'
    by(auto simp add: wf-mdecl-def)
  from conf have len2: length vs = length Ts by(rule list-all2-lengthD)

  note wf-prog-wf-syscls[OF wf] non-speculative-read[OF wf hrt wf-start ka]
  moreover
  from wf wf-start have ts-ok-start: ts-ok (init-fin-lift (λt x h. ∃ET. sconf-type-ok ET t x h)) (thr
?start-state) (shr ?start-state)
    unfolding ts-ok-init-fin-lift-init-fin-lift-state shr-start-state by(rule J-start-state-sconf-type-ok)
  moreover
  have ka: J-known-addrs start-tid ((λ(pns, body) vs. (blocks (this # pns) (Class (fst (method P C
```

M)) # *fst* (*snd* (*method P C M*))) (*Null* # *vs*) *body*, *Map.empty*)) (*the* (*snd* (*snd* (*snd* (*method P C M*))))) *vs*) ⊆ *allocated start-heap*
using *sees ka len1 len2 WT-ka*[*OF wt*]
by(*auto simp add*: *split-beta start-addrs-allocated ka-blocks intro*: *start-tid-start-addrs*[*OF wf-prog-wf-syscls*[*OF wf*] *ok*])
ultimately show *?thesis* **by**(*rule non-speculative-read-into-cut-and-update*)
qed

lemma *J-drf*:
assumes *wf*: *wf-J-prog P*
and *hrt*: *heap-read-typeable hconf P*
and *wf-start*: *wf-start-state P C M vs*
and *ka*: ⋃(*ka-Val* ‘ *set vs*) ⊆ *set start-addrs*
shows *drf* (*J-ℰ P C M vs status*) *P*
proof −
from *wf-start* **obtain** *Ts T pns body D* **where** *ok*: *start-heap-ok*
and *sees*: *P* ⊢ *C sees M*: *Ts*→*T* = ⌊(*pns*, *body*)⌋ *in D*
and *conf*: *P*,*start-heap* ⊢ *vs* [:≤] *Ts* **by** *cases auto*

from *J-cut-and-update*[*OF assms*] *wf-prog-wf-syscls*[*OF wf*] *J-start-state-sconf-type-ok*[*OF wf wf-start*]
show *?thesis*
proof(*rule known-addrs-typing.drf*[*OF mred-known-addrs-typing*[*OF wf ok*]])
from *wf sees* **have** *wf-mdecl wf-J-mdecl P D* (*M*, *Ts*, *T*, ⌊(*pns*, *body*)⌋) **by**(*rule sees-wf-mdecl*)
then obtain *T′* **where** *len1*: *length pns* = *length Ts* **and** *wt*: *P*,[*this*↦*Class D*,*pns* [↦] *Ts*] ⊢ *body* :: *T′*
by(*auto simp add*: *wf-mdecl-def*)
from *conf* **have** *len2*: *length vs* = *length Ts* **by**(*rule list-all2-lengthD*)
show *J-known-addrs start-tid* ((λ(*pns*, *body*) *vs*. (*blocks* (*this* # *pns*) (*Class* (*fst* (*method P C M*)) # *fst* (*snd* (*method P C M*))) (*Null* # *vs*) *body*, *Map.empty*)) (*the* (*snd* (*snd* (*snd* (*method P C M*))))) *vs*) ⊆ *allocated start-heap*
using *sees ka len1 len2 WT-ka*[*OF wt*]
by(*auto simp add*: *split-beta start-addrs-allocated ka-blocks intro*: *start-tid-start-addrs*[*OF wf-prog-wf-syscls*[*OF wf*] *ok*])
qed
qed

lemma *J-sc-legal*:
assumes *wf*: *wf-J-prog P*
and *hrt*: *heap-read-typeable hconf P*
and *wf-start*: *wf-start-state P C M vs*
and *ka*: ⋃(*ka-Val* ‘ *set vs*) ⊆ *set start-addrs*
shows *sc-legal* (*J-ℰ P C M vs status*) *P*
proof −
from *wf-start* **obtain** *Ts T pns body D* **where** *ok*: *start-heap-ok*
and *sees*: *P* ⊢ *C sees M*: *Ts*→*T* = ⌊(*pns*, *body*)⌋ *in D*
and *conf*: *P*,*start-heap* ⊢ *vs* [:≤] *Ts* **by** *cases auto*
interpret *known-addrs-typing*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
allocated J-known-addrs
final-expr mred P λ*t x h*. ∃ *ET*. *sconf-type-ok ET t x h P*
using *wf ok* **by**(*rule mred-known-addrs-typing*)

let *?start-vs* = *w-values P* (λ-. {}) (*map snd* (*lift-start-obs start-tid start-heap-obs*))
let *?wt-ok* = *init-fin-lift-inv sconf-type-ok*
let *?ET-start* = *J-sconf-type-ET-start P C M*
let *?start-obs* = *map snd* (*lift-start-obs start-tid start-heap-obs*)
let *?start-state* = *init-fin-lift-state status* (*J-start-state P C M vs*)

from *wf sees* **have** *wf-mdecl wf-J-mdecl P D* (*M*, *Ts*, *T*, ⌊(*pns*, *body*)⌋) **by**(*rule sees-wf-mdecl*)
then obtain *T′* **where** *len1*: *length pns* = *length Ts* **and** *wt*: *P*,[*this*↦*Class D*,*pns* [↦] *Ts*] ⊢ *body* :: *T′*
by(*auto simp add*: *wf-mdecl-def*)
from *conf* **have** *len2*: *length vs* = *length Ts* **by**(*rule list-all2-lengthD*)

note *wf-prog-wf-syscls*[*OF wf*] *non-speculative-read*[*OF wf hrt wf-start ka*]
moreover
from *wf wf-start* **have** *ts-ok-start*: *ts-ok* (*init-fin-lift* (λ*t x h*. ∃*ET*. *sconf-type-ok ET t x h*)) (*thr ?start-state*) (*shr ?start-state*)
unfolding *ts-ok-init-fin-lift-init-fin-lift-state shr-start-state* **by**(*rule J-start-state-sconf-type-ok*)
moreover have *ka-allocated*: *J-known-addrs start-tid* ((λ(*pns*, *body*) *vs*. (*blocks* (*this* # *pns*) (*Class* (*fst* (*method P C M*)) # *fst* (*snd* (*method P C M*))) (*Null* # *vs*) *body*, *Map.empty*)) (*the* (*snd* (*snd* (*snd* (*method P C M*))))) *vs*) ⊆ *allocated start-heap*
using *sees ka len1 len2 WT-ka*[*OF wt*]
by(*auto simp add*: *split-beta start-addrs-allocated ka-blocks intro*: *start-tid-start-addrs*[*OF wf-prog-wf-syscls*[*OF wf*] *ok*])
ultimately have *red-mthr.if.hb-completion ?start-state* (*lift-start-obs start-tid start-heap-obs*)
by(*rule non-speculative-read-into-hb-completion*)

thus *?thesis* **using** *wf-prog-wf-syscls*[*OF wf*] *J-start-state-sconf-type-ok*[*OF wf wf-start*]
by(*rule sc-legal*)(*rule ka-allocated*)
qed

lemma *J-jmm-consistent*:
assumes *wf*: *wf-J-prog P*
and *hrt*: *heap-read-typeable hconf P*
and *wf-start*: *wf-start-state P C M vs*
and *ka*: ⋃(*ka-Val* ‘ *set vs*) ⊆ *set start-addrs*
shows *jmm-consistent* (*J-*$\mathcal{E}$ *P C M vs status*) *P*
(**is** *jmm-consistent ?*$\mathcal{E}$ *P*)
proof −
interpret *drf ?*$\mathcal{E}$ *P* **using** *assms* **by**(*rule J-drf*)
interpret *sc-legal ?*$\mathcal{E}$ *P* **using** *assms* **by**(*rule J-sc-legal*)
show *?thesis* **by** *unfold-locales*
qed

lemma *J-ex-sc-exec*:
assumes *wf*: *wf-J-prog P*
and *hrt*: *heap-read-typeable hconf P*
and *wf-start*: *wf-start-state P C M vs*
and *ka*: ⋃(*ka-Val* ‘ *set vs*) ⊆ *set start-addrs*
shows ∃*E ws*. *E* ∈ *J-*$\mathcal{E}$ *P C M vs status* ∧ *P* ⊢ (*E*, *ws*) √ ∧ *sequentially-consistent P* (*E*, *ws*)
(**is** ∃*E ws*. - ∈ *?*$\mathcal{E}$ ∧ -)
proof −
interpret *jmm*: *executions-sc-hb ?*$\mathcal{E}$ *P* **using** *assms* **by** −(*rule executions-sc*)

let *?start-state* = *init-fin-lift-state status* (*J-start-state P C M vs*)

let *?start-mrw* = *mrw-values P Map.empty* (*map snd* (*lift-start-obs start-tid start-heap-obs*))

from *red-mthr.if.sequential-completion-Runs*[*OF red-mthr.if.cut-and-update-imp-sc-completion*[*OF J-cut-and-update*[*OF assms*]] *ta-seq-consist-convert-RA*]
obtain *ttas* **where** *Red*: *red-mthr.mthr.if.mthr.Runs P ?start-state ttas*
and *sc*: *ta-seq-consist P ?start-mrw* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* ⦃*ta*⦄$_o$) *ttas*)) **by** *blast*
let *?E* = *lappend* (*llist-of* (*lift-start-obs start-tid start-heap-obs*)) (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) ⦃*ta*⦄$_o$)) *ttas*))
from *Red* **have** *?E* ∈ *?E* **by**(*blast intro*: *red-mthr.mthr.if.E.intros*)
moreover from *Red* **have** *tsa*: *thread-start-actions-ok ?E*
by(*blast intro*: *red-mthr.thread-start-actions-ok-init-fin red-mthr.mthr.if.E.intros*)
from *sc* **have** *ta-seq-consist P Map.empty* (*lmap snd ?E*)
unfolding *lmap-lappend-distrib lmap-lconcat llist.map-comp split-def o-def lmap-llist-of map-map snd-conv*
by(*simp add*: *ta-seq-consist-lappend ta-seq-consist-start-heap-obs*)
from *ta-seq-consist-imp-sequentially-consistent*[*OF tsa jmm.E-new-actions-for-fun*[*OF* ‹*?E* ∈ *?E*›] *this*]
obtain *ws* **where** *sequentially-consistent P* (*?E*, *ws*) *P* ⊢ (*?E*, *ws*) √ **by** *iprover*
ultimately show *?thesis* **by** *blast*
qed

theorem *J-consistent*:
assumes *wf*: *wf-J-prog P*
and *hrt*: *heap-read-typeable hconf P*
and *wf-start*: *wf-start-state P C M vs*
and *ka*: ⋃(*ka-Val* ‘ *set vs*) ⊆ *set start-addrs*
shows ∃ *E ws. legal-execution P* (*J-E P C M vs status*) (*E*, *ws*)
proof −
let *?E* = *J-E P C M vs status*
interpret *sc-legal ?E P* **using** *assms* **by**(*rule J-sc-legal*)
from *J-ex-sc-exec*[*OF assms*]
obtain *E ws* **where** *E* ∈ *?E P* ⊢ (*E*, *ws*) √ *sequentially-consistent P* (*E*, *ws*) **by** *blast*
hence *legal-execution P ?E* (*E*, *ws*) **by**(*rule SC-is-legal*)
thus *?thesis* **by** *blast*
qed

end

end
theory *JMM-JVM*
imports
JMM-Framework
../JVM/JVMThreaded
begin

sublocale *JVM-heap-base* < *execd-mthr*:
heap-multithreaded-base
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
JVM-final mexecd P convert-RA
for *P*
.

context *JVM-heap-base* **begin**

abbreviation *JVMd-*$\mathcal{E}$ ::
'addr jvm-prog ⇒ *cname* ⇒ *mname* ⇒ *'addr val list* ⇒ *status*
⇒ (*'thread-id* × (*'addr*, *'thread-id*) *obs-event action*) *llist set*
where *JVMd-*$\mathcal{E}$ *P* ≡ *execd-mthr.*$\mathcal{E}$*-start P JVM-local-start P*

end

end

8.15 JMM Instantiation for bytecode

theory *DRF-JVM*
imports
JMM-Common
JMM-JVM
../BV/BVProgressThreaded
SC-Legal
begin

8.15.1 DRF guarantee for the JVM

abbreviation (*input*) *ka-xcp* :: *'addr option* ⇒ *'addr set*
where *ka-xcp* ≡ *set-option*

primrec *jvm-ka* :: *'addr jvm-thread-state* ⇒ *'addr set*
where
jvm-ka (*xcp*, *frs*) =
ka-xcp xcp ∪ (⋃(*stk*, *loc*, *C*, *M*, *pc*) ∈ *set frs*. (⋃ *v* ∈ *set stk*. *ka-Val v*) ∪ (⋃ *v* ∈ *set loc*. *ka-Val v*))

context *heap* **begin**

lemma *red-external-aggr-read-mem-typeable*:
⟦ (*ta*, *va*, *h'*) ∈ *red-external-aggr P t a M vs h*; *ReadMem ad al v* ∈ *set* {|*ta*|}$_o$ ⟧
⟹ ∃ *T'*. *P*,*h* ⊢ *ad*@*al* : *T'*
by(*auto simp add*: *red-external-aggr-def split-beta split*: *if-split-asm dest*: *heap-clone-read-typeable*)

end

context *JVM-heap-base* **begin**

definition *jvm-known-addrs* :: *'thread-id* ⇒ *'addr jvm-thread-state* ⇒ *'addr set*
where *jvm-known-addrs t xcpfrs* = {*thread-id2addr t*} ∪ *jvm-ka xcpfrs* ∪ *set start-addrs*

end

context *JVM-heap* **begin**

lemma *exec-instr-known-addrs*:
assumes *ok*: *start-heap-ok*
and *exec*: (*ta*, *xcp'*, *h'*, *frs'*) ∈ *exec-instr i P t h stk loc C M pc frs*
and *check*: *check-instr i P h stk loc C M pc frs*

shows *jvm-known-addrs t (xcp', frs') ⊆ jvm-known-addrs t (None, (stk, loc, C, M, pc) # frs) ∪ new-obs-addrs {|ta|}$_o$*
proof −

note [*simp*] = *jvm-known-addrs-def new-obs-addrs-def addr-of-sys-xcpt-start-addr*[*OF ok*] *subset-Un1 subset-Un2 subset-insert ka-Val-subset-new-obs-Addr-ReadMem SUP-subset-mono split-beta neq-Nil-conv tl-conv-drop set-drop-subset is-Ref-def*

from *exec check* **show** *?thesis*
proof(*cases i*)
case *Load* **with** *exec check* **show** *?thesis* **by** *auto*
next
case (*Store V*) **with** *exec check* **show** *?thesis*
using *set-update-subset-insert*[*of loc V*]
by(*clarsimp simp del*: *set-update-subsetI*) *blast*
next
case (*Push v*)
with *check* **have** *ka-Val v* = {} **by**(*cases v*) *simp-all*
with *Push exec check* **show** *?thesis* **by**(*simp*)
next
case (*CAS F D*)
then show *?thesis* **using** *exec check*
by(*clarsimp split*: *if-split-asm*)(*fastforce dest*!: *in-set-dropD*)+
next
case (*Invoke M' n*)
show *?thesis*
proof(*cases stk* ! *n* = *Null*)
case *True* **with** *exec check Invoke* **show** *?thesis* **by**(*simp*)
next
case [*simp*]: *False*
with *check Invoke* **obtain** *a* **where** *stkn*: *stk* ! *n* = *Addr a n* < *length stk* **by** *auto*
hence *a*: *a* ∈ (⋃ *v* ∈ *set stk*. *ka-Val v*) **by**(*fastforce dest*: *nth-mem*)
show *?thesis*
proof(*cases snd* (*snd* (*snd* (*method P* (*class-type-of* (*the* (*typeof-addr h* (*the-Addr* (*stk* ! *n*)))))
M'))) = *Native*)
case *True*
with *exec check Invoke a stkn* **show** *?thesis*
apply *clarsimp*
apply(*drule red-external-aggr-known-addrs-mono*[*OF ok*], *simp*)
apply(*auto dest*!: *in-set-takeD dest*: *bspec subsetD split*: *extCallRet.split-asm simp add*:
has-method-def is-native.simps)
done
next
case *False*
with *exec check Invoke a stkn* **show** *?thesis*
by(*auto simp add*: *set-replicate-conv-if dest*!: *in-set-takeD*)
qed
qed
next
case *Swap* **with** *exec check* **show** *?thesis*
by(*cases stk*)(*simp*, *case-tac list*, *auto*)
next
case (*BinOpInstr bop*) **with** *exec check* **show** *?thesis*
using *binop-known-addrs*[*OF ok*, *of bop hd* (*drop* (*Suc 0*) *stk*) *hd stk*]

```
    apply(cases stk)
    apply(simp, case-tac list, simp)
    apply clarsimp
    apply(drule (2) binop-progress)
    apply(auto 6 2 split: sum.split-asm)
    done
  next
   case MExit with exec check show ?thesis by(auto split: if-split-asm)
  qed(clarsimp split: if-split-asm)+
qed

lemma exec-d-known-addrs-mono:
  assumes ok: start-heap-ok
  and exec: mexecd P t (xcpfrs, h) ta (xcpfrs', h')
  shows jvm-known-addrs t xcpfrs' ⊆ jvm-known-addrs t xcpfrs ∪ new-obs-addrs {|ta|}_o
using exec
apply(cases xcpfrs)
apply(cases xcpfrs')
apply(simp add: split-beta)
apply(erule jvmd-NormalE)
apply(cases fst xcpfrs)
 apply(fastforce simp add: check-def split-beta del: subsetI dest!: exec-instr-known-addrs[OF ok])
apply(fastforce simp add: jvm-known-addrs-def split-beta dest!: in-set-dropD)
done

lemma exec-instr-known-addrs-ReadMem:
  assumes exec: (ta, xcp', h', frs') ∈ exec-instr i P t h stk loc C M pc frs
  and check: check-instr i P h stk loc C M pc frs
  and read: ReadMem ad al v ∈ set {|ta|}_o
  shows ad ∈ jvm-known-addrs t (None, (stk, loc, C, M, pc) # frs)
using assms
proof(cases i)
  case ALoad thus ?thesis using assms
     by(cases stk)(case-tac [2] list, auto simp add: split-beta is-Ref-def jvm-known-addrs-def split:
if-split-asm)
next
  case (Invoke M n)
  with check have stk ! n ≠ Null ⟶ the-Addr (stk ! n) ∈ ka-Val (stk ! n) stk ! n ∈ set stk
    by(auto simp add: is-Ref-def)
  with assms Invoke show ?thesis
     by(auto simp add: split-beta is-Ref-def simp del: ka-Val.simps nth-mem split: if-split-asm dest!:
red-external-aggr-known-addrs-ReadMem in-set-takeD del: is-AddrE)(auto simp add: jvm-known-addrs-def
simp del: ka-Val.simps nth-mem del: is-AddrE)
next
  case Getfield thus ?thesis using assms
    by(auto simp add: jvm-known-addrs-def neq-Nil-conv is-Ref-def split: if-split-asm)
next
  case CAS thus ?thesis using assms
    apply(cases stk; simp)
    subgoal for v stk
      apply(cases stk; simp)
      subgoal for v stk
        by(cases stk)(auto split: if-split-asm simp add: jvm-known-addrs-def is-Ref-def)
      done
```

```
    done
qed(auto simp add: split-beta is-Ref-def neq-Nil-conv split: if-split-asm)

lemma mexecd-known-addrs-ReadMem:
  ⟦ mexecd P t (xcpfrs, h) ta (xcpfrs', h'); ReadMem ad al v ∈ set {|ta|}_o ⟧
  ⟹ ad ∈ jvm-known-addrs t xcpfrs
apply(cases xcpfrs)
apply(cases xcpfrs')
apply simp
apply(erule jvmd-NormalE)
apply(cases fst xcpfrs)
apply(auto simp add: check-def dest: exec-instr-known-addrs-ReadMem)
done

lemma exec-instr-known-addrs-WriteMem:
  assumes exec: (ta, xcp', h', frs') ∈ exec-instr i P t h stk loc C M pc frs
  and check: check-instr i P h stk loc C M pc frs
  and write: {|ta|}_o ! n = WriteMem ad al (Addr a) n < length {|ta|}_o
 shows a ∈ jvm-known-addrs t (None, (stk, loc, C, M, pc) # frs) ∨ a ∈ new-obs-addrs (take n {|ta|}_o)
using assms
proof(cases i)
  case (Invoke M n)
  with check have stk ! n ≠ Null ⟶ the-Addr (stk ! n) ∈ ka-Val (stk ! n) stk ! n ∈ set stk
    by(auto simp add: is-Ref-def)
  thus ?thesis using assms Invoke
     by(auto simp add: is-Ref-def split-beta split: if-split-asm simp del: ka-Val.simps nth-mem dest!:
red-external-aggr-known-addrs-WriteMem in-set-takeD del: is-AddrE)(auto simp add: jvm-known-addrs-def
del: is-AddrE)
next
  case AStore with assms show ?thesis
    by(cases stk)(auto simp add: jvm-known-addrs-def split: if-split-asm)
next
  case Putfield with assms show ?thesis
    by(cases stk)(auto simp add: jvm-known-addrs-def split: if-split-asm)
next
  case CAS with assms show ?thesis
    apply(cases stk; simp)
    subgoal for v stk
      apply(cases stk; simp)
      subgoal for v stk
        by(cases stk)(auto split: if-split-asm simp add: take-Cons' jvm-known-addrs-def)
      done
    done
qed(auto simp add: split-beta split: if-split-asm)

lemma mexecd-known-addrs-WriteMem:
  ⟦ mexecd P t (xcpfrs, h) ta (xcpfrs', h'); {|ta|}_o ! n = WriteMem ad al (Addr a); n < length {|ta|}_o ⟧
  ⟹ a ∈ jvm-known-addrs t xcpfrs ∨ a ∈ new-obs-addrs (take n {|ta|}_o)
apply(cases xcpfrs)
apply(cases xcpfrs')
apply simp
apply(erule jvmd-NormalE)
apply(cases fst xcpfrs)
apply(auto simp add: check-def dest: exec-instr-known-addrs-WriteMem)
```

done

lemma *exec-instr-known-addrs-new-thread*:
assumes *exec*: $(ta, xcp', h', frs') \in exec\text{-}instr\ i\ P\ t\ h\ stk\ loc\ C\ M\ pc\ frs$
and *check*: *check-instr i P h stk loc C M pc frs*
and *new*: $NewThread\ t'\ x'\ h'' \in set\ \{\!|ta|\!\}_t$
shows $jvm\text{-}known\text{-}addrs\ t'\ x' \subseteq jvm\text{-}known\text{-}addrs\ t\ (None, (stk, loc, C, M, pc)\ \#\ frs)$
using *assms*
proof(*cases i*)
case (*Invoke M n*)
with *assms* **have** $stk\ !\ n \neq Null \longrightarrow the\text{-}Addr\ (stk\ !\ n) \in ka\text{-}Val\ (stk\ !\ n) \wedge thread\text{-}id2addr\ (addr2thread\text{-}id\ (the\text{-}Addr\ (stk\ !\ n))) = the\text{-}Addr\ (stk\ !\ n)\ stk\ !\ n \in set\ stk$
apply(*auto simp add*: *is-Ref-def split*: *if-split-asm*)
apply(*frule red-external-aggr-NewThread-idD*, *simp*, *simp*)
apply(*drule red-external-aggr-new-thread-sub-thread*)
apply(*auto intro*: *addr2thread-id-inverse*)
done
with *assms Invoke* **show** *?thesis*
apply(*auto simp add*: *is-Ref-def split-beta split*: *if-split-asm simp del*: *nth-mem del*: *is-AddrE*)
apply(*drule red-external-aggr-NewThread-idD*)
apply(*auto simp add*: *extNTA2JVM-def jvm-known-addrs-def split-beta simp del*: *nth-mem del*: *is-AddrE*)
done
qed(*auto simp add*: *split-beta split*: *if-split-asm*)

lemma *mexecd-known-addrs-new-thread*:
$[\![\ mexecd\ P\ t\ (xcpfrs, h)\ ta\ (xcpfrs', h');\ NewThread\ t'\ x'\ h'' \in set\ \{\!|ta|\!\}_t\ ]\!]$
$\Longrightarrow jvm\text{-}known\text{-}addrs\ t'\ x' \subseteq jvm\text{-}known\text{-}addrs\ t\ xcpfrs$
apply(*cases xcpfrs*)
apply(*cases xcpfrs'*)
apply *simp*
apply(*erule jvmd-NormalE*)
apply(*cases fst xcpfrs*)
apply(*auto 4 3 simp add*: *check-def dest*: *exec-instr-known-addrs-new-thread*)
done

lemma *exec-instr-New-same-addr-same*:
$[\![\ (ta, xcp', h', frs') \in exec\text{-}instr\ ins\ P\ t\ h\ stk\ loc\ C\ M\ pc\ frs;$
$\{\!|ta|\!\}_o\ !\ i = NewHeapElem\ a\ x;\ i < length\ \{\!|ta|\!\}_o;$
$\{\!|ta|\!\}_o\ !\ j = NewHeapElem\ a\ x';\ j < length\ \{\!|ta|\!\}_o\ ]\!]$
$\Longrightarrow i = j$
apply(*cases ins*)
apply(*auto simp add*: *nth-Cons' split*: *prod.split-asm if-split-asm*)
apply(*auto split*: *extCallRet.split-asm dest*: *red-external-aggr-New-same-addr-same*)
done

lemma *exec-New-same-addr-same*:
$[\![\ (ta, xcp', h', frs') \in exec\ P\ t\ (xcp, h, frs);$
$\{\!|ta|\!\}_o\ !\ i = NewHeapElem\ a\ x;\ i < length\ \{\!|ta|\!\}_o;$
$\{\!|ta|\!\}_o\ !\ j = NewHeapElem\ a\ x';\ j < length\ \{\!|ta|\!\}_o\ ]\!]$
$\Longrightarrow i = j$
apply(*cases* (P, t, xcp, h, frs) *rule*: *exec.cases*)
apply(*auto dest*: *exec-instr-New-same-addr-same*)
done

lemma *exec-1-d-New-same-addr-same*:
⟦ *P*,*t* ⊢ *Normal* (*xcp*, *h*, *frs*) −*ta*−*jvmd*→ *Normal* (*xcp'*, *h'*, *frs'*);
{|*ta*|}*o* ! *i* = *NewHeapElem a x*; *i* < *length* {|*ta*|}*o*;
{|*ta*|}*o* ! *j* = *NewHeapElem a x'*; *j* < *length* {|*ta*|}*o* ⟧
⟹ *i* = *j*
by(*erule jvmd-NormalE*)(*rule exec-New-same-addr-same*)

end

locale *JVM-allocated-heap* = *allocated-heap* +
constrains *addr2thread-id* :: ('*addr* :: *addr*) ⇒ '*thread-id*
and *thread-id2addr* :: '*thread-id* ⇒ '*addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: '*heap*
and *allocate* :: '*heap* ⇒ *htype* ⇒ ('*heap* × '*addr*) *set*
and *typeof-addr* :: '*heap* ⇒ '*addr* ⇀ *htype*
and *heap-read* :: '*heap* ⇒ '*addr* ⇒ *addr-loc* ⇒ '*addr val* ⇒ *bool*
and *heap-write* :: '*heap* ⇒ '*addr* ⇒ *addr-loc* ⇒ '*addr val* ⇒ '*heap* ⇒ *bool*
and *allocated* :: '*heap* ⇒ '*addr set*
and *P* :: '*addr jvm-prog*

sublocale *JVM-allocated-heap* < *JVM-heap*
by(*unfold-locales*)

context *JVM-allocated-heap* **begin**

lemma *exec-instr-allocated-mono*:
⟦ (*ta*, *xcp'*, *h'*, *frs'*) ∈ *exec-instr i P t h stk loc C M pc frs*; *check-instr i P h stk loc C M pc frs* ⟧
⟹ *allocated h* ⊆ *allocated h'*
apply(*cases i*)
apply(*auto 4 4 simp add*: *split-beta has-method-def is-native.simps split*: *if-split-asm sum.split-asm intro*: *allocate-allocated-mono dest*: *heap-write-allocated-same dest*!: *red-external-aggr-allocated-mono del*: *subsetI*)
done

lemma *mexecd-allocated-mono*:
mexecd P t (*xcpfrs*, *h*) *ta* (*xcpfrs'*, *h'*) ⟹ *allocated h* ⊆ *allocated h'*
apply(*cases xcpfrs*)
apply(*cases xcpfrs'*)
apply(*simp*)
apply(*erule jvmd-NormalE*)
apply(*cases fst xcpfrs*)
apply(*auto del*: *subsetI simp add*: *check-def dest*: *exec-instr-allocated-mono*)
done

lemma *exec-instr-allocatedD*:
⟦ (*ta*, *xcp'*, *h'*, *frs'*) ∈ *exec-instr i P t h stk loc C M pc frs*;
check-instr i P h stk loc C M pc frs; *NewHeapElem ad CTn* ∈ *set* {|*ta*|}*o* ⟧
⟹ *ad* ∈ *allocated h'* ∧ *ad* ∉ *allocated h*
apply(*cases i*)
apply(*auto 4 4 split*: *if-split-asm prod.split-asm dest*: *allocate-allocatedD dest*!: *red-external-aggr-allocatedD*

simp add: *has-method-def is-native.simps*)
done

lemma *mexecd-allocatedD*:
⟦ *mexecd P t* (*xcpfrs*, *h*) *ta* (*xcpfrs'*, *h'*); *NewHeapElem ad CTn* ∈ *set* {|*ta*|}$_o$ ⟧
⟹ *ad* ∈ *allocated h'* ∧ *ad* ∉ *allocated h*
apply(*cases xcpfrs*)
apply(*cases xcpfrs'*)
apply(*simp*)
apply(*erule jvmd-NormalE*)
apply(*cases fst xcpfrs*)
apply(*auto del*: *subsetI dest*: *exec-instr-allocatedD simp add*: *check-def*)
done

lemma *exec-instr-NewHeapElemD*:
⟦ (*ta*, *xcp'*, *h'*, *frs'*) ∈ *exec-instr i P t h stk loc C M pc frs*; *check-instr i P h stk loc C M pc frs*;
ad ∈ *allocated h'*; *ad* ∉ *allocated h* ⟧
⟹ ∃ *CTn*. *NewHeapElem ad CTn* ∈ *set* {|*ta*|}$_o$
apply(*cases i*)
apply(*auto 4 3 split*: *if-split-asm prod.split-asm sum.split-asm dest*: *allocate-allocatedD heap-write-allocated-same dest*!: *red-external-aggr-NewHeapElemD simp add*: *is-native.simps has-method-def*)
done

lemma *mexecd-NewHeapElemD*:
⟦ *mexecd P t* (*xcpfrs*, *h*) *ta* (*xcpfrs'*, *h'*); *ad* ∈ *allocated h'*; *ad* ∉ *allocated h* ⟧
⟹ ∃ *CTn*. *NewHeapElem ad CTn* ∈ *set* {|*ta*|}$_o$
apply(*cases xcpfrs*)
apply(*cases xcpfrs'*)
apply(*simp*)
apply(*erule jvmd-NormalE*)
apply(*cases fst xcpfrs*)
apply(*auto dest*: *exec-instr-NewHeapElemD simp add*: *check-def*)
done

lemma *mexecd-allocated-multithreaded*:
allocated-multithreaded addr2thread-id thread-id2addr empty-heap allocate typeof-addr heap-write allocated JVM-final (*mexecd P*) *P*
proof
fix *t x m ta x' m'*
assume *mexecd P t* (*x*, *m*) *ta* (*x'*, *m'*)
thus *allocated m* ⊆ *allocated m'* **by**(*rule mexecd-allocated-mono*)
next
fix *x t m ta x' m' ad CTn*
assume *mexecd P t* (*x*, *m*) *ta* (*x'*, *m'*)
and *NewHeapElem ad CTn* ∈ *set* {|*ta*|}$_o$
thus *ad* ∈ *allocated m'* ∧ *ad* ∉ *allocated m* **by**(*rule mexecd-allocatedD*)
next
fix *t x m ta x' m' ad*
assume *mexecd P t* (*x*, *m*) *ta* (*x'*, *m'*)
and *ad* ∈ *allocated m' ad* ∉ *allocated m*
thus ∃ *CTn*. *NewHeapElem ad CTn* ∈ *set* {|*ta*|}$_o$ **by**(*rule mexecd-NewHeapElemD*)
next
fix *t x m ta x' m' i a CTn j CTn'*
assume *mexecd P t* (*x*, *m*) *ta* (*x'*, *m'*)

and $\{\!|ta|\!\}_o$! i = *NewHeapElem a CTn i* < *length* $\{\!|ta|\!\}_o$
and $\{\!|ta|\!\}_o$! j = *NewHeapElem a CTn′ j* < *length* $\{\!|ta|\!\}_o$
thus $i = j$ **by**(*auto dest*: *exec-1-d-New-same-addr-same simp add*: *split-beta*)
qed

end

sublocale *JVM-allocated-heap* < *execd-mthr*: *allocated-multithreaded*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write allocated
JVM-final mexecd P
P
by(*rule mexecd-allocated-multithreaded*)

context *JVM-allocated-heap* **begin**

lemma *mexecd-known-addrs*:
assumes *wf*: *wf-prog wfmd P*
and *ok*: *start-heap-ok*
shows *known-addrs addr2thread-id thread-id2addr empty-heap allocate typeof-addr heap-write allocated jvm-known-addrs JVM-final* (*mexecd P*) *P*
proof
fix *t x m ta x′ m′*
assume *mexecd P t* (*x*, *m*) *ta* (*x′*, *m′*)
thus *jvm-known-addrs t x′* ⊆ *jvm-known-addrs t x* ∪ *new-obs-addrs* $\{\!|ta|\!\}_o$
by(*rule exec-d-known-addrs-mono*[*OF ok*])
next
fix *t x m ta x′ m′ t′ x″ m″*
assume *mexecd P t* (*x*, *m*) *ta* (*x′*, *m′*)
and *NewThread t′ x″ m″* ∈ *set* $\{\!|ta|\!\}_t$
thus *jvm-known-addrs t′ x″* ⊆ *jvm-known-addrs t x* **by**(*rule mexecd-known-addrs-new-thread*)
next
fix *t x m ta x′ m′ ad al v*
assume *mexecd P t* (*x*, *m*) *ta* (*x′*, *m′*)
and *ReadMem ad al v* ∈ *set* $\{\!|ta|\!\}_o$
thus *ad* ∈ *jvm-known-addrs t x* **by**(*rule mexecd-known-addrs-ReadMem*)
next
fix *t x m ta x′ m′ n ad al ad′*
assume *mexecd P t* (*x*, *m*) *ta* (*x′*, *m′*)
and $\{\!|ta|\!\}_o$! *n* = *WriteMem ad al* (*Addr ad′*) *n* < *length* $\{\!|ta|\!\}_o$
thus *ad′* ∈ *jvm-known-addrs t x* ∨ *ad′* ∈ *new-obs-addrs* (*take n* $\{\!|ta|\!\}_o$)
by(*rule mexecd-known-addrs-WriteMem*)
qed

end

context *JVM-heap* **begin**

lemma *exec-instr-read-typeable*:
assumes *exec*: (*ta*, *xcp′*, *h′*, *frs′*) ∈ *exec-instr i P t h stk loc C M pc frs*
and *check*: *check-instr i P h stk loc C M pc frs*
and *read*: *ReadMem ad al v* ∈ *set* $\{\!|ta|\!\}_o$
shows ∃ *T′*. *P*,*h* ⊢ *ad*@*al* : *T′*

using *exec check read*
proof(*cases i*)
  case *ALoad*
  with *assms* **show** *?thesis*
    by(*fastforce simp add*: *split-beta is-Ref-def nat-less-iff word-sless-alt intro*: *addr-loc-type.intros split*: *if-split-asm*)
next
  case (*Getfield F D*)
  with *assms* **show** *?thesis*
    by(*clarsimp simp add*: *split-beta is-Ref-def split*: *if-split-asm*)(*blast intro*: *addr-loc-type.intros dest*: *has-visible-field has-field-mono*)
next
  case (*Invoke M n*)
  with *exec check read* **obtain** *a vs ta' va T*
    where (*ta'*, *va*, *h'*) ∈ *red-external-aggr P t a M vs h*
    and *ReadMem ad al v* ∈ *set* ⦃*ta'*⦄$_o$
    by(*auto split*: *if-split-asm simp add*: *is-Ref-def*)
  thus *?thesis* **by**(*rule red-external-aggr-read-mem-typeable*)
next
  case (*CAS F D*)
  with *assms* **show** *?thesis*
    by(*clarsimp simp add*: *split-beta is-Ref-def conf-def split*: *if-split-asm*)
      (*force intro*: *addr-loc-type.intros dest*: *has-visible-field*[*THEN has-field-mono*])
qed(*auto simp add*: *split-beta is-Ref-def split*: *if-split-asm*)

lemma *exec-1-d-read-typeable*:
  ⟦ *P*,*t* ⊢ *Normal* (*xcp*, *h*, *frs*) −*ta*−*jvmd*→ *Normal* (*xcp'*, *h'*, *frs'*);
    ReadMem ad al v ∈ *set* ⦃*ta*⦄$_o$ ⟧
  ⟹ ∃ *T'*. *P*,*h* ⊢ *ad*@*al* : *T'*
apply(*erule jvmd-NormalE*)
apply(*cases* (*P*, *t*, *xcp*, *h*, *frs*) *rule*: *exec.cases*)
apply(*auto intro*: *exec-instr-read-typeable simp add*: *check-def*)
done

end

sublocale *JVM-heap-base* < *execd-mthr*:
  if-multithreaded
    JVM-final
    mexecd P
    convert-RA
  for *P*
by(*unfold-locales*)

context *JVM-heap-conf* **begin**

lemma *JVM-conf-read-heap-read-typed*:
  JVM-conf-read addr2thread-id thread-id2addr empty-heap allocate typeof-addr (*heap-read-typed P*) *heap-write hconf P*
proof −
  interpret *conf*: *heap-conf-read*
    addr2thread-id thread-id2addr
    spurious-wakeups
    empty-heap allocate typeof-addr heap-read-typed P heap-write hconf

```
    P
    by(rule heap-conf-read-heap-read-typed)
  show ?thesis by(unfold-locales)
qed

lemma exec-instr-New-typeof-addrD:
  ⟦ (ta, xcp', h', frs') ∈ exec-instr i P t h stk loc C M pc frs;
     check-instr i P h stk loc C M pc frs; hconf h;
     NewHeapElem a x ∈ set {|ta|}_o ⟧
  ⟹ typeof-addr h' a = Some x
apply(cases i)
apply(auto dest: allocate-SomeD split: prod.split-asm if-split-asm)
apply(auto 4 4 split: extCallRet.split-asm dest!: red-external-aggr-New-typeof-addrD simp add: has-method-def
is-native.simps)
done

lemma exec-1-d-New-typeof-addrD:
  ⟦ P,t ⊢ Normal (xcp, h, frs) −ta−jvmd→ Normal (xcp', h', frs'); NewHeapElem a x ∈ set {|ta|}_o;
hconf h ⟧
  ⟹ typeof-addr h' a = Some x
apply(erule jvmd-NormalE)
apply(cases xcp)
apply(auto dest: exec-instr-New-typeof-addrD simp add: check-def)
done

lemma exec-instr-non-speculative-typeable:
  assumes exec: (ta, xcp', h', frs') ∈ exec-instr i P t h stk loc C M pc frs
  and check: check-instr i P h stk loc C M pc frs
  and sc: non-speculative P vs (llist-of (map NormalAction {|ta|}_o))
  and vs-conf: vs-conf P h vs
  and hconf: hconf h
  shows (ta, xcp', h', frs') ∈ JVM-heap-base.exec-instr addr2thread-id thread-id2addr spurious-wakeups
empty-heap allocate typeof-addr (heap-read-typed P) heap-write i P t h stk loc C M pc frs
proof −
  note [simp] = JVM-heap-base.exec-instr.simps
    and [split] = if-split-asm prod.split-asm sum.split-asm
    and [split del] = if-split
  from assms show ?thesis
  proof(cases i)
    case ALoad with assms show ?thesis
      by(auto 4 3 intro!: heap-read-typedI dest: vs-confD addr-loc-type-fun)
  next
    case Getfield with assms show ?thesis
      by(auto 4 3 intro!: heap-read-typedI dest: vs-confD addr-loc-type-fun)
  next
    case CAS with assms show ?thesis
      by(auto 4 3 intro!: heap-read-typedI dest: vs-confD addr-loc-type-fun)
  next
    case Invoke with assms show ?thesis
    by(fastforce dest: red-external-aggr-non-speculative-typeable simp add: has-method-def is-native.simps)
  qed(auto)
qed

lemma exec-instr-non-speculative-vs-conf:
```

```
  assumes exec: (ta, xcp', h', frs') ∈ exec-instr i P t h stk loc C M pc frs
  and check: check-instr i P h stk loc C M pc frs
  and sc: non-speculative P vs (llist-of (take n (map NormalAction ⦃ta⦄o)))
  and vs-conf: vs-conf P h vs
  and hconf: hconf h
  shows vs-conf P h' (w-values P vs (take n (map NormalAction ⦃ta⦄o)))
proof −
  note [simp] = JVM-heap-base.exec-instr.simps take-Cons'
    and [split] = if-split-asm prod.split-asm sum.split-asm
    and [split del] = if-split
  from assms show ?thesis
  proof(cases i)
    case New with assms show ?thesis
      by(auto 4 4 dest: hext-allocate vs-conf-allocate intro: vs-conf-hext)
  next
    case NewArray with assms show ?thesis
      by(auto 4 4 dest: hext-allocate vs-conf-allocate intro: vs-conf-hext cong: if-cong)
  next
    case Invoke with assms show ?thesis
    by(fastforce dest: red-external-aggr-non-speculative-vs-conf simp add: has-method-def is-native.simps)
  next
    case AStore
    {
      assume hd (tl (tl stk)) ≠ Null
        and ¬ the-Intg (hd (tl stk)) <s 0
       and ¬ int (alen-of-htype (the (typeof-addr h (the-Addr (hd (tl (tl stk)))))))
≤ sint (the-Intg (hd (tl stk)))
        and P ⊢ the (typeof_h (hd stk)) ≤ the-Array (ty-of-htype (the (typeof-addr h (the-Addr (hd (tl (tl stk))))))))
     moreover hence nat (sint (the-Intg (hd (tl stk)))) < alen-of-htype (the (typeof-addr h (the-Addr (hd (tl (tl stk)))))))
        by(auto simp add: not-le nat-less-iff word-sle-eq word-sless-eq not-less)
      with assms AStore have nat (sint (the-Intg (hd (tl stk)))) < alen-of-htype (the (typeof-addr h' (the-Addr (hd (tl (tl stk)))))))
        by(auto dest!: hext-arrD hext-heap-write)
     ultimately have ∃ T. P,h' ⊢ the-Addr (hd (tl (tl stk)))@ACell (nat (sint (the-Intg (hd (tl stk))))) : T ∧ P,h' ⊢ hd stk :≤ T
        using assms AStore
      by(auto 4 4 simp add: is-Ref-def conf-def dest!: hext-heap-write dest: hext-arrD intro!: addr-loc-type.intros intro: typeof-addr-hext-mono type-of-hext-type-of) }
    thus ?thesis using assms AStore
       by(auto intro!: vs-confI)(blast intro: addr-loc-type-hext-mono conf-hext dest: hext-heap-write vs-confD)+
  next
    case Putfield
    show ?thesis using assms Putfield
    by(auto intro!: vs-confI dest!: hext-heap-write)(blast intro: addr-loc-type.intros addr-loc-type-hext-mono typeof-addr-hext-mono has-field-mono[OF has-visible-field] conf-hext dest: vs-confD)+
  next
    case CAS
    show ?thesis using assms CAS
    by(auto intro!: vs-confI dest!: hext-heap-write)(blast intro: addr-loc-type.intros addr-loc-type-hext-mono typeof-addr-hext-mono has-field-mono[OF has-visible-field] conf-hext dest: vs-confD)+
  qed(auto)
```

qed

lemma *mexecd-non-speculative-typeable*:
⟦ *P*,*t* ⊢ *Normal* (*xcp*, *h*, *stk*) −*ta*−*jvmd*→ *Normal* (*xcp'*, *h'*, *frs'*); *non-speculative P vs* (*llist-of* (*map NormalAction* {|*ta*|}$_o$));
vs-conf P h vs; *hconf h* ⟧
⟹ *JVM-heap-base.exec-1-d addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr* (*heap-read-typed P*) *heap-write P t* (*Normal* (*xcp*, *h*, *stk*)) *ta* (*Normal* (*xcp'*, *h'*, *frs'*))
apply(*erule jvmd-NormalE*)
apply(*cases xcp*)
apply(*auto intro*!: *JVM-heap-base.exec-1-d.intros simp add*: *JVM-heap-base.exec-d-def check-def JVM-heap-base.ex*
intro: *exec-instr-non-speculative-typeable*)
done

lemma *mexecd-non-speculative-vs-conf*:
⟦ *P*,*t* ⊢ *Normal* (*xcp*, *h*, *stk*) −*ta*−*jvmd*→ *Normal* (*xcp'*, *h'*, *frs'*);
non-speculative P vs (*llist-of* (*take n* (*map NormalAction* {|*ta*|}$_o$)));
vs-conf P h vs; *hconf h* ⟧
⟹ *vs-conf P h'* (*w-values P vs* (*take n* (*map NormalAction* {|*ta*|}$_o$)))
apply(*erule jvmd-NormalE*)
apply(*cases xcp*)
apply(*auto intro*!: *JVM-heap-base.exec-1-d.intros simp add*: *JVM-heap-base.exec-d-def check-def JVM-heap-base.ex*
intro: *exec-instr-non-speculative-vs-conf*)
done

end

locale *JVM-allocated-heap-conf* =
JVM-heap-conf
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write hconf
P
+
JVM-allocated-heap
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
allocated
P
for *addr2thread-id* :: ('*addr* :: *addr*) ⇒ '*thread-id*
and *thread-id2addr* :: '*thread-id* ⇒ '*addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: '*heap*
and *allocate* :: '*heap* ⇒ *htype* ⇒ ('*heap* × '*addr*) *set*
and *typeof-addr* :: '*heap* ⇒ '*addr* ⇀ *htype*
and *heap-read* :: '*heap* ⇒ '*addr* ⇒ *addr-loc* ⇒ '*addr val* ⇒ *bool*
and *heap-write* :: '*heap* ⇒ '*addr* ⇒ *addr-loc* ⇒ '*addr val* ⇒ '*heap* ⇒ *bool*
and *hconf* :: '*heap* ⇒ *bool*
and *allocated* :: '*heap* ⇒ '*addr set*
and *P* :: '*addr jvm-prog*
begin

lemma *mexecd-known-addrs-typing*:

assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *ok*: *start-heap-ok*
shows *known-addrs-typing addr2thread-id thread-id2addr empty-heap allocate typeof-addr heap-write allocated jvm-known-addrs JVM-final* (*mexecd P*) (λ*t* (*xcp*, *frstls*) *h*. Φ ⊢ *t*: (*xcp*, *h*, *frstls*) √) *P*
proof −
from *wf* **obtain** *wf-md* **where** *wf-prog wf-md P* **by**(*blast dest*: *wt-jvm-progD*)
then
interpret *known-addrs*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
allocated jvm-known-addrs
JVM-final mexecd P P
using *ok* **by**(*rule mexecd-known-addrs*)

show *?thesis*
proof
fix *t x m ta x′ m′*
assume *mexecd P t* (*x*, *m*) *ta* (*x′*, *m′*)
thus *m* ⊴ *m′* **by**(*auto simp add*: *split-beta intro*: *exec-1-d-hext*)
next
fix *t x m ta x′ m′ vs*
assume *exec*: *mexecd P t* (*x*, *m*) *ta* (*x′*, *m′*)
and *ts-ok*: (λ(*xcp*, *frstls*) *h*. Φ ⊢ *t*:(*xcp*, *h*, *frstls*) √) *x m*
and *vs*: *vs-conf P m vs*
and *ns*: *non-speculative P vs* (*llist-of* (*map NormalAction* ⦃*ta*⦄$_o$))

let *?mexecd* = *JVM-heap-base.mexecd addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr* (*heap-read-typed P*) *heap-write P*
have *lift*: *lifting-wf JVM-final ?mexecd* (λ*t* (*xcp*, *frstls*) *h*. Φ ⊢ *t*: (*xcp*, *h*, *frstls*) √)
by(*intro JVM-conf-read.lifting-wf-correct-state-d JVM-conf-read-heap-read-typed wf*)

from *exec ns vs ts-ok* **have** *exec′*: *?mexecd t* (*x*, *m*) *ta* (*x′*, *m′*)
by(*auto simp add*: *split-beta correct-state-def dest*: *mexecd-non-speculative-typeable*)
thus (λ(*xcp*, *frstls*) *h*. Φ ⊢ *t*:(*xcp*, *h*, *frstls*) √) *x′ m′* **using** *ts-ok*
by(*rule lifting-wf.preserves-red*[*OF lift*])
{
fix *t″ x″ m″*
assume *New*: *NewThread t″ x″ m″* ∈ *set* ⦃*ta*⦄$_t$
with *exec* **have** *m″* = *snd* (*x′*, *m′*) **by**(*rule execd-mthr.new-thread-memory*)
thus (λ(*xcp*, *frstls*) *h*. Φ ⊢ *t″*:(*xcp*, *h*, *frstls*) √) *x″ m″*
using *lifting-wf.preserves-NewThread*[**where** *?r*=*?mexecd*, *OF lift exec′ ts-ok*] *New*
by *auto* **}**
{ fix *t″ x″*
assume (λ(*xcp*, *frstls*) *h*. Φ ⊢ *t″*:(*xcp*, *h*, *frstls*) √) *x″ m*
with *lift exec′ ts-ok* **show** (λ(*xcp*, *frstls*) *h*. Φ ⊢ *t″*:(*xcp*, *h*, *frstls*) √) *x″ m′*
by(*rule lifting-wf.preserves-other*) **}**
next
fix *t x m ta x′ m′ vs n*
assume *exec*: *mexecd P t* (*x*, *m*) *ta* (*x′*, *m′*)
and *ts-ok*: (λ(*xcp*, *frstls*) *h*. Φ ⊢ *t*:(*xcp*, *h*, *frstls*) √) *x m*
and *vs*: *vs-conf P m vs*
and *ns*: *non-speculative P vs* (*llist-of* (*take n* (*map NormalAction* ⦃*ta*⦄$_o$)))
thus *vs-conf P m′* (*w-values P vs* (*take n* (*map NormalAction* ⦃*ta*⦄$_o$)))

```
    by(auto simp add: correct-state-def dest: mexecd-non-speculative-vs-conf)
  next
    fix t x m ta x' m' ad al v
    assume mexecd P t (x, m) ta (x', m')
      and (λ(xcp, frstls) h. Φ ⊢ t:(xcp, h, frstls) √) x m
      and ReadMem ad al v ∈ set {|ta|}_o
    thus ∃ T. P,m ⊢ ad@al : T
      by(auto simp add: correct-state-def split-beta dest: exec-1-d-read-typeable)
  next
    fix t x m ta x' m' ad hT
    assume mexecd P t (x, m) ta (x', m')
      and (λ(xcp, frstls) h. Φ ⊢ t:(xcp, h, frstls) √) x m
      and NewHeapElem ad hT ∈ set {|ta|}_o
    thus typeof-addr m' ad = ⌊hT⌋
      by(auto dest: exec-1-d-New-typeof-addrD[where x=hT] simp add: split-beta correct-state-def)
  qed
qed

lemma executions-sc:
  assumes wf: wf-jvm-prog_Φ P
  and wf-start: wf-start-state P C M vs
  and vs2: ⋃(ka-Val ` set vs) ⊆ set start-addrs
  shows executions-sc-hb (JVMd-ℰ P C M vs status) P
    (is executions-sc-hb ?E P)
proof −
  from wf-start obtain Ts T meth D where ok: start-heap-ok
    and sees: P ⊢ C sees M:Ts→T=⌊meth⌋ in D
    and vs1: P,start-heap ⊢ vs [:≤] Ts by cases

  interpret known-addrs-typing
    addr2thread-id thread-id2addr
    spurious-wakeups
    empty-heap allocate typeof-addr heap-read heap-write
    allocated jvm-known-addrs
    JVM-final mexecd P λt (xcp, frstls) h. Φ ⊢ t: (xcp, h, frstls) √ P
    using wf ok by(rule mexecd-known-addrs-typing)

  from wf obtain wf-md where wf': wf-prog wf-md P by(blast dest: wt-jvm-progD)
  hence wf-syscls P by(rule wf-prog-wf-syscls)
  thus ?thesis
  proof(rule executions-sc-hb)
    from correct-jvm-state-initial[OF wf wf-start]
    show correct-state-ts Φ (thr (JVM-start-state P C M vs)) start-heap
      by(simp add: correct-jvm-state-def start-state-def split-beta)
  next
    show jvm-known-addrs start-tid ((λ(mxs, mxl0, b) vs. (None, [([], Null # vs @ replicate mxl0
undefined-value, fst (method P C M), M, 0)])) (the (snd (snd (snd (method P C M))))) vs) ⊆ allocated
start-heap
      using vs2
    by(auto simp add: split-beta start-addrs-allocated jvm-known-addrs-def intro: start-tid-start-addrs[OF
‹wf-syscls P› ok])
  qed
qed
```

end

declare *split-paired-Ex* [*simp del*]
declare *eq-upto-seq-inconsist-simps* [*simp*]

context *JVM-progress* **begin**

abbreviation (*input*) *jvm-non-speculative-read-bound* :: *nat* **where**
jvm-non-speculative-read-bound ≡ *2*

lemma *exec-instr-non-speculative-read*:
assumes *hrt*: *heap-read-typeable hconf P*
and *vs*: *vs-conf P* (*shr s*) *vs*
and *hconf*: *hconf* (*shr s*)
and *exec-i*: (*ta*, *xcp'*, *h'*, *frs'*) ∈ *exec-instr i P t* (*shr s*) *stk loc C M pc frs*
and *check*: *check-instr i P* (*shr s*) *stk loc C M pc frs*
and *aok*: *execd-mthr.mthr.if.actions-ok s t ta*
and *i*: *I* < *length* ⦃*ta*⦄$_o$
and *read*: ⦃*ta*⦄$_o$! *I* = *ReadMem a'' al'' v*
and *v'*: *v'* ∈ *w-values P vs* (*map NormalAction* (*take I* ⦃*ta*⦄$_o$)) (*a''*, *al''*)
and *ns*: *non-speculative P vs* (*llist-of* (*map NormalAction* (*take I* ⦃*ta*⦄$_o$)))
shows ∃ *ta' xcp'' h'' frs''*. (*ta'*, *xcp''*, *h''*, *frs''*) ∈ *exec-instr i P t* (*shr s*) *stk loc C M pc frs* ∧
execd-mthr.mthr.if.actions-ok s t ta' ∧
I < *length* ⦃*ta'*⦄$_o$ ∧ *take I* ⦃*ta'*⦄$_o$ = *take I* ⦃*ta*⦄$_o$ ∧
⦃*ta'*⦄$_o$! *I* = *ReadMem a'' al'' v'* ∧
length ⦃*ta'*⦄$_o$ ≤ *max jvm-non-speculative-read-bound* (*length* ⦃*ta*⦄$_o$)
using *exec-i i read*
proof(*cases i*)
case [*simp*]: *ALoad*
let *?a* = *the-Addr* (*hd* (*tl stk*))
let *?i* = *the-Intg* (*hd stk*)
from *exec-i i read* **have** *Null*: *hd* (*tl stk*) ≠ *Null*
and *bounds*: *0* <=*s ?i sint ?i* < *int* (*alen-of-htype* (*the* (*typeof-addr* (*shr s*) *?a*)))
and [*simp*]: *I* = *0 a''* = *?a al''* = *ACell* (*nat* (*sint ?i*))
by(*auto split*: *if-split-asm*)

from *Null check* **obtain** *a T n*
where *a*: *length stk* > *1 hd* (*tl stk*) = *Addr a*
and *type*: *typeof-addr* (*shr s*) *?a* = ⌊*Array-type T n*⌋ **by**(*fastforce simp add*: *is-Ref-def*)
from *bounds type* **have** *nat* (*sint ?i*) < *n*
by (*simp add*: *word-sle-eq nat-less-iff*)
with *type* **have** *adal*: *P*,*shr s* ⊢ *?a*@*ACell* (*nat* (*sint ?i*)) : *T*
by(*rule addr-loc-type.intros*)
from *v' vs adal* **have** *P*,*shr s* ⊢ *v'* :≤ *T* **by**(*auto dest*!: *vs-confD dest*: *addr-loc-type-fun*)
with *hrt adal* **have** *heap-read* (*shr s*) *?a* (*ACell* (*nat* (*sint ?i*))) *v'* **using** *hconf* **by**(*rule heap-read-typeableD*)
with *type Null aok exec-i* **show** *?thesis* **apply** *auto* **using** *bounds* **by** *fastforce*+
next
case [*simp*]: (*Getfield F D*)
let *?a* = *the-Addr* (*hd stk*)

from *exec-i i read* **have** *Null*: *hd stk* ≠ *Null*
and [*simp*]: *I* = *0 a''* = *?a al''* = *CField D F*
by(*auto split*: *if-split-asm*)
with *check* **obtain** *U T fm C' a*

```
    where sees: P ⊢ D sees F:T (fm) in D
    and type: typeof-addr (shr s) ?a = ⌊U⌋
    and sub: P ⊢ class-type-of U ≼* D
    and a: hd stk = Addr a length stk > 0 by(auto simp add: is-Ref-def)
  from has-visible-field[OF sees] sub
  have P ⊢ class-type-of U has F:T (fm) in D by(rule has-field-mono)
  with type have adal: P,shr s ⊢ ?a@CField D F : T
    by(rule addr-loc-type.intros)
  from v' vs adal have P,shr s ⊢ v' :≤ T by(auto dest!: vs-confD dest: addr-loc-type-fun)
  with hrt adal have heap-read (shr s) ?a (CField D F) v' using hconf by(rule heap-read-typeableD)
  with type Null aok exec-i show ?thesis by(fastforce)
next
  case [simp]: (CAS F D)
  let ?a = the-Addr (hd (tl (tl stk)))

  from exec-i i read have Null: hd (tl (tl stk)) ≠ Null
    and [simp]: I = 0 a'' = ?a al'' = CField D F
    by(auto split: if-split-asm simp add: nth-Cons')
  with check obtain U T fm C' a
    where sees: P ⊢ D sees F:T (fm) in D
    and type: typeof-addr (shr s) ?a = ⌊U⌋
    and sub: P ⊢ class-type-of U ≼* D
    and a: hd (tl (tl stk)) = Addr a length stk > 2
    and v: P,shr s ⊢ hd stk :≤ T
    by(auto simp add: is-Ref-def)
  from has-visible-field[OF sees] sub
  have P ⊢ class-type-of U has F:T (fm) in D by(rule has-field-mono)
  with type have adal: P,shr s ⊢ ?a@CField D F : T
    by(rule addr-loc-type.intros)
  from v' vs adal have P,shr s ⊢ v' :≤ T by(auto dest!: vs-confD dest: addr-loc-type-fun)
 with hrt adal have read: heap-read (shr s) ?a (CField D F) v' using hconf by(rule heap-read-typeableD)
  show ?thesis
  proof(cases v' = hd (tl stk))
    case True
    from heap-write-total[OF hconf adal v] a obtain h'
      where heap-write (shr s) a (CField D F) (hd stk) h' by auto
    then show ?thesis using read a True aok exec-i by fastforce
  next
    case False
    then show ?thesis using read a aok exec-i
      by(fastforce intro!: disjI2)
  qed
next
  case [simp]: (Invoke M n)
  let ?a = the-Addr (stk ! n)
  let ?vs = rev (take n stk)
  from exec-i i read have Null: stk ! n ≠ Null
    and iec: snd (snd (snd (method P (class-type-of (the (typeof-addr (shr s) ?a))) M))) = Native
    by(auto split: if-split-asm)
  with check obtain a T Ts Tr D
    where a: stk ! n = Addr a n < length stk
    and type: typeof-addr (shr s) ?a = ⌊T⌋
    and extwt: P ⊢ class-type-of T sees M:Ts→Tr = Native in D D·M(Ts) :: Tr
    by(auto simp add: is-Ref-def has-method-def)
```

from *extwt* **have** *native*: *is-native P T M* **by**(*auto simp add*: *is-native.simps*)
from *Null iec type exec-i* **obtain** *ta′ va*
where *red*: (*ta′*, *va*, *h′*) ∈ *red-external-aggr P t ?a M ?vs* (*shr s*)
and *ta*: *ta* = *extTA2JVM P ta′* **by**(*fastforce*)
from *aok ta* **have** *aok′*: *execd-mthr.mthr.if.actions-ok s t ta′* **by** *simp*
from *red-external-aggr-non-speculative-read*[*OF hrt vs red*[*unfolded a the-Addr.simps*] - *aok′ hconf*, *of I a″ al″ v v′*]
native type i read v′ ns a ta
obtain *ta″ va″ h″*
where (*ta″*, *va″*, *h″*) ∈ *red-external-aggr P t a M* (*rev* (*take n stk*)) (*shr s*)
and *execd-mthr.mthr.if.actions-ok s t ta″*
and *I* < *length* ⦃*ta″*⦄*o take I* ⦃*ta″*⦄*o* = *take I* ⦃*ta′*⦄*o*
and ⦃*ta″*⦄*o* ! *I* = *ReadMem a″ al″ v′ length* ⦃*ta″*⦄*o* ≤ *length* ⦃*ta′*⦄*o* **by** *auto*
thus *?thesis* **using** *Null iec ta extwt a type*
by(*cases va″*) *force*+
qed(*auto simp add*: *split-beta split*: *if-split-asm*)

lemma *exec-1-d-non-speculative-read*:
assumes *hrt*: *heap-read-typeable hconf P*
and *vs*: *vs-conf P* (*shr s*) *vs*
and *exec*: *P*,*t* ⊢ *Normal* (*xcp*, *shr s*, *frs*) −*ta*−*jvmd*→ *Normal* (*xcp′*, *h′*, *frs′*)
and *aok*: *execd-mthr.mthr.if.actions-ok s t ta*
and *hconf*: *hconf* (*shr s*)
and *i*: *I* < *length* ⦃*ta*⦄*o*
and *read*: ⦃*ta*⦄*o* ! *I* = *ReadMem a″ al″ v*
and *v′*: *v′* ∈ *w-values P vs* (*map NormalAction* (*take I* ⦃*ta*⦄*o*)) (*a″*, *al″*)
and *ns*: *non-speculative P vs* (*llist-of* (*map NormalAction* (*take I* ⦃*ta*⦄*o*)))
shows ∃ *ta′ xcp″ h″ frs″*. *P*,*t* ⊢ *Normal* (*xcp*, *shr s*, *frs*) −*ta′*−*jvmd*→ *Normal* (*xcp″*, *h″*, *frs″*) ∧
execd-mthr.mthr.if.actions-ok s t ta′ ∧
I < *length* ⦃*ta′*⦄*o* ∧ *take I* ⦃*ta′*⦄*o* = *take I* ⦃*ta*⦄*o* ∧
⦃*ta′*⦄*o* ! *I* = *ReadMem a″ al″ v′* ∧
length ⦃*ta′*⦄*o* ≤ *max jvm-non-speculative-read-bound* (*length* ⦃*ta*⦄*o*)
using *assms*
apply −
apply(*erule jvmd-NormalE*)
apply(*cases* (*P*, *t*, *xcp*, *shr s*, *frs*) *rule*: *exec.cases*)
apply *simp*
defer
apply *simp*
apply *clarsimp*
apply(*drule* (*3*) *exec-instr-non-speculative-read*)
apply(*clarsimp simp add*: *check-def has-method-def*)
apply *simp*
apply(*rule i*)
apply(*rule read*)
apply(*rule v′*)
apply(*rule ns*)
apply(*clarsimp simp add*: *exec-1-d.simps exec-d-def*)
done

end

declare *split-paired-Ex* [*simp*]
declare *eq-upto-seq-inconsist-simps* [*simp del*]

locale *JVM-allocated-progress* =
  JVM-progress
    addr2thread-id thread-id2addr
    spurious-wakeups
    empty-heap allocate typeof-addr heap-read heap-write hconf
    P
  +
  JVM-allocated-heap-conf
    addr2thread-id thread-id2addr
    spurious-wakeups
    empty-heap allocate typeof-addr heap-read heap-write hconf
    allocated
    P
  for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
  and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
  and *spurious-wakeups* :: *bool*
  and *empty-heap* :: *'heap*
  and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
  and *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *htype*
  and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
  and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
  and *hconf* :: *'heap* ⇒ *bool*
  and *allocated* :: *'heap* ⇒ *'addr set*
  and *P* :: *'addr jvm-prog*
begin

lemma *non-speculative-read*:
  assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
  and *hrt*: *heap-read-typeable hconf P*
  and *wf-start*: *wf-start-state P C M vs*
  and *ka*: ⋃(*ka-Val* ‘ *set vs*) ⊆ *set start-addrs*
  shows *execd-mthr.if.non-speculative-read jvm-non-speculative-read-bound*
      (*init-fin-lift-state status* (*JVM-start-state P C M vs*))
      (*w-values P* (λ-. {}) (*map snd* (*lift-start-obs start-tid start-heap-obs*)))
  (**is** *execd-mthr.if.non-speculative-read - ?start-state ?start-vs*)
proof(*rule execd-mthr.if.non-speculative-readI*)
  fix *ttas s′ t x ta x′ m′ i ad al v v′*

  assume *τRed*: *execd-mthr.mthr.if.RedT P ?start-state ttas s′*
    and *sc*: *non-speculative P ?start-vs* (*llist-of* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas*)))
    and *ts′t*: *thr s′ t* = ⌊(*x*, *no-wait-locks*)⌋
    and *red*: *execd-mthr.init-fin P t* (*x*, *shr s′*) *ta* (*x′*, *m′*)
    and *aok*: *execd-mthr.mthr.if.actions-ok s′ t ta*
    and *i*: *i* < *length* ⦃*ta*⦄$_o$
    and *ns′*: *non-speculative P* (*w-values P ?start-vs* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas*))) (*llist-of* (*take i* ⦃*ta*⦄$_o$))
    and *read*: ⦃*ta*⦄$_o$! *i* = *NormalAction* (*ReadMem ad al v*)
    and *v′*: *v′* ∈ *w-values P ?start-vs* (*concat* (*map* (λ(*t*, *ta*). ⦃*ta*⦄$_o$) *ttas*) @ *take i* ⦃*ta*⦄$_o$) (*ad*, *al*)

  from *wf-start* **obtain** *Ts T meth D* **where** *ok*: *start-heap-ok*
    and *sees*: *P* ⊢ *C sees M*:*Ts*→*T* = ⌊*meth*⌋ *in D*
    and *conf*: *P*,*start-heap* ⊢ *vs* [:≤] *Ts* **by** *cases*

let *?conv* = λ*ttas*. *concat* (*map* (λ(*t*, *ta*). $\{\!|ta|\!\}_o$) *ttas*)
let *?vs'* = *w-values P ?start-vs* (*?conv ttas*)
let *?wt-ok* = *init-fin-lift* (λ*t* (*xcp*, *frstls*) *h*. $\Phi \vdash t$: (*xcp*, *h*, *frstls*) $\surd$)
let *?start-obs* = *map snd* (*lift-start-obs start-tid start-heap-obs*)

from *wf* **obtain** *wf-md* **where** *wf'*: *wf-prog wf-md P* **by**(*blast dest*: *wt-jvm-progD*)

interpret *known-addrs-typing*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
allocated jvm-known-addrs
JVM-final mexecd P λ*t* (*xcp*, *frstls*) *h*. $\Phi \vdash t$: (*xcp*, *h*, *frstls*) $\surd$
using *wf ok* **by**(*rule mexecd-known-addrs-typing*)

from *conf* **have** *len2*: *length vs* = *length Ts* **by**(*rule list-all2-lengthD*)

from *correct-jvm-state-initial*[*OF wf wf-start*]
have *correct-state-ts* Φ (*thr* (*JVM-start-state P C M vs*)) *start-heap*
by(*simp add*: *correct-jvm-state-def start-state-def split-beta*)
hence *ts-ok-start*: *ts-ok ?wt-ok* (*thr ?start-state*) (*shr ?start-state*)
unfolding *ts-ok-init-fin-lift-init-fin-lift-state* **by**(*simp add*: *start-state-def split-beta*)

have *sc'*: *non-speculative P ?start-vs* (*lmap snd* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) $\{\!|ta|\!\}_o$)) (*llist-of ttas*))))
using *sc* **by**(*simp add*: *lmap-lconcat llist.map-comp o-def split-def lconcat-llist-of*[*symmetric*])
from *start-state-vs-conf*[*OF wf-prog-wf-syscls*[*OF wf'*]]
have *vs-conf-start*: *vs-conf P* (*shr ?start-state*) *?start-vs*
by(*simp add*:*init-fin-lift-state-conv-simps start-state-def split-beta*)
with τ*Red ts-ok-start sc*
have *wt'*: *ts-ok ?wt-ok* (*thr s'*) (*shr s'*)
and *vs'*: *vs-conf P* (*shr s'*) *?vs'* **by**(*rule if-RedT-non-speculative-invar*)+

from *red i read* **obtain** *xcp frs xcp' frs' ta'*
where *x*: *x* = (*Running*, *xcp*, *frs*) **and** *x'*: *x'* = (*Running*, *xcp'*, *frs'*)
and *ta*: *ta* = *convert-TA-initial* (*convert-obs-initial ta'*)
and *red'*: *P*,*t* $\vdash$ *Normal* (*xcp*, *shr s'*, *frs*) $-$*ta'*$-$*jvmd*$\rightarrow$ *Normal* (*xcp'*, *m'*, *frs'*)
by *cases fastforce*+

from *ts't wt' x* **have** *hconf*: *hconf* (*shr s'*) **by**(*auto dest*!: *ts-okD simp add*: *correct-state-def*)

have *aok'*: *execd-mthr.mthr.if.actions-ok s' t ta'* **using** *aok* **unfolding** *ta* **by** *simp*

from *i read v' ns' ta* **have** *i* < *length* $\{\!|ta'|\!\}_o$
and $\{\!|ta'|\!\}_o$! *i* = *ReadMem ad al v*
and *v'* $\in$ *w-values P ?vs'* (*map NormalAction* (*take i* $\{\!|ta'|\!\}_o$)) (*ad*, *al*)
and *non-speculative P ?vs'* (*llist-of* (*map NormalAction* (*take i* $\{\!|ta'|\!\}_o$)))
by(*simp-all add*: *take-map*)

from *exec-1-d-non-speculative-read*[*OF hrt vs' red' aok' hconf this*]
obtain *ta'' xcp'' frs'' h''*
where *red''*: *P*,*t* $\vdash$ *Normal* (*xcp*, *shr s'*, *frs*) $-$*ta''*$-$*jvmd*$\rightarrow$ *Normal* (*xcp''*, *h''*, *frs''*)
and *aok''*: *execd-mthr.mthr.if.actions-ok s' t ta''*
and *i''*: *i* < *length* $\{\!|ta''|\!\}_o$

```
    and eq'': take i ⦃ta''⦄_o = take i ⦃ta'⦄_o
    and read'': ⦃ta''⦄_o ! i = ReadMem ad al v'
    and len'': length ⦃ta''⦄_o ≤ max jvm-non-speculative-read-bound (length ⦃ta'⦄_o) by blast

  let ?x' = (Running, xcp'', frs'')
  let ?ta' = convert-TA-initial (convert-obs-initial ta'')
  from red'' have execd-mthr.init-fin P t (x, shr s') ?ta' (?x', h'')
    unfolding x by -(rule execd-mthr.init-fin.NormalAction, simp)
  moreover from aok'' have execd-mthr.mthr.if.actions-ok s' t ?ta' by simp
  moreover from i'' have i < length ⦃?ta'⦄_o by simp
  moreover from eq'' have take i ⦃?ta'⦄_o = take i ⦃ta⦄_o unfolding ta by(simp add: take-map)
  moreover from read'' i'' have ⦃?ta'⦄_o ! i = NormalAction (ReadMem ad al v') by(simp add:
nth-map)
  moreover from len'' have length ⦃?ta'⦄_o ≤ max jvm-non-speculative-read-bound (length ⦃ta⦄_o)
    unfolding ta by simp
  ultimately
  show ∃ ta' x'' m''. execd-mthr.init-fin P t (x, shr s') ta' (x'', m'') ∧
                      execd-mthr.mthr.if.actions-ok s' t ta' ∧
                      i < length ⦃ta'⦄_o ∧ take i ⦃ta'⦄_o = take i ⦃ta⦄_o ∧
                      ⦃ta'⦄_o ! i = NormalAction (ReadMem ad al v') ∧
                      length ⦃ta'⦄_o ≤ max jvm-non-speculative-read-bound (length ⦃ta⦄_o)
    by blast
qed

lemma JVM-cut-and-update:
  assumes wf: wf-jvm-prog_Φ P
  and hrt: heap-read-typeable hconf P
  and wf-start: wf-start-state P C M vs
  and ka: ⋃(ka-Val ` set vs) ⊆ set start-addrs
  shows execd-mthr.if.cut-and-update (init-fin-lift-state status (JVM-start-state P C M vs))
        (mrw-values P Map.empty (map snd (lift-start-obs start-tid start-heap-obs)))
proof −
  from wf-start obtain Ts T meth D where ok: start-heap-ok
    and sees: P ⊢ C sees M:Ts→T = ⌊meth⌋ in D
    and conf: P,start-heap ⊢ vs [:≤] Ts by cases

  interpret known-addrs-typing
    addr2thread-id thread-id2addr
    spurious-wakeups
    empty-heap allocate typeof-addr heap-read heap-write
    allocated jvm-known-addrs
    JVM-final mexecd P λt (xcp, frstls) h. Φ ⊢ t: (xcp, h, frstls) √
    using wf ok by(rule mexecd-known-addrs-typing)

  let ?start-vs = w-values P (λ-. {}) (map snd (lift-start-obs start-tid start-heap-obs))
  let ?wt-ok = init-fin-lift (λt (xcp, frstls) h. Φ ⊢ t: (xcp, h, frstls) √)
  let ?start-obs = map snd (lift-start-obs start-tid start-heap-obs)
  let ?start-state = init-fin-lift-state status (JVM-start-state P C M vs)

  from wf obtain wf-md where wf': wf-prog wf-md P by(blast dest: wt-jvm-progD)
  hence wf-syscls P by(rule wf-prog-wf-syscls)
  moreover
  note non-speculative-read[OF wf hrt wf-start ka]
  moreover have ts-ok ?wt-ok (thr ?start-state) (shr ?start-state)
```

using *correct-jvm-state-initial*[*OF wf wf-start*]
by(*simp add*: *correct-jvm-state-def start-state-def split-beta*)
moreover have *ka*: *jvm-known-addrs start-tid* ((λ(*mxs*, *mxl0*, *b*) *vs*. (*None*, [([], *Null* # *vs* @ *replicate mxl0 undefined-value*, *fst* (*method P C M*), *M*, *0*)])) (*the* (*snd* (*snd* (*snd* (*method P C M*))))) *vs*) ⊆ *allocated start-heap*
using *ka* **by**(*auto simp add*: *split-beta start-addrs-allocated jvm-known-addrs-def intro*: *start-tid-start-addrs*[*OF* ‹*wf-syscls P*› *ok*])
ultimately show *?thesis* **by**(*rule non-speculative-read-into-cut-and-update*)
qed

lemma *JVM-drf*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *hrt*: *heap-read-typeable hconf P*
and *wf-start*: *wf-start-state P C M vs*
and *ka*: ⋃(*ka-Val* ‘ *set vs*) ⊆ *set start-addrs*
shows *drf* (*JVMd-*$\mathcal{E}$ *P C M vs status*) *P*
proof −
from *wf-start* **obtain** *Ts T meth D* **where** *ok*: *start-heap-ok*
and *sees*: *P* ⊢ *C sees M*:*Ts*→*T* = ⌊*meth*⌋ *in D*
and *conf*: *P*,*start-heap* ⊢ *vs* [:≤] *Ts* **by** *cases*

from *wf* **obtain** *wf-md* **where** *wf′*: *wf-prog wf-md P* **by**(*blast dest*: *wt-jvm-progD*)
hence *wf-syscls P* **by**(*rule wf-prog-wf-syscls*)
with *JVM-cut-and-update*[*OF assms*]
show *?thesis*
proof(*rule known-addrs-typing.drf*[*OF mexecd-known-addrs-typing*[*OF wf ok*]])
from *correct-jvm-state-initial*[*OF wf wf-start*]
show *correct-state-ts* Φ (*thr* (*JVM-start-state P C M vs*)) *start-heap*
by(*simp add*: *correct-jvm-state-def start-state-def split-beta*)
next
show *jvm-known-addrs start-tid* ((λ(*mxs*, *mxl0*, *b*) *vs*. (*None*, [([], *Null* # *vs* @ *replicate mxl0 undefined-value*, *fst* (*method P C M*), *M*, *0*)])) (*the* (*snd* (*snd* (*snd* (*method P C M*))))) *vs*) ⊆ *allocated start-heap*
using *ka* **by**(*auto simp add*: *split-beta start-addrs-allocated jvm-known-addrs-def intro*: *start-tid-start-addrs*[*OF* ‹*wf-syscls P*› *ok*])
qed
qed

lemma *JVM-sc-legal*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *hrt*: *heap-read-typeable hconf P*
and *wf-start*: *wf-start-state P C M vs*
and *ka*: ⋃(*ka-Val* ‘ *set vs*) ⊆ *set start-addrs*
shows *sc-legal* (*JVMd-*$\mathcal{E}$ *P C M vs status*) *P*
proof −
from *wf-start* **obtain** *Ts T meth D* **where** *ok*: *start-heap-ok*
and *sees*: *P* ⊢ *C sees M*:*Ts*→*T* = ⌊*meth*⌋ *in D*
and *conf*: *P*,*start-heap* ⊢ *vs* [:≤] *Ts* **by** *cases*

interpret *known-addrs-typing*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
allocated jvm-known-addrs

JVM-final mexecd P λt (xcp, frstls) h. Φ ⊢ t: (xcp, h, frstls) √
using *wf ok* **by**(*rule mexecd-known-addrs-typing*)

from *wf* **obtain** *wf-md* **where** *wf'*: *wf-prog wf-md P* **by**(*blast dest*: *wt-jvm-progD*)
hence *wf-syscls P* **by**(*rule wf-prog-wf-syscls*)

let *?start-vs = w-values P* (λ-. {}) (*map snd* (*lift-start-obs start-tid start-heap-obs*))
let *?wt-ok = init-fin-lift* (λ*t* (*xcp*, *frstls*) *h*. Φ ⊢ *t*: (*xcp*, *h*, *frstls*) √)
let *?start-obs = map snd* (*lift-start-obs start-tid start-heap-obs*)
let *?start-state = init-fin-lift-state status* (*JVM-start-state P C M vs*)

note ‹*wf-syscls P*› *non-speculative-read*[*OF wf hrt wf-start ka*]
moreover have *ts-ok ?wt-ok* (*thr ?start-state*) (*shr ?start-state*)
using *correct-jvm-state-initial*[*OF wf wf-start*]
by(*simp add*: *correct-jvm-state-def start-state-def split-beta*)
moreover
have *ka-allocated*: *jvm-known-addrs start-tid* ((λ(*mxs*, *mxl0*, *b*) *vs*. (*None*, [([], *Null* # *vs* @ *replicate mxl0 undefined-value*, *fst* (*method P C M*), *M*, *0*)])) (*the* (*snd* (*snd* (*snd* (*method P C M*))))) *vs*) ⊆ *allocated start-heap*
using *ka* **by**(*auto simp add*: *split-beta start-addrs-allocated jvm-known-addrs-def intro*: *start-tid-start-addrs*[*OF* ‹*wf-syscls P*› *ok*])
ultimately have *execd-mthr.if.hb-completion ?start-state* (*lift-start-obs start-tid start-heap-obs*)
by(*rule non-speculative-read-into-hb-completion*)

thus *?thesis* **using** ‹*wf-syscls P*›
proof(*rule sc-legal*)
from *correct-jvm-state-initial*[*OF wf wf-start*]
show *correct-state-ts* Φ (*thr* (*JVM-start-state P C M vs*)) *start-heap*
by(*simp add*: *correct-jvm-state-def start-state-def split-beta*)
qed(*rule ka-allocated*)
qed

lemma *JVM-jmm-consistent*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *hrt*: *heap-read-typeable hconf P*
and *wf-start*: *wf-start-state P C M vs*
and *ka*: ⋃(*ka-Val* ‘ *set vs*) ⊆ *set start-addrs*
shows *jmm-consistent* (*JVMd-*$\mathcal{E}$ *P C M vs status*) *P*
(**is** *jmm-consistent ?*$\mathcal{E}$ *P*)
proof −
interpret *drf ?*$\mathcal{E}$ *P* **using** *assms* **by**(*rule JVM-drf*)
interpret *sc-legal ?*$\mathcal{E}$ *P* **using** *assms* **by**(*rule JVM-sc-legal*)
show *?thesis* **by** *unfold-locales*
qed

lemma *JVM-ex-sc-exec*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *hrt*: *heap-read-typeable hconf P*
and *wf-start*: *wf-start-state P C M vs*
and *ka*: ⋃(*ka-Val* ‘ *set vs*) ⊆ *set start-addrs*
shows ∃ *E ws*. *E* ∈ *JVMd-*$\mathcal{E}$ *P C M vs status* ∧ *P* ⊢ (*E*, *ws*) √ ∧ *sequentially-consistent P* (*E*, *ws*)
(**is** ∃ *E ws*. - ∈ *?*$\mathcal{E}$ ∧ -)
proof −
interpret *jmm*: *executions-sc-hb ?*$\mathcal{E}$ *P* **using** *assms* **by** −(*rule executions-sc*)

let *?start-state* = *init-fin-lift-state status* (*JVM-start-state P C M vs*)
let *?start-mrw* = *mrw-values P Map.empty* (*map snd* (*lift-start-obs start-tid start-heap-obs*))

from *execd-mthr.if.sequential-completion-Runs*[*OF execd-mthr.if.cut-and-update-imp-sc-completion*[*OF JVM-cut-and-update*[*OF assms*]] *ta-seq-consist-convert-RA*]
obtain *ttas* **where** *Red*: *execd-mthr.mthr.if.mthr.Runs P ?start-state ttas*
and *sc*: *ta-seq-consist P ?start-mrw* (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* {|*ta*|}$_o$) *ttas*)) **by** *blast*
let *?E* = *lappend* (*llist-of* (*lift-start-obs start-tid start-heap-obs*)) (*lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) *ttas*))
from *Red* **have** *?E* ∈ *?E* **by**(*blast intro*: *execd-mthr.mthr.if.*$\mathcal{E}$*.intros*)
moreover from *Red* **have** *tsa*: *thread-start-actions-ok ?E*
by(*blast intro*: *execd-mthr.thread-start-actions-ok-init-fin execd-mthr.mthr.if.*$\mathcal{E}$*.intros*)
from *sc* **have** *ta-seq-consist P Map.empty* (*lmap snd ?E*)
unfolding *lmap-lappend-distrib lmap-lconcat llist.map-comp split-def o-def lmap-llist-of map-map snd-conv*
by(*simp add*: *ta-seq-consist-lappend ta-seq-consist-start-heap-obs*)
from *ta-seq-consist-imp-sequentially-consistent*[*OF tsa jmm.*$\mathcal{E}$*-new-actions-for-fun*[*OF* ‹*?E* ∈ *?E*›] *this*]
obtain *ws* **where** *sequentially-consistent P* (*?E*, *ws*) *P* ⊢ (*?E*, *ws*) √ **by** *iprover*
ultimately show *?thesis* **by** *blast*
qed

theorem *JVM-consistent*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *hrt*: *heap-read-typeable hconf P*
and *wf-start*: *wf-start-state P C M vs*
and *ka*: ⋃(*ka-Val* ‘ *set vs*) ⊆ *set start-addrs*
shows ∃ *E ws*. *legal-execution P* (*JVMd-*$\mathcal{E}$ *P C M vs status*) (*E*, *ws*)
proof −
let *?E* = *JVMd-*$\mathcal{E}$ *P C M vs status*
interpret *sc-legal ?E P* **using** *assms* **by**(*rule JVM-sc-legal*)
from *JVM-ex-sc-exec*[*OF assms*]
obtain *E ws* **where** *E* ∈ *?E P* ⊢ (*E*, *ws*) √ *sequentially-consistent P* (*E*, *ws*) **by** *blast*
hence *legal-execution P ?E* (*E*, *ws*) **by**(*rule SC-is-legal*)
thus *?thesis* **by** *blast*
qed

end

One could now also prove that the aggressive JVM satisfies *drf*. The key would be that *welltyped-commute* also holds for *non-speculative* prefixes from start.

end

8.16 JMM heap implementation 1

theory *JMM-Type*
imports
../Common/ExternalCallWF
../Common/ConformThreaded
JMM-Heap
begin

8.16.1 Definitions

The JMM heap only stores type information.

type-synonym *'addr JMM-heap = 'addr ⇀ htype*

translations *(type) 'addr JMM-heap <= (type) 'addr ⇒ htype option*

abbreviation *jmm-empty :: 'addr JMM-heap* **where** *jmm-empty == Map.empty*

definition *jmm-allocate :: 'addr JMM-heap ⇒ htype ⇒ ('addr JMM-heap × 'addr) set*
where *jmm-allocate h hT = (λa. (h(a ↦ hT), a)) ‘ {a. h a = None}*

definition *jmm-typeof-addr :: 'addr JMM-heap ⇒ 'addr ⇀ htype*
where *jmm-typeof-addr h = h*

definition *jmm-heap-read :: 'addr JMM-heap ⇒ 'addr ⇒ addr-loc ⇒ 'addr val ⇒ bool*
where *jmm-heap-read h a ad v = True*

context
notes [[*inductive-internals*]]
begin

inductive *jmm-heap-write :: 'addr JMM-heap ⇒ 'addr ⇒ addr-loc ⇒ 'addr val ⇒ 'addr JMM-heap ⇒ bool*
where *jmm-heap-write h a ad v h*

end

definition *jmm-hconf :: 'm prog ⇒ 'addr JMM-heap ⇒ bool (‹- ⊢jmm - √› [51,51] 50)*
where *P ⊢jmm h √ ⟷ ty-of-htype ‘ ran h ⊆ {T. is-type P T}*

definition *jmm-allocated :: 'addr JMM-heap ⇒ 'addr set*
where *jmm-allocated h = dom (jmm-typeof-addr h)*

definition *jmm-spurious-wakeups :: bool*
where *jmm-spurious-wakeups = True*

lemmas *jmm-heap-ops-defs =*
jmm-allocate-def jmm-typeof-addr-def
jmm-heap-read-def jmm-heap-write-def
jmm-allocated-def jmm-spurious-wakeups-def

type-synonym *'addr thread-id = 'addr*

abbreviation *(input) addr2thread-id :: 'addr ⇒ 'addr thread-id*
where *addr2thread-id ≡ λx. x*

abbreviation *(input) thread-id2addr :: 'addr thread-id ⇒ 'addr*
where *thread-id2addr ≡ λx. x*

interpretation *jmm: heap-base*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr jmm-heap-read jmm-heap-write

.

notation *jmm.hext* (‹- ⊴*jmm* -› [*51*,*51*] *50*)
notation *jmm.conf* (‹-,- ⊢*jmm* - :≤ -› [*51*,*51*,*51*,*51*] *50*)
notation *jmm.addr-loc-type* (‹-,- ⊢*jmm* -@- : -› [*50*, *50*, *50*, *50*, *50*] *51*)
notation *jmm.confs* (‹-,- ⊢*jmm* - [:≤] -› [*51*,*51*,*51*,*51*] *50*)
notation *jmm.tconf* (‹-,- ⊢*jmm* - √*t*› [*51*,*51*,*51*] *50*)

Now a variation of the JMM with a different read operation that permits to read only type-conformant values

interpretation *jmm'*: *heap-base*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr jmm.heap-read-typed P jmm-heap-write
for *P* **.**

notation *jmm'.hext* (‹- ⊴*jmm''* -› [*51*,*51*] *50*)
notation *jmm'.conf* (‹-,- ⊢*jmm''* - :≤ -› [*51*,*51*,*51*,*51*] *50*)
notation *jmm'.addr-loc-type* (‹-,- ⊢*jmm''* -@- : -› [*50*, *50*, *50*, *50*, *50*] *51*)
notation *jmm'.confs* (‹-,- ⊢*jmm''* - [:≤] -› [*51*,*51*,*51*,*51*] *50*)
notation *jmm'.tconf* (‹-,- ⊢*jmm''* - √*t*› [*51*,*51*,*51*] *50*)

8.16.2 Heap locale interpretations

8.16.3 Locale *heap*

lemma *jmm-heap*: *heap addr2thread-id thread-id2addr jmm-allocate jmm-typeof-addr jmm-heap-write P*
proof
fix *h' a h hT*
assume (*h'*, *a*) ∈ *jmm-allocate h hT*
thus *jmm-typeof-addr h' a* = ⌊*hT*⌋
by(*auto simp add*: *jmm-heap-ops-defs*)
next
fix *h'* :: (*'addr* :: *addr*) *JMM-heap* **and** *h hT a*
assume (*h'*, *a*) ∈ *jmm-allocate h hT*
thus *h* ⊴*jmm h'*
by(*fastforce simp add*: *jmm-heap-ops-defs intro*: *jmm.hextI*)
next
fix *h a al v* **and** *h'* :: (*'addr* :: *addr*) *JMM-heap*
assume *jmm-heap-write h a al v h'*
thus *h* ⊴*jmm h'* **by** *cases auto*
qed *simp*

interpretation *jmm*: *heap*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr jmm-heap-read jmm-heap-write
P
for *P*
by(*rule jmm-heap*)

declare *jmm.typeof-addr-thread-id2-addr-addr2thread-id* [*simp del*]

lemmas *jmm′-heap = jmm-heap*

interpretation *jmm′*: *heap*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr jmm.heap-read-typed P jmm-heap-write
P
for *P*
by(*rule jmm′-heap*)

declare *jmm′.typeof-addr-thread-id2-addr-addr2thread-id* [*simp del*]

8.16.4 Locale *heap-conf*

interpretation *jmm*: *heap-conf-base*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr jmm-heap-read jmm-heap-write jmm-hconf P
P
for *P* **.**

abbreviation (*input*) *jmm′-hconf* :: *′m prog* ⇒ *′addr JMM-heap* ⇒ *bool* (‹- ⊢*jmm″* - √› [*51*,*51*] *50*)
where *jmm′-hconf* == *jmm-hconf*

interpretation *jmm′*: *heap-conf-base*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr jmm.heap-read-typed P jmm-heap-write jmm′-hconf P
P
for *P* **.**

abbreviation *jmm-heap-read-typeable* :: (*′addr* :: *addr*) *itself* ⇒ *′m prog* ⇒ *bool*
where *jmm-heap-read-typeable tytok P* ≡ *jmm.heap-read-typeable* (*jmm-hconf P* :: *′addr JMM-heap* ⇒ *bool*) *P*

abbreviation *jmm′-heap-read-typeable* :: (*′addr* :: *addr*) *itself* ⇒ *′m prog* ⇒ *bool*
where *jmm′-heap-read-typeable tytok P* ≡ *jmm′.heap-read-typeable TYPE*(*′m*) *P* (*jmm-hconf P* :: *′addr JMM-heap* ⇒ *bool*) *P*

lemma *jmm-heap-read-typeable*: *jmm-heap-read-typeable tytok P*
by(*rule jmm.heap-read-typeableI*)(*simp add*: *jmm-heap-read-def*)

lemma *jmm′-heap-read-typeable*: *jmm′-heap-read-typeable tytok P*
by(*rule jmm′.heap-read-typeableI*)(*auto simp add*: *jmm.heap-read-typed-def jmm-heap-read-def dest*: *jmm′.addr-loc-type-fun*)

lemma *jmm-heap-conf*:
heap-conf addr2thread-id thread-id2addr jmm-empty jmm-allocate jmm-typeof-addr jmm-heap-write (*jmm-hconf P*) *P*
proof
show *P* ⊢*jmm jmm-empty* √
by(*simp add*: *jmm-hconf-def*)
next
fix *h a hT*

assume *jmm-typeof-addr h a* = ⌊*hT*⌋ *P* ⊢*jmm h* √
thus *is-htype P hT* **by**(*auto simp add*: *jmm-hconf-def jmm-heap-ops-defs intro*: *ranI*)
next
fix *h' h hT a*
assume (*h'*, *a*) ∈ *jmm-allocate h hT P* ⊢*jmm h* √ *is-htype P hT*
thus *P* ⊢*jmm h'* √
by(*fastforce simp add*: *jmm-hconf-def jmm-heap-ops-defs ran-def split*: *if-split-asm*)
next
fix *h a al v h' T*
assume *jmm-heap-write h a al v h' P* ⊢*jmm h* √
and *jmm.addr-loc-type P h a al T* **and** *P*,*h* ⊢*jmm v* :≤ *T*
thus *P* ⊢*jmm h'* √ **by**(*cases*) *simp*
qed

interpretation *jmm*: *heap-conf*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr jmm-heap-read jmm-heap-write jmm-hconf P
P
for *P*
by(*rule jmm-heap-conf*)

lemmas *jmm'-heap-conf* = *jmm-heap-conf*

interpretation *jmm'*: *heap-conf*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr jmm.heap-read-typed P jmm-heap-write jmm'-hconf P
P
for *P*
by(*rule jmm'-heap-conf*)

8.16.5 Locale *heap-progress*

lemma *jmm-heap-progress*:
heap-progress addr2thread-id thread-id2addr jmm-empty jmm-allocate jmm-typeof-addr jmm-heap-read jmm-heap-write (*jmm-hconf P*) *P*
proof
fix *h a al T*
assume *P* ⊢*jmm h* √
and *al*: *jmm.addr-loc-type P h a al T*
show ∃ *v*. *jmm-heap-read h a al v* ∧ *P*,*h* ⊢*jmm v* :≤ *T*
using *jmm.defval-conf*[*of P h T*] **unfolding** *jmm-heap-ops-defs* **by** *blast*
next
fix *h a al T v*
assume *jmm.addr-loc-type P h a al T*
show ∃ *h'*. *jmm-heap-write h a al v h'*
by(*auto intro*: *jmm-heap-write.intros*)
qed

interpretation *jmm*: *heap-progress*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr jmm-heap-read jmm-heap-write jmm-hconf P

P
for *P*
by(*rule jmm-heap-progress*)

lemma *jmm'-heap-progress*:
heap-progress addr2thread-id thread-id2addr jmm-empty jmm-allocate jmm-typeof-addr (*jmm.heap-read-typed P*) *jmm-heap-write* (*jmm'-hconf P*) *P*
proof
fix *h a al T*
assume *P* ⊢*jmm'* *h* √
and *al*: *jmm'.addr-loc-type P h a al T*
thus ∃ *v*. *jmm.heap-read-typed P h a al v* ∧ *P*,*h* ⊢*jmm'* *v* :≤ *T*
unfolding *jmm.heap-read-typed-def jmm-heap-read-def*
by(*auto dest*: *jmm'.addr-loc-type-fun intro*: *jmm'.defval-conf*)
next
fix *h a al T v*
assume *jmm'.addr-loc-type P h a al T*
and *P*,*h* ⊢*jmm'* *v* :≤ *T*
thus ∃ *h'*. *jmm-heap-write h a al v h'*
by(*auto intro*: *jmm-heap-write.intros*)
qed

interpretation *jmm'*: *heap-progress*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr jmm.heap-read-typed P jmm-heap-write jmm'-hconf P
P
for *P*
by(*rule jmm'-heap-progress*)

8.16.6 Locale *heap-conf-read*

lemma *jmm'-heap-conf-read*:
heap-conf-read addr2thread-id thread-id2addr jmm-empty jmm-allocate jmm-typeof-addr (*jmm.heap-read-typed P*) *jmm-heap-write* (*jmm'-hconf P*) *P*
by(*rule jmm.heap-conf-read-heap-read-typed*)

interpretation *jmm'*: *heap-conf-read*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr jmm.heap-read-typed P jmm-heap-write jmm'-hconf P
P
for *P*
by(*rule jmm'-heap-conf-read*)

interpretation *jmm'*: *heap-typesafe*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr jmm.heap-read-typed P jmm-heap-write jmm'-hconf P
P
for *P*
..

8.16.7 Locale *allocated-heap*

lemma *jmm-allocated-heap*:
allocated-heap addr2thread-id thread-id2addr jmm-empty jmm-allocate jmm-typeof-addr jmm-heap-write jmm-allocated P
proof
show *jmm-allocated jmm-empty* = {} **by**(*auto simp add*: *jmm-heap-ops-defs*)
next
fix *h' a h hT*
assume (*h'*, *a*) ∈ *jmm-allocate h hT*
thus *jmm-allocated h'* = *insert a* (*jmm-allocated h*) ∧ *a* ∉ *jmm-allocated h*
by(*auto simp add*: *jmm-heap-ops-defs split*: *if-split-asm*)
next
fix *h a al v h'*
assume *jmm-heap-write h a al v h'*
thus *jmm-allocated h'* = *jmm-allocated h* **by** *cases simp*
qed

interpretation *jmm*: *allocated-heap*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr jmm-heap-read jmm-heap-write
jmm-allocated
P
for *P*
by(*rule jmm-allocated-heap*)

lemmas *jmm'-allocated-heap* = *jmm-allocated-heap*

interpretation *jmm'*: *allocated-heap*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr jmm.heap-read-typed P jmm-heap-write
jmm-allocated
P
for *P*
by(*rule jmm'-allocated-heap*)

8.16.8 Syntax translations

notation *jmm'.external-WT'* (‹-,- ⊢*jmm''* (-▪-'(-')) : -› [*50*,*0*,*0*,*0*,*50*] *60*)

abbreviation *jmm'-red-external* ::
'm prog ⇒ *'addr thread-id* ⇒ *'addr JMM-heap* ⇒ *'addr* ⇒ *mname* ⇒ *'addr val list*
⇒ (*'addr* :: *addr*, *'addr thread-id*, *'addr JMM-heap*) *external-thread-action*
⇒ *'addr extCallRet* ⇒ *'addr JMM-heap* ⇒ *bool*
where *jmm'-red-external P* ≡ *jmm'.red-external* (*TYPE*('*m*)) *P P*

abbreviation *jmm'-red-external-syntax* ::
'm prog ⇒ *'addr thread-id* ⇒ *'addr* ⇒ *mname* ⇒ *'addr val list* ⇒ *'addr JMM-heap*
⇒ (*'addr* :: *addr*, *'addr thread-id*, *'addr JMM-heap*) *external-thread-action*
⇒ *'addr extCallRet* ⇒ *'addr JMM-heap* ⇒ *bool*
(‹-,- ⊢*jmm''* (‹(-▪-'(-')),/-›) −-→*ext* (‹(-),/(-)›)› [*50*, *0*, *0*, *0*, *0*, *0*, *0*, *0*, *0*] *51*)
where
P,*t* ⊢*jmm'* ⟨*a*▪*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h'*⟩ ≡ *jmm'-red-external P t h a M vs ta va h'*

abbreviation *jmm'-red-external-aggr* ::
'm prog ⇒ *'addr thread-id* ⇒ *'addr* ⇒ *mname* ⇒ *'addr val list* ⇒ *'addr JMM-heap*
⇒ ((*'addr* :: *addr*, *'addr thread-id*, *'addr JMM-heap*) *external-thread-action* × *'addr extCallRet* × *'addr JMM-heap*) *set*
where *jmm'-red-external-aggr P* ≡ *jmm'.red-external-aggr TYPE*('*m*) *P P*

abbreviation *jmm'-heap-copy-loc* ::
'm prog ⇒ *'addr* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr JMM-heap*
⇒ (*'addr* :: *addr*, *'addr thread-id*) *obs-event list* ⇒ *'addr JMM-heap* ⇒ *bool*
where *jmm'-heap-copy-loc* ≡ *jmm'.heap-copy-loc TYPE*('*m*)

abbreviation *jmm'-heap-copies* ::
'm prog ⇒ *'addr* ⇒ *'addr* ⇒ *addr-loc list* ⇒ *'addr JMM-heap*
⇒ (*'addr* :: *addr*, *'addr thread-id*) *obs-event list* ⇒ *'addr JMM-heap* ⇒ *bool*
where *jmm'-heap-copies* ≡ *jmm'.heap-copies TYPE*('*m*)

abbreviation *jmm'-heap-clone* ::
'm prog ⇒ *'addr JMM-heap* ⇒ *'addr* ⇒ *'addr JMM-heap*
⇒ ((*'addr* :: *addr*, *'addr thread-id*) *obs-event list* × *'addr*) *option* ⇒ *bool*
where *jmm'-heap-clone P* ≡ *jmm'.heap-clone TYPE*('*m*) *P P*

end

8.17 Compiler correctness for the JMM

theory *JMM-Compiler* **imports**
JMM-J
JMM-JVM
../Compiler/Correctness
../Framework/FWBisimLift
begin

lemma *action-loc-aux-compP* [*simp*]: *action-loc-aux* (*compP f P*) = *action-loc-aux P*
by(*auto 4 4 elim*!: *action-loc-aux-cases*)

lemma *action-loc-compP*: *action-loc* (*compP f P*) = *action-loc P*
by *simp*

lemma *is-volatile-compP* [*simp*]: *is-volatile* (*compP f P*) = *is-volatile P*
proof(*rule ext*)
fix *hT*
show *is-volatile* (*compP f P*) *hT* = *is-volatile P hT*
by(*cases hT*) *simp-all*
qed

lemma *saction-compP* [*simp*]: *saction* (*compP f P*) = *saction P*
by(*simp add*: *saction.simps fun-eq-iff*)

lemma *sactions-compP* [*simp*]: *sactions* (*compP f P*) = *sactions P*
by(*rule ext*)(*simp only*: *sactions-def*, *simp*)

lemma *addr-locs-compP* [*simp*]: *addr-locs* (*compP f P*) = *addr-locs P*

by(*rule ext*)(*case-tac x*, *simp-all*)

lemma *syncronizes-with-compP* [*simp*]: *synchronizes-with* (*compP f P*) = *synchronizes-with P*
by(*simp add*: *synchronizes-with.simps fun-eq-iff*)

lemma *sync-order-compP* [*simp*]: *sync-order* (*compP f P*) = *sync-order P*
by(*simp add*: *sync-order-def fun-eq-iff*)

lemma *sync-with-compP* [*simp*]: *sync-with* (*compP f P*) = *sync-with P*
by(*simp add*: *sync-with-def fun-eq-iff*)

lemma *po-sw-compP* [*simp*]: *po-sw* (*compP f P*) = *po-sw P*
by(*simp add*: *po-sw-def fun-eq-iff*)

lemma *happens-before-compP*: *happens-before* (*compP f P*) = *happens-before P*
by *simp*

lemma *addr-loc-default-compP* [*simp*]: *addr-loc-default* (*compP f P*) = *addr-loc-default P*
proof(*intro ext*)
  fix *hT al*
  show *addr-loc-default* (*compP f P*) *hT al* = *addr-loc-default P hT al*
    by(*cases* (*P*, *hT*, *al*) *rule*: *addr-loc-default.cases*) *simp-all*
qed

lemma *value-written-aux-compP* [*simp*]: *value-written-aux* (*compP f P*) = *value-written-aux P*
proof(*intro ext*)
  fix *a al*
  show *value-written-aux* (*compP f P*) *a al* = *value-written-aux P a al*
    by(*cases* (*P*, *a*, *al*) *rule*: *value-written-aux.cases*)(*simp-all add*: *value-written-aux.simps*)
qed

lemma *value-written-compP* [*simp*]: *value-written* (*compP f P*) = *value-written P*
by(*simp add*: *fun-eq-iff value-written.simps*)

lemma *is-write-seen-compP* [*simp*]: *is-write-seen* (*compP f P*) = *is-write-seen P*
by(*simp add*: *fun-eq-iff is-write-seen-def*)

lemma *justification-well-formed-compP* [*simp*]:
  justification-well-formed (*compP f P*) = *justification-well-formed P*
by(*simp add*: *fun-eq-iff justification-well-formed-def*)

lemma *happens-before-committed-compP* [*simp*]:
  happens-before-committed (*compP f P*) = *happens-before-committed P*
by(*simp add*: *fun-eq-iff happens-before-committed-def*)

lemma *happens-before-committed-weak-compP* [*simp*]:
  happens-before-committed-weak (*compP f P*) = *happens-before-committed-weak P*
by(*simp add*: *fun-eq-iff happens-before-committed-weak-def*)

lemma *sync-order-committed-compP* [*simp*]:
  sync-order-committed (*compP f P*) = *sync-order-committed P*
by(*simp add*: *fun-eq-iff sync-order-committed-def*)

lemma *value-written-committed-compP* [*simp*]:

value-written-committed (*compP f P*) = *value-written-committed P*
by(*simp add*: *fun-eq-iff value-written-committed-def*)

lemma *uncommitted-reads-see-hb-compP* [*simp*]:
uncommitted-reads-see-hb (*compP f P*) = *uncommitted-reads-see-hb P*
by(*simp add*: *fun-eq-iff uncommitted-reads-see-hb-def*)

lemma *external-actions-committed-compP* [*simp*]:
external-actions-committed (*compP f P*) = *external-actions-committed P*
by(*simp add*: *fun-eq-iff external-actions-committed-def*)

lemma *is-justified-by-compP* [*simp*]: *is-justified-by* (*compP f P*) = *is-justified-by P*
by(*simp add*: *fun-eq-iff is-justified-by.simps*)

lemma *is-weakly-justified-by-compP* [*simp*]: *is-weakly-justified-by* (*compP f P*) = *is-weakly-justified-by P*
by(*simp add*: *fun-eq-iff is-weakly-justified-by.simps*)

lemma *legal-execution-compP*: *legal-execution* (*compP f P*) = *legal-execution P*
by(*simp add*: *fun-eq-iff gen-legal-execution.simps*)

lemma *weakly-legal-execution-compP*: *weakly-legal-execution* (*compP f P*) = *weakly-legal-execution P*
by(*simp add*: *fun-eq-iff gen-legal-execution.simps*)

lemma *most-recent-write-for-compP* [*simp*]:
most-recent-write-for (*compP f P*) = *most-recent-write-for P*
by(*simp add*: *fun-eq-iff most-recent-write-for.simps*)

lemma *sequentially-consistent-compP* [*simp*]:
sequentially-consistent (*compP f P*) = *sequentially-consistent P*
by(*simp add*: *sequentially-consistent-def split-beta*)

lemma *conflict-compP* [*simp*]: *non-volatile-conflict* (*compP f P*) = *non-volatile-conflict P*
by(*simp add*: *fun-eq-iff non-volatile-conflict-def*)

lemma *correctly-synchronized-compP* [*simp*]:
correctly-synchronized (*compP f P*) = *correctly-synchronized P*
by(*simp add*: *fun-eq-iff correctly-synchronized-def*)

lemma (**in** *heap-base*) *heap-read-typed-compP* [*simp*]:
heap-read-typed (*compP f P*) = *heap-read-typed P*
by(*intro ext*)(*simp add*: *heap-read-typed-def*)

context *J-JVM-heap-conf-base* **begin**

definition *if-bisimJ2JVM* ::
(('*addr*,'*thread-id*,*status* × '*addr expr*×'*addr locals*,'*heap*,'*addr*) *state*,
('*addr*,'*thread-id*,*status* × '*addr option* × '*addr frame list*,'*heap*,'*addr*) *state*) *bisim*
where
if-bisimJ2JVM =
FWbisimulation-base.mbisim red-red0.init-fin-bisim red-red0.init-fin-bisim-wait ∘$_B$
FWbisimulation-base.mbisim red0-Red1'.init-fin-bisim red0-Red1'.init-fin-bisim-wait ∘$_B$
if-mbisim-Red1'-Red1 ∘$_B$
FWbisimulation-base.mbisim Red1-execd.init-fin-bisim Red1-execd.init-fin-bisim-wait

definition *if-tlsimJ2JVM* ::
(*'thread-id* × (*'addr*, *'thread-id*, *status* × *'addr expr* × *'addr locals*,
'heap, *'addr*, (*'addr*, *'thread-id*) *obs-event action*) *thread-action*,
'thread-id × (*'addr*, *'thread-id*, *status* × *'addr jvm-thread-state*,
'heap, *'addr*, (*'addr*, *'thread-id*) *obs-event action*) *thread-action*) *bisim*
where
if-tlsimJ2JVM =
FWbisimulation-base.mta-bisim red-red0.init-fin-bisim $\circ_B$
FWbisimulation-base.mta-bisim red0-Red1'.init-fin-bisim $\circ_B$ (=) $\circ_B$
FWbisimulation-base.mta-bisim Red1-execd.init-fin-bisim

end

sublocale *J-JVM-conf-read* < *red-mthr*: *if-τmultithreaded-wf final-expr mred P convert-RA τMOVE P*
by(*unfold-locales*)

sublocale *J-JVM-conf-read* < *execd-mthr*:
if-τmultithreaded-wf
JVM-final
mexecd (*compP2* (*compP1 P*))
convert-RA
τMOVE2 (*compP2* (*compP1 P*))
by(*unfold-locales*)

context *J-JVM-conf-read* **begin**

theorem *if-bisimJ2JVM-weak-bisim*:
assumes *wf*: *wf-J-prog P*
shows *delay-bisimulation-diverge-final*
(*red-mthr.mthr.if.redT P*) (*execd-mthr.mthr.if.redT* (*J2JVM P*)) *if-bisimJ2JVM if-tlsimJ2JVM*
red-mthr.if.mτmove execd-mthr.if.mτmove red-mthr.mthr.if.mfinal execd-mthr.mthr.if.mfinal
apply (*simp only*: *if-bisimJ2JVM-def if-tlsimJ2JVM-def J2JVM-def o-apply*)
apply(*rule delay-bisimulation-diverge-final-compose*)
apply(*rule FWdelay-bisimulation-diverge.mthr-delay-bisimulation-diverge-final*)
apply(*rule FWdelay-bisimulation-diverge.init-fin-FWdelay-bisimulation-diverge*)
apply(*rule red-red0-FWbisim*[*OF wf-prog-wwf-prog*[*OF wf*]])
apply(*rule delay-bisimulation-diverge-final-compose*)
apply(*rule FWdelay-bisimulation-diverge.mthr-delay-bisimulation-diverge-final*)
apply(*rule FWdelay-bisimulation-diverge.init-fin-FWdelay-bisimulation-diverge*)
apply(*rule red0-Red1'-FWweak-bisim*[*OF wf*])
apply(*rule delay-bisimulation-diverge-final-compose*)
apply(*rule delay-bisimulation-diverge-final.intro*)
apply(*rule bisimulation-into-delay.delay-bisimulation*)
apply(*rule if-Red1'-Red1-bisim-into-weak*[*OF compP1-pres-wf*[*OF wf*]])
apply(*rule bisimulation-final.delay-bisimulation-final-base*)
apply(*rule if-Red1'-Red1-bisimulation-final*[*OF compP1-pres-wf*[*OF wf*]])
apply(*rule FWdelay-bisimulation-diverge.mthr-delay-bisimulation-diverge-final*)
apply(*rule FWdelay-bisimulation-diverge.init-fin-FWdelay-bisimulation-diverge*)
apply(*rule Red1-exec1-FWwbisim*[*OF compP1-pres-wf*[*OF wf*]])
done

lemma *if-bisimJ2JVM-start*:

```
  assumes wf: wf-J-prog P
  and wf-start: wf-start-state P C M vs
  shows if-bisimJ2JVM (init-fin-lift-state Running (J-start-state P C M vs))
                      (init-fin-lift-state Running (JVM-start-state (J2JVM P) C M vs))
using assms
unfolding if-bisimJ2JVM-def J2JVM-def o-apply
apply(intro bisim-composeI)
   apply(rule FWbisimulation-base.init-fin-lift-state-mbisimI)
   apply(erule (1) bisim-J-J0-start[OF wf-prog-wwf-prog])
  apply(rule FWbisimulation-base.init-fin-lift-state-mbisimI)
  apply(erule (1) bisim-J0-J1-start)
 apply(erule if-bisim-J1-J1-start[OF compP1-pres-wf])
 apply simp
apply(rule FWbisimulation-base.init-fin-lift-state-mbisimI)
apply(erule bisim-J1-JVM-start[OF compP1-pres-wf])
apply simp
done

lemma red-Runs-eq-mexecd-Runs:
  fixes C M vs
  defines s: s ≡ init-fin-lift-state Running (J-start-state P C M vs)
  and comps: cs ≡ init-fin-lift-state Running (JVM-start-state (J2JVM P) C M vs)
  assumes wf: wf-J-prog P
  and wf-start: wf-start-state P C M vs
  shows red-mthr.mthr.if.ℰ P s = execd-mthr.mthr.if.ℰ (J2JVM P) cs
proof −
  from wf wf-start have bisim: if-bisimJ2JVM s cs
    unfolding s comps by(rule if-bisimJ2JVM-start)

  interpret divfin: delay-bisimulation-diverge-final
    red-mthr.mthr.if.redT P
    execd-mthr.mthr.if.redT (J2JVM P)
    if-bisimJ2JVM
    if-tlsimJ2JVM
    red-mthr.if.mτmove
    execd-mthr.if.mτmove
    red-mthr.mthr.if.mfinal
    execd-mthr.mthr.if.mfinal
    using wf by(rule if-bisimJ2JVM-weak-bisim)

  show ?thesis (is ?lhs = ?rhs)
  proof(intro equalityI subsetI)
    fix E
    assume E ∈ ?lhs
   then obtain E' where E: E = lconcat (lmap (λ(t, ta). llist-of (map (Pair t) {|ta|}_o)) (llist-of-tllist
E'))
      and E': red-mthr.if.mthr.τRuns s E'
      unfolding red-mthr.if.ℰ-conv-Runs by blast
    from divfin.simulation-τRuns1[OF bisim E']
    obtain E'' where E'': execd-mthr.if.mthr.τRuns cs E''
      and tlsim: tllist-all2 if-tlsimJ2JVM (option.rel-option if-bisimJ2JVM) E' E''
      unfolding J2JVM-def o-apply by blast
    let ?E = lconcat (lmap (λ(t, ta). llist-of (map (Pair t) {|ta|}_o)) (llist-of-tllist E''))
    from tlsim have llist-all2 if-tlsimJ2JVM (llist-of-tllist E') (llist-of-tllist E'')
```

by(*rule tllist-all2D-llist-all2-llist-of-tllist*)
hence *llist-all2* (=) (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) (*llist-of-tllist E'*))
(*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) (*llist-of-tllist E''*))
unfolding *llist-all2-lmap1 llist-all2-lmap2*
by(*rule llist-all2-mono*)(*auto simp add*: *if-tlsimJ2JVM-def FWbisimulation-base.mta-bisim-def ta-bisim-def*)
hence *?E* = *E* **unfolding** *llist.rel-eq E* **by** *simp*
also from *E''* **have** *?E* ∈ *?rhs* **unfolding** *J2JVM-def o-apply execd-mthr.if.*$\mathcal{E}$*-conv-Runs* **by** *blast*
finally (*subst*) **show** *E* ∈ *?rhs* **.**
next
fix *E*
assume *E* ∈ *?rhs*
then obtain *E'* **where** *E*: *E* = *lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) (*llist-of-tllist E'*))
and *E'*: *execd-mthr.if.mthr.*τ*Runs cs E'*
unfolding *execd-mthr.if.*$\mathcal{E}$*-conv-Runs J2JVM-def o-apply* **by** *blast*
from *divfin.simulation-*τ*Runs2*[*OF bisim*, *simplified J2JVM-def o-apply*, *OF E'*]
obtain *E''* **where** *E''*: *red-mthr.if.mthr.*τ*Runs s E''*
and *tlsim*: *tllist-all2 if-tlsimJ2JVM* (*option.rel-option if-bisimJ2JVM*) *E''* *E'* **by** *blast*
let *?E* = *lconcat* (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) (*llist-of-tllist E''*))
from *tlsim* **have** *llist-all2 if-tlsimJ2JVM* (*llist-of-tllist E''*) (*llist-of-tllist E'*)
by(*rule tllist-all2D-llist-all2-llist-of-tllist*)
hence *llist-all2* (=) (*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) (*llist-of-tllist E''*))
(*lmap* (λ(*t*, *ta*). *llist-of* (*map* (*Pair t*) {|*ta*|}$_o$)) (*llist-of-tllist E'*))
unfolding *llist-all2-lmap1 llist-all2-lmap2*
by(*rule llist-all2-mono*)(*auto simp add*: *if-tlsimJ2JVM-def FWbisimulation-base.mta-bisim-def ta-bisim-def*)
hence *?E* = *E* **unfolding** *llist.rel-eq E* **by** *simp*
also from *E''* **have** *?E* ∈ *?lhs* **unfolding** *red-mthr.if.*$\mathcal{E}$*-conv-Runs* **by** *blast*
finally (*subst*) **show** *E* ∈ *?lhs* **.**
qed
qed

lemma *red-*$\mathcal{E}$*-eq-mexecd-*$\mathcal{E}$:
⟦ *wf-J-prog P*; *wf-start-state P C M vs* ⟧
⟹ *J-*$\mathcal{E}$ *P C M vs Running* = *JVMd-*$\mathcal{E}$ (*J2JVM P*) *C M vs Running*
by(*simp only*: *red-Runs-eq-mexecd-Runs*)

theorem *J2JVM-jmm-correct*:
assumes *wf*: *wf-J-prog P*
and *wf-start*: *wf-start-state P C M vs*
shows *legal-execution P* (*J-*$\mathcal{E}$ *P C M vs Running*) (*E*, *ws*) ⟷
legal-execution (*J2JVM P*) (*JVMd-*$\mathcal{E}$ (*J2JVM P*) *C M vs Running*) (*E*, *ws*)
by(*simp only*: *red-*$\mathcal{E}$*-eq-mexecd-*$\mathcal{E}$[*OF assms*] *J2JVM-def o-apply compP1-def compP2-def legal-execution-compP*)

theorem *J2JVM-jmm-correct-weak*:
assumes *wf*: *wf-J-prog P*
and *wf-start*: *wf-start-state P C M vs*
shows *weakly-legal-execution P* (*J-*$\mathcal{E}$ *P C M vs Running*) (*E*, *ws*) ⟷
weakly-legal-execution (*J2JVM P*) (*JVMd-*$\mathcal{E}$ (*J2JVM P*) *C M vs Running*) (*E*, *ws*)
by(*simp only*: *red-*$\mathcal{E}$*-eq-mexecd-*$\mathcal{E}$[*OF assms*] *J2JVM-def o-apply compP1-def compP2-def weakly-legal-execution-compP*)

theorem *J2JVM-jmm-correctly-synchronized*:
assumes *wf*: *wf-J-prog P*

and *wf-start*: *wf-start-state P C M vs*
shows *correctly-synchronized* (*J2JVM P*) (*JVMd-*$\mathcal{E}$ (*J2JVM P*) *C M vs Running*) $\longleftrightarrow$
correctly-synchronized P (*J-*$\mathcal{E}$ *P C M vs Running*)
by(*simp only*: *red-*$\mathcal{E}$*-eq-mexecd-*$\mathcal{E}$[*OF assms*] *J2JVM-def o-apply compP1-def compP2-def correctly-synchronized-com*

end

end

8.18 JMM heap implementation 2

theory *JMM-Type2*
imports
../Common/ExternalCallWF
../Common/ConformThreaded
JMM-Heap
begin

8.18.1 Definitions

datatype *addr* = *Address htype nat* — heap type and sequence number

lemma *rec-addr-conv-case-addr* [*simp*]: *rec-addr* = *case-addr*
by(*auto intro*!: *ext split*: *addr.split*)

instantiation *addr* :: *addr* **begin**
definition *hash-addr* (*a* :: *addr*) = (*case a of Address ht n* ⇒ *int n*)
definition *monitor-finfun-to-list* (*ls* :: *addr* ⇒*f nat*) = (*SOME xs. set xs* = {*x. finfun-dom ls* $ *x* })
instance
proof
fix *ls* :: *addr* ⇒*f nat*
show *set* (*monitor-finfun-to-list ls*) = *Collect* (($) (*finfun-dom ls*))
unfolding *monitor-finfun-to-list-addr-def*
using *finite-list*[*OF finite-finfun-dom*, **where** *?f.1* = *ls*]
by(*rule someI-ex*)
qed
end

primrec *the-Address* :: *addr* ⇒ *htype* × *nat*
where *the-Address* (*Address hT n*) = (*hT*, *n*)

The JMM heap only stores which sequence numbers of a given *htype* have already been allocated.

type-synonym *JMM-heap* = *htype* ⇒ *nat set*

translations (*type*) *JMM-heap* <= (*type*) *htype* ⇒ *nat set*

definition *jmm-allocate* :: *JMM-heap* ⇒ *htype* ⇒ (*JMM-heap* × *addr*) *set*
where *jmm-allocate h hT* = (*let hhT* = *h hT in* (λ*n*. (*h*(*hT* := *insert n hhT*), *Address hT n*)) ‘ (− *hhT*))

abbreviation *jmm-empty* :: *JMM-heap* **where** *jmm-empty* == (λ-. {})

definition *jmm-typeof-addr* :: *'m prog* ⇒ *JMM-heap* ⇒ *addr* ⇀ *htype*

where *jmm-typeof-addr P h* = (λ*hT*. *if is-htype P hT then Some hT else None*) ∘ *fst* ∘ *the-Address*

definition *jmm-typeof-addr'* :: *'m prog* ⇒ *addr* ⇀ *htype*
where *jmm-typeof-addr' P* = (λ*hT*. *if is-htype P hT then Some hT else None*) ∘ *fst* ∘ *the-Address*

lemma *jmm-typeof-addr'-conv-jmm-type-addr*: *jmm-typeof-addr' P* = *jmm-typeof-addr P h*
by(*simp add*: *jmm-typeof-addr-def jmm-typeof-addr'-def*)

lemma *jmm-typeof-addr'-conv-jmm-typeof-addr*: (λ-. *jmm-typeof-addr' P*) = *jmm-typeof-addr P*
by(*simp add*: *jmm-typeof-addr-def jmm-typeof-addr'-def fun-eq-iff*)

lemma *jmm-typeof-addr-conv-jmm-typeof-addr'*: *jmm-typeof-addr* = (λ*P* -. *jmm-typeof-addr' P*)
by(*simp add*: *jmm-typeof-addr'-conv-jmm-typeof-addr*)

definition *jmm-heap-read* :: *JMM-heap* ⇒ *addr* ⇒ *addr-loc* ⇒ *addr val* ⇒ *bool*
where *jmm-heap-read h a ad v* = *True*

context
  notes [[*inductive-internals*]]
begin

inductive *jmm-heap-write* :: *JMM-heap* ⇒ *addr* ⇒ *addr-loc* ⇒ *addr val* ⇒ *JMM-heap* ⇒ *bool*
where *jmm-heap-write h a ad v h*

end

definition *jmm-hconf* :: *JMM-heap* ⇒ *bool*
where *jmm-hconf h* ⟷ *True*

definition *jmm-allocated* :: *JMM-heap* ⇒ *addr set*
where *jmm-allocated h* = {*Address CTn n*|*CTn n*. *n* ∈ *h CTn*}

definition *jmm-spurious-wakeups* :: *bool*
where *jmm-spurious-wakeups* = *True*

lemmas *jmm-heap-ops-defs* =
  jmm-allocate-def jmm-typeof-addr-def
  jmm-heap-read-def jmm-heap-write-def
  jmm-allocated-def jmm-spurious-wakeups-def

type-synonym *thread-id* = *addr*

abbreviation (*input*) *addr2thread-id* :: *addr* ⇒ *thread-id*
where *addr2thread-id* ≡ λ*x*. *x*

abbreviation (*input*) *thread-id2addr* :: *thread-id* ⇒ *addr*
where *thread-id2addr* ≡ λ*x*. *x*

interpretation *jmm*: *heap-base*
  addr2thread-id thread-id2addr
  jmm-spurious-wakeups
  jmm-empty jmm-allocate jmm-typeof-addr P jmm-heap-read jmm-heap-write
  for *P*
.

abbreviation *jmm-hext* :: *'m prog* ⇒ *JMM-heap* ⇒ *JMM-heap* ⇒ *bool* (‹- ⊢ - ⊴*jmm* -› [*51*,*51*,*51*] *50*)
where *jmm-hext* ≡ *jmm.hext TYPE*('*m*)

abbreviation *jmm-conf* :: *'m prog* ⇒ *JMM-heap* ⇒ *addr val* ⇒ *ty* ⇒ *bool*
(‹-,- ⊢*jmm* - :≤ -› [*51*,*51*,*51*,*51*] *50*)
where *jmm-conf P* ≡ *jmm.conf TYPE*('*m*) *P P*

abbreviation *jmm-addr-loc-type* :: *'m prog* ⇒ *JMM-heap* ⇒ *addr* ⇒ *addr-loc* ⇒ *ty* ⇒ *bool*
(‹-,- ⊢*jmm* -@- : -› [*50*, *50*, *50*, *50*, *50*] *51*)
where *jmm-addr-loc-type P* ≡ *jmm.addr-loc-type TYPE*('*m*) *P P*

abbreviation *jmm-confs* :: *'m prog* ⇒ *JMM-heap* ⇒ *addr val list* ⇒ *ty list* ⇒ *bool*
(‹-,- ⊢*jmm* - [:≤] -› [*51*,*51*,*51*,*51*] *50*)
where *jmm-confs P* ≡ *jmm.confs TYPE*('*m*) *P P*

abbreviation *jmm-tconf* :: *'m prog* ⇒ *JMM-heap* ⇒ *addr* ⇒ *bool* (‹-,- ⊢*jmm* - √*t*› [*51*,*51*,*51*] *50*)
where *jmm-tconf P* ≡ *jmm.tconf TYPE*('*m*) *P P*

interpretation *jmm*: *allocated-heap-base*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr P jmm-heap-read jmm-heap-write
jmm-allocated
for *P*
.

Now a variation of the JMM with a different read operation that permits to read only type-conformant values

abbreviation *jmm-heap-read-typed* :: *'m prog* ⇒ *JMM-heap* ⇒ *addr* ⇒ *addr-loc* ⇒ *addr val* ⇒ *bool*
where *jmm-heap-read-typed P* ≡ *jmm.heap-read-typed TYPE*('*m*) *P P*

interpretation *jmm'*: *heap-base*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr P jmm-heap-read-typed P jmm-heap-write
for *P* **.**

abbreviation *jmm'-hext* :: *'m prog* ⇒ *JMM-heap* ⇒ *JMM-heap* ⇒ *bool* (‹- ⊢ - ⊴*jmm''* -› [*51*,*51*,*51*] *50*)
where *jmm'-hext* ≡ *jmm'.hext TYPE*('*m*)

abbreviation *jmm'-conf* :: *'m prog* ⇒ *JMM-heap* ⇒ *addr val* ⇒ *ty* ⇒ *bool*
(‹-,- ⊢*jmm''* - :≤ -› [*51*,*51*,*51*,*51*] *50*)
where *jmm'-conf P* ≡ *jmm'.conf TYPE*('*m*) *P P*

abbreviation *jmm'-addr-loc-type* :: *'m prog* ⇒ *JMM-heap* ⇒ *addr* ⇒ *addr-loc* ⇒ *ty* ⇒ *bool*
(‹-,- ⊢*jmm''* -@- : -› [*50*, *50*, *50*, *50*, *50*] *51*)
where *jmm'-addr-loc-type P* ≡ *jmm'.addr-loc-type TYPE*('*m*) *P P*

abbreviation *jmm'-confs* :: *'m prog* ⇒ *JMM-heap* ⇒ *addr val list* ⇒ *ty list* ⇒ *bool*
(‹-,- ⊢*jmm''* - [:≤] -› [*51*,*51*,*51*,*51*] *50*)
where *jmm'-confs P* ≡ *jmm'.confs TYPE*('*m*) *P P*

abbreviation *jmm'-tconf* :: *'m prog* ⇒ *JMM-heap* ⇒ *addr* ⇒ *bool* (‹-,- ⊢*jmm''* - √*t*› [*51*,*51*,*51*] *50*)
where *jmm'-tconf P* ≡ *jmm'.tconf TYPE('m) P P*

8.18.2 Heap locale interpretations

8.18.3 Locale *heap*

lemma *jmm-heap*: *heap addr2thread-id thread-id2addr jmm-allocate* (*jmm-typeof-addr P*) *jmm-heap-write P*
proof
  fix *h' a h hT*
  assume (*h'*, *a*) ∈ *jmm-allocate h hT is-htype P hT*
  thus *jmm-typeof-addr P h' a* = ⌊*hT*⌋
    by(*auto simp add*: *jmm-heap-ops-defs*)
next
  fix *h hT h' a*
  assume (*h'*, *a*) ∈ *jmm-allocate h hT*
  thus *P* ⊢ *h* ⊴*jmm h'* **by**(*auto simp add*: *jmm-heap-ops-defs intro*: *jmm.hextI*)
next
  fix *h a al v h'*
  assume *jmm-heap-write h a al v h'*
  thus *P* ⊢ *h* ⊴*jmm h'* **by** *cases auto*
qed *simp*

interpretation *jmm*: *heap*
  addr2thread-id thread-id2addr
  jmm-spurious-wakeups
  jmm-empty jmm-allocate jmm-typeof-addr P jmm-heap-read jmm-heap-write
  P
  for *P*
by(*rule jmm-heap*)

declare *jmm.typeof-addr-thread-id2-addr-addr2thread-id* [*simp del*]

lemmas *jmm'-heap* = *jmm-heap*

interpretation *jmm'*: *heap*
  addr2thread-id thread-id2addr
  jmm-spurious-wakeups
  jmm-empty jmm-allocate jmm-typeof-addr P jmm-heap-read-typed P jmm-heap-write
  P
  for *P*
by(*rule jmm'-heap*)

declare *jmm'.typeof-addr-thread-id2-addr-addr2thread-id* [*simp del*]

lemma *jmm-heap-read-typed-default-val*:
  heap-base.heap-read-typed typeof-addr jmm-heap-read P h a al
   (*default-val* (*THE T*. *heap-base.addr-loc-type typeof-addr P h a al T*))
by(*rule heap-base.heap-read-typedI*)(*simp-all add*: *heap-base.THE-addr-loc-type jmm-heap-read-def heap-base.defval-conf*)

lemma *jmm-allocate-Eps*:
  (*SOME ha*. *ha* ∈ *jmm-allocate h hT*) = (*h'*, *a'*)

$\Longrightarrow$ *jmm-allocate h hT* $\neq \{\} \longrightarrow (h', a') \in$ *jmm-allocate h hT*
by(*auto dest*: *jmm.allocate-Eps*)

lemma *jmm-allocate-eq-empty*: *jmm-allocate h hT* $= \{\} \longleftrightarrow$ *h hT* $=$ *UNIV*
by(*auto simp add*: *jmm-allocate-def*)

lemma *jmm-allocate-otherD*:
$(h', a) \in$ *jmm-allocate h hT* $\Longrightarrow \forall\, hT'.\ hT' \neq hT \longrightarrow h'\ hT' = h\ hT'$
by(*auto simp add*: *jmm-allocate-def*)

lemma *jmm-start-heap-ok*: *jmm.start-heap-ok*
apply(*simp add*: *jmm.start-heap-ok-def jmm.start-heap-data-def initialization-list-def sys-xcpts-list-def jmm.create-initial-object-simps*)
apply(*split prod.split*, *clarify*, *clarsimp simp add*: *jmm.create-initial-object-simps jmm-allocate-eq-empty Thread-neq-sys-xcpts sys-xcpts-neqs dest*!: *jmm-allocate-Eps jmm-allocate-otherD*)+
done

8.18.4 Locale *heap-conf*

interpretation *jmm*: *heap-conf-base*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr P jmm-heap-read jmm-heap-write jmm-hconf
P
for *P* **.**

abbreviation (*input*) *jmm'-hconf* :: *JMM-heap* $\Rightarrow$ *bool*
where *jmm'-hconf* $==$ *jmm-hconf*

interpretation *jmm'*: *heap-conf-base*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr P jmm-heap-read-typed P jmm-heap-write jmm'-hconf
P
for *P* **.**

abbreviation *jmm-heap-read-typeable* :: *'m prog* $\Rightarrow$ *bool*
where *jmm-heap-read-typeable P* $\equiv$ *jmm.heap-read-typeable TYPE*(*'m*) *P jmm-hconf P*

abbreviation *jmm'-heap-read-typeable* :: *'m prog* $\Rightarrow$ *bool*
where *jmm'-heap-read-typeable P* $\equiv$ *jmm'.heap-read-typeable TYPE*(*'m*) *P jmm-hconf P*

lemma *jmm-heap-read-typeable*: *jmm-heap-read-typeable P*
by(*rule jmm.heap-read-typeableI*)(*simp add*: *jmm-heap-read-def*)

lemma *jmm'-heap-read-typeable*: *jmm'-heap-read-typeable P*
by(*rule jmm'.heap-read-typeableI*)(*auto simp add*: *jmm-heap-read-def jmm.heap-read-typed-def dest*: *jmm'.addr-loc-type-fun*)

lemma *jmm-heap-conf*:
heap-conf addr2thread-id thread-id2addr jmm-empty jmm-allocate (*jmm-typeof-addr P*) *jmm-heap-write jmm-hconf P*
by(*unfold-locales*)(*simp-all add*: *jmm-hconf-def jmm-heap-ops-defs split*: *if-split-asm*)

interpretation *jmm*: *heap-conf*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr P jmm-heap-read jmm-heap-write jmm-hconf
P
for *P*
by(*rule jmm-heap-conf*)

lemmas *jmm′-heap-conf* = *jmm-heap-conf*

interpretation *jmm′*: *heap-conf*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr P jmm-heap-read-typed P jmm-heap-write jmm′-hconf
P
for *P*
by(*rule jmm′-heap-conf*)

8.18.5 Locale *heap-progress*

lemma *jmm-heap-progress*:
heap-progress addr2thread-id thread-id2addr jmm-empty jmm-allocate (*jmm-typeof-addr P*) *jmm-heap-read jmm-heap-write jmm-hconf P*
proof
fix *h a al T*
assume *jmm-hconf h*
and *al*: *P*,*h* ⊢*jmm a*@*al* : *T*
show ∃ *v*. *jmm-heap-read h a al v* ∧ *P*,*h* ⊢*jmm v* :≤ *T*
using *jmm.defval-conf*[*of P P h T*] **unfolding** *jmm-heap-ops-defs* **by** *blast*
next
fix *h a al T v*
assume *P*,*h* ⊢*jmm a*@*al* : *T*
show ∃ *h′*. *jmm-heap-write h a al v h′*
by(*auto intro*: *jmm-heap-write.intros*)
qed

interpretation *jmm*: *heap-progress*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr P jmm-heap-read jmm-heap-write jmm-hconf
P
for *P*
by(*rule jmm-heap-progress*)

lemma *jmm′-heap-progress*:
heap-progress addr2thread-id thread-id2addr jmm-empty jmm-allocate (*jmm-typeof-addr P*) (*jmm-heap-read-typed P*) *jmm-heap-write jmm′-hconf P*
proof
fix *h a al T*
assume *jmm′-hconf h*
and *al*: *P*,*h* ⊢*jmm′ a*@*al* : *T*
thus ∃ *v*. *jmm-heap-read-typed P h a al v* ∧ *P*,*h* ⊢*jmm′ v* :≤ *T*
unfolding *jmm-heap-read-def jmm.heap-read-typed-def*
by(*blast dest*: *jmm′.addr-loc-type-fun intro*: *jmm′.defval-conf*)+

next
  fix *h a al T v*
  assume *P,h ⊢jmm′ a@al : T*
    and *P,h ⊢jmm′ v :≤ T*
  thus ∃ *h′. jmm-heap-write h a al v h′*
    by(*auto intro*: *jmm-heap-write.intros*)
qed

interpretation *jmm′*: *heap-progress*
  addr2thread-id thread-id2addr
  jmm-spurious-wakeups
  jmm-empty jmm-allocate jmm-typeof-addr P jmm-heap-read-typed P jmm-heap-write jmm′-hconf
  P
  for *P*
by(*rule jmm′-heap-progress*)

8.18.6 Locale *heap-conf-read*

lemma *jmm′-heap-conf-read*:
  heap-conf-read addr2thread-id thread-id2addr jmm-empty jmm-allocate (*jmm-typeof-addr P*) (*jmm-heap-read-typed P*) *jmm-heap-write jmm′-hconf P*
by(*rule jmm.heap-conf-read-heap-read-typed*)

interpretation *jmm′*: *heap-conf-read*
  addr2thread-id thread-id2addr
  jmm-spurious-wakeups
  jmm-empty jmm-allocate jmm-typeof-addr P jmm-heap-read-typed P jmm-heap-write jmm′-hconf
  P
  for *P*
by(*rule jmm′-heap-conf-read*)

interpretation *jmm′*: *heap-typesafe*
  addr2thread-id thread-id2addr
  jmm-spurious-wakeups
  jmm-empty jmm-allocate jmm-typeof-addr P jmm-heap-read-typed P jmm-heap-write jmm′-hconf
  P
  for *P*
..

8.18.7 Locale *allocated-heap*

lemma *jmm-allocated-heap*:
  allocated-heap addr2thread-id thread-id2addr jmm-empty jmm-allocate (*jmm-typeof-addr P*) *jmm-heap-write jmm-allocated P*
proof
  show *jmm-allocated jmm-empty* = {} **by**(*simp add*: *jmm-allocated-def*)
next
  fix *h′ a h hT*
  assume (*h′*, *a*) ∈ *jmm-allocate h hT*
  thus *jmm-allocated h′* = *insert a* (*jmm-allocated h*) ∧ *a* ∉ *jmm-allocated h*
    by(*auto simp add*: *jmm-heap-ops-defs split*: *if-split-asm*)
next
  fix *h a al v h′*
  assume *jmm-heap-write h a al v h′*

thus *jmm-allocated h′ = jmm-allocated h* **by** *cases simp*
qed

interpretation *jmm*: *allocated-heap*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr P jmm-heap-read jmm-heap-write
jmm-allocated
P
for *P*
by(*rule jmm-allocated-heap*)

lemmas *jmm′-allocated-heap = jmm-allocated-heap*

interpretation *jmm′*: *allocated-heap*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr P jmm-heap-read-typed P jmm-heap-write
jmm-allocated
P
for *P*
by(*rule jmm′-allocated-heap*)

8.18.8 Syntax translations

notation *jmm′.external-WT′* (‹-,- ⊢*jmm″* (-▪-′(-′)) : -› [*50*,*0*,*0*,*0*,*50*] *60*)

abbreviation *jmm′-red-external* ::
′m prog ⇒ *thread-id* ⇒ *JMM-heap* ⇒ *addr* ⇒ *mname* ⇒ *addr val list*
⇒ (*addr*, *thread-id*, *JMM-heap*) *external-thread-action*
⇒ *addr extCallRet* ⇒ *JMM-heap* ⇒ *bool*
where *jmm′-red-external P* ≡ *jmm′.red-external* (*TYPE*(*′m*)) *P P*

abbreviation *jmm′-red-external-syntax* ::
′m prog ⇒ *thread-id* ⇒ *addr* ⇒ *mname* ⇒ *addr val list* ⇒ *JMM-heap*
⇒ (*addr*, *thread-id*, *JMM-heap*) *external-thread-action*
⇒ *addr extCallRet* ⇒ *JMM-heap* ⇒ *bool*
(‹-,- ⊢*jmm″* (⟨(-▪-′(-′)),/-⟩) −-→*ext* (⟨(-),/(-)⟩)› [*50*, *0*, *0*, *0*, *0*, *0*, *0*, *0*, *0*] *51*)
where
P,*t* ⊢*jmm′* ⟨*a*▪*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h′*⟩ ≡ *jmm′-red-external P t h a M vs ta va h′*

abbreviation *jmm′-red-external-aggr* ::
′m prog ⇒ *thread-id* ⇒ *addr* ⇒ *mname* ⇒ *addr val list* ⇒ *JMM-heap*
⇒ ((*addr*, *thread-id*, *JMM-heap*) *external-thread-action* × *addr extCallRet* × *JMM-heap*) *set*
where *jmm′-red-external-aggr P* ≡ *jmm′.red-external-aggr TYPE*(*′m*) *P P*

abbreviation *jmm′-heap-copy-loc* ::
′m prog ⇒ *addr* ⇒ *addr* ⇒ *addr-loc* ⇒ *JMM-heap*
⇒ (*addr*, *thread-id*) *obs-event list* ⇒ *JMM-heap* ⇒ *bool*
where *jmm′-heap-copy-loc* ≡ *jmm′.heap-copy-loc TYPE*(*′m*)

abbreviation *jmm′-heap-copies* ::
′m prog ⇒ *addr* ⇒ *addr* ⇒ *addr-loc list* ⇒ *JMM-heap*
⇒ (*addr*, *thread-id*) *obs-event list* ⇒ *JMM-heap* ⇒ *bool*

where *jmm'-heap-copies* ≡ *jmm'.heap-copies TYPE('m)*

abbreviation *jmm'-heap-clone* ::
'm prog ⇒ *JMM-heap* ⇒ *addr* ⇒ *JMM-heap*
⇒ ((*addr*, *thread-id*) *obs-event list* × *addr*) *option* ⇒ *bool*
where *jmm'-heap-clone P* ≡ *jmm'.heap-clone TYPE('m) P P*

end

theory *JMM-Interp* **imports**
JMM-Compiler
../J/Deadlocked
../BV/JVMDeadlocked
JMM-Type2
DRF-J
DRF-JVM
begin

lemma *jmm'-J-typesafe*:
J-typesafe addr2thread-id thread-id2addr jmm-empty jmm-allocate (*jmm-typeof-addr P*) (*jmm-heap-read-typed P*) *jmm-heap-write jmm-hconf P*
by *unfold-locales*

lemma *jmm'-JVM-typesafe*:
JVM-typesafe addr2thread-id thread-id2addr jmm-empty jmm-allocate (*jmm-typeof-addr P*) (*jmm-heap-read-typed P*) *jmm-heap-write jmm-hconf P*
by *unfold-locales*

lemma *jmm-typeof-addr-compP* [*simp*]:
jmm-typeof-addr (*compP f P*) = *jmm-typeof-addr P*
by(*simp add*: *jmm-typeof-addr-def fun-eq-iff*)

lemma *compP2-compP1-convs*:
is-type (*compP2* (*compP1 P*)) = *is-type P*
is-class (*compP2* (*compP1 P*)) = *is-class P*
jmm'-addr-loc-type (*compP2* (*compP1 P*)) = *jmm'-addr-loc-type P*
jmm'-conf (*compP2* (*compP1 P*)) = *jmm'-conf P*
by(*simp-all add*: *compP2-def heap-base.compP-conf heap-base.compP-addr-loc-type fun-eq-iff split*: *addr-loc.splits*)

lemma *jmm'-J-JVM-conf-read*:
J-JVM-conf-read addr2thread-id thread-id2addr jmm-empty jmm-allocate (*jmm-typeof-addr P*) (*jmm-heap-read-typ P*) *jmm-heap-write jmm-hconf P*
apply(*rule J-JVM-conf-read.intro*)
apply(*rule J1-JVM-conf-read.intro*)
apply(*rule JVM-conf-read.intro*)
prefer *2*
apply(*rule JVM-heap-conf.intro*)
apply(*rule JVM-heap-conf-base'.intro*)
apply(*unfold compP2-def compP1-def compP-heap compP-heap-conf compP-heap-conf-read jmm-typeof-addr-comp*
apply *unfold-locales*
done

lemma *jmm-J-allocated-progress*:

J-allocated-progress addr2thread-id thread-id2addr jmm-empty jmm-allocate (jmm-typeof-addr P) jmm-heap-read jmm-heap-write jmm-hconf jmm-allocated P
by *unfold-locales*

lemma *jmm'-J-allocated-progress*:
J-allocated-progress addr2thread-id thread-id2addr jmm-empty jmm-allocate (jmm-typeof-addr P) (jmm-heap-read-typed P) jmm-heap-write jmm-hconf jmm-allocated P
by(*unfold-locales*)

lemma *jmm-JVM-allocated-progress*:
JVM-allocated-progress addr2thread-id thread-id2addr jmm-empty jmm-allocate (jmm-typeof-addr P) jmm-heap-read jmm-heap-write jmm-hconf jmm-allocated P
by *unfold-locales*

lemma *jmm'-JVM-allocated-progress*:
JVM-allocated-progress addr2thread-id thread-id2addr jmm-empty jmm-allocate (jmm-typeof-addr P) (jmm-heap-read-typed P) jmm-heap-write jmm-hconf jmm-allocated P
by(*unfold-locales*)

end

8.19 Specialize type safety for JMM heap implementation 2

theory *JMM-Typesafe2*
imports
JMM-Type2
JMM-Common
begin

interpretation *jmm*: *heap'*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr' P jmm-heap-read jmm-heap-write
for *P*
by(*rule heap'.intro*)(*unfold jmm-typeof-addr'-conv-jmm-typeof-addr*, *unfold-locales*)

abbreviation *jmm-addr-loc-type'* :: *'m prog* ⇒ *addr* ⇒ *addr-loc* ⇒ *ty* ⇒ *bool* (‹- ⊢*jmm* -@- : -› [*50*, *50*, *50*, *50*] *51*)
where *jmm-addr-loc-type' P* ≡ *jmm.addr-loc-type TYPE('m) P P*

lemma *jmm-addr-loc-type-conv-jmm-addr-loc-type'* [*simp*, *heap-independent*]:
jmm-addr-loc-type P h = *jmm-addr-loc-type' P*
by(*metis jmm-typeof-addr'-conv-jmm-typeof-addr heap-base'.addr-loc-type-conv-addr-loc-type*)

abbreviation *jmm-conf'* :: *'m prog* ⇒ *addr val* ⇒ *ty* ⇒ *bool* (‹- ⊢*jmm* - :≤ -› [*51*,*51*,*51*] *50*)
where *jmm-conf' P* ≡ *jmm.conf TYPE('m) P P*

lemma *jmm-conf-conv-jmm-conf'* [*simp*, *heap-independent*]:
jmm-conf P h = *jmm-conf' P*
by (*metis jmm-typeof-addr'-conv-jmm-typeof-addr heap-base'.conf-conv-conf*)

lemma *jmm-heap''*: *heap'' addr2thread-id thread-id2addr jmm-allocate* (*jmm-typeof-addr' P*) *jmm-heap-write P*

by(*unfold-locales*)(*auto simp add*: *jmm-typeof-addr'-def jmm-allocate-def split*: *if-split-asm*)

interpretation *jmm*: *heap''*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr' P jmm-heap-read jmm-heap-write
for *P*
by(*rule jmm-heap''*)

interpretation *jmm'*: *heap''*
addr2thread-id thread-id2addr
jmm-spurious-wakeups
jmm-empty jmm-allocate jmm-typeof-addr' P jmm-heap-read-typed P jmm-heap-write
for *P*
by(*rule jmm-heap''*)

abbreviation *jmm-wf-start-state* :: *'m prog ⇒ cname ⇒ mname ⇒ addr val list ⇒ bool*
where *jmm-wf-start-state P ≡ jmm.wf-start-state TYPE('m) P P*

abbreviation *if-heap-read-typed* ::
('x ⇒ bool) ⇒ ('l, 't, 'x, 'heap, 'w, ('addr :: addr, 'thread-id) obs-event) semantics
⇒ ('addr ⇒ htype option)
⇒ 'm prog ⇒ ('l, 't, status × 'x, 'heap, 'w, ('addr, 'thread-id) obs-event action) semantics
where
⋀final. if-heap-read-typed final r typeof-addr P t xh ta x'h' ≡
multithreaded-base.init-fin final r t xh ta x'h' ∧
(∀ ad al v T. NormalAction (ReadMem ad al v) ∈ set {|ta|}$_o$ ⟶ heap-base'.addr-loc-type TYPE('heap) typeof-addr P ad al T ⟶ heap-base'.conf TYPE('heap) typeof-addr P v T)

lemma *if-mthr-Runs-heap-read-typedI*:
fixes *final* **and** *r* :: *('addr, 't, 'x, 'heap, 'w, ('addr :: addr, 'thread-id) obs-event) semantics*
assumes *trsys.Runs (multithreaded-base.redT (final-thread.init-fin-final final) (multithreaded-base.init-fin final r) (map NormalAction ∘ convert-RA)) s ξ*
(**is** *trsys.Runs ?redT - -*)
and *⋀ad al v T. ⟦ NormalAction (ReadMem ad al v) ∈ lset (lconcat (lmap (llist-of ∘ obs-a ∘ snd) ξ)); heap-base'.addr-loc-type TYPE('heap) typeof-addr P ad al T ⟧ ⟹ heap-base'.conf TYPE('heap) typeof-addr P v T*
(**is** *⋀ad al v T. ⟦ ?obs ξ ad al v; ?adal ad al T ⟧ ⟹ ?conf v T*)
shows *trsys.Runs (multithreaded-base.redT (final-thread.init-fin-final final) (if-heap-read-typed final r typeof-addr P) (map NormalAction ∘ convert-RA)) s ξ*
(**is** *trsys.Runs ?redT' - -*)
using *assms*
proof(*coinduction arbitrary*: *s ξ rule*: *trsys.Runs.coinduct*[*consumes 1*, *case-names Runs*, *case-conclusion Runs Stuck Step*])
case (*Runs s ξ*)
let *?read = λξ. (∀ ad al v T. ?obs ξ ad al v ⟶ ?adal ad al T ⟶ ?conf v T)*
note *read = Runs(2)*
from *Runs(1)* **show** *?case*
proof(*cases rule*: *trsys.Runs.cases*[*consumes 1*, *case-names Stuck Step*])
case (*Stuck S*)
{ fix *tta s'*
from *‹¬ ?redT S tta s'›* **have** *¬ ?redT' S tta s'*
by(*rule contrapos-nn*)(*fastforce simp add*: *multithreaded-base.redT.simps*) **}**

```
    hence ?Stuck using ‹ξ = LNil› unfolding ‹s = S› by blast
    thus ?thesis ..
  next
    case (Step S s' ttas tta)
    from ‹ξ = LCons tta ttas› read
    have read1: ⋀ad al v T. ⟦ NormalAction (ReadMem ad al v) ∈ set {|snd tta|}_o; ?adal ad al T ⟧
⟹ ?conf v T
      and read2: ?read ttas by(auto simp add: o-def)
    from ‹?redT S tta s'› read1
    have ?redT' S tta s' by(fastforce simp add: multithreaded-base.redT.simps)
    hence ?Step using Step read2 ‹s = S› by blast
    thus ?thesis ..
  qed
qed

lemma if-mthr-Runs-heap-read-typedD:
  fixes final and r :: ('addr, 't, 'x, 'heap, 'w, ('addr :: addr, 'thread-id) obs-event) semantics
  assumes Runs': trsys.Runs (multithreaded-base.redT (final-thread.init-fin-final final) (if-heap-read-typed
final r typeof-addr P) (map NormalAction ∘ convert-RA)) s ξ
  (is ?Runs' s ξ)
  and stuck: ⋀ttas s' tta s''. ⟦
    multithreaded-base.RedT (final-thread.init-fin-final final) (if-heap-read-typed final r typeof-addr P)
(map NormalAction ∘ convert-RA) s ttas s';
    multithreaded-base.redT (final-thread.init-fin-final final) (multithreaded-base.init-fin final r) (map
NormalAction ∘ convert-RA) s' tta s'' ⟧
  ⟹ ∃tta s''. multithreaded-base.redT (final-thread.init-fin-final final) (if-heap-read-typed final r typeof-addr
P) (map NormalAction ∘ convert-RA) s' tta s''
  (is ⋀ttas s' tta s''. ⟦ ?RedT' s ttas s'; ?redT s' tta s'' ⟧ ⟹ ∃tta s''. ?redT' s' tta s'')
 shows trsys.Runs (multithreaded-base.redT (final-thread.init-fin-final final) (multithreaded-base.init-fin
final r) (map NormalAction ∘ convert-RA)) s ξ
  (is ?Runs s ξ)
proof −
  define s' where s' = s
  with Runs' have ∃ttas. ?RedT' s ttas s' ∧ ?Runs' s' ξ
    by(auto simp add: multithreaded-base.RedT-def o-def)
  thus ?Runs s' ξ
  proof(coinduct rule: trsys.Runs.coinduct[consumes 1, case-names Runs, case-conclusion Runs Stuck
Step])
    case (Runs s' ξ)
    then obtain ttas where RedT': ?RedT' s ttas s'
      and Runs': ?Runs' s' ξ by blast
    from Runs' show ?case
    proof(cases rule: trsys.Runs.cases[consumes 1, case-names Stuck Step])
      case (Stuck S)
      have ⋀tta s''. ¬ ?redT s' tta s''
      proof
        fix tta s''
        assume ?redT s' tta s''
        from stuck[OF RedT' this]
        obtain tta s'' where ?redT' s' tta s'' by blast
        with Stuck(3)[of tta s''] show False
          unfolding ‹s' = S› by contradiction
      qed
      with Stuck(1−2) have ?Stuck by simp
```

thus *?thesis* **by**(*rule disjI1*)
next
case (*Step S s'' ξ' tta*)
note *Step* = *Step*(*2*−)[*folded* ‹*s'* = *S*›]
from ‹*?redT' s' tta s''*› **have** *?redT s' tta s''*
by(*fastforce simp add*: *multithreaded-base.redT.simps*)
moreover from *RedT'* ‹*?redT' s' tta s''*›
have *?RedT' s* (*ttas* @ [*tta*]) *s''*
unfolding *multithreaded-base.RedT-def* **by**(*rule rtrancl3p-step*)
ultimately have *?Step* **using** ‹*ξ* = *LCons tta ξ'*› ‹*?Runs' s'' ξ'*› **by** *blast*
thus *?thesis* **by**(*rule disjI2*)
qed
qed
qed

lemma *heap-copy-loc-heap-read-typed*:
heap-base.heap-copy-loc (*heap-base.heap-read-typed* (λ- :: *'heap. typeof-addr*) *heap-read P*) *heap-write a a' al h obs h'* ⟷
heap-base.heap-copy-loc heap-read heap-write a a' al h obs h' ∧
(∀ *ad al v T*. *ReadMem ad al v* ∈ *set obs* ⟶ *heap-base'.addr-loc-type TYPE*(*'heap*) *typeof-addr P ad al T* ⟶ *heap-base'.conf TYPE*(*'heap*) *typeof-addr P v T*)
by(*auto elim*!: *heap-base.heap-copy-loc.cases intro*!: *heap-base.heap-copy-loc.intros dest*: *heap-base.heap-read-typed-i heap-base.heap-read-typed-typed intro*: *heap-base.heap-read-typedI simp add*: *heap-base'.addr-loc-type-conv-addr-loc-t heap-base'.conf-conv-conf*)

lemma *heap-copies-heap-read-typed*:
heap-base.heap-copies (*heap-base.heap-read-typed* (λ- :: *'heap. typeof-addr*) *heap-read P*) *heap-write a a' als h obs h'* ⟷
heap-base.heap-copies heap-read heap-write a a' als h obs h' ∧
(∀ *ad al v T*. *ReadMem ad al v* ∈ *set obs* ⟶ *heap-base'.addr-loc-type TYPE*(*'heap*) *typeof-addr P ad al T* ⟶ *heap-base'.conf TYPE*(*'heap*) *typeof-addr P v T*)
(**is** *?lhs* ⟷ *?rhs*)
proof
assume *?lhs* **thus** *?rhs*
by(*induct rule*: *heap-base.heap-copies.induct*[*consumes 1*])(*auto intro*!: *heap-base.heap-copies.intros simp add*: *heap-copy-loc-heap-read-typed*)
next
assume *?rhs* **thus** *?lhs*
by(*rule conjE*)(*induct rule*: *heap-base.heap-copies.induct*[*consumes 1*], *auto intro*!: *heap-base.heap-copies.intros simp add*: *heap-copy-loc-heap-read-typed*)
qed

lemma *heap-clone-heap-read-typed*:
heap-base.heap-clone allocate (λ- :: *'heap. typeof-addr*) (*heap-base.heap-read-typed* (λ- :: *'heap. typeof-addr*) *heap-read P*) *heap-write P a h h' obs* ⟷
heap-base.heap-clone allocate (λ- :: *'heap. typeof-addr*) *heap-read heap-write P a h h' obs* ∧
(∀ *ad al v T obs' a'*. *obs* = ⌊(*obs'*, *a'*)⌋ ⟶ *ReadMem ad al v* ∈ *set obs'* ⟶ *heap-base'.addr-loc-type TYPE*(*'heap*) *typeof-addr P ad al T* ⟶ *heap-base'.conf TYPE*(*'heap*) *typeof-addr P v T*)
by(*auto elim*!: *heap-base.heap-clone.cases intro*: *heap-base.heap-clone.intros simp add*: *heap-copies-heap-read-typed*)

lemma *red-external-heap-read-typed*:
heap-base.red-external addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate (λ- :: *'heap. typeof-addr*) (*heap-base.heap-read-typed* (λ- :: *'heap. typeof-addr*) *heap-read P*) *heap-write P t h a M vs ta va h'* ⟷

heap-base.red-external addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate (λ- :: *'heap. typeof-addr*) *heap-read heap-write P t h a M vs ta va h'* ∧
(∀ *ad al v T obs' a'. ReadMem ad al v* ∈ *set* ⦃*ta*⦄$_o$ ⟶ *heap-base'.addr-loc-type TYPE*(*'heap*) *typeof-addr P ad al T* ⟶ *heap-base'.conf TYPE*(*'heap*) *typeof-addr P v T*)
by(*auto elim*!: *heap-base.red-external.cases intro*: *heap-base.red-external.intros simp add*: *heap-clone-heap-read-typed*)

lemma *red-external-aggr-heap-read-typed*:
(*ta*, *va*, *h'*) ∈ *heap-base.red-external-aggr addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate* (λ- :: *'heap. typeof-addr*) (*heap-base.heap-read-typed* (λ- :: *'heap. typeof-addr*) *heap-read P*) *heap-write P t h a M vs* ⟷
(*ta*, *va*, *h'*) ∈ *heap-base.red-external-aggr addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate* (λ- :: *'heap. typeof-addr*) *heap-read heap-write P t h a M vs* ∧
(∀ *ad al v T obs' a'. ReadMem ad al v* ∈ *set* ⦃*ta*⦄$_o$ ⟶ *heap-base'.addr-loc-type TYPE*(*'heap*) *typeof-addr P ad al T* ⟶ *heap-base'.conf TYPE*(*'heap*) *typeof-addr P v T*)
by(*auto simp add*: *heap-base.red-external-aggr-def heap-clone-heap-read-typed split del*: *if-split split*: *if-split-asm*)

lemma *jmm'-heap-copy-locI*:
∃ *obs h'. heap-base.heap-copy-loc* (*heap-base.heap-read-typed typeof-addr jmm-heap-read P*) *jmm-heap-write a a' al h obs h'*
by(*auto intro*!: *heap-base.heap-copy-loc.intros jmm-heap-read-typed-default-val intro*: *jmm-heap-write.intros*)

lemma *jmm'-heap-copiesI*:
∃ *obs* :: (*addr*, *'thread-id*) *obs-event list*.
∃ *h'. heap-base.heap-copies* (*heap-base.heap-read-typed typeof-addr jmm-heap-read P*) *jmm-heap-write a a' als h obs h'*
proof(*induction als arbitrary*: *h*)
case *Nil*
thus *?case* **by**(*blast intro*: *heap-base.heap-copies.intros*)
next
case (*Cons al als*)
from *jmm'-heap-copy-locI*[*of typeof-addr P a a' al h*]
obtain *ob* :: (*addr*, *'thread-id*) *obs-event list* **and** *h'*
where *heap-base.heap-copy-loc* (*heap-base.heap-read-typed typeof-addr jmm-heap-read P*) *jmm-heap-write a a' al h ob h'*
by *blast*
with *Cons.IH*[*of h'*] **show** *?case*
by(*auto 4 4 intro*: *heap-base.heap-copies.intros*)
qed

lemma *jmm'-heap-cloneI*:
fixes *obsa* :: ((*addr*, *'thread-id*) *obs-event list* × *addr*) *option*
assumes *heap-base.heap-clone allocate typeof-addr jmm-heap-read jmm-heap-write P h a h' obsa*
shows ∃ *h'*. ∃ *obsa* :: ((*addr*, *'thread-id*) *obs-event list* × *addr*) *option*.
heap-base.heap-clone allocate typeof-addr (*heap-base.heap-read-typed typeof-addr jmm-heap-read P*) *jmm-heap-write P h a h' obsa*
using *assms*
proof(*cases rule*: *heap-base.heap-clone.cases*[*consumes 1*, *case-names Fail Obj Arr*])
case *Fail*
thus *?thesis* **by**(*blast intro*: *heap-base.heap-clone.intros*)
next
case (*Obj C h' a' FDTs obs h''*)

with *jmm'-heap-copiesI*[*of typeof-addr P a a' map* (λ((*F*, *D*), *Tfm*). *CField D F*) *FDTs h'*]
show *?thesis* **by**(*blast intro*: *heap-base.heap-clone.intros*)
next
case (*Arr T n h' a' FDTs obs h''*)
with *jmm'-heap-copiesI*[*of typeof-addr P a a' map* (λ((*F*, *D*), *Tfm*). *CField D F*) *FDTs* @ *map ACell* [*0*..<*n*]]
show *?thesis* **by**(*blast intro*: *heap-base.heap-clone.intros*)
qed

lemma *jmm'-red-externalI*:
⋀*final*.
⟦ *heap-base.red-external addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr jmm-heap-read jmm-heap-write P t h a M vs ta va h'*;
final-thread.actions-ok final s t ta ⟧
⟹ ∃ *ta va h'*. *heap-base.red-external addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr* (*heap-base.heap-read-typed typeof-addr jmm-heap-read P*) *jmm-heap-write P t h a M vs ta va h'* ∧ *final-thread.actions-ok final s t ta*
proof(*erule heap-base.red-external.cases*, *goal-cases*)
case *19*
thus *?case* **apply** −
apply(*drule jmm'-heap-cloneI*, *clarify*)
apply(*rename-tac obsa'*, *case-tac obsa'*)
by(*auto 4 4 intro*: *heap-base.red-external.intros simp add*: *final-thread.actions-ok-iff simp del*: *split-paired-Ex*)
next
case *20*
thus *?case* **apply** −
apply(*drule jmm'-heap-cloneI*, *clarify*)
apply(*rename-tac obsa'*, *case-tac obsa'*)
by(*auto 4 4 intro*: *heap-base.red-external.intros simp add*: *final-thread.actions-ok-iff simp del*: *split-paired-Ex*)
qed(*blast intro*: *heap-base.red-external.intros*)+

lemma *red-external-aggr-heap-read-typedI*:
⋀*final*.
⟦ (*ta*, *vah'*) ∈ *heap-base.red-external-aggr addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr jmm-heap-read jmm-heap-write P t h a M vs*;
final-thread.actions-ok final s t ta
⟧
⟹ ∃ *ta vah'*. (*ta*, *vah'*) ∈ *heap-base.red-external-aggr addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr* (*heap-base.heap-read-typed typeof-addr jmm-heap-read P*) *jmm-heap-write P t h a M vs* ∧ *final-thread.actions-ok final s t ta*
apply(*simp add*: *heap-base.red-external-aggr-def split-beta split del*: *if-split split*: *if-split-asm del*: *split-paired-Ex*)
apply(*auto simp del*: *split-paired-Ex*)
apply(*drule jmm'-heap-cloneI*)
apply(*clarify*)
apply(*rename-tac obsa*, *case-tac obsa*)
apply(*force simp add*: *final-thread.actions-ok-iff del*: *disjCI intro*: *disjI1 disjI2 simp del*: *split-paired-Ex*)
apply(*force simp add*: *final-thread.actions-ok-iff del*: *disjCI intro*: *disjI1 disjI2 simp del*: *split-paired-Ex*)
apply(*drule jmm'-heap-cloneI*)
apply *clarify*
apply(*rename-tac obsa*, *case-tac obsa*)
apply(*force simp add*: *final-thread.actions-ok-iff del*: *disjCI intro*: *disjI1 disjI2 simp del*: *split-paired-Ex*)
apply(*force simp add*: *final-thread.actions-ok-iff del*: *disjCI intro*: *disjI1 disjI2 simp del*: *split-paired-Ex*)

done

end

8.20 JMM type safety for source code

theory *JMM-J-Typesafe* **imports**
JMM-Typesafe2
DRF-J
begin

locale *J-allocated-heap-conf'* =
h: *J-heap-conf*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate λ*-. typeof-addr heap-read heap-write hconf*
P
+
h: *J-allocated-heap*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate λ*-. typeof-addr heap-read heap-write*
allocated
P
+
heap''
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
P
for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *allocated* :: *'heap* ⇒ *'addr set*
and *P* :: *'addr J-prog*

sublocale *J-allocated-heap-conf'* < *h*: *J-allocated-heap-conf*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate λ*-. typeof-addr heap-read heap-write hconf allocated*
P
by(*unfold-locales*)

context *J-allocated-heap-conf'* **begin**

lemma *red-New-type-match*:
⟦ *h.red' P t e s ta e' s'*; *NewHeapElem ad CTn* ∈ *set* {|*ta*|}$_o$; *typeof-addr ad* ≠ *None* ⟧

$\Longrightarrow$ *typeof-addr ad* = $\lfloor CTn \rfloor$
and *reds-New-type-match*:
$[\![$ *h.reds′ P t es s ta es′ s′*; *NewHeapElem ad CTn* $\in$ *set* $\{\!|ta|\!\}_o$; *typeof-addr ad* $\neq$ *None* $]\!]$
$\Longrightarrow$ *typeof-addr ad* = $\lfloor CTn \rfloor$
by(*induct rule*: *h.red-reds.inducts*)(*auto dest*: *allocate-typeof-addr-SomeD red-external-New-type-match*)

lemma *mred-known-addrs-typing′*:
assumes *wf*: *wf-J-prog P*
and *ok*: *h.start-heap-ok*
shows *known-addrs-typing′ addr2thread-id thread-id2addr empty-heap allocate typeof-addr heap-write allocated h.J-known-addrs final-expr* (*h.mred P*) (λ*t x h*. ∃ *ET*. *h.sconf-type-ok ET t x h*) *P*
proof −
interpret *known-addrs-typing*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate λ-. *typeof-addr heap-read heap-write*
allocated h.J-known-addrs
final-expr h.mred P λ*t x h*. ∃ *ET*. *h.sconf-type-ok ET t x h*
P
using *assms* **by**(*rule h.mred-known-addrs-typing*)

show *?thesis* **by** *unfold-locales*(*auto dest*: *red-New-type-match*)
qed

lemma *J-legal-read-value-typeable*:
assumes *wf*: *wf-J-prog P*
and *wf-start*: *h.wf-start-state P C M vs*
and *legal*: *weakly-legal-execution P* (*h.J-*$\mathcal{E}$ *P C M vs status*) (*E*, *ws*)
and *a*: *enat a* < *llength E*
and *read*: *action-obs E a* = *NormalAction* (*ReadMem ad al v*)
shows ∃ *T*. *P* ⊢ *ad@al* : *T* ∧ *P* ⊢ *v* :≤ *T*
proof −
note *wf*
moreover from *wf-start* **have** *h.start-heap-ok* **by** *cases*
moreover from *wf wf-start*
have *ts-ok* (λ*t x h*. ∃ *ET*. *h.sconf-type-ok ET t x h*) (*thr* (*h.J-start-state P C M vs*)) *h.start-heap*
by(*rule h.J-start-state-sconf-type-ok*)
moreover from *wf* **have** *wf-syscls P* **by**(*rule wf-prog-wf-syscls*)
ultimately show *?thesis* **using** *legal a read*
by(*rule known-addrs-typing′.weakly-legal-read-value-typeable*[*OF mred-known-addrs-typing′*])
qed

end

8.20.1 Specific part for JMM implementation 2

abbreviation *jmm-J-*$\mathcal{E}$
:: *addr J-prog* ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒ *status* ⇒ (*addr* × (*addr*, *addr*) *obs-event action*) *llist set*
where
jmm-J-$\mathcal{E}$ *P* ≡
J-heap-base.J-$\mathcal{E}$ *addr2thread-id thread-id2addr jmm-spurious-wakeups jmm-empty jmm-allocate* (*jmm-typeof-addr P*) *jmm-heap-read jmm-heap-write P*

abbreviation *jmm′-J-ℰ*
:: *addr J-prog ⇒ cname ⇒ mname ⇒ addr val list ⇒ status ⇒ (addr × (addr, addr) obs-event action) llist set*
where
jmm′-J-ℰ P ≡
J-heap-base.J-ℰ addr2thread-id thread-id2addr jmm-spurious-wakeups jmm-empty jmm-allocate (jmm-typeof-addr P) (jmm-heap-read-typed P) jmm-heap-write P

lemma *jmm-J-heap-conf*:
J-heap-conf addr2thread-id thread-id2addr jmm-empty jmm-allocate (jmm-typeof-addr P) jmm-heap-write jmm-hconf P
by(*unfold-locales*)

lemma *jmm-J-allocated-heap-conf*: *J-allocated-heap-conf addr2thread-id thread-id2addr jmm-empty jmm-allocate (jmm-typeof-addr P) jmm-heap-write jmm-hconf jmm-allocated P*
by(*unfold-locales*)

lemma *jmm-J-allocated-heap-conf′*:
J-allocated-heap-conf′ addr2thread-id thread-id2addr jmm-empty jmm-allocate (jmm-typeof-addr′ P) jmm-heap-write jmm-hconf jmm-allocated P
apply(*rule J-allocated-heap-conf′.intro*)
apply(*unfold jmm-typeof-addr′-conv-jmm-typeof-addr*)
apply(*unfold-locales*)
done

lemma *red-heap-read-typedD*:
J-heap-base.red′ addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate (λ- :: ′heap. typeof-addr) (heap-base.heap-read-typed (λ- :: ′heap. typeof-addr) heap-read P) heap-write P t e s ta e′ s′ ⟷
J-heap-base.red′ addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate (λ- :: ′heap. typeof-addr) heap-read heap-write P t e s ta e′ s′ ∧
(∀ ad al v T. ReadMem ad al v ∈ set {|ta|}$_o$ ⟶ heap-base′.addr-loc-type TYPE(′heap) typeof-addr P ad al T ⟶ heap-base′.conf TYPE(′heap) typeof-addr P v T)
(**is** *?lhs1 ⟷ ?rhs1a ∧ ?rhs1b*)
and *reds-heap-read-typedD*:
J-heap-base.reds′ addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate (λ- :: ′heap. typeof-addr) (heap-base.heap-read-typed (λ- :: ′heap. typeof-addr) heap-read P) heap-write P t es s ta es′ s′ ⟷
J-heap-base.reds′ addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate (λ- :: ′heap. typeof-addr) heap-read heap-write P t es s ta es′ s′ ∧
(∀ ad al v T. ReadMem ad al v ∈ set {|ta|}$_o$ ⟶ heap-base′.addr-loc-type TYPE(′heap) typeof-addr P ad al T ⟶ heap-base′.conf TYPE(′heap) typeof-addr P v T)
(**is** *?lhs2 ⟷ ?rhs2a ∧ ?rhs2b*)
proof −
have (*?lhs1 ⟶ ?rhs1a ∧ ?rhs1b*) ∧ (*?lhs2 ⟶ ?rhs2a ∧ ?rhs2b*)
apply(*induct rule*: *J-heap-base.red-reds.induct*)
prefer *50*
apply(*subst* (*asm*) *red-external-heap-read-typed*)
apply(*fastforce intro*!: *J-heap-base.red-reds.RedCallExternal simp add*: *convert-extTA-def*)

prefer *49*

apply(*fastforce dest*: *J-heap-base.red-reds.RedCall*)

apply(*auto intro*: *J-heap-base.red-reds.intros dest*: *heap-base.heap-read-typed-into-heap-read heap-base.heap-read-*
dest: *heap-base′.addr-loc-type-conv-addr-loc-type*[*THEN fun-cong*, *THEN fun-cong*, *THEN fun-cong*,
THEN iffD2] *heap-base′.conf-conv-conf*[*THEN fun-cong*, *THEN fun-cong*, *THEN iffD1*])
done
moreover have (*?rhs1a* ⟶ *?rhs1b* ⟶ *?lhs1*) ∧ (*?rhs2a* ⟶ *?rhs2b* ⟶ *?lhs2*)
apply(*induct rule*: *J-heap-base.red-reds.induct*)
prefer *50*
apply *simp*
apply(*intro strip*)
apply(*erule* (*1*) *J-heap-base.red-reds.RedCallExternal*)
apply(*subst red-external-heap-read-typed*, *erule conjI*)
apply(*blast+*)[*4*]

prefer *49*
apply(*fastforce dest*: *J-heap-base.red-reds.RedCall*)

apply(*auto intro*: *J-heap-base.red-reds.intros intro*!: *heap-base.heap-read-typedI dest*: *heap-base′.addr-loc-type-con*
fun-cong, *THEN fun-cong*, *THEN fun-cong*, *THEN iffD1*] *intro*: *heap-base′.conf-conv-conf*[*THEN fun-cong*,
THEN fun-cong, *THEN iffD2*])
done
ultimately show *?lhs1* ⟷ *?rhs1a* ∧ *?rhs1b ?lhs2* ⟷ *?rhs2a* ∧ *?rhs2b* **by** *blast+*
qed

lemma *if-mred-heap-read-typedD*:
multithreaded-base.init-fin final-expr (*J-heap-base.mred addr2thread-id thread-id2addr spurious-wakeups*
empty-heap allocate (λ- :: *′heap. typeof-addr*) (*heap-base.heap-read-typed* (λ- :: *′heap. typeof-addr*)
heap-read P) *heap-write P*) *t xh ta x′h′* ⟷
if-heap-read-typed final-expr (*J-heap-base.mred addr2thread-id thread-id2addr spurious-wakeups empty-heap*
allocate (λ- :: *′heap. typeof-addr*) *heap-read heap-write P*) *typeof-addr P t xh ta x′h′*
unfolding *multithreaded-base.init-fin.simps*
by(*subst red-heap-read-typedD*) *fastforce*

lemma *J-ℰ-heap-read-typedI*:
⟦ *E* ∈ *J-heap-base.J-ℰ addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate* (λ- ::
′heap. typeof-addr) *heap-read heap-write P C M vs status*;
⋀*ad al v T*. ⟦ *NormalAction* (*ReadMem ad al v*) ∈ *snd* ‘ *lset E*; *heap-base′.addr-loc-type TYPE*(*′heap*)
typeof-addr P ad al T ⟧ ⟹ *heap-base′.conf TYPE*(*′heap*) *typeof-addr P v T* ⟧
⟹ *E* ∈ *J-heap-base.J-ℰ addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate* (λ- ::
′heap. typeof-addr) (*heap-base.heap-read-typed* (λ- :: *′heap. typeof-addr*) *heap-read P*) *heap-write P C*
M vs status
apply(*erule imageE*, *hypsubst*)
apply(*rule imageI*)
apply(*erule multithreaded-base.ℰ.cases*, *hypsubst*)
apply(*rule multithreaded-base.ℰ.intros*)
apply(*subst if-mred-heap-read-typedD*[*abs-def*])
apply(*erule if-mthr-Runs-heap-read-typedI*)
apply(*auto simp add*: *image-Un lset-lmap*[*symmetric*] *lmap-lconcat llist.map-comp o-def split-def simp*
del: *lset-lmap*)
done

lemma *jmm′-redI*:
⟦ *J-heap-base.red′ addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr*

```
jmm-heap-read jmm-heap-write P t e s ta e' s';
    final-thread.actions-ok (final-thread.init-fin-final final-expr) S t ta ⟧
  ⟹ ∃ta e' s'. J-heap-base.red' addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate
typeof-addr (heap-base.heap-read-typed typeof-addr jmm-heap-read P) jmm-heap-write P t e s ta e' s'
∧ final-thread.actions-ok (final-thread.init-fin-final final-expr) S t ta
  (is ⟦ ?red'; ?aok ⟧ ⟹ ?concl)
  and jmm'-redsI:
  ⟦ J-heap-base.reds' addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr
jmm-heap-read jmm-heap-write P t es s ta es' s';
    final-thread.actions-ok (final-thread.init-fin-final final-expr) S t ta ⟧
  ⟹ ∃ta es' s'. J-heap-base.reds' addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate
typeof-addr (heap-base.heap-read-typed typeof-addr jmm-heap-read P) jmm-heap-write P t es s ta es' s'
∧
    final-thread.actions-ok (final-thread.init-fin-final final-expr) S t ta
  (is ⟦ ?reds'; ?aoks ⟧ ⟹ ?concls)
proof −
  note [simp del] = split-paired-Ex
    and [simp add] = final-thread.actions-ok-iff heap-base.THE-addr-loc-type heap-base.defval-conf
    and [intro] = jmm-heap-read-typed-default-val

  let ?v = λh a al. default-val (THE T. heap-base.addr-loc-type typeof-addr P h a al T)

  have (?red' ⟶ ?aok ⟶ ?concl) ∧ (?reds' ⟶ ?aoks ⟶ ?concls)
  proof(induct rule: J-heap-base.red-reds.induct)
    case (23 h a T n i v l)
     thus ?case by(auto 4 6 intro: J-heap-base.red-reds.RedAAcc[where v=?v h a (ACell (nat (sint
i)))])
  next
    case (35 h a D F v l)
    thus ?case by(auto 4 5 intro: J-heap-base.red-reds.RedFAcc[where v=?v h a (CField D F)])
  next
    case RedCASSucceed: (45 h a D F v v' h')
    thus ?case
    proof(cases v = ?v h a (CField D F))
      case True
      with RedCASSucceed show ?thesis
        by(fastforce intro: J-heap-base.red-reds.RedCASSucceed[where v=?v h a (CField D F)])
    next
      case False
      with RedCASSucceed show ?thesis
        by(fastforce intro: J-heap-base.red-reds.RedCASFail[where v''=?v h a (CField D F)])
    qed
  next
    case RedCASFail: (46 h a D F v'' v v' l)
    thus ?case
    proof(cases v = ?v h a (CField D F))
      case True
      with RedCASFail show ?thesis
     by(fastforce intro: J-heap-base.red-reds.RedCASSucceed[where v=?v h a (CField D F)] jmm-heap-write.intros)
    next
      case False
      with RedCASFail show ?thesis
        by(fastforce intro: J-heap-base.red-reds.RedCASFail[where v''=?v h a (CField D F)])
    qed
```

next
case (*50 s a hU M Ts T D vs ta va h' ta' e' s'*)
thus *?case*
apply *clarify*
apply(*drule jmm'-red-externalI*, *simp*)
apply(*auto 4 4 intro*: *J-heap-base.red-reds.RedCallExternal*)
done
next
case (*52 e h l V vo ta e' h' l' T*)
thus *?case*
by(*clarify*)(*iprover intro*: *J-heap-base.red-reds.BlockRed*)
qed(*blast intro*: *J-heap-base.red-reds.intros*)+
thus ⟦ *?red'*; *?aok* ⟧ ⟹ *?concl* **and** ⟦ *?reds'*; *?aoks* ⟧ ⟹ *?concls* **by** *blast*+
qed

lemma *if-mred-heap-read-not-stuck*:
⟦ *multithreaded-base.init-fin final-expr* (*J-heap-base.mred addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr jmm-heap-read jmm-heap-write P*) *t xh ta x'h'*;
final-thread.actions-ok (*final-thread.init-fin-final final-expr*) *s t ta* ⟧
⟹
∃ *ta x'h'*. *multithreaded-base.init-fin final-expr* (*J-heap-base.mred addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr* (*heap-base.heap-read-typed typeof-addr jmm-heap-read P*) *jmm-heap-write P*) *t xh ta x'h'* ∧ *final-thread.actions-ok* (*final-thread.init-fin-final final-expr*) *s t ta*
apply(*erule multithreaded-base.init-fin.cases*)
apply *hypsubst*
apply *clarify*
apply(*drule jmm'-redI*)
apply(*simp add*: *final-thread.actions-ok-iff*)
apply *clarify*
apply(*subst* (*2*) *split-paired-Ex*)
apply(*subst* (*2*) *split-paired-Ex*)
apply(*subst* (*2*) *split-paired-Ex*)
apply(*rule exI conjI*)+
apply(*rule multithreaded-base.init-fin.intros*)
apply(*simp*)
apply(*simp add*: *final-thread.actions-ok-iff*)
apply(*blast intro*: *multithreaded-base.init-fin.intros*)
apply(*blast intro*: *multithreaded-base.init-fin.intros*)
done

lemma *if-mredT-heap-read-not-stuck*:
multithreaded-base.redT (*final-thread.init-fin-final final-expr*) (*multithreaded-base.init-fin final-expr* (*J-heap-base.mred addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr jmm-heap-read jmm-heap-write P*)) *convert-RA' s tta s'*
⟹ ∃ *tta s'*. *multithreaded-base.redT* (*final-thread.init-fin-final final-expr*) (*multithreaded-base.init-fin final-expr* (*J-heap-base.mred addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr* (*heap-base.heap-read-typed typeof-addr jmm-heap-read P*) *jmm-heap-write P*)) *convert-RA' s tta s'*
apply(*erule multithreaded-base.redT.cases*)
apply *hypsubst*
apply(*drule* (*1*) *if-mred-heap-read-not-stuck*)
apply(*erule exE*)+
apply(*rename-tac ta' x'h'*)
apply(*insert redT-updWs-total*)
apply(*erule-tac x*=*t* **in** *meta-allE*)

apply(*erule-tac x=wset s* **in** *meta-allE*)
apply(*erule-tac x=*$\{\!|ta'|\!\}_w$ **in** *meta-allE*)
apply *clarsimp*
apply(*rule exI*)+
apply(*auto intro*!: *multithreaded-base.redT.intros*)[*1*]
apply *hypsubst*
apply(*rule exI conjI*)+
apply(*rule multithreaded-base.redT.redT-acquire*)
apply *assumption*+
done

lemma *J-*$\mathcal{E}$*-heap-read-typedD*:
E ∈ *J-heap-base.J-*$\mathcal{E}$ *addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate* (λ-. *typeof-addr*) (*heap-base.heap-read-typed* (λ-. *typeof-addr*) *jmm-heap-read P*) *jmm-heap-write P C M vs status*
⟹ *E* ∈ *J-heap-base.J-*$\mathcal{E}$ *addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate* (λ-. *typeof-addr*) *jmm-heap-read jmm-heap-write P C M vs status*
apply(*erule imageE*, *hypsubst*)
apply(*rule imageI*)
apply(*erule multithreaded-base.*$\mathcal{E}$*.cases*, *hypsubst*)
apply(*rule multithreaded-base.*$\mathcal{E}$*.intros*)
apply(*subst* (*asm*) *if-mred-heap-read-typedD*[*abs-def*])
apply(*erule if-mthr-Runs-heap-read-typedD*)
apply(*erule if-mredT-heap-read-not-stuck*[**where** *typeof-addr=*λ-. *typeof-addr*, *unfolded if-mred-heap-read-typedD*[*abs-def*]])
done

lemma *J-*$\mathcal{E}$*-typesafe-subset*: *jmm'-J-*$\mathcal{E}$ *P C M vs status* ⊆ *jmm-J-*$\mathcal{E}$ *P C M vs status*
unfolding *jmm-typeof-addr-def*[*abs-def*]
by(*rule subsetI*)(*erule J-*$\mathcal{E}$*-heap-read-typedD*)

lemma *J-legal-typesafe1*:
assumes *wfP*: *wf-J-prog P*
and *ok*: *jmm-wf-start-state P C M vs*
and *legal*: *legal-execution P* (*jmm-J-*$\mathcal{E}$ *P C M vs status*) (*E*, *ws*)
shows *legal-execution P* (*jmm'-J-*$\mathcal{E}$ *P C M vs status*) (*E*, *ws*)
proof −
let *?*$\mathcal{E}$ = *jmm-J-*$\mathcal{E}$ *P C M vs status*
let *?*$\mathcal{E}'$ = *jmm'-J-*$\mathcal{E}$ *P C M vs status*
from *legal* **obtain** *J*
where *justified*: *P* ⊢ (*E*, *ws*) *justified-by J*
and *range*: *range* (*justifying-exec* ∘ *J*) ⊆ *?*$\mathcal{E}$
and *E*: *E* ∈ *?*$\mathcal{E}$ **and** *wf*: *P* ⊢ (*E*, *ws*) √ **by**(*auto simp add*: *gen-legal-execution.simps*)
let *?J* = *J*(*0* := (|*committed* = {}, *justifying-exec* = *justifying-exec* (*J 1*), *justifying-ws* = *justifying-ws* (*J 1*), *action-translation* = *id*|))

from *wfP* **have** *wf-sys*: *wf-syscls P* **by**(*rule wf-prog-wf-syscls*)

from *justified* **have** *P* ⊢ (*justifying-exec* (*J 1*), *justifying-ws* (*J 1*)) √
by(*simp add*: *justification-well-formed-def*)
with *justified* **have** *P* ⊢ (*E*, *ws*) *justified-by ?J* **by**(*rule drop-0th-justifying-exec*)
moreover have *range* (*justifying-exec* ∘ *?J*) ⊆ *?*$\mathcal{E}'$
proof
fix ξ
assume ξ ∈ *range* (*justifying-exec* ∘ *?J*)
then obtain *n* **where** ξ = *justifying-exec* (*?J n*) **by** *auto*

```
    then obtain n where ξ: ξ = justifying-exec (J n) and n: n > 0 by(auto split: if-split-asm)
    from range ξ have ξ ∈ ?E by auto
    thus ξ ∈ ?E' unfolding jmm-typeof-addr'-conv-jmm-type-addr[symmetric, abs-def]
    proof(rule J-E-heap-read-typedI)
      fix ad al v T
      assume read: NormalAction (ReadMem ad al v) ∈ snd ' lset ξ
        and adal: P ⊢jmm ad@al : T
      from read obtain a where a: enat a < llength ξ action-obs ξ a = NormalAction (ReadMem ad
al v)
        unfolding lset-conv-lnth by(auto simp add: action-obs-def)
    with J-allocated-heap-conf'.mred-known-addrs-typing'[OF jmm-J-allocated-heap-conf' wfP jmm-start-heap-ok]
        J-heap-conf.J-start-state-sconf-type-ok[OF jmm-J-heap-conf wfP ok]
        wf-sys is-justified-by-imp-is-weakly-justified-by[OF justified wf] range n
      have ∃ T. P ⊢jmm ad@al : T ∧ P ⊢jmm v :≤ T
        unfolding jmm-typeof-addr'-conv-jmm-type-addr[symmetric, abs-def] ξ
        by(rule known-addrs-typing'.read-value-typeable-justifying)
      thus P ⊢jmm v :≤ T using adal
        by(auto dest: jmm.addr-loc-type-fun[unfolded jmm-typeof-addr-conv-jmm-typeof-addr', unfolded
heap-base'.addr-loc-type-conv-addr-loc-type])
    qed
  qed
  moreover from E have E ∈ ?E'
    unfolding jmm-typeof-addr'-conv-jmm-type-addr[symmetric, abs-def]
  proof(rule J-E-heap-read-typedI)
    fix ad al v T
    assume read: NormalAction (ReadMem ad al v) ∈ snd ' lset E
      and adal: P ⊢jmm ad@al : T
    from read obtain a where a: enat a < llength E action-obs E a = NormalAction (ReadMem ad
al v)
      unfolding lset-conv-lnth by(auto simp add: action-obs-def)
    with jmm-J-allocated-heap-conf' wfP ok legal-imp-weakly-legal-execution[OF legal]
    have ∃ T. P ⊢jmm ad@al : T ∧ P ⊢jmm v :≤ T
      unfolding jmm-typeof-addr'-conv-jmm-type-addr[symmetric, abs-def]
      by(rule J-allocated-heap-conf'.J-legal-read-value-typeable)
    thus P ⊢jmm v :≤ T using adal
      by(auto dest: jmm.addr-loc-type-fun[unfolded jmm-typeof-addr-conv-jmm-typeof-addr', unfolded
heap-base'.addr-loc-type-conv-addr-loc-type])
  qed
  ultimately show ?thesis using wf unfolding gen-legal-execution.simps by blast
qed

lemma J-weakly-legal-typesafe1:
  assumes wfP: wf-J-prog P
  and ok: jmm-wf-start-state P C M vs
  and legal: weakly-legal-execution P (jmm-J-E P C M vs status) (E, ws)
  shows weakly-legal-execution P (jmm'-J-E P C M vs status) (E, ws)
proof −
  let ?E = jmm-J-E P C M vs status
  let ?E' = jmm'-J-E P C M vs status
  from legal obtain J
    where justified: P ⊢ (E, ws) weakly-justified-by J
    and range: range (justifying-exec ∘ J) ⊆ ?E
    and E: E ∈ ?E and wf: P ⊢ (E, ws) √ by(auto simp add: gen-legal-execution.simps)
 let ?J = J(0 := (|committed = {}, justifying-exec = justifying-exec (J 1), justifying-ws = justifying-ws
```

```
(J 1), action-translation = id⦈)

  from wfP have wf-sys: wf-syscls P by(rule wf-prog-wf-syscls)

  from justified have P ⊢ (justifying-exec (J 1), justifying-ws (J 1)) √
    by(simp add: justification-well-formed-def)
  with justified have P ⊢ (E, ws) weakly-justified-by ?J by(rule drop-0th-weakly-justifying-exec)
  moreover have range (justifying-exec ∘ ?J) ⊆ ?E'
  proof
    fix ξ
    assume ξ ∈ range (justifying-exec ∘ ?J)
    then obtain n where ξ = justifying-exec (?J n) by auto
    then obtain n where ξ: ξ = justifying-exec (J n) and n: n > 0 by(auto split: if-split-asm)
    from range ξ have ξ ∈ ?E by auto
    thus ξ ∈ ?E' unfolding jmm-typeof-addr'-conv-jmm-type-addr[symmetric, abs-def]
    proof(rule J-E-heap-read-typedI)
      fix ad al v T
      assume read: NormalAction (ReadMem ad al v) ∈ snd ' lset ξ
        and adal: P ⊢jmm ad@al : T
      from read obtain a where a: enat a < llength ξ action-obs ξ a = NormalAction (ReadMem ad
al v)
        unfolding lset-conv-lnth by(auto simp add: action-obs-def)
      with J-allocated-heap-conf'.mred-known-addrs-typing'[OF jmm-J-allocated-heap-conf' wfP jmm-start-heap-ok]
        J-heap-conf.J-start-state-sconf-type-ok[OF jmm-J-heap-conf wfP ok]
        wf-sys justified range n
      have ∃ T. P ⊢jmm ad@al : T ∧ P ⊢jmm v :≤ T
        unfolding jmm-typeof-addr'-conv-jmm-type-addr[symmetric, abs-def] ξ
        by(rule known-addrs-typing'.read-value-typeable-justifying)
      thus P ⊢jmm v :≤ T using adal
        by(auto dest: jmm.addr-loc-type-fun[unfolded jmm-typeof-addr-conv-jmm-typeof-addr', unfolded
heap-base'.addr-loc-type-conv-addr-loc-type])
    qed
  qed
  moreover from E have E ∈ ?E'
    unfolding jmm-typeof-addr'-conv-jmm-type-addr[symmetric, abs-def]
  proof(rule J-E-heap-read-typedI)
    fix ad al v T
    assume read: NormalAction (ReadMem ad al v) ∈ snd ' lset E
      and adal: P ⊢jmm ad@al : T
    from read obtain a where a: enat a < llength E action-obs E a = NormalAction (ReadMem ad
al v)
      unfolding lset-conv-lnth by(auto simp add: action-obs-def)
    with jmm-J-allocated-heap-conf' wfP ok legal
    have ∃ T. P ⊢jmm ad@al : T ∧ P ⊢jmm v :≤ T
      unfolding jmm-typeof-addr'-conv-jmm-type-addr[symmetric, abs-def]
      by(rule J-allocated-heap-conf'.J-legal-read-value-typeable)
    thus P ⊢jmm v :≤ T using adal
      by(auto dest: jmm.addr-loc-type-fun[unfolded jmm-typeof-addr-conv-jmm-typeof-addr', unfolded
heap-base'.addr-loc-type-conv-addr-loc-type])
  qed
  ultimately show ?thesis using wf unfolding gen-legal-execution.simps by blast
qed

lemma J-legal-typesafe2:
```

assumes *legal*: *legal-execution P* (*jmm'-J-$\mathcal{E}$ P C M vs status*) (*E*, *ws*)
shows *legal-execution P* (*jmm-J-$\mathcal{E}$ P C M vs status*) (*E*, *ws*)
proof −
let *?$\mathcal{E}$* = *jmm-J-$\mathcal{E}$ P C M vs status*
let *?$\mathcal{E}'$* = *jmm'-J-$\mathcal{E}$ P C M vs status*
from *legal* **obtain** *J*
where *justified*: *P* ⊢ (*E*, *ws*) *justified-by J*
and *range*: *range* (*justifying-exec* ∘ *J*) ⊆ *?$\mathcal{E}'$*
and *E*: *E* ∈ *?$\mathcal{E}'$* **and** *wf*: *P* ⊢ (*E*, *ws*) √ **by**(*auto simp add*: *gen-legal-execution.simps*)
from *range E* **have** *range* (*justifying-exec* ∘ *J*) ⊆ *?$\mathcal{E}$ E* ∈ *?$\mathcal{E}$*
using *J-$\mathcal{E}$-typesafe-subset*[*of P status C M vs*] **by** *blast*+
with *justified wf*
show *?thesis* **by**(*auto simp add*: *gen-legal-execution.simps*)
qed

lemma *J-weakly-legal-typesafe2*:
assumes *legal*: *weakly-legal-execution P* (*jmm'-J-$\mathcal{E}$ P C M vs status*) (*E*, *ws*)
shows *weakly-legal-execution P* (*jmm-J-$\mathcal{E}$ P C M vs status*) (*E*, *ws*)
proof −
let *?$\mathcal{E}$* = *jmm-J-$\mathcal{E}$ P C M vs status*
let *?$\mathcal{E}'$* = *jmm'-J-$\mathcal{E}$ P C M vs status*
from *legal* **obtain** *J*
where *justified*: *P* ⊢ (*E*, *ws*) *weakly-justified-by J*
and *range*: *range* (*justifying-exec* ∘ *J*) ⊆ *?$\mathcal{E}'$*
and *E*: *E* ∈ *?$\mathcal{E}'$* **and** *wf*: *P* ⊢ (*E*, *ws*) √ **by**(*auto simp add*: *gen-legal-execution.simps*)
from *range E* **have** *range* (*justifying-exec* ∘ *J*) ⊆ *?$\mathcal{E}$ E* ∈ *?$\mathcal{E}$*
using *J-$\mathcal{E}$-typesafe-subset*[*of P status C M vs*] **by** *blast*+
with *justified wf*
show *?thesis* **by**(*auto simp add*: *gen-legal-execution.simps*)
qed

theorem *J-weakly-legal-typesafe*:
assumes *wf-J-prog P*
and *jmm-wf-start-state P C M vs*
shows *weakly-legal-execution P* (*jmm-J-$\mathcal{E}$ P C M vs status*) = *weakly-legal-execution P* (*jmm'-J-$\mathcal{E}$ P C M vs status*)
apply(*rule ext iffI*)+
apply(*clarify*, *erule J-weakly-legal-typesafe1*[*OF assms*])
apply(*clarify*, *erule J-weakly-legal-typesafe2*)
done

theorem *J-legal-typesafe*:
assumes *wf-J-prog P*
and *jmm-wf-start-state P C M vs*
shows *legal-execution P* (*jmm-J-$\mathcal{E}$ P C M vs status*) = *legal-execution P* (*jmm'-J-$\mathcal{E}$ P C M vs status*)
apply(*rule ext iffI*)+
apply(*clarify*, *erule J-legal-typesafe1*[*OF assms*])
apply(*clarify*, *erule J-legal-typesafe2*)
done

end

8.21 JMM type safety for bytecode

theory *JMM-JVM-Typesafe*
imports
JMM-Typesafe2
DRF-JVM
begin

locale *JVM-allocated-heap-conf'* =
h: *JVM-heap-conf*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate λ*-. typeof-addr heap-read heap-write hconf*
P
+
h: *JVM-allocated-heap*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate λ*-. typeof-addr heap-read heap-write*
allocated
P
+
heap''
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr heap-read heap-write
P
for *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'addr* ⇀ *htype*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *bool*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *hconf* :: *'heap* ⇒ *bool*
and *allocated* :: *'heap* ⇒ *'addr set*
and *P* :: *'addr jvm-prog*

sublocale *JVM-allocated-heap-conf'* < *h*: *JVM-allocated-heap-conf*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate λ*-. typeof-addr heap-read heap-write hconf allocated*
P
by(*unfold-locales*)

context *JVM-allocated-heap-conf'* **begin**

lemma *exec-instr-New-type-match*:
⟦ (*ta*, *s'*) ∈ *h.exec-instr i P t h stk loc C M pc frs*; *NewHeapElem ad CTn* ∈ *set* {|*ta*|}$_o$; *typeof-addr ad* ≠ *None* ⟧
⟹ *typeof-addr ad* = ⌊*CTn*⌋
by(*cases i*)(*auto split*: *if-split-asm prod.split-asm dest*: *allocate-typeof-addr-SomeD red-external-aggr-New-type-match*)

lemma *mexecd-New-type-match*:
⟦ *h.mexecd P t* (*xcpfrs*, *h*) *ta* (*xcpfrs'*, *h'*); *NewHeapElem ad CTn* ∈ *set* {|*ta*|}$_o$; *typeof-addr ad* ≠ *None* ⟧
⟹ *typeof-addr ad* = ⌊*CTn*⌋
apply(*cases xcpfrs*)
apply(*cases xcpfrs'*)
apply(*simp add*: *split-beta*)
apply(*erule h.jvmd-NormalE*)
apply(*cases fst xcpfrs*)
apply(*auto 4 3 dest*: *exec-instr-New-type-match*)
done

lemma *mexecd-known-addrs-typing'*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *ok*: *h.start-heap-ok*
shows *known-addrs-typing' addr2thread-id thread-id2addr empty-heap allocate typeof-addr heap-write allocated h.jvm-known-addrs JVM-final* (*h.mexecd P*) (λ*t* (*xcp*, *frs*) *h*. *h.correct-state* Φ *t* (*xcp*, *h*, *frs*)) *P*
proof −
interpret *known-addrs-typing*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate λ-. *typeof-addr heap-read heap-write*
allocated h.jvm-known-addrs
JVM-final h.mexecd P λ*t* (*xcp*, *frs*) *h*. *h.correct-state* Φ *t* (*xcp*, *h*, *frs*)
P
using *assms* **by**(*rule h.mexecd-known-addrs-typing*)

show *?thesis* **by**(*unfold-locales*)(*erule mexecd-New-type-match*)
qed

lemma *JVM-weakly-legal-read-value-typeable*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *wf-start*: *h.wf-start-state P C M vs*
and *legal*: *weakly-legal-execution P* (*h.JVMd-*$\mathcal{E}$ *P C M vs status*) (*E*, *ws*)
and *a*: *enat a* < *llength E*
and *read*: *action-obs E a* = *NormalAction* (*ReadMem ad al v*)
shows ∃ *T*. *P* ⊢ *ad*@*al* : *T* ∧ *P* ⊢ *v* :≤ *T*
proof −
note *wf*
moreover from *wf-start* **have** *h.start-heap-ok* **by** *cases*
moreover from *wf wf-start*
have *ts-ok* (λ*t* (*xcp*, *frs*) *h*. *h.correct-state* Φ *t* (*xcp*, *h*, *frs*)) (*thr* (*h.JVM-start-state P C M vs*)) *h.start-heap*
using *h.correct-jvm-state-initial*[*OF wf wf-start*]
by(*simp add*: *h.correct-jvm-state-def h.start-state-def split-beta*)
moreover from *wf* **obtain** *wf-md* **where** *wf'*: *wf-prog wf-md P* **by**(*blast dest*: *wt-jvm-progD*)
hence *wf-syscls P* **by**(*rule wf-prog-wf-syscls*)
ultimately show *?thesis* **using** *legal a read*
by(*rule known-addrs-typing'.weakly-legal-read-value-typeable*[*OF mexecd-known-addrs-typing'*])
qed

end

abbreviation *jmm-JVMd-*$\mathcal{E}$
:: *addr jvm-prog* $\Rightarrow$ *cname* $\Rightarrow$ *mname* $\Rightarrow$ *addr val list* $\Rightarrow$ *status* $\Rightarrow$ (*addr* $\times$ (*addr*, *addr*) *obs-event action*) *llist set*
where
jmm-JVMd-$\mathcal{E}$ *P* $\equiv$
JVM-heap-base.JVMd-$\mathcal{E}$ *addr2thread-id thread-id2addr jmm-spurious-wakeups jmm-empty jmm-allocate* (*jmm-typeof-addr P*) *jmm-heap-read jmm-heap-write P*

abbreviation *jmm'-JVMd-*$\mathcal{E}$
:: *addr jvm-prog* $\Rightarrow$ *cname* $\Rightarrow$ *mname* $\Rightarrow$ *addr val list* $\Rightarrow$ *status* $\Rightarrow$ (*addr* $\times$ (*addr*, *addr*) *obs-event action*) *llist set*
where
jmm'-JVMd-$\mathcal{E}$ *P* $\equiv$
JVM-heap-base.JVMd-$\mathcal{E}$ *addr2thread-id thread-id2addr jmm-spurious-wakeups jmm-empty jmm-allocate* (*jmm-typeof-addr P*) (*jmm-heap-read-typed P*) *jmm-heap-write P*

abbreviation *jmm-JVM-start-state*
:: *addr jvm-prog* $\Rightarrow$ *cname* $\Rightarrow$ *mname* $\Rightarrow$ *addr val list* $\Rightarrow$ (*addr*,*thread-id*,*addr jvm-thread-state*,*JMM-heap*,*addr*) *state*
where *jmm-JVM-start-state* $\equiv$ *JVM-heap-base.JVM-start-state addr2thread-id jmm-empty jmm-allocate*

lemma *jmm-JVM-heap-conf*:
JVM-heap-conf addr2thread-id thread-id2addr jmm-empty jmm-allocate (*jmm-typeof-addr P*) *jmm-heap-write jmm-hconf P*
by(*unfold-locales*)

lemma *jmm-JVMd-allocated-heap-conf'*:
JVM-allocated-heap-conf' addr2thread-id thread-id2addr jmm-empty jmm-allocate (*jmm-typeof-addr' P*) *jmm-heap-write jmm-hconf jmm-allocated P*
apply(*rule JVM-allocated-heap-conf'.intro*)
apply(*unfold jmm-typeof-addr'-conv-jmm-typeof-addr*)
apply(*unfold-locales*)
done

lemma *exec-instr-heap-read-typed*:
(*ta*, *xcphfrs'*) $\in$ *JVM-heap-base.exec-instr addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate* (λ- :: *'heap. typeof-addr*) (*heap-base.heap-read-typed* (λ- :: *'heap. typeof-addr*) *heap-read P*) *heap-write i P t h stk loc C M pc frs* $\longleftrightarrow$
(*ta*, *xcphfrs'*) $\in$ *JVM-heap-base.exec-instr addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate* (λ- :: *'heap. typeof-addr*) *heap-read heap-write i P t h stk loc C M pc frs* $\wedge$
($\forall$ *ad al v T. ReadMem ad al v* $\in$ *set* $\{\!|ta|\!\}_o$ $\longrightarrow$ *heap-base'.addr-loc-type TYPE('heap) typeof-addr P ad al T* $\longrightarrow$ *heap-base'.conf TYPE('heap) typeof-addr P v T*)
apply(*cases i*)
apply(*simp-all add*: *JVM-heap-base.exec-instr.simps split-beta cong*: *conj-cong*)

apply(*auto dest*: *heap-base.heap-read-typed-into-heap-read del*: *disjCI*)[*5*]
apply(*blast dest*: *heap-base.heap-read-typed-typed heap-base'.addr-loc-type-conv-addr-loc-type*[*THEN fun-cong*, *THEN fun-cong*, *THEN fun-cong*, *THEN iffD2*] *heap-base'.conf-conv-conf*[*THEN fun-cong*, *THEN fun-cong*, *THEN iffD1*])
apply(*auto dest*: *heap-base'.addr-loc-type-conv-addr-loc-type*[*THEN fun-cong*, *THEN fun-cong*, *THEN fun-cong*, *THEN iffD1*] *intro*: *heap-base'.conf-conv-conf*[*THEN fun-cong*, *THEN fun-cong*, *THEN iffD2*] *heap-base.heap-read-typedI*)[*1*]

apply(*blast dest*: *heap-base.heap-read-typed-typed heap-base'.addr-loc-type-conv-addr-loc-type*[*THEN fun-cong*, *THEN fun-cong*, *THEN fun-cong*, *THEN iffD2*] *heap-base'.conf-conv-conf*[*THEN fun-cong*, *THEN fun-cong*, *THEN iffD1*])
apply(*auto dest*: *heap-base'.addr-loc-type-conv-addr-loc-type*[*THEN fun-cong*, *THEN fun-cong*, *THEN fun-cong*, *THEN iffD1*] *intro*: *heap-base'.conf-conv-conf*[*THEN fun-cong*, *THEN fun-cong*, *THEN iffD2*] *heap-base.heap-read-typedI*)[*1*]
apply(*blast dest*: *heap-base.heap-read-typed-typed heap-base'.addr-loc-type-conv-addr-loc-type*[*THEN fun-cong*, *THEN fun-cong*, *THEN fun-cong*, *THEN iffD2*] *heap-base'.conf-conv-conf*[*THEN fun-cong*, *THEN fun-cong*, *THEN iffD1*])
subgoal by(*auto dest*: *heap-base.heap-read-typed-into-heap-read*)
apply(*blast dest*: *heap-base.heap-read-typed-typed heap-base'.addr-loc-type-conv-addr-loc-type*[*THEN fun-cong*, *THEN fun-cong*, *THEN fun-cong*, *THEN iffD2*] *heap-base'.conf-conv-conf*[*THEN fun-cong*, *THEN fun-cong*, *THEN iffD1*])
subgoal by(*auto dest*: *heap-base'.addr-loc-type-conv-addr-loc-type*[*THEN fun-cong*, *THEN fun-cong*, *THEN fun-cong*, *THEN iffD1*] *intro*: *heap-base'.conf-conv-conf*[*THEN fun-cong*, *THEN fun-cong*, *THEN iffD2*] *heap-base.heap-read-typedI*)[*1*]
subgoal by(*auto 4 3 dest*: *heap-base.heap-read-typed-into-heap-read intro*: *heap-base.heap-read-typedI simp add*: *heap-base'.conf-conv-conf heap-base'.addr-loc-type-conv-addr-loc-type*)

apply(*subst red-external-aggr-heap-read-typed*)
apply(*fastforce*)
apply *auto*
done

lemma *exec-heap-read-typed*:
(*ta*, *xcphfrs'*) ∈ *JVM-heap-base.exec addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate* (λ- :: *'heap. typeof-addr*) (*heap-base.heap-read-typed* (λ-. *typeof-addr*) *heap-read P*) *heap-write P t xcphfrs* ⟷
(*ta*, *xcphfrs'*) ∈ *JVM-heap-base.exec addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate* (λ- :: *'heap. typeof-addr*) *heap-read heap-write P t xcphfrs* ∧
(∀ *ad al v T*. *ReadMem ad al v* ∈ *set* ⦃*ta*⦄$_o$ ⟶ *heap-base'.addr-loc-type TYPE*(*'heap*) *typeof-addr P ad al T* ⟶ *heap-base'.conf TYPE*(*'heap*) *typeof-addr P v T*)
apply(*cases xcphfrs*)
apply(*cases fst xcphfrs*)
apply(*case-tac* [!] *snd* (*snd xcphfrs*))
apply(*auto simp add*: *JVM-heap-base.exec.simps exec-instr-heap-read-typed*)
done

lemma *exec-1-d-heap-read-typed*:
JVM-heap-base.exec-1-d addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate (λ- :: *'heap. typeof-addr*) (*heap-base.heap-read-typed* (λ-. *typeof-addr*) *heap-read P*) *heap-write P t* (*Normal xcphfrs*) *ta* (*Normal xcphfrs'*) ⟷
JVM-heap-base.exec-1-d addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate (λ- :: *'heap. typeof-addr*) *heap-read heap-write P t* (*Normal xcphfrs*) *ta* (*Normal xcphfrs'*) ∧
(∀ *ad al v T*. *ReadMem ad al v* ∈ *set* ⦃*ta*⦄$_o$ ⟶ *heap-base'.addr-loc-type TYPE*(*'heap*) *typeof-addr P ad al T* ⟶ *heap-base'.conf TYPE*(*'heap*) *typeof-addr P v T*)
by(*auto elim*!: *JVM-heap-base.jvmd-NormalE intro*: *JVM-heap-base.exec-1-d-NormalI simp add*: *exec-heap-read-typ JVM-heap-base.exec-d-def*)

lemma *mexecd-heap-read-typed*:
JVM-heap-base.mexecd addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate (λ- :: *'heap. typeof-addr*) (*heap-base.heap-read-typed* (λ- :: *'heap. typeof-addr*) *heap-read P*) *heap-write P t xcpfrsh ta xcpfrsh'* ⟷
JVM-heap-base.mexecd addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate (λ- ::

'heap. typeof-addr) heap-read heap-write P t xcpfrsh ta xcpfrsh' ∧
(∀ ad al v T. ReadMem ad al v ∈ set {|ta|}$_o$ ⟶ heap-base'.addr-loc-type TYPE('heap) typeof-addr P ad al T ⟶ heap-base'.conf TYPE('heap) typeof-addr P v T)
by(*simp add*: *split-beta exec-1-d-heap-read-typed*)

lemma *if-mexecd-heap-read-typed*:
multithreaded-base.init-fin JVM-final (JVM-heap-base.mexecd addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate (λ- :: 'heap. typeof-addr) (heap-base.heap-read-typed (λ- :: 'heap. typeof-addr) heap-read P) heap-write P) t xh ta x'h' ⟷
if-heap-read-typed JVM-final (JVM-heap-base.mexecd addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate (λ- :: 'heap. typeof-addr) heap-read heap-write P) typeof-addr P t xh ta x'h'
unfolding *multithreaded-base.init-fin.simps*
by(*subst mexecd-heap-read-typed*) *fastforce*

lemma *JVMd-$\mathcal{E}$-heap-read-typedI*:
⟦ E ∈ JVM-heap-base.JVMd-$\mathcal{E}$ addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate (λ- :: 'heap. typeof-addr) heap-read heap-write P C M vs status;
⋀ad al v T. ⟦ NormalAction (ReadMem ad al v) ∈ snd ` lset E; heap-base'.addr-loc-type TYPE('heap) typeof-addr P ad al T ⟧ ⟹ heap-base'.conf TYPE('heap) typeof-addr P v T ⟧
⟹ E ∈ JVM-heap-base.JVMd-$\mathcal{E}$ addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate (λ- :: 'heap. typeof-addr) (heap-base.heap-read-typed (λ- :: 'heap. typeof-addr) heap-read P) heap-write P C M vs status
apply(*erule imageE*, *hypsubst*)
apply(*rule imageI*)
apply(*erule multithreaded-base.$\mathcal{E}$.cases*, *hypsubst*)
apply(*rule multithreaded-base.$\mathcal{E}$.intros*)
apply(*subst if-mexecd-heap-read-typed*[*abs-def*])
apply(*erule if-mthr-Runs-heap-read-typedI*)
apply(*auto simp add*: *image-Un lset-lmap*[*symmetric*] *lmap-lconcat llist.map-comp o-def split-def simp del*: *lset-lmap*)
done

lemma *jmm'-exec-instrI*:
⟦ (ta, xcphfrs) ∈ JVM-heap-base.exec-instr addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr jmm-heap-read jmm-heap-write i P t h stk loc C M pc frs;
final-thread.actions-ok (final-thread.init-fin-final JVM-final) s t ta ⟧
⟹ ∃ta xcphfrs. (ta, xcphfrs) ∈ JVM-heap-base.exec-instr addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr (heap-base.heap-read-typed typeof-addr jmm-heap-read P) jmm-heap-write i P t h stk loc C M pc frs ∧ final-thread.actions-ok (final-thread.init-fin-final JVM-final) s t ta
apply(*cases i*)
apply(*auto simp add*: *JVM-heap-base.exec-instr.simps split-beta final-thread.actions-ok-iff intro*!: *jmm-heap-read-typed-default-val rev-image-eqI simp del*: *split-paired-Ex split*: *if-split-asm*)
subgoal for *F D*
apply(*cases hd (tl stk) = (default-val (THE T. heap-base.addr-loc-type typeof-addr P h (the-Addr (hd (tl (tl stk)))) (CField D F) T))*)
subgoal by(*auto simp add*: *JVM-heap-base.exec-instr.simps split-beta final-thread.actions-ok-iff intro*!: *jmm-heap-read-typed-default-val rev-image-eqI simp del*: *split-paired-Ex split*: *if-split-asm del*: *disjCI intro*!: *disjI1 exI*)
subgoal by(*auto simp add*: *JVM-heap-base.exec-instr.simps split-beta final-thread.actions-ok-iff intro*!: *jmm-heap-read-typed-default-val rev-image-eqI simp del*: *split-paired-Ex split*: *if-split-asm del*: *disjCI intro*!: *disjI2 exI*)
done
subgoal for *F D*

apply(*cases hd (tl stk) = (default-val (THE T. heap-base.addr-loc-type typeof-addr P h (the-Addr (hd (tl (tl stk)))) (CField D F) T))*)
subgoal by(*auto simp add*: *JVM-heap-base.exec-instr.simps split-beta final-thread.actions-ok-iff intro*!: *jmm-heap-read-typed-default-val rev-image-eqI simp del*: *split-paired-Ex split*: *if-split-asm del*: *disjCI intro*!: *disjI1 exI jmm-heap-write.intros*)
subgoal by(*rule exI conjI disjI2 refl jmm-heap-read-typed-default-val*|*assumption*)+ *auto*
done
subgoal by(*drule red-external-aggr-heap-read-typedI*)(*fastforce simp add*: *final-thread.actions-ok-iff simp del*: *split-paired-Ex*)+
done

lemma *jmm'-execI*:
⟦ (*ta*, *xcphfrs'*) ∈ *JVM-heap-base.exec addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr jmm-heap-read jmm-heap-write P t xcphfrs*;
final-thread.actions-ok (*final-thread.init-fin-final JVM-final*) *s t ta* ⟧
⟹ ∃ *ta xcphfrs'*. (*ta*, *xcphfrs'*) ∈ *JVM-heap-base.exec addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr* (*heap-base.heap-read-typed typeof-addr jmm-heap-read P*) *jmm-heap-write P t xcphfrs* ∧ *final-thread.actions-ok* (*final-thread.init-fin-final JVM-final*) *s t ta*
apply(*cases xcphfrs*)
apply(*cases snd* (*snd xcphfrs*))
apply(*simp add*: *JVM-heap-base.exec.simps*)
apply(*cases fst xcphfrs*)
apply(*fastforce simp add*: *JVM-heap-base.exec.simps dest*!: *jmm'-exec-instrI*)+
done

lemma *jmm'-execdI*:
⟦ *JVM-heap-base.exec-1-d addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr jmm-heap-read jmm-heap-write P t* (*Normal xcphfrs*) *ta* (*Normal xcphfrs'*);
final-thread.actions-ok (*final-thread.init-fin-final JVM-final*) *s t ta* ⟧
⟹ ∃ *ta xcphfrs'*. *JVM-heap-base.exec-1-d addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr* (*heap-base.heap-read-typed typeof-addr jmm-heap-read P*) *jmm-heap-write P t* (*Normal xcphfrs*) *ta* (*Normal xcphfrs'*) ∧ *final-thread.actions-ok* (*final-thread.init-fin-final JVM-final*) *s t ta*
apply(*erule JVM-heap-base.jvmd-NormalE*)
apply(*drule* (*1*) *jmm'-execI*)
apply(*force intro*: *JVM-heap-base.exec-1-d-NormalI simp add*: *JVM-heap-base.exec-d-def*)
done

lemma *jmm'-mexecdI*:
⟦ *JVM-heap-base.mexecd addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr jmm-heap-read jmm-heap-write P t xcpfrsh ta xcpfrsh'*;
final-thread.actions-ok (*final-thread.init-fin-final JVM-final*) *s t ta* ⟧
⟹ ∃ *ta xcpfrsh'*. *JVM-heap-base.mexecd addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr* (*heap-base.heap-read-typed typeof-addr jmm-heap-read P*) *jmm-heap-write P t xcpfrsh ta xcpfrsh'* ∧ *final-thread.actions-ok* (*final-thread.init-fin-final JVM-final*) *s t ta*
by(*simp add*: *split-beta*)(*drule* (*1*) *jmm'-execdI*, *auto 4 10*)

lemma *if-mexecd-heap-read-not-stuck*:
⟦ *multithreaded-base.init-fin JVM-final* (*JVM-heap-base.mexecd addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr jmm-heap-read jmm-heap-write P*) *t xh ta x'h'*;
final-thread.actions-ok (*final-thread.init-fin-final JVM-final*) *s t ta* ⟧
⟹ ∃ *ta x'h'*. *multithreaded-base.init-fin JVM-final* (*JVM-heap-base.mexecd addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr* (*heap-base.heap-read-typed typeof-addr jmm-heap-read P*) *jmm-heap-write P*) *t xh ta x'h'* ∧ *final-thread.actions-ok* (*final-thread.init-fin-final JVM-final*) *s t ta*

apply(*erule multithreaded-base.init-fin.cases*)
apply *hypsubst*
apply(*drule jmm'-mexecdI*)
apply(*simp add*: *final-thread.actions-ok-iff*)
apply *clarify*
apply(*subst* (*2*) *split-paired-Ex*)
apply(*subst* (*2*) *split-paired-Ex*)
apply(*subst* (*2*) *split-paired-Ex*)
apply(*rule exI conjI*)+
apply(*rule multithreaded-base.init-fin.intros*)
apply *simp*
apply(*simp add*: *final-thread.actions-ok-iff*)
apply(*blast intro*: *multithreaded-base.init-fin.intros*)
apply(*blast intro*: *multithreaded-base.init-fin.intros*)
done

lemma *if-mExecd-heap-read-not-stuck*:
multithreaded-base.redT (*final-thread.init-fin-final JVM-final*) (*multithreaded-base.init-fin JVM-final* (*JVM-heap-base.mexecd addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr jmm-heap-read jmm-heap-write P*)) *convert-RA' s tta s'*
$\Longrightarrow \exists$ *tta s'*. *multithreaded-base.redT* (*final-thread.init-fin-final JVM-final*) (*multithreaded-base.init-fin JVM-final* (*JVM-heap-base.mexecd addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate typeof-addr* (*heap-base.heap-read-typed typeof-addr jmm-heap-read P*) *jmm-heap-write P*)) *convert-RA' s tta s'*
apply(*erule multithreaded-base.redT.cases*)
apply *hypsubst*
thm *if-mexecd-heap-read-not-stuck*
apply(*drule* (*1*) *if-mexecd-heap-read-not-stuck*)
apply(*erule exE*)+
apply(*rename-tac ta' x'h'*)
apply(*insert redT-updWs-total*)
apply(*erule-tac x=t* **in** *meta-allE*)
apply(*erule-tac x=wset s* **in** *meta-allE*)
apply(*erule-tac x=*$\{\!|ta'|\!\}_w$ **in** *meta-allE*)
apply *clarsimp*
apply(*rule exI*)+
apply(*auto intro*!: *multithreaded-base.redT.intros*)[*1*]
apply *hypsubst*
apply(*rule exI conjI*)+
apply(*rule multithreaded-base.redT.redT-acquire*)
apply *assumption*+
done

lemma *JVM-legal-typesafe1*:
assumes *wfP*: *wf-jvm-prog P*
and *ok*: *jmm-wf-start-state P C M vs*
and *legal*: *legal-execution P* (*jmm-JVMd-*$\mathcal{E}$ *P C M vs status*) (*E*, *ws*)
shows *legal-execution P* (*jmm'-JVMd-*$\mathcal{E}$ *P C M vs status*) (*E*, *ws*)
proof −
let *?*$\mathcal{E}$ = *jmm-JVMd-*$\mathcal{E}$ *P C M vs status*
let *?*$\mathcal{E}'$ = *jmm'-JVMd-*$\mathcal{E}$ *P C M vs status*
from *legal* **obtain** *J*
where *justified*: *P* ⊢ (*E*, *ws*) *justified-by J*
and *range*: *range* (*justifying-exec* ∘ *J*) ⊆ *?*$\mathcal{E}$

and *E*: $E \in ?\mathcal{E}$ **and** *wf*: $P \vdash (E,\ ws)\ \surd$ **by**(*auto simp add*: *gen-legal-execution.simps*)
let $?J = J(0 := (\!|committed = \{\}, justifying\text{-}exec = justifying\text{-}exec\ (J\ 1), justifying\text{-}ws = justifying\text{-}ws\ (J\ 1),\ action\text{-}translation = id|\!))$

from *wfP* **obtain** Φ **where** Φ: *wf-jvm-prog*$_\Phi$ *P* **by**(*auto simp add*: *wf-jvm-prog-def*)
hence *wf-sys*: *wf-syscls P* **by**(*auto dest*: *wt-jvm-progD intro*: *wf-prog-wf-syscls*)

from *justified* **have** $P \vdash (justifying\text{-}exec\ (J\ 1),\ justifying\text{-}ws\ (J\ 1))\ \surd$ **by**(*simp add*: *justification-well-formed-def*)
with *justified* **have** $P \vdash (E,\ ws)$ *justified-by* $?J$ **by**(*rule drop-0th-justifying-exec*)
moreover have *range* $(justifying\text{-}exec \circ ?J) \subseteq ?\mathcal{E}'$
proof
fix ξ
assume $\xi \in range\ (justifying\text{-}exec \circ ?J)$
then obtain *n* **where** $\xi = justifying\text{-}exec\ (?J\ n)$ **by** *auto*
then obtain *n* **where** ξ: $\xi = justifying\text{-}exec\ (J\ n)$ **and** *n*: $n > 0$ **by**(*auto split*: *if-split-asm*)
from *range* ξ **have** $\xi \in ?\mathcal{E}$ **by** *auto*
thus $\xi \in ?\mathcal{E}'$ **unfolding** *jmm-typeof-addr'-conv-jmm-type-addr*[*symmetric*, *abs-def*]
proof(*rule JVMd-*$\mathcal{E}$*-heap-read-typedI*)
fix *ad al v T*
assume *read*: *NormalAction* (*ReadMem ad al v*) $\in$ *snd* ‘ *lset* ξ
and *adal*: $P \vdash jmm\ ad@al : T$
from *read* **obtain** *a* **where** *a*: *enat a* < *llength* ξ *action-obs* ξ *a* = *NormalAction* (*ReadMem ad al v*)
unfolding *lset-conv-lnth* **by**(*auto simp add*: *action-obs-def*)

have *ts-ok* (λt (*xcp*, *frs*) *h*. *JVM-heap-conf-base.correct-state addr2thread-id jmm-empty jmm-allocate* (λ-. *jmm-typeof-addr' P*) *jmm-hconf P* Φ *t* (*xcp*, *h*, *frs*)) (*thr* (*jmm-JVM-start-state P C M vs*)) *jmm.start-heap*

using *JVM-heap-conf.correct-jvm-state-initial*[*OF jmm-JVM-heap-conf* Φ *ok*]
by(*simp add*: *JVM-heap-conf-base.correct-jvm-state-def jmm-typeof-addr'-conv-jmm-typeof-addr heap-base.start-state-def split-beta*)
with *JVM-allocated-heap-conf'.mexecd-known-addrs-typing'*[*OF jmm-JVMd-allocated-heap-conf'* Φ *jmm-start-heap-ok*]
have $\exists T.\ P \vdash jmm\ ad@al : T \wedge P \vdash jmm\ v :\leq T$
using *wf-sys is-justified-by-imp-is-weakly-justified-by*[*OF justified wf*] *range n a*
unfolding *jmm-typeof-addr'-conv-jmm-type-addr*[*symmetric*, *abs-def*] ξ
by(*rule known-addrs-typing'.read-value-typeable-justifying*)
thus $P \vdash jmm\ v :\leq T$ **using** *adal*
by(*auto dest*: *jmm.addr-loc-type-fun*[*unfolded jmm-typeof-addr-conv-jmm-typeof-addr'*, *unfolded heap-base'.addr-loc-type-conv-addr-loc-type*])
qed
qed
moreover from *E* **have** $E \in ?\mathcal{E}'$
unfolding *jmm-typeof-addr'-conv-jmm-type-addr*[*symmetric*, *abs-def*]
proof(*rule JVMd-*$\mathcal{E}$*-heap-read-typedI*)
fix *ad al v T*
assume *read*: *NormalAction* (*ReadMem ad al v*) $\in$ *snd* ‘ *lset E*
and *adal*: $P \vdash jmm\ ad@al : T$
from *read* **obtain** *a* **where** *a*: *enat a* < *llength E action-obs E a* = *NormalAction* (*ReadMem ad al v*)
unfolding *lset-conv-lnth* **by**(*auto simp add*: *action-obs-def*)
with *jmm-JVMd-allocated-heap-conf'* Φ *ok legal-imp-weakly-legal-execution*[*OF legal*]

```
    have ∃ T. P ⊢jmm ad@al : T ∧ P ⊢jmm v :≤ T
      unfolding jmm-typeof-addr'-conv-jmm-typeof-addr[symmetric, abs-def]
      by(rule JVM-allocated-heap-conf'.JVM-weakly-legal-read-value-typeable)
    thus P ⊢jmm v :≤ T using adal
      by(auto dest: jmm.addr-loc-type-fun[unfolded jmm-typeof-addr-conv-jmm-typeof-addr', unfolded
heap-base'.addr-loc-type-conv-addr-loc-type])
  qed
  ultimately show ?thesis using wf unfolding gen-legal-execution.simps by blast
qed

lemma JVM-weakly-legal-typesafe1:
  assumes wfP: wf-jvm-prog P
  and ok: jmm-wf-start-state P C M vs
  and legal: weakly-legal-execution P (jmm-JVMd-ℰ P C M vs status) (E, ws)
  shows weakly-legal-execution P (jmm'-JVMd-ℰ P C M vs status) (E, ws)
proof −
  let ?ℰ = jmm-JVMd-ℰ P C M vs status
  let ?ℰ' = jmm'-JVMd-ℰ P C M vs status
  from legal obtain J
    where justified: P ⊢ (E, ws) weakly-justified-by J
    and range: range (justifying-exec ∘ J) ⊆ ?ℰ
    and E: E ∈ ?ℰ and wf: P ⊢ (E, ws) √ by(auto simp add: gen-legal-execution.simps)
  let ?J = J(0 := (|committed = {}, justifying-exec = justifying-exec (J 1), justifying-ws = justifying-ws
(J 1), action-translation = id|))

  from wfP obtain Φ where Φ: wf-jvm-prog_Φ P by(auto simp add: wf-jvm-prog-def)
  hence wf-sys: wf-syscls P by(auto dest: wt-jvm-progD intro: wf-prog-wf-syscls)

  from justified have P ⊢ (justifying-exec (J 1), justifying-ws (J 1)) √ by(simp add: justifica-
tion-well-formed-def)
  with justified have P ⊢ (E, ws) weakly-justified-by ?J by(rule drop-0th-weakly-justifying-exec)
  moreover have range (justifying-exec ∘ ?J) ⊆ ?ℰ'
  proof
    fix ξ
    assume ξ ∈ range (justifying-exec ∘ ?J)
    then obtain n where ξ = justifying-exec (?J n) by auto
    then obtain n where ξ: ξ = justifying-exec (J n) and n: n > 0 by(auto split: if-split-asm)
    from range ξ have ξ ∈ ?ℰ by auto
    thus ξ ∈ ?ℰ' unfolding jmm-typeof-addr'-conv-jmm-type-addr[symmetric, abs-def]
    proof(rule JVMd-ℰ-heap-read-typedI)
      fix ad al v T
      assume read: NormalAction (ReadMem ad al v) ∈ snd ' lset ξ
        and adal: P ⊢jmm ad@al : T
      from read obtain a where a: enat a < llength ξ action-obs ξ a = NormalAction (ReadMem ad
al v)
        unfolding lset-conv-lnth by(auto simp add: action-obs-def)

    have ts-ok (λt (xcp, frs) h. JVM-heap-conf-base.correct-state addr2thread-id jmm-empty jmm-allocate
(λ-. jmm-typeof-addr' P) jmm-hconf P Φ t (xcp, h, frs)) (thr (jmm-JVM-start-state P C M vs))
jmm.start-heap

      using JVM-heap-conf.correct-jvm-state-initial[OF jmm-JVM-heap-conf Φ ok]
      by(simp add: JVM-heap-conf-base.correct-jvm-state-def jmm-typeof-addr'-conv-jmm-typeof-addr
heap-base.start-state-def split-beta)
```

with *JVM-allocated-heap-conf'.mexecd-known-addrs-typing'*[*OF jmm-JVMd-allocated-heap-conf' Φ jmm-start-heap-ok*]
have ∃ *T*. *P* ⊢*jmm ad@al* : *T* ∧ *P* ⊢*jmm v* :≤ *T*
using *wf-sys justified range n a*
unfolding *jmm-typeof-addr'-conv-jmm-type-addr*[*symmetric*, *abs-def*] *ξ*
by(*rule known-addrs-typing'.read-value-typeable-justifying*)
thus *P* ⊢*jmm v* :≤ *T* **using** *adal*
by(*auto dest*: *jmm.addr-loc-type-fun*[*unfolded jmm-typeof-addr-conv-jmm-typeof-addr'*, *unfolded heap-base'.addr-loc-type-conv-addr-loc-type*])
qed
qed
moreover from *E* **have** *E* ∈ *?E'*
unfolding *jmm-typeof-addr'-conv-jmm-type-addr*[*symmetric*, *abs-def*]
proof(*rule JVMd-ℰ-heap-read-typedI*)
fix *ad al v T*
assume *read*: *NormalAction* (*ReadMem ad al v*) ∈ *snd* ' *lset E*
and *adal*: *P* ⊢*jmm ad@al* : *T*
from *read* **obtain** *a* **where** *a*: *enat a* < *llength E action-obs E a* = *NormalAction* (*ReadMem ad al v*)
unfolding *lset-conv-lnth* **by**(*auto simp add*: *action-obs-def*)
with *jmm-JVMd-allocated-heap-conf' Φ ok legal*
have ∃ *T*. *P* ⊢*jmm ad@al* : *T* ∧ *P* ⊢*jmm v* :≤ *T*
unfolding *jmm-typeof-addr'-conv-jmm-typeof-addr*[*symmetric*, *abs-def*]
by(*rule JVM-allocated-heap-conf'.JVM-weakly-legal-read-value-typeable*)
thus *P* ⊢*jmm v* :≤ *T* **using** *adal*
by(*auto dest*: *jmm.addr-loc-type-fun*[*unfolded jmm-typeof-addr-conv-jmm-typeof-addr'*, *unfolded heap-base'.addr-loc-type-conv-addr-loc-type*])
qed
ultimately show *?thesis* **using** *wf* **unfolding** *gen-legal-execution.simps* **by** *blast*
qed

lemma *JVMd-ℰ-heap-read-typedD*:
E ∈ *JVM-heap-base.JVMd-ℰ addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate* (λ-. *typeof-addr*) (*heap-base.heap-read-typed* (λ-. *typeof-addr*) *jmm-heap-read P*) *jmm-heap-write P C M vs status*
⟹ *E* ∈ *JVM-heap-base.JVMd-ℰ addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate* (λ-. *typeof-addr*) *jmm-heap-read jmm-heap-write P C M vs status*
apply(*erule imageE*, *hypsubst*)
apply(*rule imageI*)
apply(*erule multithreaded-base.ℰ.cases*, *hypsubst*)
apply(*rule multithreaded-base.ℰ.intros*)
apply(*subst* (*asm*) *if-mexecd-heap-read-typed*[*abs-def*])
apply(*erule if-mthr-Runs-heap-read-typedD*)
apply(*erule if-mExecd-heap-read-not-stuck*[**where** *typeof-addr*=λ-. *typeof-addr*, *unfolded if-mexecd-heap-read-typed*
done

lemma *JVMd-ℰ-typesafe-subset*: *jmm'-JVMd-ℰ P C M vs status* ⊆ *jmm-JVMd-ℰ P C M vs status*
unfolding *jmm-typeof-addr-def*[*abs-def*]
by(*rule subsetI*)(*erule JVMd-ℰ-heap-read-typedD*)

lemma *JVMd-legal-typesafe2*:
assumes *legal*: *legal-execution P* (*jmm'-JVMd-ℰ P C M vs status*) (*E*, *ws*)
shows *legal-execution P* (*jmm-JVMd-ℰ P C M vs status*) (*E*, *ws*)
proof −

let *?E = jmm-JVMd-E P C M vs status*
let *?E' = jmm'-JVMd-E P C M vs status*
from *legal* **obtain** *J*
where *justified*: $P \vdash (E, ws)$ *justified-by J*
and *range*: *range* (*justifying-exec* $\circ$ *J*) $\subseteq$ *?E'*
and *E*: $E \in$ *?E'* **and** *wf*: $P \vdash (E, ws)\ \surd$ **by**(*auto simp add*: *gen-legal-execution.simps*)
from *range E* **have** *range* (*justifying-exec* $\circ$ *J*) $\subseteq$ *?E E* $\in$ *?E*
using *JVMd-E-typesafe-subset*[*of P status C M vs*] **by** *blast*+
with *justified wf*
show *?thesis* **by**(*auto simp add*: *gen-legal-execution.simps*)
qed

theorem *JVMd-weakly-legal-typesafe2*:
assumes *legal*: *weakly-legal-execution P* (*jmm'-JVMd-E P C M vs status*) (*E*, *ws*)
shows *weakly-legal-execution P* (*jmm-JVMd-E P C M vs status*) (*E*, *ws*)
proof −
let *?E = jmm-JVMd-E P C M vs status*
let *?E' = jmm'-JVMd-E P C M vs status*
from *legal* **obtain** *J*
where *justified*: $P \vdash (E, ws)$ *weakly-justified-by J*
and *range*: *range* (*justifying-exec* $\circ$ *J*) $\subseteq$ *?E'*
and *E*: $E \in$ *?E'* **and** *wf*: $P \vdash (E, ws)\ \surd$ **by**(*auto simp add*: *gen-legal-execution.simps*)
from *range E* **have** *range* (*justifying-exec* $\circ$ *J*) $\subseteq$ *?E E* $\in$ *?E*
using *JVMd-E-typesafe-subset*[*of P status C M vs*] **by** *blast*+
with *justified wf*
show *?thesis* **by**(*auto simp add*: *gen-legal-execution.simps*)
qed

theorem *JVMd-weakly-legal-typesafe*:
assumes *wf-jvm-prog P*
and *jmm-wf-start-state P C M vs*
shows *weakly-legal-execution P* (*jmm-JVMd-E P C M vs status*) = *weakly-legal-execution P* (*jmm'-JVMd-E P C M vs status*)
apply(*rule ext iffI*)+
apply(*clarify*, *erule JVM-weakly-legal-typesafe1*[*OF assms*])
apply(*clarify*, *erule JVMd-weakly-legal-typesafe2*)
done

theorem *JVMd-legal-typesafe*:
assumes *wf-jvm-prog P*
and *jmm-wf-start-state P C M vs*
shows *legal-execution P* (*jmm-JVMd-E P C M vs status*) = *legal-execution P* (*jmm'-JVMd-E P C M vs status*)
apply(*rule ext iffI*)+
apply(*clarify*, *erule JVM-legal-typesafe1*[*OF assms*])
apply(*clarify*, *erule JVMd-legal-typesafe2*)
done

end

8.22 Compiler correctness for JMM heap implementation 2

theory *JMM-Compiler-Type2*

imports
JMM-Compiler
JMM-J-Typesafe
JMM-JVM-Typesafe
JMM-Interp
begin

theorem *J2JVM-jmm-correct*:
assumes *wf*: *wf-J-prog P*
and *wf-start*: *jmm-wf-start-state P C M vs*
shows *legal-execution P* (*jmm-J-*$\mathcal{E}$ *P C M vs Running*) (*E*, *ws*) $\longleftrightarrow$
legal-execution (*J2JVM P*) (*jmm-JVMd-*$\mathcal{E}$ (*J2JVM P*) *C M vs Running*) (*E*, *ws*)
using *JVMd-legal-typesafe*[*OF wt-J2JVM*[*OF wf*], *of C M vs Running*, *symmetric*] *wf-start*
by(*simp only*: *J-legal-typesafe*[*OF assms*] *J-JVM-conf-read.red-*$\mathcal{E}$*-eq-mexecd-*$\mathcal{E}$[*OF jmm'-J-JVM-conf-read assms*] *J2JVM-def o-apply compP1-def compP2-def legal-execution-compP heap-base.wf-start-state-compP jmm-typeof-addr-compP heap-base.heap-read-typed-compP*)

theorem *J2JVM-jmm-correct-weak*:
assumes *wf*: *wf-J-prog P*
and *wf-start*: *jmm-wf-start-state P C M vs*
shows *weakly-legal-execution P* (*jmm-J-*$\mathcal{E}$ *P C M vs Running*) (*E*, *ws*) $\longleftrightarrow$
weakly-legal-execution (*J2JVM P*) (*jmm-JVMd-*$\mathcal{E}$ (*J2JVM P*) *C M vs Running*) (*E*, *ws*)
using *JVMd-weakly-legal-typesafe*[*OF wt-J2JVM*[*OF wf*], *of C M vs Running*, *symmetric*] *wf-start*
by(*simp only*: *J-weakly-legal-typesafe*[*OF assms*] *J-JVM-conf-read.red-*$\mathcal{E}$*-eq-mexecd-*$\mathcal{E}$[*OF jmm'-J-JVM-conf-read assms*] *J2JVM-def o-apply compP1-def compP2-def weakly-legal-execution-compP heap-base.wf-start-state-compP jmm-typeof-addr-compP heap-base.heap-read-typed-compP*)

theorem *J2JVM-jmm-correctly-synchronized*:
assumes *wf*: *wf-J-prog P*
and *wf-start*: *jmm-wf-start-state P C M vs*
and *ka*: $\bigcup$(*ka-Val* ' *set vs*) $\subseteq$ *set jmm.start-addrs*
shows *correctly-synchronized* (*J2JVM P*) (*jmm-JVMd-*$\mathcal{E}$ (*J2JVM P*) *C M vs Running*) $\longleftrightarrow$
correctly-synchronized P (*jmm-J-*$\mathcal{E}$ *P C M vs Running*)
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?lhs*
show *?rhs* **unfolding** *correctly-synchronized-def*
proof(*intro strip*)
fix *E ws a a'*
assume *E*: *E* $\in$ *jmm-J-*$\mathcal{E}$ *P C M vs Running*
and *wf-exec*: *P* $\vdash$ (*E*, *ws*) $\surd$
and *sc*: *sequentially-consistent P* (*E*, *ws*)
and *actions*: *a* $\in$ *actions E a'* $\in$ *actions E*
and *conflict*: *P*,*E* $\vdash$ *a* $\dagger$ *a'*

from *E wf-exec sc*
have *legal-execution P* (*jmm-J-*$\mathcal{E}$ *P C M vs Running*) (*E*, *ws*)
by(*rule sc-legal.SC-is-legal*[*OF J-allocated-progress.J-sc-legal*[*OF jmm-J-allocated-progress wf jmm-heap-read-typeable wf-start ka*]])
hence *legal-execution* (*J2JVM P*) (*jmm-JVMd-*$\mathcal{E}$ (*J2JVM P*) *C M vs Running*) (*E*, *ws*)
by(*simp only*: *J2JVM-jmm-correct*[*OF wf wf-start*])
hence *E* $\in$ *jmm-JVMd-*$\mathcal{E}$ (*J2JVM P*) *C M vs Running J2JVM P* $\vdash$ (*E*, *ws*) $\surd$
by(*simp-all add*: *gen-legal-execution.simps*)
moreover from *sc* **have** *sequentially-consistent* (*J2JVM P*) (*E*, *ws*)

```
    by(simp add: J2JVM-def compP2-def)
  moreover from conflict have J2JVM P,E ⊢ a † a'
    by(simp add: J2JVM-def compP2-def)
  ultimately have J2JVM P,E ⊢ a ≤hb a' ∨ J2JVM P,E ⊢ a' ≤hb a
    using ‹?lhs› actions by(auto simp add: correctly-synchronized-def)
  thus P,E ⊢ a ≤hb a' ∨ P,E ⊢ a' ≤hb a
    by(simp add: J2JVM-def compP2-def)
 qed
next
 assume ?rhs
 show ?lhs unfolding correctly-synchronized-def
 proof(intro strip)
  fix E ws a a'
  assume E: E ∈ jmm-JVMd-ℰ (J2JVM P) C M vs Running
    and wf-exec: J2JVM P ⊢ (E, ws) √
    and sc: sequentially-consistent (J2JVM P) (E, ws)
    and actions: a ∈ actions E a' ∈ actions E
    and conflict: J2JVM P,E ⊢ a † a'

  from wf have wf-jvm-prog (J2JVM P) by(rule wt-J2JVM)
  then obtain Φ where wf': wf-jvm-prog_Φ (J2JVM P)
    by(auto simp add: wf-jvm-prog-def)
  from wf-start have wf-start': jmm-wf-start-state (J2JVM P) C M vs
    by(simp add: J2JVM-def compP2-def heap-base.wf-start-state-compP)
  from E wf-exec sc
  have legal-execution (J2JVM P) (jmm-JVMd-ℰ (J2JVM P) C M vs Running) (E, ws)
  by(rule sc-legal.SC-is-legal[OF JVM-allocated-progress.JVM-sc-legal[OF jmm-JVM-allocated-progress
wf' jmm-heap-read-typeable wf-start' ka]])
  hence legal-execution P (jmm-J-ℰ P C M vs Running) (E, ws)
    by(simp only: J2JVM-jmm-correct[OF wf wf-start])
  hence E ∈ jmm-J-ℰ P C M vs Running P ⊢ (E, ws) √
    by(simp-all add: gen-legal-execution.simps)
  moreover from sc have sequentially-consistent P (E, ws)
    by(simp add: J2JVM-def compP2-def)
  moreover from conflict have P,E ⊢ a † a'
    by(simp add: J2JVM-def compP2-def)
  ultimately have P,E ⊢ a ≤hb a' ∨ P,E ⊢ a' ≤hb a
    using ‹?rhs› actions by(auto simp add: correctly-synchronized-def)
  thus J2JVM P,E ⊢ a ≤hb a' ∨ J2JVM P,E ⊢ a' ≤hb a
    by(simp add: J2JVM-def compP2-def)
 qed
qed

end


theory JMM
imports
 JMM-DRF
 SC-Legal
 DRF-J
 DRF-JVM
 JMM-Type
 JMM-Interp
```

JMM-Typesafe
JMM-J-Typesafe
JMM-JVM-Typesafe
JMM-Compiler-Type2
begin

end
theory *MM-Main*
imports
SC
SC-Interp
SC-Collections
JMM
begin

end

Chapter 9

Schedulers

9.1 Refinement for multithreaded states

theory *State-Refinement*
imports
../Framework/FWSemantics
../Common/StartConfig
begin

type-synonym
(*'l*,*'t*,*'m*,*'m-t*,*'m-w*,*'s-i*) *state-refine* = (*'l*,*'t*) *locks* × (*'m-t* × *'m*) × *'m-w* × *'s-i*

locale *state-refine-base* =
fixes *final* :: *'x* ⇒ *bool*
and *r* :: *'t* ⇒ (*'x* × *'m*) ⇒ ((*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* × *'x* × *'m*) *Predicate.pred*
and *convert-RA* :: *'l released-locks* ⇒ *'o list*
and *thr-α* :: *'m-t* ⇒ (*'l*,*'t*,*'x*) *thread-info*
and *thr-invar* :: *'m-t* ⇒ *bool*
and *ws-α* :: *'m-w* ⇒ (*'w*,*'t*) *wait-sets*
and *ws-invar* :: *'m-w* ⇒ *bool*
and *is-α* :: *'s-i* ⇒ *'t interrupts*
and *is-invar* :: *'s-i* ⇒ *bool*
begin

fun *state-α* :: (*'l*,*'t*,*'m*,*'m-t*,*'m-w*, *'s-i*) *state-refine* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*) *state*
where *state-α* (*ls*, (*ts*, *m*), *ws*, *is*) = (*ls*, (*thr-α ts*, *m*), *ws-α ws*, *is-α is*)

lemma *state-α-conv* [*simp*]:
locks (*state-α s*) = *locks s*
thr (*state-α s*) = *thr-α* (*thr s*)
shr (*state-α s*) = *shr s*
wset (*state-α s*) = *ws-α* (*wset s*)
interrupts (*state-α s*) = *is-α* (*interrupts s*)
by(*case-tac* [!] *s*) *auto*

inductive *state-invar* :: (*'l*,*'t*,*'m*,*'m-t*,*'m-w*,*'s-i*) *state-refine* ⇒ *bool*
where ⟦ *thr-invar ts*; *ws-invar ws*; *is-invar is* ⟧ ⟹ *state-invar* (*ls*, (*ts*, *m*), *ws*, *is*)

inductive-simps *state-invar-simps* [*simp*]:
state-invar (*ls*, (*ts*, *m*), *ws*, *is*)

lemma *state-invarD* [*simp*]:
assumes *state-invar s*
shows *thr-invar* (*thr s*) *ws-invar* (*wset s*) *is-invar* (*interrupts s*)
using *assms* **by**(*case-tac* [!] *s*) *auto*

end

sublocale *state-refine-base* < α: *final-thread final* .
sublocale *state-refine-base* < α:
multithreaded-base
final
λ*t xm ta x'm'*. *Predicate.eval* (*r t xm*) (*ta*, *x'm'*)
.

definition (**in** *heap-base*) *start-state-refine* ::
'm-t ⇒ (*'thread-id* ⇒ (*'x* × *'addr released-locks*) ⇒ *'m-t* ⇒ *'m-t*) ⇒ *'m-w* ⇒ *'s-i*
⇒ (*cname* ⇒ *mname* ⇒ *ty list* ⇒ *ty* ⇒ *'md* ⇒ *'addr val list* ⇒ *'x*) ⇒ *'md prog* ⇒ *cname* ⇒ *mname* ⇒ *'addr val list*
⇒ (*'addr*, *'thread-id*, *'heap*, *'m-t*, *'m-w*, *'s-i*) *state-refine*
where
⋀*is-empty*.
start-state-refine thr-empty thr-update ws-empty is-empty f P C M vs =
(*let* (*D*, *Ts*, *T*, *m*) = *method P C M*
in (*K*$ *None*, (*thr-update start-tid* (*f D M Ts T* (*the m*) *vs*, *no-wait-locks*) *thr-empty*, *start-heap*), *ws-empty*, *is-empty*))

definition *Jinja-output* ::
's ⇒ *'thread-id* ⇒ (*'addr*, *'thread-id*, *'x*, *'heap*, *'addr*, (*'addr*, *'thread-id*) *obs-event*) *thread-action*
⇒ (*'thread-id* × (*'addr*, *'thread-id*) *obs-event list*) *option*
where *Jinja-output* σ *t ta* = (*if* {|*ta*|}$_o$ = [] *then None else Some* (*t*, {|*ta*|}$_o$))

lemmas [*code*] =
heap-base.start-state-refine-def

end

9.2 Abstract scheduler

theory *Scheduler*
imports
State-Refinement
../Framework/FWProgressAux
../Framework/FWLTS

../Basic/JT-ICF

begin

Define an unfold operation that puts everything into one function to avoid duplicate evaluation.

definition *unfold-tllist'* :: (*'a* ⇒ *'b* × *'a* + *'c*) ⇒ *'a* ⇒ (*'b*, *'c*) *tllist*
where [*code del*]:

unfold-tllist′ f =
unfold-tllist (λ*a*. ∃ *c*. *f a* = *Inr c*) (*projr* ∘ *f*) (*fst* ∘ *projl* ∘ *f*) (*snd* ∘ *projl* ∘ *f*)

lemma *unfold-tllist′* [*code*]:
unfold-tllist′ f a =
(*case f a of Inr c* ⇒ *TNil c* | *Inl* (*b*, *a′*) ⇒ *TCons b* (*unfold-tllist′ f a′*))
by(*rule tllist.expand*)(*auto simp add*: *unfold-tllist′-def split*: *sum.split-asm*)

type-synonym
('*l*,'*t*,'*x*,'*m*,'*w*,'*o*,'*m-t*,'*m-w*,'*s-i*,'*s*) *scheduler* =
'*s* ⇒ ('*l*,'*t*,'*m*,'*m-t*,'*m-w*,'*s-i*) *state-refine* ⇒ ('*t* × (('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* × '*x* × '*m*) *option* × '*s*) *option*

locale *scheduler-spec-base* =
state-refine-base
final r convert-RA
thr-α thr-invar
ws-α ws-invar
is-α is-invar
for *final* :: '*x* ⇒ *bool*
and *r* :: '*t* ⇒ ('*x* × '*m*) ⇒ (('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* × '*x* × '*m*) *Predicate.pred*
and *convert-RA* :: '*l released-locks* ⇒ '*o list*
and *schedule* :: ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*,'*m-t*,'*m-w*,'*s-i*,'*s*) *scheduler*
and *σ-invar* :: '*s* ⇒ '*t set* ⇒ *bool*
and *thr-α* :: '*m-t* ⇒ ('*l*,'*t*,'*x*) *thread-info*
and *thr-invar* :: '*m-t* ⇒ *bool*
and *ws-α* :: '*m-w* ⇒ ('*w*,'*t*) *wait-sets*
and *ws-invar* :: '*m-w* ⇒ *bool*
and *is-α* :: '*s-i* ⇒ '*t interrupts*
and *is-invar* :: '*s-i* ⇒ *bool*

locale *scheduler-spec* =
scheduler-spec-base
final r convert-RA
schedule σ-invar
thr-α thr-invar
ws-α ws-invar
is-α is-invar
for *final* :: '*x* ⇒ *bool*
and *r* :: '*t* ⇒ ('*x* × '*m*) ⇒ (('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* × '*x* × '*m*) *Predicate.pred*
and *convert-RA* :: '*l released-locks* ⇒ '*o list*
and *schedule* :: ('*l*,'*t*,'*x*,'*m*,'*w*,'*o*,'*m-t*,'*m-w*,'*s-i*,'*s*) *scheduler*
and *σ-invar* :: '*s* ⇒ '*t set* ⇒ *bool*
and *thr-α* :: '*m-t* ⇒ ('*l*,'*t*,'*x*) *thread-info*
and *thr-invar* :: '*m-t* ⇒ *bool*
and *ws-α* :: '*m-w* ⇒ ('*w*,'*t*) *wait-sets*
and *ws-invar* :: '*m-w* ⇒ *bool*
and *is-α* :: '*s-i* ⇒ '*t interrupts*
and *is-invar* :: '*s-i* ⇒ *bool*
+
fixes *invariant* :: ('*l*,'*t*,'*x*,'*m*,'*w*) *state set*
assumes *schedule-NoneD*:
⟦ *schedule σ s* = *None*; *state-invar s*; *σ-invar σ* (*dom* (*thr-α* (*thr s*))); *state-α s* ∈ *invariant* ⟧

$\Longrightarrow \alpha.active\text{-}threads\ (state\text{-}\alpha\ s) = \{\}$
and *schedule-Some-NoneD*:
$[\![\ schedule\ \sigma\ s = \lfloor(t,\ None,\ \sigma')\rfloor;\ state\text{-}invar\ s;\ \sigma\text{-}invar\ \sigma\ (dom\ (thr\text{-}\alpha\ (thr\ s)));\ state\text{-}\alpha\ s \in invariant\ ]\!]$
$\Longrightarrow \exists x\ ln\ n.\ thr\text{-}\alpha\ (thr\ s)\ t = \lfloor(x,\ ln)\rfloor \wedge ln\ \$\ n > 0 \wedge \neg\ waiting\ (ws\text{-}\alpha\ (wset\ s)\ t) \wedge may\text{-}acquire\text{-}all\ (locks\ s)\ t\ ln$
and *schedule-Some-SomeD*:
$[\![\ schedule\ \sigma\ s = \lfloor(t,\ \lfloor(ta,\ x',\ m')\rfloor,\ \sigma')\rfloor;\ state\text{-}invar\ s;\ \sigma\text{-}invar\ \sigma\ (dom\ (thr\text{-}\alpha\ (thr\ s)));\ state\text{-}\alpha\ s \in invariant\ ]\!]$
$\Longrightarrow \exists x.\ thr\text{-}\alpha\ (thr\ s)\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor \wedge Predicate.eval\ (r\ t\ (x,\ shr\ s))\ (ta,\ x',\ m') \wedge$
$\alpha.actions\text{-}ok\ (state\text{-}\alpha\ s)\ t\ ta$
and *schedule-invar-None*:
$[\![\ schedule\ \sigma\ s = \lfloor(t,\ None,\ \sigma')\rfloor;\ state\text{-}invar\ s;\ \sigma\text{-}invar\ \sigma\ (dom\ (thr\text{-}\alpha\ (thr\ s)));\ state\text{-}\alpha\ s \in invariant\ ]\!]$
$\Longrightarrow \sigma\text{-}invar\ \sigma'\ (dom\ (thr\text{-}\alpha\ (thr\ s)))$
and *schedule-invar-Some*:
$[\![\ schedule\ \sigma\ s = \lfloor(t,\ \lfloor(ta,\ x',\ m')\rfloor,\ \sigma')\rfloor;\ state\text{-}invar\ s;\ \sigma\text{-}invar\ \sigma\ (dom\ (thr\text{-}\alpha\ (thr\ s)));\ state\text{-}\alpha\ s \in invariant\ ]\!]$
$\Longrightarrow \sigma\text{-}invar\ \sigma'\ (dom\ (thr\text{-}\alpha\ (thr\ s)) \cup \{t.\ \exists x\ m.\ NewThread\ t\ x\ m \in set\ \{\!|ta|\!\}_t\})$

locale *pick-wakeup-spec-base* =
state-refine-base
final r convert-RA
thr-α thr-invar
ws-α ws-invar
is-α is-invar
for *final* :: $'x \Rightarrow bool$
and $r :: 't \Rightarrow ('x \times 'm) \Rightarrow (('l,'t,'x,'m,'w,'o)\ thread\text{-}action \times 'x \times 'm)\ Predicate.pred$
and *convert-RA* :: $'l\ released\text{-}locks \Rightarrow 'o\ list$
and *pick-wakeup* :: $'s \Rightarrow 't \Rightarrow 'w \Rightarrow 'm\text{-}w \Rightarrow 't\ option$
and $\sigma\text{-}invar :: 's \Rightarrow 't\ set \Rightarrow bool$
and $thr\text{-}\alpha :: 'm\text{-}t \Rightarrow ('l,'t,'x)\ thread\text{-}info$
and *thr-invar* :: $'m\text{-}t \Rightarrow bool$
and $ws\text{-}\alpha :: 'm\text{-}w \Rightarrow ('w,'t)\ wait\text{-}sets$
and *ws-invar* :: $'m\text{-}w \Rightarrow bool$
and $is\text{-}\alpha :: 's\text{-}i \Rightarrow 't\ interrupts$
and *is-invar* :: $'s\text{-}i \Rightarrow bool$

locale *pick-wakeup-spec* =
pick-wakeup-spec-base
final r convert-RA
pick-wakeup σ-invar
thr-α thr-invar
ws-α ws-invar
is-α is-invar
for *final* :: $'x \Rightarrow bool$
and $r :: 't \Rightarrow ('x \times 'm) \Rightarrow (('l,'t,'x,'m,'w,'o)\ thread\text{-}action \times 'x \times 'm)\ Predicate.pred$
and *convert-RA* :: $'l\ released\text{-}locks \Rightarrow 'o\ list$
and *pick-wakeup* :: $'s \Rightarrow 't \Rightarrow 'w \Rightarrow 'm\text{-}w \Rightarrow 't\ option$
and $\sigma\text{-}invar :: 's \Rightarrow 't\ set \Rightarrow bool$
and $thr\text{-}\alpha :: 'm\text{-}t \Rightarrow ('l,'t,'x)\ thread\text{-}info$
and *thr-invar* :: $'m\text{-}t \Rightarrow bool$
and $ws\text{-}\alpha :: 'm\text{-}w \Rightarrow ('w,'t)\ wait\text{-}sets$
and *ws-invar* :: $'m\text{-}w \Rightarrow bool$

and *is-α* :: *'s-i* ⇒ *'t interrupts*
and *is-invar* :: *'s-i* ⇒ *bool*
+
assumes *pick-wakeup-NoneD*:
⟦ *pick-wakeup σ t w ws* = *None*; *ws-invar ws*; *σ-invar σ T*; *dom* (*ws-α ws*) ⊆ *T*; *t* ∈ *T* ⟧
⟹ *InWS w* ∉ *ran* (*ws-α ws*)
and *pick-wakeup-SomeD*:
⟦ *pick-wakeup σ t w ws* = ⌊*t'*⌋; *ws-invar ws*; *σ-invar σ T*; *dom* (*ws-α ws*) ⊆ *T*; *t* ∈ *T* ⟧
⟹ *ws-α ws t'* = ⌊*InWS w*⌋

locale *scheduler-base-aux* =
state-refine-base
final r convert-RA
thr-α thr-invar
ws-α ws-invar
is-α is-invar
for *final* :: *'x* ⇒ *bool*
and *r* :: *'t* ⇒ (*'x* × *'m*) ⇒ ((*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* × *'x* × *'m*) *Predicate.pred*
and *convert-RA* :: *'l released-locks* ⇒ *'o list*
and *thr-α* :: *'m-t* ⇒ (*'l*,*'t*,*'x*) *thread-info*
and *thr-invar* :: *'m-t* ⇒ *bool*
and *thr-lookup* :: *'t* ⇒ *'m-t* ⇀ (*'x* × *'l released-locks*)
and *thr-update* :: *'t* ⇒ *'x* × *'l released-locks* ⇒ *'m-t* ⇒ *'m-t*
and *ws-α* :: *'m-w* ⇒ (*'w*,*'t*) *wait-sets*
and *ws-invar* :: *'m-w* ⇒ *bool*
and *ws-lookup* :: *'t* ⇒ *'m-w* ⇀ *'w wait-set-status*
and *is-α* :: *'s-i* ⇒ *'t interrupts*
and *is-invar* :: *'s-i* ⇒ *bool*
and *is-memb* :: *'t* ⇒ *'s-i* ⇒ *bool*
and *is-ins* :: *'t* ⇒ *'s-i* ⇒ *'s-i*
and *is-delete* :: *'t* ⇒ *'s-i* ⇒ *'s-i*
begin

definition *free-thread-id* :: *'m-t* ⇒ *'t* ⇒ *bool*
where *free-thread-id ts t* ⟷ *thr-lookup t ts* = *None*

fun *redT-updT* :: *'m-t* ⇒ (*'t*,*'x*,*'m*) *new-thread-action* ⇒ *'m-t*
where
redT-updT ts (*NewThread t' x m*) = *thr-update t'* (*x*, *no-wait-locks*) *ts*
| *redT-updT ts* - = *ts*

definition *redT-updTs* :: *'m-t* ⇒ (*'t*,*'x*,*'m*) *new-thread-action list* ⇒ *'m-t*
where *redT-updTs* = *foldl redT-updT*

primrec *thread-ok* :: *'m-t* ⇒ (*'t*,*'x*,*'m*) *new-thread-action* ⇒ *bool*
where
thread-ok ts (*NewThread t x m*) = *free-thread-id ts t*
| *thread-ok ts* (*ThreadExists t b*) = (*b* ≠ *free-thread-id ts t*)

We use *local.redT-updT* in *thread-ok* instead of *redT-updT'* like in theory *JinjaThreads.FWThread*. This fixes *'x* in the (*'t*, *'x*, *'m*) *new-thread-action list* type, but avoids *undefined*, which raises an exception during execution in the generated code.

primrec *thread-oks* :: *'m-t* ⇒ (*'t*,*'x*,*'m*) *new-thread-action list* ⇒ *bool*
where

```
  thread-oks ts [] = True
| thread-oks ts (ta#tas) = (thread-ok ts ta ∧ thread-oks (redT-updT ts ta) tas)

definition wset-actions-ok :: 'm-w ⇒ 't ⇒ ('t,'w) wait-set-action list ⇒ bool
where
  wset-actions-ok ws t was ⟷
   ws-lookup t ws =
   (if Notified ∈ set was then ⌊PostWS WSNotified⌋
    else if WokenUp ∈ set was then ⌊PostWS WSWokenUp⌋
    else None)

primrec cond-action-ok :: ('l,'t,'m,'m-t,'m-w,'s-i) state-refine ⇒ 't ⇒ 't conditional-action ⇒ bool
where
  ⋀ln. cond-action-ok s t (Join T) =
   (case thr-lookup T (thr s)
      of None ⇒ True
    | ⌊(x, ln)⌋ ⇒ t ≠ T ∧ final x ∧ ln = no-wait-locks ∧ ws-lookup T (wset s) = None)
| cond-action-ok s t Yield = True

definition cond-action-oks :: ('l,'t,'m,'m-t,'m-w,'s-i) state-refine ⇒ 't ⇒ 't conditional-action list ⇒
bool
where
  cond-action-oks s t cts = list-all (cond-action-ok s t) cts

primrec redT-updI :: 's-i ⇒ 't interrupt-action ⇒ 's-i
where
  redT-updI is (Interrupt t) = is-ins t is
| redT-updI is (ClearInterrupt t) = is-delete t is
| redT-updI is (IsInterrupted t b) = is

primrec redT-updIs :: 's-i ⇒ 't interrupt-action list ⇒ 's-i
where
  redT-updIs is [] = is
| redT-updIs is (ia # ias) = redT-updIs (redT-updI is ia) ias

primrec interrupt-action-ok :: 's-i ⇒ 't interrupt-action ⇒ bool
where
  interrupt-action-ok is (Interrupt t) = True
| interrupt-action-ok is (ClearInterrupt t) = True
| interrupt-action-ok is (IsInterrupted t b) = (b = (is-memb t is))

primrec interrupt-actions-ok :: 's-i ⇒ 't interrupt-action list ⇒ bool
where
  interrupt-actions-ok is [] = True
| interrupt-actions-ok is (ia # ias) ⟷ interrupt-action-ok is ia ∧ interrupt-actions-ok (redT-updI is
ia) ias

definition actions-ok :: ('l,'t,'m,'m-t,'m-w,'s-i) state-refine ⇒ 't ⇒ ('l,'t,'x,'m,'w,'o') thread-action
⇒ bool
where
  actions-ok s t ta ⟷
   lock-ok-las (locks s) t {|ta|}_l ∧
   thread-oks (thr s) {|ta|}_t ∧
   cond-action-oks s t {|ta|}_c ∧
```

```
  wset-actions-ok (wset s) t {|ta|}_w ∧
  interrupt-actions-ok (interrupts s) {|ta|}_i

end

locale scheduler-base =
  scheduler-base-aux
    final r convert-RA
    thr-α thr-invar thr-lookup thr-update
    ws-α ws-invar ws-lookup
    is-α is-invar is-memb is-ins is-delete
  +
  scheduler-spec-base
    final r convert-RA
    schedule σ-invar
    thr-α thr-invar
    ws-α ws-invar
    is-α is-invar
  +
  pick-wakeup-spec-base
    final r convert-RA
    pick-wakeup σ-invar
    thr-α thr-invar
    ws-α ws-invar
    is-α is-invar
  for final :: 'x ⇒ bool
  and r :: 't ⇒ ('x × 'm) ⇒ (('l,'t,'x,'m,'w,'o) thread-action × 'x × 'm) Predicate.pred
  and convert-RA :: 'l released-locks ⇒ 'o list
  and schedule :: ('l,'t,'x,'m,'w,'o,'m-t,'m-w,'s-i,'s) scheduler
  and output :: 's ⇒ 't ⇒ ('l,'t,'x,'m,'w,'o) thread-action ⇒ 'q option
  and pick-wakeup :: 's ⇒ 't ⇒ 'w ⇒ 'm-w ⇒ 't option
  and σ-invar :: 's ⇒ 't set ⇒ bool
  and thr-α :: 'm-t ⇒ ('l,'t,'x) thread-info
  and thr-invar :: 'm-t ⇒ bool
  and thr-lookup :: 't ⇒ 'm-t ⇀ ('x × 'l released-locks)
  and thr-update :: 't ⇒ 'x × 'l released-locks ⇒ 'm-t ⇒ 'm-t
  and ws-α :: 'm-w ⇒ ('w,'t) wait-sets
  and ws-invar :: 'm-w ⇒ bool
  and ws-lookup :: 't ⇒ 'm-w ⇀ 'w wait-set-status
  and ws-update :: 't ⇒ 'w wait-set-status ⇒ 'm-w ⇒ 'm-w
  and ws-delete :: 't ⇒ 'm-w ⇒ 'm-w
  and ws-iterate :: 'm-w ⇒ ('t × 'w wait-set-status, 'm-w) set-iterator
  and is-α :: 's-i ⇒ 't interrupts
  and is-invar :: 's-i ⇒ bool
  and is-memb :: 't ⇒ 's-i ⇒ bool
  and is-ins :: 't ⇒ 's-i ⇒ 's-i
  and is-delete :: 't ⇒ 's-i ⇒ 's-i
begin

primrec exec-updW :: 's ⇒ 't ⇒ 'm-w ⇒ ('t,'w) wait-set-action ⇒ 'm-w
where
  exec-updW σ t ws (Notify w) =
   (case pick-wakeup σ t w ws
    of None ⇒ ws
```

```
    | Some t ⇒ ws-update t (PostWS WSNotified) ws)
| exec-updW σ t ws (NotifyAll w) =
   ws-iterate ws (λ-. True) (λ(t, w') ws'. if w' = InWS w then ws-update t (PostWS WSNotified) ws'
else ws')
            ws
| exec-updW σ t ws (Suspend w) = ws-update t (InWS w) ws
| exec-updW σ t ws (WakeUp t') =
   (case ws-lookup t' ws of ⌊InWS w⌋ ⇒ ws-update t' (PostWS WSWokenUp) ws | - ⇒ ws)
| exec-updW σ t ws Notified = ws-delete t ws
| exec-updW σ t ws WokenUp = ws-delete t ws

definition exec-updWs :: 's ⇒ 't ⇒ 'm-w ⇒ ('t,'w) wait-set-action list ⇒ 'm-w
where exec-updWs σ t = foldl (exec-updW σ t)

definition exec-upd ::
  's ⇒ ('l,'t,'m,'m-t,'m-w,'s-i) state-refine ⇒ 't ⇒ ('l,'t,'x,'m,'w,'o) thread-action ⇒ 'x ⇒ 'm
  ⇒ ('l,'t,'m,'m-t,'m-w,'s-i) state-refine
where [simp]:
  exec-upd σ s t ta x' m' =
   (redT-updLs (locks s) t {|ta|}_l,
    (thr-update t (x', redT-updLns (locks s) t (snd (the (thr-lookup t (thr s)))) {|ta|}_l) (redT-updTs (thr
s) {|ta|}_t), m'),
     exec-updWs σ t (wset s) {|ta|}_w, redT-updIs (interrupts s) {|ta|}_i)

definition execT ::
  's ⇒ ('l,'t,'m,'m-t,'m-w,'s-i) state-refine
  ⇒ ('s × 't × ('l,'t,'x,'m,'w,'o) thread-action × ('l,'t,'m,'m-t,'m-w,'s-i) state-refine) option
where
  execT σ s =
  (do {
     (t, tax'm', σ') ← schedule σ s;
     case tax'm' of
       None ⇒
       (let (x, ln) = the (thr-lookup t (thr s));
            ta = (K$ [], [], [], [], [], convert-RA ln);
          s' = (acquire-all (locks s) t ln, (thr-update t (x, no-wait-locks) (thr s), shr s), wset s, interrupts
s)
        in ⌊(σ', t, ta, s')⌋)
     | ⌊(ta, x', m')⌋ ⇒ ⌊(σ', t, ta, exec-upd σ s t ta x' m')⌋
   })

primrec exec-step ::
  's × ('l,'t,'m,'m-t,'m-w,'s-i) state-refine ⇒
  ('s × 't × ('l,'t,'x,'m,'w,'o) thread-action) × 's × ('l,'t,'m,'m-t,'m-w,'s-i) state-refine + ('l,'t,'m,'m-t,'m-w,'s-i)
state-refine
where
  exec-step (σ, s) =
   (case execT σ s of
      None ⇒ Inr s
    | Some (σ', t, ta, s') ⇒ Inl ((σ, t, ta), σ', s'))

declare exec-step.simps [simp del]

definition exec-aux ::
```

's × (*'l*,*'t*,*'m*,*'m-t*,*'m-w*,*'s-i*) *state-refine*
⇒ (*'s* × *'t* × (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action*, (*'l*,*'t*,*'m*,*'m-t*,*'m-w*,*'s-i*) *state-refine*) *tllist*
where
exec-aux σs = *unfold-tllist' exec-step σs*

definition *exec* :: *'s* ⇒ (*'l*,*'t*,*'m*,*'m-t*,*'m-w*,*'s-i*) *state-refine* ⇒ (*'q*, (*'l*,*'t*,*'m*,*'m-t*,*'m-w*,*'s-i*) *state-refine*) *tllist*
where
exec σ s = *tmap the id* (*tfilter undefined* (λ*q*. *q* ≠ *None*) (*tmap* (λ(*σ*, *t*, *ta*). *output σ t ta*) *id* (*exec-aux* (*σ*, *s*))))

end

Implement *pick-wakeup* by *map-sel'*

definition *pick-wakeup-via-sel* ::
(*'m-w* ⇒ (*'t* ⇒ *'w wait-set-status* ⇒ *bool*) ⇀ *'t* × *'w wait-set-status*)
⇒ *'s* ⇒ *'t* ⇒ *'w* ⇒ *'m-w* ⇒ *'t option*
where *pick-wakeup-via-sel ws-sel σ t w ws* = *map-option fst* (*ws-sel ws* (λ*t w'*. *w'* = *InWS w*))

lemma *pick-wakeup-spec-via-sel*:
assumes *sel*: *map-sel' ws-α ws-invar ws-sel*
shows *pick-wakeup-spec* (*pick-wakeup-via-sel* (λ*s P*. *ws-sel s* (λ(*k*,*v*). *P k v*))) *σ-invar ws-α ws-invar*
proof −
interpret *ws*: *map-sel' ws-α ws-invar ws-sel* **by**(*rule sel*)
show *?thesis*
by(*unfold-locales*)(*auto simp add*: *pick-wakeup-via-sel-def ran-def dest*: *ws.sel'-noneD ws.sel'-SomeD*)
qed

locale *scheduler-ext-base* =
scheduler-base-aux
final r convert-RA
thr-α thr-invar thr-lookup thr-update
ws-α ws-invar ws-lookup
is-α is-invar is-memb is-ins is-delete
for *final* :: *'x* ⇒ *bool*
and *r* :: *'t* ⇒ (*'x* × *'m*) ⇒ ((*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* × *'x* × *'m*) *Predicate.pred*
and *convert-RA* :: *'l released-locks* ⇒ *'o list*
and *thr-α* :: *'m-t* ⇒ (*'l*,*'t*,*'x*) *thread-info*
and *thr-invar* :: *'m-t* ⇒ *bool*
and *thr-lookup* :: *'t* ⇒ *'m-t* ⇀ (*'x* × *'l released-locks*)
and *thr-update* :: *'t* ⇒ *'x* × *'l released-locks* ⇒ *'m-t* ⇒ *'m-t*
and *thr-iterate* :: *'m-t* ⇒ (*'t* × (*'x* × *'l released-locks*), *'s-t*) *set-iterator*
and *ws-α* :: *'m-w* ⇒ (*'w*,*'t*) *wait-sets*
and *ws-invar* :: *'m-w* ⇒ *bool*
and *ws-lookup* :: *'t* ⇒ *'m-w* ⇀ *'w wait-set-status*
and *ws-update* :: *'t* ⇒ *'w wait-set-status* ⇒ *'m-w* ⇒ *'m-w*
and *ws-sel* :: *'m-w* ⇒ (*'t* × *'w wait-set-status* ⇒ *bool*) ⇀ (*'t* × *'w wait-set-status*)
and *is-α* :: *'s-i* ⇒ *'t interrupts*
and *is-invar* :: *'s-i* ⇒ *bool*
and *is-memb* :: *'t* ⇒ *'s-i* ⇒ *bool*
and *is-ins* :: *'t* ⇒ *'s-i* ⇒ *'s-i*
and *is-delete* :: *'t* ⇒ *'s-i* ⇒ *'s-i*
+
fixes *thr'-α* :: *'s-t* ⇒ *'t set*

```
  and thr'-invar :: 's-t ⇒ bool
  and thr'-empty :: unit ⇒ 's-t
  and thr'-ins-dj :: 't ⇒ 's-t ⇒ 's-t
begin

abbreviation pick-wakeup :: 's ⇒ 't ⇒ 'w ⇒ 'm-w ⇒ 't option
where pick-wakeup ≡ pick-wakeup-via-sel (λs P. ws-sel s (λ(k,v). P k v))

fun active-threads :: ('l,'t,'m,'m-t,'m-w,'s-i) state-refine ⇒ 's-t
where
  active-threads (ls, (ts, m), ws, is) =
   thr-iterate ts (λ-. True)
      (λ(t, (x, ln)) ts'. if ln = no-wait-locks
                          then if Predicate.holds
                                  (do {
                                     (ta, -) ← r t (x, m);
                                     Predicate.if-pred (actions-ok (ls, (ts, m), ws, is) t ta)
                                   })
                               then thr'-ins-dj t ts'
                               else ts'
                          else if ¬ waiting (ws-lookup t ws) ∧ may-acquire-all ls t ln then thr'-ins-dj t ts' else
ts')
      (thr'-empty ())

end

locale scheduler-aux =
  scheduler-base-aux
    final r convert-RA
    thr-α thr-invar thr-lookup thr-update
    ws-α ws-invar ws-lookup
    is-α is-invar is-memb is-ins is-delete
  +
  thr: finite-map thr-α thr-invar +
  thr: map-lookup thr-α thr-invar thr-lookup +
  thr: map-update thr-α thr-invar thr-update +
  ws: map ws-α ws-invar +
  ws: map-lookup ws-α ws-invar ws-lookup +
  is: set is-α is-invar +
  is: set-memb is-α is-invar is-memb +
  is: set-ins is-α is-invar is-ins +
  is: set-delete is-α is-invar is-delete
  for final :: 'x ⇒ bool
  and r :: 't ⇒ ('x × 'm) ⇒ (('l,'t,'x,'m,'w,'o) thread-action × 'x × 'm) Predicate.pred
  and convert-RA :: 'l released-locks ⇒ 'o list
  and thr-α :: 'm-t ⇒ ('l,'t,'x) thread-info
  and thr-invar :: 'm-t ⇒ bool
  and thr-lookup :: 't ⇒ 'm-t ⇀ ('x × 'l released-locks)
  and thr-update :: 't ⇒ 'x × 'l released-locks ⇒ 'm-t ⇒ 'm-t
  and ws-α :: 'm-w ⇒ ('w,'t) wait-sets
  and ws-invar :: 'm-w ⇒ bool
  and ws-lookup :: 't ⇒ 'm-w ⇀ 'w wait-set-status
  and is-α :: 's-i ⇒ 't interrupts
  and is-invar :: 's-i ⇒ bool
```

and *is-memb* :: *'t* ⇒ *'s-i* ⇒ *bool*
and *is-ins* :: *'t* ⇒ *'s-i* ⇒ *'s-i*
and *is-delete* :: *'t* ⇒ *'s-i* ⇒ *'s-i*
begin

lemma *free-thread-id-correct* [*simp*]:
thr-invar ts ⟹ *free-thread-id ts* = *FWThread.free-thread-id* (*thr-α ts*)
by(*auto simp add*: *free-thread-id-def fun-eq-iff thr.lookup-correct intro*: *free-thread-id.intros*)

lemma *redT-updT-correct* [*simp*]:
assumes *thr-invar ts*
shows *thr-α* (*redT-updT ts nta*) = *FWThread.redT-updT* (*thr-α ts*) *nta*
and *thr-invar* (*redT-updT ts nta*)
by(*case-tac* [!] *nta*)(*simp-all add*: *thr.update-correct assms*)

lemma *redT-updTs-correct* [*simp*]:
assumes *thr-invar ts*
shows *thr-α* (*redT-updTs ts ntas*) = *FWThread.redT-updTs* (*thr-α ts*) *ntas*
and *thr-invar* (*redT-updTs ts ntas*)
using *assms*
by(*induct ntas arbitrary*: *ts*)(*simp-all add*: *redT-updTs-def*)

lemma *thread-ok-correct* [*simp*]:
thr-invar ts ⟹ *thread-ok ts nta* ⟷ *FWThread.thread-ok* (*thr-α ts*) *nta*
by(*cases nta*) *simp-all*

lemma *thread-oks-correct* [*simp*]:
thr-invar ts ⟹ *thread-oks ts ntas* ⟷ *FWThread.thread-oks* (*thr-α ts*) *ntas*
by(*induct ntas arbitrary*: *ts*) *simp-all*

lemma *wset-actions-ok-correct* [*simp*]:
ws-invar ws ⟹ *wset-actions-ok ws t was* ⟷ *FWWait.wset-actions-ok* (*ws-α ws*) *t was*
by(*simp add*: *wset-actions-ok-def FWWait.wset-actions-ok-def ws.lookup-correct*)

lemma *cond-action-ok-correct* [*simp*]:
state-invar s ⟹ *cond-action-ok s t cta* ⟷ *α.cond-action-ok* (*state-α s*) *t cta*
by(*cases s*,*cases cta*)(*auto simp add*: *thr.lookup-correct ws.lookup-correct*)

lemma *cond-action-oks-correct* [*simp*]:
assumes *state-invar s*
shows *cond-action-oks s t ctas* ⟷ *α.cond-action-oks* (*state-α s*) *t ctas*
by(*induct ctas*)(*simp-all add*: *cond-action-oks-def assms*)

lemma *redT-updI-correct* [*simp*]:
assumes *is-invar is*
shows *is-α* (*redT-updI is ia*) = *FWInterrupt.redT-updI* (*is-α is*) *ia*
and *is-invar* (*redT-updI is ia*)
using *assms*
by(*case-tac* [!] *ia*)(*auto simp add*: *is.ins-correct is.delete-correct*)

lemma *redT-updIs-correct* [*simp*]:
assumes *is-invar is*
shows *is-α* (*redT-updIs is ias*) = *FWInterrupt.redT-updIs* (*is-α is*) *ias*
and *is-invar* (*redT-updIs is ias*)

using *assms*
by(*induct ias arbitrary*: *is*)(*auto*)

lemma *interrupt-action-ok-correct* [*simp*]:
is-invar is $\Longrightarrow$ *interrupt-action-ok is ia* $\longleftrightarrow$ *FWInterrupt.interrupt-action-ok* (*is-*α *is*) *ia*
by(*cases ia*)(*auto simp add*: *is.memb-correct*)

lemma *interrupt-actions-ok-correct* [*simp*]:
is-invar is $\Longrightarrow$ *interrupt-actions-ok is ias* $\longleftrightarrow$ *FWInterrupt.interrupt-actions-ok* (*is-*α *is*) *ias*
by(*induct ias arbitrary*:*is*) *simp-all*

lemma *actions-ok-correct* [*simp*]:
state-invar s $\Longrightarrow$ *actions-ok s t ta* $\longleftrightarrow$ α*.actions-ok* (*state-*α *s*) *t ta*
by(*auto simp add*: *actions-ok-def*)

end

locale *scheduler* =
scheduler-base
final r convert-RA
schedule output pick-wakeup σ*-invar*
thr-α *thr-invar thr-lookup thr-update*
ws-α *ws-invar ws-lookup ws-update ws-delete ws-iterate*
is-α *is-invar is-memb is-ins is-delete*
+
scheduler-aux
final r convert-RA
thr-α *thr-invar thr-lookup thr-update*
ws-α *ws-invar ws-lookup*
is-α *is-invar is-memb is-ins is-delete*
+
scheduler-spec
final r convert-RA
schedule σ*-invar*
thr-α *thr-invar*
ws-α *ws-invar*
is-α *is-invar*
invariant
+
pick-wakeup-spec
final r convert-RA
pick-wakeup σ*-invar*
thr-α *thr-invar*
ws-α *ws-invar*
is-α *is-invar*
+
ws: *map-update ws-*α *ws-invar ws-update* +
ws: *map-delete ws-*α *ws-invar ws-delete* +
ws: *map-iteratei ws-*α *ws-invar ws-iterate*
for *final* :: $'x \Rightarrow$ *bool*
and *r* :: $'t \Rightarrow ('x \times 'm) \Rightarrow (('l,'t,'x,'m,'w,'o)$ *thread-action* $\times$ $'x \times 'm)$ *Predicate.pred*
and *convert-RA* :: $'l$ *released-locks* $\Rightarrow$ $'o$ *list*
and *schedule* :: $('l,'t,'x,'m,'w,'o,'m\text{-}t,'m\text{-}w,'s\text{-}i,'s)$ *scheduler*
and *output* :: $'s \Rightarrow 't \Rightarrow ('l,'t,'x,'m,'w,'o)$ *thread-action* $\Rightarrow$ $'q$ *option*

```
  and pick-wakeup :: 's ⇒ 't ⇒ 'w ⇒ 'm-w ⇒ 't option
  and σ-invar :: 's ⇒ 't set ⇒ bool
  and thr-α :: 'm-t ⇒ ('l,'t,'x) thread-info
  and thr-invar :: 'm-t ⇒ bool
  and thr-lookup :: 't ⇒ 'm-t ⇀ ('x × 'l released-locks)
  and thr-update :: 't ⇒ 'x × 'l released-locks ⇒ 'm-t ⇒ 'm-t
  and ws-α :: 'm-w ⇒ ('w,'t) wait-sets
  and ws-invar :: 'm-w ⇒ bool
  and ws-lookup :: 't ⇒ 'm-w ⇀ 'w wait-set-status
  and ws-update :: 't ⇒ 'w wait-set-status ⇒ 'm-w ⇒ 'm-w
  and ws-delete :: 't ⇒ 'm-w ⇒ 'm-w
  and ws-iterate :: 'm-w ⇒ ('t × 'w wait-set-status, 'm-w) set-iterator
  and is-α :: 's-i ⇒ 't interrupts
  and is-invar :: 's-i ⇒ bool
  and is-memb :: 't ⇒ 's-i ⇒ bool
  and is-ins :: 't ⇒ 's-i ⇒ 's-i
  and is-delete :: 't ⇒ 's-i ⇒ 's-i
  and invariant :: ('l,'t,'x,'m,'w) state set
  +
  assumes invariant: invariant3p α.redT invariant
begin

lemma exec-updW-correct:
  assumes invar: ws-invar ws σ-invar σ T dom (ws-α ws) ⊆ T t ∈ T
  shows redT-updW t (ws-α ws) wa (ws-α (exec-updW σ t ws wa)) (is ?thesis1)
  and ws-invar (exec-updW σ t ws wa) (is ?thesis2)
proof −
  from invar have ?thesis1 ∧ ?thesis2
  proof(cases wa)
    case [simp]: (Notify w)
    show ?thesis
    proof(cases pick-wakeup σ t w ws)
      case (Some t')
      hence ws-α ws t' = ⌊InWS w⌋ using invar by(rule pick-wakeup-SomeD)
      with Some show ?thesis using invar by(auto simp add: ws.update-correct)
    next
      case None
      hence InWS w ∉ ran (ws-α ws) using invar by(rule pick-wakeup-NoneD)
      with None show ?thesis using invar by(auto simp add: ran-def)
    qed
  next
    case [simp]: (NotifyAll w)
    let ?f = λ(t, w') ws'. if w' = InWS w then ws-update t (PostWS WSNotified) ws' else ws'
    let ?I = λT ws'. (∀ k. if k∉T ∧ ws-α ws k = ⌊InWS w⌋ then ws-α ws' k = ⌊PostWS WSNotified⌋
else ws-α ws' k = ws-α ws k) ∧ ws-invar ws'
    from invar have ?I (dom (ws-α ws)) ws by(auto simp add: ws.lookup-correct)
    with ‹ws-invar ws› have ?I {} (ws-iterate ws (λ-. True) ?f ws)
    proof(rule ws.iterate-rule-P[where I=?I])
      fix t w' T ws'
      assume t: t ∈ T and w': ws-α ws t = ⌊w'⌋
        and T: T ⊆ dom (ws-α ws) and I: ?I T ws'
      { fix t'
        assume t' ∉ T − {t} ws-α ws t' = ⌊InWS w⌋
        with t I w' invar have ws-α (?f (t, w') ws') t' = ⌊PostWS WSNotified⌋
```

```
        by(auto)(simp-all add: ws.update-correct) }
      moreover {
       fix t'
       assume t' ∈ T − {t} ∨ ws-α ws t' ≠ ⌊InWS w⌋
       with t I w' invar have ws-α (?f (t,w') ws') t' = ws-α ws t'
        by(auto simp add: ws.update-correct) }
      moreover
      have ws-invar (?f (t, w') ws') using I by(simp add: ws.update-correct)
      ultimately show ?I (T − {t}) (?f (t, w') ws') by safe simp
    qed
     hence ws-α (ws-iterate ws (λ-. True) ?f ws) = (λt. if ws-α ws t = ⌊InWS w⌋ then ⌊PostWS
WSNotified⌋ else ws-α ws t)
      and ws-invar (ws-iterate ws (λ-. True) ?f ws) by(simp-all add: fun-eq-iff)
    thus ?thesis by simp
  next
    case WakeUp thus ?thesis using assms
      by(auto simp add: ws.lookup-correct ws.update-correct split: wait-set-status.split)
  qed(simp-all add: ws.update-correct ws.delete-correct map-upd-eq-restrict)
  thus ?thesis1 ?thesis2 by simp-all
qed

lemma exec-updWs-correct:
  assumes ws-invar ws σ-invar σ T dom (ws-α ws) ⊆ T t ∈ T
  shows redT-updWs t (ws-α ws) was (ws-α (exec-updWs σ t ws was)) (is ?thesis1)
  and ws-invar (exec-updWs σ t ws was) (is ?thesis2)
proof −
  from ‹ws-invar ws› ‹dom (ws-α ws) ⊆ T›
  have ?thesis1 ∧ ?thesis2
  proof(induct was arbitrary: ws)
    case Nil thus ?case by(auto simp add: exec-updWs-def redT-updWs-def)
  next
    case (Cons wa was)
    let ?ws' = exec-updW σ t ws wa
    from ‹ws-invar ws› ‹σ-invar σ T› ‹dom (ws-α ws) ⊆ T› ‹t ∈ T›
    have invar': ws-invar ?ws' and red: redT-updW t (ws-α ws) wa (ws-α ?ws')
      by(rule exec-updW-correct)+
    have dom (ws-α ?ws') ⊆ T
    proof
      fix t' assume t' ∈ dom (ws-α ?ws')
      with red have t' ∈ dom (ws-α ws) ∨ t = t'
        by(auto dest!: redT-updW-Some-otherD split: wait-set-status.split-asm)
      with ‹dom (ws-α ws) ⊆ T› ‹t ∈ T› show t' ∈ T by auto
    qed
     with invar' have redT-updWs t (ws-α ?ws') was (ws-α (exec-updWs σ t ?ws' was)) ∧ ws-invar
(exec-updWs σ t ?ws' was)
      by(rule Cons.hyps)
    thus ?case using red
      by(auto simp add: exec-updWs-def redT-updWs-def intro: rtrancl3p-step-converse)
  qed
  thus ?thesis1 ?thesis2 by simp-all
qed

lemma exec-upd-correct:
  assumes state-invar s σ-invar σ (dom (thr-α (thr s))) t ∈ (dom (thr-α (thr s)))
```

```
  and wset-thread-ok (ws-α (wset s)) (thr-α (thr s))
  shows redT-upd (state-α s) t ta x' m' (state-α (exec-upd σ s t ta x' m'))
  and state-invar (exec-upd σ s t ta x' m')
using assms unfolding wset-thread-ok-conv-dom
by(auto simp add: thr.update-correct thr.lookup-correct intro: exec-updWs-correct)

lemma execT-None:
  assumes invar: state-invar s σ-invar σ (dom (thr-α (thr s))) state-α s ∈ invariant
  and exec: execT σ s = None
  shows α.active-threads (state-α s) = {}
using assms
by(cases schedule σ s)(fastforce simp add: execT-def thr.lookup-correct dest: schedule-Some-NoneD
schedule-NoneD)+

lemma execT-Some:
  assumes invar: state-invar s σ-invar σ (dom (thr-α (thr s))) state-α s ∈ invariant
  and wstok: wset-thread-ok (ws-α (wset s)) (thr-α (thr s))
  and exec: execT σ s = ⌊(σ', t, ta, s')⌋
  shows α.redT (state-α s) (t, ta) (state-α s') (is ?thesis1)
  and state-invar s' (is ?thesis2)
  and σ-invar σ' (dom (thr-α (thr s'))) (is ?thesis3)
proof −
  note [simp del] = redT-upd-simps exec-upd-def

  have ?thesis1 ∧ ?thesis2 ∧ ?thesis3
  proof(cases fst (snd (the (schedule σ s))))
    case None
    with exec invar have schedule: schedule σ s = ⌊(t, None, σ')⌋
      and ta: ta = (K$ [], [], [], [], [], convert-RA (snd (the (thr-α (thr s) t))))
     and s': s' = (acquire-all (locks s) t (snd (the (thr-α (thr s) t))), (thr-update t (fst (the (thr-α (thr
s) t)), no-wait-locks) (thr s), shr s), wset s, interrupts s)
     by(auto simp add: execT-def bind-eq-Some-conv thr.lookup-correct split-beta split del: option.split-asm)
    from schedule-Some-NoneD[OF schedule invar]

    obtain x ln n where t: thr-α (thr s) t = ⌊(x, ln)⌋
      and 0 < ln $ n ¬ waiting (ws-α (wset s) t) may-acquire-all (locks s) t ln by blast
   hence ?thesis1 using ta s' invar by(auto intro: α.redT.redT-acquire simp add: thr.update-correct)
    moreover from invar s' have ?thesis2 by(simp add: thr.update-correct)
     moreover from t s' invar have dom (thr-α (thr s')) = dom (thr-α (thr s)) by(auto simp add:
thr.update-correct)
    hence ?thesis3 using invar schedule by(auto intro: schedule-invar-None)
    ultimately show ?thesis by simp
  next
    case (Some taxm)
    with exec invar obtain x' m'
      where schedule: schedule σ s = ⌊(t, ⌊(ta, x', m')⌋, σ')⌋
      and s': s' = exec-upd σ s t ta x' m'
      by(cases taxm)(fastforce simp add: execT-def bind-eq-Some-conv split del: option.split-asm)
    from schedule-Some-SomeD[OF schedule invar]
    obtain x where t: thr-α (thr s) t = ⌊(x, no-wait-locks)⌋
      and Predicate.eval (r t (x, shr s)) (ta, x', m')
      and aok: α.actions-ok (state-α s) t ta by blast
    with s' have ?thesis1 using invar wstok
      by(fastforce intro: α.redT.intros exec-upd-correct)
```

```
    moreover from invar s' t wstok have ?thesis2 by(auto intro: exec-upd-correct)
    moreover {
      from schedule invar
      have σ-invar σ' (dom (thr-α (thr s)) ∪ {t. ∃x m. NewThread t x m ∈ set {|ta|}_t})
        by(rule schedule-invar-Some)
     also have dom (thr-α (thr s)) ∪ {t. ∃x m. NewThread t x m ∈ set {|ta|}_t} = dom (thr-α (thr s'))
        using invar s' aok t
      by(auto simp add: exec-upd-def thr.lookup-correct thr.update-correct simp del: split-paired-Ex)(fastforce
dest: redT-updTs-new-thread intro: redT-updTs-Some1 redT-updTs-new-thread-ts simp del: split-paired-Ex)+
      finally have σ-invar σ' (dom (thr-α (thr s'))) . }
    ultimately show ?thesis by simp
  qed
  thus ?thesis1 ?thesis2 ?thesis3 by simp-all
qed

lemma exec-step-into-redT:
  assumes invar: state-invar s σ-invar σ (dom (thr-α (thr s))) state-α s ∈ invariant
  and wstok: wset-thread-ok (ws-α (wset s)) (thr-α (thr s))
  and exec: exec-step (σ, s) = Inl ((σ'', t, ta), σ', s')
  shows α.redT (state-α s) (t, ta) (state-α s') σ'' = σ
  and state-invar s' σ-invar σ' (dom (thr-α (thr s'))) state-α s' ∈ invariant
proof −
  from exec have execT: execT σ s = ⌊(σ', t, ta, s')⌋
    and q: σ'' = σ by(auto simp add: exec-step.simps split-beta)
  from invar wstok execT show red: α.redT (state-α s) (t, ta) (state-α s')
    and invar': state-invar s' σ-invar σ' (dom (thr-α (thr s'))) σ'' = σ
    by(rule execT-Some)+(rule q)
  from invariant red ‹state-α s ∈ invariant›
  show state-α s' ∈ invariant by(rule invariant3pD)
qed

lemma exec-step-InrD:
  assumes state-invar s σ-invar σ (dom (thr-α (thr s))) state-α s ∈ invariant
  and exec-step (σ, s) = Inr s'
  shows α.active-threads (state-α s) = {}
  and s' = s
using assms
by(auto simp add: exec-step-def dest: execT-None)

lemma (in multithreaded-base) red-in-active-threads:
  assumes s −t▷ta→ s'
  shows t ∈ active-threads s
using assms
by cases(auto intro: active-threads.intros)

lemma exec-aux-into-Runs:
  assumes state-invar s σ-invar σ (dom (thr-α (thr s))) state-α s ∈ invariant
  and wset-thread-ok (ws-α (wset s)) (thr-α (thr s))
  shows α.mthr.Runs (state-α s) (lmap snd (llist-of-tllist (exec-aux (σ, s)))) (is ?thesis1)
  and tfinite (exec-aux (σ, s)) ⟹ state-invar (terminal (exec-aux (σ, s))) (is - ⟹ ?thesis2)
proof −
  from assms show ?thesis1
  proof(coinduction arbitrary: σ s)
    case (Runs σ s)
```

```
    note invar = ‹state-invar s› ‹σ-invar σ (dom (thr-α (thr s)))› ‹state-α s ∈ invariant›
      and wstok = ‹wset-thread-ok (ws-α (wset s)) (thr-α (thr s))›
    show ?case
    proof(cases exec-aux (σ, s))
      case (TNil s')
      hence α.active-threads (state-α s) = {} s' = s
       by(auto simp add: exec-aux-def unfold-tllist' split: sum.split-asm dest: exec-step-InrD[OF invar])
      hence ?Stuck using TNil by(auto dest: α.red-in-active-threads)
      thus ?thesis ..
    next
      case (TCons σtta ttls')
      then obtain t ta σ' s' σ''
        where [simp]: σtta = (σ'', t, ta)
        and [simp]: ttls' = exec-aux (σ', s')
        and step: exec-step (σ, s) = Inl ((σ'', t, ta), σ', s')
        unfolding exec-aux-def by(subst (asm) (2) unfold-tllist')(fastforce split: sum.split-asm)
      from invar wstok step
      have redT: α.redT (state-α s) (t, ta) (state-α s')
        and [simp]: σ'' = σ
        and invar': state-invar s' σ-invar σ' (dom (thr-α (thr s'))) state-α s' ∈ invariant
        by(rule exec-step-into-redT)+
      from wstok α.redT-preserves-wset-thread-ok[OF redT]
      have wset-thread-ok (ws-α (wset s')) (thr-α (thr s')) by simp
      with invar' redT TCons have ?Step by(auto simp del: split-paired-Ex)
      thus ?thesis ..
    qed
  qed
next
  assume tfinite (exec-aux (σ, s))
  thus ?thesis2 using assms
  proof(induct exec-aux (σ, s) arbitrary: σ s rule: tfinite-induct)
    case TNil thus ?case
      by(auto simp add: exec-aux-def unfold-tllist' split-beta split: sum.split-asm dest: exec-step-InrD)
  next
    case (TCons σtta ttls)
    from ‹TCons σtta ttls = exec-aux (σ, s)›
    obtain σ'' t ta σ' s'
      where [simp]: σtta = (σ'', t, ta)
      and ttls: ttls = exec-aux (σ', s')
      and step: exec-step (σ, s) = Inl ((σ'', t, ta), σ', s')
      unfolding exec-aux-def by(subst (asm) (2) unfold-tllist')(fastforce split: sum.split-asm)
    note ttls moreover
   from ‹state-invar s› ‹σ-invar σ (dom (thr-α (thr s)))› ‹state-α s ∈ invariant› ‹wset-thread-ok (ws-α
(wset s)) (thr-α (thr s))› step
    have [simp]: σ'' = σ
      and invar': state-invar s' σ-invar σ' (dom (thr-α (thr s'))) state-α s' ∈ invariant
      and redT: α.redT (state-α s) (t, ta) (state-α s')
      by(rule exec-step-into-redT)+
    note invar' moreover
    from α.redT-preserves-wset-thread-ok[OF redT] ‹wset-thread-ok (ws-α (wset s)) (thr-α (thr s))›
    have wset-thread-ok (ws-α (wset s')) (thr-α (thr s')) by simp
    ultimately have state-invar (terminal (exec-aux (σ', s'))) by(rule TCons)
    with ‹TCons σtta ttls = exec-aux (σ, s)›[symmetric]
    show ?case unfolding ttls by simp
```

qed
qed

end

locale *scheduler-ext-aux* =
scheduler-ext-base
final r convert-RA
thr-α thr-invar thr-lookup thr-update thr-iterate
ws-α ws-invar ws-lookup ws-update ws-sel
is-α is-invar is-memb is-ins is-delete
thr′-α thr′-invar thr′-empty thr′-ins-dj
+
scheduler-aux
final r convert-RA
thr-α thr-invar thr-lookup thr-update
ws-α ws-invar ws-lookup
is-α is-invar is-memb is-ins is-delete
+
thr: *map-iteratei thr-α thr-invar thr-iterate* +
ws: *map-update ws-α ws-invar ws-update* +
ws: *map-sel′ ws-α ws-invar ws-sel* +
thr′: *finite-set thr′-α thr′-invar* +
thr′: *set-empty thr′-α thr′-invar thr′-empty* +
thr′: *set-ins-dj thr′-α thr′-invar thr′-ins-dj*
for *final* :: *'x ⇒ bool*
and *r* :: *'t ⇒ ('x × 'm) ⇒ (('l,'t,'x,'m,'w,'o) thread-action × 'x × 'm) Predicate.pred*
and *convert-RA* :: *'l released-locks ⇒ 'o list*
and *thr-α* :: *'m-t ⇒ ('l,'t,'x) thread-info*
and *thr-invar* :: *'m-t ⇒ bool*
and *thr-lookup* :: *'t ⇒ 'm-t ⇀ ('x × 'l released-locks)*
and *thr-update* :: *'t ⇒ 'x × 'l released-locks ⇒ 'm-t ⇒ 'm-t*
and *thr-iterate* :: *'m-t ⇒ ('t × ('x × 'l released-locks), 's-t) set-iterator*
and *ws-α* :: *'m-w ⇒ ('w,'t) wait-sets*
and *ws-invar* :: *'m-w ⇒ bool*
and *ws-lookup* :: *'t ⇒ 'm-w ⇀ 'w wait-set-status*
and *ws-update* :: *'t ⇒ 'w wait-set-status ⇒ 'm-w ⇒ 'm-w*
and *ws-sel* :: *'m-w ⇒ (('t × 'w wait-set-status) ⇒ bool) ⇀ ('t × 'w wait-set-status)*
and *is-α* :: *'s-i ⇒ 't interrupts*
and *is-invar* :: *'s-i ⇒ bool*
and *is-memb* :: *'t ⇒ 's-i ⇒ bool*
and *is-ins* :: *'t ⇒ 's-i ⇒ 's-i*
and *is-delete* :: *'t ⇒ 's-i ⇒ 's-i*
and *thr′-α* :: *'s-t ⇒ 't set*
and *thr′-invar* :: *'s-t ⇒ bool*
and *thr′-empty* :: *unit ⇒ 's-t*
and *thr′-ins-dj* :: *'t ⇒ 's-t ⇒ 's-t*
begin

lemma *active-threads-correct* [*simp*]:
assumes *state-invar s*
shows *thr′-α* (*active-threads s*) = *α.active-threads* (*state-α s*) (**is** *?thesis1*)
and *thr′-invar* (*active-threads s*) (**is** *?thesis2*)
proof −

```
  obtain ls ts m ws is where s: s = (ls, (ts, m), ws, is) by(cases s) fastforce
  let ?f = λ(t, (x, ln)) TS. if ln = no-wait-locks
          then if Predicate.holds (do { (ta, -) ← r t (x, m); Predicate.if-pred (actions-ok (ls, (ts, m),
ws, is) t ta) })
               then thr'-ins-dj t TS else TS
          else if ¬ waiting (ws-lookup t ws) ∧ may-acquire-all ls t ln then thr'-ins-dj t TS else TS
  let ?I = λT TS. thr'-invar TS ∧ thr'-α TS ⊆ dom (thr-α ts) − T ∧ (∀t. t ∉ T ⟶ t ∈ thr'-α TS
⟷ t ∈ α.active-threads (state-α s))

  from assms s have thr-invar ts by simp
  moreover have ?I (dom (thr-α ts)) (thr'-empty ())
    by(auto simp add: thr'.empty-correct s elim: α.active-threads.cases)
  ultimately have ?I {} (thr-iterate ts (λ-. True) ?f (thr'-empty ()))
  proof(rule thr.iterate-rule-P[where I=?I])
    fix t xln T TS
    assume tT: t ∈ T
      and tst: thr-α ts t = ⌊xln⌋
      and Tdom: T ⊆ dom (thr-α ts)
      and I: ?I T TS
    obtain x ln where xln: xln = (x, ln) by(cases xln)
    from tT I have t: t ∉ thr'-α TS by blast

    from I have invar: thr'-invar TS ..
    hence thr'-invar (?f (t, xln) TS) using t
      unfolding xln by(auto simp add: thr'.ins-dj-correct)
    moreover from I have thr'-α TS ⊆ dom (thr-α ts) − T by blast
    hence thr'-α (?f (t, xln) TS) ⊆ dom (thr-α ts) − (T − {t})
      using invar tst t by(auto simp add: xln thr'.ins-dj-correct)
    moreover
    {
      fix t'
      assume t': t' ∉ T − {t}
      have t' ∈ thr'-α (?f (t, xln) TS) ⟷ t' ∈ α.active-threads (state-α s) (is ?lhs ⟷ ?rhs)
      proof(cases t' = t)
        case True
        show ?thesis
        proof
          assume ?lhs
          with True xln invar tst ‹state-invar s› t show ?rhs
         by(fastforce simp add: holds-eq thr'.ins-dj-correct s split-beta ws.lookup-correct split: if-split-asm
elim!: bindE if-predE intro: α.active-threads.intros)
        next
          assume ?rhs
          with True xln invar tst ‹state-invar s› t show ?lhs
         by(fastforce elim!: α.active-threads.cases simp add: holds-eq s thr'.ins-dj-correct ws.lookup-correct
elim!: bindE if-predE intro: bindI if-predI)
        qed
      next
        case False
        with t' have t' ∉ T by simp
        with I have t' ∈ thr'-α TS ⟷ t' ∈ α.active-threads (state-α s) by blast
        thus ?thesis using xln False invar t by(auto simp add: thr'.ins-dj-correct)
      qed
    }
```

ultimately show *?I* (*T* − {*t*}) (*?f* (*t*, *xln*) *TS*) **by** *blast*
qed
thus *?thesis1 ?thesis2* **by**(*auto simp add*: *s*)
qed

end

locale *scheduler-ext* =
scheduler-ext-aux
final r convert-RA
thr-α thr-invar thr-lookup thr-update thr-iterate
ws-α ws-invar ws-lookup ws-update ws-sel
is-α is-invar is-memb is-ins is-delete
thr′-α thr′-invar thr′-empty thr′-ins-dj
+
scheduler-spec
final r convert-RA
schedule σ-invar
thr-α thr-invar
ws-α ws-invar
is-α is-invar
invariant
+
ws: *map-delete ws-α ws-invar ws-delete* +
ws: *map-iteratei ws-α ws-invar ws-iterate*
for *final* :: *'x* ⇒ *bool*
and *r* :: *'t* ⇒ (*'x* × *'m*) ⇒ ((*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* × *'x* × *'m*) *Predicate.pred*
and *convert-RA* :: *'l released-locks* ⇒ *'o list*
and *schedule* :: (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*,*'m-t*,*'m-w*,*'s-i*,*'s*) *scheduler*
and *output* :: *'s* ⇒ *'t* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* ⇒ *'q option*
and *σ-invar* :: *'s* ⇒ *'t set* ⇒ *bool*
and *thr-α* :: *'m-t* ⇒ (*'l*,*'t*,*'x*) *thread-info*
and *thr-invar* :: *'m-t* ⇒ *bool*
and *thr-lookup* :: *'t* ⇒ *'m-t* ⇀ (*'x* × *'l released-locks*)
and *thr-update* :: *'t* ⇒ *'x* × *'l released-locks* ⇒ *'m-t* ⇒ *'m-t*
and *thr-iterate* :: *'m-t* ⇒ (*'t* × (*'x* × *'l released-locks*), *'s-t*) *set-iterator*
and *ws-α* :: *'m-w* ⇒ (*'w*,*'t*) *wait-sets*
and *ws-invar* :: *'m-w* ⇒ *bool*
and *ws-empty* :: *unit* ⇒ *'m-w*
and *ws-lookup* :: *'t* ⇒ *'m-w* ⇀ *'w wait-set-status*
and *ws-update* :: *'t* ⇒ *'w wait-set-status* ⇒ *'m-w* ⇒ *'m-w*
and *ws-delete* :: *'t* ⇒ *'m-w* ⇒ *'m-w*
and *ws-iterate* :: *'m-w* ⇒ (*'t* × *'w wait-set-status*, *'m-w*) *set-iterator*
and *ws-sel* :: *'m-w* ⇒ (*'t* × *'w wait-set-status* ⇒ *bool*) ⇀ (*'t* × *'w wait-set-status*)
and *is-α* :: *'s-i* ⇒ *'t interrupts*
and *is-invar* :: *'s-i* ⇒ *bool*
and *is-memb* :: *'t* ⇒ *'s-i* ⇒ *bool*
and *is-ins* :: *'t* ⇒ *'s-i* ⇒ *'s-i*
and *is-delete* :: *'t* ⇒ *'s-i* ⇒ *'s-i*
and *thr′-α* :: *'s-t* ⇒ *'t set*
and *thr′-invar* :: *'s-t* ⇒ *bool*
and *thr′-empty* :: *unit* ⇒ *'s-t*
and *thr′-ins-dj* :: *'t* ⇒ *'s-t* ⇒ *'s-t*
and *invariant* :: (*'l*,*'t*,*'x*,*'m*,*'w*) *state set*

```
  +
  assumes invariant: invariant3p α.redT invariant

sublocale scheduler-ext <
  pick-wakeup-spec
    final r convert-RA
    pick-wakeup σ-invar
    thr-α thr-invar
    ws-α ws-invar
by(rule pick-wakeup-spec-via-sel)(unfold-locales)

sublocale scheduler-ext <
  scheduler
    final r convert-RA
    schedule output pick-wakeup σ-invar
    thr-α thr-invar thr-lookup thr-update
    ws-α ws-invar ws-lookup ws-update ws-delete ws-iterate
    is-α is-invar is-memb is-ins is-delete
    invariant
by(unfold-locales)(rule invariant)
```

9.2.1 Schedulers for deterministic small-step semantics

The default code equations for *Predicate.the* impose the type class constraint *eq* on the predicate elements. For the semantics, which contains the heap, there might be no such instance, so we use new constants for which other code equations can be used. These do not add the type class constraint, but may fail more often with non-uniqueness exception.

```
definition singleton2 where [simp]: singleton2 = Predicate.singleton
definition the-only2 where [simp]: the-only2 = Predicate.the-only
definition the2 where [simp]: the2 = Predicate.the

context multithreaded-base begin

definition step-thread ::
  (('l,'t,'x,'m,'w,'o) thread-action ⇒ 's) ⇒ ('l,'t,'x,'m,'w) state ⇒ 't
  ⇒ ('t × (('l,'t,'x,'m,'w,'o) thread-action × 'x × 'm) option × 's) option
where
  ⋀ln. step-thread update-state s t =
   (case thr s t of
      ⌊(x, ln)⌋ ⇒
      if ln = no-wait-locks then
        if ∃ta x' m'. t ⊢ (x, shr s) −ta→ (x', m') ∧ actions-ok s t ta then
          let
            (ta, x', m') = THE (ta, x', m'). t ⊢ (x, shr s) −ta→ (x', m') ∧ actions-ok s t ta
          in
            ⌊(t, ⌊(ta, x', m')⌋, update-state ta)⌋
        else
          None
      else if may-acquire-all (locks s) t ln ∧ ¬ waiting (wset s t) then
        ⌊(t, None, update-state (K$ [], [], [], [], [], convert-RA ln))⌋
      else
        None
    | None ⇒ None)
```

lemma *step-thread-NoneD*:
step-thread update-state s t = None $\Longrightarrow$ *t* $\notin$ *active-threads s*
unfolding *step-thread-def*
by(*fastforce simp add*: *split-beta elim*!: *active-threads.cases split*: *if-split-asm*)

lemma *inactive-step-thread-eq-NoneI*:
t $\notin$ *active-threads s* $\Longrightarrow$ *step-thread update-state s t = None*
unfolding *step-thread-def*
by(*fastforce simp add*: *split-beta split*: *if-split-asm intro*: *active-threads.intros*)

lemma *step-thread-eq-None-conv*:
step-thread update-state s t = None $\longleftrightarrow$ *t* $\notin$ *active-threads s*
by(*blast dest*: *step-thread-NoneD intro*: *inactive-step-thread-eq-NoneI*)

lemma *step-thread-eq-Some-activeD*:
step-thread update-state s t = $\lfloor$(*t'*, *taxmσ'*)$\rfloor$
$\Longrightarrow$ *t'* = *t* $\wedge$ *t* $\in$ *active-threads s*
unfolding *step-thread-def*
by(*fastforce split*: *if-split-asm simp add*: *split-beta intro*: *active-threads.intros*)

declare *actions-ok-iff* [*simp del*]
declare *actions-ok.cases* [*rule del*]

lemma *step-thread-Some-NoneD*:
step-thread update-state s t' = $\lfloor$(*t*, *None*, *σ'*)$\rfloor$
$\Longrightarrow$ $\exists$ *x ln n. thr s t* = $\lfloor$(*x*, *ln*)$\rfloor$ $\wedge$ *ln* $ *n* > *0* $\wedge$ $\neg$ *waiting* (*wset s t*) $\wedge$ *may-acquire-all* (*locks s*) *t ln* $\wedge$ *σ'* = *update-state* (*K*$ [], [], [], [], *convert-RA ln*)
unfolding *step-thread-def*
by(*auto split*: *if-split-asm simp add*: *split-beta elim*!: *neq-no-wait-locksE*)

lemma *step-thread-Some-SomeD*:
⟦ *deterministic I*; *step-thread update-state s t'* = $\lfloor$(*t*, $\lfloor$(*ta*, *x'*, *m'*)$\rfloor$, *σ'*)$\rfloor$; *s* $\in$ *I* ⟧
$\Longrightarrow$ $\exists$ *x*. *thr s t* = $\lfloor$(*x*, *no-wait-locks*)$\rfloor$ $\wedge$ *t* $\vdash$ ⟨*x*, *shr s*⟩ −*ta*→ ⟨*x'*, *m'*⟩ $\wedge$ *actions-ok s t ta* $\wedge$ *σ'* = *update-state ta*
unfolding *step-thread-def*
by(*auto simp add*: *split-beta deterministic-THE split*: *if-split-asm*)

end

context *scheduler-base-aux* **begin**

definition *step-thread* ::
(('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* ⇒ '*s*) ⇒ ('*l*,'*t*,'*m*,'*m-t*,'*m-w*,'*s-i*) *state-refine* ⇒ '*t* ⇒
('*t* × (('*l*,'*t*,'*x*,'*m*,'*w*,'*o*) *thread-action* × '*x* × '*m*) *option* × '*s*) *option*
where
⋀*ln. step-thread update-state s t* =
(*case thr-lookup t* (*thr s*) *of*
$\lfloor$(*x*, *ln*)$\rfloor$ ⇒
if ln = no-wait-locks then
let
reds = *do* {
(*ta*, *x'*, *m'*) ← *r t* (*x*, *shr s*);
if actions-ok s t ta then Predicate.single (*ta*, *x'*, *m'*) *else bot*

```
        }
      in
        if Predicate.holds (reds ≫= (λ-. Predicate.single ())) then
          let
            (ta, x', m') = the2 reds
          in
            ⌊(t, ⌊(ta, x', m')⌋, update-state ta)⌋
        else
          None
      else if may-acquire-all (locks s) t ln ∧ ¬ waiting (ws-lookup t (wset s)) then
        ⌊(t, None, update-state (K$ [], [], [], [], [],convert-RA ln))⌋
      else
        None
  | None ⇒ None)
```

end

context *scheduler-aux* **begin**

lemma *deterministic-THE2*:
assumes α.*deterministic I*
and *tst*: *thr-*α (*thr s*) *t* = ⌊(*x*, *no-wait-locks*)⌋
and *red*: *Predicate.eval* (*r t* (*x*, *shr s*)) (*ta*, *x'*, *m'*)
and *aok*: α.*actions-ok* (*state-*α *s*) *t ta*
and *I*: *state-*α *s* ∈ *I*
shows *Predicate.the* (*r t* (*x*, *shr s*) ≫= (λ(*ta*, *x'*, *m'*). *if* α.*actions-ok* (*state-*α *s*) *t ta then Predicate.single* (*ta*, *x'*, *m'*) *else bot*)) = (*ta*, *x'*, *m'*)
proof −
show *?thesis* **unfolding** *the-def*
apply(*rule the-equality*)
apply(*rule bindI*[*OF red*])
apply(*simp add*: *singleI aok*)
apply(*erule bindE*)
apply(*clarsimp split*: *if-split-asm*)
apply(*drule* (*1*) α.*deterministicD*[*OF* ‹α.*deterministic I*›, **where** *s*=*state-*α *s*, *simplified*, *OF red - tst aok*])
apply(*rule I*)
apply *simp*
done
qed

lemma *step-thread-correct*:
assumes *det*: α.*deterministic I*
and *invar*: σ-*invar* σ (*dom* (*thr-*α (*thr s*))) *state-invar s state-*α *s* ∈ *I*
shows
map-option (*apsnd* (*apsnd* σ-α)) (*step-thread update-state s t*) = α.*step-thread* (σ-α ∘ *update-state*) (*state-*α *s*) *t* (**is** *?thesis1*)
and (⋀*ta*. *FWThread.thread-oks* (*thr-*α (*thr s*)) ⦃*ta*⦄$_t$ ⟹ σ-*invar* (*update-state ta*) (*dom* (*thr-*α (*thr s*)) ∪ {*t*. ∃*x m*. *NewThread t x m* ∈ *set* ⦃*ta*⦄$_t$})) ⟹ *case-option True* (λ(*t*, *taxm*, σ). σ-*invar* σ (*case taxm of None* ⇒ *dom* (*thr-*α (*thr s*)) | *Some* (*ta*, *x'*, *m'*) ⇒ *dom* (*thr-*α (*thr s*)) ∪ {*t*. ∃*x m*. *NewThread t x m* ∈ *set* ⦃*ta*⦄$_t$})) (*step-thread update-state s t*)
(**is** (⋀*ta*. *?tso ta* ⟹ *?inv ta*) ⟹ *?thesis2*)
proof −
have *?thesis1* ∧ ((∀ *ta*. *?tso ta* ⟶ *?inv ta*) ⟶ *?thesis2*)
```
```

```
  proof(cases step-thread update-state s t)
    case None
    with invar show ?thesis
      apply (auto simp add: thr.lookup-correct α.step-thread-def step-thread-def ws.lookup-correct
        split-beta holds-eq split: if-split-asm cong del: image-cong-simp)
       apply metis
      apply metis
      done
  next
    case (Some a)
    then obtain t' taxm σ'
      where rrs: step-thread update-state s t = ⌊(t', taxm, σ')⌋ by(cases a) auto
    show ?thesis
    proof(cases taxm)
      case None
      with rrs invar have ?thesis1
       by(auto simp add: thr.lookup-correct ws.lookup-correct α.step-thread-def step-thread-def split-beta
split: if-split-asm)
      moreover {
        let ?ta = (K$ [], [], [], [], [], convert-RA (snd (the (thr-lookup t (thr s)))))
        assume ?tso ?ta ⟶ ?inv ?ta
        hence ?thesis2 using None rrs
       by(auto simp add: thr.lookup-correct ws.lookup-correct α.step-thread-def step-thread-def split-beta
split: if-split-asm) }
      ultimately show ?thesis by blast
    next
      case (Some a)
      with rrs obtain ta x' m'
        where rrs: step-thread update-state s t =  ⌊(t', ⌊(ta, x', m')⌋, σ')⌋
        by(cases a) fastforce
      with invar have ?thesis1
        by (auto simp add: thr.lookup-correct ws.lookup-correct α.step-thread-def step-thread-def
          split-beta α.deterministic-THE [OF det, where s=state-α s, simplified]
          deterministic-THE2[OF det] holds-eq split: if-split-asm
          cong del: image-cong-simp) blast+
      moreover {
        assume ?tso ta ⟶ ?inv ta
        hence ?thesis2 using rrs invar
       by(auto simp add: thr.lookup-correct ws.lookup-correct α.step-thread-def step-thread-def split-beta
α.deterministic-THE[OF det, where s=state-α s, simplified] deterministic-THE2[OF det] holds-eq
split: if-split-asm)(auto simp add: α.actions-ok-iff)
      }
      ultimately show ?thesis by blast
    qed
  qed
  thus ?thesis1 (⋀ta. ?tso ta ⟹ ?inv ta) ⟹ ?thesis2 by blast+
qed

lemma step-thread-eq-None-conv:
  assumes det: α.deterministic I
  and invar: state-invar s state-α s ∈ I
  shows step-thread update-state s t = None ⟷ t ∉ α.active-threads (state-α s)
using assms step-thread-correct(1)[OF det - invar(1), of λ- -. True, of id update-state t]
by(simp add: map-option.id α.step-thread-eq-None-conv)
```

lemma *step-thread-Some-NoneD*:
assumes *det*: α.*deterministic I*
and *step*: *step-thread update-state s t'* = $\lfloor$(*t*, *None*, σ')$\rfloor$
and *invar*: *state-invar s state-*α *s* $\in$ *I*
shows $\exists$ *x ln n*. *thr-*α (*thr s*) *t* = $\lfloor$(*x*, *ln*)$\rfloor$ $\wedge$ *ln* $ *n* > *0* $\wedge$ $\neg$ *waiting* (*ws-*α (*wset s*) *t*) $\wedge$ *may-acquire-all* (*locks s*) *t ln* $\wedge$ σ' = *update-state* (*K*$ [], [], [], [], [], *convert-RA ln*)
using *assms step-thread-correct*(*1*)[*OF det - invar*(*1*), *of* λ- -. *True*, *of id update-state t'*]
by(*fastforce simp add*: *map-option.id dest*: α.*step-thread-Some-NoneD*[*OF sym*])

lemma *step-thread-Some-SomeD*:
assumes *det*: α.*deterministic I*
and *step*: *step-thread update-state s t'* = $\lfloor$(*t*, $\lfloor$(*ta*, *x'*, *m'*)$\rfloor$, σ')$\rfloor$
and *invar*: *state-invar s state-*α *s* $\in$ *I*
shows $\exists$ *x*. *thr-*α (*thr s*) *t* = $\lfloor$(*x*, *no-wait-locks*)$\rfloor$ $\wedge$ *Predicate.eval* (*r t* (*x*, *shr s*)) (*ta*, *x'*, *m'*) $\wedge$ *actions-ok s t ta* $\wedge$ σ' = *update-state ta*
using *assms step-thread-correct*(*1*)[*OF det - invar*(*1*), *of* λ- -. *True*, *of id update-state t'*]
by(*auto simp add*: *map-option.id dest*: α.*step-thread-Some-SomeD*[*OF det sym*])

end

9.2.2 Code Generator setup

lemmas [*code*] =
scheduler-base-aux.free-thread-id-def
scheduler-base-aux.redT-updT.simps
scheduler-base-aux.redT-updTs-def
scheduler-base-aux.thread-ok.simps
scheduler-base-aux.thread-oks.simps
scheduler-base-aux.wset-actions-ok-def
scheduler-base-aux.cond-action-ok.simps
scheduler-base-aux.cond-action-oks-def
scheduler-base-aux.redT-updI.simps
scheduler-base-aux.redT-updIs.simps
scheduler-base-aux.interrupt-action-ok.simps
scheduler-base-aux.interrupt-actions-ok.simps
scheduler-base-aux.actions-ok-def
scheduler-base-aux.step-thread-def

lemmas [*code*] =
scheduler-base.exec-updW.simps
scheduler-base.exec-updWs-def
scheduler-base.exec-upd-def
scheduler-base.execT-def
scheduler-base.exec-step.simps
scheduler-base.exec-aux-def
scheduler-base.exec-def

lemmas [*code*] =
scheduler-ext-base.active-threads.simps

lemma *singleton2-code* [*code*]:
singleton2 dfault (*Predicate.Seq f*) =
(*case f* () *of*

Predicate.Empty ⇒ *dfault* ()
| *Predicate.Insert x P* ⇒
if Predicate.is-empty P then x else Code.abort (*STR ''singleton2 not unique''*) (λ-. *singleton2 dfault* (*Predicate.Seq f*))
| *Predicate.Join P xq* ⇒
if Predicate.is-empty P then
the-only2 dfault xq
else if Predicate.null xq then singleton2 dfault P else Code.abort (*STR ''singleton2 not unique''*) (λ-. *singleton2 dfault* (*Predicate.Seq f*)))
unfolding *singleton2-def the-only2-def*
by(*auto simp only*: *singleton-code Code.abort-def split*: *seq.split if-split*)

lemma *the-only2-code* [*code*]:
the-only2 dfault Predicate.Empty = *Code.abort* (*STR ''the-only2 empty''*) *dfault*
the-only2 dfault (*Predicate.Insert x P*) =
(*if Predicate.is-empty P then x else Code.abort* (*STR ''the-only2 not unique''*) (λ-. *the-only2 dfault* (*Predicate.Insert x P*)))
the-only2 dfault (*Predicate.Join P xq*) =
(*if Predicate.is-empty P then*
the-only2 dfault xq
else if Predicate.null xq then
singleton2 dfault P
else
Code.abort (*STR ''the-only2 not unique''*) (λ-. *the-only2 dfault* (*Predicate.Join P xq*)))
unfolding *singleton2-def the-only2-def* **by** *simp-all*

lemma *the2-eq* [*code*]:
the2 A = *singleton2* (λ*x*. *Code.abort* (*STR ''not-unique''*) (λ-. *the2 A*)) *A*
unfolding *the2-def singleton2-def* **by**(*rule the-eq*)

end

9.3 Random scheduler

theory *Random-Scheduler*
imports
Scheduler
begin

type-synonym *random-scheduler* = *Random.seed*

abbreviation (*input*)
random-scheduler-invar :: *random-scheduler* ⇒ *'t set* ⇒ *bool*
where *random-scheduler-invar* ≡ λ- -. *True*

locale *random-scheduler-base* =
scheduler-ext-base
final r convert-RA
thr-α thr-invar thr-lookup thr-update thr-iterate
ws-α ws-invar ws-lookup ws-update ws-sel
is-α is-invar is-memb is-ins is-delete
thr'-α thr'-invar thr'-empty thr'-ins-dj
for *final* :: *'x* ⇒ *bool*

```
  and r :: 't ⇒ ('x × 'm) ⇒ (('l,'t,'x,'m,'w,'o) thread-action × 'x × 'm) Predicate.pred
  and convert-RA :: 'l released-locks ⇒ 'o list
  and output :: random-scheduler ⇒ 't ⇒ ('l,'t,'x,'m,'w,'o) thread-action ⇒ 'q option
  and thr-α :: 'm-t ⇒ ('l,'t,'x) thread-info
  and thr-invar :: 'm-t ⇒ bool
  and thr-lookup :: 't ⇒ 'm-t ⇀ ('x × 'l released-locks)
  and thr-update :: 't ⇒ 'x × 'l released-locks ⇒ 'm-t ⇒ 'm-t
  and thr-iterate :: 'm-t ⇒ ('t × ('x × 'l released-locks), 's-t) set-iterator
  and ws-α :: 'm-w ⇒ ('w,'t) wait-sets
  and ws-invar :: 'm-w ⇒ bool
  and ws-lookup :: 't ⇒ 'm-w ⇀ 'w wait-set-status
  and ws-update :: 't ⇒ 'w wait-set-status ⇒ 'm-w ⇒ 'm-w
  and ws-delete :: 't ⇒ 'm-w ⇒ 'm-w
  and ws-iterate :: 'm-w ⇒ ('t × 'w wait-set-status, 'm-w) set-iterator
  and ws-sel :: 'm-w ⇒ ('t × 'w wait-set-status ⇒ bool) ⇀ ('t × 'w wait-set-status)
  and is-α :: 's-i ⇒ 't interrupts
  and is-invar :: 's-i ⇒ bool
  and is-memb :: 't ⇒ 's-i ⇒ bool
  and is-ins :: 't ⇒ 's-i ⇒ 's-i
  and is-delete :: 't ⇒ 's-i ⇒ 's-i
  and thr'-α :: 's-t ⇒ 't set
  and thr'-invar :: 's-t ⇒ bool
  and thr'-empty :: unit ⇒ 's-t
  and thr'-ins-dj :: 't ⇒ 's-t ⇒ 's-t
  +
  fixes thr'-to-list :: 's-t ⇒ 't list
begin

definition next-thread :: random-scheduler ⇒ 's-t ⇒ ('t × random-scheduler) option
where
  next-thread seed active =
  (let ts = thr'-to-list active
   in if ts = [] then None else Some (Random.select (thr'-to-list active) seed))

definition random-scheduler :: ('l,'t,'x,'m,'w,'o,'m-t,'m-w,'s-i,random-scheduler) scheduler
where
  random-scheduler seed s =
   (do {
      (t, seed') ← next-thread seed (active-threads s);
      step-thread (λta. seed') s t
   })

end

locale random-scheduler =
  random-scheduler-base
    final r convert-RA output
    thr-α thr-invar thr-lookup thr-update thr-iterate
    ws-α ws-invar ws-lookup ws-update ws-delete ws-iterate ws-sel
    is-α is-invar is-memb is-ins is-delete
    thr'-α thr'-invar thr'-empty thr'-ins-dj thr'-to-list
  +
  scheduler-ext-aux
    final r convert-RA
```

thr-α thr-invar thr-lookup thr-update thr-iterate
ws-α ws-invar ws-lookup ws-update ws-sel
is-α is-invar is-memb is-ins is-delete
thr'-α thr'-invar thr'-empty thr'-ins-dj
+
ws: *map-delete ws-α ws-invar ws-delete* +
ws: *map-iteratei ws-α ws-invar ws-iterate* +
thr': *set-to-list thr'-α thr'-invar thr'-to-list*
for *final* :: *'x ⇒ bool*
and *r* :: *'t ⇒ ('x × 'm) ⇒ (('l,'t,'x,'m,'w,'o) thread-action × 'x × 'm) Predicate.pred*
and *convert-RA* :: *'l released-locks ⇒ 'o list*
and *output* :: *random-scheduler ⇒ 't ⇒ ('l,'t,'x,'m,'w,'o) thread-action ⇒ 'q option*
and *thr-α* :: *'m-t ⇒ ('l,'t,'x) thread-info*
and *thr-invar* :: *'m-t ⇒ bool*
and *thr-lookup* :: *'t ⇒ 'm-t ⇀ ('x × 'l released-locks)*
and *thr-update* :: *'t ⇒ 'x × 'l released-locks ⇒ 'm-t ⇒ 'm-t*
and *thr-iterate* :: *'m-t ⇒ ('t × ('x × 'l released-locks), 's-t) set-iterator*
and *ws-α* :: *'m-w ⇒ ('w,'t) wait-sets*
and *ws-invar* :: *'m-w ⇒ bool*
and *ws-lookup* :: *'t ⇒ 'm-w ⇀ 'w wait-set-status*
and *ws-update* :: *'t ⇒ 'w wait-set-status ⇒ 'm-w ⇒ 'm-w*
and *ws-delete* :: *'t ⇒ 'm-w ⇒ 'm-w*
and *ws-iterate* :: *'m-w ⇒ ('t × 'w wait-set-status, 'm-w) set-iterator*
and *ws-sel* :: *'m-w ⇒ ('t × 'w wait-set-status ⇒ bool) ⇀ ('t × 'w wait-set-status)*
and *is-α* :: *'s-i ⇒ 't interrupts*
and *is-invar* :: *'s-i ⇒ bool*
and *is-memb* :: *'t ⇒ 's-i ⇒ bool*
and *is-ins* :: *'t ⇒ 's-i ⇒ 's-i*
and *is-delete* :: *'t ⇒ 's-i ⇒ 's-i*
and *thr'-α* :: *'s-t ⇒ 't set*
and *thr'-invar* :: *'s-t ⇒ bool*
and *thr'-empty* :: *unit ⇒ 's-t*
and *thr'-ins-dj* :: *'t ⇒ 's-t ⇒ 's-t*
and *thr'-to-list* :: *'s-t ⇒ 't list*
begin

lemma *next-thread-eq-None-iff*:
assumes *thr'-invar active random-scheduler-invar seed T*
shows *next-thread seed active = None ⟷ thr'-α active = {}*
using *thr'.to-list-correct*[*OF assms*(*1*)]
by(*auto simp add*: *next-thread-def neq-Nil-conv*)

lemma *next-thread-eq-SomeD*:
assumes *next-thread seed active = Some* (*t*, *seed'*)
and *thr'-invar active random-scheduler-invar seed T*
shows *t ∈ thr'-α active*
using *assms*
by(*auto simp add*: *next-thread-def thr'.to-list-correct split*: *if-split-asm dest*: *select*[*of - seed*])

lemma *random-scheduler-spec*:
assumes *det*: *α.deterministic I*
shows *scheduler-spec final r random-scheduler random-scheduler-invar thr-α thr-invar ws-α ws-invar is-α is-invar I*
proof

fix σ s
assume *rr*: *random-scheduler* σ s = *None*
and *invar*: *random-scheduler-invar* σ (*dom* (*thr-α* (*thr s*))) *state-invar s state-α s* ∈ *I*
from *invar*(*2*) **have** *thr'-invar* (*active-threads s*) **by**(*rule active-threads-correct*)
thus *α.active-threads* (*state-α s*) = {} **using** *rr invar*
by(*auto simp add*: *random-scheduler-def Option-bind-eq-None-conv next-thread-eq-None-iff step-thread-eq-None-conv*[*OF det*] *dest*: *next-thread-eq-SomeD*)
next
fix σ s t σ'
assume *rr*: *random-scheduler* σ s = ⌊(*t*, *None*, σ')⌋
and *invar*: *random-scheduler-invar* σ (*dom* (*thr-α* (*thr s*))) *state-invar s state-α s* ∈ *I*
thus ∃ *x ln n*. *thr-α* (*thr s*) *t* = ⌊(*x*, *ln*)⌋ ∧ *0* < *ln* $ *n* ∧ ¬ *waiting* (*ws-α* (*wset s*) *t*) ∧ *may-acquire-all* (*locks s*) *t ln*
by(*fastforce simp add*: *random-scheduler-def bind-eq-Some-conv dest*: *step-thread-Some-NoneD*[*OF det*])
next
fix σ s t *ta* x' m' σ'
assume *rr*: *random-scheduler* σ s = ⌊(*t*, ⌊(*ta*, x', m')⌋, σ')⌋
and *invar*: *random-scheduler-invar* σ (*dom* (*thr-α* (*thr s*))) *state-invar s state-α s* ∈ *I*
thus ∃ *x*. *thr-α* (*thr s*) *t* = ⌊(*x*, *no-wait-locks*)⌋ ∧ *Predicate.eval* (*r t* (*x*, *shr s*)) (*ta*, x', m') ∧ *α.actions-ok* (*state-α s*) *t ta*
by(*auto simp add*: *random-scheduler-def bind-eq-Some-conv dest*: *step-thread-Some-SomeD*[*OF det*])
qed *simp-all*

end

sublocale *random-scheduler-base* <
scheduler-base
final r convert-RA
random-scheduler output pick-wakeup-via-sel (λ*s P*. *ws-sel s* (λ(*k*,*v*). *P k v*)) *random-scheduler-invar*
thr-α thr-invar thr-lookup thr-update
ws-α ws-invar ws-lookup ws-update ws-delete ws-iterate
is-α is-invar is-memb is-ins is-delete
for *n0* **.**

sublocale *random-scheduler* <
pick-wakeup-spec
final r convert-RA
pick-wakeup-via-sel (λ*s P*. *ws-sel s* (λ(*k*,*v*). *P k v*)) *random-scheduler-invar*
thr-α thr-invar
ws-α ws-invar
is-α is-invar
by(*rule pick-wakeup-spec-via-sel*)(*unfold-locales*)

context *random-scheduler* **begin**

lemma *random-scheduler-scheduler*:
assumes *det*: *α.deterministic I*
shows
scheduler
final r convert-RA
random-scheduler (*pick-wakeup-via-sel* (λ*s P*. *ws-sel s* (λ(*k*,*v*). *P k v*))) *random-scheduler-invar*
thr-α thr-invar thr-lookup thr-update
ws-α ws-invar ws-lookup ws-update ws-delete ws-iterate

is-α is-invar is-memb is-ins is-delete
I
proof −
interpret *scheduler-spec*
final r convert-RA
random-scheduler random-scheduler-invar
thr-α thr-invar
ws-α ws-invar
is-α is-invar
I
using *det* **by**(*rule random-scheduler-spec*)

show *?thesis* **by**(*unfold-locales*)(*rule α.deterministic-invariant3p*[*OF det*])
qed

end

9.3.1 Code generator setup

lemmas [*code*] =
random-scheduler-base.next-thread-def
random-scheduler-base.random-scheduler-def

end

9.4 Round robin scheduler

theory *Round-Robin*
imports
Scheduler
begin

A concrete scheduler must pick one possible reduction step from the small-step semantics for invidivual threads. Currently, this is only possible if there is only one such by using *Predicate.the*.

9.4.1 Concrete schedulers

9.4.2 Round-robin schedulers

type-synonym *'queue round-robin* = *'queue* × *nat*
— Waiting queue of threads and remaining number of steps of the first thread until it has to return resources

primrec *enqueue-new-thread* :: *'t list* ⇒ (*'t*,*'x*,*'m*) *new-thread-action* ⇒ *'t list*
where
enqueue-new-thread queue (*NewThread t x m*) = *queue* @ [*t*]
| *enqueue-new-thread queue* (*ThreadExists t b*) = *queue*

definition *enqueue-new-threads* :: *'t list* ⇒ (*'t*,*'x*,*'m*) *new-thread-action list* ⇒ *'t list*
where
enqueue-new-threads = *foldl enqueue-new-thread*

primrec *round-robin-update-state* :: *nat* ⇒ *'t list round-robin* ⇒ *'t* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* ⇒ *'t list round-robin*
where
round-robin-update-state n0 (*queue*, *n*) *t ta* =
(*let queue'* = *enqueue-new-threads queue* ⦃*ta*⦄$_t$
in if n = *0* ∨ *Yield* ∈ *set* ⦃*ta*⦄$_c$ *then* (*rotate1 queue'*, *n0*) *else* (*queue'*, *n* − *1*))

context *multithreaded-base* **begin**

abbreviation *round-robin-step* :: *nat* ⇒ *'t list round-robin* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*) *state* ⇒ *'t* ⇒ (*'t* × ((*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* × *'x* × *'m*) *option* × *'t list round-robin*) *option*
where
round-robin-step n0 σ *s t* ≡ *step-thread* (*round-robin-update-state n0* σ *t*) *s t*

partial-function (*option*) *round-robin-reschedule* :: *'t* ⇒
't list ⇒ *nat* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*) *state* ⇒ (*'t* × ((*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* × *'x* × *'m*) *option* × *'t list round-robin*) *option*
where
round-robin-reschedule t0 queue n0 s =
(*let*
t = *hd queue*;
queue' = *tl queue*
in
if t = *t0 then*
None
else
case round-robin-step n0 (*t* # *queue'*, *n0*) *s t of*
None ⇒ *round-robin-reschedule t0* (*queue'* @ [*t*]) *n0 s*
| ⌊*ttaxmσ*⌋ ⇒ ⌊*ttaxmσ*⌋)

fun *round-robin* :: *nat* ⇒ *'t list round-robin* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*) *state* ⇒ (*'t* × ((*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* × *'x* × *'m*) *option* × *'t list round-robin*) *option*
where
round-robin n0 ([], *n*) *s* = *None*
| *round-robin n0* (*t* # *queue*, *n*) *s* =
(*case round-robin-step n0* (*t* # *queue*, *n*) *s t of*
⌊*ttaxmσ*⌋ ⇒ ⌊*ttaxmσ*⌋
| *None* ⇒ *round-robin-reschedule t* (*queue* @ [*t*]) *n0 s*)

end

primrec *round-robin-invar* :: *'t list round-robin* ⇒ *'t set* ⇒ *bool*
where *round-robin-invar* (*queue*, *n*) *T* ⟷ *set queue* = *T* ∧ *distinct queue*

lemma *set-enqueue-new-thread*:
set (*enqueue-new-thread queue nta*) = *set queue* ∪ {*t*. ∃ *x m*. *nta* = *NewThread t x m*}
by(*cases nta*) *auto*

lemma *set-enqueue-new-threads*:
set (*enqueue-new-threads queue ntas*) = *set queue* ∪ {*t*. ∃ *x m*. *NewThread t x m* ∈ *set ntas*}
apply(*induct ntas arbitrary*: *queue*)
apply(*auto simp add*: *enqueue-new-threads-def set-enqueue-new-thread*)
done

lemma *enqueue-new-thread-eq-Nil* [*simp*]:
enqueue-new-thread queue nta = [] ⟷ *queue* = [] ∧ (∃ *t b*. *nta* = *ThreadExists t b*)
by(*cases nta*) *simp-all*

lemma *enqueue-new-threads-eq-Nil* [*simp*]:
enqueue-new-threads queue ntas = [] ⟷ *queue* = [] ∧ *set ntas* ⊆ {*ThreadExists t b*|*t b*. *True*}
apply(*induct ntas arbitrary*: *queue*)
apply(*auto simp add*: *enqueue-new-threads-def*)
done

lemma *distinct-enqueue-new-threads*:
fixes *ts* :: ('*l*,'*t*,'*x*) *thread-info*
and *ntas* :: ('*t*,'*x*,'*m*) *new-thread-action list*
assumes *thread-oks ts ntas set queue* = *dom ts distinct queue*
shows *distinct* (*enqueue-new-threads queue ntas*)
using *assms*
proof(*induct ntas arbitrary*: *ts queue*)
case *Nil* **thus** *?case* **by**(*simp add*: *enqueue-new-threads-def*)
next
case (*Cons nt ntas*)
from ‹*thread-oks ts* (*nt* # *ntas*)›
have *thread-ok ts nt* **and** *thread-oks* (*redT-updT ts nt*) *ntas* **by** *simp-all*
from ‹*thread-ok ts nt*› ‹*set queue* = *dom ts*› ‹*distinct queue*›
have *set* (*enqueue-new-thread queue nt*) = *dom* (*redT-updT ts nt*) ∧ *distinct* (*enqueue-new-thread queue nt*)
by(*cases nt*)(*auto*)
with ‹*thread-oks* (*redT-updT ts nt*) *ntas*›
have *distinct* (*enqueue-new-threads* (*enqueue-new-thread queue nt*) *ntas*)
by(*blast intro*: *Cons.hyps*)
thus *?case* **by**(*simp add*: *enqueue-new-threads-def*)
qed

lemma *round-robin-reschedule-induct* [*consumes 1*, *case-names head rotate*]:
assumes *major*: *t0* ∈ *set queue*
and *head*: ⋀*queue*. *P* (*t0* # *queue*)
and *rotate*: ⋀*queue t*. ⟦ *t* ≠ *t0*; *t0* ∈ *set queue*; *P* (*queue* @ [*t*]) ⟧ ⟹ *P* (*t* # *queue*)
shows *P queue*
using *major*
proof(*induct n*≡*length* (*takeWhile* (λ*x*. *x*≠*t0*) *queue*) *arbitrary*: *queue*)
case *0*
then obtain *queue*′ **where** *queue* = *t0* # *queue*′
by(*cases queue*)(*auto split*: *if-split-asm*)
thus *?case* **by**(*simp add*: *head*)
next
case (*Suc n*)
then obtain *t queue*′ **where** [*simp*]: *queue* = *t* # *queue*′
and *t*: *t* ≠ *t0* **and** *n*: *n* = *length* (*takeWhile* (λ*x*. *x* ≠ *t0*) *queue*′)
and *t0*: *t0* ∈ *set queue*′
by(*cases queue*)(*auto split*: *if-split-asm*)
from *n t0* **have** *n* = *length* (*takeWhile* (λ*x*. *x* ≠ *t0*) (*queue*′ @ [*t*])) **by**(*simp*)
moreover from *t0* **have** *t0* ∈ *set* (*queue*′ @ [*t*]) **by** *simp*
ultimately have *P* (*queue*′ @ [*t*]) **by**(*rule Suc.hyps*)
with *t t0* **show** *?case* **by**(*simp add*: *rotate*)
qed

context *multithreaded-base* **begin**

declare *actions-ok-iff* [*simp del*]
declare *actions-ok.cases* [*rule del*]

lemma *round-robin-step-invar-None*:
⟦ *round-robin-step n0 σ s t′ = ⌊(t, None, σ′)⌋*; *round-robin-invar σ (dom (thr s))* ⟧
⟹ *round-robin-invar σ′ (dom (thr s))*
by(*cases σ*)(*auto dest*: *step-thread-Some-NoneD simp add*: *set-enqueue-new-threads distinct-enqueue-new-threads*)

lemma *round-robin-step-invar-Some*:
⟦ *deterministic I*; *round-robin-step n0 σ s t′ = ⌊(t, ⌊(ta, x′, m′)⌋, σ′)⌋*; *round-robin-invar σ (dom (thr s))*; *s ∈ I* ⟧
⟹ *round-robin-invar σ′ (dom (thr s) ∪ {t. ∃ x m. NewThread t x m ∈ set ⦃ta⦄t})*
apply(*cases σ*)
apply *clarsimp*
apply(*frule* (*1*) *step-thread-Some-SomeD*)
apply(*auto split*: *if-split-asm simp add*: *split-beta set-enqueue-new-threads deterministic-THE*)
apply(*auto simp add*: *actions-ok-iff distinct-enqueue-new-threads*)
done

lemma *round-robin-reschedule-Cons*:
round-robin-reschedule t0 (t0 # queue) n0 s = None
t ≠ t0 ⟹ round-robin-reschedule t0 (t # queue) n0 s =
(case round-robin-step n0 (t # queue, n0) s t of
None ⇒ round-robin-reschedule t0 (queue @ [t]) n0 s
| Some ttaxmσ ⇒ Some ttaxmσ)
by(*simp-all add*: *round-robin-reschedule.simps*)

lemma *round-robin-reschedule-NoneD*:
assumes *rrr*: *round-robin-reschedule t0 queue n0 s = None*
and *t0*: *t0 ∈ set queue*
shows *set (takeWhile (λt′. t′ ≠ t0) queue) ∩ active-threads s = {}*
using *t0 rrr*
proof(*induct queue rule*: *round-robin-reschedule-induct*)
case (*head queue*)
thus *?case* **by** *simp*
next
case (*rotate queue t*)
from ‹*round-robin-reschedule t0 (t # queue) n0 s = None*› ‹*t ≠ t0*›
have *round-robin-step n0 (t # queue, n0) s t = None*
and *round-robin-reschedule t0 (queue @ [t]) n0 s = None*
by(*simp-all add*: *round-robin-reschedule-Cons*)
from *this*(*1*) **have** *t ∉ active-threads s* **by**(*rule step-thread-NoneD*)
moreover from ‹*round-robin-reschedule t0 (queue @ [t]) n0 s = None*›
have *set (takeWhile (λt′. t′ ≠ t0) (queue @ [t])) ∩ active-threads s = {}*
by(*rule rotate.hyps*)
moreover have *takeWhile (λt′. t′ ≠ t0) (queue @ [t]) = takeWhile (λt′. t′ ≠ t0) queue*
using ‹*t0 ∈ set queue*› **by** *simp*
ultimately show *?case* **using** ‹*t ≠ t0*› **by** *simp*
qed

lemma *round-robin-reschedule-Some-NoneD*:

assumes *rrr*: *round-robin-reschedule t0 queue n0 s* = ⌊(*t*, *None*, *σ′*)⌋
and *t0*: *t0* ∈ *set queue*
shows ∃ *x ln n*. *thr s t* = ⌊(*x*, *ln*)⌋ ∧ *ln* $ *n* > *0* ∧ ¬ *waiting* (*wset s t*) ∧ *may-acquire-all* (*locks s*) *t ln*
using *t0 rrr*
proof(*induct queue rule*: *round-robin-reschedule-induct*)
case *head* **thus** *?case* **by**(*simp add*: *round-robin-reschedule-Cons*)
next
case (*rotate queue t′*)
show *?case*
proof(*cases round-robin-step n0* (*t′* # *queue*, *n0*) *s t′*)
case *None*
with ‹*round-robin-reschedule t0* (*t′* # *queue*) *n0 s* = ⌊(*t*, *None*, *σ′*)⌋› ‹*t′* ≠ *t0*›
have *round-robin-reschedule t0* (*queue* @ [*t′*]) *n0 s* = ⌊(*t*, *None*, *σ′*)⌋
by(*simp add*: *round-robin-reschedule-Cons*)
thus *?thesis* **by**(*rule rotate.hyps*)
next
case (*Some a*)
with ‹*round-robin-reschedule t0* (*t′* # *queue*) *n0 s* = ⌊(*t*, *None*, *σ′*)⌋› ‹*t′* ≠ *t0*›
have *round-robin-step n0* (*t′* # *queue*, *n0*) *s t′* = ⌊(*t*, *None*, *σ′*)⌋
by(*simp add*: *round-robin-reschedule-Cons*)
thus *?thesis* **by**(*blast dest*: *step-thread-Some-NoneD*)
qed
qed

lemma *round-robin-reschedule-Some-SomeD*:
assumes *deterministic I*
and *rrr*: *round-robin-reschedule t0 queue n0 s* = ⌊(*t*, ⌊(*ta*, *x′*, *m′*)⌋, *σ′*)⌋
and *t0*: *t0* ∈ *set queue*
and *I*: *s* ∈ *I*
shows ∃ *x*. *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ ∧ *t* ⊢ ⟨*x*, *shr s*⟩ −*ta*→ ⟨*x′*, *m′*⟩ ∧ *actions-ok s t ta*
using *t0 rrr*
proof(*induct queue rule*: *round-robin-reschedule-induct*)
case *head* **thus** *?case* **by**(*simp add*: *round-robin-reschedule-Cons*)
next
case (*rotate queue t′*)
show *?case*
proof(*cases round-robin-step n0* (*t′* # *queue*, *n0*) *s t′*)
case *None*
with ‹*round-robin-reschedule t0* (*t′* # *queue*) *n0 s* = ⌊(*t*, ⌊(*ta*, *x′*, *m′*)⌋, *σ′*)⌋› ‹*t′* ≠ *t0*›
have *round-robin-reschedule t0* (*queue* @ [*t′*]) *n0 s* = ⌊(*t*, ⌊(*ta*, *x′*, *m′*)⌋, *σ′*)⌋
by(*simp add*: *round-robin-reschedule-Cons*)
thus *?thesis* **by**(*rule rotate.hyps*)
next
case (*Some a*)
with ‹*round-robin-reschedule t0* (*t′* # *queue*) *n0 s* = ⌊(*t*, ⌊(*ta*, *x′*, *m′*)⌋, *σ′*)⌋› ‹*t′* ≠ *t0*›
have *round-robin-step n0* (*t′* # *queue*, *n0*) *s t′* = ⌊(*t*, ⌊(*ta*, *x′*, *m′*)⌋, *σ′*)⌋
by(*simp add*: *round-robin-reschedule-Cons*)
thus *?thesis* **using** *I* **by**(*blast dest*: *step-thread-Some-SomeD*[*OF* ‹*deterministic I*›])
qed
qed

lemma *round-robin-reschedule-invar-None*:
assumes *rrr*: *round-robin-reschedule t0 queue n0 s* = ⌊(*t*, *None*, *σ′*)⌋

```
  and invar: round-robin-invar (queue, n0) (dom (thr s))
  and t0: t0 ∈ set queue
  shows round-robin-invar σ' (dom (thr s))
using t0 rrr invar
proof(induct queue rule: round-robin-reschedule-induct)
  case head thus ?case by(simp add: round-robin-reschedule-Cons)
next
  case (rotate queue t')
  show ?case
  proof(cases round-robin-step n0 (t' # queue, n0) s t')
    case None
    with ‹round-robin-reschedule t0 (t' # queue) n0 s = ⌊(t, None, σ')⌋› ‹t' ≠ t0›
    have round-robin-reschedule t0 (queue @ [t']) n0 s = ⌊(t, None, σ')⌋
      by(simp add: round-robin-reschedule-Cons)
    moreover from ‹round-robin-invar (t' # queue, n0) (dom (thr s))›
    have round-robin-invar (queue @ [t'], n0) (dom (thr s)) by simp
    ultimately show ?thesis by(rule rotate.hyps)
  next
    case (Some a)
    with ‹round-robin-reschedule t0 (t' # queue) n0 s = ⌊(t, None, σ')⌋› ‹t' ≠ t0›
    have round-robin-step n0 (t' # queue, n0) s t' = ⌊(t, None, σ')⌋
      by(simp add: round-robin-reschedule-Cons)
    thus ?thesis using ‹round-robin-invar (t' # queue, n0) (dom (thr s))›
      by(rule round-robin-step-invar-None)
  qed
qed

lemma round-robin-reschedule-invar-Some:
  assumes deterministic I
  and rrr: round-robin-reschedule t0 queue n0 s = ⌊(t, ⌊(ta, x', m')⌋, σ')⌋
  and invar: round-robin-invar (queue, n0) (dom (thr s))
  and t0: t0 ∈ set queue
  and s ∈ I
  shows round-robin-invar σ' (dom (thr s) ∪ {t. ∃ x m. NewThread t x m ∈ set {|ta|}_t})
using t0 rrr invar
proof(induct queue rule: round-robin-reschedule-induct)
  case head thus ?case by(simp add: round-robin-reschedule-Cons)
next
  case (rotate queue t')
  show ?case
  proof(cases round-robin-step n0 (t' # queue, n0) s t')
    case None
    with ‹round-robin-reschedule t0 (t' # queue) n0 s = ⌊(t, ⌊(ta, x', m')⌋, σ')⌋› ‹t' ≠ t0›
    have round-robin-reschedule t0 (queue @ [t']) n0 s = ⌊(t, ⌊(ta, x', m')⌋, σ')⌋
      by(simp add: round-robin-reschedule-Cons)
    moreover from ‹round-robin-invar (t' # queue, n0) (dom (thr s))›
    have round-robin-invar (queue @ [t'], n0) (dom (thr s)) by simp
    ultimately show ?thesis by(rule rotate.hyps)
  next
    case (Some a)
    with ‹round-robin-reschedule t0 (t' # queue) n0 s = ⌊(t, ⌊(ta, x', m')⌋, σ')⌋› ‹t' ≠ t0›
    have round-robin-step n0 (t' # queue, n0) s t' = ⌊(t, ⌊(ta, x', m')⌋, σ')⌋
      by(simp add: round-robin-reschedule-Cons)
    thus ?thesis using ‹round-robin-invar (t' # queue, n0) (dom (thr s))› ‹s ∈ I›
```

```
    by(rule round-robin-step-invar-Some[OF ‹deterministic I›])
  qed
qed

lemma round-robin-NoneD:
  assumes rr: round-robin n0 σ s = None
  and invar: round-robin-invar σ (dom (thr s))
  shows active-threads s = {}
proof −
  obtain queue n where σ: σ = (queue, n) by(cases σ)
  show ?thesis
  proof(cases queue)
    case Nil
    thus ?thesis using invar σ by(fastforce elim: active-threads.cases)
  next
    case (Cons t queue')
    with rr σ have round-robin-step n0 (t # queue', n) s t = None
      and round-robin-reschedule t (queue' @ [t]) n0 s = None by simp-all
    from ‹round-robin-step n0 (t # queue', n) s t = None›
    have t ∉ active-threads s by(rule step-thread-NoneD)
    moreover from ‹round-robin-reschedule t (queue' @ [t]) n0 s = None›
    have set (takeWhile (λx. x ≠ t) (queue' @ [t])) ∩ active-threads s = {}
      by(rule round-robin-reschedule-NoneD) simp
    moreover from invar σ Cons
    have takeWhile (λx. x ≠ t) (queue' @ [t]) = queue'
      by(subst takeWhile-append2) auto
    moreover from invar have active-threads s ⊆ set queue
      using σ by(auto elim: active-threads.cases)
    ultimately show ?thesis using Cons by auto
  qed
qed

lemma round-robin-Some-NoneD:
  assumes rr: round-robin n0 σ s = ⌊(t, None, σ')⌋
  shows ∃x ln n. thr s t = ⌊(x, ln)⌋ ∧ ln $ n > 0 ∧ ¬ waiting (wset s t) ∧ may-acquire-all (locks s)
t ln
proof −
  obtain queue n where σ: σ = (queue, n) by(cases σ)
  with rr have queue ≠ [] by clarsimp
  then obtain t' queue' where queue: queue = t' # queue'
    by(auto simp add: neq-Nil-conv)
  show ?thesis
  proof(cases round-robin-step n0 (t' # queue', n) s t')
    case (Some a)
    with rr queue σ have round-robin-step n0 (t' # queue', n) s t' = ⌊(t, None, σ')⌋ by simp
    thus ?thesis by(blast dest: step-thread-Some-NoneD)
  next
    case None
    with rr queue σ have round-robin-reschedule t' (queue' @ [t']) n0 s = ⌊(t, None, σ')⌋ by simp
    thus ?thesis by(rule round-robin-reschedule-Some-NoneD)simp
  qed
qed

lemma round-robin-Some-SomeD:
```

```
  assumes deterministic I
  and rr: round-robin n0 σ s = ⌊(t, ⌊(ta, x', m')⌋, σ')⌋
  and s ∈ I
  shows ∃x. thr s t = ⌊(x, no-wait-locks)⌋ ∧ t ⊢ ⟨x, shr s⟩ −ta→ ⟨x', m'⟩ ∧ actions-ok s t ta
proof −
  obtain queue n where σ: σ = (queue, n) by(cases σ)
  with rr have queue ≠ [] by clarsimp
  then obtain t' queue' where queue: queue = t' # queue'
    by(auto simp add: neq-Nil-conv)
  show ?thesis
  proof(cases round-robin-step n0 (t' # queue', n) s t')
    case (Some a)
    with rr queue σ have round-robin-step n0 (t' # queue', n) s t' = ⌊(t, ⌊(ta, x', m')⌋, σ')⌋ by simp
    thus ?thesis using ‹s ∈ I› by(blast dest: step-thread-Some-SomeD[OF ‹deterministic I›])
  next
    case None
    with rr queue σ have round-robin-reschedule t' (queue' @ [t']) n0 s = ⌊(t, ⌊(ta, x', m')⌋, σ')⌋ by
simp
    thus ?thesis by(rule round-robin-reschedule-Some-SomeD[OF ‹deterministic I›])(simp-all add: ‹s
∈ I›)
  qed
qed

lemma round-robin-invar-None:
  assumes rr: round-robin n0 σ s = ⌊(t, None, σ')⌋
  and invar: round-robin-invar σ (dom (thr s))
  shows round-robin-invar σ' (dom (thr s))
proof −
  obtain queue n where σ: σ = (queue, n) by(cases σ)
  with rr have queue ≠ [] by clarsimp
  then obtain t' queue' where queue: queue = t' # queue'
    by(auto simp add: neq-Nil-conv)
  show ?thesis
  proof(cases round-robin-step n0 (t' # queue', n) s t')
    case (Some a)
    with rr queue σ have round-robin-step n0 (t' # queue', n) s t' = ⌊(t, None, σ')⌋ by simp
    thus ?thesis using invar unfolding σ queue by(rule round-robin-step-invar-None)
  next
    case None
    with rr queue σ have round-robin-reschedule t' (queue' @ [t']) n0 s = ⌊(t, None, σ')⌋ by simp
    moreover from invar queue σ have round-robin-invar (queue' @ [t'], n0) (dom (thr s)) by simp
    ultimately show ?thesis by(rule round-robin-reschedule-invar-None) simp
  qed
qed

lemma round-robin-invar-Some:
  assumes deterministic I
  and rr: round-robin n0 σ s = ⌊(t, ⌊(ta, x', m')⌋, σ')⌋
  and invar: round-robin-invar σ (dom (thr s)) s ∈ I
  shows round-robin-invar σ' (dom (thr s) ∪ {t. ∃x m. NewThread t x m ∈ set {|ta|}_t})
proof −
  obtain queue n where σ: σ = (queue, n) by(cases σ)
  with rr have queue ≠ [] by clarsimp
  then obtain t' queue' where queue: queue = t' # queue'
```

by(*auto simp add*: *neq-Nil-conv*)
show *?thesis*
proof(*cases round-robin-step n0* (*t′ # queue′*, *n*) *s t′*)
case (*Some a*)
with *rr queue σ* **have** *round-robin-step n0* (*t′ # queue′*, *n*) *s t′* = ⌊(*t*, ⌊(*ta*, *x′*, *m′*)⌋, *σ′*)⌋ **by** *simp*
thus *?thesis* **using** *invar* **unfolding** *σ queue* **by**(*rule round-robin-step-invar-Some*[*OF* ‹*deterministic I*›])
next
case *None*
with *rr queue σ* **have** *round-robin-reschedule t′* (*queue′* @ [*t′*]) *n0 s* = ⌊(*t*, ⌊(*ta*, *x′*, *m′*)⌋, *σ′*)⌋ **by** *simp*
moreover from *invar queue σ*
have *round-robin-invar* (*queue′* @ [*t′*], *n0*) (*dom* (*thr s*)) **by** *simp*
ultimately show *?thesis* **by**(*rule round-robin-reschedule-invar-Some*[*OF* ‹*deterministic I*›])(*simp-all add*: ‹*s* ∈ *I*›)
qed
qed

end

locale *round-robin-base* =
scheduler-base-aux
final r convert-RA
thr-α thr-invar thr-lookup thr-update
ws-α ws-invar ws-lookup
is-α is-invar is-memb is-ins is-delete
for *final* :: *'x* ⇒ *bool*
and *r* :: *'t* ⇒ (*'x* × *'m*) ⇒ ((*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* × *'x* × *'m*) *Predicate.pred*
and *convert-RA* :: *'l released-locks* ⇒ *'o list*
and *output* :: *'queue round-robin* ⇒ *'t* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* ⇒ *'q option*
and *thr-α* :: *'m-t* ⇒ (*'l*,*'t*,*'x*) *thread-info*
and *thr-invar* :: *'m-t* ⇒ *bool*
and *thr-lookup* :: *'t* ⇒ *'m-t* ⇀ (*'x* × *'l released-locks*)
and *thr-update* :: *'t* ⇒ *'x* × *'l released-locks* ⇒ *'m-t* ⇒ *'m-t*
and *ws-α* :: *'m-w* ⇒ (*'w*,*'t*) *wait-sets*
and *ws-invar* :: *'m-w* ⇒ *bool*
and *ws-lookup* :: *'t* ⇒ *'m-w* ⇀ *'w wait-set-status*
and *ws-update* :: *'t* ⇒ *'w wait-set-status* ⇒ *'m-w* ⇒ *'m-w*
and *ws-delete* :: *'t* ⇒ *'m-w* ⇒ *'m-w*
and *ws-iterate* :: *'m-w* ⇒ (*'t* × *'w wait-set-status*, *'m-w*) *set-iterator*
and *ws-sel* :: *'m-w* ⇒ (*'t* × *'w wait-set-status* ⇒ *bool*) ⇀ (*'t* × *'w wait-set-status*)
and *is-α* :: *'s-i* ⇒ *'t interrupts*
and *is-invar* :: *'s-i* ⇒ *bool*
and *is-memb* :: *'t* ⇒ *'s-i* ⇒ *bool*
and *is-ins* :: *'t* ⇒ *'s-i* ⇒ *'s-i*
and *is-delete* :: *'t* ⇒ *'s-i* ⇒ *'s-i*
+
fixes *queue-α* :: *'queue* ⇒ *'t list*
and *queue-invar* :: *'queue* ⇒ *bool*
and *queue-empty* :: *unit* ⇒ *'queue*
and *queue-isEmpty* :: *'queue* ⇒ *bool*
and *queue-enqueue* :: *'t* ⇒ *'queue* ⇒ *'queue*
and *queue-dequeue* :: *'queue* ⇒ *'t* × *'queue*
and *queue-push* :: *'t* ⇒ *'queue* ⇒ *'queue*

```
begin

definition queue-rotate1 :: 'queue ⇒ 'queue
where queue-rotate1 = case-prod queue-enqueue ∘ queue-dequeue

primrec enqueue-new-thread :: 'queue ⇒ ('t,'x,'m) new-thread-action ⇒ 'queue
where
  enqueue-new-thread ts (NewThread t x m) = queue-enqueue t ts
| enqueue-new-thread ts (ThreadExists t b) = ts

definition enqueue-new-threads :: 'queue ⇒ ('t,'x,'m) new-thread-action list ⇒ 'queue
where
  enqueue-new-threads = foldl enqueue-new-thread

primrec round-robin-update-state :: nat ⇒ 'queue round-robin ⇒ 't ⇒ ('l,'t,'x,'m,'w,'o) thread-action
⇒ 'queue round-robin
where
  round-robin-update-state n0 (queue, n) t ta =
   (let queue' = enqueue-new-threads queue {|ta|}_t
    in if n = 0 ∨ Yield ∈ set {|ta|}_c then (queue-rotate1 queue', n0) else (queue', n − 1))

abbreviation round-robin-step ::
  nat ⇒ 'queue round-robin ⇒ ('l,'t,'m,'m-t,'m-w,'s-i) state-refine ⇒ 't
  ⇒ ('t × (('l,'t,'x,'m,'w,'o) thread-action × 'x × 'm) option × 'queue round-robin) option
where
  round-robin-step n0 σ s t ≡ step-thread (round-robin-update-state n0 σ t) s t

partial-function (option) round-robin-reschedule ::
  't ⇒ 'queue ⇒ nat ⇒ ('l,'t,'m,'m-t,'m-w,'s-i) state-refine
  ⇒ ('t × (('l,'t,'x,'m,'w,'o) thread-action × 'x × 'm) option × 'queue round-robin) option
where
  round-robin-reschedule t0 queue n0 s =
   (let
      (t, queue') = queue-dequeue queue
    in
      if t = t0 then
        None
      else
        case round-robin-step n0 (queue-push t queue', n0) s t of
          None ⇒ round-robin-reschedule t0 (queue-enqueue t queue') n0 s
        | ⌊ttaxmσ⌋ ⇒ ⌊ttaxmσ⌋)

primrec round-robin :: nat ⇒ ('l,'t,'x,'m,'w,'o,'m-t,'m-w,'s-i,'queue round-robin) scheduler
where
  round-robin n0 (queue, n) s =
   (if queue-isEmpty queue then None
    else
      let
        (t, queue') = queue-dequeue queue
      in
        (case round-robin-step n0 (queue-push t queue', n) s t of
           ⌊ttaxmσ⌋ ⇒ ⌊ttaxmσ⌋
         | None ⇒ round-robin-reschedule t (queue-enqueue t queue') n0 s))
```

primrec *round-robin-invar* :: *'queue round-robin* ⇒ *'t set* ⇒ *bool*
where *round-robin-invar* (*queue*, *n*) *T* ⟷ *queue-invar queue* ∧ *Round-Robin.round-robin-invar* (*queue-α queue*, *n*) *T*

definition *round-robin-α* :: *'queue round-robin* ⇒ *'t list round-robin*
where *round-robin-α* = *apfst queue-α*

definition *round-robin-start* :: *nat* ⇒ *'t* ⇒ *'queue round-robin*
where *round-robin-start n0 t* = (*queue-enqueue t* (*queue-empty* ()), *n0*)

lemma *round-robin-invar-correct*:
round-robin-invar σ T ⟹ *Round-Robin.round-robin-invar* (*round-robin-α σ*) *T*
by(*cases σ*)(*simp add*: *round-robin-α-def*)

end

locale *round-robin* =
round-robin-base
final r convert-RA output
thr-α thr-invar thr-lookup thr-update
ws-α ws-invar ws-lookup ws-update ws-delete ws-iterate ws-sel
is-α is-invar is-memb is-ins is-delete
queue-α queue-invar queue-empty queue-isEmpty queue-enqueue queue-dequeue queue-push
+
scheduler-aux
final r convert-RA
thr-α thr-invar thr-lookup thr-update
ws-α ws-invar ws-lookup
is-α is-invar is-memb is-ins is-delete
+
ws: *map-update ws-α ws-invar ws-update* +
ws: *map-delete ws-α ws-invar ws-delete* +
ws: *map-iteratei ws-α ws-invar ws-iterate* +
ws: *map-sel′ ws-α ws-invar ws-sel* +
queue: *list queue-α queue-invar* +
queue: *list-empty queue-α queue-invar queue-empty* +
queue: *list-isEmpty queue-α queue-invar queue-isEmpty* +
queue: *list-enqueue queue-α queue-invar queue-enqueue* +
queue: *list-dequeue queue-α queue-invar queue-dequeue* +
queue: *list-push queue-α queue-invar queue-push*
for *final* :: *'x* ⇒ *bool*
and *r* :: *'t* ⇒ (*'x* × *'m*) ⇒ ((*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* × *'x* × *'m*) *Predicate.pred*
and *convert-RA* :: *'l released-locks* ⇒ *'o list*
and *output* :: *'queue round-robin* ⇒ *'t* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* ⇒ *'q option*
and *thr-α* :: *'m-t* ⇒ (*'l*,*'t*,*'x*) *thread-info*
and *thr-invar* :: *'m-t* ⇒ *bool*
and *thr-lookup* :: *'t* ⇒ *'m-t* ⇀ (*'x* × *'l released-locks*)
and *thr-update* :: *'t* ⇒ *'x* × *'l released-locks* ⇒ *'m-t* ⇒ *'m-t*
and *ws-α* :: *'m-w* ⇒ (*'w*,*'t*) *wait-sets*
and *ws-invar* :: *'m-w* ⇒ *bool*
and *ws-lookup* :: *'t* ⇒ *'m-w* ⇀ *'w wait-set-status*
and *ws-update* :: *'t* ⇒ *'w wait-set-status* ⇒ *'m-w* ⇒ *'m-w*
and *ws-delete* :: *'t* ⇒ *'m-w* ⇒ *'m-w*
and *ws-iterate* :: *'m-w* ⇒ (*'t* × *'w wait-set-status*, *'m-w*) *set-iterator*

and *ws-sel* :: *'m-w* ⇒ (*'t* × *'w wait-set-status* ⇒ *bool*) ⇀ (*'t* × *'w wait-set-status*)
and *is-α* :: *'s-i* ⇒ *'t interrupts*
and *is-invar* :: *'s-i* ⇒ *bool*
and *is-memb* :: *'t* ⇒ *'s-i* ⇒ *bool*
and *is-ins* :: *'t* ⇒ *'s-i* ⇒ *'s-i*
and *is-delete* :: *'t* ⇒ *'s-i* ⇒ *'s-i*
and *queue-α* :: *'queue* ⇒ *'t list*
and *queue-invar* :: *'queue* ⇒ *bool*
and *queue-empty* :: *unit* ⇒ *'queue*
and *queue-isEmpty* :: *'queue* ⇒ *bool*
and *queue-enqueue* :: *'t* ⇒ *'queue* ⇒ *'queue*
and *queue-dequeue* :: *'queue* ⇒ *'t* × *'queue*
and *queue-push* :: *'t* ⇒ *'queue* ⇒ *'queue*
begin

lemma *queue-rotate1-correct*:
assumes *queue-invar queue queue-α queue* ≠ []
shows *queue-α* (*queue-rotate1 queue*) = *rotate1* (*queue-α queue*)
and *queue-invar* (*queue-rotate1 queue*)
using *assms*
apply(*auto simp add*: *queue-rotate1-def split-beta queue.dequeue-correct queue.enqueue-correct*)
by(*cases queue-α queue*) *simp-all*

lemma *enqueue-thread-correct*:
assumes *queue-invar queue*
shows *queue-α* (*enqueue-new-thread queue nta*) = *Round-Robin.enqueue-new-thread* (*queue-α queue*) *nta*
and *queue-invar* (*enqueue-new-thread queue nta*)
using *assms*
by(*case-tac* [!] *nta*)(*simp-all add*: *queue.enqueue-correct*)

lemma *enqueue-threads-correct*:
assumes *queue-invar queue*
shows *queue-α* (*enqueue-new-threads queue ntas*) = *Round-Robin.enqueue-new-threads* (*queue-α queue*) *ntas*
and *queue-invar* (*enqueue-new-threads queue ntas*)
using *assms*
apply(*induct ntas arbitrary*: *queue*)
apply(*simp-all add*: *enqueue-new-threads-def Round-Robin.enqueue-new-threads-def enqueue-thread-correct*)
done

lemma *round-robin-update-thread-correct*:
assumes *round-robin-invar σ T t'* ∈ *T*
shows *round-robin-α* (*round-robin-update-state n0 σ t ta*) = *Round-Robin.round-robin-update-state n0* (*round-robin-α σ*) *t ta*
using *assms*
apply(*cases σ*)
apply(*auto simp add*: *round-robin-α-def queue-rotate1-correct enqueue-threads-correct del*: *conjI*)
apply(*subst* (*1 2*) *queue-rotate1-correct*)
apply(*auto simp add*: *enqueue-threads-correct*)
done

lemma *round-robin-step-correct*:
assumes *det*: *α.deterministic I*

```
  and invar: round-robin-invar σ (dom (thr-α (thr s))) state-invar s state-α s ∈ I
  shows
  map-option (apsnd (apsnd round-robin-α)) (round-robin-step n0 σ s t) =
   α.round-robin-step n0 (round-robin-α σ) (state-α s) t (is ?thesis1)
 and case-option True (λ(t, taxm, σ). round-robin-invar σ (case taxm of None ⇒ dom (thr-α (thr s)) |
Some (ta, x', m') ⇒ dom (thr-α (thr s)) ∪ {t. ∃x m. NewThread t x m ∈ set ⦃ta⦄t})) (round-robin-step
n0 σ s t)
  (is ?thesis2)
proof −
  have ?thesis1 ∧ ?thesis2
  proof(cases dom (thr-α (thr s)) = {})
    case True
    thus ?thesis using invar
      apply(cases σ)
      apply(auto dest: step-thread-Some-NoneD[OF det] step-thread-Some-SomeD[OF det])
      apply(fastforce simp add: α.step-thread-eq-None-conv elim: α.active-threads.cases intro: sym)
      done
  next
    case False
    then obtain t' where t': t' ∈ dom (thr-α (thr s)) by blast
    hence ?thesis1
       using step-thread-correct(1)[of I round-robin-invar σ s round-robin-α round-robin-update-state
n0 σ t t, OF det invar]
      unfolding o-def using invar
      by(subst (asm) round-robin-update-thread-correct) auto
    moreover
    { fix ta :: ('l, 't, 'x, 'm, 'w, 'o) thread-action
      assume FWThread.thread-oks (thr-α (thr s)) ⦃ta⦄t
      moreover from t' invar have queue-α (fst σ) ≠ [] by(cases σ) auto
      ultimately have round-robin-invar (round-robin-update-state n0 σ t ta) (dom (thr-α (thr s)) ∪
{t. ∃x m. NewThread t x m ∈ set ⦃ta⦄t})
           using invar t' by(cases σ)(auto simp add: queue-rotate1-correct enqueue-threads-correct
set-enqueue-new-threads iff del: domIff intro: distinct-enqueue-new-threads) }
    from step-thread-correct(2)[OF det, of round-robin-invar σ s round-robin-update-state n0 σ t t,
OF invar this]
    have ?thesis2 using t' invar by simp
    ultimately show ?thesis by blast
  qed
  thus ?thesis1 ?thesis2 by blast+
qed

lemma round-robin-reschedule-correct:
  assumes det: α.deterministic I
  and invar: round-robin-invar (queue, n) (dom (thr-α (thr s))) state-invar s state-α s ∈ I
  and t0: t0 ∈ set (queue-α queue)
  shows map-option (apsnd (apsnd round-robin-α)) (round-robin-reschedule t0 queue n0 s) =
     α.round-robin-reschedule t0 (queue-α queue) n0 (state-α s)
   and case-option True (λ(t, taxm, σ). round-robin-invar σ (case taxm of None ⇒ dom (thr-α
(thr s)) | Some (ta, x', m') ⇒ dom (thr-α (thr s)) ∪ {t. ∃x m. NewThread t x m ∈ set ⦃ta⦄t}))
(round-robin-reschedule t0 queue n0 s)
using t0 invar
proof(induct queue-α queue arbitrary: queue n rule: round-robin-reschedule-induct)
  case head
  { case 1 thus ?case using head[symmetric]
```

by(*subst round-robin-reschedule.simps*)(*subst* α*.round-robin-reschedule.simps*, *clarsimp simp add*: *split-beta queue.dequeue-correct*)
next
case *2* **thus** *?case* **using** *head*[*symmetric*]
by(*subst round-robin-reschedule.simps*)(*clarsimp simp add*: *split-beta queue.dequeue-correct*) **}**
next
case (*rotate* α*queue' t*)
obtain *t' queue'* **where** *queue'*: *queue-dequeue queue* = (*t'*, *queue'*) **by**(*cases queue-dequeue queue*)
note [*simp*] = ‹*t* # α*queue'* = *queue-*α *queue*›[*symmetric*]
{ case *1*
with *queue'* **have** [*simp*]: *t'* = *t* α*queue'* = *queue-*α *queue' queue-invar queue'* **by**(*auto elim*: *queue.removelE*)
from *1 queue'* **have** *invar'*: *round-robin-invar* (*queue-push t queue'*, *n0*) (*dom* (*thr-*α (*thr s*)))
by(*auto simp add*: *queue.push-correct*)
show *?case*
proof(*cases round-robin-step n0* (*queue-push t queue'*, *n0*) *s t*)
case *Some* **thus** *?thesis*
using *queue'* ‹*t* ≠ *t0*› *round-robin-step-correct*[*OF det invar'* ‹*state-invar s*›, *of n0 t*] *invar'* ‹*state-*α *s* ∈ *I*›
by(*subst round-robin-reschedule.simps*)(*subst* α*.round-robin-reschedule.simps*, *auto simp add*: *round-robin-*α*-def queue.push-correct*)
next
case *None*
hence α*None*: α*.round-robin-step n0* (*queue-*α (*queue-push t queue'*), *n0*) (*state-*α *s*) *t* = *None*
using *round-robin-step-correct*[*OF det invar'* ‹*state-invar s*›, *of n0 t*] *invar'* ‹*state-*α *s* ∈ *I*›
by(*auto simp add*: *queue.push-correct round-robin-*α*-def*)
have α*queue'* @ [*t*] = *queue-*α (*queue-enqueue t queue'*) **by**(*simp add*: *queue.enqueue-correct*)
moreover from *invar'*
have *round-robin-invar* (*queue-enqueue t queue'*, *n0*) (*dom* (*thr-*α (*thr s*)))
by(*auto simp add*: *queue.enqueue-correct queue.push-correct*)
ultimately
have *map-option* (*apsnd* (*apsnd round-robin-*α)) (*round-robin-reschedule t0* (*queue-enqueue t queue'*) *n0 s*) =
α*.round-robin-reschedule t0* (*queue-*α (*queue-enqueue t queue'*)) *n0* (*state-*α *s*)
using ‹*state-invar s*› ‹*state-*α *s* ∈ *I*› **by**(*rule rotate.hyps*)
thus *?thesis* **using** *None* α*None* ‹*t* ≠ *t0*› *invar' queue'*
by(*subst round-robin-reschedule.simps*)(*subst* α*.round-robin-reschedule.simps*, *auto simp add*: *queue.enqueue-correct queue.push-correct*)
qed
next
case *2*
with *queue'* **have** [*simp*]: *t'* = *t* α*queue'* = *queue-*α *queue' queue-invar queue'* **by**(*auto elim*: *queue.removelE*)
from *2 queue'* **have** *invar'*: *round-robin-invar* (*queue-push t queue'*, *n0*) (*dom* (*thr-*α (*thr s*)))
by(*auto simp add*: *queue.push-correct*)
show *?case*
proof(*cases round-robin-step n0* (*queue-push t queue'*, *n0*) *s t*)
case *Some* **thus** *?thesis*
using *queue'* ‹*t* ≠ *t0*› *round-robin-step-correct*[*OF det invar'* ‹*state-invar s*›, *of n0 t*] *invar'* ‹*state-*α *s* ∈ *I*›
by(*subst round-robin-reschedule.simps*)(*auto simp add*: *round-robin-*α*-def queue.push-correct*)
next
case *None*
have α*queue'* @ [*t*] = *queue-*α (*queue-enqueue t queue'*) **by**(*simp add*: *queue.enqueue-correct*)

```
    moreover from invar'
    have round-robin-invar (queue-enqueue t queue', n0) (dom (thr-α (thr s)))
      by(auto simp add: queue.enqueue-correct queue.push-correct)
    ultimately
   have case-option True (λ(t, taxm, σ). round-robin-invar σ (case-option (dom (thr-α (thr s))) (λ(ta,
x', m'). dom (thr-α (thr s)) ∪ {t. ∃ x m. NewThread t x m ∈ set ⦃ta⦄ₜ}) taxm)) (round-robin-reschedule
t0 (queue-enqueue t queue') n0 s)
      using ‹state-invar s› ‹state-α s ∈ I› by(rule rotate.hyps)
    thus ?thesis using None ‹t ≠ t0› invar' queue'
    by(subst round-robin-reschedule.simps)(auto simp add: queue.enqueue-correct queue.push-correct)
  qed
 }
qed

lemma round-robin-correct:
 assumes det: α.deterministic I
 and invar: round-robin-invar σ (dom (thr-α (thr s))) state-invar s state-α s ∈ I
 shows map-option (apsnd (apsnd round-robin-α)) (round-robin n0 σ s) =
       α.round-robin n0 (round-robin-α σ) (state-α s)
   (is ?thesis1)
 and case-option True (λ(t, taxm, σ). round-robin-invar σ (case taxm of None ⇒ dom (thr-α (thr s))
| Some (ta, x', m') ⇒ dom (thr-α (thr s)) ∪ {t. ∃ x m. NewThread t x m ∈ set ⦃ta⦄ₜ})) (round-robin
n0 σ s)
   (is ?thesis2)
proof −
 obtain queue n where σ: σ = (queue, n) by(cases σ)
 have ?thesis1 ∧ ?thesis2
 proof(cases queue-α queue)
   case Nil thus ?thesis using invar σ
    by(auto simp add: split-beta queue.isEmpty-correct round-robin-α-def)
 next
   case (Cons t αqueue')
   with invar σ obtain queue'
    where [simp]: queue-dequeue queue = (t, queue') αqueue' = queue-α queue' queue-invar queue'
    by(auto elim: queue.removelE)
   from invar σ Cons have invar': round-robin-invar (queue-push t queue', n) (dom (thr-α (thr s)))
    by(auto simp add: queue.push-correct)
  from invar σ Cons have invar'': round-robin-invar (queue-enqueue t queue', n0) (dom (thr-α (thr
s)))
    by(auto simp add: queue.enqueue-correct)
   show ?thesis
   proof(cases round-robin-step n0 (queue-push t queue', n) s t)
    case Some
    with σ Cons invar show ?thesis
     using round-robin-step-correct[OF det invar' ‹state-invar s›, of n0 t]
     by(auto simp add: queue.isEmpty-correct queue.push-correct round-robin-α-def)
   next
    case None
    from invar σ Cons have t ∈ set (queue-α (queue-enqueue t queue'))
     by(auto simp add: queue.enqueue-correct)
     from round-robin-reschedule-correct[OF det invar'' ‹state-invar s›, OF ‹state-α s ∈ I› this, of
n0] None σ Cons invar
      round-robin-step-correct[OF det invar' ‹state-invar s›, of n0 t]
      show ?thesis by(auto simp add: queue.isEmpty-correct queue.push-correct round-robin-α-def
```

queue.enqueue-correct)
qed
qed
thus *?thesis1 ?thesis2* **by** *simp-all*
qed

lemma *round-robin-scheduler-spec*:
assumes *det*: α.*deterministic I*
shows *scheduler-spec final r* (*round-robin n0*) *round-robin-invar thr-α thr-invar ws-α ws-invar is-α is-invar I*
proof
fix σ *s*
assume *rr*: *round-robin n0* σ *s* = *None*
and *invar*: *round-robin-invar* σ (*dom* (*thr-α* (*thr s*))) *state-invar s state-α s* $\in$ *I*
from *round-robin-correct*[*OF det*, *OF invar*, *of n0*] *rr*
have α.*round-robin n0* (*round-robin-α* σ) (*state-α s*) = *None* **by** *simp*
moreover from *invar* **have** *Round-Robin.round-robin-invar* (*round-robin-α* σ) (*dom* (*thr* (*state-α s*)))
by(*simp add*: *round-robin-invar-correct*)
ultimately show α.*active-threads* (*state-α s*) = {} **by**(*rule* α.*round-robin-NoneD*)
next
fix σ *s t* σ'
assume *rr*: *round-robin n0* σ *s* = $\lfloor$(*t*, *None*, σ')$\rfloor$
and *invar*: *round-robin-invar* σ (*dom* (*thr-α* (*thr s*))) *state-invar s state-α s* $\in$ *I*
from *round-robin-correct*[*OF det*, *OF invar*, *of n0*] *rr*
have *rr'*: α.*round-robin n0* (*round-robin-α* σ) (*state-α s*) = $\lfloor$(*t*, *None*, *round-robin-α* σ')$\rfloor$ **by** *simp*
then show $\exists$ *x ln n*. *thr-α* (*thr s*) *t* = $\lfloor$(*x*, *ln*)$\rfloor$ $\wedge$ *0* < *ln* $ *n* $\wedge$ $\neg$ *waiting* (*ws-α* (*wset s*) *t*) $\wedge$ *may-acquire-all* (*locks s*) *t ln*
by(*rule* α.*round-robin-Some-NoneD*[**where** *s*=*state-α s*, *unfolded state-α-conv*])
next
fix σ *s t ta x' m'* σ'
assume *rr*: *round-robin n0* σ *s* = $\lfloor$(*t*, $\lfloor$(*ta*, *x'*, *m'*)$\rfloor$, σ')$\rfloor$
and *invar*: *round-robin-invar* σ (*dom* (*thr-α* (*thr s*))) *state-invar s state-α s* $\in$ *I*
from *round-robin-correct*[*OF det*, *OF invar*, *of n0*] *rr*
have *rr'*: α.*round-robin n0* (*round-robin-α* σ) (*state-α s*) = $\lfloor$(*t*, $\lfloor$(*ta*, *x'*, *m'*)$\rfloor$, *round-robin-α* σ')$\rfloor$ **by** *simp*
thus $\exists$ *x*. *thr-α* (*thr s*) *t* = $\lfloor$(*x*, *no-wait-locks*)$\rfloor$ $\wedge$ *Predicate.eval* (*r t* (*x*, *shr s*)) (*ta*, *x'*, *m'*) $\wedge$ α.*actions-ok* (*state-α s*) *t ta*
using ‹*state-α s* $\in$ *I*› **by**(*rule* α.*round-robin-Some-SomeD*[*OF det*, **where** *s*=*state-α s*, *unfolded state-α-conv*])
next
fix σ *s t* σ'
assume *rr*: *round-robin n0* σ *s* = $\lfloor$(*t*, *None*, σ')$\rfloor$
and *invar*: *round-robin-invar* σ (*dom* (*thr-α* (*thr s*))) *state-invar s state-α s* $\in$ *I*
from *round-robin-correct*[*OF det*, *OF invar*, *of n0*] *rr*
show *round-robin-invar* σ' (*dom* (*thr-α* (*thr s*))) **by** *simp*
next
fix σ *s t ta x' m'* σ'
assume *rr*: *round-robin n0* σ *s* = $\lfloor$(*t*, $\lfloor$(*ta*, *x'*, *m'*)$\rfloor$, σ')$\rfloor$
and *invar*: *round-robin-invar* σ (*dom* (*thr-α* (*thr s*))) *state-invar s state-α s* $\in$ *I*
from *round-robin-correct*[*OF det*, *OF invar*, *of n0*] *rr*
show *round-robin-invar* σ' (*dom* (*thr-α* (*thr s*)) $\cup$ {*t*. $\exists$ *x m*. *NewThread t x m* $\in$ *set* $\{\!|ta|\!\}_t$}) **by** *simp*
qed

lemma *round-robin-start-invar*:
round-robin-invar (*round-robin-start n0 t0*) {*t0*}
by(*simp add*: *round-robin-start-def queue.empty-correct queue.enqueue-correct*)

end

sublocale *round-robin-base* <
scheduler-base
final r convert-RA
round-robin n0 output pick-wakeup-via-sel (λ*s P*. *ws-sel s* (λ(*k*,*v*). *P k v*)) *round-robin-invar*
thr-α thr-invar thr-lookup thr-update
ws-α ws-invar ws-lookup ws-update ws-delete ws-iterate
is-α is-invar is-memb is-ins is-delete
for *n0* **.**

sublocale *round-robin* <
pick-wakeup-spec
final r convert-RA
pick-wakeup-via-sel (λ*s P*. *ws-sel s* (λ(*k*,*v*). *P k v*)) *round-robin-invar*
thr-α thr-invar
ws-α ws-invar
is-α is-invar
by(*rule pick-wakeup-spec-via-sel*)(*unfold-locales*)

context *round-robin* **begin**

lemma *round-robin-scheduler*:
assumes *det*: *α.deterministic I*
shows
scheduler
final r convert-RA
(*round-robin n0*) (*pick-wakeup-via-sel* (λ*s P*. *ws-sel s* (λ(*k*,*v*). *P k v*))) *round-robin-invar*
thr-α thr-invar thr-lookup thr-update
ws-α ws-invar ws-lookup ws-update ws-delete ws-iterate
is-α is-invar is-memb is-ins is-delete
I
proof −
interpret *scheduler-spec*
final r convert-RA
round-robin n0 round-robin-invar
thr-α thr-invar
ws-α ws-invar
is-α is-invar
I
using *det* **by**(*rule round-robin-scheduler-spec*)

show *?thesis* **by**(*unfold-locales*)(*rule α.deterministic-invariant3p*[*OF det*])
qed

end

lemmas [*code*] =
round-robin-base.queue-rotate1-def

round-robin-base.enqueue-new-thread.simps
round-robin-base.enqueue-new-threads-def
round-robin-base.round-robin-update-state.simps
round-robin-base.round-robin-reschedule.simps
round-robin-base.round-robin.simps
round-robin-base.round-robin-start-def

end
theory *SC-Schedulers*
imports
Random-Scheduler
Round-Robin
../MM/SC-Collections

../Basic/JT-ICF

begin

abbreviation *sc-start-state-refine* ::
'm-t ⇒ (*thread-id* ⇒ (*'x* × *addr released-locks*) ⇒ *'m-t* ⇒ *'m-t*) ⇒ *'m-w* ⇒ *'s-i*
⇒ (*cname* ⇒ *mname* ⇒ *ty list* ⇒ *ty* ⇒ *'md* ⇒ *addr val list* ⇒ *'x*) ⇒ *'md prog* ⇒ *cname* ⇒ *mname* ⇒ *addr val list*
⇒ (*addr*, *thread-id*, *heap*, *'m-t*, *'m-w*, *'s-i*) *state-refine*
where
⋀*is-empty*.
sc-start-state-refine thr-empty thr-update ws-empty is-empty f P ≡
heap-base.start-state-refine addr2thread-id sc-empty (*sc-allocate P*) *thr-empty thr-update ws-empty is-empty f P*

abbreviation *sc-state-α* ::
(*'l*, *'t* :: *linorder*, *'m*, (*'t*, *'x* × *'l* ⇒*f nat*) *rm*, (*'t*, *'w wait-set-status*) *rm*, *'t rs*) *state-refine*
⇒ (*'l*,*'t*,*'x*,*'m*,*'w*) *state*
where *sc-state-α* ≡ *state-refine-base.state-α rm-α rm-α rs-α*

lemma *sc-state-α-sc-start-state-refine* [*simp*]:
sc-state-α (*sc-start-state-refine* (*rm-empty* ()) *rm-update* (*rm-empty* ()) (*rs-empty* ()) *f P C M vs*) = *sc-start-state f P C M vs*
by(*simp add*: *heap-base.start-state-refine-def state-refine-base.state-α.simps split-beta sc.start-state-def rm-correct rs-correct*)

locale *sc-scheduler* =
scheduler
final r convert-RA
schedule output pick-wakeup σ-invar
rm-α rm-invar rm-lookup rm-update
rm-α rm-invar rm-lookup rm-update rm-delete rm-iteratei
rs-α rs-invar rs-memb rs-ins rs-delete
invariant
for *final* :: *'x* ⇒ *bool*
and *r* :: *'t* ⇒ (*'x* × *'m*) ⇒ ((*'l*,*'t* :: *linorder*,*'x*,*'m*,*'w*,*'o*) *thread-action* × *'x* × *'m*) *Predicate.pred*
and *convert-RA* :: *'l released-locks* ⇒ *'o list*
and *schedule* :: (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*,(*'t*, *'x* × *'l* ⇒*f nat*) *rm*,(*'t*, *'w wait-set-status*) *rm*, *'t rs*, *'s*) *scheduler*
and *output* :: *'s* ⇒ *'t* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *thread-action* ⇒ *'q option*
and *pick-wakeup* :: *'s* ⇒ *'t* ⇒ *'w* ⇒ (*'t*, *'w wait-set-status*) *RBT.rbt* ⇒ *'t option*

and *σ-invar* :: *'s ⇒ 't set ⇒ bool*
and *invariant* :: *('l,'t,'x,'m,'w) state set*

locale *sc-round-robin-base* =
round-robin-base
final r convert-RA output
rm-α rm-invar rm-lookup rm-update
rm-α rm-invar rm-lookup rm-update rm-delete rm-iteratei rm-sel
rs-α rs-invar rs-memb rs-ins rs-delete
fifo-α fifo-invar fifo-empty fifo-isEmpty fifo-enqueue fifo-dequeue fifo-push
for *final* :: *'x ⇒ bool*
and *r* :: *'t ⇒ ('x × 'm) ⇒ (('l,'t :: linorder,'x,'m,'w,'o) thread-action × 'x × 'm) Predicate.pred*
and *convert-RA* :: *'l released-locks ⇒ 'o list*
and *output* :: *'t fifo round-robin ⇒ 't ⇒ ('l,'t,'x,'m,'w,'o) thread-action ⇒ 'q option*

locale *sc-round-robin* =
round-robin
final r convert-RA output
rm-α rm-invar rm-lookup rm-update
rm-α rm-invar rm-lookup rm-update rm-delete rm-iteratei rm-sel
rs-α rs-invar rs-memb rs-ins rs-delete
fifo-α fifo-invar fifo-empty fifo-isEmpty fifo-enqueue fifo-dequeue fifo-push
for *final* :: *'x ⇒ bool*
and *r* :: *'t ⇒ ('x × 'm) ⇒ (('l,'t :: linorder,'x,'m,'w,'o) thread-action × 'x × 'm) Predicate.pred*
and *convert-RA* :: *'l released-locks ⇒ 'o list*
and *output* :: *'t fifo round-robin ⇒ 't ⇒ ('l,'t,'x,'m,'w,'o) thread-action ⇒ 'q option*

sublocale *sc-round-robin* < *sc-round-robin-base* **.**

locale *sc-random-scheduler-base* =
random-scheduler-base
final r convert-RA output
rm-α rm-invar rm-lookup rm-update rm-iteratei
rm-α rm-invar rm-lookup rm-update rm-delete rm-iteratei rm-sel
rs-α rs-invar rs-memb rs-ins rs-delete
lsi-α lsi-invar lsi-empty lsi-ins-dj lsi-to-list
for *final* :: *'x ⇒ bool*
and *r* :: *'t ⇒ ('x × 'm) ⇒ (('l,'t :: linorder,'x,'m,'w,'o) thread-action × 'x × 'm) Predicate.pred*
and *convert-RA* :: *'l released-locks ⇒ 'o list*
and *output* :: *random-scheduler ⇒ 't ⇒ ('l,'t,'x,'m,'w,'o) thread-action ⇒ 'q option*

locale *sc-random-scheduler* =
random-scheduler
final r convert-RA output
rm-α rm-invar rm-lookup rm-update rm-iteratei
rm-α rm-invar rm-lookup rm-update rm-delete rm-iteratei rm-sel
rs-α rs-invar rs-memb rs-ins rs-delete
lsi-α lsi-invar lsi-empty lsi-ins-dj lsi-to-list
for *final* :: *'x ⇒ bool*
and *r* :: *'t ⇒ ('x × 'm) ⇒ (('l,'t :: linorder,'x,'m,'w,'o) thread-action × 'x × 'm) Predicate.pred*
and *convert-RA* :: *'l released-locks ⇒ 'o list*
and *output* :: *random-scheduler ⇒ 't ⇒ ('l,'t,'x,'m,'w,'o) thread-action ⇒ 'q option*

sublocale *sc-random-scheduler* < *sc-random-scheduler-base* **.**

No spurious wake-ups in generated code

overloading *sc-spurious-wakeups* ≡ *sc-spurious-wakeups*
begin
definition *sc-spurious-wakeups* [*code*]: *sc-spurious-wakeups* ≡ *False*
end

end

9.5 Tabulation for lookup functions

theory *TypeRelRefine*
imports
../Common/TypeRel
HOL−Library.AList-Mapping
begin

9.5.1 Auxiliary lemmata

lemma *rtranclp-tranclpE*:
assumes $r\widehat{}**\ x\ y$
obtains (*refl*) $x = y$
| (*trancl*) $r\widehat{}{+}{+}\ x\ y$
using *assms*
by(*cases*)(*blast dest*: *rtranclp-into-tranclp1*)+

lemma *map-of-map2*: *map-of* (*map* (λ(*k*, *v*). (*k*, *f k v*)) *xs*) *k* = *map-option* (*f k*) (*map-of xs k*)
by(*induct xs*) *auto*

lemma *map-of-map-K*: *map-of* (*map* (λ*k*. (*k*, *c*)) *xs*) *k* = (*if k* ∈ *set xs then Some c else None*)
by(*induct xs*) *auto*

lift-definition *map-values* :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*c*) *mapping*
is λ*f m k*. *map-option* (*f k*) (*m k*) **.**

lemma *map-values-Mapping* [*simp*]:
map-values f (*Mapping.Mapping m*) = *Mapping.Mapping* (λ*k*. *map-option* (*f k*) (*m k*))
by(*rule map-values.abs-eq*)

lemma *map-Mapping*: *Mapping.map f g* (*Mapping.Mapping m*) = *Mapping.Mapping* (*map-option g* ∘ *m* ∘ *f*)
by(*rule map.abs-eq*)

abbreviation *subclst* :: '*m prog* ⇒ *cname* ⇒ *cname* ⇒ *bool*
where *subclst P* ≡ (*subcls1 P*)$\widehat{}{+}{+}$

9.5.2 Representation type for tabulated lookup functions

type-synonym
'*m prog-impl'* =
'*m cdecl list* ×
(*cname*, '*m class*) *mapping* ×
(*cname*, *cname set*) *mapping* ×
(*cname*, (*vname*, *cname* × *ty* × *fmod*) *mapping*) *mapping* ×

(*cname*, (*mname*, *cname* × *ty list* × *ty* × *'m option*) *mapping*) *mapping*

lift-definition *tabulate-class* :: *'m cdecl list* ⇒ (*cname*, *'m class*) *mapping*
is *class* ∘ *Program* .

lift-definition *tabulate-subcls* :: *'m cdecl list* ⇒ (*cname*, *cname set*) *mapping*
is λ*P C*. *if is-class* (*Program P*) *C then Some* {*D*. *Program P* ⊢ *C* ≼* *D*} *else None* .

lift-definition *tabulate-sees-field* :: *'m cdecl list* ⇒ (*cname*, (*vname*, *cname* × *ty* × *fmod*) *mapping*) *mapping*
is λ*P C*. *if is-class* (*Program P*) *C then*
Some (λ*F*. *if* ∃ *T fm D*. *Program P* ⊢ *C sees F*:*T* (*fm*) *in D then Some* (*field* (*Program P*) *C F*) *else None*)
else None .

lift-definition *tabulate-Method* :: *'m cdecl list* ⇒ (*cname*, (*mname*, *cname* × *ty list* × *ty* × *'m option*) *mapping*) *mapping*
is λ*P C*. *if is-class* (*Program P*) *C then*
Some (λ*M*. *if* ∃ *Ts T mthd D*. *Program P* ⊢ *C sees M*:*Ts*→*T*=*mthd in D then Some* (*method* (*Program P*) *C M*) *else None*)
else None .

fun *wf-prog-impl'* :: *'m prog-impl'* ⇒ *bool*
where
wf-prog-impl' (*P*, *c*, *s*, *f*, *m*) ⟷
c = *tabulate-class P* ∧
s = *tabulate-subcls P* ∧
f = *tabulate-sees-field P* ∧
m = *tabulate-Method P*

9.5.3 Implementation type for tabulated lookup functions

typedef *'m prog-impl* = {*P* :: *'m prog-impl'*. *wf-prog-impl' P*}
morphisms *impl-of ProgRefine*
proof
show ([], *Mapping.empty*, *Mapping.empty*, *Mapping.empty*, *Mapping.empty*) ∈ *?prog-impl*
apply *clarsimp*
by *transfer* (*simp-all add*: *fun-eq-iff is-class-def rel-funI*)
qed

lemma *impl-of-ProgImpl* [*simp*]:
wf-prog-impl' Pfsm ⟹ *impl-of* (*ProgRefine Pfsm*) = *Pfsm*
by(*simp add*: *ProgRefine-inverse*)

definition *program* :: *'m prog-impl* ⇒ *'m prog*
where *program* = *Program* ∘ *fst* ∘ *impl-of*

code-datatype *program*

lemma *prog-impl-eq-iff*:
Pi = *Pi'* ⟷ *program Pi* = *program Pi'* **for** *Pi Pi'*
apply(*cases Pi*)
apply(*cases Pi'*)
apply(*auto simp add*: *ProgRefine-inverse program-def ProgRefine-inject*)

done

lemma *wf-prog-impl′-impl-of* [*simp*, *intro*!]:
wf-prog-impl′ (*impl-of Pi*) **for** *Pi*
using *impl-of*[*of Pi*] **by** *simp*

lemma *ProgImpl-impl-of* [*simp*, *code abstype*]:
ProgRefine (*impl-of Pi*) = *Pi* **for** *Pi*
by(*rule impl-of-inverse*)

lemma *program-ProgRefine* [*simp*]: *wf-prog-impl′ Psfm* ⟹ *program* (*ProgRefine Psfm*) = *Program* (*fst Psfm*)
by(*simp add*: *program-def*)

lemma *classes-program* [*code*]: *classes* (*program P*) = *fst* (*impl-of P*)
by(*simp add*: *program-def*)

lemma *class-program* [*code*]: *class* (*program Pi*) = *Mapping.lookup* (*fst* (*snd* (*impl-of Pi*))) **for** *Pi*
by(*cases Pi*)(*clarsimp simp add*: *tabulate-class-def lookup.rep-eq Mapping-inverse*)

9.5.4 Refining sub class and lookup functions to use precomputed mappings

declare *subcls′.equation* [*code del*]

lemma *subcls′-program* [*code*]:
subcls′ (*program Pi*) *C D* ⟷
C = *D* ∨
(*case Mapping.lookup* (*fst* (*snd* (*snd* (*impl-of Pi*)))) *C of None* ⇒ *False*
| *Some m* ⇒ *D* ∈ *m*) **for** *Pi*
apply(*cases Pi*)
apply(*clarsimp simp add*: *subcls′-def tabulate-subcls-def lookup.rep-eq Mapping-inverse*)
apply(*auto elim*!: *rtranclp-tranclpE dest*: *subcls-is-class intro*: *tranclp-into-rtranclp*)
done

lemma *subcls′-i-i-i-program* [*code*]:
subcls′-i-i-i P C D = (*if subcls′ P C D then Predicate.single* () *else bot*)
by(*rule pred-eqI*)(*auto elim*: *subcls′-i-i-iE intro*: *subcls′-i-i-iI*)

lemma *subcls′-i-i-o-program* [*code*]:
subcls′-i-i-o (*program Pi*) *C* =
sup (*Predicate.single C*) (*case Mapping.lookup* (*fst* (*snd* (*snd* (*impl-of Pi*)))) *C of None* ⇒ *bot* | *Some m* ⇒ *pred-of-set m*) **for** *Pi*
by(*cases Pi*)(*fastforce simp add*: *subcls′-i-i-o-def subcls′-def tabulate-subcls-def lookup.rep-eq Mapping-inverse intro*!: *pred-eqI split*: *if-split-asm elim*: *rtranclp-tranclpE dest*: *subcls-is-class intro*: *tranclp-into-rtranclp*)

lemma *rtranclp-FioB-i-i-subcls1-i-i-o-code* [*code-unfold*]:
rtranclp-FioB-i-i (*subcls1-i-i-o P*) = *subcls′-i-i-i P*
by(*auto simp add*: *fun-eq-iff subcls1-i-i-o-def subcls′-def rtranclp-FioB-i-i-def subcls′-i-i-i-def*)

declare *Method.equation*[*code del*]
lemma *Method-program* [*code*]:
program Pi ⊢ *C sees M*:*Ts*→*T*=*meth in D* ⟷
(*case Mapping.lookup* (*snd* (*snd* (*snd* (*snd* (*impl-of Pi*))))) *C of*
None ⇒ *False*

| *Some m* ⇒
(*case Mapping.lookup m M of*
None ⇒ *False*
| *Some* (*D′*, *Ts′*, *T′*, *meth′*) ⇒ *Ts* = *Ts′* ∧ *T* = *T′* ∧ *meth* = *meth′* ∧ *D* = *D′*)) **for** *Pi*
by(*cases Pi*)(*auto split*: *if-split-asm dest*: *sees-method-is-class simp add*: *tabulate-Method-def lookup.rep-eq Mapping-inverse*)

lemma *Method-i-i-i-o-o-o-o-program* [*code*]:
Method-i-i-i-o-o-o-o (*program Pi*) *C M* =
(*case Mapping.lookup* (*snd* (*snd* (*snd* (*snd* (*impl-of Pi*))))) *C of*
None ⇒ *bot*
| *Some m* ⇒
(*case Mapping.lookup m M of*
None ⇒ *bot*
| *Some* (*D*, *Ts*, *T*, *meth*) ⇒ *Predicate.single* (*Ts*, *T*, *meth*, *D*))) **for** *Pi*
by(*auto simp add*: *Method-i-i-i-o-o-o-o-def Method-program intro*!: *pred-eqI*)

lemma *Method-i-i-i-o-o-o-i-program* [*code*]:
Method-i-i-i-o-o-o-i (*program Pi*) *C M D* =
(*case Mapping.lookup* (*snd* (*snd* (*snd* (*snd* (*impl-of Pi*))))) *C of*
None ⇒ *bot*
| *Some m* ⇒
(*case Mapping.lookup m M of*
None ⇒ *bot*
| *Some* (*D′*, *Ts*, *T*, *meth*) ⇒ *if D* = *D′ then Predicate.single* (*Ts*, *T*, *meth*) *else bot*)) **for** *Pi*
by(*auto simp add*: *Method-i-i-i-o-o-o-i-def Method-program intro*!: *pred-eqI*)

declare *sees-field.equation*[*code del*]

lemma *sees-field-program* [*code*]:
program Pi ⊢ *C sees F*:*T* (*fd*) *in D* ⟷
(*case Mapping.lookup* (*fst* (*snd* (*snd* (*snd* (*impl-of Pi*))))) *C of*
None ⇒ *False*
| *Some m* ⇒
(*case Mapping.lookup m F of*
None ⇒ *False*
| *Some* (*D′*, *T′*, *fd′*) ⇒ *T* = *T′* ∧ *fd* = *fd′* ∧ *D* = *D′*)) **for** *Pi*
by(*cases Pi*)(*auto split*: *if-split-asm dest*: *has-visible-field*[*THEN has-field-is-class*] *simp add*: *tabulate-sees-field-def lookup.rep-eq Mapping-inverse*)

lemma *sees-field-i-i-i-o-o-o-program* [*code*]:
sees-field-i-i-i-o-o-o (*program Pi*) *C F* =
(*case Mapping.lookup* (*fst* (*snd* (*snd* (*snd* (*impl-of Pi*))))) *C of*
None ⇒ *bot*
| *Some m* ⇒
(*case Mapping.lookup m F of*
None ⇒ *bot*
| *Some* (*D*, *T*, *fd*) ⇒ *Predicate.single*(*T*, *fd*, *D*))) **for** *Pi*
by(*auto simp add*: *sees-field-program sees-field-i-i-i-o-o-o-def intro*: *pred-eqI*)

lemma *sees-field-i-i-i-o-o-i-program* [*code*]:
sees-field-i-i-i-o-o-i (*program Pi*) *C F D* =
(*case Mapping.lookup* (*fst* (*snd* (*snd* (*snd* (*impl-of Pi*))))) *C of*
None ⇒ *bot*

| *Some m* ⇒
(*case Mapping.lookup m F of*
None ⇒ *bot*
| *Some* (*D'*, *T*, *fd*) ⇒ *if D* = *D' then Predicate.single*(*T*, *fd*) *else bot*)) **for** *Pi*
by(*auto simp add*: *sees-field-program sees-field-i-i-i-o-o-i-def intro*: *pred-eqI*)

lemma *field-program* [*code*]:
field (*program Pi*) *C F* =
(*case Mapping.lookup* (*fst* (*snd* (*snd* (*snd* (*impl-of Pi*))))) *C of*
None ⇒ *Code.abort* (*STR* ''*not-unique*'') (λ-. *Predicate.the bot*)
| *Some m* ⇒
(*case Mapping.lookup m F of*
None ⇒ *Code.abort* (*STR* ''*not-unique*'') (λ-. *Predicate.the bot*)
| *Some* (*D'*, *T*, *fd*) ⇒ (*D'*, *T*, *fd*))) **for** *Pi*
unfolding *field-def*
by(*cases Pi*)(*fastforce simp add*: *Predicate.the-def tabulate-sees-field-def lookup.rep-eq Mapping-inverse split*: *if-split-asm intro*: *arg-cong*[**where** *f*=*The*] *dest*: *has-visible-field*[*THEN has-field-is-class*] *sees-field-fun*)

9.5.5 Implementation for precomputing mappings

definition *tabulate-program* :: *'m cdecl list* ⇒ *'m prog-impl*
where *tabulate-program P* = *ProgRefine* (*P*, *tabulate-class P*, *tabulate-subcls P*, *tabulate-sees-field P*, *tabulate-Method P*)

lemma *impl-of-tabulate-program* [*code abstract*]:
impl-of (*tabulate-program P*) = (*P*, *tabulate-class P*, *tabulate-subcls P*, *tabulate-sees-field P*, *tabulate-Method P*)
by(*simp add*: *tabulate-program-def*)

lemma *Program-code* [*code*]:
Program = *program* ∘ *tabulate-program*
by(*simp add*: *program-def fun-eq-iff tabulate-program-def*)

class

lemma *tabulate-class-code* [*code*]:
tabulate-class = *Mapping.of-alist*
by *transfer* (*simp add*: *fun-eq-iff*)

subcls

inductive *subcls1'* :: *'m cdecl list* ⇒ *cname* ⇒ *cname* ⇒ *bool*
where
find: *C* ≠ *Object* ⟹ *subcls1'* ((*C*, *D*, *rest*) # *P*) *C D*
| *step*: ⟦ *C* ≠ *Object*; *C* ≠ *C'*; *subcls1' P C D* ⟧ ⟹ *subcls1'* ((*C'*, *D'*, *rest*) # *P*) *C D*

code-pred
(*modes*: *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
subcls1' **.**

lemma *subcls1-into-subcls1'*:
assumes *subcls1* (*Program P*) *C D*
shows *subcls1' P C D*
proof −
from *assms* **obtain** *rest* **where** *map-of P C* = ⌊(*D*, *rest*)⌋ *C* ≠ *Object* **by** *cases simp*

```
  thus ?thesis by(induct P)(auto split: if-split-asm intro: subcls1'.intros)
qed

lemma subcls1'-into-subcls1:
  assumes subcls1' P C D
  shows subcls1 (Program P) C D
using assms
proof(induct)
  case find thus ?case by(auto intro: subcls1.intros)
next
  case step thus ?case by(auto elim!: subcls1.cases intro: subcls1.intros)
qed

lemma subcls1-eq-subcls1':
  subcls1 (Program P) = subcls1' P
by(auto simp add: fun-eq-iff intro: subcls1-into-subcls1' subcls1'-into-subcls1)

definition subcls'' :: 'm cdecl list ⇒ cname ⇒ cname ⇒ bool
where subcls'' P = (subcls1' P)^**

code-pred
  (modes: i ⇒ i ⇒ i ⇒ bool)
  [inductify]
  subcls'' .

lemma subcls''-eq-subcls: subcls'' P = subcls (Program P)
by(simp add: subcls''-def subcls1-eq-subcls1')

lemma subclst-snd-classD:
  assumes subclst (Program P) C D
  shows D ∈ fst ` snd ` set P
using assms
by(induct)(fastforce elim!: subcls1.cases dest!: map-of-SomeD intro: rev-image-eqI)+

definition check-acyclicity :: (cname, cname set) mapping ⇒ 'm cdecl list ⇒ unit
where check-acyclicity - - = ()

definition cyclic-class-hierarchy :: unit
where [code del]: cyclic-class-hierarchy = ()

declare [[code abort: cyclic-class-hierarchy]]

lemma check-acyclicity-code:
  check-acyclicity mapping P =
   (let - =
     map (λ(C, D, -).
       if C = Object then ()
       else
         (case Mapping.lookup mapping D of
            None ⇒ ()
          | Some Cs ⇒ if C ∈ Cs then cyclic-class-hierarchy else ()))
       P
    in ())
by simp
```

lemma *tablulate-subcls-code* [*code*]:
  tabulate-subcls P =
  (*let cnames* = *map fst P*;
      cnames′ = *map* (*fst* ∘ *snd*) *P*;
      mapping = *Mapping.tabulate cnames* (λ*C*. *set* (*C* # [*D* ← *cnames′*. *subcls″ P C D*]));
      - = *check-acyclicity mapping P*
   in mapping
)
apply(*auto simp add*: *tabulate-subcls-def Mapping.tabulate-def fun-eq-iff is-class-def o-def map-of-map2*[*simplified split-def*] *Mapping-inject*)
 apply(*subst map-of-map2*[*simplified split-def*])
 apply(*auto simp add*: *fun-eq-iff subcls″-eq-subcls map-of-map-K dest*: *subclst-snd-classD elim*: *rtranclp-tranclpE*)[*1*]
apply(*subst map-of-map2*[*simplified split-def*])
apply(*rule sym*)
apply *simp*
apply(*case-tac map-of P x*)
apply *auto*
done

Fields

Problem: Does not terminate for cyclic class hierarchies! This problem already occurs in Jinja's well-formedness checker: *wf-cdecl* calls *wf-mdecl* before checking for acyclicity, but *wf-J-mdecl* involves the type judgements, which in turn requires *Fields* (via *sees-field*). Checking acyclicity before executing *Fields′* for tabulation is difficult because we would have to intertwine tabulation and well-formedness checking. Possible (local) solution: additional termination parameter (like memoisation for *rtranclp*) and list option as error return parameter.

inductive
  Fields′ :: *′m cdecl list* ⇒ *cname* ⇒ ((*vname* × *cname*) × (*ty* × *fmod*)) *list* ⇒ *bool*
for *P* :: *′m cdecl list*
where
  rec:
  ⟦ *map-of P C* = *Some*(*D*,*fs*,*ms*); *C* ≠ *Object*; *Fields′ P D FDTs*;
     FDTs′ = *map* (λ(*F*,*Tm*). ((*F*,*C*),*Tm*)) *fs* @ *FDTs* ⟧
  ⟹ *Fields′ P C FDTs′*
| *Object*:
  ⟦ *map-of P Object* = *Some*(*D*,*fs*,*ms*); *FDTs* = *map* (λ(*F*,*T*). ((*F*,*Object*),*T*)) *fs* ⟧
  ⟹ *Fields′ P Object FDTs*

lemma *Fields′-into-Fields*:
  assumes *Fields′ P C FDTs*
  shows *Program P* ⊢ *C has-fields FDTs*
using *assms*
by *induct*(*auto intro*: *Fields.intros*)

lemma *Fields-into-Fields′*:
  assumes *Program P* ⊢ *C has-fields FDTs*
  shows *Fields′ P C FDTs*
using *assms*
by *induct*(*auto intro*: *Fields′.intros*)

lemma *Fields′-eq-Fields*:
Fields′ P = Fields (Program P)
by(*auto simp add*: *fun-eq-iff intro*: *Fields′-into-Fields Fields-into-Fields′*)

code-pred
(*modes*: *i ⇒ i ⇒ o ⇒ bool*)
Fields′ **.**

definition *fields′* :: *′m cdecl list ⇒ cname ⇒ ((vname × cname) × (ty × fmod)) list*
where *fields′ P C = (if ∃ FDTs. Fields′ P C FDTs then THE FDTs. Fields′ P C FDTs else [])*

lemma *eval-Fields′-conv*:
Predicate.eval (Fields′-i-i-o P C) = Fields′ P C
by(*auto intro*: *Fields′-i-i-oI elim*: *Fields′-i-i-oE intro*!: *ext*)

lemma *fields′-code* [*code*]:
fields′ P C =
(let FDTs = Fields′-i-i-o P C in if Predicate.holds (FDTs ⤜ (λ-. Predicate.single ())) then Predicate.the FDTs else [])
by(*auto simp add*: *fields′-def holds-eq Fields′-i-i-o-def intro*: *Fields′-i-i-oI Predicate.the-eqI*[*THEN sym*])

lemma *The-Fields* [*simp*]:
P ⊢ C has-fields FDTs ⟹ The (Fields P C) = FDTs
by(*auto dest*: *has-fields-fun*)

lemma *tabulate-sees-field-code* [*code*]:
tabulate-sees-field P =
Mapping.tabulate (map fst P) (λC. Mapping.of-alist (map (λ((F, D), Tfm). (F, (D, Tfm))) (fields′ P C)))
apply(*simp add*: *tabulate-sees-field-def tabulate-def is-class-def fields′-def Fields′-eq-Fields Mapping-inject*)
apply(*rule ext*)
apply *clarsimp*
apply(*rule conjI*)
apply(*clarsimp simp add*: *o-def*)
apply(*subst map-of-map2*[*unfolded split-def*])
apply *simp*
apply *transfer*
apply(*rule conjI*)
apply *clarsimp*
apply(*rule ext*)
apply *clarsimp*
apply(*rule conjI*)
apply(*clarsimp simp add*: *sees-field-def Fields′-eq-Fields*)
apply(*drule* (*1*) *has-fields-fun*, *clarsimp*)
apply *clarify*
apply(*rule sym*)
apply(*rule ccontr*)
apply(*clarsimp simp add*: *sees-field-def Fields′-eq-Fields*)
apply *clarsimp*
apply(*rule ext*)
apply(*clarsimp simp add*: *sees-field-def*)
apply(*clarsimp simp add*: *o-def*)
apply(*subst map-of-map2*[*simplified split-def*])

apply(*rule sym*)
apply(*clarsimp*)
apply(*rule ccontr*)
apply *simp*
done

Methods

Same termination problem as for *Fields'*

inductive *Methods'* :: *'m cdecl list* ⇒ *cname* ⇒ (*mname* × (*ty list* × *ty* × *'m option*) × *cname*) *list* ⇒ *bool*
  for *P* :: *'m cdecl list*
where
  ⟦ *map-of P Object* = *Some*(*D*,*fs*,*ms*); *Mm* = *map* (λ(*M*, *rest*). (*M*, (*rest*, *Object*))) *ms* ⟧
   ⟹ *Methods' P Object Mm*
| ⟦ *map-of P C* = *Some*(*D*,*fs*,*ms*); *C* ≠ *Object*; *Methods' P D Mm*;
     Mm' = *map* (λ(*M*, *rest*). (*M*, (*rest*, *C*))) *ms* @ *Mm* ⟧
   ⟹ *Methods' P C Mm'*

lemma *Methods'-into-Methods*:
  assumes *Methods' P C Mm*
  shows *Program P* ⊢ *C sees-methods* (*map-of Mm*)
using *assms*
apply *induct*
 apply(*clarsimp simp add*: *o-def split-def*)
 apply(*rule sees-methods-Object*)
  apply *fastforce*
 apply(*rule ext*)
 apply(*subst map-of-map2*[*unfolded split-def*])
 apply(*simp add*: *o-def*)

apply(*rule sees-methods-rec*)
   apply *fastforce*
  apply *simp*
 apply *assumption*
apply(*clarsimp simp add*: *map-add-def map-of-map2*)
done

lemma *Methods-into-Methods'*:
  assumes *Program P* ⊢ *C sees-methods Mm*
  shows ∃ *Mm'*. *Methods' P C Mm'* ∧ *Mm* = *map-of Mm'*
using *assms*
by *induct*(*auto intro*: *Methods'.intros simp add*: *map-of-map2 map-add-def*)

code-pred
  (*modes*: *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
  Methods'
.

definition *methods'* :: *'m cdecl list* ⇒ *cname* ⇒ (*mname* × (*ty list* × *ty* × *'m option*) × *cname*) *list*
where *methods' P C* = (*if* ∃ *Mm*. *Methods' P C Mm then THE Mm*. *Methods' P C Mm else* [])

lemma *methods'-code* [*code*]:
  methods' P C =

(*let Mm = Methods'-i-i-o P C*
in if Predicate.holds (Mm ⤜ (λ-. Predicate.single ())) then Predicate.the Mm else [])
unfolding *methods'-def*
by(*auto simp add*: *holds-eq Methods'-i-i-o-def Predicate.the-def*)

lemma *Methods'-fun*:
assumes *Methods' P C Mm*
shows *Methods' P C Mm' ⟹ Mm = Mm'*
using *assms*
apply(*induct arbitrary*: *Mm'*)
apply(*fastforce elim*: *Methods'.cases*)
apply(*rotate-tac −1*)
apply(*erule Methods'.cases*)
apply(*fastforce*)
apply *clarify*
apply(*simp*)
done

lemma *The-Methods'* [*simp*]: *Methods' P C Mm ⟹ The (Methods' P C) = Mm*
by(*auto dest*: *Methods'-fun*)

lemma *methods-def2* [*simp*]: *Methods' P C Mm ⟹ methods' P C = Mm*
by(*auto simp add*: *methods'-def*)

lemma *tabulate-Method-code* [*code*]:
tabulate-Method P =
Mapping.tabulate (map fst P) (λC. Mapping.of-alist (map (λ(M, (rest, D)). (M, D, rest)) (methods' P C)))
apply(*simp add*: *tabulate-Method-def tabulate-def o-def lookup.rep-eq Mapping-inject*)
apply(*rule ext*)
apply *clarsimp*
apply(*rule conjI*)
apply *clarify*
apply(*rule sym*)
apply(*subst map-of-map2*[*unfolded split-def*])
apply(*simp add*: *is-class-def*)
apply *transfer*
apply(*rule ext*)
apply(*simp add*: *map-of-map2*)
apply(*rule conjI*)
apply(*clarsimp simp add*: *map-of-map2 Method-def*)
apply(*drule Methods-into-Methods'*)
apply *clarsimp*
apply(*simp add*: *split-def*)
apply(*subst map-of-map2*[*unfolded split-def*])
apply *simp*
apply *clarify*
apply(*clarsimp simp add*: *methods'-def*)
apply(*frule Methods'-into-Methods*)
apply(*clarsimp simp add*: *Method-def*)
apply(*simp add*: *split-def*)
apply(*subst map-of-map2*[*unfolded split-def*])
apply(*fastforce intro*: *ccontr*)
apply *clarify*

apply(*rule sym*)
apply(*simp add*: *map-of-eq-None-iff is-class-def*)
apply(*simp only*: *set-map*[*symmetric*] *map-map o-def fst-conv*)
apply *simp*
done

Merge modules TypeRel, Decl and TypeRelRefine to avoid cyclic modules

code-identifier
  code-module *TypeRel* ⇀
    (*SML*) *TypeRel* **and** (*Haskell*) *TypeRel* **and** (*OCaml*) *TypeRel*
| **code-module** *TypeRelRefine* ⇀
    (*SML*) *TypeRel* **and** (*Haskell*) *TypeRel* **and** (*OCaml*) *TypeRel*
| **code-module** *Decl* ⇀
    (*SML*) *TypeRel* **and** (*Haskell*) *TypeRel* **and** (*OCaml*) *TypeRel*

ML-val ‹@{*code Program*}›

end

theory *PCompilerRefine*
imports
  TypeRelRefine
  ../Compiler/PCompiler
begin

9.5.6 *compP*

Applying the compiler to a tabulated program either compiles every method twice (once for the program itself and once for method lookup) or recomputes the class and method lookup tabulation from scratch. We follow the second approach.

fun *compP-code'* :: (*cname* ⇒ *mname* ⇒ *ty list* ⇒ *ty* ⇒ *'a* ⇒ *'b*) ⇒ *'a prog-impl'* ⇒ *'b prog-impl'*
where
  compP-code' f (*P*, *Cs*, *s*, *F*, *m*) =
  (*let P'* = *map* (*compC f*) *P*
   in (*P'*, *tabulate-class P'*, *s*, *F*, *tabulate-Method P'*))

definition *compP-code* :: (*cname* ⇒ *mname* ⇒ *ty list* ⇒ *ty* ⇒ *'a* ⇒ *'b*) ⇒ *'a prog-impl* ⇒ *'b prog-impl*
where *compP-code f P* = *ProgRefine* (*compP-code' f* (*impl-of P*))

declare *compP.simps* [*simp del*] *compP.simps*[*symmetric*, *simp*]

lemma *compP-code-code* [*code abstract*]:
  impl-of (*compP-code f P*) = *compP-code' f* (*impl-of P*)
apply(*cases P*)
apply(*simp add*: *compP-code-def*)
apply(*subst ProgRefine-inverse*)
apply(*auto simp add*: *tabulate-subcls-def tabulate-sees-field-def Mapping-inject intro*!: *ext*)
done

declare *compP.simps* [*simp*] *compP.simps*[*symmetric*, *simp del*]

lemma *compP-program* [*code*]:

compP f (*program P*) = *program* (*compP-code f P*)
by(*cases P*)(*clarsimp simp add*: *program-def compP-code-code*)

Merge module names to avoid cycles in module dependency

code-identifier
code-module *PCompiler* ⇀
(*SML*) *PCompiler* **and** (*OCaml*) *PCompiler* **and** (*Haskell*) *PCompiler*
| **code-module** *PCompilerRefine* ⇀
(*SML*) *PCompiler* **and** (*OCaml*) *PCompiler* **and** (*Haskell*) *PCompiler*

ML-val ‹@{*code compP*}›

end

9.6 Executable semantics for J

theory *J-Execute*
imports
SC-Schedulers
../J/Threaded
begin

interpretation *sc*:
J-heap-base
addr2thread-id
thread-id2addr
sc-spurious-wakeups
sc-empty
sc-allocate P
sc-typeof-addr
sc-heap-read
sc-heap-write
for *P* **.**

abbreviation *sc-red* ::
((*addr*, *thread-id*, *heap*) *external-thread-action* ⇒ (*addr*, *thread-id*, *'o*, *heap*) *Jinja-thread-action*)
⇒ *addr J-prog* ⇒ *thread-id* ⇒ *addr expr* ⇒ *heap* × *addr locals*
⇒ (*addr*, *thread-id*, *'o*, *heap*) *Jinja-thread-action* ⇒ *addr expr* ⇒ *heap* × *addr locals* ⇒ *bool*
(‹-,-,- ⊢*sc* ((*1*⟨-,/-⟩) -−→/ (*1*⟨-,/-⟩))› [*51*,*51*,*0*,*0*,*0*,*0*,*0*,*0*] *81*)
where
sc-red extTA P ≡ *sc.red* (*TYPE*(*addr J-mb*)) *P extTA P*

fun *sc-red-i-i-i-i-i-i-i-i-i-Fii-i-oB-Fii-i-i-oB-i-i-i-i-i-o-o-o*
where
sc-red-i-i-i-i-i-i-i-i-i-Fii-i-oB-Fii-i-i-oB-i-i-i-i-i-o-o-o P t ((*e*, *xs*), *h*) =
red-i-i-i-i-i-i-i-Fii-i-oB-Fii-i-i-oB-i-i-i-i-i-o-o-o
addr2thread-id thread-id2addr sc-spurious-wakeups
sc-empty (*sc-allocate P*) *sc-typeof-addr sc-heap-read-i-i-i-o sc-heap-write-i-i-i-i-o*
(*extTA2J P*) *P t e* (*h*, *xs*)
⤜ (λ(*ta*, *e*, *h*, *xs*). *Predicate.single* (*ta*, (*e*, *xs*), *h*))

abbreviation *sc-J-start-state-refine* ::
addr J-prog ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒

(*addr*, *thread-id*, *heap*, (*thread-id*, (*addr expr* × *addr locals*) × *addr released-locks*) *rm*, (*thread-id*, *addr wait-set-status*) *rm*, *thread-id rs*) *state-refine*
where
sc-J-start-state-refine ≡
sc-start-state-refine
(*rm-empty* ()) *rm-update* (*rm-empty* ()) (*rs-empty* ())
(λ*C M Ts T* (*pns*, *body*) *vs*. (*blocks* (*this* # *pns*) (*Class C* # *Ts*) (*Null* # *vs*) *body*, *Map.empty*))

lemma *eval-sc-red-i-i-i-i-i-i-Fii-i-oB-Fii-i-i-oB-i-i-i-i-i-o-o-o*:
(λ*t xm ta x′m′*. *Predicate.eval* (*sc-red-i-i-i-i-i-i-i-i-Fii-i-oB-Fii-i-i-oB-i-i-i-i-i-o-o-o P t xm*) (*ta*, *x′m′*))
=
(λ*t* ((*e*, *xs*), *h*) *ta* ((*e′*, *xs′*), *h′*). *extTA2J P*,*P*,*t* ⊢*sc* ⟨*e*, (*h*, *xs*)⟩ −*ta*→ ⟨*e′*, (*h′*, *xs′*)⟩))
by(*auto elim*!: *red-i-i-i-i-i-i-Fii-i-oB-Fii-i-i-oB-i-i-i-i-i-o-o-oE intro*!: *red-i-i-i-i-i-i-Fii-i-oB-Fii-i-i-oB-i-i-i-i-i-o-o-oI ext SUP1-I simp add*: *eval-sc-heap-write-i-i-i-i-o eval-sc-heap-read-i-i-i-o*)

lemma *sc-J-start-state-invar*: (λ-. *True*) (*sc-state-α* (*sc-J-start-state-refine P C M vs*))
by *simp*

9.6.1 Round-robin scheduler

interpretation *J-rr*:
sc-round-robin-base
final-expr sc-red-i-i-i-i-i-i-i-i-Fii-i-oB-Fii-i-i-oB-i-i-i-i-i-o-o-o P convert-RA Jinja-output
for *P*
.

definition *sc-rr-J-start-state* :: *nat* ⇒ *'m prog* ⇒ *thread-id fifo round-robin*
where *sc-rr-J-start-state n0 P* = *J-rr.round-robin-start n0* (*sc-start-tid P*)

definition *exec-J-rr* ::
nat ⇒ *addr J-prog* ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒
(*thread-id* × (*addr*, *thread-id*) *obs-event list*,
(*addr*, *thread-id*) *locks* × ((*thread-id*, (*addr expr* × *addr locals*) × *addr released-locks*) *rm* × *heap*)
×
(*thread-id*, *addr wait-set-status*) *rm* × *thread-id rs*) *tllist*
where
exec-J-rr n0 P C M vs = *J-rr.exec P n0* (*sc-rr-J-start-state n0 P*) (*sc-J-start-state-refine P C M vs*)

interpretation *J-rr*:
sc-round-robin
final-expr sc-red-i-i-i-i-i-i-i-i-Fii-i-oB-Fii-i-i-oB-i-i-i-i-i-o-o-o P convert-RA Jinja-output
for *P*
by(*unfold-locales*)

interpretation *J-rr*:
sc-scheduler
final-expr sc-red-i-i-i-i-i-i-i-i-Fii-i-oB-Fii-i-i-oB-i-i-i-i-i-o-o-o P convert-RA
J-rr.round-robin P n0 Jinja-output pick-wakeup-via-sel (λ*s P*. *rm-sel s* (λ(*k*,*v*). *P k v*)) *J-rr.round-robin-invar*
UNIV
for *P n0*
unfolding *sc-scheduler-def*
apply(*rule J-rr.round-robin-scheduler*)
apply(*unfold eval-sc-red-i-i-i-i-i-i-Fii-i-oB-Fii-i-i-oB-i-i-i-i-i-o-o-o*)
apply(*rule sc.red-mthr-deterministic*[*OF sc-deterministic-heap-ops*])

apply(*simp add*: *sc-spurious-wakeups*)
done

9.6.2 Random scheduler

interpretation *J-rnd*:
sc-random-scheduler-base
final-expr sc-red-i-i-i-i-i-i-i-i-i-Fii-i-oB-Fii-i-i-oB-i-i-i-i-i-o-o-o P convert-RA Jinja-output
for *P*
.

definition *sc-rnd-J-start-state* :: *Random.seed* ⇒ *random-scheduler*
where *sc-rnd-J-start-state seed* = *seed*

definition *exec-J-rnd* ::
Random.seed ⇒ *addr J-prog* ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒
(*thread-id* × (*addr*, *thread-id*) *obs-event list*,
(*addr*, *thread-id*) *locks* × ((*thread-id*, (*addr expr* × *addr locals*) × *addr released-locks*) *rm* × *heap*)
×
(*thread-id*, *addr wait-set-status*) *rm* × *thread-id rs*) *tllist*
where
exec-J-rnd seed P C M vs = *J-rnd.exec P* (*sc-rnd-J-start-state seed*) (*sc-J-start-state-refine P C M vs*)

interpretation *J-rnd*:
sc-random-scheduler
final-expr sc-red-i-i-i-i-i-i-i-i-i-Fii-i-oB-Fii-i-i-oB-i-i-i-i-i-o-o-o P convert-RA Jinja-output
for *P*
by(*unfold-locales*)

interpretation *J-rnd*:
sc-scheduler
final-expr sc-red-i-i-i-i-i-i-i-i-i-Fii-i-oB-Fii-i-i-oB-i-i-i-i-i-o-o-o P convert-RA
J-rnd.random-scheduler P Jinja-output pick-wakeup-via-sel (λ*s P*. *rm-sel s* (λ(*k*,*v*). *P k v*)) λ- -. *True*
UNIV
for *P*
unfolding *sc-scheduler-def*
apply(*rule J-rnd.random-scheduler-scheduler*)
apply(*unfold eval-sc-red-i-i-i-i-i-i-Fii-i-oB-Fii-i-i-oB-i-i-i-i-i-o-o-o*)
apply(*rule sc.red-mthr-deterministic*[*OF sc-deterministic-heap-ops*])
apply(*simp add*: *sc-spurious-wakeups*)
done

ML-val ‹@{*code exec-J-rr*}›

ML-val ‹@{*code exec-J-rnd*}›

end

9.7 Executable semantics for the JVM

theory *ExternalCall-Execute*
imports

../Common/ExternalCall
../Basic/Set-without-equal
begin

9.7.1 Translated versions of external calls for the JVM

locale *heap-execute* = *addr-base* +
constrains *addr2thread-id* :: ('*addr* :: *addr*) ⇒ '*thread-id*
and *thread-id2addr* :: '*thread-id* ⇒ '*addr*
fixes *spurious-wakeups* :: *bool*
and *empty-heap* :: '*heap*
and *allocate* :: '*heap* ⇒ *htype* ⇒ ('*heap* × '*addr*) *set*
and *typeof-addr* :: '*heap* ⇒ '*addr* ⇒ *htype option*
and *heap-read* :: '*heap* ⇒ '*addr* ⇒ *addr-loc* ⇒ '*addr val set*
and *heap-write* :: '*heap* ⇒ '*addr* ⇒ *addr-loc* ⇒ '*addr val* ⇒ '*heap set*

sublocale *heap-execute* < *execute*: *heap-base*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr
λh a ad v. v ∈ heap-read h a ad λh a ad v h'. h' ∈ heap-write h a ad v
.

context *heap-execute* **begin**

definition *heap-copy-loc* :: '*addr* ⇒ '*addr* ⇒ *addr-loc* ⇒ '*heap* ⇒ (('*addr*, '*thread-id*) *obs-event list* × '*heap*) *set*
where [*simp*]:
heap-copy-loc a a' al h = {(*obs*, *h'*). *execute.heap-copy-loc a a' al h obs h'*}

lemma *heap-copy-loc-code*:
heap-copy-loc a a' al h =
(*do* {
v ← *heap-read h a al*;
h' ← *heap-write h a' al v*;
{([*ReadMem a al v*, *WriteMem a' al v*], *h'*)}
})
by(*auto simp add*: *execute.heap-copy-loc.simps*)

definition *heap-copies* :: '*addr* ⇒ '*addr* ⇒ *addr-loc list* ⇒ '*heap* ⇒ (('*addr*, '*thread-id*) *obs-event list* × '*heap*) *set*
where [*simp*]: *heap-copies a a' al h* = {(*obs*, *h'*). *execute.heap-copies a a' al h obs h'*}

lemma *heap-copies-code*:
shows *heap-copies-Nil*:
heap-copies a a' [] *h* = {([], *h*)}
and *heap-copies-Cons*:
heap-copies a a' (*al* # *als*) *h* =
(*do* {
(*ob*, *h'*) ← *heap-copy-loc a a' al h*;
(*obs*, *h''*) ← *heap-copies a a' als h'*;
{(*ob* @ *obs*, *h''*)}
})
by(*fastforce elim*!: *execute.heap-copies-cases intro*: *execute.heap-copies.intros*)+

definition *heap-clone* :: *'m prog* ⇒ *'heap* ⇒ *'addr* ⇒ (*'heap* × ((*'addr*, *'thread-id*) *obs-event list* × *'addr*) *option*) *set*
where [*simp*]: *heap-clone P h a* = {(*h'*, *obsa*). *execute.heap-clone P h a h' obsa*}

lemma *heap-clone-code*:
heap-clone P h a =
(*case typeof-addr h a of*
⌊*Class-type C*⌋ ⇒
let HA = *allocate h* (*Class-type C*)
in if HA = {} *then* {(*h*, *None*)} *else do* {
(*h'*, *a'*) ← *HA*;
FDTs ← *set-of-pred* (*Fields-i-i-o P C*);
(*obs*, *h''*) ← *heap-copies a a'* (*map* (λ((*F*, *D*), *Tfm*). *CField D F*) *FDTs*) *h'*;
{(*h''*, ⌊(*NewHeapElem a'* (*Class-type C*) # *obs*, *a'*)⌋)}
}
| ⌊*Array-type T n*⌋ ⇒
let HA = *allocate h* (*Array-type T n*)
in if HA = {} *then* {(*h*, *None*)} *else do* {
(*h'*, *a'*) ← *HA*;
FDTs ← *set-of-pred* (*Fields-i-i-o P Object*);
(*obs*, *h''*) ← *heap-copies a a'* (*map* (λ((*F*, *D*), *Tfm*). *CField D F*) *FDTs* @ *map ACell* [*0*..<*n*]) *h'*;
{(*h''*, ⌊(*NewHeapElem a'* (*Array-type T n*) # *obs*, *a'*)⌋)}
}
| - ⇒ {})
by (*auto 4 3 elim*!: *execute.heap-clone.cases split*: *ty.splits prod.split-asm htype.splits intro*: *execute.heap-clone.intros simp add*: *eval-Fields-conv split-beta prod-eq-iff*)
(*auto simp add*: *eval-Fields-conv Bex-def*)

definition *red-external-aggr* ::
'm prog ⇒ *'thread-id* ⇒ *'addr* ⇒ *mname* ⇒ *'addr val list* ⇒ *'heap* ⇒
((*'addr*, *'thread-id*, *'heap*) *external-thread-action* × *'addr extCallRet* × *'heap*) *set*
where [*simp*]:
red-external-aggr P t a M vs h = *execute.red-external-aggr P t a M vs h*

lemma *red-external-aggr-code*:
red-external-aggr P t a M vs h =
(*if M* = *wait then*
let ad-t = *thread-id2addr t*
in {(⦃*Unlock*→*a*, *Lock*→*a*, *IsInterrupted t True*, *ClearInterrupt t*, *ObsInterrupted t*⦄, *execute.RetEXC InterruptedException*, *h*),
(⦃*Suspend a*, *Unlock*→*a*, *Lock*→*a*, *ReleaseAcquire*→*a*, *IsInterrupted t False*, *SyncUnlock a*⦄, *RetStaySame*, *h*),
(⦃*UnlockFail*→*a*⦄, *execute.RetEXC IllegalMonitorState*, *h*),
(⦃*Notified*⦄, *RetVal Unit*, *h*),
(⦃*WokenUp*, *ClearInterrupt t*, *ObsInterrupted t*⦄, *execute.RetEXC InterruptedException*, *h*)} ∪
(*if spurious-wakeups then* {(⦃*Unlock*→*a*, *Lock*→*a*, *ReleaseAcquire*→*a*, *IsInterrupted t False*, *SyncUnlock a*⦄, *RetVal Unit*, *h*)} *else* {})
else if M = *notify then*
{(⦃*Notify a*, *Unlock*→*a*, *Lock*→*a*⦄, *RetVal Unit*, *h*),
(⦃*UnlockFail*→*a*⦄, *execute.RetEXC IllegalMonitorState*, *h*)}
else if M = *notifyAll then*

```
      {(⦃NotifyAll a, Unlock→a, Lock→a ⦄, RetVal Unit, h),
       (⦃UnlockFail→a⦄, execute.RetEXC IllegalMonitorState, h)}
    else if M = clone then
      do {
        (h', obsa) ← heap-clone P h a;
        {case obsa of None ⇒ (ε, execute.RetEXC OutOfMemory, h')
          | Some (obs, a') ⇒ ((K$ [], [], [], [], [], obs), RetVal (Addr a'), h')}
      }
    else if M = hashcode then {(ε, RetVal (Intg (word-of-int (hash-addr a))), h)}
    else if M = print then {(⦃ExternalCall a M vs Unit⦄, RetVal Unit, h)}
    else if M = currentThread then {(ε, RetVal (Addr (thread-id2addr t)), h)}
    else if M = interrupted then
     {(⦃IsInterrupted t True, ClearInterrupt t, ObsInterrupted t⦄, RetVal (Bool True), h),
      (⦃IsInterrupted t False⦄, RetVal (Bool False), h)}
    else if M = yield then {(⦃Yield⦄, RetVal Unit, h)}
    else
      let T = ty-of-htype (the (typeof-addr h a))
      in if P ⊢ T ≤ Class Thread then
        let t-a = addr2thread-id a
        in if M = start then
             {(⦃NewThread t-a (the-Class T, run, a) h, ThreadStart t-a⦄, RetVal Unit, h),
              (⦃ThreadExists t-a True⦄, execute.RetEXC IllegalThreadState, h)}
           else if M = join then
             {(⦃Join t-a, IsInterrupted t False, ThreadJoin t-a⦄, RetVal Unit, h),
              (⦃IsInterrupted t True, ClearInterrupt t, ObsInterrupted t⦄, execute.RetEXC InterruptedEx-
ception, h)}
           else if M = interrupt then
             {(⦃ThreadExists t-a True, WakeUp t-a, Interrupt t-a, ObsInterrupt t-a⦄, RetVal Unit, h),
              (⦃ThreadExists t-a False⦄, RetVal Unit, h)}
           else if M = isInterrupted then
             {(⦃IsInterrupted t-a False⦄, RetVal (Bool False), h),
              (⦃IsInterrupted t-a True, ObsInterrupted t-a⦄, RetVal (Bool True), h)}
         else {(⦃⦄, undefined)}
    else {(⦃⦄, undefined)})
by (auto simp add: execute.red-external-aggr-def
  split del: option.splits) auto

end

lemmas [code] =
  heap-execute.heap-copy-loc-code
  heap-execute.heap-copies-code
  heap-execute.heap-clone-code
  heap-execute.red-external-aggr-code

end
```

9.8 An optimized JVM

```
theory JVMExec-Execute2
imports
  ../BV/BVNoTypeError
  ExternalCall-Execute
```

begin

This JVM must lookup the method declaration of the top call frame at every step to find the next instruction. It is more efficient to refine it such that the instruction list and the exception table are cached in the call frame. Even further, this theory adds keeps track of *drop pc ins*, whose head is the next instruction to execute.

locale *JVM-heap-execute* = *heap-execute* +
  constrains *addr2thread-id* :: ('*addr* :: *addr*) ⇒ '*thread-id*
  and *thread-id2addr* :: '*thread-id* ⇒ '*addr*
  and *spurious-wakeups* :: *bool*
  and *empty-heap* :: '*heap*
  and *allocate* :: '*heap* ⇒ *htype* ⇒ ('*heap* × '*addr*) *set*
  and *typeof-addr* :: '*heap* ⇒ '*addr* ⇒ *htype option*
  and *heap-read* :: '*heap* ⇒ '*addr* ⇒ *addr-loc* ⇒ '*addr val set*
  and *heap-write* :: '*heap* ⇒ '*addr* ⇒ *addr-loc* ⇒ '*addr val* ⇒ '*heap set*

sublocale *JVM-heap-execute* < *execute*: *JVM-heap-base*
  addr2thread-id thread-id2addr
  spurious-wakeups
  empty-heap allocate typeof-addr
  λh a ad v. v ∈ heap-read h a ad λh a ad v h'. h' ∈ heap-write h a ad v
.

type-synonym
  '*addr frame'* = ('*addr instr list* × '*addr instr list* × *ex-table*) × '*addr val list* × '*addr val list* × *cname* × *mname* × *pc*

type-synonym
  ('*addr*, '*heap*) *jvm-state'* = '*addr option* × '*heap* × '*addr frame'* *list*

type-synonym
  '*addr jvm-thread-state'* = '*addr option* × '*addr frame'* *list*

type-synonym
  ('*addr*, '*thread-id*, '*heap*) *jvm-thread-action'* = ('*addr*, '*thread-id*, '*addr jvm-thread-state'*,'*heap*) *Jinja-thread-action*

type-synonym
  ('*addr*, '*thread-id*, '*heap*) *jvm-ta-state'* = ('*addr*, '*thread-id*, '*heap*) *jvm-thread-action'* × ('*addr*, '*heap*) *jvm-state'*

fun *frame'-of-frame* :: '*addr jvm-prog* ⇒ '*addr frame* ⇒ '*addr frame'*
where
  frame'-of-frame P (*stk*, *loc*, *C*, *M*, *pc*) =
  ((*drop pc* (*instrs-of P C M*), *instrs-of P C M*, *ex-table-of P C M*), *stk*, *loc*, *C*, *M*, *pc*)

fun *jvm-state'-of-jvm-state* :: '*addr jvm-prog* ⇒ ('*addr*, '*heap*) *jvm-state* ⇒ ('*addr*, '*heap*) *jvm-state'*
where *jvm-state'-of-jvm-state P* (*xcp*, *h*, *frs*) = (*xcp*, *h*, *map* (*frame'-of-frame P*) *frs*)

fun *jvm-thread-state'-of-jvm-thread-state* :: '*addr jvm-prog* ⇒ '*addr jvm-thread-state* ⇒ '*addr jvm-thread-state'*
where
  jvm-thread-state'-of-jvm-thread-state P (*xcp*, *frs*) = (*xcp*, *map* (*frame'-of-frame P*) *frs*)

definition *jvm-thread-action'-of-jvm-thread-action* ::
  '*addr jvm-prog* ⇒ ('*addr*, '*thread-id*, '*heap*) *jvm-thread-action* ⇒ ('*addr*, '*thread-id*, '*heap*) *jvm-thread-action'*

where
jvm-thread-action'-of-jvm-thread-action P = convert-extTA (jvm-thread-state'-of-jvm-thread-state P)

fun *jvm-ta-state'-of-jvm-ta-state* ::
'addr jvm-prog ⇒ ('addr, 'thread-id, 'heap) jvm-ta-state ⇒ ('addr, 'thread-id, 'heap) jvm-ta-state'
where
jvm-ta-state'-of-jvm-ta-state P (ta, s) = (jvm-thread-action'-of-jvm-thread-action P ta, jvm-state'-of-jvm-state P s)

abbreviation (*input*) *frame-of-frame'* :: *'addr frame' ⇒ 'addr frame*
where *frame-of-frame' ≡ snd*

definition *jvm-state-of-jvm-state'* :: *('addr, 'heap) jvm-state' ⇒ ('addr, 'heap) jvm-state*
where [*simp*]:
jvm-state-of-jvm-state' = map-prod id (map-prod id (map frame-of-frame'))

definition *jvm-thread-state-of-jvm-thread-state'* :: *'addr jvm-thread-state' ⇒ 'addr jvm-thread-state*
where [*simp*]:
jvm-thread-state-of-jvm-thread-state' = map-prod id (map frame-of-frame')

definition *jvm-thread-action-of-jvm-thread-action'* ::
('addr, 'thread-id, 'heap) jvm-thread-action' ⇒ ('addr, 'thread-id, 'heap) jvm-thread-action
where [*simp*]:
jvm-thread-action-of-jvm-thread-action' = convert-extTA jvm-thread-state-of-jvm-thread-state'

definition *jvm-ta-state-of-jvm-ta-state'* ::
('addr, 'thread-id, 'heap) jvm-ta-state' ⇒ ('addr, 'thread-id, 'heap) jvm-ta-state
where [*simp*]:
jvm-ta-state-of-jvm-ta-state' = map-prod jvm-thread-action-of-jvm-thread-action' jvm-state-of-jvm-state'

fun *frame'-ok* :: *'addr jvm-prog ⇒ 'addr frame' ⇒ bool*
where
frame'-ok P ((ins', insxt), stk, loc, C, M, pc) ⟷
ins' = drop pc (instrs-of P C M) ∧ insxt = snd (snd (the (snd (snd (snd (method P C M))))))

lemma *frame'-ok-frame'-of-frame* [*iff*]:
frame'-ok P (frame'-of-frame P f)
by(*cases f*)(*simp*)

lemma *frames'-ok-inverse* [*simp*]:
∀ x∈set frs. frame'-ok P x ⟹ map (frame'-of-frame P ∘ frame-of-frame') frs = frs
by(*rule map-idI*) *auto*

fun *jvm-state'-ok* :: *'addr jvm-prog ⇒ ('addr, 'heap) jvm-state' ⇒ bool*
where *jvm-state'-ok P (xcp, h, frs) = (∀ f ∈ set frs. frame'-ok P f)*

lemma *jvm-state'-ok-jvm-state'-of-jvm-state* [*iff*]:
jvm-state'-ok P (jvm-state'-of-jvm-state P s)
by(*cases s*) *simp*

fun *jvm-thread-state'-ok* :: *'addr jvm-prog ⇒ 'addr jvm-thread-state' ⇒ bool*
where *jvm-thread-state'-ok P (xcp, frs) ⟷ (∀ f ∈ set frs. frame'-ok P f)*

lemma *jvm-thread-state'-ok-jvm-thread-state'-of-jvm-thread-state* [*iff*]:

jvm-thread-state'-ok P (jvm-thread-state'-of-jvm-thread-state P s)
by(*cases s*) *simp*

definition *jvm-thread-action'-ok* :: *'addr jvm-prog* ⇒ (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action'* ⇒ *bool*
where *jvm-thread-action'-ok P ta* ⟷ (∀ *nt* ∈ *set* ⦃*ta*⦄$_t$. ∀ *t x h*. *nt* = *NewThread t x h* ⟶ *jvm-thread-state'-ok P x*)

lemma *jvm-thread-action'-ok-jvm-thread-action'-of-jvm-thread-action* [*iff*]:
jvm-thread-action'-ok P (jvm-thread-action'-of-jvm-thread-action P ta)
by(*cases ta*)(*fastforce dest*: *sym simp add*: *jvm-thread-action'-ok-def jvm-thread-action'-of-jvm-thread-action-def*)

lemma *jvm-thread-action'-ok-ε* [*simp*]: *jvm-thread-action'-ok P ε*
by(*simp add*: *jvm-thread-action'-ok-def*)

fun *jvm-ta-state'-ok* :: *'addr jvm-prog* ⇒ (*'addr*, *'thread-id*, *'heap*) *jvm-ta-state'* ⇒ *bool*
where *jvm-ta-state'-ok P* (*ta*, *s*) ⟷ *jvm-thread-action'-ok P ta* ∧ *jvm-state'-ok P s*

lemma *jvm-ta-state'-ok-jvm-ta-state'-of-jvm-ta-state* [*iff*]:
jvm-ta-state'-ok P (jvm-ta-state'-of-jvm-ta-state P tas)
by(*cases tas*)(*simp*)

lemma *frame-of-frame'-inverse* [*simp*]: *frame-of-frame'* ∘ *frame'-of-frame P* = *id*
by(*clarsimp simp add*: *fun-eq-iff*)

lemma *convert-new-thread-action-frame-of-frame'-inverse* [*simp*]:
convert-new-thread-action (*map-prod id* (*map frame-of-frame'*)) ∘ *convert-new-thread-action* (*jvm-thread-state'-of-*
P) = *id*
by(*auto intro*!: *convert-new-thread-action-eqI simp add*: *fun-eq-iff List.map.id*)

primrec *extRet2JVM'* ::
'addr instr list ⇒ *'addr instr list* ⇒ *ex-table*
⇒ *nat* ⇒ *'heap* ⇒ *'addr val list* ⇒ *'addr val list* ⇒ *cname* ⇒ *mname* ⇒ *pc* ⇒ *'addr frame' list*
⇒ *'addr extCallRet* ⇒ (*'addr*, *'heap*) *jvm-state'*
where
extRet2JVM' ins' ins xt n h stk loc C M pc frs (*RetVal v*) = (*None*, *h*, ((*tl ins'*, *ins*, *xt*), *v* # *drop* (*Suc n*) *stk*, *loc*, *C*, *M*, *pc* + *1*) # *frs*)
| *extRet2JVM' ins' ins xt n h stk loc C M pc frs* (*RetExc a*) = (⌊*a*⌋, *h*, ((*ins'*, *ins*, *xt*), *stk*, *loc*, *C*, *M*, *pc*) # *frs*)
| *extRet2JVM' ins' ins xt n h stk loc C M pc frs RetStaySame* = (*None*, *h*, ((*ins'*, *ins*, *xt*), *stk*, *loc*, *C*, *M*, *pc*) # *frs*)

definition *extNTA2JVM'* :: *'addr jvm-prog* ⇒ (*cname* × *mname* × *'addr*) ⇒ *'addr jvm-thread-state'*
where *extNTA2JVM' P* ≡ (λ(*C*, *M*, *a*). *let* (*D*,*Ts*,*T*,*meth*) = *method P C M*; (*mxs*,*mxl0*,*ins*,*xt*) = *the meth*
in (*None*, [((*ins*, *ins*, *xt*), [],*Addr a* # *replicate mxl0 undefined-value*, *D*, *M*, *0*)]))

abbreviation *extTA2JVM'* ::
'addr jvm-prog ⇒ (*'addr*, *'thread-id*, *'heap*) *external-thread-action* ⇒ (*'addr*, *'thread-id*, *'heap*) *jvm-thread-action'*
where *extTA2JVM' P* ≡ *convert-extTA* (*extNTA2JVM' P*)

lemma *jvm-state'-ok-extRet2JVM'* [*simp*]:
assumes [*simp*]: *ins* = *instrs-of P C M xt* = *ex-table-of P C M* ∀ *f* ∈ *set frs*. *frame'-ok P f*

shows *jvm-state'-ok P (extRet2JVM' (drop pc ins) ins xt n h stk loc C M pc frs va)*
by(*cases va*)(*simp-all add*: *drop-tl drop-Suc*)

lemma *jvm-state'-of-jvm-state-extRet2JVM* [*simp*]:
assumes [*simp*]: *ins = instrs-of P C M xt = ex-table-of P C M* ∀ *f* ∈ *set frs. frame'-ok P f*
shows
jvm-state'-of-jvm-state P (extRet2JVM n h' stk loc C M pc (map frame-of-frame' frs) va) =
extRet2JVM' (drop pc (instrs-of P C M)) ins xt n h' stk loc C M pc frs va
by(*cases va*)(*simp-all add*: *drop-tl drop-Suc*)

lemma *extRet2JVM'-extRet2JVM* [*simp*]:
jvm-state-of-jvm-state' (extRet2JVM' ins' ins xt n h' stk loc C M pc frs va) =
extRet2JVM n h' stk loc C M pc (map frame-of-frame' frs) va
by(*cases va*) *simp-all*

lemma *jvm-ta-state'-ok-inverse*:
assumes *jvm-ta-state'-ok P tas*
shows *jvm-ta-state-of-jvm-ta-state' tas* ∈ *A* ⟷ *tas* ∈ *jvm-ta-state'-of-jvm-ta-state P ' A*
using *assms*
apply(*cases tas*)
apply(*fastforce simp add*: *o-def jvm-thread-action'-of-jvm-thread-action-def jvm-thread-action'-ok-def intro*!: *map-idI*[*symmetric*] *map-idI convert-new-thread-action-eqI dest*: *bspec intro*!: *rev-image-eqI elim*!: *rev-iffD1*[*OF - arg-cong*[**where** *f*=λ*x*. *x* : *A*]])
done

context *JVM-heap-execute* **begin**

primrec *exec-instr* ::
'addr instr list ⇒ *'addr instr list* ⇒ *ex-table*
⇒ *'addr instr* ⇒ *'addr jvm-prog* ⇒ *'thread-id* ⇒ *'heap* ⇒ *'addr val list* ⇒ *'addr val list*
⇒ *cname* ⇒ *mname* ⇒ *pc* ⇒ *'addr frame' list*
⇒ (*('addr, 'thread-id, 'heap) jvm-ta-state'*) *set*
where
exec-instr ins' ins xt (Load n) P t h stk loc C$_0$ *M*$_0$ *pc frs =*
{(ε, (*None, h,* ((*tl ins', ins, xt*), (*loc* ! *n*) # *stk, loc, C*$_0$, *M*$_0$, *pc*+1)#*frs*))}
| *exec-instr ins' ins xt (Store n) P t h stk loc C*$_0$ *M*$_0$ *pc frs =*
{(ε, (*None, h,* ((*tl ins', ins, xt*), *tl stk, loc*[*n*:=*hd stk*], *C*$_0$, *M*$_0$, *pc*+1)#*frs*))}
| *exec-instr ins' ins xt (Push v) P t h stk loc C*$_0$ *M*$_0$ *pc frs =*
{(ε, (*None, h,* ((*tl ins', ins, xt*), *v* # *stk, loc, C*$_0$, *M*$_0$, *pc*+1)#*frs*))}
| *exec-instr ins' ins xt (New C) P t h stk loc C*$_0$ *M*$_0$ *pc frs =*
(*let HA = allocate h (Class-type C)*
in if HA = {} *then* {(ε, ⌊*execute.addr-of-sys-xcpt OutOfMemory*⌋, *h,* ((*ins', ins, xt*), *stk, loc, C*$_0$, *M*$_0$, *pc*) # *frs*)}
else do { (*h', a*) ← *HA*;
{(⦃*NewHeapElem a* (*Class-type C*)⦄, *None, h',* ((*tl ins', ins, xt*), *Addr a* # *stk, loc, C*$_0$, *M*$_0$, *pc* + 1)#*frs*)}})
| *exec-instr ins' ins xt (NewArray T) P t h stk loc C0 M0 pc frs =*
(*let si = the-Intg* (*hd stk*);
i = nat (*sint si*)
in if si <*s* 0
then {(ε, ⌊*execute.addr-of-sys-xcpt NegativeArraySize*⌋, *h,* ((*ins', ins, xt*), *stk, loc, C0, M0, pc*) # *frs*)}

```
      else let HA = allocate h (Array-type T i) in
       if HA = {} then {(ε, ⌊execute.addr-of-sys-xcpt OutOfMemory⌋, h, ((ins', ins, xt), stk, loc, C0,
M0, pc) # frs)}
        else do { (h', a) ← HA;
             {(⦇NewHeapElem a (Array-type T i) ⦈, None, h', ((tl ins', ins, xt), Addr a # tl stk, loc,
C0, M0, pc + 1) # frs)}})
| exec-instr ins' ins xt ALoad P t h stk loc C0 M0 pc frs =
  (let va = hd (tl stk)
   in (if va = Null then {(ε, ⌊execute.addr-of-sys-xcpt NullPointer⌋, h, ((ins', ins, xt), stk, loc, C0,
M0, pc) # frs)}
      else
       let i = the-Intg (hd stk);
           a = the-Addr va;
           len = alen-of-htype (the (typeof-addr h a))
       in if i <s 0 ∨ int len ≤ sint i then
            {(ε, ⌊execute.addr-of-sys-xcpt ArrayIndexOutOfBounds⌋, h, ((ins', ins, xt), stk, loc, C0,
M0, pc) # frs)}
          else do {
            v ← heap-read h a (ACell (nat (sint i)));
            {(⦇ReadMem a (ACell (nat (sint i))) v⦈, None, h, ((tl ins', ins, xt), v # tl (tl stk), loc,
C0, M0, pc + 1) # frs)}
          }))
| exec-instr ins' ins xt AStore P t h stk loc C0 M0 pc frs =
  (let ve = hd stk;
       vi = hd (tl stk);
       va = hd (tl (tl stk))
   in (if va = Null then {(ε, ⌊execute.addr-of-sys-xcpt NullPointer⌋, h, ((ins', ins, xt), stk, loc, C0,
M0, pc) # frs)}
       else (let i = the-Intg vi;
                 idx = nat (sint i);
                 a = the-Addr va;
                 hT = the (typeof-addr h a);
                 T = ty-of-htype hT;
                 len = alen-of-htype hT;
                 U = the (execute.typeof-h h ve)
             in (if i <s 0 ∨ int len ≤ sint i then
                   {(ε, ⌊execute.addr-of-sys-xcpt ArrayIndexOutOfBounds⌋, h, ((ins', ins, xt), stk, loc,
C0, M0, pc) # frs)}
                 else if P ⊢ U ≤ the-Array T then
                   do {
                     h' ← heap-write h a (ACell idx) ve;
                     {(⦇WriteMem a (ACell idx) ve⦈, None, h', ((tl ins', ins, xt), tl (tl (tl stk)), loc,
C0, M0, pc+1) # frs)}
                   }
                 else {(ε, (⌊execute.addr-of-sys-xcpt ArrayStore⌋, h, ((ins', ins, xt), stk, loc, C0, M0, pc)
# frs))})))))
| exec-instr ins' ins xt ALength P t h stk loc C0 M0 pc frs =
  {(ε, (let va = hd stk
        in if va = Null
           then (⌊execute.addr-of-sys-xcpt NullPointer⌋, h, ((ins', ins, xt), stk, loc, C0, M0, pc) # frs)
           else (None, h, ((tl ins', ins, xt), Intg (word-of-int (int (alen-of-htype (the (typeof-addr h
(the-Addr va)))))) # tl stk, loc, C0, M0, pc+1) # frs)))}
| exec-instr ins' ins xt (Getfield F C) P t h stk loc C0 M0 pc frs =
  (let v = hd stk
```

```
   in if v = Null then {(ε, ⌊execute.addr-of-sys-xcpt NullPointer⌋, h, ((ins', ins, xt), stk, loc, C0, M0,
pc) # frs)}
      else let a = the-Addr v
           in do {
              v' ← heap-read h a (CField C F);
              {(⦃ReadMem a (CField C F) v'⦄, None, h, ((tl ins', ins, xt), v' # (tl stk), loc, C0, M0,
pc + 1) # frs)}
           })
| exec-instr ins' ins xt (Putfield F C) P t h stk loc C0 M0 pc frs =
  (let v = hd stk;
       r = hd (tl stk)
   in if r = Null then {(ε, ⌊execute.addr-of-sys-xcpt NullPointer⌋, h, ((ins', ins, xt), stk, loc, C0, M0,
pc) # frs)}
      else let a = the-Addr r
           in do {
              h' ← heap-write h a (CField C F) v;
              {(⦃WriteMem a (CField C F) v⦄, None, h', ((tl ins', ins, xt), tl (tl stk), loc, C0, M0, pc
+ 1) # frs)}
           })
| exec-instr ins' ins xt (CAS F C) P t h stk loc C0 M0 pc frs =
  (let v'' = hd stk; v' = hd (tl stk); v = hd (tl (tl stk))
   in if v = Null then {(ε, ⌊execute.addr-of-sys-xcpt NullPointer⌋, h, ((ins', ins, xt), stk, loc, C0, M0,
pc) # frs)}
      else let a = the-Addr v
           in do {
              v''' ← heap-read h a (CField C F);
              if v''' = v' then do {
                h' ← heap-write h a (CField C F) v'';
                {(⦃ReadMem a (CField C F) v', WriteMem a (CField C F) v''⦄, None, h', ((tl ins', ins,
xt), Bool True # tl (tl (tl stk)), loc, C0, M0, pc + 1) # frs)}
              } else {(⦃ReadMem a (CField C F) v'''⦄, None, h, ((tl ins', ins, xt), Bool False # tl (tl
(tl stk)), loc, C0, M0, pc + 1) # frs)}
           })
| exec-instr ins' ins xt (Checkcast T) P t h stk loc C0 M0 pc frs =
   {(ε, let U = the (typeof_h (hd stk))
        in if P ⊢ U ≤ T then (None, h, ((tl ins', ins, xt), stk, loc, C0, M0, pc + 1) # frs)
           else (⌊execute.addr-of-sys-xcpt ClassCast⌋, h, ((ins', ins, xt), stk, loc, C0, M0, pc) # frs))}
| exec-instr ins' ins xt (Instanceof T) P t h stk loc C0 M0 pc frs =
   {(ε, None, h, ((tl ins', ins, xt), Bool (hd stk ≠ Null ∧ P ⊢ the (typeof_h (hd stk)) ≤ T) # tl stk,
loc, C0, M0, pc + 1) # frs)}
| exec-instr ins' ins xt (Invoke M n) P t h stk loc C0 M0 pc frs =
   (let r = stk ! n
    in (if r = Null then {(ε, ⌊execute.addr-of-sys-xcpt NullPointer⌋, h, ((ins', ins, xt), stk, loc, C0,
M0, pc) # frs)}
      else (let ps = rev (take n stk);
                a = the-Addr r;
                T = the (typeof-addr h a);
                (D,Ts,T,meth)= method P (class-type-of T) M
           in case meth of
                Native ⇒
                  do {
                    (ta, va, h') ← red-external-aggr P t a M ps h;
                    {(extTA2JVM' P ta, extRet2JVM' ins' ins xt n h' stk loc C0 M0 pc frs va)}
                  }
```

```
                | ⌊(mxs,mxl0,ins'',xt'')⌋ ⇒
                    let f' = ((ins'', ins'', xt''), [],[r]@ps@(replicate mxl0 undefined-value),D,M,0)
                    in {(ε, None, h, f' # ((ins', ins, xt), stk, loc, C0, M0, pc) # frs)}))))
| exec-instr ins' ins xt Return P t h stk0 loc0 C0 M0 pc frs =
  {(ε, (if frs=[] then (None, h, [])
        else
          let v = hd stk0;
              ((ins', ins, xt), stk,loc,C,m,pc) = hd frs;
               n = length (fst (snd (method P C0 M0)))
          in (None, h, ((tl ins', ins, xt), v#(drop (n+1) stk),loc,C,m,pc+1)#tl frs)))}
| exec-instr ins' ins xt Pop P t h stk loc C0 M0 pc frs =
  {(ε, (None, h, ((tl ins', ins, xt), tl stk, loc, C0, M0, pc+1)#frs))}
| exec-instr ins' ins xt Dup P t h stk loc C0 M0 pc frs =
  {(ε, (None, h, ((tl ins', ins, xt), hd stk # stk, loc, C0, M0, pc+1)#frs))}
| exec-instr ins' ins xt Swap P t h stk loc C0 M0 pc frs =
  {(ε, (None, h, ((tl ins', ins, xt), hd (tl stk) # hd stk # tl (tl stk), loc, C0, M0, pc+1)#frs))}
| exec-instr ins' ins xt (BinOpInstr bop) P t h stk loc C0 M0 pc frs =
  {(ε,
    case the (execute.binop bop (hd (tl stk)) (hd stk)) of
      Inl v ⇒ (None, h, ((tl ins', ins, xt), v # tl (tl stk), loc, C0, M0, pc + 1) # frs)
    | Inr a ⇒ (Some a, h, ((ins', ins, xt), stk, loc, C0, M0, pc) # frs))}
| exec-instr ins' ins xt (IfFalse i) P t h stk loc C0 M0 pc frs =
  {(ε, (let pc' = if hd stk = Bool False then nat(int pc+i) else pc+1
        in (None, h, ((drop pc' ins, ins, xt), tl stk, loc, C0, M0, pc')#frs)))}
| exec-instr ins' ins xt (Goto i) P t h stk loc C0 M0 pc frs =
  {let pc' = nat(int pc+i)
   in (ε, (None, h, ((drop pc' ins, ins, xt), stk, loc, C0, M0, pc')#frs))}
| exec-instr ins' ins xt ThrowExc P t h stk loc C0 M0 pc frs =
  {(ε, (let xp' = if hd stk = Null then ⌊execute.addr-of-sys-xcpt NullPointer⌋ else ⌊the-Addr(hd stk)⌋
        in (xp', h, ((ins', ins, xt), stk, loc, C0, M0, pc)#frs)))}
| exec-instr ins' ins xt MEnter P t h stk loc C0 M0 pc frs =
  {let v = hd stk
   in if v = Null
      then (ε, ⌊execute.addr-of-sys-xcpt NullPointer⌋, h, ((ins', ins, xt), stk, loc, C0, M0, pc) # frs)
      else (⦃Lock→the-Addr v, SyncLock (the-Addr v)⦄, None, h, ((tl ins', ins, xt), tl stk, loc, C0, M0,
pc + 1) # frs)}
| exec-instr ins' ins xt MExit P t h stk loc C0 M0 pc frs =
  (let v = hd stk
   in if v = Null
      then {(ε, ⌊execute.addr-of-sys-xcpt NullPointer⌋, h, ((ins', ins, xt), stk, loc, C0, M0, pc)#frs)}
      else {(⦃Unlock→the-Addr v, SyncUnlock (the-Addr v)⦄, None, h, ((tl ins', ins, xt), tl stk, loc, C0,
M0, pc + 1) # frs),
            (⦃UnlockFail→the-Addr v⦄, ⌊execute.addr-of-sys-xcpt IllegalMonitorState⌋, h, ((ins', ins, xt),
stk, loc, C0, M0, pc) # frs)})

fun exception-step :: 'addr jvm-prog ⇒ 'addr ⇒ 'heap ⇒ 'addr frame' ⇒ 'addr frame' list ⇒ ('addr,
'heap) jvm-state'
where
  exception-step P a h ((ins', ins, xt), stk, loc, C, M, pc) frs =
   (case match-ex-table P (execute.cname-of h a) pc xt of
         None ⇒ (⌊a⌋, h, frs)
       | Some (pc', d) ⇒ (None, h, ((drop pc' ins, ins, xt), Addr a # drop (size stk − d) stk, loc, C,
M, pc') # frs))
```

fun *exec* :: *'addr jvm-prog* ⇒ *'thread-id* ⇒ (*'addr*, *'heap*) *jvm-state'* ⇒ (*'addr*, *'thread-id*, *'heap*) *jvm-ta-state' set*
where
exec P t (*xcp*, *h*, []) = {}
| *exec P t* (*None*, *h*, ((*ins'*, *ins*, *xt*), *stk*, *loc*, *C*, *M*, *pc*) # *frs*) =
exec-instr ins' ins xt (*hd ins'*) *P t h stk loc C M pc frs*
| *exec P t* (⌊*a*⌋, *h*, *fr* # *frs*) = {(*ε*, *exception-step P a h fr frs*)}

definition *exec-1* ::
'addr jvm-prog ⇒ *'thread-id* ⇒ (*'addr*, *'heap*) *jvm-state'*
⇒ ((*'addr*, *'thread-id*, *'heap*) *jvm-thread-action'* × (*'addr*, *'heap*) *jvm-state'*) *Predicate.pred*
where *exec-1 P t σ* = *pred-of-set* (*exec P t σ*)

lemma *check-exec-instr-ok*:
assumes *wf*: *wf-prog wf-md P*
and *execute.check-instr i P h stk loc C M pc* (*map frame-of-frame' frs*)
and *P* ⊢ *C sees M*:*Ts*→*T* = ⌊*m*⌋ *in D*
and *jvm-state'-ok P* (*None*, *h*, ((*ins'*, *ins*, *xt*), *stk*, *loc*, *C*, *M*, *pc*) # *frs*)
and *tas* ∈ *exec-instr ins' ins xt i P t h stk loc C M pc frs*
shows *jvm-ta-state'-ok P tas*
proof −
note [*simp*] = *drop-Suc drop-tl split-beta jvm-thread-action'-ok-def has-method-def*
note [*split*] = *if-split-asm sum.split*
from *assms* **show** *?thesis*
proof(*cases i*)
case *Return*
thus *?thesis* **using** *assms* **by**(*cases frs*) *auto*
next
case *Invoke*
thus *?thesis* **using** *assms*
apply(*cases m*)
apply(*auto simp add*: *extNTA2JVM'-def dest*: *sees-method-idemp execute.red-external-aggr-new-thread-sub-thread sub-Thread-sees-run*[*OF wf*])
apply(*drule execute.red-external-aggr-new-thread-sub-thread*, *clarsimp*, *clarsimp*, *assumption*, *clarsimp*)
apply(*drule sub-Thread-sees-run*[*OF wf*], *clarsimp*)
apply(*fastforce dest*: *sees-method-idemp*)
apply(*drule execute.red-external-aggr-new-thread-sub-thread*, *clarsimp*, *clarsimp*, *assumption*, *clarsimp*)
apply(*drule sub-Thread-sees-run*[*OF wf*], *clarsimp*)
apply(*fastforce dest*: *sees-method-idemp*)
done
next
case *Goto* **thus** *?thesis* **using** *assms*
by(*cases m*) *simp*
next
case *IfFalse* **thus** *?thesis* **using** *assms*
by(*cases m*) *simp*
qed(*auto*)
qed

lemma *check-exec-instr-complete*:
assumes *wf*: *wf-prog wf-md P*
and *execute.check-instr i P h stk loc C M pc* (*map frame-of-frame' frs*)

and $P \vdash C$ *sees* M:$Ts \rightarrow T = \lfloor m \rfloor$ *in* D
and *jvm-state'-ok* P (*None*, h, ((*ins'*, *ins*, *xt*), *stk*, *loc*, C, M, *pc*) # *frs*)
and *tas* ∈ *execute.exec-instr i P t h stk loc C M pc* (*map frame-of-frame' frs*)
shows *jvm-ta-state'-of-jvm-ta-state P tas* ∈ *exec-instr ins' ins xt i P t h stk loc C M pc frs*
proof −
note [*simp*] =
drop-Suc drop-tl split-beta jvm-thread-action'-ok-def jvm-thread-action'-of-jvm-thread-action-def has-method-def
ta-upd-simps map-tl
note [*split*] = *if-split-asm sum.split*
from *assms* **show** *?thesis*
proof(*cases i*)
case *Return* **thus** *?thesis* **using** *assms* **by**(*cases frs*) *auto*
next
case *Goto* **thus** *?thesis* **using** *assms*
by(*cases m*) *simp*
next
case *IfFalse* **thus** *?thesis* **using** *assms*
by(*cases m*) *simp*
next
case *Invoke* **thus** *?thesis* **using** *assms*
apply(*cases m*)
apply(*auto intro*!: *rev-bexI convert-new-thread-action-eqI simp add*: *extNTA2JVM'-def extNTA2JVM-def dest*: *execute.red-external-aggr-new-thread-sub-thread sub-Thread-sees-run*[*OF wf*] *sees-method-idemp*)
apply(*drule execute.red-external-aggr-new-thread-sub-thread*, *clarsimp*, *clarsimp*, *assumption*, *clarsimp*)
apply(*drule sub-Thread-sees-run*[*OF wf*], *clarsimp*)
apply(*fastforce dest*: *sees-method-idemp*)
apply(*drule execute.red-external-aggr-new-thread-sub-thread*, *clarsimp*, *clarsimp*, *assumption*, *clarsimp*)
apply(*drule sub-Thread-sees-run*[*OF wf*], *clarsimp*)
apply(*fastforce dest*: *sees-method-idemp*)
done
qed(*auto 4 4*)
qed

lemma *check-exec-instr-refine*:
assumes *wf*: *wf-prog wf-md P*
and *execute.check-instr i P h stk loc C M pc* (*map frame-of-frame' frs*)
and $P \vdash C$ *sees* M:$Ts \rightarrow T = \lfloor m \rfloor$ *in* D
and *jvm-state'-ok* P (*None*, h, ((*ins'*, *ins*, *xt*), *stk*, *loc*, C, M, *pc*) # *frs*)
and *tas* ∈ *exec-instr ins' ins xt i P t h stk loc C M pc frs*
shows *tas* ∈ *jvm-ta-state'-of-jvm-ta-state P* ‘ *execute.exec-instr i P t h stk loc C M pc* (*map frame-of-frame' frs*)
proof −
note [*simp*] =
drop-Suc drop-tl split-beta jvm-thread-action'-ok-def jvm-thread-action'-of-jvm-thread-action-def has-method-def
ta-upd-simps map-tl o-def
note [*split*] = *if-split-asm sum.split*
from *assms* **have** *jvm-ta-state-of-jvm-ta-state' tas* ∈ *execute.exec-instr i P t h stk loc C M pc* (*map frame-of-frame' frs*)
proof (*cases i*)
case *Invoke* **thus** *?thesis* **using** *assms*

by(*fastforce simp add*: *extNTA2JVM'-def extNTA2JVM-def split-def extRet2JVM'-extRet2JVM*[*simplified*])
next
case *Return* **thus** *?thesis* **using** *assms* **by**(*auto simp add*: *neq-Nil-conv*)
qed (*auto cong del*: *image-cong-simp*)
also from *assms* **have** *ok'*: *jvm-ta-state'-ok P tas* **by**(*rule check-exec-instr-ok*)
hence *jvm-ta-state-of-jvm-ta-state' tas* ∈ *execute.exec-instr i P t h stk loc C M pc* (*map frame-of-frame' frs*) ⟷
tas ∈ *jvm-ta-state'-of-jvm-ta-state P ‘ execute.exec-instr i P t h stk loc C M pc* (*map frame-of-frame' frs*)
by(*rule jvm-ta-state'-ok-inverse*)
finally show *?thesis* **.**
qed

lemma *exception-step-ok*:
assumes *frame'-ok P fr* ∀ *f*∈*set frs. frame'-ok P f*
shows *jvm-state'-ok P* (*exception-step P a h fr frs*)
and *exception-step P a h fr frs* = *jvm-state'-of-jvm-state P* (*execute.exception-step P a h* (*snd fr*) (*map frame-of-frame' frs*))
using *assms*
by(*cases fr*, *case-tac the* (*snd* (*snd* (*snd* (*method P d e*)))), *clarsimp*)+

lemma *exec-step-conv*:
assumes *wf-prog wf-md P*
and *jvm-state'-ok P s*
and *execute.check P* (*jvm-state-of-jvm-state' s*)
shows *exec P t s* = *jvm-ta-state'-of-jvm-ta-state P ‘ execute.exec P t* (*jvm-state-of-jvm-state' s*)
using *assms*
apply(*cases s*)
apply(*rename-tac xcp h frs*)
apply(*case-tac frs*)
apply(*simp*)
apply(*case-tac xcp*)
prefer *2*
apply(*simp add*: *jvm-thread-action'-of-jvm-thread-action-def exception-step-ok*)
apply(*clarsimp simp add*: *execute.check-def*)
apply(*rule equalityI*)
apply(*clarsimp simp add*: *has-method-def*)
apply(*erule* (*2*) *check-exec-instr-refine*)
apply *fastforce*
apply(*simp add*: *hd-drop-conv-nth*)
apply(*clarsimp simp add*: *has-method-def*)
apply(*drule* (*2*) *check-exec-instr-complete*)
apply *fastforce*
apply(*assumption*)
apply(*simp add*: *hd-drop-conv-nth*)
done

lemma *exec-step-ok*:
assumes *wf-prog wf-md P*
and *jvm-state'-ok P s*
and *execute.check P* (*jvm-state-of-jvm-state' s*)
and *tas* ∈ *exec P t s*
shows *jvm-ta-state'-ok P tas*
using *assms*

apply(*cases s*)
apply(*rename-tac xcp h frs*)
apply(*case-tac frs*)
apply *simp*
apply(*rename-tac fr frs'*)
apply(*case-tac xcp*)
apply(*clarsimp simp add*: *execute.check-def has-method-def*)
apply(*erule* (*2*) *check-exec-instr-ok*)
apply *fastforce*
apply(*simp add*: *hd-drop-conv-nth*)
apply(*case-tac fr*)
apply(*rename-tac cache stk loc C M pc*)
apply(*case-tac the* (*snd* (*snd* (*snd* (*method P C M*)))))
apply *auto*
done

end

locale *JVM-heap-execute-conf-read* = *JVM-heap-execute* +
execute: *JVM-conf-read*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr
λh a ad v. *v* ∈ *heap-read h a ad λh a ad v h'*. *h'* ∈ *heap-write h a ad v*
+
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*
and *thread-id2addr* :: *'thread-id* ⇒ *'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap* ⇒ *htype* ⇒ (*'heap* × *'addr*) *set*
and *typeof-addr* :: *'heap* ⇒ *'addr* ⇒ *htype option*
and *heap-read* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val set*
and *heap-write* :: *'heap* ⇒ *'addr* ⇒ *addr-loc* ⇒ *'addr val* ⇒ *'heap set*
and *hconf* :: *'heap* ⇒ *bool*
and *P* :: *'addr jvm-prog*
begin

lemma *exec-correct-state*:
assumes *wt*: *wf-jvm-prog*$_\Phi$ *P*
and *correct*: *execute.correct-state* Φ *t* (*jvm-state-of-jvm-state' s*)
and *ok*: *jvm-state'-ok P s*
shows *exec P t s* = *jvm-ta-state'-of-jvm-ta-state P* ‘ *execute.exec P t* (*jvm-state-of-jvm-state' s*)
(**is** *?thesis1*)
and (*ta*, *s'*) ∈ *exec P t s* ⟹ *execute.correct-state* Φ *t* (*jvm-state-of-jvm-state' s'*) (**is** - ⟹ *?thesis2*)
and *tas* ∈ *exec P t s* ⟹ *jvm-ta-state'-ok P tas*
proof −
from *wt* **obtain** *wf-md* **where** *wf*: *wf-prog wf-md P* **by**(*blast dest*: *wt-jvm-progD*)
from *execute.no-type-error*[*OF wt correct*]
have *check*: *execute.check P* (*jvm-state-of-jvm-state' s*)
by(*simp add*: *execute.exec-d-def split*: *if-split-asm*)
with *wf ok* **show** *eq*: *?thesis1* **by**(*rule exec-step-conv*)

{ **fix** *tas*

assume *tas* ∈ *exec P t s*
with *wf ok check* **show** *jvm-ta-state′-ok P tas*
by(*rule exec-step-ok*) **}**
note *this*[*of* (*ta*, *s′*)]
moreover
assume (*ta*, *s′*) ∈ *exec P t s*
moreover
hence (*ta*, *s′*) ∈ *jvm-ta-state′-of-jvm-ta-state P ‘ execute.exec P t* (*jvm-state-of-jvm-state′ s*)
unfolding *eq* **by** *simp*
ultimately have *jvm-ta-state-of-jvm-ta-state′* (*ta*, *s′*) ∈ *execute.exec P t* (*jvm-state-of-jvm-state′ s*)
using *jvm-ta-state′-ok-inverse*[*of P* (*ta*, *s′*)] **by** *blast*
hence *execute.exec-1 P t* (*jvm-state-of-jvm-state′ s*) (*jvm-thread-action-of-jvm-thread-action′ ta*) (*jvm-state-of-jvm-state′ s′*)
by(*simp add*: *execute.exec-1-iff*)
with *wt correct* **show** *?thesis2* **by**(*rule execute.BV-correct-1*)
qed

end

lemmas [*code*] =
JVM-heap-execute.exec-instr.simps
JVM-heap-execute.exception-step.simps
JVM-heap-execute.exec.simps
JVM-heap-execute.exec-1-def

end

theory *JVM-Execute2*
imports
SC-Schedulers
JVMExec-Execute2
../BV/BVProgressThreaded
begin

abbreviation *sc-heap-read-cset* :: *heap* ⇒ *addr* ⇒ *addr-loc* ⇒ *addr val set*
where *sc-heap-read-cset h ad al* ≡ *set-of-pred* (*sc-heap-read-i-i-i-o h ad al*)

abbreviation *sc-heap-write-cset* :: *heap* ⇒ *addr* ⇒ *addr-loc* ⇒ *addr val* ⇒ *heap set*
where *sc-heap-write-cset h ad al v* ≡ *set-of-pred* (*sc-heap-write-i-i-i-i-o h ad al v*)

interpretation *sc*:
JVM-heap-execute
addr2thread-id
thread-id2addr
sc-spurious-wakeups
sc-empty
sc-allocate P
sc-typeof-addr
sc-heap-read-cset
sc-heap-write-cset
rewrites ⋀*h ad al v*. *v* ∈ *sc-heap-read-cset h ad al* ≡ *sc-heap-read h ad al v*
and ⋀*h ad al v h′*. *h′* ∈ *sc-heap-write-cset h ad al v* ≡ *sc-heap-write h ad al v h′*
for *P*

apply(*simp-all add*: *eval-sc-heap-read-i-i-i-o eval-sc-heap-write-i-i-i-i-o*)
done

interpretation *sc*:
JVM-heap-execute-conf-read
addr2thread-id
thread-id2addr
sc-spurious-wakeups
sc-empty
sc-allocate P
sc-typeof-addr
sc-heap-read-cset
sc-heap-write-cset
sc-hconf P
P
rewrites $\bigwedge h\ ad\ al\ v.\ v \in$ *sc-heap-read-cset h ad al* $\equiv$ *sc-heap-read h ad al v*
and $\bigwedge h\ ad\ al\ v\ h'.\ h' \in$ *sc-heap-write-cset h ad al v* $\equiv$ *sc-heap-write h ad al v h'*
for *P*
proof −
show *unfolds*: $\bigwedge h\ ad\ al\ v.\ v \in$ *sc-heap-read-cset h ad al* $\equiv$ *sc-heap-read h ad al v*
$\bigwedge h\ ad\ al\ v\ h'.\ h' \in$ *sc-heap-write-cset h ad al v* $\equiv$ *sc-heap-write h ad al v h'*
by(*simp-all add*: *eval-sc-heap-read-i-i-i-o eval-sc-heap-write-i-i-i-i-o*)
show *JVM-heap-execute-conf-read*
addr2thread-id thread-id2addr
sc-empty (*sc-allocate P*)
sc-typeof-addr sc-heap-read-cset sc-heap-write-cset
(*sc-hconf P*) *P*
apply(*rule JVM-heap-execute-conf-read.intro*)
apply(*unfold unfolds*)
apply(*unfold-locales*)
done
qed

abbreviation *sc-JVM-start-state* :: *addr jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒ (*addr*,*thread-id*,*addr jvm-thread-state*,*heap*,*addr*) *state*
where *sc-JVM-start-state P* ≡ *sc.execute.JVM-start-state TYPE*(*addr jvm-method*) *P P*

abbreviation *sc-exec* :: *addr jvm-prog* ⇒ *thread-id* ⇒ (*addr*, *heap*) *jvm-state′* ⇒ (*addr*, *thread-id*, *heap*) *jvm-ta-state′ set*
where *sc-exec P* ≡ *sc.exec TYPE*(*addr jvm-method*) *P P*

abbreviation *sc-execute-mexec* :: *addr jvm-prog* ⇒ *thread-id* ⇒ (*addr jvm-thread-state* × *heap*)
⇒ (*addr*, *thread-id*, *heap*) *jvm-thread-action* ⇒ (*addr jvm-thread-state* × *heap*) ⇒ *bool*
where *sc-execute-mexec P* ≡ *sc.execute.mexec TYPE*(*addr jvm-method*) *P P*

fun *sc-mexec* ::
addr jvm-prog ⇒ *thread-id* ⇒ (*addr jvm-thread-state′* × *heap*)
⇒ ((*addr*, *thread-id*, *heap*) *jvm-thread-action′* × *addr jvm-thread-state′* × *heap*) *Predicate.pred*
where
sc-mexec P t ((*xcp*, *frs*), *h*) =
sc.exec-1 (*TYPE*(*addr jvm-method*)) *P P t* (*xcp*, *h*, *frs*) ⤜ (λ(*ta*, *xcp*, *h*, *frs*). *Predicate.single* (*ta*, (*xcp*, *frs*), *h*))

abbreviation *sc-jvm-start-state-refine* ::

addr jvm-prog ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒
(*addr*, *thread-id*, *heap*, (*thread-id*, (*addr jvm-thread-state′*) × *addr released-locks*) *rbt*, (*thread-id*, *addr wait-set-status*) *rbt*, *thread-id rs*) *state-refine*
where
sc-jvm-start-state-refine ≡
sc-start-state-refine (*rm-empty* ()) *rm-update* (*rm-empty* ()) (*rs-empty* ()) (λ*C M Ts T* (*mxs*, *mxl0*, *ins*, *xt*) *vs*. (*None*, [((*ins*, *ins*, *xt*), [], *Null* # *vs* @ *replicate mxl0 undefined-value*, *C*, *M*, *0*)]))

fun *jvm-mstate-of-jvm-mstate′* ::
(*addr*,*thread-id*,*addr jvm-thread-state′*,*heap*,*addr*) *state* ⇒ (*addr*,*thread-id*,*addr jvm-thread-state*,*heap*,*addr*) *state*
where
jvm-mstate-of-jvm-mstate′ (*ls*, (*ts*, *m*), *ws*) = (*ls*, (λ*t*. *map-option* (*map-prod jvm-thread-state-of-jvm-thread-state′ id*) (*ts t*), *m*), *ws*)

definition *sc-jvm-state-invar* :: *addr jvm-prog* ⇒ *ty*$_P$ ⇒ (*addr*,*thread-id*,*addr jvm-thread-state′*,*heap*,*addr*) *state set*
where
sc-jvm-state-invar P Φ ≡
{*s*. *jvm-mstate-of-jvm-mstate′ s* ∈ *sc.execute.correct-jvm-state P* Φ} ∩
{*s*. *ts-ok* (λ*t* (*xcp*, *frs*) *h*. *jvm-state′-ok P* (*xcp*, *h*, *frs*)) (*thr s*) (*shr s*)}

fun *JVM-final′* :: *′addr jvm-thread-state′* ⇒ *bool*
where *JVM-final′* (*xcp*, *frs*) ⟷ *frs* = []

lemma *shr-jvm-mstate-of-jvm-mstate′* [*simp*]: *shr* (*jvm-mstate-of-jvm-mstate′ s*) = *shr s*
by(*cases s*) *clarsimp*

lemma *jvm-mstate-of-jvm-mstate′-sc-start-state* [*simp*]:
jvm-mstate-of-jvm-mstate′
(*sc-start-state* (λ*C M Ts T* (*mxs*, *mxl0*, *ins*, *xt*) *vs*. (*None*, [((*ins*, *ins*, *xt*), [], *Null* # *vs* @ *replicate mxl0 undefined-value*, *C*, *M*, *0*)])) *P C M vs*) = *sc-JVM-start-state P C M vs*
by(*simp add*: *sc.start-state-def split-beta fun-eq-iff*)

lemma *sc-jvm-start-state-invar*:
assumes *wf-jvm-prog*$_\Phi$ *P*
and *sc-wf-start-state P C M vs*
shows *sc-state-α* (*sc-jvm-start-state-refine P C M vs*) ∈ *sc-jvm-state-invar P* Φ
unfolding *sc-jvm-state-invar-def Int-iff mem-Collect-eq*
apply(*rule conjI*)
apply(*simp add*: *sc.execute.correct-jvm-state-initial*[*OF assms*])
apply(*rule ts-okI*)
using ‹*sc-wf-start-state P C M vs*›
apply(*auto simp add*: *sc.start-state-def split-beta sc-wf-start-state-iff split*: *if-split-asm dest*: *sees-method-idemp*)
done

lemma *invariant3p-sc-jvm-state-invar*:
assumes *wf-jvm-prog*$_\Phi$ *P*
shows *invariant3p* (*multithreaded-base.redT JVM-final′* (λ*t xm ta x′m′*. *Predicate.eval* (*sc-mexec P t xm*) (*ta*, *x′m′*)) *convert-RA*) (*sc-jvm-state-invar P* Φ)
proof(*rule invariant3pI*)
fix *s tl s′*
assume *red*: *multithreaded-base.redT JVM-final′* (λ*t xm ta x′m′*. *Predicate.eval* (*sc-mexec P t xm*) (*ta*, *x′m′*)) *convert-RA s tl s′*

```
    and invar: s ∈ sc-jvm-state-invar P Φ
  obtain t ta where tl: tl = (t, ta) by(cases tl)
  from red have red': multithreaded-base.redT JVM-final (sc-execute-mexec P) convert-RA (jvm-mstate-of-jvm-mst
s) (t, jvm-thread-action-of-jvm-thread-action' ta) (jvm-mstate-of-jvm-mstate' s')
  proof(cases rule: multithreaded-base.redT.cases[consumes 1, case-names normal acquire])
    case (acquire s t x ln n s')
    thus ?thesis using tl by(cases s)(auto intro!: multithreaded-base.redT.redT-acquire)
  next
    case (normal t x s ta x' m' s')
    obtain xcp frs where x: x = (xcp, frs) by(cases x)
    with invar normal tl
    have correct: sc.execute.correct-state P Φ t (jvm-state-of-jvm-state' (xcp, shr s, frs))
      and ok: jvm-state'-ok P (xcp, shr s, frs)
      apply −
      apply(case-tac [!] s)
      apply(fastforce simp add: sc-jvm-state-invar-def sc.execute.correct-jvm-state-def dest: ts-okD)+
      done
    note eq = sc.exec-correct-state(1)[OF assms this]
    with normal x tl
    have sc-execute-mexec P t (jvm-thread-state-of-jvm-thread-state' x, shr (jvm-mstate-of-jvm-mstate'
s)) (jvm-thread-action-of-jvm-thread-action' ta) (jvm-thread-state-of-jvm-thread-state' x', m')
     by(auto simp add: sc.exec-1-def eq jvm-thread-action'-of-jvm-thread-action-def sc.execute.exec-1-iff)
    with normal tl show ?thesis
     by(cases s)(fastforce intro!: multithreaded-base.redT.redT-normal simp add: final-thread.actions-ok-iff
fun-eq-iff map-redT-updTs elim: rev-iffD1[OF - thread-oks-ts-change] cond-action-oks-final-change)
  qed
  moreover from invar
  have sc.execute.correct-state-ts P Φ (thr (jvm-mstate-of-jvm-mstate' s)) (shr (jvm-mstate-of-jvm-mstate'
s))
    and lock-thread-ok (locks (jvm-mstate-of-jvm-mstate' s)) (thr (jvm-mstate-of-jvm-mstate' s))
    by(simp-all add: sc-jvm-state-invar-def sc.execute.correct-jvm-state-def)
  ultimately have sc.execute.correct-state-ts P Φ (thr (jvm-mstate-of-jvm-mstate' s')) (shr (jvm-mstate-of-jvm-mst
s'))
    and lock-thread-ok (locks (jvm-mstate-of-jvm-mstate' s')) (thr (jvm-mstate-of-jvm-mstate' s'))
   by(blast intro: lifting-wf.redT-preserves[OF sc.execute.lifting-wf-correct-state, OF assms] sc.execute.exec-mthr.re
  hence jvm-mstate-of-jvm-mstate' s' ∈ sc.execute.correct-jvm-state P Φ
    by(simp add: sc.execute.correct-jvm-state-def)
  moreover from red have ts-ok (λt (xcp, frs) h. ∀f∈set frs. frame'-ok P f) (thr s') (shr s') unfolding
tl
  proof(cases rule: multithreaded-base.redT.cases[consumes 1, case-names normal acquire])
    case acquire thus ?thesis using invar
      by(fastforce simp add: sc-jvm-state-invar-def intro!: ts-okI dest: ts-okD bspec split: if-split-asm)
  next
    case (normal t x s ta x' m' s')
    obtain xcp frs where x: x = (xcp, frs) by(cases x)
    with invar normal tl
    have correct: sc.execute.correct-state P Φ t (jvm-state-of-jvm-state' (xcp, shr s, frs))
      and ok: jvm-state'-ok P (xcp, shr s, frs)
      apply −
      apply(case-tac [!] s)
      apply(fastforce simp add: sc-jvm-state-invar-def sc.execute.correct-jvm-state-def dest: ts-okD)+
      done
    from normal x invar show ?thesis
    apply(auto simp add: sc.exec-1-def final-thread.actions-ok-iff jvm-thread-action'-ok-def sc-jvm-state-invar-def)
```

```
    apply hypsubst-thin
    apply(drule sc.exec-correct-state(3)[OF assms correct ok])
    apply(rule ts-okI)
    apply(clarsimp split: if-split-asm simp add: jvm-thread-action'-ok-def)
    apply(drule (1) bspec)
    apply simp
    apply(case-tac thr s t)
     apply(drule (2) redT-updTs-new-thread)
     apply clarsimp
     apply(drule (1) bspec)
     apply simp
     apply(drule (1) bspec)
     apply simp
    apply(erule thin-rl)
    apply(frule (1) redT-updTs-Some)
    apply(fastforce dest: ts-okD)
    done
 qed
 ultimately show s' ∈ sc-jvm-state-invar P Φ by(simp add: sc-jvm-state-invar-def)
qed

lemma sc-exec-deterministic:
 assumes wf-jvm-prog_Φ P
 shows multithreaded-base.deterministic JVM-final' (λt xm ta x'm'. Predicate.eval (sc-mexec P t xm)
(ta, x'm')) convert-RA
    (sc-jvm-state-invar P Φ)
proof −
 from assms sc-deterministic-heap-ops
 have det: multithreaded-base.deterministic JVM-final (sc-execute-mexec P) convert-RA {s. sc.execute.correct-state-ts
P Φ (thr s) (shr s)}
   by(rule sc.execute.mexec-deterministic)(simp add: sc-spurious-wakeups)
 show ?thesis
 proof(rule multithreaded-base.determisticI)
   fix s t x ta' x' m' ta'' x'' m''
   assume inv: s ∈ sc-jvm-state-invar P Φ
     and tst: thr s t = ⌊(x, no-wait-locks)⌋
     and exec1: Predicate.eval (sc-mexec P t (x, shr s)) (ta', x', m')
     and exec2: Predicate.eval (sc-mexec P t (x, shr s)) (ta'', x'', m'')
     and aok1: final-thread.actions-ok JVM-final' s t ta'
     and aok2: final-thread.actions-ok JVM-final' s t ta''
   obtain xcp frs where x: x = (xcp, frs) by(cases x)
   from inv tst x have correct: sc.execute.correct-state P Φ t (jvm-state-of-jvm-state' (xcp, shr s, frs))
     and ok: jvm-state'-ok P (xcp, shr s, frs)
    by(cases s, fastforce simp add: sc-jvm-state-invar-def sc.execute.correct-jvm-state-def dest: ts-okD)+
   note eq = sc.exec-correct-state(1)[OF assms this]

   from exec1 exec2 x
   have sc-execute-mexec P t (jvm-thread-state-of-jvm-thread-state' x, shr (jvm-mstate-of-jvm-mstate'
s)) (jvm-thread-action-of-jvm-thread-action' ta') (jvm-thread-state-of-jvm-thread-state' x', m')
    and sc-execute-mexec P t (jvm-thread-state-of-jvm-thread-state' x, shr (jvm-mstate-of-jvm-mstate'
s)) (jvm-thread-action-of-jvm-thread-action' ta'') (jvm-thread-state-of-jvm-thread-state' x'', m'')
    by(auto simp add: sc.exec-1-def eq jvm-thread-action'-of-jvm-thread-action-def sc.execute.exec-1-iff)
    moreover have thr (jvm-mstate-of-jvm-mstate' s) t = ⌊(jvm-thread-state-of-jvm-thread-state' x,
no-wait-locks)⌋
```

using *tst* **by**(*cases s*) *clarsimp*
moreover have *final-thread.actions-ok JVM-final* (*jvm-mstate-of-jvm-mstate' s*) *t* (*jvm-thread-action-of-jvm-thr*
ta')
and *final-thread.actions-ok JVM-final* (*jvm-mstate-of-jvm-mstate' s*) *t* (*jvm-thread-action-of-jvm-thread-action'*
ta'')
using *aok1 aok2*
by −(*case-tac* [!] *s*,*auto simp add*: *final-thread.actions-ok-iff elim*: *rev-iffD1*[*OF - thread-oks-ts-change*]
cond-action-oks-final-change)
moreover have *sc.execute.correct-state-ts P* Φ (*thr* (*jvm-mstate-of-jvm-mstate' s*)) (*shr* (*jvm-mstate-of-jvm-mst*
s))
using *inv*
by(*cases s*)(*auto intro*!: *ts-okI simp add*: *sc-jvm-state-invar-def sc.execute.correct-jvm-state-def*
dest: *ts-okD*)
ultimately
have *jvm-thread-action-of-jvm-thread-action' ta' = jvm-thread-action-of-jvm-thread-action' ta'' ∧*
jvm-thread-state-of-jvm-thread-state' x' = jvm-thread-state-of-jvm-thread-state' x'' ∧
m' = m''
by−(*drule* (*4*) *multithreaded-base.deterministicD*[*OF det*], *simp-all*)
moreover from *exec1 exec2 x*
have (*ta'*, (*fst x'*, *m'*, *snd x'*)) ∈ *sc-exec P t* (*xcp*, *shr s*, *frs*)
and (*ta''*, (*fst x''*, *m''*, *snd x''*)) ∈ *sc-exec P t* (*xcp*, *shr s*, *frs*)
by(*auto simp add*: *sc.exec-1-def*)
hence *jvm-ta-state'-ok P* (*ta'*, (*fst x'*, *m'*, *snd x'*))
and *jvm-ta-state'-ok P* (*ta''*, (*fst x''*, *m''*, *snd x''*))
by(*blast intro*: *sc.exec-correct-state*[*OF assms correct ok*])+
hence *ta' = jvm-thread-action'-of-jvm-thread-action P* (*jvm-thread-action-of-jvm-thread-action' ta'*)
and *ta'' = jvm-thread-action'-of-jvm-thread-action P* (*jvm-thread-action-of-jvm-thread-action'*
ta'')
and *x' = jvm-thread-state'-of-jvm-thread-state P* (*jvm-thread-state-of-jvm-thread-state' x'*)
and *x'' = jvm-thread-state'-of-jvm-thread-state P* (*jvm-thread-state-of-jvm-thread-state' x''*)
apply −
apply(*case-tac* [!] *ta'*)
apply(*case-tac* [!] *ta''*)
apply(*case-tac* [!] *x'*)
apply(*case-tac* [!] *x''*)
apply(*fastforce simp add*: *jvm-thread-action'-of-jvm-thread-action-def jvm-thread-action'-ok-def*
intro!: *map-idI*[*symmetric*] *convert-new-thread-action-eqI dest*: *bspec*)+
done
ultimately
show *ta' = ta'' ∧ x' = x'' ∧ m' = m''* **by** *simp*
qed(*rule invariant3p-sc-jvm-state-invar*[*OF assms*])
qed

9.8.1 Round-robin scheduler

interpretation *JVM-rr*:
sc-round-robin-base
JVM-final' sc-mexec P convert-RA Jinja-output
for *P*
.

definition *sc-rr-JVM-start-state* :: *nat* ⇒ *'m prog* ⇒ *thread-id fifo round-robin*
where *sc-rr-JVM-start-state n0 P = JVM-rr.round-robin-start n0* (*sc-start-tid P*)

definition *exec-JVM-rr* ::
nat ⇒ *addr jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒
(*thread-id* × (*addr*, *thread-id*) *obs-event list*,
(*addr*, *thread-id*) *locks* × ((*thread-id*, *addr jvm-thread-state′* × *addr released-locks*) *RBT.rbt* × *heap*)
×
(*thread-id*, *addr wait-set-status*) *RBT.rbt* × *thread-id rs*) *tllist*
where
exec-JVM-rr n0 P C M vs = *JVM-rr.exec P n0* (*sc-rr-JVM-start-state n0 P*) (*sc-jvm-start-state-refine P C M vs*)

interpretation *JVM-rr*:
sc-round-robin
JVM-final′ sc-mexec P convert-RA Jinja-output
for *P*
by(*unfold-locales*)

lemma *JVM-rr*:
assumes *wf-jvm-prog*$_\Phi$ *P*
shows
sc-scheduler
JVM-final′ (*sc-mexec P*) *convert-RA*
(*JVM-rr.round-robin P n0*) (*pick-wakeup-via-sel* (λ*s P*. *rm-sel s* (λ(*k*,*v*). *P k v*))) *JVM-rr.round-robin-invar*
(*sc-jvm-state-invar P* Φ)
unfolding *sc-scheduler-def*
apply(*rule JVM-rr.round-robin-scheduler*)
apply(*rule sc-exec-deterministic*[*OF assms*])
done

9.8.2 Random scheduler

interpretation *JVM-rnd*:
sc-random-scheduler-base
JVM-final′ sc-mexec P convert-RA Jinja-output
for *P*
.

definition *sc-rnd-JVM-start-state* :: *Random.seed* ⇒ *random-scheduler*
where *sc-rnd-JVM-start-state seed* = *seed*

definition *exec-JVM-rnd* ::
Random.seed ⇒ *addr jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒
(*thread-id* × (*addr*, *thread-id*) *obs-event list*,
(*addr*, *thread-id*) *locks* × ((*thread-id*, *addr jvm-thread-state′* × *addr released-locks*) *RBT.rbt* × *heap*)
×
(*thread-id*, *addr wait-set-status*) *RBT.rbt* × *thread-id rs*) *tllist*
where *exec-JVM-rnd seed P C M vs* = *JVM-rnd.exec P* (*sc-rnd-JVM-start-state seed*) (*sc-jvm-start-state-refine P C M vs*)

interpretation *JVM-rnd*:
sc-random-scheduler
JVM-final′ sc-mexec P convert-RA Jinja-output
for *P*
by(*unfold-locales*)

lemma *JVM-rnd*:
  assumes *wf-jvm-prog*$_\Phi$ *P*
  shows
  sc-scheduler
    JVM-final' (*sc-mexec P*) *convert-RA*
    (*JVM-rnd.random-scheduler P*) (*pick-wakeup-via-sel* (λ*s P*. *rm-sel s* (λ(*k*,*v*). *P k v*))) (λ- -. *True*)
    (*sc-jvm-state-invar P* Φ)
unfolding *sc-scheduler-def*
apply(*rule JVM-rnd.random-scheduler-scheduler*)
apply(*rule sc-exec-deterministic*[*OF assms*])
done

ML-val ‹@{*code exec-JVM-rr*}›

ML-val ‹@{*code exec-JVM-rnd*}›

end

9.9 Code generator setup

theory *Code-Generation*
imports
  J-Execute
  JVM-Execute2
  ../Compiler/Preprocessor
  ../BV/BCVExec
  ../Compiler/Compiler
  Coinductive.Lazy-TLList
  HOL−Library.Code-Cardinality
  HOL−Library.Code-Target-Int
  HOL−Library.Code-Target-Numeral
begin

Avoid module dependency cycles.

code-identifier
  code-module *More-Set* ⇀ (*SML*) *Set*
| **code-module** *Set* ⇀ (*SML*) *Set*
| **code-module** *Complete-Lattices* ⇀ (*SML*) *Set*
| **code-module** *Complete-Partial-Order* ⇀ (*SML*) *Set*

new code equation for *insort-insert-key* to avoid module dependency cycle with *set*.

lemma *insort-insert-key-code* [*code*]:
  insort-insert-key f x xs =
  (*if List.member* (*map f xs*) (*f x*) *then xs else insort-key f x xs*)
by(*simp add*: *insort-insert-key-def List.member-def split del*: *if-split*)

equations on predicate operations for code inlining

lemma *eq-i-o-conv-single*: *eq-i-o* = *Predicate.single*
by(*rule ext*)(*simp add*: *Predicate.single-bind eq.equation*)

lemma *eq-o-i-conv-single*: *eq-o-i* = *Predicate.single*
by(*rule ext*)(*simp add*: *Predicate.single-bind eq.equation*)

lemma *sup-case-exp-case-exp-same*:
sup-class.sup
(*case-exp cNew cNewArray cCast cInstanceOf cVal cBinOp cVar cLAss cAAcc cAAss cALen cFAcc cFAss cCAS cCall cBlock cSync cInSync cSeq cCond cWhile cThrow cTry e*)
(*case-exp cNew′ cNewArray′ cCast′ cInstanceOf′ cVal′ cBinOp′ cVar′ cLAss′ cAAcc′ cAAss′ cALen′ cFAcc′ cFAss′ cCAS′ cCall′ cBlock′ cSync′ cInSync′ cSeq′ cCond′ cWhile′ cThrow′ cTry′ e*)
=
(*case e of*
new C ⇒ *sup-class.sup* (*cNew C*) (*cNew′ C*)
| *newArray T e* ⇒ *sup-class.sup* (*cNewArray T e*) (*cNewArray′ T e*)
| *Cast T e* ⇒ *sup-class.sup* (*cCast T e*) (*cCast′ T e*)
| *InstanceOf e T* ⇒ *sup-class.sup* (*cInstanceOf e T*) (*cInstanceOf′ e T*)
| *Val v* ⇒ *sup-class.sup* (*cVal v*) (*cVal′ v*)
| *BinOp e bop e′* ⇒ *sup-class.sup* (*cBinOp e bop e′*) (*cBinOp′ e bop e′*)
| *Var V* ⇒ *sup-class.sup* (*cVar V*) (*cVar′ V*)
| *LAss V e* ⇒ *sup-class.sup* (*cLAss V e*) (*cLAss′ V e*)
| *AAcc a e* ⇒ *sup-class.sup* (*cAAcc a e*) (*cAAcc′ a e*)
| *AAss a i e* ⇒ *sup-class.sup* (*cAAss a i e*) (*cAAss′ a i e*)
| *ALen a* ⇒ *sup-class.sup* (*cALen a*) (*cALen′ a*)
| *FAcc e F D* ⇒ *sup-class.sup* (*cFAcc e F D*) (*cFAcc′ e F D*)
| *FAss e F D e′* ⇒ *sup-class.sup* (*cFAss e F D e′*) (*cFAss′ e F D e′*)
| *CompareAndSwap e D F e′ e″* ⇒ *sup-class.sup* (*cCAS e D F e′ e″*) (*cCAS′ e D F e′ e″*)
| *Call e M es* ⇒ *sup-class.sup* (*cCall e M es*) (*cCall′ e M es*)
| *Block V T vo e* ⇒ *sup-class.sup* (*cBlock V T vo e*) (*cBlock′ V T vo e*)
| *Synchronized v e e′* ⇒ *sup-class.sup* (*cSync v e e′*) (*cSync′ v e e′*)
| *InSynchronized v a e* ⇒ *sup-class.sup* (*cInSync v a e*) (*cInSync′ v a e*)
| *Seq e e′* ⇒ *sup-class.sup* (*cSeq e e′*) (*cSeq′ e e′*)
| *Cond b e e′* ⇒ *sup-class.sup* (*cCond b e e′*) (*cCond′ b e e′*)
| *While b e* ⇒ *sup-class.sup* (*cWhile b e*) (*cWhile′ b e*)
| *throw e* ⇒ *sup-class.sup* (*cThrow e*) (*cThrow′ e*)
| *TryCatch e C V e′* ⇒ *sup-class.sup* (*cTry e C V e′*) (*cTry′ e C V e′*))
apply(*cases e*)
apply(*simp-all*)
done

lemma *sup-case-exp-case-exp-other*:
fixes *p* :: *'a* :: *semilattice-sup* **shows**
sup-class.sup
(*case-exp cNew cNewArray cCast cInstanceOf cVal cBinOp cVar cLAss cAAcc cAAss cALen cFAcc cFAss cCAS cCall cBlock cSync cInSync cSeq cCond cWhile cThrow cTry e*)
(*sup-class.sup* (*case-exp cNew′ cNewArray′ cCast′ cInstanceOf′ cVal′ cBinOp′ cVar′ cLAss′ cAAcc′ cAAss′ cALen′ cFAcc′ cFAss′ cCAS′ cCall′ cBlock′ cSync′ cInSync′ cSeq′ cCond′ cWhile′ cThrow′ cTry′ e*) *p*) =
sup-class.sup (*case e of*
new C ⇒ *sup-class.sup* (*cNew C*) (*cNew′ C*)
| *newArray T e* ⇒ *sup-class.sup* (*cNewArray T e*) (*cNewArray′ T e*)
| *Cast T e* ⇒ *sup-class.sup* (*cCast T e*) (*cCast′ T e*)
| *InstanceOf e T* ⇒ *sup-class.sup* (*cInstanceOf e T*) (*cInstanceOf′ e T*)
| *Val v* ⇒ *sup-class.sup* (*cVal v*) (*cVal′ v*)
| *BinOp e bop e′* ⇒ *sup-class.sup* (*cBinOp e bop e′*) (*cBinOp′ e bop e′*)
| *Var V* ⇒ *sup-class.sup* (*cVar V*) (*cVar′ V*)
| *LAss V e* ⇒ *sup-class.sup* (*cLAss V e*) (*cLAss′ V e*)
| *AAcc a e* ⇒ *sup-class.sup* (*cAAcc a e*) (*cAAcc′ a e*)
| *AAss a i e* ⇒ *sup-class.sup* (*cAAss a i e*) (*cAAss′ a i e*)

| *ALen a* ⇒ *sup-class.sup* (*cALen a*) (*cALen′ a*)
| *FAcc e F D* ⇒ *sup-class.sup* (*cFAcc e F D*) (*cFAcc′ e F D*)
| *FAss e F D e′* ⇒ *sup-class.sup* (*cFAss e F D e′*) (*cFAss′ e F D e′*)
| *CompareAndSwap e D F e′ e″* ⇒ *sup-class.sup* (*cCAS e D F e′ e″*) (*cCAS′ e D F e′ e″*)
| *Call e M es* ⇒ *sup-class.sup* (*cCall e M es*) (*cCall′ e M es*)
| *Block V T vo e* ⇒ *sup-class.sup* (*cBlock V T vo e*) (*cBlock′ V T vo e*)
| *Synchronized v e e′* ⇒ *sup-class.sup* (*cSync v e e′*) (*cSync′ v e e′*)
| *InSynchronized v a e* ⇒ *sup-class.sup* (*cInSync v a e*) (*cInSync′ v a e*)
| *Seq e e′* ⇒ *sup-class.sup* (*cSeq e e′*) (*cSeq′ e e′*)
| *Cond b e e′* ⇒ *sup-class.sup* (*cCond b e e′*) (*cCond′ b e e′*)
| *While b e* ⇒ *sup-class.sup* (*cWhile b e*) (*cWhile′ b e*)
| *throw e* ⇒ *sup-class.sup* (*cThrow e*) (*cThrow′ e*)
| *TryCatch e C V e′* ⇒ *sup-class.sup* (*cTry e C V e′*) (*cTry′ e C V e′*)) *p*
apply(*cases e*)
apply(*simp-all add*: *inf-sup-aci sup.assoc*)
done

lemma *sup-bot1*: *sup-class.sup bot a* = (*a* :: *′a* :: {*semilattice-sup*, *order-bot*})
by(*rule sup-absorb2*)*auto*

lemma *sup-bot2*: *sup-class.sup a bot* = (*a* :: *′a* :: {*semilattice-sup*, *order-bot*})
by(*rule sup-absorb1*) *auto*

lemma *sup-case-val-case-val-same*:
sup-class.sup (*case-val cUnit cNull cBool cIntg cAddr v*) (*case-val cUnit′ cNull′ cBool′ cIntg′ cAddr′ v*) =
(*case v of*
Unit ⇒ *sup-class.sup cUnit cUnit′*
| *Null* ⇒ *sup-class.sup cNull cNull′*
| *Bool b* ⇒ *sup-class.sup* (*cBool b*) (*cBool′ b*)
| *Intg i* ⇒ *sup-class.sup* (*cIntg i*) (*cIntg′ i*)
| *Addr a* ⇒ *sup-class.sup* (*cAddr a*) (*cAddr′ a*))
apply(*cases v*)
apply *simp-all*
done

lemma *sup-case-bool-case-bool-same*:
sup-class.sup (*case-bool t f b*) (*case-bool t′ f′ b*) =
(*if b then sup-class.sup t t′ else sup-class.sup f f′*)
by *simp*

lemmas *predicate-code-inline* [*code-unfold*] =
Predicate.single-bind Predicate.bind-single split
eq-i-o-conv-single eq-o-i-conv-single
sup-case-exp-case-exp-same sup-case-exp-case-exp-other unit.case
sup-bot1 sup-bot2 sup-case-val-case-val-same sup-case-bool-case-bool-same

lemma *op-case-ty-case-ty-same*:
f (*case-ty cVoid cBoolean cInteger cNT cClass cArray e*)
(*case-ty cVoid′ cBoolean′ cInteger′ cNT′ cClass′ cArray′ e*) =
(*case e of*
Void ⇒ *f cVoid cVoid′*
| *Boolean* ⇒ *f cBoolean cBoolean′*
| *Integer* ⇒ *f cInteger cInteger′*

| *NT* ⇒ *f cNT cNT′*
| *Class C* ⇒ *f* (*cClass C*) (*cClass′ C*)
| *Array T* ⇒ *f* (*cArray T*) (*cArray′ T*))
by(*simp split*: *ty.split*)

declare *op-case-ty-case-ty-same*[**where** *f*=*sup-class.sup*, *code-unfold*]

lemma *op-case-bop-case-bop-same*:
f (*case-bop cEq cNotEq cLessThan cLessOrEqual cGreaterThan cGreaterOrEqual cAdd cSubtract cMult cDiv cMod cBinAnd cBinOr cBinXor cShiftLeft cShiftRightZeros cShiftRightSigned bop*)
(*case-bop cEq′ cNotEq′ cLessThan′ cLessOrEqual′ cGreaterThan′ cGreaterOrEqual′ cAdd′ cSubtract′ cMult′ cDiv′ cMod′ cBinAnd′ cBinOr′ cBinXor′ cShiftLeft′ cShiftRightZeros′ cShiftRightSigned′ bop*)
= *case-bop* (*f cEq cEq′*) (*f cNotEq cNotEq′*) (*f cLessThan cLessThan′*) (*f cLessOrEqual cLessOrEqual′*) (*f cGreaterThan cGreaterThan′*) (*f cGreaterOrEqual cGreaterOrEqual′*) (*f cAdd cAdd′*) (*f cSubtract cSubtract′*) (*f cMult cMult′*) (*f cDiv cDiv′*) (*f cMod cMod′*) (*f cBinAnd cBinAnd′*) (*f cBinOr cBinOr′*) (*f cBinXor cBinXor′*) (*f cShiftLeft cShiftLeft′*) (*f cShiftRightZeros cShiftRightZeros′*) (*f cShiftRightSigned cShiftRightSigned′*) *bop*
by(*simp split*: *bop.split*)

lemma *sup-case-bop-case-bop-other* [*code-unfold*]:
fixes *p* :: *'a* :: *semilattice-sup* **shows**
sup-class.sup (*case-bop cEq cNotEq cLessThan cLessOrEqual cGreaterThan cGreaterOrEqual cAdd cSubtract cMult cDiv cMod cBinAnd cBinOr cBinXor cShiftLeft cShiftRightZeros cShiftRightSigned bop*)
(*sup-class.sup* (*case-bop cEq′ cNotEq′ cLessThan′ cLessOrEqual′ cGreaterThan′ cGreaterOrEqual′ cAdd′ cSubtract′ cMult′ cDiv′ cMod′ cBinAnd′ cBinOr′ cBinXor′ cShiftLeft′ cShiftRightZeros′ cShiftRightSigned′ bop*) *p*)
= *sup-class.sup* (*case-bop* (*sup-class.sup cEq cEq′*) (*sup-class.sup cNotEq cNotEq′*) (*sup-class.sup cLessThan cLessThan′*) (*sup-class.sup cLessOrEqual cLessOrEqual′*) (*sup-class.sup cGreaterThan cGreaterThan′*) (*sup-class.sup cGreaterOrEqual cGreaterOrEqual′*) (*sup-class.sup cAdd cAdd′*) (*sup-class.sup cSubtract cSubtract′*) (*sup-class.sup cMult cMult′*) (*sup-class.sup cDiv cDiv′*) (*sup-class.sup cMod cMod′*) (*sup-class.sup cBinAnd cBinAnd′*) (*sup-class.sup cBinOr cBinOr′*) (*sup-class.sup cBinXor cBinXor′*) (*sup-class.sup cShiftLeft cShiftLeft′*) (*sup-class.sup cShiftRightZeros cShiftRightZeros′*) (*sup-class.sup cShiftRightSigned cShiftRightSigned′*) *bop*) *p*
apply(*cases bop*)
apply(*simp-all add*: *inf-sup-aci sup.assoc*)
done

declare *op-case-bop-case-bop-same*[**where** *f*=*sup-class.sup*, *code-unfold*]

end

theory *JVMExec-Execute*
imports
../JVM/JVMExec
ExternalCall-Execute
begin

9.9.1 Manual translation of the JVM to use sets instead of predicates

locale *JVM-heap-execute* = *heap-execute* +
constrains *addr2thread-id* :: (*'addr* :: *addr*) ⇒ *'thread-id*

and *thread-id2addr* :: *'thread-id ⇒ 'addr*
and *spurious-wakeups* :: *bool*
and *empty-heap* :: *'heap*
and *allocate* :: *'heap ⇒ htype ⇒ ('heap × 'addr) set*
and *typeof-addr* :: *'heap ⇒ 'addr ⇒ htype option*
and *heap-read* :: *'heap ⇒ 'addr ⇒ addr-loc ⇒ 'addr val set*
and *heap-write* :: *'heap ⇒ 'addr ⇒ addr-loc ⇒ 'addr val ⇒ 'heap set*

sublocale *JVM-heap-execute* < *execute*: *JVM-heap-base*
addr2thread-id thread-id2addr
spurious-wakeups
empty-heap allocate typeof-addr
λh a ad v. v ∈ heap-read h a ad λh a ad v h'. h' ∈ heap-write h a ad v
.

context *JVM-heap-execute* **begin**

definition *exec-instr* ::
'addr instr ⇒ 'addr jvm-prog ⇒ 'thread-id ⇒ 'heap ⇒ 'addr val list ⇒ 'addr val list
⇒ cname ⇒ mname ⇒ pc ⇒ 'addr frame list
⇒ (('addr, 'thread-id, 'heap) jvm-thread-action × ('addr, 'heap) jvm-state) set
where [*simp*]: *exec-instr = execute.exec-instr*

lemma *exec-instr-code* [*code*]:
exec-instr (Load n) P t h stk loc C₀ M₀ pc frs =
{(ε, (None, h, ((loc ! n) # stk, loc, C₀, M₀, pc+1)#frs))}
exec-instr (Store n) P t h stk loc C₀ M₀ pc frs =
{(ε, (None, h, (tl stk, loc[n:=hd stk], C₀, M₀, pc+1)#frs))}
exec-instr (Push v) P t h stk loc C₀ M₀ pc frs =
{(ε, (None, h, (v # stk, loc, C₀, M₀, pc+1)#frs))}
exec-instr (New C) P t h stk loc C₀ M₀ pc frs =
(let HA = allocate h (Class-type C) in
if HA = {} then {(ε, ⌊execute.addr-of-sys-xcpt OutOfMemory⌋, h, (stk, loc, C₀, M₀, pc) # frs)}
else do { (h', a) ← HA; {(⦃NewHeapElem a (Class-type C)⦄, None, h', (Addr a # stk, loc, C₀, M₀, pc + 1)#frs)} })
exec-instr (NewArray T) P t h stk loc C0 M0 pc frs =
(let si = the-Intg (hd stk);
i = nat (sint si)
in if si <s 0
then {(ε, ⌊execute.addr-of-sys-xcpt NegativeArraySize⌋, h, (stk, loc, C0, M0, pc) # frs)}
else let HA = allocate h (Array-type T i) in
if HA = {} then {(ε, ⌊execute.addr-of-sys-xcpt OutOfMemory⌋, h, (stk, loc, C0, M0, pc) # frs)}
else do { (h', a) ← HA; {(⦃NewHeapElem a (Array-type T i)⦄, None, h', (Addr a # tl stk, loc, C0, M0, pc + 1) # frs)}})
exec-instr ALoad P t h stk loc C0 M0 pc frs =
(let va = hd (tl stk)
in (if va = Null then {(ε, ⌊execute.addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C0, M0, pc) # frs)}
else
let i = the-Intg (hd stk);
a = the-Addr va;
len = alen-of-htype (the (typeof-addr h a))
in if i <s 0 ∨ int len ≤ sint i then
{(ε, ⌊execute.addr-of-sys-xcpt ArrayIndexOutOfBounds⌋, h, (stk, loc, C0, M0, pc) # frs)}

```
          else do {
              v ← heap-read h a (ACell (nat (sint i)));
               {(⦃ReadMem a (ACell (nat (sint i))) v⦄, None, h, (v # tl (tl stk), loc, C0, M0, pc +
1) # frs)}
            }))
  exec-instr AStore P t h stk loc C0 M0 pc frs =
  (let ve = hd stk;
       vi = hd (tl stk);
       va = hd (tl (tl stk))
   in (if va = Null then {(ε, ⌊execute.addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C0, M0, pc) # frs)}
       else (let i = the-Intg vi;
                 idx = nat (sint i);
                 a = the-Addr va;
                 hT = the (typeof-addr h a);
                 T = ty-of-htype hT;
                 len = alen-of-htype hT;
                 U = the (execute.typeof-h h ve)
             in (if i <s 0 ∨ int len ≤ sint i then
                    {(ε, ⌊execute.addr-of-sys-xcpt ArrayIndexOutOfBounds⌋, h, (stk, loc, C0, M0, pc) #
frs)}
                 else if P ⊢ U ≤ the-Array T then
                     do {
                       h' ← heap-write h a (ACell idx) ve;
                       {(⦃WriteMem a (ACell idx) ve⦄, None, h', (tl (tl (tl stk)), loc, C0, M0, pc+1) #
frs)}
                     }
                 else {(ε, (⌊execute.addr-of-sys-xcpt ArrayStore⌋, h, (stk, loc, C0, M0, pc) # frs))})))))
  exec-instr ALength P t h stk loc C0 M0 pc frs =
   {(ε, (let va = hd stk
         in if va = Null
            then (⌊execute.addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C0, M0, pc) # frs)
           else (None, h, (Intg (word-of-int (int (alen-of-htype (the (typeof-addr h (the-Addr va)))))) #
tl stk, loc, C0, M0, pc+1) # frs)))}
  exec-instr (Getfield F C) P t h stk loc C0 M0 pc frs =
   (let v = hd stk
    in if v = Null then {(ε, ⌊execute.addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C0, M0, pc) # frs)}
       else let a = the-Addr v
            in do {
              v' ← heap-read h a (CField C F);
              {(⦃ReadMem a (CField C F) v'⦄, None, h, (v' # (tl stk), loc, C0, M0, pc + 1) # frs)}
            })
  exec-instr (Putfield F C) P t h stk loc C0 M0 pc frs =
  (let v = hd stk;
       r = hd (tl stk)
   in if r = Null then {(ε, ⌊execute.addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C0, M0, pc) # frs)}
      else let a = the-Addr r
           in do {
               h' ← heap-write h a (CField C F) v;
               {(⦃WriteMem a (CField C F) v⦄, None, h', (tl (tl stk), loc, C0, M0, pc + 1) # frs)}
             })
 exec-instr (Checkcast T) P t h stk loc C0 M0 pc frs =
 {(ε, let U = the (typeof_h (hd stk))
      in if P ⊢ U ≤ T then (None, h, (stk, loc, C0, M0, pc + 1) # frs)
         else (⌊execute.addr-of-sys-xcpt ClassCast⌋, h, (stk, loc, C0, M0, pc) # frs))}
```

```
exec-instr (Instanceof T) P t h stk loc C0 M0 pc frs =
 {(ε, None, h, (Bool (hd stk ≠ Null ∧ P ⊢ the (typeof_h (hd stk)) ≤ T) # tl stk, loc, C0, M0, pc +
1) # frs)}
exec-instr (Invoke M n) P t h stk loc C0 M0 pc frs =
 (let r = stk ! n
  in (if r = Null then {(ε, ⌊execute.addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C0, M0, pc) # frs)}
      else (let ps = rev (take n stk);
                a = the-Addr r;
                T = the (typeof-addr h a);
                (D,M',Ts,meth)= method P (class-type-of T) M
       in case meth of
            Native ⇒
                 do {
                   (ta, va, h') ← red-external-aggr P t a M ps h;
                   {(extTA2JVM P ta, extRet2JVM n h' stk loc C0 M0 pc frs va)}
                 }
          | ⌊(mxs,mxl0,ins,xt)⌋ ⇒
            let f' = ([],[r]@ps@(replicate mxl0 undefined-value),D,M,0)
            in {(ε, None, h, f' # (stk, loc, C0, M0, pc) # frs)})))
exec-instr Return P t h stk0 loc0 C0 M0 pc frs =
 {(ε, (if frs=[] then (None, h, [])
       else
         let v = hd stk0;
             (stk,loc,C,m,pc) = hd frs;
              n = length (fst (snd (method P C0 M0)))
         in (None, h, (v#(drop (n+1) stk),loc,C,m,pc+1)#tl frs)))}
exec-instr Pop P t h stk loc C0 M0 pc frs = {(ε, (None, h, (tl stk, loc, C0, M0, pc+1)#frs))}
exec-instr Dup P t h stk loc C0 M0 pc frs = {(ε, (None, h, (hd stk # stk, loc, C0, M0, pc+1)#frs))}
exec-instr Swap P t h stk loc C0 M0 pc frs = {(ε, (None, h, (hd (tl stk) # hd stk # tl (tl stk), loc,
C0, M0, pc+1)#frs))}
exec-instr (BinOpInstr bop) P t h stk loc C0 M0 pc frs =
 {(ε,
   case the (execute.binop bop (hd (tl stk)) (hd stk)) of
     Inl v ⇒ (None, h, (v # tl (tl stk), loc, C0, M0, pc + 1) # frs)
   | Inr a ⇒ (Some a, h, (stk, loc, C0, M0, pc) # frs))}
exec-instr (IfFalse i) P t h stk loc C0 M0 pc frs =
 {(ε, (let pc' = if hd stk = Bool False then nat(int pc+i) else pc+1
       in (None, h, (tl stk, loc, C0, M0, pc')#frs)))}
exec-instr (Goto i) P t h stk loc C0 M0 pc frs = {(ε, (None, h, (stk, loc, C0, M0, nat(int pc+i))#frs))}
exec-instr ThrowExc P t h stk loc C0 M0 pc frs =
 {(ε, (let xp' = if hd stk = Null then ⌊execute.addr-of-sys-xcpt NullPointer⌋ else ⌊the-Addr(hd stk)⌋
       in (xp', h, (stk, loc, C0, M0, pc)#frs)))}
exec-instr MEnter P t h stk loc C0 M0 pc frs =
 {(let v = hd stk
   in if v = Null
      then (ε, ⌊execute.addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C0, M0, pc) # frs)
       else ({|Lock→the-Addr v, SyncLock (the-Addr v)|}, None, h, (tl stk, loc, C0, M0, pc + 1) #
frs))}
exec-instr MExit P t h stk loc C0 M0 pc frs =
 (let v = hd stk
  in if v = Null
     then {(ε, ⌊execute.addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C0, M0, pc) # frs)}
      else {({|Unlock→the-Addr v, SyncUnlock (the-Addr v)|}, None, h, (tl stk, loc, C0, M0, pc + 1)
# frs),
```

($\{\!|$*UnlockFail*→*the-Addr v*$|\!\}$, $\lfloor$*execute.addr-of-sys-xcpt IllegalMonitorState*$\rfloor$, *h*, (*stk*, *loc*, C_0, M_0, *pc*) # *frs*)})
by(*auto 4 4 intro*: *rev-bexI*)

definition *exec* :: *'addr jvm-prog* ⇒ *'thread-id* ⇒ (*'addr*, *'heap*) *jvm-state* ⇒ (*'addr*, *'thread-id*, *'heap*) *jvm-ta-state set*
where *exec* = *execute.exec*

lemma *exec-code*:
exec P t (*xcp*, *h*, []) = {}
exec P t (*None*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs*) = *exec-instr* (*instrs-of P C M* ! *pc*) *P t h stk loc C M pc frs*
exec P t ($\lfloor a \rfloor$, *h*, *fr* # *frs*) = {(ε, *execute.exception-step P a h fr frs*)}
by(*simp-all add*: *exec-def*)

definition *exec-1* ::
'addr jvm-prog ⇒ *'thread-id* ⇒ (*'addr*, *'heap*) *jvm-state*
⇒ ((*'addr*, *'thread-id*, *'heap*) *jvm-thread-action* × (*'addr*, *'heap*) *jvm-state*) *Predicate.pred*
where *exec-1 P t σ* = *pred-of-set* (*exec P t σ*)

lemma *exec-1I*: *execute.exec-1 P t σ ta σ'* ⟹ *Predicate.eval* (*exec-1 P t σ*) (*ta*, *σ'*)
by(*erule execute.exec-1.cases*)(*simp add*: *exec-1-def exec-def*)

lemma *exec-1E*:
assumes *Predicate.eval* (*exec-1 P t σ*) (*ta*, *σ'*)
obtains *execute.exec-1 P t σ ta σ'*
using *assms*
by(*auto simp add*: *exec-1-def exec-def intro*: *execute.exec-1.intros*)

lemma *exec-1-eq* [*simp*]:
Predicate.eval (*exec-1 P t σ*) (*ta*, *σ'*) ⟷ *execute.exec-1 P t σ ta σ'*
by(*auto intro*: *exec-1I elim*: *exec-1E*)

lemma *exec-1-eq'*:
Predicate.eval (*exec-1 P t σ*) = (λ(*ta*, *σ'*). *execute.exec-1 P t σ ta σ'*)
by(*rule ext*)(*simp split*: *prod.split*)

end

lemmas [*code*] =
JVM-heap-execute.exec-instr-code
JVM-heap-base.exception-step.simps
JVM-heap-execute.exec-code
JVM-heap-execute.exec-1-def

end

theory *JVM-Execute*
imports
SC-Schedulers
JVMExec-Execute
../BV/BVProgressThreaded
begin

abbreviation *sc-heap-read-cset* :: *heap* ⇒ *addr* ⇒ *addr-loc* ⇒ *addr val set*
where *sc-heap-read-cset h ad al* ≡ *set-of-pred* (*sc-heap-read-i-i-i-o h ad al*)

abbreviation *sc-heap-write-cset* :: *heap* ⇒ *addr* ⇒ *addr-loc* ⇒ *addr val* ⇒ *heap set*
where *sc-heap-write-cset h ad al v* ≡ *set-of-pred* (*sc-heap-write-i-i-i-i-o h ad al v*)

interpretation *sc*:
JVM-heap-execute
addr2thread-id
thread-id2addr
sc-spurious-wakeups
sc-empty
sc-allocate P
sc-typeof-addr
sc-heap-read-cset
sc-heap-write-cset
rewrites ⋀*h ad al v*. *v* ∈ *sc-heap-read-cset h ad al* ≡ *sc-heap-read h ad al v*
and ⋀*h ad al v h′*. *h′* ∈ *sc-heap-write-cset h ad al v* ≡ *sc-heap-write h ad al v h′*
for *P*
apply(*simp-all add*: *eval-sc-heap-read-i-i-i-o eval-sc-heap-write-i-i-i-i-o*)
done

interpretation *sc-execute*:
JVM-conf-read
addr2thread-id
thread-id2addr
sc-spurious-wakeups
sc-empty
sc-allocate P
sc-typeof-addr
sc-heap-read
sc-heap-write
sc-hconf P
for *P*
by(*unfold-locales*)

fun *sc-mexec* ::
addr jvm-prog ⇒ *thread-id* ⇒ (*addr jvm-thread-state* × *heap*)
⇒ ((*addr*, *thread-id*, *heap*) *jvm-thread-action* × *addr jvm-thread-state* × *heap*) *Predicate.pred*
where
sc-mexec P t ((*xcp*, *frs*), *h*) =
sc.exec-1 (*TYPE*(*addr jvm-method*)) *P P t* (*xcp*, *h*, *frs*) ⤜ (λ(*ta*, *xcp*, *h*, *frs*). *Predicate.single* (*ta*, (*xcp*, *frs*), *h*))

abbreviation *sc-jvm-start-state-refine* ::
addr jvm-prog ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒
(*addr*, *thread-id*, *heap*, (*thread-id*, (*addr jvm-thread-state*) × *addr released-locks*) *rm*, (*thread-id*, *addr wait-set-status*) *rm*, *thread-id rs*) *state-refine*
where
sc-jvm-start-state-refine ≡
sc-start-state-refine (*rm-empty* ()) *rm-update* (*rm-empty* ()) (*rs-empty* ()) (λ*C M Ts T* (*mxs*, *mxl0*, *b*) *vs*. (*None*, [([], *Null* # *vs* @ *replicate mxl0 undefined-value*, *C*, *M*, *0*)]))

abbreviation *sc-jvm-state-invar* :: *addr jvm-prog* ⇒ ty_P ⇒ (*addr*,*thread-id*,*addr jvm-thread-state*,*heap*,*addr*) *state set*
where *sc-jvm-state-invar P* Φ ≡ {*s*. *sc-execute.correct-state-ts P* Φ (*thr s*) (*shr s*)}

lemma *eval-sc-mexec*:
(λ*t xm ta x*′*m*′. *Predicate.eval* (*sc-mexec P t xm*) (*ta*, *x*′*m*′)) =
(λ*t* ((*xcp*, *frs*), *h*) *ta* ((*xcp*′, *frs*′), *h*′). *sc.execute.exec-1* (*TYPE*(*addr jvm-method*)) *P P t* (*xcp*, *h*, *frs*) *ta* (*xcp*′, *h*′, *frs*′))
by(*rule ext*)+(*fastforce intro*!: *SUP1-I simp add*: *sc.exec-1-eq*′)

lemma *sc-jvm-start-state-invar*:
assumes *wf-jvm-prog*$_\Phi$ *P*
and *sc-wf-start-state P C M vs*
shows *sc-state-α* (*sc-jvm-start-state-refine P C M vs*) ∈ *sc-jvm-state-invar P* Φ
using *sc-execute.correct-jvm-state-initial*[*OF assms*]
by(*simp add*: *sc-execute.correct-jvm-state-def*)

9.9.2 Round-robin scheduler

interpretation *JVM-rr*:
sc-round-robin-base
JVM-final sc-mexec P convert-RA Jinja-output
for *P*
.

definition *sc-rr-JVM-start-state* :: *nat* ⇒ *′m prog* ⇒ *thread-id fifo round-robin*
where *sc-rr-JVM-start-state n0 P* = *JVM-rr.round-robin-start n0* (*sc-start-tid P*)

definition *exec-JVM-rr* ::
nat ⇒ *addr jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒
(*thread-id* × (*addr*, *thread-id*) *obs-event list*,
(*addr*, *thread-id*) *locks* × ((*thread-id*, *addr jvm-thread-state* × *addr released-locks*) *rm* × *heap*) ×
(*thread-id*, *addr wait-set-status*) *rm* × *thread-id rs*) *tllist*
where
exec-JVM-rr n0 P C M vs = *JVM-rr.exec P n0* (*sc-rr-JVM-start-state n0 P*) (*sc-jvm-start-state-refine P C M vs*)

interpretation *JVM-rr*:
sc-round-robin
JVM-final sc-mexec P convert-RA Jinja-output
for *P*
by(*unfold-locales*)

lemma *JVM-rr*:
assumes *wf-jvm-prog*$_\Phi$ *P*
shows
sc-scheduler
JVM-final (*sc-mexec P*) *convert-RA*
(*JVM-rr.round-robin P n0*) (*pick-wakeup-via-sel* (λ*s P*. *rm-sel s* (λ(*k*,*v*). *P k v*))) *JVM-rr.round-robin-invar*
(*sc-jvm-state-invar P* Φ)
unfolding *sc-scheduler-def*
apply(*rule JVM-rr.round-robin-scheduler*)
apply(*unfold eval-sc-mexec*)
apply(*rule sc-execute.mexec-deterministic*[*OF assms sc-deterministic-heap-ops*])

apply(*simp add*: *sc-spurious-wakeups*)
done

9.9.3 Random scheduler

interpretation *JVM-rnd*:
sc-random-scheduler-base
JVM-final sc-mexec P convert-RA Jinja-output
for *P*
.

definition *sc-rnd-JVM-start-state* :: *Random.seed* ⇒ *random-scheduler*
where *sc-rnd-JVM-start-state seed* = *seed*

definition *exec-JVM-rnd* ::
Random.seed ⇒ *addr jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒
(*thread-id* × (*addr*, *thread-id*) *obs-event list*,
(*addr*, *thread-id*) *locks* × ((*thread-id*, *addr jvm-thread-state* × *addr released-locks*) *rm* × *heap*) ×
(*thread-id*, *addr wait-set-status*) *rm* × *thread-id rs*) *tllist*
where *exec-JVM-rnd seed P C M vs* = *JVM-rnd.exec P* (*sc-rnd-JVM-start-state seed*) (*sc-jvm-start-state-refine P C M vs*)

interpretation *JVM-rnd*:
sc-random-scheduler
JVM-final sc-mexec P convert-RA Jinja-output
for *P*
by(*unfold-locales*)

lemma *JVM-rnd*:
assumes *wf-jvm-prog*$_\Phi$ *P*
shows
sc-scheduler
JVM-final (*sc-mexec P*) *convert-RA*
(*JVM-rnd.random-scheduler P*) (*pick-wakeup-via-sel* (λ*s P*. *rm-sel s* (λ(*k*,*v*). *P k v*))) (λ- -. *True*)
(*sc-jvm-state-invar P* Φ)
unfolding *sc-scheduler-def*
apply(*rule JVM-rnd.random-scheduler-scheduler*)
apply(*unfold eval-sc-mexec*)
apply(*rule sc-execute.mexec-deterministic*[*OF assms sc-deterministic-heap-ops*])
apply(*simp add*: *sc-spurious-wakeups*)
done

ML-val ‹@{*code exec-JVM-rr*}›

ML-val ‹@{*code exec-JVM-rnd*}›

end

9.10 String representation of types

theory *ToString* **imports**
../J/Expr
../JVM/JVMInstructions

```
  ../Basic/JT-ICF
begin

class toString =
  fixes toString :: 'a ⇒ String.literal

instantiation bool :: toString begin
definition [code]: toString b = (case b of True ⇒ STR ''True'' | False ⇒ STR ''False'')
instance proof qed
end

instantiation char :: toString begin
definition [code]: toString (c :: char) = String.implode [c]
instance proof qed
end

instantiation String.literal :: toString begin
definition [code]: toString (s :: String.literal) = s
instance proof qed
end

fun list-toString :: String.literal ⇒ 'a :: toString list ⇒ String.literal list
where
  list-toString sep [] = []
| list-toString sep [x] = [toString x]
| list-toString sep (x#xs) = toString x # sep # list-toString sep xs

instantiation list :: (toString) toString begin
definition [code]:
  toString (xs :: 'a list) = sum-list (STR ''['' # list-toString (STR '','') xs @ [STR '']''])
instance proof qed
end

definition digit-toString :: int ⇒ String.literal
where
  digit-toString k = (if k = 0 then STR ''0''
    else if k = 1 then STR ''1''
    else if k = 2 then STR ''2''
    else if k = 3 then STR ''3''
    else if k = 4 then STR ''4''
    else if k = 5 then STR ''5''
    else if k = 6 then STR ''6''
    else if k = 7 then STR ''7''
    else if k = 8 then STR ''8''
    else if k = 9 then STR ''9''
    else undefined)

function int-toString :: int ⇒ String.literal list
where
  int-toString n =
  (if n < 0 then STR ''−'' # int-toString (− n)
   else if n < 10 then [digit-toString n ]
   else int-toString (n div 10) @ [digit-toString (n mod 10)])
by pat-completeness simp
```

termination by *size-change*

instantiation *int* :: *toString* **begin**
definition [*code*]: *toString i* = *sum-list* (*int-toString i*)
instance proof qed
end

instantiation *nat* :: *toString* **begin**
definition [*code*]: *toString n* = *toString* (*int n*)
instance proof qed
end

instantiation *option* :: (*toString*) *toString* **begin**
primrec *toString-option* :: *'a option* ⇒ *String.literal* **where**
  toString None = *STR ''None''*
| *toString* (*Some a*) = *sum-list* [*STR ''Some ('', toString a, STR '')''*]
instance proof qed
end

instantiation *finfun* :: ({*toString*, *card-UNIV*, *equal*, *linorder*}, *toString*) *toString* **begin**
definition [*code*]:
  toString (*f* :: *'a* ⇒*f 'b*) =
   sum-list
    (*STR ''(''*
    # *toString* (*finfun-default f*)
    # *concat* (*map* (λ*x*. [*STR '',''*, *toString x*, *STR ''|->''*, *toString* (*f* $ *x*)]) (*finfun-to-list f*))
    @ [*STR '')''*])
instance proof qed
end

instantiation *word* :: (*len*) *toString* **begin**
definition [*code*]: *toString* (*w* :: *'a word*) = *toString* (*sint w*)
instance proof qed
end

instantiation *fun* :: (*type*, *type*) *toString* **begin**
definition [*code*]: *toString* (*f* :: *'a* ⇒ *'b*) = *STR ''fn''*
instance proof qed
end

instantiation *val* :: (*toString*) *toString* **begin**
fun *toString-val* :: (*'a* :: *toString*) *val* ⇒ *String.literal*
where
  toString Unit = *STR ''Unit''*
| *toString Null* = *STR ''Null''*
| *toString* (*Bool b*) = *sum-list* [*STR ''Bool ''*, *toString b*]
| *toString* (*Intg i*) = *sum-list* [*STR ''Intg ''*, *toString i*]
| *toString* (*Addr a*) = *sum-list* [*STR ''Addr ''*, *toString a*]
instance proof qed
end

instantiation *ty* :: *toString* **begin**
primrec *toString-ty* :: *ty* ⇒ *String.literal*
where

```
  toString Void = STR ''Void''
| toString Boolean = STR ''Boolean''
| toString Integer = STR ''Integer''
| toString NT = STR ''NT''
| toString (Class C) = sum-list [STR ''Class '', toString C]
| toString (T⌊⌋) = sum-list [toString T, STR ''[]'']
instance proof qed
end

instantiation bop :: toString begin
primrec toString-bop :: bop ⇒ String.literal where
  toString Eq = STR ''==''
| toString NotEq = STR ''!=''
| toString LessThan = STR ''<''
| toString LessOrEqual = STR ''<=''
| toString GreaterThan = STR ''>''
| toString GreaterOrEqual = STR ''>=''
| toString Add = STR ''+''
| toString Subtract = STR ''−''
| toString Mult = STR ''∗''
| toString Div = STR ''/''
| toString Mod = STR ''%''
| toString BinAnd = STR ''&''
| toString BinOr = STR ''|''
| toString BinXor = STR ''^''
| toString ShiftLeft = STR ''<<''
| toString ShiftRightZeros = STR ''>>''
| toString ShiftRightSigned = STR ''>>>''
instance proof qed
end

instantiation addr-loc :: toString begin
primrec toString-addr-loc :: addr-loc ⇒ String.literal where
  toString (CField C F) = sum-list [STR ''CField '', F, STR ''{'', C, STR ''}'']
| toString (ACell n) = sum-list [STR ''ACell '', toString n]
instance proof qed
end

instantiation htype :: toString begin
fun toString-htype :: htype ⇒ String.literal where
  toString (Class-type C) = C
| toString (Array-type T n) = sum-list [toString T, STR ''['', toString n, STR '']'']
instance proof qed
end

instantiation obs-event :: (toString, toString) toString begin
primrec toString-obs-event :: ('a :: toString, 'b :: toString) obs-event ⇒ String.literal
where
  toString (ExternalCall ad M vs v) =
   sum-list [STR ''ExternalCall '', M, STR ''('', toString vs, STR '') = '', toString v]
| toString (ReadMem ad al v) =
   sum-list [STR ''ReadMem '', toString ad, STR ''@'', toString al, STR ''='', toString v]
| toString (WriteMem ad al v) =
   sum-list [STR ''WriteMem '', toString ad, STR ''@'', toString al, STR ''='', toString v]
```

```
| toString (NewHeapElem ad hT) = sum-list [STR ''Allocate '', toString ad, STR '':'', toString hT]
| toString (ThreadStart t) = sum-list [STR ''ThreadStart '', toString t]
| toString (ThreadJoin t) = sum-list [STR ''ThreadJoin '', toString t]
| toString (SyncLock ad) = sum-list [STR ''SyncLock '', toString ad]
| toString (SyncUnlock ad) = sum-list [STR ''SyncUnlock '', toString ad]
| toString (ObsInterrupt t) = sum-list [STR ''Interrupt '', toString t]
| toString (ObsInterrupted t) = sum-list [STR ''Interrupted '', toString t]
instance proof qed
end

instantiation prod :: (toString, toString) toString begin
definition toString = (λ(a, b). sum-list [STR ''('', toString a, STR '', '', toString b, STR '')''])
instance proof qed
end

instantiation fmod-ext :: (toString) toString begin
definition toString fd = sum-list [STR ''{|volatile='', toString (volatile fd), STR '', '', toString
(fmod.more fd), STR ''|}'']
instance proof qed
end

instantiation unit :: toString begin
definition toString (u :: unit) = STR ''()''
instance proof qed
end

instantiation exp :: (toString, toString, toString) toString begin
fun toString-exp :: ('a :: toString, 'b :: toString, 'c :: toString) exp ⇒ String.literal
where
  toString (new C) = sum-list [STR ''new '', C]
| toString (newArray T e) = sum-list [STR ''new '', toString T, STR ''['', toString e, STR '']'']
| toString (Cast T e) = sum-list [STR ''('', toString T, STR '') ('', toString e, STR '')'']
| toString (InstanceOf e T) = sum-list [STR ''('', toString e, STR '') instanceof '', toString T]
| toString (Val v) = sum-list [STR ''Val ('', toString v, STR '')'']
| toString (e1 «bop» e2) = sum-list [STR ''('', toString e1, STR '') '', toString bop, STR '' ('', toString
e2, STR '')'']
| toString (Var V) = sum-list [STR ''Var '', toString V]
| toString (V := e) = sum-list [toString V, STR '' := ('', toString e, STR '')'']
| toString (AAcc a i) = sum-list [STR ''('', toString a, STR '')['', toString i, STR '']'']
| toString (AAss a i e) = sum-list [STR ''('', toString a, STR '')['', toString i, STR ''] := ('', toString
e, STR '')'']
| toString (ALen a) = sum-list [STR ''('', toString a, STR ').length'']
| toString (FAcc e F D) = sum-list [STR ''('', toString e, STR ').'', F, STR ''{'', D, STR ''}'']
| toString (FAss e F D e') = sum-list [STR ''('', toString e, STR ').'', F, STR ''{'', D, STR ''} :=
('', toString e', STR '')'']
| toString (Call e M es) = sum-list ([STR ''('', toString e, STR ').'', M, STR ''('']  @ map toString es
@ [STR '')'']])
| toString (Block V T vo e) = sum-list ([STR ''{'', toString V, STR '':'', toString T] @ (case vo of
None ⇒ [] | Some v ⇒ [STR ''='', toString v]) @ [STR ''; '', toString e, STR ''}'']])
| toString (Synchronized V e e') = sum-list [STR ''synchronized-'', toString V, STR ''-('', toString e,
STR '') {'', toString e', STR ''}'']
| toString (InSynchronized V ad e) = sum-list [STR ''insynchronized-'', toString V, STR ''-('', toString
ad, STR '') {'', toString e, STR ''}'']
| toString (e;;e') = sum-list [toString e, STR ''; '', toString e']
```

| *toString* (*if* (*e*) *e′ else e″*) = *sum-list* [*STR ″if (″*, *toString e*, *STR ″*) { *″*, *toString e′*, *STR ″* } *else* { *″*, *toString e″*, *STR ″*}*″*]
| *toString* (*while* (*e*) *e′*) = *sum-list* [*STR ″while (″*, *toString e*, *STR ″*) { *″*, *toString e′*, *STR ″* }*″*]
| *toString* (*throw e*) = *sum-list* [*STR ″throw (″*, *toString e*, *STR ″*)*″*]
| *toString* (*try e catch*(*C V*) *e′*) = *sum-list* [*STR ″try* { *″*, *toString e*, *STR ″* } *catch (″*, *C*, *STR ″ ″*, *toString V*, *STR ″*) { *″*, *toString e′*, *STR ″* }*″*]
instance proof qed
end

instantiation *instr* :: (*toString*) *toString* **begin**
primrec *toString-instr* :: *′a instr* ⇒ *String.literal* **where**
  toString (*Load i*) = *sum-list* [*STR ″Load (″*, *toString i*, *STR ″*)*″*]
| *toString* (*Store i*) = *sum-list* [*STR ″Store (″*, *toString i*, *STR ″*)*″*]
| *toString* (*Push v*) = *sum-list* [*STR ″Push (″*, *toString v*, *STR ″*)*″*]
| *toString* (*New C*) = *sum-list* [*STR ″New ″*, *toString C*]
| *toString* (*NewArray T*) = *sum-list* [*STR ″NewArray ″*, *toString T*]
| *toString ALoad* = *STR ″ALoad″*
| *toString AStore* = *STR ″AStore″*
| *toString ALength* = *STR ″ALength″*
| *toString* (*Getfield F D*) = *sum-list* [*STR ″Getfield ″*, *toString F*, *STR ″ ″*, *toString D*]
| *toString* (*Putfield F D*) = *sum-list* [*STR ″Putfield ″*, *toString F*, *STR ″ ″*, *toString D*]
| *toString* (*Checkcast T*) = *sum-list* [*STR ″Checkcast ″*, *toString T*]
| *toString* (*Instanceof T*) = *sum-list* [*STR ″Instanceof ″*, *toString T*]
| *toString* (*Invoke M n*) = *sum-list* [*STR ″Invoke ″*, *toString M*, *STR ″ ″*, *toString n*]
| *toString Return* = *STR ″Return″*
| *toString Pop* = *STR ″Pop″*
| *toString Dup* = *STR ″Dup″*
| *toString Swap* = *STR ″Swap″*
| *toString* (*BinOpInstr bop*) = *sum-list* [*STR ″BinOpInstr ″*, *toString bop*]
| *toString* (*Goto i*) = *sum-list* [*STR ″Goto ″*, *toString i*]
| *toString* (*IfFalse i*) = *sum-list* [*STR ″IfFalse ″*, *toString i*]
| *toString ThrowExc* = *STR ″ThrowExc″*
| *toString MEnter* = *STR ″monitorenter″*
| *toString MExit* = *STR ″monitorexit″*
instance proof qed
end

instantiation *trie* :: (*toString*, *toString*) *toString* **begin**
definition [*code*]: *toString* (*t* :: (*′a*, *′b*) *trie*) = *toString* (*tm-to-list t*)
instance proof qed
end

instantiation *rbt* :: ({*toString*,*linorder*}, *toString*) *toString* **begin**
definition [*code*]:
  toString (*t* :: (*′a*, *′b*) *rbt*) =
   sum-list (*list-toString* (*STR ″*,↩*″*) (*rm-to-list t*))
instance proof qed
end

instantiation *assoc-list* :: (*toString*, *toString*) *toString* **begin**
definition [*code*]: *toString* = *toString* ∘ *Assoc-List.impl-of*
instance proof qed
end

code-printing
  class-instance *String.literal* :: *toString* ⇀ (*Haskell*) −

end

9.11 Setup for converter Java2Jinja

theory *Java2Jinja*
imports
  Code-Generation
  ToString
begin

code-identifier
  code-module *Java2Jinja* ⇀ (*SML*) *Code-Generation*

definition *j-Program* :: *addr J-mb cdecl list* ⇒ *addr J-prog*
where *j-Program* = *Program*

export-code *wf-J-prog′ j-Program* **in** *SML* **file** ‹*JWellForm.ML*›

Functions for extracting calls to the native print method

definition *purge* **where**
  ⋀*run.*
  purge run =
  lmap (λ*obs. case obs of ExternalCall - -* (*Cons* (*Intg i*) -) *v* ⇒ *i*)
  (*lfilter*
    (λ*obs. case obs of ExternalCall - M* (*Cons* (*Intg i*) *Nil*) - ⇒ *M* = *print* | - ⇒ *False*)
    (*lconcat* (*lmap* (*llist-of* ∘ *snd*) (*llist-of-tllist run*))))

Various other functions

instantiation *heapobj* :: *toString* **begin**
primrec *toString-heapobj* :: *heapobj* ⇒ *String.literal* **where**
  toString (*Obj C fs*) = *sum-list* [*STR ″*(*Obj ″*, *toString C*, *STR ″*, *″*, *toString fs*, *STR ″*)*″*]
| *toString* (*Arr T si fs el*) =
   sum-list [*STR ″*([*″*, *toString si*, *STR ″*]*″*, *toString T*, *STR ″*, *″*, *toString fs*, *STR ″*, *″*, *toString* (*map snd* (*rm-to-list el*)), *STR ″*)*″*]
instance proof qed
end

definition *case-llist′* **where** *case-llist′* = *case-llist*
definition *case-tllist′* **where** *case-tllist′* = *case-tllist*
definition *terminal′* **where** *terminal′* = *terminal*
definition *llist-of-tllist′* **where** *llist-of-tllist′* = *llist-of-tllist*
definition *thr′* **where** *thr′* = *thr*
definition *shr′* **where** *shr′* = *shr*

definition *heap-toString* :: *heap* ⇒ *String.literal*
where *heap-toString* = *toString*

definition *thread-toString* :: (*thread-id*, (*addr expr* × *addr locals*) × (*addr* ⇒*f nat*)) *rbt* ⇒ *String.literal*
where *thread-toString* = *toString*

definition *thread-toString′* :: (*thread-id*, *addr jvm-thread-state′* × (*addr* ⇒*f nat*)) *rbt* ⇒ *String.literal*

where *thread-toString′ = toString*

definition *trace-toString* :: *thread-id* × (*addr*, *thread-id*) *obs-event list* ⇒ *String.literal*
where *trace-toString = toString*

code-identifier
  code-module *Cardinality* ⇀ (*SML*) *Set*
| **code-module** *Code-Cardinality* ⇀ (*SML*) *Set*
| **code-module** *Conditionally-Complete-Lattices* ⇀ (*SML*) *Set*
| **code-module** *List* ⇀ (*SML*) *Set*
| **code-module** *Predicate* ⇀ (*SML*) *Set*
| **code-module** *Parity* ⇀ (*SML*) *Bit-Operations*
| **type-class** *semiring-parity* ⇀ (*SML*) *Bit-Operations.semiring-parity*
| **class-instance** *int* :: *semiring-parity* ⇀ (*SML*) *Bit-Operations.semiring-parity-int*
| **class-instance** *int* :: *ring-parity* ⇀ (*SML*) *Bit-Operations.semiring-parity-int*
| **constant** *member-i-i* ⇀ (*SML*) *Set.member-i-i*

export-code
  wf-J-prog′ exec-J-rr exec-J-rnd
  j-Program
  purge case-llist′ case-tllist′ terminal′ llist-of-tllist′
  thr′ shr′ heap-toString thread-toString trace-toString
  in *SML*
  file ‹*J-Execute.ML*›

definition *j2jvm* :: *addr J-prog* ⇒ *addr jvm-prog* **where** *j2jvm = J2JVM*

export-code
  wf-jvm-prog′ exec-JVM-rr exec-JVM-rnd j2jvm
  j-Program
  purge case-llist′ case-tllist′ terminal′ llist-of-tllist′
  thr′ shr′ heap-toString thread-toString′ trace-toString
  in *SML*
  file ‹*JVM-Execute2.ML*›

end
theory *Execute-Main*
imports
  SC-Schedulers
  PCompilerRefine
  Code-Generation
  JVM-Execute
  Java2Jinja
begin

end

1662

Chapter 10

Examples

10.1 Apprentice challenge

theory *ApprenticeChallenge*
imports
../Execute/Code-Generation
begin

This theory implements the apprentice challenge by Porter and Moore [5].

definition *ThreadC* :: *addr J-mb cdecl*
where
ThreadC =
(*Thread*, *Object*, [],
[(*run*, [], *Void*, ⌊([], *unit*)⌋),
(*start*, [], *Void*, *Native*),
(*join*, [], *Void*, *Native*),
(*interrupt*, [], *Void*, *Native*),
(*isInterrupted*, [], *Boolean*, *Native*)])

definition *Container* :: *cname*
where *Container* = *STR ''Container''*

definition *ContainerC* :: *addr J-mb cdecl*
where *ContainerC* = (*Container*, *Object*, [(*STR ''counter''*, *Integer*, (|*volatile*=*False*|))], [])

definition *String* :: *cname*
where *String* = *STR ''String''*

definition *StringC* :: *addr J-mb cdecl*
where
StringC = (*String*, *Object*, [], [])

definition *Job* :: *cname*
where *Job* = *STR ''Job''*

definition *JobC* :: *addr J-mb cdecl*
where
JobC =
(*Job*, *Thread*, [(*STR ''objref''*, *Class Container*, (|*volatile*=*False*|))],
[(*STR ''incr''*, [], *Class Job*, ⌊([],

```
    sync(Var (STR ''objref''))
     ((Var (STR ''objref''))·STR ''counter''{STR ''''} := ((Var (STR ''objref''))·STR ''counter''{STR
''''} «Add» Val (Intg 1)));;
    Var this)⌋),
   (STR ''setref'', [Class Container], Void, ⌊([STR ''o''],
    LAss (STR ''objref'') (Var (STR ''o'')))⌋),
   (run, [], Void, ⌊([],
    while (true) (Var this·STR ''incr''([])))⌋)
   ])

definition Apprentice :: cname
where Apprentice = STR ''Apprentice''

definition ApprenticeC :: addr J-mb cdecl
where
  ApprenticeC =
  (Apprentice, Object, [],
   [(STR ''main'', [Class String⌊⌉], Void, ⌊([STR ''args''],
    {STR ''container'':Class Container=None;
      (STR ''container'' := new Container);;
      (while (true)
        {STR ''job'':Class Job=None;
          (STR ''job'' := new Job);;
          (Var (STR ''job'')·STR ''setref''([Var (STR ''container'')]));;
          (Var (STR ''job'')·Type.start([]))
        }
      )
    })⌋)])

definition ApprenticeChallenge
where
  ApprenticeChallenge = Program (SystemClasses @ [StringC, ThreadC, ContainerC, JobC, Appren-
ticeC])

definition ApprenticeChallenge-annotated
where ApprenticeChallenge-annotated = annotate-prog-code ApprenticeChallenge

lemma wf-J-prog ApprenticeChallenge-annotated
by eval

lemmas [code-unfold] =
  Container-def Job-def String-def Apprentice-def

definition main :: String.literal where main = STR ''main''

ML-val ‹
  val - = tracing started;
  val program = @{code ApprenticeChallenge-annotated};
  val - = tracing prg;
  val compiled = @{code J2JVM} program;
  val - = tracing compiled;

  @{code exec-J-rr}
    @{code 1 :: nat}
```

```
    program
    @{code Apprentice}
    @{code main}
    [ @{code Null}];

  val - = tracing J-rr;
  @{code exec-JVM-rr}
    @{code 1 :: nat}
    compiled
    @{code Apprentice}
    @{code main}
    [ @{code Null}];
  val - = tracing JVM-rr;
›

end
```

10.2 Buffer example

```
theory BufferExample imports
  ../Execute/Code-Generation
begin

definition ThreadC :: addr J-mb cdecl
where
  ThreadC =
  (Thread, Object, [],
    [(run, [], Void, ⌊([], unit)⌋),
     (start, [], Void, Native),
     (join, [], Void, Native),
     (interrupt, [], Void, Native),
     (isInterrupted, [], Boolean, Native)])

definition IntegerC :: addr J-mb cdecl
where IntegerC = (STR ''Integer'', Object, [(STR ''value'', Integer, {|volatile=False|})], [])

definition Buffer :: cname
where Buffer = STR ''Buffer''

definition BufferC :: addr J-mb cdecl
where
  BufferC =
   (Buffer, Object,
    [(STR ''buffer'', Class Object⌊⌉, {|volatile=False|}),
     (STR ''front'', Integer, {|volatile=False|}),
     (STR ''back'', Integer, {|volatile=False|}),
     (STR ''size'', Integer, {|volatile=False|})],
    [(STR ''constructor'', [Integer], Void, ⌊([STR ''size''],
      (STR ''buffer'' := newA (Class Object)⌊Var (STR ''size'')⌉);;
      (STR ''front'' := Val (Intg 0));;
      (STR ''back'' := Val (Intg (− 1)));;
      (Var this•(STR ''size''){STR ''''} := Val (Intg 0)))⌋),
     (STR ''empty'', [], Boolean, ⌊([], sync(Var this) (Var (STR ''size'') «Eq» Val (Intg 0)))⌋),
```

```
    (STR ''full'', [], Boolean, ⌊([],
     sync(Var this) (Var (STR ''size'') «Eq» ((Var (STR ''buffer''))·length)))⌋),
    (STR ''get'', [], Class Object, ⌊([],
     sync(Var this) (
       (while (Var this·(STR ''empty'')([]))
         (try (Var this·wait([])) catch(InterruptedException (STR ''e'')) unit));;
       (STR ''size'' := (Var (STR ''size'') «Subtract» Val (Intg 1)));;
       {(STR ''result''):Class Object=None;
         ((STR ''result'') := ((Var (STR ''buffer''))⌊Var (STR ''front'')⌉));;
         (STR ''front'' := (Var (STR ''front'') «Add» Val (Intg 1)));;
         (if ((Var (STR ''front'')) «Eq» ((Var (STR ''buffer''))·length))
            (STR ''front'' := Val (Intg 0))
          else unit);;
         (Var this·notifyAll([]));;
         Var (STR ''result'')
       }
     ))⌋),
    (STR ''put'', [Class Object], Void, ⌊([STR ''o''],
     sync(Var this) (
       (while (Var this·STR ''full''([]))
         (try (Var this·wait([])) catch(InterruptedException STR ''e'') unit));;
       (STR ''back'' := (Var (STR ''back'') «Add» Val (Intg 1)));;
       (if (Var (STR ''back'') «Eq» ((Var (STR ''buffer''))·length))
          (STR ''back'' := Val (Intg 0))
        else unit);;
       (AAss (Var (STR ''buffer'')) (Var (STR ''back'')) (Var (STR ''o'')));;
       (STR ''size'' := ((Var (STR ''size'')) «Add» Val (Intg 1)));;
       (Var this·notifyAll([]))
     ))⌋])
  ])

definition Producer :: cname
where Producer = STR ''Producer''

definition ProducerC :: int ⇒ addr J-mb cdecl
where
  ProducerC n =
   (Producer, Thread, [(STR ''buffer'', Class Buffer, (|volatile=False|))],
    [(run, [], Void, ⌊([],
     {STR ''i'':Integer=⌊Intg 0⌋;
        while (Var (STR ''i'') «NotEq» Val (Intg (word-of-int n))) (
        (Var (STR ''buffer''))·STR ''put''([{STR ''temp'':Class (STR ''Integer'')=None; (STR ''temp''
:= new (STR ''Integer'');; ((FAss (Var (STR ''temp'')) (STR ''value'') (STR '''') (Var (STR ''i'')));;
Var (STR ''temp''))} ]);;
          STR ''i'' := (Var (STR ''i'') «Add» (Val (Intg 1))))
     })⌋)])

definition Consumer :: cname
where Consumer = STR ''Consumer''

definition ConsumerC :: int ⇒ addr J-mb cdecl
where
  ConsumerC n =
  (Consumer, Thread, [(STR ''buffer'', Class Buffer, (|volatile=False|))],
```

```
  [(run, [], Void, ⌊([],
   {STR ''i'':Integer=⌊Intg 0⌋;
     while (Var (STR ''i'') «NotEq» Val (Intg (word-of-int n))) (
       {STR ''o'':Class Object=None;
         Seq (STR ''o'' := ((Var (STR ''buffer''))•STR ''get''([])))
             (STR ''i'' := (Var (STR ''i'') «Add» Val (Intg 1)))})
   })⌋)])

definition String :: cname
where String = STR ''String''

definition StringC :: addr J-mb cdecl
where
  StringC = (String, Object, [], [])

definition Test :: cname
where Test = STR ''Test''

definition TestC :: addr J-mb cdecl
where
  TestC =
  (Test, Object, [],
   [(STR ''main'', [Class String⌊⌉], Void, ⌊([STR ''args''],
    {STR ''b'':Class Buffer=None; (STR ''b'' := new Buffer);;
     (Var (STR ''b'')•STR ''constructor''([Val (Intg 10)]));;
     {STR ''p'':Class Producer=None; STR ''p'' := new Producer;;
       {STR ''c'':Class Consumer=None;
          (STR ''c'' := new Consumer);;
          (Var (STR ''c'')•STR ''buffer''{STR ''''} := Var (STR ''b''));;
          (Var (STR ''p'')•STR ''buffer''{STR ''''} := Var (STR ''b''));;
          (Var (STR ''c'')•Type.start([]));;(Var (STR ''p'')•Type.start([]))
       }
     }
   })⌋)])

definition BufferExample
where
  BufferExample n = Program (SystemClasses @ [ThreadC, StringC, IntegerC, BufferC, ProducerC
n, ConsumerC n, TestC])

definition BufferExample-annotated
where
  BufferExample-annotated n = annotate-prog-code (BufferExample n)

lemmas [code-unfold] =
  IntegerC-def Buffer-def Producer-def Consumer-def Test-def
  String-def

lemma wf-J-prog (BufferExample-annotated 10)
by eval

definition main where main = STR ''main''
definition five :: int where five = 5
```

ML-val ‹
val program = @{*code BufferExample-annotated*} @{*code five*};
val compiled = @{*code J2JVM*} *program*;

val run =
@{*code exec-J-rr*}
@{*code 1* :: *nat*}
program
@{*code Test*}
@{*code main*}
[@{*code Null*}];
val - = @{*code terminal*} *run*;

val jvm-run =
@{*code exec-JVM-rr*}
@{*code 1* :: *nat*}
compiled
@{*code Test*}
@{*code main*}
[@{*code Null*}];
val - = @{*code terminal*} *run*;
›

end
theory *Examples-Main*
imports
ApprenticeChallenge
BufferExample
begin
end
theory *JinjaThreads*
imports
Basic/Basic-Main
Common/Common-Main
J/J-Main
JVM/JVM-Main
BV/BV-Main
Compiler/Compiler-Main
MM/MM-Main
Execute/Execute-Main
Examples/Examples-Main
begin

end

Bibliography

[1] Andreas Lochbihler. Type safe nondeterminism - a formal semantics of Java threads. In *Proceedings of the 2008 International Workshop on Foundations of Object-Oriented Languages*, 2008.

[2] Andreas Lochbihler. Verifying a compiler for Java threads. In Andrew D. Gordon, editor, *Programming Languages and Systems (ESOP 2010)*, volume 6012 of *Lecture Notes in Computer Science*, pages 427–447. Springer, 2010.

[3] Andreas Lochbihler. Java and the Java memory model – a unified, machine-checked formalisation. In Helmut Seidl, editor, *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*, pages 493–513. Springer, 2012.

[4] Andreas Lochbihler and Lukas Bulwahn. Animating the formalised semantics of a Java-like language. In Marko van Eekelen, Herman Geuvers, Julien Schmalz, and Freek Wiedijk, editors, *Interactive Theorem Proving (ITP 2011)*, volume 6898 of *Lecture Notes in Computer Science*, pages 216–232. Springer, 2011.

[5] J Strother Moore and George Porter. The apprentice challenge. *ACM Trans. Program. Lang. Syst.*, 24(3):193–216, May 2002.