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

1 The generic multithreaded semantics **7**
1.1 State of the multithreaded semantics . . . 7
1.2 All about a managing a single lock . . . 15
1.3 Semantics of the thread actions for locking . . . 20
1.4 Semantics of the thread actions for thread creation . . . 24
1.5 Semantics of the thread actions for wait, notify and interrupt . . . 27
1.6 Semantics of the thread actions for purely conditional purpose such as Join . . . 30
1.7 Wellformedness conditions for the multithreaded state . . . 32
1.8 Semantics of the thread action ReleaseAcquire for the thread state . . . 35
1.9 Semantics of the thread actions for interruption . . . 35
1.10 The multithreaded semantics . . . 38
1.11 Auxiliary definitions for the progress theorem for the multithreaded semantics . 45
1.12 Deadlock formalisation . . . 48
1.13 Progress theorem for the multithreaded semantics . . . 54
1.14 Lifting of thread-local properties to the multithreaded case . . . 56
1.15 Labelled transition systems . . . 59
1.16 The multithreaded semantics as a labelled transition system . . . 72
1.17 Various notions of bisimulation . . . 76
1.18 Bisimulation relations for the multithreaded semantics . . . 88
1.19 Preservation of deadlock across bisimulations . . . 106
1.20 Semantic properties of lifted predicates . . . 109
1.21 Synthetic first and last actions for each thread . . . 111

2 Data Flow Analysis Framework **123**
2.1 Semilattices . . . 123
2.2 The Error Type . . . 127
2.3 More about Options . . . 129
2.4 Products as Semilattices . . . 130
2.5 Fixed Length Lists . . . 131
2.6 Typing and Dataflow Analysis Framework . . . 133
2.7 More on Semilattices . . . 133
2.8 Lifting the Typing Framework to err, app, and eff . . . 135
2.9 Kildall's Algorithm . . . 136
2.10 The Lightweight Bytecode Verifier . . . 140
2.11 Correctness of the LBV . . . 144
2.12 Completeness of the LBV . . . 145

3 Concepts for all JinjaThreads Languages **147**
3.1 JinjaThreads types 147
3.2 Class Declarations and Programs 151
3.3 Relations between Jinja Types 153
3.4 Jinja Values 162
3.5 Exceptions 164
3.6 System Classes 167
3.7 An abstract heap model 168
3.8 Observable events in JinjaThreads 174
3.9 The initial configuration 175
3.10 Conformance Relations for Type Soundness Proofs 179
3.11 Semantics of method calls that cannot be defined inside JinjaThreads 184
3.12 Generic Well-formedness of programs 195
3.13 Properties of external calls in well-formed programs 202
3.14 Conformance for threads 205
3.15 Binary Operators 206
3.16 The Jinja Type System as a Semilattice 217

4 JinjaThreads source language **223**
4.1 Program State 223
4.2 Expressions 223
4.3 Abstract heap locales for source code programs 232
4.4 Small Step Semantics 234
4.5 Weak well-formedness of Jinja programs 245
4.6 Well-typedness of Jinja expressions 246
4.7 Definite assignment 251
4.8 Well-formedness Constraints 254
4.9 The source language as an instance of the framework 255
4.10 Runtime Well-typedness 263
4.11 Progress of Small Step Semantics 267
4.12 Preservation of definite assignment 268
4.13 Type Safety Proof 269
4.14 Progress and type safety theorem for the multithreaded system 270
4.15 Preservation of Deadlock 274
4.16 Program annotation 275

5 Jinja Virtual Machine **281**
5.1 State of the JVM 281
5.2 Instructions of the JVM 282
5.3 Abstract heap locales for byte code programs 284
5.4 JVM Instruction Semantics 286
5.5 Exception handling in the JVM 290
5.6 Program Execution in the JVM 291
5.7 A Defensive JVM 292
5.8 Instantiating the framework semantics with the JVM 297

6 Bytecode verifier **301**
6.1 The JVM Type System as Semilattice . . . 301
6.2 Effect of Instructions on the State Type . . . 305
6.3 The Bytecode Verifier . . . 315
6.4 BV Type Safety Invariant . . . 316
6.5 BV Type Safety Proof . . . 320
6.6 Welltyped Programs produce no Type Errors . . . 327
6.7 Progress result for both of the multithreaded JVMs . . . 327
6.8 Preservation of deadlock for the JVMs . . . 333
6.9 Monotonicity of eff and app . . . 334
6.10 The Typing Framework for the JVM . . . 335
6.11 LBV for the JVM . . . 337
6.12 Kildall for the JVM . . . 339
6.13 Code generation for the byte code verifier . . . 340

7 Compilation **343**
7.1 Method calls in expressions . . . 343
7.2 The JinjaThreads source language with explicit call stacks . . . 346
7.3 Bisimulation proof for between source code small step semantics with and without callstacks for single threads . . . 362
7.4 The intermediate language J1 . . . 366
7.5 Abstract heap locales for J1 programs . . . 371
7.6 Semantics of the intermediate language . . . 373
7.7 Deadlock perservation for the intermediate language . . . 396
7.8 Program Compilation . . . 397
7.9 Compilation Stage 2 . . . 402
7.10 Various Operations for Exception Tables . . . 406
7.11 Type rules for the intermediate language . . . 411
7.12 Well-Formedness of Intermediate Language . . . 414
7.13 Preservation of Well-Typedness in Stage 2 . . . 415
7.14 Unobservable steps for the JVM . . . 428
7.15 JVM Semantics for the delay bisimulation proof from intermediate language to byte code . . . 437
7.16 The delay bisimulation between intermediate language and JVM . . . 471
7.17 Correctness of Stage: From intermediate language to JVM . . . 501
7.18 Correctness of Stage 2: From JVM to intermediate language . . . 503
7.19 Correctness of Stage 2: The multithreaded setting . . . 506
7.20 Indexing variables in variable lists . . . 510
7.21 Compilation Stage 1 . . . 511
7.22 The bisimulation relation betwenn source and intermediate language . . . 514
7.23 Unlocking a sync block never fails . . . 523
7.24 Semantic Correctness of Stage 1 . . . 530
7.25 Preservation of well-formedness from source code to intermediate language . . . 544
7.26 Correctness of both stages . . . 545

8 Memory Models **551**
8.1 Sequential consistency . . . 552
8.2 Sequential consistency with efficient data structures . . . 559
8.3 Orders as predicates . . . 567
8.4 Axiomatic specification of the JMM . . . 574
8.5 The data race free guarantee of the JMM . . . 600
8.6 Sequentially consistent executions are legal . . . 600
8.7 Non-speculative prefixes of executions . . . 602
8.8 Sequentially consistent completion of executions in the JMM . . . 605
8.9 Happens-before consistent completion of executions in the JMM . . . 613
8.10 Locales for heap operations with set of allocated addresses . . . 617
8.11 Combination of locales for heap operations and interleaving . . . 620
8.12 Type-safety proof for the Java memory model . . . 636
8.13 JMM Instantiation with Jinja – common parts . . . 641
8.14 JMM Instantiation for J . . . 651
8.15 JMM Instantiation for bytecode . . . 661
8.16 JMM heap implementation 1 . . . 669
8.17 Compiler correctness for the JMM . . . 674
8.18 JMM heap implementation 2 . . . 678
8.19 Specialize type safety for JMM heap implementation 2 . . . 686
8.20 JMM type safety for source code . . . 690
8.21 JMM type safety for bytecode . . . 694
8.22 Compiler correctness for JMM heap implementation 2 . . . 699

9 Schedulers **701**
9.1 Refinement for multithreaded states . . . 701
9.2 Abstract scheduler . . . 702
9.3 Random scheduler . . . 720
9.4 Round robin scheduler . . . 723
9.5 Tabulation for lookup functions . . . 734
9.6 Executable semantics for J . . . 743
9.7 Executable semantics for the JVM . . . 745
9.8 An optimized JVM . . . 748
9.9 Code generator setup . . . 761
9.10 String representation of types . . . 771
9.11 Setup for converter Java2Jinja . . . 776

10 Examples **779**
10.1 Apprentice challenge . . . 779
10.2 Buffer example . . . 780

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*

⟨*ML*⟩
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*
⟨*proof*⟩

lemma *thr-a-conv* [*simp*]: *thr-a* (*ls*, *nts*, *jswss*) = *nts*
⟨*proof*⟩

lemma *cond-a-conv* [*simp*]: *cond-a* (*ls*, *nts*, *js*, *wws*) = *js*
⟨*proof*⟩

lemma *wset-a-conv* [*simp*]: *wset-a* (*ls*, *nts*, *js*, *wss*, *isobs*) = *wss*
⟨*proof*⟩

lemma *interrupt-a-conv* [*simp*]: *interrupt-a* (*ls*, *nts*, *js*, *ws*, *is*, *obs*) = *is*
⟨*proof*⟩

lemma *obs-a-conv* [*simp*]: *obs-a* (*ls*, *nts*, *js*, *wss*, *is*, *obs*) = *obs*
⟨*proof*⟩

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*

⟨*ML*⟩

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*:
$⦃ta\text{-}update\text{-}obs\ ta\ obs⦄_l = ⦃ta⦄_l$ $⦃ta\text{-}update\text{-}obs\ ta\ obs⦄_t = ⦃ta⦄_t$ $⦃ta\text{-}update\text{-}obs\ ta\ obs⦄_w = ⦃ta⦄_w$
$⦃ta\text{-}update\text{-}obs\ ta\ obs⦄_c = ⦃ta⦄_c$ $⦃ta\text{-}update\text{-}obs\ ta\ obs⦄_i = ⦃ta⦄_i$ $⦃ta\text{-}update\text{-}obs\ ta\ obs⦄_o = ⦃ta⦄_o\ @\ [obs]$
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*:
$\{\!|ta\text{-}update\text{-}interrupt\ ta\ i|\!\}_l = \{\!|ta|\!\}_l$ $\{\!|ta\text{-}update\text{-}interrupt\ ta\ i|\!\}_t = \{\!|ta|\!\}_t$ $\{\!|ta\text{-}update\text{-}interrupt\ ta\ i|\!\}_c = \{\!|ta|\!\}_c$
$\{\!|ta\text{-}update\text{-}interrupt\ ta\ i|\!\}_w = \{\!|ta|\!\}_w$ $\{\!|ta\text{-}update\text{-}interrupt\ ta\ i|\!\}_i = \{\!|ta|\!\}_i\ @\ [i]$ $\{\!|ta\text{-}update\text{-}interrupt\ ta\ i|\!\}_o = \{\!|ta|\!\}_o$
⟨*proof*⟩

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*:
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (LockAction\ (la,\ l))\ ta|\!\}_o = \{\!|ta\text{-}update\text{-}locks\ ta\ la\ l|\!\}_o$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (NewThreadAction\ nt)\ ta|\!\}_o = \{\!|ta\text{-}update\text{-}NewThread\ ta\ nt|\!\}_o$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (ConditionalAction\ ca)\ ta|\!\}_o = \{\!|ta\text{-}update\text{-}Conditional\ ta\ ca|\!\}_o$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (WaitSetAction\ wa)\ ta|\!\}_o = \{\!|ta\text{-}update\text{-}wait\text{-}set\ ta\ wa|\!\}_o$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (InterruptAction\ ia)\ ta|\!\}_o = \{\!|ta\text{-}update\text{-}interrupt\ ta\ ia|\!\}_o$
$\{\!|thread\text{-}action'\text{-}to\text{-}thread\text{-}action\ (ObsAction\ ob)\ ta|\!\}_o = \{\!|ta\text{-}update\text{-}obs\ ta\ ob|\!\}_o$
⟨*proof*⟩

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*))

⟨*ML*⟩
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*)
⟨*proof*⟩

lemma *neq-no-wait-locksE*:
fixes *ln* **assumes** *ln* ≠ *no-wait-locks* **obtains** *l* **where** *ln* $ *l* > *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *thr-conv* [*simp*]: *thr* (*ls*, (*ts*, *m*), *ws*) = *ts*
⟨*proof*⟩

lemma *shr-conv* [*simp*]: *shr* (*ls*, (*ts*, *m*), *ws*, *is*) = *m*
⟨*proof*⟩

lemma *wset-conv* [*simp*]: *wset* (*ls*, (*ts*, *m*), *ws*, *is*) = *ws*
⟨*proof*⟩

lemma *interrupts-conv* [*simp*]: *interrupts* (*ls*, (*ts*, *m*), *ws*, *is*) = *is*
⟨*proof*⟩

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*
⟨*proof*⟩

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*'
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *inj-convert-new-thread-action* [*simp*]:
inj (*convert-new-thread-action f*) = *inj f*
⟨*proof*⟩

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 $(\lambda x.\ x) = (id :: ('t, 'x, 'm)\ new\text{-}thread\text{-}action \Rightarrow ('t, 'x, 'm)\ new\text{-}thread\text{-}action)$
(**is** *?thesis2*)
⟨*proof*⟩

definition *convert-extTA* :: $('x \Rightarrow 'x') \Rightarrow ('l,'t,'x,'m,'w,'o)\ thread\text{-}action \Rightarrow ('l,'t,'x','m,'w,'o)\ thread\text{-}action$
where *convert-extTA f ta* = $(\{\!|ta|\!\}_l,\ map\ (convert\text{-}new\text{-}thread\text{-}action\ f)\ \{\!|ta|\!\}_t,\ snd\ (snd\ ta))$

lemma *convert-extTA-simps* [*simp*]:
$convert\text{-}extTA\ f\ \varepsilon = \varepsilon$
$\{\!|convert\text{-}extTA\ f\ ta|\!\}_l = \{\!|ta|\!\}_l$
$\{\!|convert\text{-}extTA\ f\ ta|\!\}_t = map\ (convert\text{-}new\text{-}thread\text{-}action\ f)\ \{\!|ta|\!\}_t$
$\{\!|convert\text{-}extTA\ f\ ta|\!\}_c = \{\!|ta|\!\}_c$
$\{\!|convert\text{-}extTA\ f\ ta|\!\}_w = \{\!|ta|\!\}_w$
$\{\!|convert\text{-}extTA\ f\ ta|\!\}_i = \{\!|ta|\!\}_i$
$convert\text{-}extTA\ f\ (las, tas, was, cas, is, obs) = (las, map\ (convert\text{-}new\text{-}thread\text{-}action\ f)\ tas, was, cas, is, obs)$
⟨*proof*⟩

lemma *convert-extTA-eq-conv*:
$convert\text{-}extTA\ f\ ta = ta' \longleftrightarrow$
$\{\!|ta|\!\}_l = \{\!|ta'|\!\}_l \wedge \{\!|ta|\!\}_c = \{\!|ta'|\!\}_c \wedge \{\!|ta|\!\}_w = \{\!|ta'|\!\}_w \wedge \{\!|ta|\!\}_o = \{\!|ta'|\!\}_o \wedge \{\!|ta|\!\}_i = \{\!|ta'|\!\}_i \wedge length\ \{\!|ta|\!\}_t = length\ \{\!|ta'|\!\}_t \wedge$
$(\forall n < length\ \{\!|ta|\!\}_t.\ convert\text{-}new\text{-}thread\text{-}action\ f\ (\{\!|ta|\!\}_t\ !\ n) = \{\!|ta'|\!\}_t\ !\ n)$
⟨*proof*⟩

lemma *convert-extTA-compose* [*simp*]:
$convert\text{-}extTA\ f\ (convert\text{-}extTA\ g\ ta) = convert\text{-}extTA\ (f\ o\ g)\ ta$
⟨*proof*⟩

lemma *obs-a-convert-extTA* [*simp*]: $obs\text{-}a\ (convert\text{-}extTA\ f\ ta) = obs\text{-}a\ ta$
⟨*proof*⟩

Actions for thread start/finish

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

instance *action* :: (*type*) *obs-action*
⟨*proof*⟩

definition *convert-obs-initial* :: $('l,'t,'x,'m,'w,'o)\ thread\text{-}action \Rightarrow ('l,'t,'x,'m,'w,'o\ action)\ thread\text{-}action$
where
$convert\text{-}obs\text{-}initial\ ta = (\{\!|ta|\!\}_l, \{\!|ta|\!\}_t, \{\!|ta|\!\}_c, \{\!|ta|\!\}_w, \{\!|ta|\!\}_i, map\ NormalAction\ \{\!|ta|\!\}_o)$

lemma *inj-NormalAction* [*simp*]: *inj NormalAction*
⟨*proof*⟩

lemma *convert-obs-initial-inject* [*simp*]:
$convert\text{-}obs\text{-}initial\ ta = convert\text{-}obs\text{-}initial\ ta' \longleftrightarrow ta = ta'$
⟨*proof*⟩

lemma *convert-obs-initial-empty-TA* [*simp*]:
$convert\text{-}obs\text{-}initial\ \varepsilon = \varepsilon$

⟨*proof*⟩

lemma *convert-obs-initial-eq-empty-TA* [*simp*]:
convert-obs-initial ta = $\varepsilon \longleftrightarrow$ *ta* = ε
ε = *convert-obs-initial ta* $\longleftrightarrow$ *ta* = ε
⟨*proof*⟩

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*$|\!\}_i$ = $\{\!|ta|\!\}_i$
⟨*proof*⟩

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

⟨*ML*⟩
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* $\lfloor(t', n)\rfloor$ *t* = (*if t* = *t'* *then Suc n else 0*)

lemma *has-locks-iff*:
has-locks l t = *n* $\longleftrightarrow$
(*l* = *None* ∧ *n* = *0*) ∨
($\exists n'$. *l* = $\lfloor(t, n')\rfloor$ ∧ *Suc n'* = *n*) ∨ ($\exists t'\ n'$. *l* = $\lfloor(t', n')\rfloor$ ∧ *t'* ≠ *t* ∧ *n* = *0*)
⟨*proof*⟩

lemma *has-locksE*:
⟦ *has-locks l t* = *n*;
⟦ *l* = *None*; *n* = *0* ⟧ ⟹ *P*;
$\bigwedge n'$. ⟦ *l* = $\lfloor(t, n')\rfloor$; *Suc n'* = *n* ⟧ ⟹ *P*;
$\bigwedge t'\ n'$. ⟦ *l* = $\lfloor(t', n')\rfloor$; *t'* ≠ *t*; *n* = *0* ⟧ ⟹ *P* ⟧
⟹ *P*
⟨*proof*⟩

inductive *may-lock* :: '*t lock* ⇒ '*t* ⇒ *bool* **where**

may-lock None t
| *may-lock* $\lfloor(t, n)\rfloor$ *t*

lemma *may-lock-iff* [*code*]:
may-lock l t = (*case l of None* ⇒ *True* | $\lfloor(t', n)\rfloor$ ⇒ $t = t'$)
⟨*proof*⟩

lemma *may-lockI*:
$l = None \vee (\exists n.\ l = \lfloor(t, n)\rfloor) \Longrightarrow$ *may-lock l t*
⟨*proof*⟩

lemma *may-lockE* [*consumes 1*, *case-names None Locked*]:
⟦ *may-lock l t*; $l = None \Longrightarrow P$; $\bigwedge n.\ l = \lfloor(t, n)\rfloor \Longrightarrow P$ ⟧ $\Longrightarrow P$
⟨*proof*⟩

lemma *may-lock-may-lock-t-eq*:
⟦ *may-lock l t*; *may-lock l t'* ⟧ $\Longrightarrow (l = None) \vee (t = t')$
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *has-lock-may-lock*:
has-lock l t ⟹ *may-lock l t*
⟨*proof*⟩

lemma *has-lock-may-lock-t-eq*:
⟦ *has-lock l t*; *may-lock l t'* ⟧ $\Longrightarrow t = t'$
⟨*proof*⟩

lemma *has-locks-has-locks-t-eq*:
⟦*has-locks l t* = *Suc n*; *has-locks l t'* = *Suc n'*⟧ $\Longrightarrow t = t'$
⟨*proof*⟩

lemma *has-lock-has-lock-t-eq*:
⟦ *has-lock l t*; *has-lock l t'* ⟧ $\Longrightarrow t = t'$
⟨*proof*⟩

lemma *not-may-lock-conv*:
¬ *may-lock l t* ⟷ (∃ *t'*. $t' \neq t$ ∧ *has-lock l t'*)
⟨*proof*⟩

fun *lock-lock* :: $'t\ lock \Rightarrow 't \Rightarrow 't\ lock$ **where**
$lock\text{-}lock\ None\ t = \lfloor(t,\ 0)\rfloor$
$|\ lock\text{-}lock\ \lfloor(t',\ n)\rfloor\ t = \lfloor(t',\ Suc\ n)\rfloor$

fun *unlock-lock* :: $'t\ lock \Rightarrow 't\ lock$ **where**
$unlock\text{-}lock\ None = None$
$|\ unlock\text{-}lock\ \lfloor(t,\ n)\rfloor = (case\ n\ of\ 0 \Rightarrow None\ |\ Suc\ n' \Rightarrow \lfloor(t,\ n')\rfloor)$

fun *release-all* :: $'t\ lock \Rightarrow 't \Rightarrow 't\ lock$ **where**
$release\text{-}all\ None\ t = None$
$|\ release\text{-}all\ \lfloor(t',\ n)\rfloor\ t = (if\ t = t'\ then\ None\ else\ \lfloor(t',\ n)\rfloor)$

fun *acquire-locks* :: $'t\ lock \Rightarrow 't \Rightarrow nat \Rightarrow 't\ lock$ **where**
$acquire\text{-}locks\ L\ t\ 0 = L$
$|\ acquire\text{-}locks\ L\ t\ (Suc\ m) = acquire\text{-}locks\ (lock\text{-}lock\ L\ t)\ t\ m$

lemma *acquire-locks-conv*:
$acquire\text{-}locks\ L\ t\ n = (case\ L\ of\ None \Rightarrow (case\ n\ of\ 0 \Rightarrow None\ |\ Suc\ m \Rightarrow \lfloor(t,\ m)\rfloor)\ |\ \lfloor(t',\ m)\rfloor \Rightarrow \lfloor(t',\ n + m)\rfloor)$
⟨*proof*⟩

lemma *lock-lock-ls-Some*:
$\exists t'\ n.\ lock\text{-}lock\ l\ t = \lfloor(t',\ n)\rfloor$
⟨*proof*⟩

lemma *unlock-lock-SomeD*:
$unlock\text{-}lock\ l = \lfloor(t',\ n)\rfloor \Longrightarrow l = \lfloor(t',\ Suc\ n)\rfloor$
⟨*proof*⟩

lemma *has-locks-Suc-lock-lock-has-locks-Suc-Suc*:
$has\text{-}locks\ l\ t = Suc\ n \Longrightarrow has\text{-}locks\ (lock\text{-}lock\ l\ t)\ t = Suc\ (Suc\ n)$
⟨*proof*⟩

lemma *has-locks-lock-lock-conv* [*simp*]:
$may\text{-}lock\ l\ t \Longrightarrow has\text{-}locks\ (lock\text{-}lock\ l\ t)\ t = Suc\ (has\text{-}locks\ l\ t)$
⟨*proof*⟩

lemma *has-locks-release-all-conv* [*simp*]:
$has\text{-}locks\ (release\text{-}all\ l\ t)\ t = 0$
⟨*proof*⟩

lemma *may-lock-lock-lock-conv* [*simp*]: $may\text{-}lock\ (lock\text{-}lock\ l\ t)\ t = may\text{-}lock\ l\ t$
⟨*proof*⟩

lemma *has-locks-acquire-locks-conv* [*simp*]:
$may\text{-}lock\ l\ t \Longrightarrow has\text{-}locks\ (acquire\text{-}locks\ l\ t\ n)\ t = has\text{-}locks\ l\ t + n$
⟨*proof*⟩

lemma *may-lock-unlock-lock-conv* [*simp*]:
$has\text{-}lock\ l\ t \Longrightarrow may\text{-}lock\ (unlock\text{-}lock\ l)\ t = may\text{-}lock\ l\ t$
⟨*proof*⟩

lemma *may-lock-release-all-conv* [*simp*]:
  $\mathit{may\text{-}lock}\ (\mathit{release\text{-}all}\ l\ t)\ t = \mathit{may\text{-}lock}\ l\ t$
⟨*proof*⟩

lemma *may-lock-t-may-lock-unlock-lock-t*:
  $\mathit{may\text{-}lock}\ l\ t \Longrightarrow \mathit{may\text{-}lock}\ (\mathit{unlock\text{-}lock}\ l)\ t$
⟨*proof*⟩

lemma *may-lock-has-locks-lock-lock-0*:
  $[\![\mathit{may\text{-}lock}\ l\ t';\ t \neq t']\!] \Longrightarrow \mathit{has\text{-}locks}\ (\mathit{lock\text{-}lock}\ l\ t')\ t = 0$
⟨*proof*⟩

lemma *has-locks-unlock-lock-conv* [*simp*]:
  $\mathit{has\text{-}lock}\ l\ t \Longrightarrow \mathit{has\text{-}locks}\ (\mathit{unlock\text{-}lock}\ l)\ t = \mathit{has\text{-}locks}\ l\ t - 1$
⟨*proof*⟩

lemma *has-lock-lock-lock-unlock-lock-id* [*simp*]:
  $\mathit{has\text{-}lock}\ l\ t \Longrightarrow \mathit{lock\text{-}lock}\ (\mathit{unlock\text{-}lock}\ l)\ t = l$
⟨*proof*⟩

lemma *may-lock-unlock-lock-lock-lock-id* [*simp*]:
  $\mathit{may\text{-}lock}\ l\ t \Longrightarrow \mathit{unlock\text{-}lock}\ (\mathit{lock\text{-}lock}\ l\ t) = l$
⟨*proof*⟩

lemma *may-lock-has-locks-0*:
  $[\![\ \mathit{may\text{-}lock}\ l\ t;\ t \neq t'\ ]\!] \Longrightarrow \mathit{has\text{-}locks}\ l\ t' = 0$
⟨*proof*⟩

fun *upd-lock* :: $'t\ \mathit{lock} \Rightarrow 't \Rightarrow \mathit{lock\text{-}action} \Rightarrow 't\ \mathit{lock}$
where
  $\mathit{upd\text{-}lock}\ l\ t\ \mathit{Lock} = \mathit{lock\text{-}lock}\ l\ t$
| $\mathit{upd\text{-}lock}\ l\ t\ \mathit{Unlock} = \mathit{unlock\text{-}lock}\ l$
| $\mathit{upd\text{-}lock}\ l\ t\ \mathit{UnlockFail} = l$
| $\mathit{upd\text{-}lock}\ l\ t\ \mathit{ReleaseAquire} = \mathit{release\text{-}all}\ l\ t$

fun *upd-locks* :: $'t\ \mathit{lock} \Rightarrow 't \Rightarrow \mathit{lock\text{-}action}\ \mathit{list} \Rightarrow 't\ \mathit{lock}$
where
  $\mathit{upd\text{-}locks}\ l\ t\ [] = l$
| $\mathit{upd\text{-}locks}\ l\ t\ (L\ \#\ \mathit{Ls}) = \mathit{upd\text{-}locks}\ (\mathit{upd\text{-}lock}\ l\ t\ L)\ t\ \mathit{Ls}$

lemma *upd-locks-append* [*simp*]:
  $\mathit{upd\text{-}locks}\ l\ t\ (\mathit{Ls}\ @\ \mathit{Ls}') = \mathit{upd\text{-}locks}\ (\mathit{upd\text{-}locks}\ l\ t\ \mathit{Ls})\ t\ \mathit{Ls}'$
⟨*proof*⟩

lemma *upd-lock-Some-thread-idD*:
  assumes *ul*: $\mathit{upd\text{-}lock}\ l\ t\ L = \lfloor(t',\ n)\rfloor$
  and *tt'*: $t \neq t'$
  shows $\exists\, n.\ l = \lfloor(t',\ n)\rfloor$
⟨*proof*⟩

lemma *has-lock-upd-lock-implies-has-lock*:
$\llbracket$ *has-lock* (*upd-lock l t L*) *t'*; $t \neq t'$ $\rrbracket \Longrightarrow$ *has-lock l t'*
⟨*proof*⟩

lemma *has-lock-upd-locks-implies-has-lock*:
$\llbracket$ *has-lock* (*upd-locks l t Ls*) *t'*; $t \neq t'$ $\rrbracket \Longrightarrow$ *has-lock l t'*
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *not-lock-action-okE* [*consumes 1*, *case-names Lock Unlock UnlockFail*]:
$\llbracket$ ¬ *lock-action-ok l t L*;
  $\llbracket$ *L* = *Lock*; ¬ *may-lock l t* $\rrbracket \Longrightarrow Q$;
  $\llbracket$ *L* = *Unlock*; ¬ *has-lock l t* $\rrbracket \Longrightarrow Q$;
  $\llbracket$ *L* = *UnlockFail*; *has-lock l t* $\rrbracket \Longrightarrow Q\rrbracket$
$\Longrightarrow Q$
⟨*proof*⟩

lemma *may-lock-upd-lock-conv* [*simp*]:
lock-action-ok l t L $\Longrightarrow$ *may-lock* (*upd-lock l t L*) *t* = *may-lock l t*
⟨*proof*⟩

lemma *may-lock-upd-locks-conv* [*simp*]:
lock-actions-ok l t Ls $\Longrightarrow$ *may-lock* (*upd-locks l t Ls*) *t* = *may-lock l t*
⟨*proof*⟩

lemma *lock-actions-ok-Lock-may-lock*:
$\llbracket$ *lock-actions-ok l t Ls*; *Lock* ∈ *set Ls* $\rrbracket \Longrightarrow$ *may-lock l t*
⟨*proof*⟩

lemma *has-locks-lock-lock-conv'* [*simp*]:
$\llbracket$ *may-lock l t'*; $t \neq t'$ $\rrbracket \Longrightarrow$ *has-locks* (*lock-lock l t'*) *t* = *has-locks l t*
⟨*proof*⟩

lemma *has-locks-unlock-lock-conv'* [*simp*]:
$\llbracket$ *has-lock l t'*; $t \neq t'$ $\rrbracket \Longrightarrow$ *has-locks* (*unlock-lock l*) *t* = *has-locks l t*
⟨*proof*⟩

lemma *has-locks-release-all-conv'* [*simp*]:
$t \neq t' \Longrightarrow$ *has-locks* (*release-all l t'*) *t* = *has-locks l t*

⟨*proof*⟩

lemma *has-locks-acquire-locks-conv'* [*simp*]:
$\llbracket$ *may-lock l t*; $t \neq t'$ $\rrbracket \Longrightarrow$ *has-locks* (*acquire-locks l t n*) t' = *has-locks l t'*
⟨*proof*⟩

lemma *lock-action-ok-has-locks-upd-lock-eq-has-locks* [*simp*]:
$\llbracket$ *lock-action-ok l t' L*; $t \neq t'$ $\rrbracket \Longrightarrow$ *has-locks* (*upd-lock l t' L*) *t* = *has-locks l t*
⟨*proof*⟩

lemma *lock-actions-ok-has-locks-upd-locks-eq-has-locks* [*simp*]:
$\llbracket$ *lock-actions-ok l t' Ls*; $t \neq t'$ $\rrbracket \Longrightarrow$ *has-locks* (*upd-locks l t' Ls*) *t* = *has-locks l t*
⟨*proof*⟩

lemma *has-lock-acquire-locks-implies-has-lock*:
$\llbracket$ *has-lock* (*acquire-locks l t n*) t'; $t \neq t'$ $\rrbracket \Longrightarrow$ *has-lock l t'*
⟨*proof*⟩

lemma *has-lock-has-lock-acquire-locks*:
has-lock l T $\Longrightarrow$ *has-lock* (*acquire-locks l t n*) *T*
⟨*proof*⟩

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*⟩

lemma *lock-actions-ok'E*[*consumes 1*, *case-names ok Lock*]:
$\llbracket$ *lock-actions-ok' l t las*;
lock-actions-ok l t las $\Longrightarrow$ *P*;
⋀*xs ys*. $\llbracket$ *las* = *xs* @ *Lock* # *ys*; *lock-actions-ok l t xs*; ¬ *may-lock* (*upd-locks l t xs*) *t* $\rrbracket \Longrightarrow$ *P* $\rrbracket$
$\Longrightarrow$ *P*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *upd-locks-empty-conv* [*simp*]: (λ(*l*, *las*). *upd-locks l t las*) ∘$ ($*ls*, *K*$ []$) = *ls*
⟨*proof*⟩

lemma *redT-updLs-Some-thread-idD*:
⟦ *has-lock* (*redT-updLs ls t las* $ *l*) *t*′; *t* ≠ *t*′ ⟧ ⟹ *has-lock* (*ls* $ *l*) *t*′
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *lock-ok-lasE*:
⟦ *lock-ok-las ls t las*; (⋀*l*. *lock-actions-ok* (*ls* $ *l*) *t* (*las* $ *l*)) ⟹ *Q* ⟧ ⟹ *Q*
⟨*proof*⟩

lemma *lock-ok-lasD*:
lock-ok-las ls t las ⟹ *lock-actions-ok* (*ls* $ *l*) *t* (*las* $ *l*)
⟨*proof*⟩

lemma *lock-ok-las-code* [*code*]:
lock-ok-las ls t las = *finfun-All* ((λ(*l*, *la*). *lock-actions-ok l t la*) ∘$ ($*ls*, *las*$))
⟨*proof*⟩

lemma *lock-ok-las-may-lock*:
⟦ *lock-ok-las ls t las*; *Lock* ∈ *set* (*las* $ *l*) ⟧ ⟹ *may-lock* (*ls* $ *l*) *t*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*]:

$\bigwedge ln.$ ($\bigwedge l.$ *ln* $ *l* > *0* ⟹ *may-lock* (*ls* $ *l*) *t*) ⟹ *may-acquire-all ls t ln*
⟨*proof*⟩

lemma *may-acquire-allE*:
$\bigwedge ln.$ ⟦ *may-acquire-all ls t ln*; ∀ *l*. *ln* $ *l* > *0* ⟶ *may-lock* (*ls* $ *l*) *t* ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *may-acquire-allD* [*dest*]:
$\bigwedge ln.$ ⟦ *may-acquire-all ls t ln*; *ln* $ *l* > *0* ⟧ ⟹ *may-lock* (*ls* $ *l*) *t*
⟨*proof*⟩

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′*
⟨*proof*⟩

lemma *may-acquire-all-code* [*code*]:
$\bigwedge ln.$ *may-acquire-all ls t ln* ⟷ *finfun-All* ((λ(*lock*, *n*). *n* > *0* ⟶ *may-lock lock t*) ∘$ ($*ls*, *ln*$))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *collect-locksE*:
⟦ *l* ∈ *collect-locks las*; *Lock* ∈ *set* (*las* $ *l*) ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *collect-locksD*:
l ∈ *collect-locks las* ⟹ *Lock* ∈ *set* (*las* $ *l*)
⟨*proof*⟩

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*⟩

lemma *must-acquire-lock-contains-lock*:
must-acquire-lock las ⟹ *Lock* ∈ *set las*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *collect-locks'E*:
⟦ *l* ∈ *collect-locks' las*; *must-acquire-lock* (*las* \$ *l*) ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *collect-locks'-subset-collect-locks*:
collect-locks' las ⊆ *collect-locks las*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *lock-ok-las'D*: *lock-ok-las' ls t las* ⟹ *lock-actions-ok'* (*ls* \$ *l*) *t* (*las* \$ *l*)
⟨*proof*⟩

lemma *not-lock-ok-las'-conv*:
¬ *lock-ok-las' ls t las* ⟷ (∃ *l*. ¬ *lock-actions-ok'* (*ls* \$ *l*) *t* (*las* \$ *l*))
⟨*proof*⟩

lemma *lock-ok-las'-code*:
lock-ok-las' ls t las = *finfun-All* ((λ(*l*, *la*). *lock-actions-ok' l t la*) ∘\$ (\$*ls*, *las*\$))
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *lock-ok-las′-into-lock-on-las*:
$[\![$*lock-ok-las′ ls t las*; $\bigwedge$*l*. *l* ∈ *collect-locks′ las* ⟹ *may-lock* (*ls* $ *l*) *t*$]\!]$ ⟹ *lock-ok-las ls t las*
⟨*proof*⟩

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*)
⟨*proof*⟩

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′*
⟨*proof*⟩

lemma *redT-updT-None*:
redT-updT ts ta t = *None* ⟹ *ts t* = *None*
⟨*proof*⟩

lemma *redT-updTs-None*: *redT-updTs ts tas t* = *None* ⟹ *ts t* = *None*
⟨*proof*⟩

lemma *redT-updT-Some1*:
ts t = ⌊*xw*⌋ ⟹ ∃ *xw*. *redT-updT ts ta t* = ⌊*xw*⌋
⟨*proof*⟩

lemma *redT-updTs-Some1*:
  $ts\ t = \lfloor xw \rfloor \Longrightarrow \exists xw.\ redT\text{-}updTs\ ts\ tas\ t = \lfloor xw \rfloor$
⟨*proof*⟩

lemma *redT-updT-finite-dom-inv*:
  $finite\ (dom\ (redT\text{-}updT\ ts\ ta)) = finite\ (dom\ ts)$
⟨*proof*⟩

lemma *redT-updTs-finite-dom-inv*:
  $finite\ (dom\ (redT\text{-}updTs\ ts\ tas)) = finite\ (dom\ ts)$
⟨*proof*⟩

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 $redT\text{-}updT' :: ('l,'t,'x)\ thread\text{-}info \Rightarrow ('t,'x','m)\ new\text{-}thread\text{-}action \Rightarrow ('l,'t,'x)\ thread\text{-}info$
where
  $redT\text{-}updT'\ ts\ (NewThread\ t'\ x\ m) = ts(t' \mapsto (undefined,\ no\text{-}wait\text{-}locks))$
| $redT\text{-}updT'\ ts\ \text{-} = ts$

fun $redT\text{-}updTs' :: ('l,'t,'x)\ thread\text{-}info \Rightarrow ('t,'x','m)\ new\text{-}thread\text{-}action\ list \Rightarrow ('l,'t,'x)\ thread\text{-}info$
where
  $redT\text{-}updTs'\ ts\ [] = ts$
| $redT\text{-}updTs'\ ts\ (ta\#tas) = redT\text{-}updTs'\ (redT\text{-}updT'\ ts\ ta)\ tas$

lemma *redT-updT'-None*:
  $redT\text{-}updT'\ ts\ ta\ t = None \Longrightarrow ts\ t = None$
⟨*proof*⟩

primrec $thread\text{-}ok :: ('l,'t,'x)\ thread\text{-}info \Rightarrow ('t,'x','m)\ new\text{-}thread\text{-}action \Rightarrow bool$
where
  $thread\text{-}ok\ ts\ (NewThread\ t\ x\ m) = free\text{-}thread\text{-}id\ ts\ t$
| $thread\text{-}ok\ ts\ (ThreadExists\ t\ b) = (b \neq free\text{-}thread\text{-}id\ ts\ t)$

fun $thread\text{-}oks :: ('l,'t,'x)\ thread\text{-}info \Rightarrow ('t,'x','m)\ new\text{-}thread\text{-}action\ list \Rightarrow bool$
where
  $thread\text{-}oks\ ts\ [] = True$
| $thread\text{-}oks\ ts\ (ta\#tas) = (thread\text{-}ok\ ts\ ta \wedge thread\text{-}oks\ (redT\text{-}updT'\ ts\ ta)\ tas)$

lemma *thread-ok-ts-change*:
  $(\bigwedge t.\ ts\ t = None \longleftrightarrow ts'\ t = None) \Longrightarrow thread\text{-}ok\ ts\ ta \longleftrightarrow thread\text{-}ok\ ts'\ ta$
⟨*proof*⟩

lemma *thread-oks-ts-change*:
  $(\bigwedge t.\ ts\ t = None \longleftrightarrow ts'\ t = None) \Longrightarrow thread\text{-}oks\ ts\ tas \longleftrightarrow thread\text{-}oks\ ts'\ tas$
⟨*proof*⟩

lemma *redT-updT'-eq-None-conv*:
  $(\bigwedge t.\ ts\ t = None \longleftrightarrow ts'\ t = None) \Longrightarrow redT\text{-}updT'\ ts\ ta\ t = None \longleftrightarrow redT\text{-}updT\ ts'\ ta\ t = None$
⟨*proof*⟩

lemma *redT-updTs'-eq-None-conv*:

$(\bigwedge t.\ ts\ t = None \longleftrightarrow ts'\ t = None) \Longrightarrow redT\text{-}updTs'\ ts\ tas\ t = None \longleftrightarrow redT\text{-}updTs\ ts'\ tas\ t = None$
⟨proof⟩

lemma *thread-oks-redT-updT-conv* [*simp*]:
$thread\text{-}oks\ (redT\text{-}updT'\ ts\ ta)\ tas = thread\text{-}oks\ (redT\text{-}updT\ ts\ ta)\ tas$
⟨proof⟩

lemma *thread-oks-append* [*simp*]:
$thread\text{-}oks\ ts\ (tas\ @\ tas') = (thread\text{-}oks\ ts\ tas \wedge thread\text{-}oks\ (redT\text{-}updTs'\ ts\ tas)\ tas')$
⟨proof⟩

lemma *thread-oks-redT-updTs-conv* [*simp*]:
$thread\text{-}oks\ (redT\text{-}updTs'\ ts\ ta)\ tas = thread\text{-}oks\ (redT\text{-}updTs\ ts\ ta)\ tas$
⟨proof⟩

lemma *redT-updT-Some*:
$[\![\ ts\ t = \lfloor xw \rfloor;\ thread\text{-}ok\ ts\ ta\ ]\!] \Longrightarrow redT\text{-}updT\ ts\ ta\ t = \lfloor xw \rfloor$
⟨proof⟩

lemma *redT-updTs-Some*:
$[\![\ ts\ t = \lfloor xw \rfloor;\ thread\text{-}oks\ ts\ tas\ ]\!] \Longrightarrow redT\text{-}updTs\ ts\ tas\ t = \lfloor xw \rfloor$
⟨proof⟩

lemma *redT-updT'-Some*:
$[\![\ ts\ t = \lfloor xw \rfloor;\ thread\text{-}ok\ ts\ ta\ ]\!] \Longrightarrow redT\text{-}updT'\ ts\ ta\ t = \lfloor xw \rfloor$
⟨proof⟩

lemma *redT-updTs'-Some*:
$[\![\ ts\ t = \lfloor xw \rfloor;\ thread\text{-}oks\ ts\ tas\ ]\!] \Longrightarrow redT\text{-}updTs'\ ts\ tas\ t = \lfloor xw \rfloor$
⟨proof⟩

lemma *thread-ok-new-thread*:
$thread\text{-}ok\ ts\ (NewThread\ t\ m'\ x) \Longrightarrow ts\ t = None$
⟨proof⟩

lemma *thread-oks-new-thread*:
$[\![\ thread\text{-}oks\ ts\ tas;\ NewThread\ t\ x\ m \in set\ tas\ ]\!] \Longrightarrow ts\ t = None$
⟨proof⟩

lemma *redT-updT-new-thread-ts*:
$thread\text{-}ok\ ts\ (NewThread\ t\ x\ m) \Longrightarrow redT\text{-}updT\ ts\ (NewThread\ t\ x\ m)\ t = \lfloor (x,\ no\text{-}wait\text{-}locks) \rfloor$
⟨proof⟩

lemma *redT-updTs-new-thread-ts*:
$[\![\ thread\text{-}oks\ ts\ tas;\ NewThread\ t\ x\ m \in set\ tas\ ]\!] \Longrightarrow redT\text{-}updTs\ ts\ tas\ t = \lfloor (x,\ no\text{-}wait\text{-}locks) \rfloor$
⟨proof⟩

lemma *redT-updT-new-thread*:
$[\![\ redT\text{-}updT\ ts\ ta\ t = \lfloor (x,\ w) \rfloor;\ thread\text{-}ok\ ts\ ta;\ ts\ t = None\ ]\!] \Longrightarrow \exists\, m.\ ta = NewThread\ t\ x\ m \wedge w$

$= \textit{no-wait-locks}$
⟨*proof*⟩

lemma *redT-updTs-new-thread*:
$[\![$ *redT-updTs ts tas t* $= \lfloor(x,\ w)\rfloor$; *thread-oks ts tas*; *ts t* $=$ *None* $]\!]$
$\Longrightarrow \exists\, m$.*NewThread t x m* $\in$ *set tas* $\wedge$ *w* $=$ *no-wait-locks*
⟨*proof*⟩

lemma *redT-updT-upd*:
$[\![$ *ts t* $= \lfloor xw \rfloor$; *thread-ok ts ta* $]\!] \Longrightarrow$ (*redT-updT ts ta*)($t \mapsto xw'$) $=$ *redT-updT* (*ts*($t \mapsto xw'$)) *ta*
⟨*proof*⟩

lemma *redT-updTs-upd*:
$[\![$ *ts t* $= \lfloor xw \rfloor$; *thread-oks ts tas* $]\!] \Longrightarrow$ (*redT-updTs ts tas*)($t \mapsto xw'$) $=$ *redT-updTs* (*ts*($t \mapsto xw'$)) *tas*
⟨*proof*⟩

lemma *thread-ok-upd*:
ts t $= \lfloor xln \rfloor \Longrightarrow$ *thread-ok* (*ts*($t \mapsto xln'$)) *ta* $=$ *thread-ok ts ta*
⟨*proof*⟩

lemma *thread-oks-upd*:
ts t $= \lfloor xln \rfloor \Longrightarrow$ *thread-oks* (*ts*($t \mapsto xln'$)) *tas* $=$ *thread-oks ts tas*
⟨*proof*⟩

lemma *thread-ok-convert-new-thread-action* [*simp*]:
thread-ok ts (*convert-new-thread-action f ta*) $=$ *thread-ok ts ta*
⟨*proof*⟩

lemma *redT-updT'-convert-new-thread-action-eq-None*:
redT-updT' ts (*convert-new-thread-action f ta*) *t* $=$ *None* $\longleftrightarrow$ *redT-updT' ts ta t* $=$ *None*
⟨*proof*⟩

lemma *thread-oks-convert-new-thread-action* [*simp*]:
thread-oks ts (*map* (*convert-new-thread-action f*) *tas*) $=$ *thread-oks ts tas*
⟨*proof*⟩

lemma *map-redT-updT*:
map-option (*map-prod f id*) (*redT-updT ts ta t*) $=$
redT-updT (λt. *map-option* (*map-prod f id*) (*ts t*)) (*convert-new-thread-action f ta*) *t*
⟨*proof*⟩

lemma *map-redT-updTs*:
map-option (*map-prod f id*) (*redT-updTs ts tas t*) $=$
redT-updTs (λt. *map-option* (*map-prod f id*) (*ts t*)) (*map* (*convert-new-thread-action f*) *tas*) *t*
⟨*proof*⟩

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
$ws\ t' = \lfloor InWS\ w \rfloor \Longrightarrow redT\text{-}updW\ t\ ws\ (Notify\ w)\ (ws(t' \mapsto PostWS\ WSNotified))$
| $(\bigwedge t'.\ ws\ t' \neq \lfloor InWS\ w \rfloor) \Longrightarrow redT\text{-}updW\ t\ ws\ (Notify\ w)\ ws$
| $redT\text{-}updW\ t\ ws\ (NotifyAll\ w)\ (\lambda t.\ if\ ws\ t = \lfloor InWS\ w \rfloor\ then\ \lfloor PostWS\ WSNotified \rfloor\ else\ ws\ t)$
| $redT\text{-}updW\ t\ ws\ (Suspend\ w)\ (ws(t \mapsto InWS\ w))$
| $ws\ t' = \lfloor InWS\ w \rfloor \Longrightarrow redT\text{-}updW\ t\ ws\ (WakeUp\ t')\ (ws(t' \mapsto PostWS\ WSInterrupted))$
| $(\bigwedge w.\ ws\ t' \neq \lfloor InWS\ w \rfloor) \Longrightarrow redT\text{-}updW\ t\ ws\ (WakeUp\ t')\ ws$
| $redT\text{-}updW\ t\ ws\ Notified\ (ws(t := None))$
| $redT\text{-}updW\ t\ ws\ WokenUp\ (ws(t := None))$

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

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

lemma *redT-updW-total*: $\exists\, ws'.\ redT\text{-}updW\ t\ ws\ wa\ ws'$
⟨*proof*⟩

lemma *redT-updWs-total*: $\exists\, ws'.\ redT\text{-}updWs\ t\ ws\ was\ ws'$
⟨*proof*⟩

lemma *redT-updWs-trans*: $\llbracket\ redT\text{-}updWs\ t\ ws\ was\ ws';\ redT\text{-}updWs\ t\ ws'\ was'\ ws''\ \rrbracket \Longrightarrow redT\text{-}updWs\ t\ ws\ (was\ @\ was')\ ws''$
⟨*proof*⟩

lemma *redT-updW-None-implies-None*:
$\llbracket\ redT\text{-}updW\ t'\ ws\ wa\ ws';\ ws\ t = None;\ t \neq t'\ \rrbracket \Longrightarrow ws'\ t = None$
⟨*proof*⟩

lemma *redT-updWs-None-implies-None*:
assumes $redT\text{-}updWs\ t'\ ws\ was\ ws'$
and $t \neq t'$ **and** $ws\ t = None$
shows $ws'\ t = None$
⟨*proof*⟩

lemma *redT-updW-PostWS-imp-PostWS*:
$\llbracket\ redT\text{-}updW\ t\ ws\ wa\ ws';\ ws\ t'' = \lfloor PostWS\ w \rfloor;\ t'' \neq t\ \rrbracket \Longrightarrow\ ws'\ t'' = \lfloor PostWS\ w \rfloor$
⟨*proof*⟩

lemma *redT-updWs-PostWS-imp-PostWS*:
$\llbracket\ redT\text{-}updWs\ t\ ws\ was\ ws';\ t'' \neq t;\ ws\ t'' = \lfloor PostWS\ w \rfloor\ \rrbracket \Longrightarrow ws'\ t'' = \lfloor PostWS\ w \rfloor$

⟨*proof*⟩

lemma *redT-updW-Some-otherD*:
$\llbracket$ *redT-updW* t' *ws wa ws'*; $ws'\ t = \lfloor w \rfloor$; $t \neq t'$ $\rrbracket$
$\Longrightarrow$ (*case* w *of InWS* $w' \Rightarrow ws\ t = \lfloor InWS\ w' \rfloor$ | - $\Rightarrow ws\ t = \lfloor w \rfloor \vee (\exists w'.\ ws\ t = \lfloor InWS\ w' \rfloor)$)
⟨*proof*⟩

lemma *redT-updWs-Some-otherD*:
$\llbracket$ *redT-updWs* t' *ws was ws'*; $ws'\ t = \lfloor w \rfloor$; $t \neq t'$ $\rrbracket$
$\Longrightarrow$ (*case* w *of InWS* $w' \Rightarrow ws\ t = \lfloor InWS\ w' \rfloor$ | - $\Rightarrow ws\ t = \lfloor w \rfloor \vee (\exists w'.\ ws\ t = \lfloor InWS\ w' \rfloor)$)
⟨*proof*⟩

lemma *redT-updW-None-SomeD*:
$\llbracket$ *redT-updW t ws wa ws'*; $ws'\ t' = \lfloor w \rfloor$; $ws\ t' = None$ $\rrbracket \Longrightarrow t = t' \wedge (\exists w'.\ w = InWS\ w' \wedge wa = Suspend\ w')$
⟨*proof*⟩

lemma *redT-updWs-None-SomeD*:
$\llbracket$ *redT-updWs t ws was ws'*; $ws'\ t' = \lfloor w \rfloor$; $ws\ t' = None$ $\rrbracket \Longrightarrow t = t' \wedge (\exists w'.\ Suspend\ w' \in set\ was)$
⟨*proof*⟩

lemma *redT-updW-neq-Some-SomeD*:
$\llbracket$ *redT-updW* t' *ws wa ws'*; $ws'\ t = \lfloor InWS\ w \rfloor$; $ws\ t \neq \lfloor InWS\ w \rfloor$ $\rrbracket \Longrightarrow t = t' \wedge wa = Suspend\ w$
⟨*proof*⟩

lemma *redT-updWs-neq-Some-SomeD*:
$\llbracket$ *redT-updWs t ws was ws'*; $ws'\ t' = \lfloor InWS\ w \rfloor$; $ws\ t' \neq \lfloor InWS\ w \rfloor$ $\rrbracket \Longrightarrow t = t' \wedge Suspend\ w \in set\ was$
⟨*proof*⟩

lemma *redT-updW-not-Suspend-Some*:
$\llbracket$ *redT-updW t ws wa ws'*; $ws'\ t = \lfloor w' \rfloor$; $ws\ t = \lfloor w \rfloor$; $\bigwedge w.\ wa \neq Suspend\ w$ $\rrbracket$
$\Longrightarrow w' = w \vee (\exists w''\ w'''.\ w = InWS\ w'' \wedge w' = PostWS\ w''')$
⟨*proof*⟩

lemma *redT-updWs-not-Suspend-Some*:
$\llbracket$ *redT-updWs t ws was ws'*; $ws'\ t = \lfloor w' \rfloor$; $ws\ t = \lfloor w \rfloor$; $\bigwedge w.\ Suspend\ w \notin set\ was$ $\rrbracket$
$\Longrightarrow w' = w \vee (\exists w''\ w'''.\ w = InWS\ w'' \wedge w' = PostWS\ w''')$
⟨*proof*⟩

lemma *redT-updWs-WokenUp-SuspendD*:
$\llbracket$ *redT-updWs t ws was ws'*; $Notified \in set\ was \vee WokenUp \in set\ was$; $ws'\ t = \lfloor w \rfloor$ $\rrbracket \Longrightarrow \exists w.\ Suspend\ w \in set\ was$
⟨*proof*⟩

lemma *redT-updW-Woken-Up-same-no-Notified-Interrupted*:
$\llbracket$ *redT-updW t ws wa ws'*; $ws'\ t = \lfloor PostWS\ w \rfloor$; $ws\ t = \lfloor PostWS\ w \rfloor$; $\bigwedge w.\ wa \neq Suspend\ w$ $\rrbracket$
$\Longrightarrow wa \neq Notified \wedge wa \neq WokenUp$
⟨*proof*⟩

lemma *redT-updWs-Woken-Up-same-no-Notified-Interrupted*:
$\llbracket$ *redT-updWs t ws was ws'*; $ws'\ t = \lfloor PostWS\ w \rfloor$; $ws\ t = \lfloor PostWS\ w \rfloor$; $\bigwedge w.\ Suspend\ w \notin set\ was$ $\rrbracket$
$\Longrightarrow Notified \notin set\ was \wedge WokenUp \notin set\ was$
⟨*proof*⟩

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*
⟨*proof*⟩

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'*⌋)
⟨*proof*⟩

lemma *waiting-code* [*code*]:
waiting None = *False*
⋀*w*. *waiting* ⌊*PostWS w*⌋ = *False*
⋀*w*. *waiting* ⌊*InWS w*⌋ = *True*
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *cond-action-oks-conv-set*:
  $\textit{cond-action-oks s t cts} \longleftrightarrow (\forall\, ct \in \textit{set cts. cond-action-ok s t ct})$
⟨*proof*⟩

lemma *cond-action-ok-Join*:
  $\bigwedge ln.\ [\![\ \textit{cond-action-ok s t (Join T)};\ \textit{thr s T} = \lfloor(x,\ ln)\rfloor\ ]\!] \Longrightarrow \textit{final } x \wedge ln = \textit{no-wait-locks} \wedge \textit{wset s T} = \textit{None}$
⟨*proof*⟩

lemma *cond-action-oks-Join*:
  $\bigwedge ln.\ [\![\ \textit{cond-action-oks s t cas};\ \textit{Join T} \in \textit{set cas};\ \textit{thr s T} = \lfloor(x,\ ln)\rfloor\ ]\!]$
  $\Longrightarrow \textit{final } x \wedge ln = \textit{no-wait-locks} \wedge \textit{wset s T} = \textit{None} \wedge t \neq T$
⟨*proof*⟩

lemma *cond-action-oks-upd*:
  assumes *tst*: $\textit{thr s t} = \lfloor xln \rfloor$
  shows $\textit{cond-action-oks (locks s, ((thr s)(t} \mapsto xln'), \textit{shr s), wset s, interrupts s) t cas} = \textit{cond-action-oks s t cas}$
⟨*proof*⟩

lemma *cond-action-ok-shr-change*:
  $\textit{cond-action-ok (ls, (ts, m), ws, is) t ct} \Longrightarrow \textit{cond-action-ok (ls, (ts, m'), ws, is) t ct}$
⟨*proof*⟩

lemma *cond-action-oks-shr-change*:
  $\textit{cond-action-oks (ls, (ts, m), ws, is) t cts} \Longrightarrow \textit{cond-action-oks (ls, (ts, m'), ws, is) t cts}$
⟨*proof*⟩

primrec *cond-action-ok'* :: $('l,'t,'x,'m,'w)\ \textit{state} \Rightarrow 't \Rightarrow 't\ \textit{conditional-action} \Rightarrow \textit{bool}$
where
  $\textit{cond-action-ok'}$ - - $(\textit{Join } t) = \textit{True}$
| $\textit{cond-action-ok'}$ - - $\textit{Yield} = \textit{True}$

primrec *cond-action-oks'* :: $('l,'t,'x,'m,'w)\ \textit{state} \Rightarrow 't \Rightarrow 't\ \textit{conditional-action list} \Rightarrow \textit{bool}$ **where**
  $\textit{cond-action-oks' s t } [] = \textit{True}$
| $\textit{cond-action-oks' s t } (ct\#cts) = (\textit{cond-action-ok' s t ct} \wedge \textit{cond-action-oks' s t cts})$

lemma *cond-action-oks'-append* [*simp*]:
  $\textit{cond-action-oks' s t (cts @ cts')} \longleftrightarrow \textit{cond-action-oks' s t cts} \wedge \textit{cond-action-oks' s t cts'}$
⟨*proof*⟩

lemma *cond-action-oks'-subset-Join*:
  $\textit{set cts} \subseteq \textit{insert Yield (range Join)} \Longrightarrow \textit{cond-action-oks' s t cts}$
⟨*proof*⟩

end

definition *collect-cond-actions* :: $'t\ \textit{conditional-action list} \Rightarrow 't\ \textit{set}$ **where**
  $\textit{collect-cond-actions cts} = \{t.\ \textit{Join } t \in \textit{set cts}\}$

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

lemma *cond-action-ok-final-change*:
  $[\![\ \textit{final-thread.cond-action-ok final1 s1 t ca};$

$\bigwedge t.\ thr\ s1\ t = None \longleftrightarrow thr\ s2\ t = None$;
$\bigwedge t\ x1.\ [\![\ thr\ s1\ t = \lfloor(x1,\ no\text{-}wait\text{-}locks)\rfloor;\ final1\ x1;\ wset\ s1\ t = None\ ]\!]$
$\Longrightarrow \exists\, x2.\ thr\ s2\ t = \lfloor(x2,\ no\text{-}wait\text{-}locks)\rfloor \wedge final2\ x2 \wedge ln2 = no\text{-}wait\text{-}locks \wedge wset\ s2\ t = None\ ]\!]$
$\Longrightarrow final\text{-}thread.cond\text{-}action\text{-}ok\ final2\ s2\ t\ ca$
⟨proof⟩

lemma *cond-action-oks-final-change*:
assumes *major*: *final-thread.cond-action-oks final1 s1 t cas*
and *minor*: $\bigwedge t.\ thr\ s1\ t = None \longleftrightarrow thr\ s2\ t = None$
$\bigwedge t\ x1.\ [\![\ thr\ s1\ t = \lfloor(x1,\ no\text{-}wait\text{-}locks)\rfloor;\ final1\ x1;\ wset\ s1\ t = None\ ]\!]$
$\Longrightarrow \exists\, x2.\ thr\ s2\ t = \lfloor(x2,\ no\text{-}wait\text{-}locks)\rfloor \wedge final2\ x2 \wedge ln2 = no\text{-}wait\text{-}locks \wedge wset\ s2\ t = None$
shows *final-thread.cond-action-oks final2 s2 t cas*
⟨proof⟩

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
$lock\text{-}thread\text{-}ok :: ('l,\ 't)\ locks \Rightarrow ('l,\ 't,'x)\ thread\text{-}info \Rightarrow bool$
where [*code del*]:
$lock\text{-}thread\text{-}ok\ ls\ ts \equiv \forall\, l\ t.\ has\text{-}lock\ (ls\ \$\ l)\ t \longrightarrow (\exists\, xw.\ ts\ t = \lfloor xw\rfloor)$

lemma *lock-thread-ok-code* [*code*]:
$lock\text{-}thread\text{-}ok\ ls\ ts = finfun\text{-}All\ ((\lambda l.\ case\ l\ of\ None \Rightarrow True \mid \lfloor(t,\ n)\rfloor \Rightarrow (ts\ t \neq None)) \circ\$\ ls)$
⟨proof⟩

lemma *lock-thread-okI*:
$(\bigwedge l\ t.\ has\text{-}lock\ (ls\ \$\ l)\ t \Longrightarrow \exists\, xw.\ ts\ t = \lfloor xw\rfloor) \Longrightarrow lock\text{-}thread\text{-}ok\ ls\ ts$
⟨proof⟩

lemma *lock-thread-okD*:
$[\![\ lock\text{-}thread\text{-}ok\ ls\ ts;\ has\text{-}lock\ (ls\ \$\ l)\ t\ ]\!] \Longrightarrow \exists\, xw.\ ts\ t = \lfloor xw\rfloor$
⟨proof⟩

lemma *lock-thread-okD'*:
$[\![\ lock\text{-}thread\text{-}ok\ ls\ ts;\ has\text{-}locks\ (ls\ \$\ l)\ t = Suc\ n\ ]\!] \Longrightarrow \exists\, xw.\ ts\ t = \lfloor xw\rfloor$
⟨proof⟩

lemma *lock-thread-okE*:
$[\![\ lock\text{-}thread\text{-}ok\ ls\ ts;\ \forall\, l\ t.\ has\text{-}lock\ (ls\ \$\ l)\ t \longrightarrow (\exists\, xw.\ ts\ t = \lfloor xw\rfloor) \Longrightarrow P\ ]\!] \Longrightarrow P$
⟨proof⟩

lemma *lock-thread-ok-upd*:
$lock\text{-}thread\text{-}ok\ ls\ ts \Longrightarrow lock\text{-}thread\text{-}ok\ ls\ (ts(t \mapsto xw))$

⟨*proof*⟩

lemma *lock-thread-ok-has-lockE*:
assumes *lock-thread-ok ls ts*
and *has-lock* (*ls* $ *l*) *t*
obtains *x ln′* **where** $ts\ t = \lfloor(x,\ ln')\rfloor$
⟨*proof*⟩

lemma *redT-updLs-preserves-lock-thread-ok*:
assumes *lto*: *lock-thread-ok ls ts*
and *tst*: $ts\ t = \lfloor xw \rfloor$
shows *lock-thread-ok* (*redT-updLs ls t las*) *ts*
⟨*proof*⟩

lemma *redT-updTs-preserves-lock-thread-ok*:
assumes *lto*: *lock-thread-ok ls ts*
shows *lock-thread-ok ls* (*redT-updTs ts nts*)
⟨*proof*⟩

lemma *lock-thread-ok-has-lock*:
assumes *lock-thread-ok ls ts*
and *has-lock* (*ls* $ *l*) *t*
obtains *xw* **where** $ts\ t = \lfloor xw \rfloor$
⟨*proof*⟩

lemma *lock-thread-ok-None-has-locks-0*:
⟦ *lock-thread-ok ls ts*; *ts t* = *None* ⟧ ⟹ *has-locks* (*ls* $ *l*) *t* = *0*
⟨*proof*⟩

lemma *redT-upds-preserves-lock-thread-ok*:
⟦*lock-thread-ok ls ts*; $ts\ t = \lfloor xw \rfloor$; *thread-oks ts tas*⟧
⟹ *lock-thread-ok* (*redT-updLs ls t las*) ((*redT-updTs ts tas*)($t \mapsto xw'$))
⟨*proof*⟩

lemma *acquire-all-preserves-lock-thread-ok*:
fixes *ln*
shows ⟦ *lock-thread-ok ls ts*; $ts\ t = \lfloor(x,\ ln)\rfloor$ ⟧ ⟹ *lock-thread-ok* (*acquire-all ls t ln*) (*ts*($t \mapsto xw$))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *wset-thread-okD*:
⟦ *wset-thread-ok ws ts*; *ts t* = *None* ⟧ ⟹ *ws t* = *None*
⟨*proof*⟩

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

lemma *wset-thread-ok-upd*:
$wset\text{-}thread\text{-}ok\ ls\ ts \Longrightarrow wset\text{-}thread\text{-}ok\ ls\ (ts(t \mapsto xw))$
⟨*proof*⟩

lemma *wset-thread-ok-upd-None*:
$wset\text{-}thread\text{-}ok\ ws\ ts \Longrightarrow wset\text{-}thread\text{-}ok\ (ws(t := None))\ (ts(t := None))$
⟨*proof*⟩

lemma *wset-thread-ok-upd-Some*:
$wset\text{-}thread\text{-}ok\ ws\ ts \Longrightarrow wset\text{-}thread\text{-}ok\ (ws(t := wo))\ (ts(t \mapsto xln))$
⟨*proof*⟩

lemma *wset-thread-ok-upd-ws*:
$[\![\ wset\text{-}thread\text{-}ok\ ws\ ts;\ ts\ t = \lfloor xln \rfloor\ ]\!] \Longrightarrow wset\text{-}thread\text{-}ok\ (ws(t := w))\ ts$
⟨*proof*⟩

lemma *wset-thread-ok-NotifyAllI*:
$wset\text{-}thread\text{-}ok\ ws\ ts \Longrightarrow wset\text{-}thread\text{-}ok\ (\lambda t.\ if\ ws\ t = \lfloor w\ t \rfloor\ then\ \lfloor w'\ t \rfloor\ else\ ws\ t)\ ts$
⟨*proof*⟩

lemma *redT-updTs-preserves-wset-thread-ok*:
assumes *wto*: *wset-thread-ok ws ts*
shows *wset-thread-ok ws* (*redT-updTs ts nts*)
⟨*proof*⟩

lemma *redT-updW-preserve-wset-thread-ok*:
$[\![\ wset\text{-}thread\text{-}ok\ ws\ ts;\ redT\text{-}updW\ t\ ws\ wa\ ws';\ ts\ t = \lfloor xln \rfloor\ ]\!] \Longrightarrow wset\text{-}thread\text{-}ok\ ws'\ ts$
⟨*proof*⟩

lemma *redT-updWs-preserve-wset-thread-ok*:
$[\![\ wset\text{-}thread\text{-}ok\ ws\ ts;\ redT\text{-}updWs\ t\ ws\ was\ ws';\ ts\ t = \lfloor xln \rfloor\ ]\!] \Longrightarrow wset\text{-}thread\text{-}ok\ ws'\ ts$
⟨*proof*⟩

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

context *final-thread* **begin**

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

lemma *wset-final-okI*:
$(\bigwedge t\ w.\ ws\ t = \lfloor w \rfloor \Longrightarrow \exists x\ ln.\ ts\ t = \lfloor (x, ln) \rfloor \wedge \neg\ final\ x) \Longrightarrow wset\text{-}final\text{-}ok\ ws\ ts$
⟨*proof*⟩

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

lemma *wset-final-okE*:
assumes $wset\text{-}final\text{-}ok\ ws\ ts\ ws\ t = \lfloor w \rfloor$
and $\bigwedge x\ ln.\ ts\ t = \lfloor (x, ln) \rfloor \Longrightarrow \neg\ final\ x \Longrightarrow thesis$
shows *thesis*
⟨*proof*⟩

lemma *wset-final-ok-imp-wset-thread-ok*:
wset-final-ok ws ts $\Longrightarrow$ *wset-thread-ok ws ts*
⟨*proof*⟩

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'*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *upd-threadRs-comp-empty* [*simp*]: (λ(*l*, *n*, *las*). *upd-threadRs n l t las*) ∘$ ($*ls*, ($*lns*, *K*$ []$)$)
= *lns*
⟨*proof*⟩

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

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'* ∈ *collect-interrupts ias* ⟧ ⟹ *t'* ∈ *is*
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *collect-interrupt-subset*: $Ts \subseteq Ts' \Longrightarrow collect\text{-}interrupt\ ia\ Ts \subseteq collect\text{-}interrupt\ ia\ Ts'$
⟨*proof*⟩

lemma *foldr-collect-interrupt-subset*:
$Ts \subseteq Ts' \Longrightarrow foldr\ collect\text{-}interrupt\ ias\ Ts \subseteq foldr\ collect\text{-}interrupt\ ias\ Ts'$
⟨*proof*⟩

lemma *interrupt-actions-ok-all-nthI*:
assumes $\bigwedge n.\ n < length\ ias \Longrightarrow interrupt\text{-}action\text{-}ok\ (redT\text{-}updIs\ is\ (take\ n\ ias))\ (ias\ !\ n)$
shows *interrupt-actions-ok is ias*
⟨*proof*⟩

lemma *interrupt-actions-ok-nthD*:
assumes *interrupt-actions-ok is ias*
and $n < length\ ias$
shows $interrupt\text{-}action\text{-}ok\ (redT\text{-}updIs\ is\ (take\ n\ ias))\ (ias\ !\ n)$
⟨*proof*⟩

lemma *interrupt-actions-ok'-all-nthI*:
assumes $\bigwedge n.\ n < length\ ias \Longrightarrow interrupt\text{-}action\text{-}ok'\ (redT\text{-}updIs\ is\ (take\ n\ ias))\ (ias\ !\ n)$
shows *interrupt-actions-ok' is ias*
⟨*proof*⟩

lemma *interrupt-actions-ok'-nthD*:
assumes *interrupt-actions-ok' is ias*
and $n < length\ ias$
shows $interrupt\text{-}action\text{-}ok'\ (redT\text{-}updIs\ is\ (take\ n\ ias))\ (ias\ !\ n)$
⟨*proof*⟩

lemma *interrupt-action-ok-imp-interrupt-action-ok'* [*simp*]:
$interrupt\text{-}action\text{-}ok\ is\ ia \Longrightarrow interrupt\text{-}action\text{-}ok'\ is\ ia$
⟨*proof*⟩

lemma *interrupt-actions-ok-imp-interrupt-actions-ok'* [*simp*]:
$interrupt\text{-}actions\text{-}ok\ is\ ias \Longrightarrow interrupt\text{-}actions\text{-}ok'\ is\ ias$
⟨*proof*⟩

lemma *collect-interruptsE*:
assumes $t' \in collect\text{-}interrupts\ ias'$
obtains n' **where** $n' < length\ ias'\ ias'\ !\ n' = IsInterrupted\ t'\ True$
and $Interrupt\ t' \notin set\ (take\ n'\ ias')$
⟨*proof*⟩

lemma *collect-interrupts-prefix*:
$collect\text{-}interrupts\ ias \subseteq collect\text{-}interrupts\ (ias\ @\ ias')$
⟨*proof*⟩

lemma *redT-updI-insert-Interrupt*:
$[\![\ t \in redT\text{-}updI\ is\ ia;\ t \notin is\ ]\!] \Longrightarrow ia = Interrupt\ t$
⟨*proof*⟩

lemma *redT-updIs-insert-Interrupt*:
$[\![\ t \in redT\text{-}updIs\ is\ ias;\ t \notin is\ ]\!] \Longrightarrow Interrupt\ t \in set\ ias$
⟨*proof*⟩

lemma *interrupt-actions-ok-takeI*:
interrupt-actions-ok is ias $\Longrightarrow$ *interrupt-actions-ok is* (*take n ias*)
⟨*proof*⟩

lemma *interrupt-actions-ok'-collect-interrupts-imp-interrupt-actions-ok*:
assumes *int*: *interrupt-actions-ok' is ias*
and *ci*: *collect-interrupts ias* $\subseteq$ *is*
and *int'*: *interrupt-actions-ok is' ias*
shows *interrupt-actions-ok is ias*
⟨*proof*⟩

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'*
$\Longrightarrow$ *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
$\bigwedge$*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*⦄$_i$ ⟧
$\Longrightarrow$ *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*) $\{\!|ta|\!\}_i$
⟨*proof*⟩

lemma *actions-ok-thread-oksD*:
actions-ok s t ta ⟹ *thread-oks* (*thr s*) $\{\!|ta|\!\}_t$
⟨*proof*⟩

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*) $\{\!|ta|\!\}_i$
⟨*proof*⟩

lemma *actions-ok'-ta-upd-obs*:
actions-ok' s t (*ta-update-obs ta obs*) ⟷ *actions-ok' s t ta*
⟨*proof*⟩

lemma *actions-ok'-empty*: *actions-ok' s t* ε ⟷ *wset s t* = *None*
⟨*proof*⟩

lemma *actions-ok'-convert-extTA*:
actions-ok' s t (*convert-extTA f ta*) = *actions-ok' s t ta*
⟨*proof*⟩

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* $\{\!|ta|\!\}_i$
⟨*proof*⟩

lemma *actions-subset-refl* [*intro*]:
actions-subset ta ta
⟨*proof*⟩

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*
⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma *final-threadE*:
assumes *final-thread s t*
obtains *x* **where** *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ *final x wset s t* = *None*
⟨*proof*⟩

lemma *mfinalI*:
(⋀*t x ln*. *thr s t* = ⌊(*x*, *ln*)⌋ ⟹ *final x* ∧ *ln* = *no-wait-locks* ∧ *wset s t* = *None*) ⟹ *mfinal s*
⟨*proof*⟩

lemma *mfinalD*:
fixes *ln*
assumes *mfinal s thr s t* = ⌊(*x*, *ln*)⌋
shows *final x ln* = *no-wait-locks wset s t* = *None*
⟨*proof*⟩

lemma *mfinalE*:
fixes *ln*
assumes *mfinal s thr s t* = ⌊(*x*, *ln*)⌋
obtains *final x ln* = *no-wait-locks wset s t* = *None*
⟨*proof*⟩

lemma *mfinal-def2*: *mfinal s* ⟷ *dom* (*thr s*) ⊆ *final-threads s*
⟨*proof*⟩

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*▷*ta*→ *s'*
and *normal*: ⋀*x x' m' ws'*.
⟦ *t* ⊢ ⟨*x*, *shr s*⟩ −*ta*→ ⟨*x'*, *m'*⟩;
thr s t = ⌊(*x*, *no-wait-locks*)⌋;
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* ↦ (*x'*, *redT-updLns* (*locks s*) *t* *no-wait-locks* $\{|ta|\}_l$)), *m'*), *ws'*, *redT-updIs* (*interrupts s*) $\{|ta|\}_i$) ⟧
⟹ *thesis*
and *acquire*: ⋀*x ln n*.
⟦ *thr s t* = ⌊(*x*, *ln*)⌋;
ta = (*K*$ [], [], [], [], [], *convert-RA ln*);
¬ *waiting* (*wset s t*);
may-acquire-all (*locks s*) *t ln*; *0* < *ln* $ *n*;

$s' = (acquire\text{-}all\ (locks\ s)\ t\ ln,\ ((thr\ s)(t \mapsto (x,\ no\text{-}wait\text{-}locks)),\ shr\ s),\ wset\ s,\ interrupts\ s)$ ⟧
⟹ *thesis*
shows *thesis*
⟨*proof*⟩

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' ⟹ *RedT s ttas s'*
⟨*proof*⟩

lemma *RedTE*:
⟦ *RedT s ttas s'*; *rtrancl3p redT s ttas s'* ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *RedTD*:
RedT s ttas s' ⟹ *rtrancl3p redT s ttas s'*
⟨*proof*⟩

lemma *RedT-induct* [*consumes 1*, *case-names refl step*]:
⟦ *s* −▷*ttas*→∗ *s'*;
⋀*s*. *P s* [] *s*;
⋀*s ttas s' t ta s''*. ⟦ *s* −▷*ttas*→∗ *s'*; *P s ttas s'*; *s'* −*t*▷*ta*→ *s''* ⟧ ⟹ *P s* (*ttas* @ [(*t*, *ta*)]) *s''*⟧
⟹ *P s ttas s'*
⟨*proof*⟩

lemma *RedT-induct'* [*consumes 1*, *case-names refl step*]:
⟦ *s* −▷*ttas*→∗ *s'*;
P s [] *s*;
⋀*ttas s' t ta s''*. ⟦ *s* −▷*ttas*→∗ *s'*; *P s ttas s'*; *s'* −*t*▷*ta*→ *s''* ⟧ ⟹ *P s* (*ttas* @ [(*t*, *ta*)]) *s''*⟧
⟹ *P s ttas s'*
⟨*proof*⟩

lemma *RedT-lift-preserveD*:
assumes *Red*: *s* −▷*ttas*→∗ *s'*
and *P*: *P s*
and *preserve*: ⋀*s t tas s'*. ⟦ *s* −*t*▷*tas*→ *s'*; *P s* ⟧ ⟹ *P s'*
shows *P s'*
⟨*proof*⟩

lemma *RedT-refl* [*intro*, *simp*]:
s −▷[]→∗ *s*
⟨*proof*⟩

lemma *redT-has-locks-inv*:
⟦ ⟨*ls*, (*ts*, *m*), *ws*, *is*⟩ −*t*▷*ta*→ ⟨*ls'*, (*ts'*, *m'*), *ws'*, *is'*⟩; *t* ≠ *t'* ⟧ ⟹
has-locks (*ls* $ *l*) *t'* = *has-locks* (*ls'* $ *l*) *t'*
⟨*proof*⟩

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

lemma *redT-ts-Some-inv*:
$[\![\langle ls, (ts, m), ws, is\rangle -t\triangleright ta\rightarrow \langle ls', (ts', m'), ws', is'\rangle; t \neq t'; ts\ t' = \lfloor x\rfloor]\!] \Longrightarrow ts'\ t' = \lfloor x\rfloor$
⟨*proof*⟩

lemma *redT-thread-not-disappear*:
$[\![s -t\triangleright ta\rightarrow s'; thr\ s'\ t' = None]\!] \Longrightarrow thr\ s\ t' = None$
⟨*proof*⟩

lemma *RedT-thread-not-disappear*:
$[\![s -\triangleright ttas\rightarrow* s'; thr\ s'\ t' = None]\!] \Longrightarrow thr\ s\ t' = None$
⟨*proof*⟩

lemma *redT-preserves-wset-thread-ok*:
$[\![s -t\triangleright ta\rightarrow s'; wset\text{-}thread\text{-}ok\ (wset\ s)\ (thr\ s)]\!] \Longrightarrow wset\text{-}thread\text{-}ok\ (wset\ s')\ (thr\ s')$
⟨*proof*⟩

lemma *RedT-preserves-wset-thread-ok*:
$[\![s -\triangleright ttas\rightarrow* s'; wset\text{-}thread\text{-}ok\ (wset\ s)\ (thr\ s)]\!] \Longrightarrow wset\text{-}thread\text{-}ok\ (wset\ s')\ (thr\ s')$
⟨*proof*⟩

lemma *redT-new-thread-ts-Some*:
$[\![s -t\triangleright ta\rightarrow s'; NewThread\ t'\ x\ m'' \in set\ \{\!|ta|\!\}_t; wset\text{-}thread\text{-}ok\ (wset\ s)\ (thr\ s)]\!]$
$\Longrightarrow thr\ s'\ t' = \lfloor(x, no\text{-}wait\text{-}locks)\rfloor$
⟨*proof*⟩

lemma *RedT-new-thread-ts-not-None*:
$[\![s -\triangleright ttas\rightarrow* s'; NewThread\ t\ x\ m'' \in set\ (concat\ (map\ (thr\text{-}a \circ snd)\ ttas)); wset\text{-}thread\text{-}ok\ (wset\ s)$
$(thr\ s)]\!]$
$\Longrightarrow thr\ s'\ t \neq None$
⟨*proof*⟩

lemma *redT-preserves-lock-thread-ok*:
$[\![s -t\triangleright ta\rightarrow s'; lock\text{-}thread\text{-}ok\ (locks\ s)\ (thr\ s)]\!] \Longrightarrow lock\text{-}thread\text{-}ok\ (locks\ s')\ (thr\ s')$
⟨*proof*⟩

lemma *RedT-preserves-lock-thread-ok*:
$[\![s -\triangleright ttas\rightarrow* s'; lock\text{-}thread\text{-}ok\ (locks\ s)\ (thr\ s)]\!] \Longrightarrow lock\text{-}thread\text{-}ok\ (locks\ s')\ (thr\ s')$
⟨*proof*⟩

lemma *redT-ex-new-thread*:
assumes $s -t'\triangleright ta\rightarrow s'\ wset\text{-}thread\text{-}ok\ (wset\ s)\ (thr\ s)\ thr\ s'\ t = \lfloor(x, w)\rfloor\ thr\ s\ t = None$
shows $\exists m.\ NewThread\ t\ x\ m \in set\ \{\!|ta|\!\}_t \wedge w = no\text{-}wait\text{-}locks$
⟨*proof*⟩

lemma *redT-ex-new-thread'*:
assumes $s -t'\triangleright ta\rightarrow s'\ thr\ s'\ t = \lfloor(x, w)\rfloor\ thr\ s\ t = None$
shows $\exists m\ x.\ NewThread\ t\ x\ m \in set\ \{\!|ta|\!\}_t$
⟨*proof*⟩

definition *deterministic* :: $('l,'t,'x,'m,'w)$ *state set* ⇒ *bool*
where
  deterministic I ⟷
  (∀ *s t x ta′ x′ m′ ta″ x″ m″*.
    s ∈ *I*
    ⟶ *thr s t* = ⌊(*x*, *no-wait-locks*)⌋
    ⟶ *t* ⊢ ⟨*x*, *shr s*⟩ −*ta′*→ ⟨*x′*, *m′*⟩
    ⟶ *t* ⊢ ⟨*x*, *shr s*⟩ −*ta″*→ ⟨*x″*, *m″*⟩
    ⟶ *actions-ok s t ta′* ⟶ *actions-ok s t ta″*
    ⟶ *ta′* = *ta″* ∧ *x′* = *x″* ∧ *m′* = *m″*) ∧ *invariant3p redT I*

lemma *determisticI*:
  ⟦⋀*s t x ta′ x′ m′ ta″ x″ m″*.
    ⟦ *s* ∈ *I*; *thr s t* = ⌊(*x*, *no-wait-locks*)⌋;
      t ⊢ ⟨*x*, *shr s*⟩ −*ta′*→ ⟨*x′*, *m′*⟩; *t* ⊢ ⟨*x*, *shr s*⟩ −*ta″*→ ⟨*x″*, *m″*⟩;
      actions-ok s t ta′; *actions-ok s t ta″* ⟧
    ⟹ *ta′* = *ta″* ∧ *x′* = *x″* ∧ *m′* = *m″*;
   invariant3p redT I ⟧
  ⟹ *deterministic I*
⟨*proof*⟩

lemma *deterministicD*:
  ⟦ *deterministic I*;
   t ⊢ ⟨*x*, *shr s*⟩ −*ta′*→ ⟨*x′*, *m′*⟩; *t* ⊢ ⟨*x*, *shr s*⟩ −*ta″*→ ⟨*x″*, *m″*⟩;
   thr s t = ⌊(*x*, *no-wait-locks*)⌋; *actions-ok s t ta′*; *actions-ok s t ta″*; *s* ∈ *I* ⟧
  ⟹ *ta′* = *ta″* ∧ *x′* = *x″* ∧ *m′* = *m″*
⟨*proof*⟩

lemma *deterministic-invariant3p*:
  deterministic I ⟹ *invariant3p redT I*
⟨*proof*⟩

lemma *deterministic-THE*:
  ⟦ *deterministic I*; *thr s t* = ⌊(*x*, *no-wait-locks*)⌋; *t* ⊢ ⟨*x*, *shr s*⟩ −*ta*→ ⟨*x′*, *m′*⟩; *actions-ok s t ta*; *s* ∈
I ⟧
  ⟹ (*THE* (*ta*, *x′*, *m′*). *t* ⊢ ⟨*x*, *shr s*⟩ −*ta*→ ⟨*x′*, *m′*⟩ ∧ *actions-ok s t ta*) = (*ta*, *x′*, *m′*)
⟨*proof*⟩

end

locale *multithreaded* = *multithreaded-base* +
  constrains *final* :: *'x* ⇒ *bool*
  and *r* :: $('l,'t,'x,'m,'w,'o)$ *semantics*
  and *convert-RA* :: *'l released-locks* ⇒ *'o list*
  assumes *new-thread-memory*: ⟦ *t* ⊢ *s* −*ta*→ *s′*; *NewThread t′ x m* ∈ *set* $\{\!|ta|\!\}_t$ ⟧ ⟹ *m* = *snd s′*
  and *final-no-red*: ⟦ *t* ⊢ (*x*, *m*) −*ta*→ (*x′*, *m′*); *final x* ⟧ ⟹ *False*
begin

lemma *redT-new-thread-common*:
  ⟦ *s* −*t*▷*ta*→ *s′*; *NewThread t′ x m″* ∈ *set* $\{\!|ta|\!\}_t$; $\{\!|ta|\!\}_w$ = [] ⟧ ⟹ *m″* = *shr s′*
⟨*proof*⟩

lemma *redT-new-thread*:
  assumes *s* −*t′*▷*ta*→ *s′* *thr s′ t* = ⌊(*x*, *w*)⌋ *thr s t* = *None* $\{\!|ta|\!\}_w$ = []

shows *NewThread t x (shr s′) ∈ set* $\{\!|ta|\!\}_t$ ∧ *w = no-wait-locks*
⟨*proof*⟩

lemma *final-no-redT*:
⟦ *s −t▷ta→ s′*; *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ ⟧ ⟹ ¬ *final x*
⟨*proof*⟩

lemma *mfinal-no-redT*:
assumes *redT*: *s −t▷ta→ s′* **and** *mfinal*: *mfinal s*
shows *False*
⟨*proof*⟩

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* $\{\!|ta|\!\}_i$
⟨*proof*⟩

context *multithreaded-base* **begin**

definition *must-sync* :: ′*t* ⇒ ′*x* ⇒ ′*m* ⇒ *bool* (‹- ⊢ ⟨-,/ -⟩/ ≀› [*50*, *0*,*0*] *81*) **where**
t ⊢ ⟨*x*, *m*⟩ ≀ ⟷ (∃ *ta x′ m′ s*. *t* ⊢ ⟨*x*, *m*⟩ −*ta*→ ⟨*x′*, *m′*⟩ ∧ *shr s* = *m* ∧ *actions-ok s t ta*)

lemma *must-sync-def2*:
t ⊢ ⟨*x*, *m*⟩ ≀ ⟷ (∃ *ta x′ m′ s*. *t* ⊢ ⟨*x*, *m*⟩ −*ta*→ ⟨*x′*, *m′*⟩ ∧ *actions-ok s t ta*)
⟨*proof*⟩

lemma *must-syncI*:
∃ *ta x′ m′ s*. *t* ⊢ ⟨*x*, *m*⟩ −*ta*→ ⟨*x′*, *m′*⟩ ∧ *actions-ok s t ta* ⟹ *t* ⊢ ⟨*x*, *m*⟩ ≀
⟨*proof*⟩

lemma *must-syncE*:
⟦ *t* ⊢ ⟨*x*, *m*⟩ ≀; ⋀*ta x′ m′ s*. ⟦ *t* ⊢ ⟨*x*, *m*⟩ −*ta*→ ⟨*x′*, *m′*⟩; *actions-ok s t ta*; *m* = *shr s* ⟧ ⟹ *thesis* ⟧ ⟹ *thesis*
⟨*proof*⟩

definition *can-sync* :: ′*t* ⇒ ′*x* ⇒ ′*m* ⇒ (′*l* + ′*t* + ′*t*) *set* ⇒ *bool* (‹- ⊢ ⟨-,/ -⟩/ -/ ≀› [*50*,*0*,*0*,*0*] *81*)
where

$t \vdash \langle x,\ m\rangle\ LT\ \wr \equiv \exists\, ta\ x'\ m'.\ t \vdash \langle x,\ m\rangle\ -ta\rightarrow \langle x',\ m'\rangle \wedge (LT = \textit{collect-waits}\ ta)$

lemma *can-syncI*:
$[\![\ t \vdash \langle x,\ m\rangle\ -ta\rightarrow \langle x',\ m'\rangle;$
$LT = \textit{collect-waits}\ ta\ ]\!]$
$\Longrightarrow t \vdash \langle x,\ m\rangle\ LT\ \wr$
⟨*proof*⟩

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 $LT = \textit{collect-waits}\ ta$
⟨*proof*⟩

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

lemma *active-threads-iff*:
active-threads s =
$\{t.\ \exists\, x\ ln.\ thr\ s\ t = Some\ (x,\ ln) \wedge$
$(\textit{if}\ ln = \textit{no-wait-locks}$
$\textit{then}\ \exists\, ta\ x'\ m'.\ t \vdash (x,\ shr\ s)\ -ta\rightarrow (x',\ m') \wedge \textit{actions-ok}\ s\ t\ ta$
$\textit{else}\ \neg\ waiting\ (wset\ s\ t) \wedge \textit{may-acquire-all}\ (locks\ s)\ t\ ln)\}$
⟨*proof*⟩

lemma *active-thread-ex-red*:
assumes $t \in \textit{active-threads}\ s$
shows $\exists\, ta\ s'.\ s\ -t\triangleright ta\rightarrow s'$
⟨*proof*⟩

end

Well-formedness conditions for final

context *final-thread* **begin**

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

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* = ⌊(*x*, *ln*)⌋;
¬ *final x* ⟹ *thesis*; *ln* ≠ *no-wait-locks* ⟹ *thesis*; $\bigwedge w.$ *wset s t* = ⌊*w*⌋ ⟹ *thesis* ⟧
⟹ *thesis*
⟨*proof*⟩

lemma *not-final-thread-iff*:
not-final-thread s t ⟷ (∃ *x ln*. *thr s t* = ⌊(*x*, *ln*)⌋ ∧ (¬ *final x* ∨ *ln* ≠ *no-wait-locks* ∨ (∃ *w*. *wset s t* = ⌊*w*⌋)))
⟨*proof*⟩

lemma *not-final-thread-conv*:
not-final-thread s t ⟷ *thr s t* ≠ *None* ∧ ¬ *final-thread s t*
⟨*proof*⟩

lemma *not-final-thread-existsE*:
assumes *not-final-thread s t*
and $\bigwedge x\ ln.$ *thr s t* = ⌊(*x*, *ln*)⌋ ⟹ *thesis*
shows *thesis*
⟨*proof*⟩

lemma *not-final-thread-final-thread-conv*:
thr s t ≠ *None* ⟹ ¬ *final-thread s t* ⟷ *not-final-thread s t*
⟨*proof*⟩

lemma *may-join-cond-action-oks*:
assumes $\bigwedge t'.$ *Join t'* ∈ *set cas* ⟹ ¬ *not-final-thread s t'* ∧ *t* ≠ *t'*
shows *cond-action-oks s t cas*
⟨*proof*⟩

end

context *multithreaded* **begin**

lemma *red-not-final-thread*:
s −*t*▹*ta*→ *s'* ⟹ *not-final-thread s t*
⟨*proof*⟩

lemma *redT-preserves-final-thread*:
⟦ *s* −*t'*▹*ta*→ *s'*; *final-thread s t* ⟧ ⟹ *final-thread s' t*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *wset-Suspend-okD1*:
s ∈ *wset-Suspend-ok I* ⟹ *s* ∈ *I*
⟨*proof*⟩

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″*)⌋
⟨*proof*⟩

lemma *wset-Suspend-ok-imp-wset-thread-ok*:
s ∈ *wset-Suspend-ok I* ⟹ *wset-thread-ok* (*wset s*) (*thr s*)
⟨*proof*⟩

lemma *invariant3p-wset-Suspend-ok*:
assumes *I*: *invariant3p redT I*
shows *invariant3p redT* (*wset-Suspend-ok I*)
⟨*proof*⟩

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*]:
$(\bigwedge x.\ x \in A \longrightarrow x \in B) \Longrightarrow$ *all-final-except ts A* $\longrightarrow$ *all-final-except ts B*
⟨*proof*⟩

lemma *all-final-except-mono′*:
$[\![$ *all-final-except ts A*; $\bigwedge x.\ x \in A \Longrightarrow x \in B\ ]\!] \Longrightarrow$ *all-final-except ts B*
⟨*proof*⟩

lemma *all-final-exceptI*:
$(\bigwedge t.$ *not-final-thread s t* $\Longrightarrow t \in Ts) \Longrightarrow$ *all-final-except s Ts*
⟨*proof*⟩

lemma *all-final-exceptD*:
$[\![$ *all-final-except s Ts*; *not-final-thread s t* $]\!] \Longrightarrow t \in Ts$
⟨*proof*⟩

inductive *must-wait* :: $('l,'t,'x,'m,'w)$ *state* $\Rightarrow\ 't \Rightarrow ('l + 't + 't) \Rightarrow\ 't$ *set* $\Rightarrow$ *bool*
for s :: $('l,'t,'x,'m,'w)$ *state* **and** t :: $'t$ **where**
— Lock l
$[\![$ *has-lock* (*locks s* \$ l) t'; $t' \neq t$; $t' \in Ts\ ]\!] \Longrightarrow$ *must-wait s t* (*Inl l*) *Ts*
| — Join t’
$[\![$ *not-final-thread s t′*; $t' \in Ts\ ]\!] \Longrightarrow$ *must-wait s t* (*Inr* (*Inl t′*)) *Ts*
| — IsInterrupted t’ True
$[\![$ *all-final-except s Ts*; $t' \notin$ *interrupts s* $]\!] \Longrightarrow$ *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) $t'\ t' \neq t\ t' \in Ts$
| t' **where** $lt =$ *Inr* (*Inl t′*) *not-final-thread s t′* $t' \in Ts$
| t' **where** $lt =$ *Inr* (*Inr t′*) *all-final-except s Ts* $t' \notin$ *interrupts s*
⟨*proof*⟩

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 $\longleftrightarrow$
(*case lt of Inl l* $\Rightarrow \exists t' \in Ts.\ t \neq t' \wedge$ *has-lock* (*locks s* \$ l) t'
| *Inr* (*Inl t′*) $\Rightarrow$ *not-final-thread s t′* $\wedge\ t' \in Ts$
| *Inr* (*Inr t′*) $\Rightarrow$ *all-final-except s Ts* $\wedge\ t' \notin$ *interrupts s*)
⟨*proof*⟩

end

Deadlock as a system-wide property

context *multithreaded-base* **begin**

definition
$deadlock :: ('l,'t,'x,'m,'w)\ state \Rightarrow bool$
where
$deadlock\ s$
$\equiv\ (\forall t\ x.\ thr\ s\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor \wedge \neg\ final\ x \wedge wset\ s\ t = None$
$\longrightarrow t \vdash \langle x,\ shr\ s\rangle \wr \wedge (\forall LT.\ t \vdash \langle x,\ shr\ s\rangle\ LT \wr \longrightarrow (\exists lt \in LT.\ must\text{-}wait\ s\ t\ lt\ (dom\ (thr\ s)))))$
$\wedge (\forall t\ x\ ln.\ thr\ s\ t = \lfloor(x,\ ln)\rfloor \wedge (\exists l.\ ln\ \$\ l > 0) \wedge \neg\ waiting\ (wset\ s\ t)$
$\longrightarrow (\exists l\ t'.\ ln\ \$\ l > 0 \wedge t \neq t' \wedge thr\ s\ t' \neq None \wedge has\text{-}lock\ (locks\ s\ \$\ l)\ t'))$
$\wedge (\forall t\ x\ w.\ thr\ s\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor \longrightarrow wset\ s\ t \neq \lfloor PostWS\ w\rfloor)$

lemma *deadlockI*:
$[\![\bigwedge t\ x.\ [\![thr\ s\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor;\ \neg\ final\ x;\ wset\ s\ t = None]\!]$
$\Longrightarrow t \vdash \langle x,\ shr\ s\rangle \wr \wedge (\forall LT.\ t \vdash \langle x,\ shr\ s\rangle\ LT \wr \longrightarrow (\exists lt \in LT.\ must\text{-}wait\ s\ t\ lt\ (dom\ (thr\ s))));$
$\bigwedge t\ x\ ln\ l.\ [\![thr\ s\ t = \lfloor(x,\ ln)\rfloor;\ ln\ \$\ l > 0;\ \neg\ waiting\ (wset\ s\ t)]\!]$
$\Longrightarrow \exists l\ t'.\ ln\ \$\ l > 0 \wedge t \neq t' \wedge thr\ s\ t' \neq None \wedge has\text{-}lock\ (locks\ s\ \$\ l)\ t';$
$\bigwedge t\ x\ w.\ thr\ s\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor \Longrightarrow wset\ s\ t \neq \lfloor PostWS\ w\rfloor]\!]$
$\Longrightarrow deadlock\ s$
⟨*proof*⟩

lemma *deadlockE*:
assumes $deadlock\ s$
obtains $\forall t\ x.\ thr\ s\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor \wedge \neg\ final\ x \wedge wset\ s\ t = None$
$\longrightarrow t \vdash \langle x,\ shr\ s\rangle \wr \wedge (\forall LT.\ t \vdash \langle x,\ shr\ s\rangle\ LT \wr \longrightarrow (\exists lt \in LT.\ must\text{-}wait\ s\ t\ lt\ (dom\ (thr\ s))))$
and $\forall t\ x\ ln.\ thr\ s\ t = \lfloor(x,\ ln)\rfloor \wedge (\exists l.\ ln\ \$\ l > 0) \wedge \neg\ waiting\ (wset\ s\ t)$
$\longrightarrow (\exists l\ t'.\ ln\ \$\ l > 0 \wedge t \neq t' \wedge thr\ s\ t' \neq None \wedge has\text{-}lock\ (locks\ s\ \$\ l)\ t')$
and $\forall t\ x\ w.\ thr\ s\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor \longrightarrow wset\ s\ t \neq \lfloor PostWS\ w\rfloor$
⟨*proof*⟩

lemma *deadlockD1*:
assumes $deadlock\ s$
and $thr\ s\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor$
and $\neg\ final\ x$
and $wset\ s\ t = None$
obtains $t \vdash \langle x,\ shr\ s\rangle \wr$
and $\forall LT.\ t \vdash \langle x,\ shr\ s\rangle\ LT \wr \longrightarrow (\exists lt \in LT.\ must\text{-}wait\ s\ t\ lt\ (dom\ (thr\ s)))$
⟨*proof*⟩

lemma *deadlockD2*:
fixes ln
assumes $deadlock\ s$
and $thr\ s\ t = \lfloor(x,\ ln)\rfloor$
and $ln\ \$\ l > 0$
and $\neg\ waiting\ (wset\ s\ t)$
obtains $l'\ t'$ **where** $ln\ \$\ l' > 0\ \ t \neq t'\ \ thr\ s\ t' \neq None\ \ has\text{-}lock\ (locks\ s\ \$\ l')\ t'$
⟨*proof*⟩

lemma *deadlockD3*:
assumes $deadlock\ s$
and $thr\ s\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor$
shows $\forall w.\ wset\ s\ t \neq \lfloor PostWS\ w\rfloor$
⟨*proof*⟩

lemma *deadlock-def2*:
$deadlock\ s \longleftrightarrow$

$(\forall t\ x.\ thr\ s\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor \wedge \neg\ final\ x \wedge wset\ s\ t = None$
$\longrightarrow t \vdash \langle x,\ shr\ s\rangle \wr \wedge (\forall LT.\ t \vdash \langle x,\ shr\ s\rangle\ LT \wr \longrightarrow (\exists lt \in LT.\ must\text{-}wait\ s\ t\ lt\ (dom\ (thr\ s)))))$
$\wedge (\forall t\ x\ ln.\ thr\ s\ t = \lfloor(x,\ ln)\rfloor \wedge ln \neq no\text{-}wait\text{-}locks \wedge \neg\ waiting\ (wset\ s\ t)$
$\longrightarrow (\exists l.\ ln\ \$\ l > 0 \wedge must\text{-}wait\ s\ t\ (Inl\ l)\ (dom\ (thr\ s))))$
$\wedge (\forall t\ x\ w.\ thr\ s\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor \longrightarrow wset\ s\ t \neq \lfloor PostWS\ WSNotified\rfloor \wedge wset\ s\ t \neq \lfloor PostWS\ WSWokenUp\rfloor)$
⟨*proof*⟩

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*⟩

lemma *mfinal-deadlock*:
mfinal s $\Longrightarrow$ *deadlock s*
⟨*proof*⟩

Now deadlock for single threads

lemma *must-wait-mono*:
$(\bigwedge x.\ x \in A \longrightarrow x \in B) \Longrightarrow must\text{-}wait\ s\ t\ lt\ A \longrightarrow must\text{-}wait\ s\ t\ lt\ B$
⟨*proof*⟩

lemma *must-wait-mono'*:
$[\![\ must\text{-}wait\ s\ t\ lt\ A;\ A \subseteq B\ ]\!] \Longrightarrow must\text{-}wait\ s\ t\ lt\ B$
⟨*proof*⟩

end

lemma *UN-mono*: $[\![\ x \in A \longrightarrow x \in A';\ x \in B \longrightarrow x \in B'\ ]\!] \Longrightarrow x \in A \cup B \longrightarrow x \in A' \cup B'$
⟨*proof*⟩

lemma *Collect-mono-conv* [*mono*]: $x \in \{x.\ P\ x\} \longleftrightarrow P\ x$
⟨*proof*⟩

context *multithreaded-base* **begin**

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

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

| *deadlockedAcquire*:
$\bigwedge ln.\ [\![\ thr\ s\ t = \lfloor(x,\ ln)\rfloor;\ \neg\ waiting\ (wset\ s\ t);\ ln\ \$\ l > 0;\ has\text{-}lock\ (locks\ s\ \$\ l)\ t';\ t' \neq t;$

$t' \in$ *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* = ⌊(*x*, *ln*)⌋; ¬ *waiting* (*wset s t*); *ln* $ *l* > *0*; *must-wait s t* (*Inl l*) (*deadlocked s* ∪ *final-threads s*) ⟧
$\Longrightarrow t \in$ *deadlocked s*
⟨*proof*⟩

lemma *deadlocked-elims* [*consumes 1*, *case-names lock wait acquire*]:
assumes $t \in$ *deadlocked s*
and *lock*: $\bigwedge$*x*. ⟦ *thr s t* = ⌊(*x*, *no-wait-locks*)⌋; *t* ⊢ ⟨*x*, *shr s*⟩ ≀; *wset s t* = *None*;
$\bigwedge$*LT*. *t* ⊢ ⟨*x*, *shr s*⟩ *LT* ≀ ⟹ ∃ *lt* ∈ *LT*. *must-wait s t lt* (*deadlocked s* ∪ *final-threads s*) ⟧
⟹ *thesis*
and *wait*: $\bigwedge$*x ln*. ⟦ *thr s t* = ⌊(*x*, *ln*)⌋; *all-final-except s* (*deadlocked s*); *waiting* (*wset s t*) ⟧
⟹ *thesis*
and *acquire*: $\bigwedge$*x ln l t'*.
⟦ *thr s t* = ⌊(*x*, *ln*)⌋; ¬ *waiting* (*wset s t*); *0* < *ln* $ *l*; *has-lock* (*locks s* $ *l*) *t'*; *t* ≠ *t'*;
t' ∈ *deadlocked s* ∨ *final-thread s t'* ⟧ ⟹ *thesis*
shows *thesis*
⟨*proof*⟩

lemma *deadlocked-coinduct*
[*consumes 1*, *case-names deadlocked*, *case-conclusion deadlocked Lock Wait Acquire*, *coinduct set*: *deadlocked*]:
assumes *major*: *t* ∈ *X*
and *step*:
$\bigwedge$*t*. *t* ∈ *X* ⟹
(∃ *x*. *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ ∧ *t* ⊢ ⟨*x*, *shr s*⟩ ≀ ∧ *wset s t* = *None* ∧
(∀ *LT*. *t* ⊢ ⟨*x*, *shr s*⟩ *LT* ≀ ⟶ (∃ *lt*∈*LT*. *must-wait s t lt* (*X* ∪ *deadlocked s* ∪ *final-threads s*))))
∨
(∃ *x ln*. *thr s t* = ⌊(*x*, *ln*)⌋ ∧ *all-final-except s* (*X* ∪ *deadlocked s*) ∧ *waiting* (*wset s t*)) ∨
(∃ *x l t' ln*. *thr s t* = ⌊(*x*, *ln*)⌋ ∧ ¬ *waiting* (*wset s t*) ∧ *0* < *ln* $ *l* ∧ *has-lock* (*locks s* $ *l*) *t'* ∧
t' ≠ *t* ∧ ((*t'* ∈ *X* ∨ *t'* ∈ *deadlocked s*) ∨ *final-thread s t'*))
shows *t* ∈ *deadlocked s*
⟨*proof*⟩

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

lemma *deadlocked'I*:
($\bigwedge$*t*. *not-final-thread s t* ⟹ *t* ∈ *deadlocked s*) ⟹ *deadlocked' s*
⟨*proof*⟩

lemma *deadlocked'D2*:
⟦ *deadlocked' s*; *not-final-thread s t*; *t* ∈ *deadlocked s* ⟹ *thesis* ⟧ ⟹ *thesis*
⟨*proof*⟩

lemma *not-deadlocked'I*:
⟦ *not-final-thread s t*; *t* ∉ *deadlocked s* ⟧ ⟹ ¬ *deadlocked' s*
⟨*proof*⟩

lemma *deadlocked'-intro*:

$\llbracket \forall t.\ \textit{not-final-thread}\ s\ t \longrightarrow t \in \textit{deadlocked}\ s\ \rrbracket \Longrightarrow \textit{deadlocked}'\ s$
⟨*proof*⟩

lemma *deadlocked-thread-exists*:
assumes $t \in \textit{deadlocked}\ s$
and $\bigwedge x\ ln.\ \textit{thr}\ s\ t = \lfloor(x,\ ln)\rfloor \Longrightarrow \textit{thesis}$
shows *thesis*
⟨*proof*⟩

end

context *multithreaded* **begin**

lemma *red-no-deadlock*:
assumes *P*: $s\ -t\triangleright ta\rightarrow\ s'$
and *dead*: $t \in \textit{deadlocked}\ s$
shows *False*
⟨*proof*⟩

lemma *deadlocked′-no-red*:
$\llbracket\ s\ -t\triangleright ta\rightarrow\ s';\ \textit{deadlocked}'\ s\ \rrbracket \Longrightarrow \textit{False}$
⟨*proof*⟩

lemma *not-final-thread-deadlocked-final-thread* [*iff*]:
$\textit{thr}\ s\ t = \lfloor \textit{xln} \rfloor \Longrightarrow \textit{not-final-thread}\ s\ t \lor t \in \textit{deadlocked}\ s \lor \textit{final-thread}\ s\ t$
⟨*proof*⟩

lemma *all-waiting-deadlocked*:
assumes *not-final-thread s t*
and *lock-thread-ok* (*locks s*) (*thr s*)
and *normal*: $\bigwedge t\ x.\ \llbracket\ \textit{thr}\ s\ t = \lfloor(x,\ \textit{no-wait-locks})\rfloor;\ \neg\ \textit{final}\ x;\ \textit{wset}\ s\ t = \textit{None}\ \rrbracket$
$\Longrightarrow t \vdash \langle x,\ \textit{shr}\ s\rangle \wr \land (\forall LT.\ t \vdash \langle x,\ \textit{shr}\ s\rangle\ LT \wr \longrightarrow (\exists lt{\in}LT.\ \textit{must-wait}\ s\ t\ lt\ (\textit{final-threads}$
$s)))$
and *acquire*: $\bigwedge t\ x\ ln\ l.\ \llbracket\ \textit{thr}\ s\ t = \lfloor(x,\ ln)\rfloor;\ \neg\ \textit{waiting}\ (\textit{wset}\ s\ t);\ ln\ \$\ l > 0\ \rrbracket$
$\Longrightarrow \exists l'.\ ln\ \$\ l' > 0 \land \neg\ \textit{may-lock}\ (\textit{locks}\ s\ \$\ l')\ t$
and *wakeup*: $\bigwedge t\ x\ w.\ \textit{thr}\ s\ t = \lfloor(x,\ \textit{no-wait-locks})\rfloor \Longrightarrow \textit{wset}\ s\ t \neq \lfloor \textit{PostWS}\ w\rfloor$
shows $t \in \textit{deadlocked}\ s$
⟨*proof*⟩

Equivalence proof for both notions of deadlock

lemma *deadlock-implies-deadlocked′*:
assumes *dead*: *deadlock s*
shows $\textit{deadlocked}'\ s$
⟨*proof*⟩

lemma *deadlocked′-implies-deadlock*:
assumes *dead*: $\textit{deadlocked}'\ s$
shows *deadlock s*
⟨*proof*⟩

lemma *deadlock-eq-deadlocked′*:
$\textit{deadlock} = \textit{deadlocked}'$
⟨*proof*⟩

lemma *deadlock-no-red*:
  ⟦ s −t▹ta→ s'; *deadlock* s ⟧ ⟹ *False*
⟨*proof*⟩

lemma *deadlock-no-active-threads*:
  assumes *dead*: *deadlock* s
  shows *active-threads* s = {}
⟨*proof*⟩

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 ∈ *wf-state*
  and *Red*: s −t▹ta→ s'
  shows *deadlocked* s ⊆ *deadlocked* s'
⟨*proof*⟩

corollary *RedT-deadlocked-subset*:
  assumes *wfs*: s ∈ *wf-state*
  and *Red*: s −▹*ttas*→∗ s'
  shows *deadlocked* s ⊆ *deadlocked* s'
⟨*proof*⟩

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* ⇒ *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 *wf-stateD*: $s \in$ *wf-state* ⟹ *lock-thread-ok* (*locks s*) (*thr s*) ∧ *wset-final-ok* (*wset s*) (*thr s*)
and *wf-red*:
⟦ $s \in$ *wf-state*; *thr s t* = ⌊(*x*, *no-wait-locks*)⌋;
t ⊢ (*x*, *shr s*) −*ta*→ (*x'*, *m'*); ¬ *waiting* (*wset s t*) ⟧
⟹ ∃ *ta' x' m'*. *t* ⊢ (*x*, *shr s*) −*ta'*→ (*x'*, *m'*) ∧ (*actions-ok s t ta'* ∨ *actions-ok' s t ta'* ∧ *actions-subset ta' ta*)

and *red-wait-set-not-final*:
⟦ $s \in$ *wf-state*; *thr s t* = ⌊(*x*, *no-wait-locks*)⌋;
t ⊢ (*x*, *shr s*) −*ta*→ (*x'*, *m'*); ¬ *waiting* (*wset s t*); *Suspend w* ∈ *set* $\{\!|ta|\!\}_w$ ⟧
⟹ ¬ *final x'*

and *wf-progress*:
⟦ $s \in$ *wf-state*; *thr s t* = ⌊(*x*, *no-wait-locks*)⌋; ¬ *final x* ⟧
⟹ ∃ *ta x' m'*. *t* ⊢ ⟨*x*, *shr s*⟩ −*ta*→ ⟨*x'*, *m'*⟩

and *ta-Wakeup-no-join-no-lock-no-interrupt*:
⟦ $s \in$ *wf-state*; *thr s t* = ⌊(*x*, *no-wait-locks*)⌋; *t* ⊢ *xm* −*ta*→ *xm'*; *Notified* ∈ *set* $\{\!|ta|\!\}_w$ ∨ *WokenUp* ∈ *set* $\{\!|ta|\!\}_w$ ⟧
⟹ *collect-waits ta* = {}

and *ta-satisfiable*:
⟦ $s \in$ *wf-state*; *thr s t* = ⌊(*x*, *no-wait-locks*)⌋; *t* ⊢ ⟨*x*, *shr s*⟩ −*ta*→ ⟨*x'*, *m'*⟩ ⟧
⟹ ∃ *s'*. *actions-ok s' t ta*
begin

lemma *wf-redE*:
assumes $s \in$ *wf-state thr s t* = ⌊(*x*, *no-wait-locks*)⌋
and *t* ⊢ ⟨*x*, *shr s*⟩ −*ta*→ ⟨*x''*, *m''*⟩ ¬ *waiting* (*wset s t*)
obtains *ta' x' m'*
where *t* ⊢ ⟨*x*, *shr s*⟩ −*ta'*→ ⟨*x'*, *m'*⟩ *actions-ok' s t ta' actions-subset ta' ta*
| *ta' x' m'* **where** *t* ⊢ ⟨*x*, *shr s*⟩ −*ta'*→ ⟨*x'*, *m'*⟩ *actions-ok s t ta'*
⟨*proof*⟩

lemma *wf-progressE*:
assumes $s \in$ *wf-state*
and *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ ¬ *final x*
obtains *ta x' m'* **where** *t* ⊢ ⟨*x*, *shr s*⟩ −*ta*→ ⟨*x'*, *m'*⟩
⟨*proof*⟩

lemma *wf-progress-satisfiable*:
⟦ $s \in$ *wf-state*; *thr s t* = ⌊(*x*, *no-wait-locks*)⌋; ¬ *final x* ⟧
⟹ ∃ *ta x' m' s'*. *t* ⊢ ⟨*x*, *shr s*⟩ −*ta*→ ⟨*x'*, *m'*⟩ ∧ *actions-ok s' t ta*
⟨*proof*⟩

theorem *redT-progress*:
assumes *wfs*: $s \in$ *wf-state*
and *ndead*: ¬ *deadlock s*
shows ∃ *t' ta' s'*. *s* −*t'*▹*ta'*→ *s'*
⟨*proof*⟩

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

lemma *ts-okI*:
$[\![\ \bigwedge t\ x\ ln.\ ts\ t = \lfloor(x,\ ln)\rfloor \Longrightarrow P\ t\ x\ m\ ]\!] \Longrightarrow ts\text{-}ok\ P\ ts\ m$
⟨*proof*⟩

lemma *ts-okE*:
$[\![\ ts\text{-}ok\ P\ ts\ m;\ [\![\ \bigwedge t\ x\ ln.\ ts\ t = \lfloor(x,\ ln)\rfloor \Longrightarrow P\ t\ x\ m\ ]\!] \Longrightarrow Q\ ]\!] \Longrightarrow Q$
⟨*proof*⟩

lemma *ts-okD*:
$\bigwedge ln.\ [\![\ ts\text{-}ok\ P\ ts\ m;\ ts\ t = \lfloor(x,\ ln)\rfloor\ ]\!] \Longrightarrow P\ t\ x\ m$
⟨*proof*⟩

lemma *ts-ok-True* [*simp*]:
$ts\text{-}ok\ (\lambda t\ m\ x.\ True)\ ts\ m$
⟨*proof*⟩

lemma *ts-ok-conj*:
$ts\text{-}ok\ (\lambda t\ x\ m.\ P\ t\ x\ m \wedge Q\ t\ x\ m) = (\lambda ts\ m.\ ts\text{-}ok\ P\ ts\ m \wedge ts\text{-}ok\ Q\ ts\ m)$
⟨*proof*⟩

lemma *ts-ok-mono*:
$[\![\ ts\text{-}ok\ P\ ts\ m;\ \bigwedge t\ x.\ P\ t\ x\ m \Longrightarrow Q\ t\ x\ m\ ]\!] \Longrightarrow ts\text{-}ok\ Q\ ts\ m$
⟨*proof*⟩

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

definition
$ts\text{-}inv :: ('i \Rightarrow 't \Rightarrow 'x \Rightarrow 'm \Rightarrow bool) \Rightarrow ('t \rightharpoonup 'i) \Rightarrow ('l,'t,'x)\ thread\text{-}info \Rightarrow 'm \Rightarrow bool$
where
$\bigwedge ln.\ ts\text{-}inv\ P\ I\ ts\ m \equiv \forall t.\ case\ (ts\ t)\ of\ None \Rightarrow True \mid \lfloor(x,\ ln)\rfloor \Rightarrow \exists i.\ I\ t = \lfloor i\rfloor \wedge P\ i\ t\ x\ m$

lemma *ts-invI*:
$[\![\ \bigwedge t\ x\ ln.\ ts\ t = \lfloor(x,\ ln)\rfloor \Longrightarrow \exists i.\ I\ t = \lfloor i\rfloor \wedge P\ i\ t\ x\ m\ ]\!] \Longrightarrow ts\text{-}inv\ P\ I\ ts\ m$

⟨*proof*⟩

lemma *ts-invE*:
$[\![$ *ts-inv P I ts m*; $\forall$ *t x ln. ts t* = $\lfloor$(*x*, *ln*)$\rfloor$ $\longrightarrow$ ($\exists$ *i. I t* = $\lfloor i \rfloor$ $\wedge$ *P i t x m*) $\Longrightarrow$ *R* $]\!]$ $\Longrightarrow$ *R*
⟨*proof*⟩

lemma *ts-invD*:
$\bigwedge$*ln.* $[\![$ *ts-inv P I ts m*; *ts t* = $\lfloor$(*x*, *ln*)$\rfloor$ $]\!]$ $\Longrightarrow$ $\exists$ *i. I t* = $\lfloor i \rfloor$ $\wedge$ *P i t x m*
⟨*proof*⟩

Wellformedness properties for lifting

definition
ts-inv-ok :: ('*l*,'*t*,'*x*) *thread-info* $\Rightarrow$ ('*t* $\rightharpoonup$ '*i*) $\Rightarrow$ *bool*
where
ts-inv-ok ts I $\equiv$ $\forall$ *t. ts t* = *None* $\longleftrightarrow$ *I t* = *None*

lemma *ts-inv-okI*:
($\bigwedge$*t. ts t* = *None* $\longleftrightarrow$ *I t* = *None*) $\Longrightarrow$ *ts-inv-ok ts I*
⟨*proof*⟩

lemma *ts-inv-okI2*:
($\bigwedge$*t.* ($\exists$ *v. ts t* = $\lfloor v \rfloor$) $\longleftrightarrow$ ($\exists$ *v. I t* = $\lfloor v \rfloor$)) $\Longrightarrow$ *ts-inv-ok ts I*
⟨*proof*⟩

lemma *ts-inv-okE*:
$[\![$ *ts-inv-ok ts I*; $\forall$ *t. ts t* = *None* $\longleftrightarrow$ *I t* = *None* $\Longrightarrow$ *P* $]\!]$ $\Longrightarrow$ *P*
⟨*proof*⟩

lemma *ts-inv-okE2*:
$[\![$ *ts-inv-ok ts I*; $\forall$ *t.* ($\exists$ *v. ts t* = $\lfloor v \rfloor$) $\longleftrightarrow$ ($\exists$ *v. I t* = $\lfloor v \rfloor$) $\Longrightarrow$ *P* $]\!]$ $\Longrightarrow$ *P*
⟨*proof*⟩

lemma *ts-inv-okD*:
ts-inv-ok ts I $\Longrightarrow$ (*ts t* = *None*) $\longleftrightarrow$ (*I t* = *None*)
⟨*proof*⟩

lemma *ts-inv-okD2*:
ts-inv-ok ts I $\Longrightarrow$ ($\exists$ *v. ts t* = $\lfloor v \rfloor$) $\longleftrightarrow$ ($\exists$ *v. I t* = $\lfloor v \rfloor$)
⟨*proof*⟩

lemma *ts-inv-ok-conv-dom-eq*:
ts-inv-ok ts I $\longleftrightarrow$ (*dom ts* = *dom I*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

fun *upd-inv* :: $('t \rightharpoonup 'i) \Rightarrow ('i \Rightarrow 't \Rightarrow 'x \Rightarrow 'm \Rightarrow bool) \Rightarrow ('t,'x,'m)$ *new-thread-action* $\Rightarrow ('t \rightharpoonup 'i)$
where
  upd-inv I P (*NewThread t x m*) = *I*(*t* ↦ *SOME i*. *P i t x m*)
| *upd-inv I P* - = *I*

fun *upd-invs* :: $('t \rightharpoonup 'i) \Rightarrow ('i \Rightarrow 't \Rightarrow 'x \Rightarrow 'm \Rightarrow bool) \Rightarrow ('t,'x,'m)$ *new-thread-action list* $\Rightarrow ('t \rightharpoonup 'i)$
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*
⟨*proof*⟩

lemma *ts-inv-ok-upd-inv'*:
 ts-inv-ok ts I ⟹ *ts-inv-ok* (*redT-updT' ts ta*) (*upd-inv I P ta*)
⟨*proof*⟩

lemma *ts-inv-ok-upd-invs'*:
  ts-inv-ok ts I ⟹ *ts-inv-ok* (*redT-updTs' ts tas*) (*upd-invs I P tas*)
⟨*proof*⟩

lemma *ts-inv-ok-upd-inv*:
 ts-inv-ok ts I ⟹ *ts-inv-ok* (*redT-updT ts ta*) (*upd-inv I P ta*)
⟨*proof*⟩

lemma *ts-inv-ok-upd-invs*:
  ts-inv-ok ts I ⟹ *ts-inv-ok* (*redT-updTs ts tas*) (*upd-invs I P tas*)
⟨*proof*⟩

lemma *ts-inv-ok-inv-ext-upd-inv*:
  ⟦ *ts-inv-ok ts I*; *thread-ok ts ta* ⟧ ⟹ $I \subseteq_m$ *upd-inv I P ta*
⟨*proof*⟩

lemma *ts-inv-ok-inv-ext-upd-invs*:
  ⟦ *ts-inv-ok ts I*; *thread-oks ts tas*⟧
  ⟹ $I \subseteq_m$ *upd-invs I P tas*
⟨*proof*⟩

lemma *upd-invs-Some*:
  ⟦ *thread-oks ts tas*; *I t* = ⌊*i*⌋; *ts t* = ⌊*x*⌋ ⟧ ⟹ *upd-invs I Q tas t* = ⌊*i*⌋
⟨*proof*⟩

lemma *upd-inv-Some-eq*:
  ⟦ *thread-ok ts ta*; *ts t* = ⌊*x*⌋ ⟧ ⟹ *upd-inv I Q ta t* = *I t*
⟨*proof*⟩

lemma *upd-invs-Some-eq*: ⟦ *thread-oks ts tas*; *ts t* = ⌊*x*⌋ ⟧ ⟹ *upd-invs I Q tas t* = *I t*
⟨*proof*⟩

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 $\exists\, i.\ \mathit{upd\text{-}invs}\ I\ Q\ \mathit{tas}\ t = \lfloor i \rfloor \wedge Q\ i\ t\ x\ m$
⟨*proof*⟩

lemma *ts-ok-into-ts-inv-const*:
assumes *ts-ok P ts m*
obtains *I* **where** $\mathit{ts\text{-}inv}\ (\lambda\text{-}.\ P)\ I\ \mathit{ts}\ m$
⟨*proof*⟩

lemma *ts-inv-const-into-ts-ok*:
$\mathit{ts\text{-}inv}\ (\lambda\text{-}.\ P)\ I\ \mathit{ts}\ m \Longrightarrow \mathit{ts\text{-}ok}\ P\ \mathit{ts}\ m$
⟨*proof*⟩

lemma *ts-inv-into-ts-ok-Ex*:
$\mathit{ts\text{-}inv}\ Q\ I\ \mathit{ts}\ m \Longrightarrow \mathit{ts\text{-}ok}\ (\lambda t\ x\ m.\ \exists\, i.\ Q\ i\ t\ x\ m)\ \mathit{ts}\ m$
⟨*proof*⟩

lemma *ts-ok-Ex-into-ts-inv*:
$\mathit{ts\text{-}ok}\ (\lambda t\ x\ m.\ \exists\, i.\ Q\ i\ t\ x\ m)\ \mathit{ts}\ m \Longrightarrow \exists\, I.\ \mathit{ts\text{-}inv}\ Q\ I\ \mathit{ts}\ m$
⟨*proof*⟩

lemma *Ex-ts-inv-conv-ts-ok*:
$(\exists\, I.\ \mathit{ts\text{-}inv}\ Q\ I\ \mathit{ts}\ m) \longleftrightarrow (\mathit{ts\text{-}ok}\ (\lambda t\ x\ m.\ \exists\, i.\ Q\ i\ t\ x\ m)\ \mathit{ts}\ m)$
⟨*proof*⟩

end

1.15 Labelled transition systems

theory *LTS*
imports
../Basic/Auxiliary
Coinductive.TLList
begin

unbundle *no floor-ceiling-syntax*

lemma *rel-option-mono*:
$[\![\ \mathit{rel\text{-}option}\ R\ x\ y;\ \bigwedge x\ y.\ R\ x\ y \Longrightarrow R'\ x\ y\ ]\!] \Longrightarrow \mathit{rel\text{-}option}\ R'\ x\ y$
⟨*proof*⟩

lemma *nth-concat-conv*:
$n < \mathit{length}\ (\mathit{concat}\ \mathit{xss})$
$\Longrightarrow \exists\, m\ n'.\ \mathit{concat}\ \mathit{xss}\ !\ n = (\mathit{xss}\ !\ m)\ !\ n' \wedge n' < \mathit{length}\ (\mathit{xss}\ !\ m) \wedge$
$m < \mathit{length}\ \mathit{xss} \wedge n = (\sum i{<}m.\ \mathit{length}\ (\mathit{xss}\ !\ i)) + n'$
⟨*proof*⟩

definition *flip* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b \Rightarrow 'a \Rightarrow 'c$
where $\mathit{flip}\ f = (\lambda b\ a.\ f\ a\ b)$

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

⟨*ML*⟩

lemma *flip-conv* [*flip-simps*]: *flip f b a = f a b*
⟨*proof*⟩

lemma *flip-flip* [*flip-simps*, *simp*]: *flip* (*flip f*) = *f*
⟨*proof*⟩

lemma *list-all2-flip* [*flip-simps*]: *list-all2* (*flip P*) *xs ys* = *list-all2 P ys xs*
⟨*proof*⟩

lemma *llist-all2-flip* [*flip-simps*]: *llist-all2* (*flip P*) *xs ys* = *llist-all2 P ys xs*
⟨*proof*⟩

lemma *rtranclp-flipD*:
  assumes (*flip r*)^** *x y*
  shows *r*^** *y x*
⟨*proof*⟩

lemma *rtranclp-flip* [*flip-simps*]:
  (*flip r*)^** = *flip r*^**
⟨*proof*⟩

lemma *rel-prod-flip* [*flip-simps*]:
  rel-prod (*flip R*) (*flip S*) = *flip* (*rel-prod R S*)
⟨*proof*⟩

lemma *rel-option-flip* [*flip-simps*]:
  rel-option (*flip R*) = *flip* (*rel-option R*)
⟨*proof*⟩

lemma *tllist-all2-flip* [*flip-simps*]:
  tllist-all2 (*flip P*) (*flip Q*) *xs ys* ⟷ *tllist-all2 P Q ys xs*
⟨*proof*⟩

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'* $\wedge$ *s'* −*ltl tls*→∗ ∞))
(λ(*s*, *tls*). (*SOME s'*. *trsys s* (*lhd tls*) *s'* $\wedge$ *s'* −*ltl tls*→∗ ∞, *ltl tls*))
(*s*, *tls*)

coinductive *Rtrancl3p* :: *'s* ⇒ (*'tl*, *'s*) *tllist* ⇒ *bool*
where
Rtrancl3p-stop: ($\bigwedge$*tl s'*. ¬ *s* −*tl*→ *s'*) ⟹ *Rtrancl3p s* (*TNil s*)
| *Rtrancl3p-into-Rtrancl3p*: $\bigwedge$*tl*. ⟦ *s* −*tl*→ *s'*; *Rtrancl3p s' tlss* ⟧ ⟹ *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'*. ¬ *s* −*tl*→ *s'*) ⟹ *Runs s LNil*
| *Step*: $\bigwedge$*tl*. ⟦ *s* −*tl*→ *s'*; *Runs s' tls* ⟧ ⟹ *Runs s* (*LCons tl tls*)

coinductive *Runs-table* :: *'s* ⇒ (*'s* × *'tl* × *'s*) *llist* ⇒ *bool*
where
Stuck: ($\bigwedge$*tl s'*. ¬ *s* −*tl*→ *s'*) ⟹ *Runs-table s LNil*
| *Step*: $\bigwedge$*tl*. ⟦ *s* −*tl*→ *s'*; *Runs-table s' stlss* ⟧ ⟹ *Runs-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*→∗ ∞
shows ¬ *lfinite bs*
⟨*proof*⟩

lemma *inf-step2inf-step-table-LNil* [*simp*]: *inf-step2inf-step-table s LNil* = *LNil*
⟨*proof*⟩

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'* $\wedge$ *s'* −*tls*→∗ ∞)
(*inf-step2inf-step-table* (*SOME s'*. *trsys s tl s'* $\wedge$ *s'* −*tls*→∗ ∞) *tls*)
⟨*proof*⟩

lemma *lnull-inf-step2inf-step-table* [*simp*]:
lnull (*inf-step2inf-step-table s tls*) ⟷ *lnull tls*
⟨*proof*⟩

lemma *inf-step2inf-step-table-eq-LNil*:
inf-step2inf-step-table s tls = *LNil* ⟷ *tls* = *LNil*
⟨*proof*⟩

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*→∗ ∞)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *lmap-inf-step2inf-step-table*: *lmap* (*fst* ∘ *snd*) (*inf-step2inf-step-table s tls*) = *tls*
⟨*proof*⟩

lemma *inf-step-imp-inf-step-table*:
assumes *s* −*tls*→∗ ∞
shows ∃ *stls*. *s* −*stls*→∗*t* ∞ ∧ *tls* = *lmap* (*fst* ∘ *snd*) *stls*
⟨*proof*⟩

lemma *inf-step-table-imp-inf-step*:
s−*stls*→∗*t* ∞ ⟹*s* −*lmap* (*fst* ∘ *snd*) *stls*→∗ ∞
⟨*proof*⟩

lemma *Runs-table-into-Runs*:
Runs-table s stlss ⟹ *Runs s* (*lmap* (λ(*s*, *tl*, *s*′). *tl*) *stlss*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *Trsys-into-Runs*:
assumes *s* −*tls*→∗ *s*′
and *Runs s*′ *tls*′
shows *Runs s* (*lappend* (*llist-of tls*) *tls*′)
⟨*proof*⟩

lemma *rtrancl3p-into-Rtrancl3p*:
⟦ *rtrancl3p trsys a bs a*′; ⋀*b a*′′. ¬ *a*′ −*b*→ *a*′′ ⟧ ⟹ *Rtrancl3p a* (*tllist-of-llist a*′ (*llist-of bs*))
⟨*proof*⟩

lemma *Rtrancl3p-into-Runs*:
Rtrancl3p s tlss $\Longrightarrow$ *Runs s* (*llist-of-tllist tlss*)
⟨*proof*⟩

lemma *Runs-into-Rtrancl3p*:
assumes *Runs s tls*
obtains *tlss* **where** *tls* = *llist-of-tllist tlss Rtrancl3p s tlss*
⟨*proof*⟩

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′*
⟨*proof*⟩

end

1.15.2 Labelled transition systems with internal actions

locale *τtrsys* = *trsys* +
constrains *trsys* :: (*′s*, *′tl*) *trsys*
fixes *τmove* :: (*′s*, *′tl*) *trsys*
begin

inductive *silent-move* :: *′s* ⇒ *′s* ⇒ *bool* (‹- −τ→ -› [*50*, *50*] *60*)
where [*intro*]: !!*tl*. ⟦ *trsys s tl s′*; *τmove s tl s′* ⟧ $\Longrightarrow$ *s* −τ→ *s′*

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

lemma *silent-move-iff*: *silent-move* = (λ*s s′*. (∃ *tl*. *trsys s tl s′* ∧ *τmove s tl s′*))
⟨*proof*⟩

abbreviation *silent-moves* :: *′s* ⇒ *′s* ⇒ *bool* (‹- −τ→∗ -› [*50*, *50*] *60*)
where *silent-moves* == *silent-move*^∗∗

abbreviation *silent-movet* :: *′s* ⇒ *′s* ⇒ *bool* (‹- −τ→+ -› [*50*, *50*] *60*)
where *silent-movet* == *silent-move*^++

coinductive *τdiverge* :: *′s* ⇒ *bool* (‹- −τ→ ∞› [*50*] *60*)
where
τdivergeI: ⟦ *s* −τ→ *s′*; *s′* −τ→ ∞ ⟧ $\Longrightarrow$ *s* −τ→ ∞

coinductive *τinf-step* :: *′s* ⇒ *′tl llist* ⇒ *bool* (‹- −τ−-→∗ ∞› [*50*, *0*] *60*)
where
τinf-step-Cons: ⋀*tl*. ⟦ *s* −τ→∗ *s′*; *s′* −*tl*→ *s″*; ¬ *τmove s′ tl s″*; *s″* −τ−*tls*→∗ ∞ ⟧ $\Longrightarrow$ *s* −τ−*LCons tl tls*→∗ ∞
| *τinf-step-Nil*: *s* −τ→ ∞ $\Longrightarrow$ *s* −τ−*LNil*→∗ ∞

coinductive *τinf-step-table* :: *′s* ⇒ (*′s* × *′s* × *′tl* × *′s*) *llist* ⇒ *bool* (‹- −τ−-→∗*t* ∞› [*50*, *0*] *80*)
where
τinf-step-table-Cons:
⋀*tl*. ⟦ *s* −τ→∗ *s′*; *s′* −*tl*→ *s″*; ¬ *τmove s′ tl s″*; *s″* −τ−*tls*→∗*t* ∞ ⟧ $\Longrightarrow$ *s* −τ−*LCons* (*s*, *s′*, *tl*, *s″*) *tls*→∗*t* ∞

| *τinf-step-table-Nil*:
s −τ→ ∞ ⟹ s −τ−LNil→∗t ∞

definition *τinf-step2τinf-step-table* :: *′s ⇒ ′tl llist ⇒ (′s × ′s × ′tl × ′s) llist*
where
τinf-step2τinf-step-table s tls =
unfold-llist
(λ(s, tls). lnull tls)
(λ(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′′))
(λ(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′′, ltl tls))
(s, tls)

definition *silent-move-from* :: *′s ⇒ ′s ⇒ ′s ⇒ bool*
where *silent-move-from s0 s1 s2 ⟷ silent-moves s0 s1 ∧ silent-move s1 s2*

inductive *τrtrancl3p* :: *′s ⇒ ′tl list ⇒ ′s ⇒ bool* (‹- −τ−-→∗ -› [50, 0, 50] 60)
where
τrtrancl3p-refl: *τrtrancl3p s [] s*
| *τrtrancl3p-step*: ⋀*tl*. ⟦ *s −tl→ s′*; ¬ *τmove s tl s′*; *τrtrancl3p s′ tls s′′* ⟧ ⟹ *τrtrancl3p s (tl # tls) s′′*
| *τrtrancl3p-τstep*: ⋀*tl*. ⟦ *s −tl→ s′*; *τmove s tl s′*; *τrtrancl3p s′ tls s′′* ⟧ ⟹ *τrtrancl3p s tls s′′*

coinductive *τRuns* :: *′s ⇒ (′tl, ′s option) tllist ⇒ bool* (‹- ⇓ -› [50, 50] 51)
where
Terminate: ⟦ *s −τ→∗ s′*; ⋀*tl s′′*. ¬ *s′ −tl→ s′′* ⟧ ⟹ *s ⇓ TNil ⌊s′⌋*
| *Diverge*: *s −τ→ ∞ ⟹ s ⇓ TNil None*
| *Proceed*: ⋀*tl*. ⟦ *s −τ→∗ s′*; *s′ −tl→ s′′*; ¬ *τmove s′ tl s′′*; *s′′ ⇓ tls* ⟧ ⟹ *s ⇓ TCons tl tls*

inductive-simps *τRuns-simps*:
s ⇓ TNil (Some s′)
s ⇓ TNil None
s ⇓ TCons tl′ tls

coinductive *τRuns-table* :: *′s ⇒ (′tl × ′s, ′s option) tllist ⇒ bool*
where
Terminate: ⟦ *s −τ→∗ s′*; ⋀*tl s′′*. ¬ *s′ −tl→ s′′* ⟧ ⟹ *τRuns-table s (TNil ⌊s′⌋)*
| *Diverge*: *s −τ→ ∞ ⟹ τRuns-table s (TNil None)*
| *Proceed*:
⋀*tl*. ⟦ *s −τ→∗ s′*; *s′ −tl→ s′′*; ¬ *τmove s′ tl s′′*; *τRuns-table s′′ tls* ⟧
⟹ *τRuns-table s (TCons (tl, s′′) tls)*

definition *silent-move2* :: *′s ⇒ ′tl ⇒ ′s ⇒ bool*
where ⋀*tl*. *silent-move2 s tl s′ ⟷ s −tl→ s′ ∧ τmove s tl s′*

abbreviation *silent-moves2* :: *′s ⇒ ′tl list ⇒ ′s ⇒ bool*
where *silent-moves2 ≡ rtrancl3p silent-move2*

coinductive *τRuns-table2* :: *′s ⇒ (′tl list × ′s × ′tl × ′s, (′tl list × ′s) + ′tl llist) tllist ⇒ bool*
where
Terminate: ⟦ *silent-moves2 s tls s′*; ⋀*tl s′′*. ¬ *s′ −tl→ s′′* ⟧ ⟹ *τRuns-table2 s (TNil (Inl (tls, s′)))*

| *Diverge*: *trsys.inf-step silent-move2 s tls* $\Longrightarrow$ *τRuns-table2 s* (*TNil* (*Inr tls*))
| *Proceed*:
$\bigwedge$*tl*. ⟦ *silent-moves2 s tls s′*; *s′ −tl→ s″*; ¬ *τmove s′ tl s″*; *τRuns-table2 s″ tlsstlss* ⟧
$\Longrightarrow$ *τRuns-table2 s* (*TCons* (*tls*, *s′*, *tl*, *s″*) *tlsstlss*)

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

lemma *inf-step-table-all-τ-into-τdiverge*:
⟦ *s −stls→∗t* ∞; ∀ (*s*, *tl*, *s′*) ∈ *lset stls*. *τmove s tl s′* ⟧ $\Longrightarrow$ *s −τ→* ∞
⟨*proof*⟩

lemma *inf-step-table-lappend-llist-ofD*:
s −lappend (*llist-of stls*) (*LCons* (*x*, *tl′*, *x′*) *xs*)*→∗t* ∞
$\Longrightarrow$ (*s −map* (*fst* ∘ *snd*) *stls→∗ x*) ∧ (*x −LCons* (*x*, *tl′*, *x′*) *xs→∗t* ∞)
⟨*proof*⟩

lemma *inf-step-table-lappend-llist-of-τ-into-τmoves*:
assumes *lfinite stls*
shows ⟦ *s −lappend stls* (*LCons* (*x*, *tl′ x′*) *xs*)*→∗t* ∞; ∀ (*s*, *tl*, *s′*)∈*lset stls*. *τmove s tl s′* ⟧ $\Longrightarrow$ *s*
−τ→∗ x
⟨*proof*⟩

lemma *inf-step-table-into-τinf-step*:
s −stls→∗t ∞ $\Longrightarrow$ *s −τ−lmap* (*fst* ∘ *snd*) (*lfilter* (λ(*s*, *tl*, *s′*). ¬ *τmove s tl s′*) *stls*)*→∗* ∞
⟨*proof*⟩

lemma *inf-step-into-τinf-step*:
assumes *s −tls→∗* ∞
shows ∃ *A*. *s −τ−lnths tls A→∗* ∞
⟨*proof*⟩

lemma *silent-moves-into-τrtrancl3p*:
s −τ→∗ s′ $\Longrightarrow$ *s −τ−*[]*→∗ s′*
⟨*proof*⟩

lemma *τrtrancl3p-into-silent-moves*:
s −τ−[]*→∗ s′* $\Longrightarrow$ *s −τ→∗ s′*
⟨*proof*⟩

lemma *τrtrancl3p-Nil-eq-τmoves*:
s −τ−[]*→∗ s′* ⟷ *s −τ→∗ s′*
⟨*proof*⟩

lemma *τrtrancl3p-trans* [*trans*]:
⟦ *s −τ−tls→∗ s′*; *s′ −τ−tls′→∗ s″* ⟧ $\Longrightarrow$ *s −τ−tls* @ *tls′→∗ s″*
⟨*proof*⟩

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*⟩

lemma *τrtrancl3p-snocI*:
$\bigwedge tl.$ ⟦ *τrtrancl3p s tls s''*; *s'' −τ→∗ s'''*; *s''' −tl→ s'*; ¬ *τmove s''' tl s'* ⟧
⟹ *τrtrancl3p s* (*tls* @ [*tl*]) *s'*
⟨*proof*⟩

lemma *τdiverge-rtranclp-silent-move*:
⟦ *silent-move^∗∗ s s'*; *s' −τ→ ∞* ⟧ ⟹ *s −τ→ ∞*
⟨*proof*⟩

lemma *τdiverge-trancl-coinduct* [*consumes 1*, *case-names τdiverge*]:
assumes *X*: *X s*
and *step*: $\bigwedge s.$ *X s* ⟹ ∃ *s'*. *silent-move^++ s s'* ∧ (*X s'* ∨ *s' −τ→ ∞*)
shows *s −τ→ ∞*
⟨*proof*⟩

lemma *τdiverge-trancl-measure-coinduct* [*consumes 2*, *case-names τdiverge*]:
assumes *major*: *X s t wfP μ*
and *step*: $\bigwedge s\ t.$ *X s t* ⟹ ∃ *s' t'*. (*μ t' t* ∧ *s'* = *s* ∨ *silent-move^++ s s'*) ∧ (*X s' t'* ∨ *s' −τ→ ∞*)
shows *s −τ→ ∞*
⟨*proof*⟩

lemma *τinf-step2τinf-step-table-LNil* [*simp*]: *τinf-step2τinf-step-table s LNil* = *LNil*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *lnull-τinf-step2τinf-step-table* [*simp*]:
lnull (*τinf-step2τinf-step-table s tls*) ⟷ *lnull tls*
⟨*proof*⟩

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''*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *lmap-τinf-step2τinf-step-table*: *lmap* (*fst* ∘ *snd* ∘ *snd*) (*τinf-step2τinf-step-table s tls*) = *tls*
⟨*proof*⟩

lemma *τinf-step-into-τinf-step-table*:
  $s\ -\tau-tls\rightarrow*\ \infty \Longrightarrow s\ -\tau-\tau inf\text{-}step2\tau inf\text{-}step\text{-}table\ s\ tls\rightarrow*t\ \infty$
⟨*proof*⟩

lemma *τinf-step-imp-τinf-step-table*:
  assumes $s\ -\tau-tls\rightarrow*\ \infty$
  shows $\exists\, sstls.\ s\ -\tau-sstls\rightarrow*t\ \infty \wedge tls = lmap\ (fst \circ snd \circ snd)\ sstls$
⟨*proof*⟩

lemma *τinf-step-table-into-τinf-step*:
  $s\ -\tau-sstls\rightarrow*t\ \infty \Longrightarrow s\ -\tau-lmap\ (fst \circ snd \circ snd)\ sstls\rightarrow*\ \infty$
⟨*proof*⟩

lemma *silent-move-fromI* [*intro*]:
  $[\![\ silent\text{-}moves\ s0\ s1;\ silent\text{-}move\ s1\ s2\ ]\!] \Longrightarrow silent\text{-}move\text{-}from\ s0\ s1\ s2$
⟨*proof*⟩

lemma *silent-move-fromE* [*elim*]:
  assumes *silent-move-from s0 s1 s2*
  obtains *silent-moves s0 s1 silent-move s1 s2*
⟨*proof*⟩

lemma *rtranclp-silent-move-from-imp-silent-moves*:
  assumes $s'x$: $silent\text{-}move^{**}\ s'\ x$
  shows $(silent\text{-}move\text{-}from\ s')\hat{}**\ x\ z \Longrightarrow silent\text{-}moves\ s'\ z$
⟨*proof*⟩

lemma *τdiverge-not-wfP-silent-move-from*:
  assumes $s\ -\tau\rightarrow\ \infty$
  shows $\neg\ wfP\ (flip\ (silent\text{-}move\text{-}from\ s))$
⟨*proof*⟩

lemma *wfP-silent-move-from-unroll*:
  assumes $wfPs'$: $\bigwedge s'.\ s\ -\tau\rightarrow\ s' \Longrightarrow wfP\ (flip\ (silent\text{-}move\text{-}from\ s'))$
  shows $wfP\ (flip\ (silent\text{-}move\text{-}from\ s))$
  ⟨*proof*⟩

lemma *not-wfP-silent-move-from-τdiverge*:
  assumes $\neg\ wfP\ (flip\ (silent\text{-}move\text{-}from\ s))$
  shows $s\ -\tau\rightarrow\ \infty$
⟨*proof*⟩

lemma *τdiverge-neq-wfP-silent-move-from*:
  $s\ -\tau\rightarrow\ \infty \neq wfP\ (flip\ (silent\text{-}move\text{-}from\ s))$
⟨*proof*⟩

lemma *not-τdiverge-to-no-τmove*:
  assumes $\neg\ s\ -\tau\rightarrow\ \infty$
  shows $\exists\, s'.\ s\ -\tau\rightarrow*\ s' \wedge (\forall\, s''.\ \neg\ s'\ -\tau\rightarrow\ s'')$
⟨*proof*⟩

lemma *τdiverge-conv-τRuns*:
  $s\ -\tau\rightarrow\ \infty \longleftrightarrow s \Downarrow TNil\ None$

⟨*proof*⟩

lemma *τinf-step-into-τRuns*:
s −*τ*−*tls*→∗ ∞ ⟹ *s* ⇓ *tllist-of-llist None tls*
⟨*proof*⟩

lemma *τ-into-τRuns*:
⟦ *s* −*τ*→ *s*′; *s*′ ⇓ *tls* ⟧ ⟹ *s* ⇓ *tls*
⟨*proof*⟩

lemma *τrtrancl3p-into-τRuns*:
assumes *s* −*τ*−*tls*→∗ *s*′
and *s*′ ⇓ *tls*′
shows *s* ⇓ *lappendt* (*llist-of tls*) *tls*′
⟨*proof*⟩

lemma *τRuns-table-into-τRuns*:
τRuns-table s stlsss ⟹ *s* ⇓ *tmap fst id stlsss*
⟨*proof*⟩

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*′′ ∧ ¬ *τmove s*′ (*thd tls*) *s*′′ ∧ *s*′′ ⇓ *ttl tls*, *ttl tls*))
(*s*, *tls*)

lemma *is-TNil-τRuns2τRuns-table* [*simp*]:
is-TNil (*τRuns2τRuns-table s tls*) ⟷ *is-TNil tls*
thm *unfold-tllist.disc*
⟨*proof*⟩

lemma *thd-τRuns2τRuns-table* [*simp*]:
¬ *is-TNil tls* ⟹
thd (*τRuns2τRuns-table s tls*) =
(*thd tls*, *SOME s*′′. ∃ *s*′. *s* −*τ*→∗ *s*′ ∧ *s*′ −*thd tls*→ *s*′′ ∧ ¬ *τmove s*′ (*thd tls*) *s*′′ ∧ *s*′′ ⇓ *ttl tls*)
⟨*proof*⟩

lemma *ttl-τRuns2τRuns-table* [*simp*]:
¬ *is-TNil tls* ⟹
ttl (*τRuns2τRuns-table s tls*) =
τRuns2τRuns-table (*SOME s*′′. ∃ *s*′. *s* −*τ*→∗ *s*′ ∧ *s*′ −*thd tls*→ *s*′′ ∧ ¬ *τmove s*′ (*thd tls*) *s*′′ ∧ *s*′′ ⇓ *ttl tls*) (*ttl tls*)
⟨*proof*⟩

lemma *terminal-τRuns2τRuns-table* [*simp*]:
is-TNil tls ⟹ *terminal* (*τRuns2τRuns-table s tls*) = *terminal tls*
⟨*proof*⟩

lemma *τRuns2τRuns-table-simps* [*simp*, *nitpick-simp*]:

τ*Runs2*τ*Runs-table s* (*TNil so*) = *TNil so*
$\bigwedge$*tl.*
τ*Runs2*τ*Runs-table s* (*TCons tl tls*) =
(*let s''* = *SOME s''.* $\exists$ *s'. s* $-\tau\rightarrow*$ *s'* $\wedge$ *s'* $-tl\rightarrow$ *s''* $\wedge$ $\neg$ τ*move s' tl s''* $\wedge$ *s''* $\Downarrow$ *tls*
in TCons (*tl, s''*) (τ*Runs2*τ*Runs-table s'' tls*))
⟨*proof*⟩

lemma τ*Runs2*τ*Runs-table-inverse*:
tmap fst id (τ*Runs2*τ*Runs-table s tls*) = *tls*
⟨*proof*⟩

lemma τ*Runs-into-*τ*Runs-table*:
assumes *s* $\Downarrow$ *tls*
shows $\exists$ *stlsss. tls* = *tmap fst id stlsss* $\wedge$ τ*Runs-table s stlsss*
⟨*proof*⟩

lemma τ*Runs-lappendtE*:
assumes σ $\Downarrow$ *lappendt tls tls'*
and *lfinite tls*
obtains σ' **where** σ $-\tau-$*list-of tls*$\rightarrow*$ σ'
and σ' $\Downarrow$ *tls'*
⟨*proof*⟩

lemma τ*Runs-total*:
$\exists$ *tls.* σ $\Downarrow$ *tls*
⟨*proof*⟩

lemma *silent-move2-into-silent-move*:
fixes *tl*
assumes *silent-move2 s tl s'*
shows *s* $-\tau\rightarrow$ *s'*
⟨*proof*⟩

lemma *silent-move-into-silent-move2*:
assumes *s* $-\tau\rightarrow$ *s'*
shows $\exists$ *tl. silent-move2 s tl s'*
⟨*proof*⟩

lemma *silent-moves2-into-silent-moves*:
assumes *silent-moves2 s tls s'*
shows *s* $-\tau\rightarrow*$ *s'*
⟨*proof*⟩

lemma *silent-moves-into-silent-moves2*:
assumes *s* $-\tau\rightarrow*$ *s'*
shows $\exists$ *tls. silent-moves2 s tls s'*
⟨*proof*⟩

lemma *inf-step-silent-move2-into-*τ*diverge*:
trsys.inf-step silent-move2 s tls $\Longrightarrow$ *s* $-\tau\rightarrow$ ∞
⟨*proof*⟩

lemma τ*diverge-into-inf-step-silent-move2*:
assumes *s* $-\tau\rightarrow$ ∞

obtains *tls* **where** *trsys.inf-step silent-move2 s tls*
⟨*proof*⟩

lemma *τRuns-into-τrtrancl3p*:
assumes *runs*: $s \Downarrow tlss$
and *fin*: *tfinite tlss*
and *terminal*: *terminal tlss* = *Some* s'
shows *τrtrancl3p s* (*list-of* (*llist-of-tllist tlss*)) s'
⟨*proof*⟩

lemma *τRuns-terminal-stuck*:
assumes *Runs*: $s \Downarrow tlss$
and *fin*: *tfinite tlss*
and *terminal*: *terminal tlss* = *Some* s'
and *proceed*: $s' -tls\rightarrow s''$
shows *False*
⟨*proof*⟩

lemma *Runs-table-silent-diverge*:
⟦ *Runs-table s stlss*; ∀ (*s*, *tl*, s') ∈ *lset stlss*. *τmove s tl* s'; ¬ *lfinite stlss* ⟧
$\Longrightarrow s -\tau\rightarrow \infty$
⟨*proof*⟩

lemma *Runs-table-silent-rtrancl*:
assumes *lfinite stlss*
and *Runs-table s stlss*
and ∀ (*s*, *tl*, s') ∈ *lset stlss*. *τmove s tl* s'
shows $s -\tau\rightarrow*$ *llast* (*LCons s* (*lmap* (λ(*s*, *tl*, s'). s') *stlss*)) (**is** *?thesis1*)
and *llast* (*LCons s* (*lmap* (λ(*s*, *tl*, s'). s') *stlss*)) $-tl'\rightarrow s'' \Longrightarrow$ *False* (**is** *PROP ?thesis2*)
⟨*proof*⟩

lemma *Runs-table-silent-lappendD*:
fixes *s stlss*
defines $s' \equiv$ *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 $s -\tau\rightarrow* s'$ (**is** *?thesis1*)
and *Runs-table* s' *stlss'* (**is** *?thesis2*)
and *stlss'* ≠ *LNil* $\Longrightarrow s' =$ *fst* (*lhd stlss'*) (**is** *PROP ?thesis3*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *τRuns-table2-into-τRuns*:
τRuns-table2 s tlsstlss
$\Longrightarrow s \Downarrow$ *tmap* (λ(*tls*, s', *tl*, s''). *tl*) (λ*x*. *case x of Inl* (*tls*, s') ⇒ *Some* s' | *Inr* - ⇒ *None*) *tlsstlss*

⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

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''''*
⟨*proof*⟩

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''*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *τRuns-table2-lappendtD*:
assumes *Runs*: *τRuns-table2 s* (*lappendt tlsstlss tlsstlss'*)
and *fin*: *lfinite tlsstlss*
shows ∃ *s'*. *τRuns-table2 s' tlsstlss'*
⟨*proof*⟩

end

lemma *τmoves-False*: *τtrsys.silent-move r* (λ*s ta s'*. *False*) = (λ*s s'*. *False*)
⟨*proof*⟩

lemma *τrtrancl3p-False-eq-rtrancl3p*: *τtrsys.τrtrancl3p r* (λ*s tl s'*. *False*) = *rtrancl3p r*
⟨*proof*⟩

lemma *τdiverge-empty-τmove*:
τtrsys.τdiverge r (λ*s ta s'*. *False*) = (λ*s*. *False*)

⟨*proof*⟩

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* ⟨*proof*⟩
sublocale *multithreaded-base* < *mthr*: *trsys redT* ⟨*proof*⟩
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'*)
⟨*proof*⟩

context *multithreaded* **begin**

sublocale *trsys r t* **for** *t* ⟨*proof*⟩

sublocale *mthr*: *trsys redT* ⟨*proof*⟩

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*

⟨*ML*⟩
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* ⟨*proof*⟩

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* ⟨*proof*⟩

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*⟩

lemma *mτmove-False*: *τmultithreaded.mτmove* (λ*s ta s′*. *False*) = (λ*s ta s′*. *False*)
⟨*proof*⟩

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*$ [], [], [], [], [], [])
⟨*proof*⟩

lemma *mτmove-heap*:
  assumes *redT*: *redT s* (*t*, *ta*) *s′*
  and *mτmove*: *mτmove s* (*t*, *ta*) *s′*
  shows *shr s′* = *shr s*
⟨*proof*⟩

lemma *τmredT-thread-preserved*:
  τmredT s s′ ⟹ *thr s t* = *None* ⟷ *thr s′ t* = *None*
⟨*proof*⟩

lemma *τmRedT-thread-preserved*:
  τmredT^** *s s′* ⟹ *thr s t* = *None* ⟷ *thr s′ t* = *None*
⟨*proof*⟩

lemma *τmtRedT-thread-preserved*:
  τmredT^++ *s s′* ⟹ *thr s t* = *None* ⟷ *thr s′ t* = *None*
⟨*proof*⟩

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*⟩

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')*
⟨*proof*⟩

lemma *τmtRed-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')*
⟨*proof*⟩

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 *τmredT s (redT-upd-ε s t x' m')*
⟨*proof*⟩

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 *τmredT^** s (redT-upd-ε s t x' m')*
⟨*proof*⟩

lemma *red-rtrancl-τ-heapD-inv*:
  ⟦ *silent-moves t s s'*; *wfs t s* ⟧ ⟹ *snd s' = snd s*
⟨*proof*⟩

lemma *red-trancl-τ-heapD-inv*:
  ⟦ *silent-movet t s s'*; *wfs t s* ⟧ ⟹ *snd s' = snd s*
⟨*proof*⟩

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')*
⟨*proof*⟩

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*
⟨*proof*⟩

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)*

⟨*proof*⟩

lemma $\tau mredT$-*preserves-final-thread*:
⟦ $\tau mredT\ s\ s'$; *final-thread s t* ⟧ $\Longrightarrow$ *final-thread s′ t*
⟨*proof*⟩

lemma $\tau mRedT$-*preserves-final-thread*:
⟦ $\tau mredT$^** *s s′*; *final-thread s t* ⟧ $\Longrightarrow$ *final-thread s′ t*
⟨*proof*⟩

lemma *silent-moves2-silentD*:
assumes *rtrancl3p mthr.silent-move2 s ttas s′*
and $(t,\ ta) \in$ *set ttas*
shows $ta = \varepsilon$
⟨*proof*⟩

lemma *inf-step-silentD*:
assumes *step*: *trsys.inf-step mthr.silent-move2 s ttas*
and *lset*: $(t,\ ta) \in$ *lset ttas*
shows $ta = \varepsilon$
⟨*proof*⟩

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 $m\mu t$ :: ′*m* ⇒ (′*l*,′*t*,′*x*) *thread-info* ⇒ (′*l*,′*t*,′*x*) *thread-info* ⇒ *bool*
for *m* **and** *ts* **and** *ts′*
where
$m\mu tI$:
$\bigwedge ln$. ⟦ *finite* (*dom ts*); *ts t* = ⌊(*x*, *ln*)⌋; *ts′ t* = ⌊(*x′*, *ln′*)⌋; μ (*x*, *m*) (*x′*, *m*); $\bigwedge t'$. $t' \neq t \Longrightarrow$ *ts t′* = *ts′ t′* ⟧
$\Longrightarrow m\mu t$ *m ts ts′*

definition $m\mu$ :: (′*l*,′*t*,′*x*,′*m*,′*w*) *state* ⇒ (′*l*,′*t*,′*x*,′*m*,′*w*) *state* ⇒ *bool*
where $m\mu$ *s s′* $\longleftrightarrow$ *shr s* = *shr s′* ∧ $m\mu t$ (*shr s*) (*thr s*) (*thr s′*)

lemma $m\mu t$-*thr-dom-eq*: $m\mu t$ *m ts ts′* $\Longrightarrow$ *dom ts* = *dom ts′*
⟨*proof*⟩

lemma $m\mu$-*finite-thrD*:
assumes $m\mu t$ *m ts ts′*
shows *finite* (*dom ts*) *finite* (*dom ts′*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *wf-mμ*: *wfP mμ*
⟨*proof*⟩

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*)
⟨*proof*⟩

end

lemma *bisimulation-flip-simps* [*flip-simps*]:
bisimulation trsys2 trsys1 (*flip bisim*) (*flip tlsim*) = *bisimulation trsys1 trsys2 bisim tlsim*
⟨*proof*⟩

context *bisimulation* **begin**

lemma *simulation1-rtrancl*:
⟦*s1* −*1*−*tls1*→∗ *s1'*; *s1* ≈ *s2*⟧
⟹ ∃ *s2' tls2*. *s2* −*2*−*tls2*→∗ *s2'* ∧ *s1'* ≈ *s2'* ∧ *tls1* [∼] *tls2*
⟨*proof*⟩

lemma *simulation2-rtrancl*:
⟦*s2* −*2*−*tls2*→∗ *s2'*; *s1* ≈ *s2*⟧
⟹ ∃ *s1' tls1*. *s1* −*1*−*tls1*→∗ *s1'* ∧ *s1'* ≈ *s2'* ∧ *tls1* [∼] *tls2*
⟨*proof*⟩

lemma *simulation1-inf-step*:
assumes *red1*: *s1* −*1*−*tls1*→∗ ∞ **and** *bisim*: *s1* ≈ *s2*
shows ∃ *tls2*. *s2* −*2*−*tls2*→∗ ∞ ∧ *tls1* [[∼]] *tls2*
⟨*proof*⟩

lemma *simulation2-inf-step*:
⟦ *s2* −*2*−*tls2*→∗ ∞; *s1* ≈ *s2* ⟧ ⟹ ∃ *tls1*. *s1* −*1*−*tls1*→∗ ∞ ∧ *tls1* [[∼]] *tls2*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *final-simulation2*:
⟦ *s1* ≈ *s2*; *s2* −*2*−*tls2*→∗ *s2'*; *final2 s2'* ⟧
⟹ ∃ *s1' tls1*. *s1* −*1*−*tls1*→∗ *s1'* ∧ *s1'* ≈ *s2'* ∧ *final1 s1'* ∧ *tls1* [∼] *tls2*
⟨*proof*⟩

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 (‹-/ −τ*1*→ -› [*50*, *50*] *60*) **and**

trsys2.silent-move (‹-/ −τ2→ -› [50, 50] 60)

notation
trsys1.silent-moves (‹-/ −τ1→* -› [50, 50] 60) **and**
trsys2.silent-moves (‹-/ −τ2→* -› [50, 50] 60)

notation
trsys1.silent-movet (‹-/ −τ1→+ -› [50, 50] 60) **and**
trsys2.silent-movet (‹-/ −τ2→+ -› [50, 50] 60)

notation
trsys1.τrtrancl3p (‹- −τ1−-→* -› [50, 0, 50] 60) **and**
trsys2.τrtrancl3p (‹- −τ2−-→* -› [50, 0, 50] 60)

notation
trsys1.τinf-step (‹- −τ1−-→* ∞› [50, 0] 80) **and**
trsys2.τinf-step (‹- −τ2−-→* ∞› [50, 0] 80)

notation
trsys1.τdiverge (‹- −τ1→ ∞› [50] 80) **and**
trsys2.τdiverge (‹- −τ2→ ∞› [50] 80)

notation
trsys1.τinf-step-table (‹- −τ1−-→*t ∞› [50, 0] 80) **and**
trsys2.τinf-step-table (‹- −τ2−-→*t ∞› [50, 0] 80)

notation
trsys1.τRuns (‹- ⇓1 -› [50, 50] 51) **and**
trsys2.τRuns (‹- ⇓2 -› [50, 50] 51)

lemma *simulation-silent1I′*:
assumes ∃ *s2′*. (*if μ1 s1′ s1 then trsys2.silent-moves else trsys2.silent-movet*) *s2 s2′* ∧ *s1′* ≈ *s2′*
shows *s1′* ≈ *s2* ∧ *μ1^++ s1′ s1* ∨ (∃ *s2′*. *s2* −τ2→+ *s2′* ∧ *s1′* ≈ *s2′*)
⟨*proof*⟩

lemma *simulation-silent2I′*:
assumes ∃ *s1′*. (*if μ2 s2′ s2 then trsys1.silent-moves else trsys1.silent-movet*) *s1 s1′* ∧ *s1′* ≈ *s2′*
shows *s1* ≈ *s2′* ∧ *μ2^++ s2′ s2* ∨ (∃ *s1′*. *s1* −τ1→+ *s1′* ∧ *s1′* ≈ *s2′*)
⟨*proof*⟩

end

locale *delay-bisimulation-obs* = *delay-bisimulation-base* - - - - *τmove1 τmove2*
for *τmove1* :: *′s1* ⇒ *′tl1* ⇒ *′s1* ⇒ *bool*
and *τmove2* :: *′s2* ⇒ *′tl2* ⇒ *′s2* ⇒ *bool* +
assumes *simulation1*:
⟦ *s1* ≈ *s2*; *s1* −1−*tl1*→ *s1′*; ¬ *τmove1 s1 tl1 s1′* ⟧
⟹ ∃ *s2′ s2″ tl2*. *s2* −τ2→* *s2′* ∧ *s2′* −2−*tl2*→ *s2″* ∧ ¬ *τmove2 s2′ tl2 s2″* ∧ *s1′* ≈ *s2″* ∧ *tl1* ∼ *tl2*
and *simulation2*:
⟦ *s1* ≈ *s2*; *s2* −2−*tl2*→ *s2′*; ¬ *τmove2 s2 tl2 s2′* ⟧
⟹ ∃ *s1′ s1″ tl1*. *s1* −τ1→* *s1′* ∧ *s1′* −1−*tl1*→ *s1″* ∧ ¬ *τmove1 s1′ tl1 s1″* ∧ *s1″* ≈ *s2′* ∧ *tl1* ∼ *tl2*
begin

lemma *delay-bisimulation-obs-flip*: *delay-bisimulation-obs trsys2 trsys1* (*flip bisim*) (*flip tlsim*) *τmove2 τmove1*
⟨*proof*⟩

end

lemma *delay-bisimulation-obs-flip-simps* [*flip-simps*]:
delay-bisimulation-obs trsys2 trsys1 (*flip bisim*) (*flip tlsim*) *τmove2 τmove1* =
delay-bisimulation-obs trsys1 trsys2 bisim tlsim τmove1 τmove2
⟨*proof*⟩

locale *delay-bisimulation-diverge* = *delay-bisimulation-obs* - - - - *τmove1 τmove2*
for *τmove1* :: *'s1* ⇒ *'tl1* ⇒ *'s1* ⇒ *bool*
and *τmove2* :: *'s2* ⇒ *'tl2* ⇒ *'s2* ⇒ *bool* +
assumes *simulation-silent1*:
⟦ *s1* ≈ *s2*; *s1* −*τ1*→ *s1'* ⟧ ⟹ ∃ *s2'*. *s2* −*τ2*→∗ *s2'* ∧ *s1'* ≈ *s2'*
and *simulation-silent2*:
⟦ *s1* ≈ *s2*; *s2* −*τ2*→ *s2'* ⟧ ⟹ ∃ *s1'*. *s1* −*τ1*→∗ *s1'* ∧ *s1'* ≈ *s2'*
and *τdiverge-bisim-inv*: *s1* ≈ *s2* ⟹ *s1* −*τ1*→ ∞ ⟷ *s2* −*τ2*→ ∞
begin

lemma *delay-bisimulation-diverge-flip*: *delay-bisimulation-diverge trsys2 trsys1* (*flip bisim*) (*flip tlsim*) *τmove2 τmove1*
⟨*proof*⟩

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
⟨*proof*⟩

context *delay-bisimulation-diverge* **begin**

lemma *simulation-silents1*:
assumes *bisim*: *s1* ≈ *s2* **and** *moves*: *s1* −*τ1*→∗ *s1'*
shows ∃ *s2'*. *s2* −*τ2*→∗ *s2'* ∧ *s1'* ≈ *s2'*
⟨*proof*⟩

lemma *simulation-silents2*:
⟦ *s1* ≈ *s2*; *s2* −*τ2*→∗ *s2'* ⟧ ⟹ ∃ *s1'*. *s1* −*τ1*→∗ *s1'* ∧ *s1'* ≈ *s2'*
⟨*proof*⟩

lemma *simulation1-τrtrancl3p*:
⟦ *s1* −*τ1*−*tls1*→∗ *s1'*; *s1* ≈ *s2* ⟧
⟹ ∃ *tls2 s2'*. *s2* −*τ2*−*tls2*→∗ *s2'* ∧ *s1'* ≈ *s2'* ∧ *tls1* [∼] *tls2*
⟨*proof*⟩

lemma *simulation2-τrtrancl3p*:
⟦ *s2* −*τ2*−*tls2*→∗ *s2'*; *s1* ≈ *s2* ⟧
⟹ ∃ *tls1 s1'*. *s1* −*τ1*−*tls1*→∗ *s1'* ∧ *s1'* ≈ *s2'* ∧ *tls1* [∼] *tls2*
⟨*proof*⟩

lemma *simulation1-τinf-step*:
assumes *τinf1*: *s1 −τ1−tls1→∗ ∞* **and** *bisim*: *s1 ≈ s2*
shows ∃ *tls2*. *s2 −τ2−tls2→∗ ∞* ∧ *tls1* [[∼]] *tls2*
⟨*proof*⟩

lemma *simulation2-τinf-step*:
⟦ *s2 −τ2−tls2→∗ ∞*; *s1 ≈ s2* ⟧ =⇒ ∃ *tls1*. *s1 −τ1−tls1→∗ ∞* ∧ *tls1* [[∼]] *tls2*
⟨*proof*⟩

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*⟩

lemma *no-τmove2-τs-to-no-τmove1*:
⟦ *s1 ≈ s2*; ⋀*s2′*. ¬ *s2 −τ2→ s2′* ⟧ =⇒ ∃ *s1′*. *s1 −τ1→∗ s1′* ∧ (∀ *s1″*. ¬ *s1′ −τ1→ s1″*) ∧ *s1′* ≈ *s2*
⟨*proof*⟩

lemma *no-move1-to-no-move2*:
assumes *s1 ≈ s2*
and *no-moves1*: ⋀*tl1 s1′*. ¬ *s1 −1−tl1→ s1′*
shows ∃ *s2′*. *s2 −τ2→∗ s2′* ∧ (∀ *tl2 s2″*. ¬ *s2′ −2−tl2→ s2″*) ∧ *s1 ≈ s2′*
⟨*proof*⟩

lemma *no-move2-to-no-move1*:
⟦ *s1 ≈ s2*; ⋀*tl2 s2′*. ¬ *s2 −2−tl2→ s2′* ⟧
=⇒ ∃ *s1′*. *s1 −τ1→∗ s1′* ∧ (∀ *tl1 s1″*. ¬ *s1′ −1−tl1→ s1″*) ∧ *s1′ ≈ s2*
⟨*proof*⟩

lemma *simulation-τRuns-table1*:
assumes *bisim*: *s1 ≈ s2*
and *run1*: *trsys1.τRuns-table s1 stlsss1*
shows ∃ *stlsss2*. *trsys2.τRuns-table s2 stlsss2* ∧ *tllist-all2* (λ(*tl1*, *s1″*) (*tl2*, *s2″*). *tl1* ∼ *tl2* ∧ *s1″* ≈ *s2″*) (*rel-option bisim*) *stlsss1 stlsss2*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *simulation-τRuns2*:
⟦ *s1 ≈ s2*; *s2* ⇓*2 tls2* ⟧

⟹ ∃ *tls1*. *s1* ⇓*1 tls1* ∧ *tllist-all2 tlsim* (*rel-option bisim*) *tls1 tls2*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

context *delay-bisimulation-diverge-final* **begin**

lemma *delay-bisimulation-diverge*:
delay-bisimulation-diverge trsys1 trsys2 bisim tlsim τmove1 τmove2
⟨*proof*⟩

lemma *delay-bisimulation-final-base*:
delay-bisimulation-final-base trsys1 trsys2 bisim τmove1 τmove2 final1 final2
⟨*proof*⟩

lemma *final-simulation1*:
⟦ *s1* ≈ *s2*; *s1* −τ*1*−*tls1*→∗ *s1'*; *final1 s1'* ⟧
⟹ ∃ *s2' tls2*. *s2* −τ*2*−*tls2*→∗ *s2'* ∧ *s1'* ≈ *s2'* ∧ *final2 s2'* ∧ *tls1* [∼] *tls2*
⟨*proof*⟩

lemma *final-simulation2*:
⟦ *s1* ≈ *s2*; *s2* −τ*2*−*tls2*→∗ *s2'*; *final2 s2'* ⟧
⟹ ∃ *s1' tls1*. *s1* −τ*1*−*tls1*→∗ *s1'* ∧ *s1'* ≈ *s2'* ∧ *final1 s1'* ∧ *tls1* [∼] *tls2*
⟨*proof*⟩

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 $\tau move2$ :: $'s2 \Rightarrow 'tl2 \Rightarrow 's2 \Rightarrow bool$
and $\mu 1$ :: $'s1 \Rightarrow 's1 \Rightarrow bool$
and $\mu 2$ :: $'s2 \Rightarrow 's2 \Rightarrow bool$ +
assumes *simulation-silent1*:
$[\![s1 \approx s2; s1 -\tau 1\rightarrow s1']\!] \Longrightarrow s1' \approx s2 \wedge \mu 1\hat{}{++}\ s1'\ s1 \vee (\exists s2'.\ s2 -\tau 2\rightarrow+ s2' \wedge s1' \approx s2')$
and *simulation-silent2*:
$[\![s1 \approx s2; s2 -\tau 2\rightarrow s2']\!] \Longrightarrow s1 \approx s2' \wedge \mu 2\hat{}{++}\ s2'\ s2 \vee (\exists s1'.\ s1 -\tau 1\rightarrow+ s1' \wedge s1' \approx s2')$
and *wf-μ1*: *wfP* $\mu 1$
and *wf-μ2*: *wfP* $\mu 2$
begin

lemma *delay-bisimulation-measure-flip*:
delay-bisimulation-measure trsys2 trsys1 (*flip bisim*) (*flip tlsim*) $\tau move2$ $\tau move1$ $\mu 2$ $\mu 1$
⟨*proof*⟩

end

lemma *delay-bisimulation-measure-flip-simps* [*flip-simps*]:
delay-bisimulation-measure trsys2 trsys1 (*flip bisim*) (*flip tlsim*) $\tau move2$ $\tau move1$ $\mu 2$ $\mu 1$ =
delay-bisimulation-measure trsys1 trsys2 bisim tlsim $\tau move1$ $\tau move2$ $\mu 1$ $\mu 2$
⟨*proof*⟩

context *delay-bisimulation-measure* **begin**

lemma *simulation-silentst1*:
assumes *bisim*: $s1 \approx s2$ **and** *moves*: $s1 -\tau 1\rightarrow+ s1'$
shows $s1' \approx s2 \wedge \mu 1\hat{}{++}\ s1'\ s1 \vee (\exists s2'.\ s2 -\tau 2\rightarrow+ s2' \wedge s1' \approx s2')$
⟨*proof*⟩

lemma *simulation-silentst2*:
$[\![s1 \approx s2; s2 -\tau 2\rightarrow+ s2']\!] \Longrightarrow s1 \approx s2' \wedge \mu 2\hat{}{++}\ s2'\ s2 \vee (\exists s1'.\ s1 -\tau 1\rightarrow+ s1' \wedge s1' \approx s2')$
⟨*proof*⟩

lemma *τdiverge-simulation1*:
assumes *diverge1*: $s1 -\tau 1\rightarrow \infty$
and *bisim*: $s1 \approx s2$
shows $s2 -\tau 2\rightarrow \infty$
⟨*proof*⟩

lemma *τdiverge-simulation2*:
$[\![s2 -\tau 2\rightarrow \infty; s1 \approx s2]\!] \Longrightarrow s1 -\tau 1\rightarrow \infty$
⟨*proof*⟩

lemma *τdiverge-bisim-inv*:
$s1 \approx s2 \Longrightarrow s1 -\tau 1\rightarrow \infty \longleftrightarrow s2 -\tau 2\rightarrow \infty$
⟨*proof*⟩

end

sublocale *delay-bisimulation-measure* < *delay-bisimulation-diverge*
⟨*proof*⟩

Counter example for *delay-bisimulation-diverge trsys1 trsys2 bisim tlsim* $\tau move1$ $\tau move2$ $\Longrightarrow \exists \mu 1\ \mu 2.$ *delay-bisimulation-measure trsys1 trsys2 bisim tlsim* $\tau move1$ $\tau move2$ $\mu 1$ $\mu 2$

(only τmoves):

```
--|
| v
--a  ~  x
  |     |
  |     |
  v     v
--b  ~  y--
| ^     ^ |
--|     |--
```

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* ⇒ '*s1* ⇒ *bool*
and μ*2* :: '*s2* ⇒ '*s2* ⇒ *bool*
and *final1* :: '*s1* ⇒ *bool*
and *final2* :: '*s2* ⇒ *bool*

sublocale *delay-bisimulation-measure-final* < *delay-bisimulation-diverge-final*
⟨*proof*⟩

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* ⇒ '*s1* ⇒ *bool*
and τ*moves2* :: '*s2* ⇒ '*s2* ⇒ *bool*
assumes τ*inv*: ⟦ *s1* ≈ *s2*; *s1* −*1*−*tl1*→ *s1*'; *s2* −*2*−*tl2*→ *s2*'; *s1*' ≈ *s2*'; *tl1* ∼ *tl2* ⟧
⟹ τ*move1 s1 tl1 s1*' ⟷ τ*move2 s2 tl2 s2*'
begin

lemma τ*inv-flip*:
τ*inv trsys2 trsys1* (*flip bisim*) (*flip tlsim*) τ*move2* τ*move1*
⟨*proof*⟩

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*
⟨*proof*⟩

locale *bisimulation-into-delay* =
bisimulation + *τinv* +
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

lemma *bisimulation-into-delay-flip*:
bisimulation-into-delay trsys2 trsys1 (*flip bisim*) (*flip tlsim*) *τmove2 τmove1*
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

lemma *simulation-silent2-aux*:
$[\![\ s1 \approx s2;\ s2 -\tau 2\rightarrow s2'\ ]\!] \Longrightarrow s1 \approx s2' \wedge \mu 2^{++}\ s2'\ s2 \vee (\exists s1'.\ s1 -\tau 1\rightarrow+\ s1' \wedge s1' \approx s2')$
⟨*proof*⟩

lemma *simulation1-aux*:
assumes *bisim*: $s1 \approx s2$ **and** *tr1*: $s1 -1-tl1\rightarrow s1'$ **and** *τ1*: $\neg\ \tau move1\ s1\ tl1\ s1'$
shows $\exists s2'\ s2''\ tl2.\ s2 -\tau 2\rightarrow*\ s2' \wedge s2' -2-tl2\rightarrow s2'' \wedge \neg\ \tau move2\ s2'\ tl2\ s2'' \wedge s1' \approx s2'' \wedge tl1 \sim tl2$
⟨*proof*⟩

lemma *simulation2-aux*:
$[\![\ s1 \approx s2;\ s2 -2-tl2\rightarrow s2';\ \neg\ \tau move2\ s2\ tl2\ s2'\ ]\!]$
$\Longrightarrow \exists s1'\ s1''\ tl1.\ s1 -\tau 1\rightarrow*\ s1' \wedge s1' -1-tl1\rightarrow s1'' \wedge \neg\ \tau move1\ s1'\ tl1\ s1'' \wedge s1'' \approx s2' \wedge tl1 \sim tl2$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *delay-bisimulation*:
delay-bisimulation-diverge trsys1 trsys2 bisim tlsim τmove1 τmove2
⟨*proof*⟩

end

sublocale *bisimulation-into-delay* < *delay-bisimulation-diverge*
⟨*proof*⟩

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*⟩

context *bisimulation-final* **begin**

lemma *delay-bisimulation-final-base*:
delay-bisimulation-final-base trsys1 trsys2 bisim τmove1 τmove2 final1 final2
⟨*proof*⟩

end

sublocale *bisimulation-final* < *delay-bisimulation-final-base*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bisim-composeE* [*elim*!]:
assumes *bisim*: (*bisim12* $\circ_B$ *bisim23*) *s1 s3*
obtains *s2* **where** *bisim12 s1 s2 bisim23 s2 s3*
⟨*proof*⟩

lemma *bisim-compose-assoc* [*simp*]:
(*bisim12* $\circ_B$ *bisim23*) $\circ_B$ *bisim34* = *bisim12* $\circ_B$ *bisim23* $\circ_B$ *bisim34*
⟨*proof*⟩

lemma *bisim-compose-conv-relcomp*:
case-prod (*bisim-compose bisim12 bisim23*) = (*λx. x* ∈ *relcomp* (*Collect* (*case-prod bisim12*)) (*Collect* (*case-prod bisim23*)))
⟨*proof*⟩

lemma *list-all2-bisim-composeI*:
⟦ *list-all2 A xs ys*; *list-all2 B ys zs* ⟧
⟹ *list-all2* (*A* $\circ_B$ *B*) *xs zs*
⟨*proof*⟩

lemma *delay-bisimulation-diverge-compose*:
assumes *wbisim12*: *delay-bisimulation-diverge trsys1 trsys2 bisim12 tlsim12 τmove1 τmove2*
and *wbisim23*: *delay-bisimulation-diverge trsys2 trsys3 bisim23 tlsim23 τmove2 τmove3*
shows *delay-bisimulation-diverge trsys1 trsys3* (*bisim12* $\circ_B$ *bisim23*) (*tlsim12* $\circ_B$ *tlsim23*) *τmove1*

τmove3
⟨*proof*⟩

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*)
⟨*proof*⟩

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* $\circ_B$ *bisim23*) (*tlsim12* $\circ_B$ *tlsim23*) *τmove1 τmove3 final1 final3*
⟨*proof*⟩

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* ⇒ (*'x1* × *'m1*, *'x2* × *'m2*) *bisim*) ⇒ ((*'t*,*'x1*,*'m1*) *new-thread-action*, (*'t*,*'x2*,*'m2*) *new-thread-action*) *bisim*
where
[*code del*]: *nta-bisim bisim* (*NewThread t x m*) *ta* = (∃ *x' m'*. *ta* = *NewThread t x' m'* ∧ *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'* ⇒ *t* = *t'* ∧ *bisim t* (*x*, *m*) (*x'*, *m'*) | - ⇒ *False*)
⟨*proof*⟩

lemma *nta-bisim-simps-sym* [*simp*]:
nta-bisim bisim ta (*NewThread t x m*) = (∃ *x' m'*. *ta* = *NewThread t x' m'* ∧ *bisim t* (*x'*, *m'*) (*x*, *m*))
nta-bisim bisim ta (*ThreadExists t b*) = (*ta* = *ThreadExists t b*)
⟨*proof*⟩

definition *ta-bisim* :: (*'t* ⇒ (*'x1* × *'m1*, *'x2* × *'m2*) *bisim*) ⇒ ((*'l*,*'t*,*'x1*,*'m1*,*'w*,*'o*) *thread-action*, (*'l*,*'t*,*'x2*,*'m2*,*'w*,*'o*) *thread-action*) *bisim*
where
ta-bisim bisim ta1 ta2 ≡
$\{\!|$ *ta1* $|\!\}_l$ = $\{\!|$ *ta2* $|\!\}_l$ ∧ $\{\!|$ *ta1* $|\!\}_w$ = $\{\!|$ *ta2* $|\!\}_w$ ∧ $\{\!|$ *ta1* $|\!\}_c$ = $\{\!|$ *ta2* $|\!\}_c$ ∧ $\{\!|$ *ta1* $|\!\}_o$ = $\{\!|$ *ta2* $|\!\}_o$ ∧ $\{\!|$ *ta1* $|\!\}_i$ = $\{\!|$ *ta2* $|\!\}_i$ ∧
list-all2 (*nta-bisim bisim*) $\{\!|$ *ta1* $|\!\}_t$ $\{\!|$ *ta2* $|\!\}_t$

lemma *ta-bisim-empty* [*iff*]: *ta-bisim bisim ε ε*
⟨*proof*⟩

lemma *ta-bisim-ε* [*simp*]:
ta-bisim b ε ta′ ⟷ ta′ = ε ta-bisim b ta ε ⟷ ta = ε
⟨*proof*⟩

lemma *nta-bisim-mono*:
assumes *major*: *nta-bisim bisim ta ta′*
and *mono*: ⋀*t s1 s2*. *bisim t s1 s2* ⟹ *bisim′ t s1 s2*
shows *nta-bisim bisim′ ta ta′*
⟨*proof*⟩

lemma *ta-bisim-mono*:
assumes *major*: *ta-bisim bisim ta1 ta2*
and *mono*: ⋀*t s1 s2*. *bisim t s1 s2* ⟹ *bisim′ t s1 s2*
shows *ta-bisim bisim′ ta1 ta2*
⟨*proof*⟩

lemma *nta-bisim-flip* [*flip-simps*]:
nta-bisim (λ*t*. *flip* (*bisim t*)) = *flip* (*nta-bisim bisim*)
⟨*proof*⟩

lemma *ta-bisim-flip* [*flip-simps*]:
ta-bisim (λ*t*. *flip* (*bisim t*)) = *flip* (*ta-bisim bisim*)
⟨*proof*⟩

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 $ta1 \sim m\ ta2 \equiv ta\text{-}bisim\ bisim\ ta1\ ta2$

definition $tbisim :: bool \Rightarrow 't \Rightarrow ('x1 \times 'l\ released\text{-}locks)\ option \Rightarrow 'm1 \Rightarrow ('x2 \times 'l\ released\text{-}locks)\ option \Rightarrow 'm2 \Rightarrow bool$ **where**
$\bigwedge ln.\ tbisim\ nw\ t\ ts1\ m1\ ts2\ m2 \longleftrightarrow$
$(case\ ts1\ of\ None \Rightarrow ts2 = None$
$\mid \lfloor(x1,\ ln)\rfloor \Rightarrow (\exists x2.\ ts2 = \lfloor(x2,\ ln)\rfloor \wedge t \vdash (x1,\ m1) \approx (x2,\ m2) \wedge (nw \vee x1 \approx w\ x2)))$

lemma $tbisim\text{-}NoneI$: $tbisim\ w\ t\ None\ m\ None\ m'$
⟨proof⟩

lemma $tbisim\text{-}SomeI$:
$\bigwedge ln.\ [\![\ t \vdash (x,\ m) \approx (x',\ m');\ nw \vee x \approx w\ x'\ ]\!] \Longrightarrow tbisim\ nw\ t\ (Some\ (x,\ ln))\ m\ (Some\ (x',\ ln))\ m'$
⟨proof⟩

lemma $tbisim\text{-}cases[consumes\ 1,\ case\text{-}names\ None\ Some]$:
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 $thesis$
⟨proof⟩

definition $mbisim :: (('l,'t,'x1,'m1,'w)\ state,\ ('l,'t,'x2,'m2,'w)\ state)\ bisim$ (‹- ≈m -› $[50,\ 50]\ 60$)
where
$s1 \approx m\ s2 \equiv$
$finite\ (dom\ (thr\ s1)) \wedge locks\ s1 = locks\ s2 \wedge wset\ s1 = wset\ s2 \wedge wset\text{-}thread\text{-}ok\ (wset\ s1)\ (thr\ s1)$
$\wedge$
$interrupts\ s1 = interrupts\ s2\ \wedge$
$(\forall t.\ tbisim\ (wset\ s2\ t = None)\ t\ (thr\ s1\ t)\ (shr\ s1)\ (thr\ s2\ t)\ (shr\ s2))$

lemma $mbisim\text{-}thrNone\text{-}eq$: $s1 \approx m\ s2 \Longrightarrow thr\ s1\ t = None \longleftrightarrow thr\ s2\ t = None$
⟨proof⟩

lemma $mbisim\text{-}thrD1$:
$\bigwedge ln.\ [\![\ s1 \approx m\ s2;\ thr\ s1\ t = \lfloor(x,\ ln)\rfloor\ ]\!]$
$\Longrightarrow \exists x'.\ thr\ s2\ t = \lfloor(x',\ ln)\rfloor \wedge t \vdash (x,\ shr\ s1) \approx (x',\ shr\ s2) \wedge (wset\ s1\ t = None \vee x \approx w\ x')$
⟨proof⟩

lemma $mbisim\text{-}thrD2$:
$\bigwedge ln.\ [\![\ s1 \approx m\ s2;\ thr\ s2\ t = \lfloor(x,\ ln)\rfloor\ ]\!]$
$\Longrightarrow \exists x'.\ thr\ s1\ t = \lfloor(x',\ ln)\rfloor \wedge t \vdash (x',\ shr\ s1) \approx (x,\ shr\ s2) \wedge (wset\ s2\ t = None \vee x' \approx w\ x)$
⟨proof⟩

lemma $mbisim\text{-}dom\text{-}eq$: $s1 \approx m\ s2 \Longrightarrow dom\ (thr\ s1) = dom\ (thr\ s2)$
⟨proof⟩

lemma $mbisim\text{-}wset\text{-}thread\text{-}ok1$:
$s1 \approx m\ s2 \Longrightarrow wset\text{-}thread\text{-}ok\ (wset\ s1)\ (thr\ s1)$
⟨proof⟩

lemma $mbisim\text{-}wset\text{-}thread\text{-}ok2$:
assumes $s1 \approx m\ s2$
shows $wset\text{-}thread\text{-}ok\ (wset\ s2)\ (thr\ s2)$
⟨proof⟩

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*);
    ⋀*t*. *thr s1 t* = *None* ⟹ *thr s2 t* = *None*;
    ⋀*t x1 ln*. *thr s1 t* = ⌊(*x1*, *ln*)⌋ ⟹ ∃ *x2*. *thr s2 t* = ⌊(*x2*, *ln*)⌋ ∧ *t* ⊢ (*x1*, *shr s1*) ≈ (*x2*, *shr s2*)
∧ (*wset s2 t* = *None* ∨ *x1* ≈*w x2*) ⟧
  ⟹ *s1* ≈*m s2*
⟨*proof*⟩

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*);
    ⋀*t*. *thr s2 t* = *None* ⟹ *thr s1 t* = *None*;
    ⋀*t x2 ln*. *thr s2 t* = ⌊(*x2*, *ln*)⌋ ⟹ ∃ *x1*. *thr s1 t* = ⌊(*x1*, *ln*)⌋ ∧ *t* ⊢ (*x1*, *shr s1*) ≈ (*x2*, *shr s2*)
∧ (*wset s2 t* = *None* ∨ *x1* ≈*w x2*) ⟧
  ⟹ *s1* ≈*m s2*
⟨*proof*⟩

lemma *mbisim-finite1*:
  s1 ≈*m s2* ⟹ *finite* (*dom* (*thr s1*))
⟨*proof*⟩

lemma *mbisim-finite2*:
  s1 ≈*m s2* ⟹ *finite* (*dom* (*thr s2*))
⟨*proof*⟩

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*) ∼*T* (*t'*, *ta2*) ⟷ *t* = *t'* ∧ *ta1* ∼*m ta2*
⟨*proof*⟩

definition *bisim-inv* :: *bool* **where**
  bisim-inv ≡ (∀ *s1 ta1 s1' s2 t*. *t* ⊢ *s1* ≈ *s2* ⟶ *t* ⊢ *s1* −*1*−*ta1*→ *s1'* ⟶ (∃ *s2'*. *t* ⊢ *s1'* ≈ *s2'*)) ∧
      (∀ *s2 ta2 s2' s1 t*. *t* ⊢ *s1* ≈ *s2* ⟶ *t* ⊢ *s2* −*2*−*ta2*→ *s2'* ⟶ (∃ *s1'*. *t* ⊢ *s1'* ≈ *s2'*))

lemma *bisim-invI*:
  ⟦ ⋀*s1 ta1 s1' s2 t*. ⟦ *t* ⊢ *s1* ≈ *s2*; *t* ⊢ *s1* −*1*−*ta1*→ *s1'* ⟧ ⟹ ∃ *s2'*. *t* ⊢ *s1'* ≈ *s2'*;
    ⋀*s2 ta2 s2' s1 t*. ⟦ *t* ⊢ *s1* ≈ *s2*; *t* ⊢ *s2* −*2*−*ta2*→ *s2'* ⟧ ⟹ ∃ *s1'*. *t* ⊢ *s1'* ≈ *s2'* ⟧
  ⟹ *bisim-inv*
⟨*proof*⟩

lemma *bisim-invD1*:
  ⟦ *bisim-inv*; *t* ⊢ *s1* ≈ *s2*; *t* ⊢ *s1* −*1*−*ta1*→ *s1'* ⟧ ⟹ ∃ *s2'*. *t* ⊢ *s1'* ≈ *s2'*
⟨*proof*⟩

lemma *bisim-invD2*:
  ⟦ *bisim-inv*; *t* ⊢ *s1* ≈ *s2*; *t* ⊢ *s2* −*2*−*ta2*→ *s2'* ⟧ ⟹ ∃ *s1'*. *t* ⊢ *s1'* ≈ *s2'*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

lemma *mta-bisim-flip* [*flip-simps*]:
FWbisimulation-base.mta-bisim (λ*t*. *flip* (*bisim t*)) = *flip* (*FWbisimulation-base.mta-bisim bisim*)
⟨*proof*⟩

lemma *flip-const* [*simp*]: *flip* (λ*a b*. *c*) = (λ*a b*. *c*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

end

lemma *FWbisimulation-base-aux-flip-simps* [*flip-simps*]:
FWbisimulation-base-aux final2 r2 final1 r1 = *FWbisimulation-base-aux final1 r1 final2 r2*
⟨*proof*⟩

sublocale *FWbisimulation-base-aux* < *mthr*:
bisimulation-final-base
r1.redT
r2.redT
mbisim
mta-bisim
r1.mfinal
r2.mfinal
⟨*proof*⟩

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*⟩

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*⟩

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
⟨*proof*⟩

lemma *activate-cond-actions1-flip* [*flip-simps*]:
FWdelay-bisimulation-base.activate-cond-actions1 final2 r2 final1 (λt. flip (bisim t)) τmove2 s2 s1 =
FWdelay-bisimulation-base.activate-cond-actions2 final1 final2 r2 bisim τmove2 s1 s2
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *activate-cond-action2-flip* [*flip-simps*]:
FWdelay-bisimulation-base.activate-cond-action2 final2 final1 r1 (λt. flip (bisim t)) τmove1 s2 s1 =
FWdelay-bisimulation-base.activate-cond-action1 final1 r1 final2 bisim τmove1 s1 s2
⟨*proof*⟩

lemma *activate-cond-actions2-flip* [*flip-simps*]:
FWdelay-bisimulation-base.activate-cond-actions2 final2 final1 r1 (λt. flip (bisim t)) τmove1 s2 s1 =
FWdelay-bisimulation-base.activate-cond-actions1 final1 r1 final2 bisim τmove1 s1 s2
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

context *FWdelay-bisimulation-base* **begin**

lemma *shr-activate-cond-action1* [*simp*]: *shr* (*activate-cond-action1 s1 s2 ct*) = *shr s1*
⟨*proof*⟩

lemma *shr-activate-cond-actions1* [*simp*]: *shr* (*activate-cond-actions1 s1 s2 cts*) = *shr s1*
⟨*proof*⟩

lemma *shr-activate-cond-action2* [*simp*]: *shr* (*activate-cond-action2 s1 s2 ct*) = *shr s2*
⟨*proof*⟩

lemma *shr-activate-cond-actions2* [*simp*]: *shr* (*activate-cond-actions2 s1 s2 cts*) = *shr s2*
⟨*proof*⟩

lemma *locks-activate-cond-action1* [*simp*]: *locks* (*activate-cond-action1 s1 s2 ct*) = *locks s1*
⟨*proof*⟩

lemma *locks-activate-cond-actions1* [*simp*]: *locks* (*activate-cond-actions1 s1 s2 cts*) = *locks s1*
⟨*proof*⟩

lemma *locks-activate-cond-action2* [*simp*]: *locks* (*activate-cond-action2 s1 s2 ct*) = *locks s2*
⟨*proof*⟩

lemma *locks-activate-cond-actions2* [*simp*]: *locks* (*activate-cond-actions2 s1 s2 cts*) = *locks s2*
⟨*proof*⟩

lemma *wset-activate-cond-action1* [*simp*]: *wset* (*activate-cond-action1 s1 s2 ct*) = *wset s1*
⟨*proof*⟩

lemma *wset-activate-cond-actions1* [*simp*]: *wset* (*activate-cond-actions1 s1 s2 cts*) = *wset s1*
⟨*proof*⟩

lemma *wset-activate-cond-action2* [*simp*]: *wset* (*activate-cond-action2 s1 s2 ct*) = *wset s2*
⟨*proof*⟩

lemma *wset-activate-cond-actions2* [*simp*]: *wset* (*activate-cond-actions2 s1 s2 cts*) = *wset s2*
⟨*proof*⟩

lemma *interrupts-activate-cond-action1* [*simp*]: *interrupts* (*activate-cond-action1 s1 s2 ct*) = *interrupts s1*
⟨*proof*⟩

lemma *interrupts-activate-cond-actions1* [*simp*]: *interrupts* (*activate-cond-actions1 s1 s2 cts*) = *interrupts s1*
⟨*proof*⟩

lemma *interrupts-activate-cond-action2* [*simp*]: *interrupts* (*activate-cond-action2 s1 s2 ct*) = *interrupts s2*
⟨*proof*⟩

lemma *interrupts-activate-cond-actions2* [*simp*]: *interrupts* (*activate-cond-actions2 s1 s2 cts*) = *interrupts s2*
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *cond-actions-ok-τmred2-inv*:
⟦ *τmred2 s2 s2'*; *r2.cond-action-ok s2 t ct* ⟧ ⟹ *r2.cond-action-ok s2' t ct*
⟨*proof*⟩

lemma *cond-actions-ok-τmRed1-inv*:
⟦ *τmRed1 s1 s1'*; *r1.cond-action-ok s1 t ct* ⟧ ⟹ *r1.cond-action-ok s1' t ct*
⟨*proof*⟩

lemma *cond-actions-ok-τmRed2-inv*:
⟦ *τmRed2 s2 s2'*; *r2.cond-action-ok s2 t ct* ⟧ ⟹ *r2.cond-action-ok s2' t ct*
⟨*proof*⟩

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*
⟨*proof*⟩

context *FWdelay-bisimulation-lift* **begin**

lemma *FWdelay-bisimulation-lift-flip*:
FWdelay-bisimulation-lift final2 r2 final1 r1 (*λt. flip* (*bisim t*)) *τmove2 τmove1*
⟨*proof*⟩

end

lemma *FWdelay-bisimulation-lift-flip-simps* [*flip-simps*]:
FWdelay-bisimulation-lift final2 r2 final1 r1 (*λt. flip* (*bisim t*)) *τmove2 τmove1* =
FWdelay-bisimulation-lift final1 r1 final2 r2 bisim τmove1 τmove2
⟨*proof*⟩

context *FWdelay-bisimulation-lift* **begin**

lemma *τinv-lift*: *τinv r1.redT r2.redT mbisim mta-bisim mτmove1 mτmove2*
⟨*proof*⟩

end

sublocale *FWdelay-bisimulation-lift* < *mthr*: *τinv r1.redT r2.redT mbisim mta-bisim mτmove1 mτmove2*
⟨*proof*⟩

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*
⟨*proof*⟩

context *FWdelay-bisimulation-final-base* **begin**

lemma *FWdelay-bisimulation-final-base-flip*:
FWdelay-bisimulation-final-base final2 r2 final1 r1 (*λt. flip* (*bisim t*)) *τmove2 τmove1*
⟨*proof*⟩

end

lemma *FWdelay-bisimulation-final-base-flip-simps* [*flip-simps*]:
FWdelay-bisimulation-final-base final2 r2 final1 r1 ($\lambda t.$ *flip* (*bisim t*)) τ*move2* τ*move1* =
FWdelay-bisimulation-final-base final1 r1 final2 r2 bisim τ*move1* τ*move2*
⟨*proof*⟩

context *FWdelay-bisimulation-final-base* **begin**

lemma *cond-actions-ok-bisim-ex-*τ*1-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*: $\bigwedge t'.\ t' \neq t \Longrightarrow$ *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 τ*mRed1* (*ls*, (*ts1*, *m1*), *ws*, *is*) *s1'*
and $\bigwedge t'.\ t' \neq t \Longrightarrow$ *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*⟩

lemma *cond-actions-oks-bisim-ex-*τ*1-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*: $\bigwedge t'.\ t' \neq t \Longrightarrow$ *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 τ*mRed1* (*ls*, (*ts1*, *m1*), *ws*, *is*) *s1'*
and $\bigwedge t'.\ t' \neq t \Longrightarrow$ *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*
⟨*proof*⟩

lemma *cond-actions-ok-bisim-ex-*τ*2-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*: $\bigwedge t'.\ t' \neq t \Longrightarrow$ *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 τ*mRed2* (*ls*, (*ts2*, *m2*), *ws*, *is*) *s2'*
and $\bigwedge t'.\ t' \neq t \Longrightarrow$ *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'*
⟨*proof*⟩

lemma *cond-actions-oks-bisim-ex-*τ*2-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*: $\bigwedge t'.\ t' \neq t \Longrightarrow$ *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 $\bigwedge t'$. $t' \neq t \Longrightarrow$ *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′*
⟨*proof*⟩

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*⟩

lemma *mfinal2-inv-simulation*:
s1 ≈m s2 ⟹ ∃ *s1′*. *r1.mthr.silent-moves s1 s1′* ∧ *s1′ ≈m s2* ∧ *r2.final-threads s2* ⊆ *r1.final-threads s1′* ∧ *shr s1′ = shr s1*
⟨*proof*⟩

lemma *mfinal1-simulation*:
assumes *s1 ≈m s2* **and** *r1.mfinal s1*
shows ∃ *s2′*. *r2.mthr.silent-moves s2 s2′* ∧ *s1 ≈m s2′* ∧ *r2.mfinal s2′* ∧ *shr s2′ = shr s2*
⟨*proof*⟩

lemma *mfinal2-simulation*:
⟦ *s1 ≈m s2*; *r2.mfinal s2* ⟧
⟹ ∃ *s1′*. *r1.mthr.silent-moves s1 s1′* ∧ *s1′ ≈m s2* ∧ *r1.mfinal s1′* ∧ *shr s1′ = shr s1*
⟨*proof*⟩

end

locale *FWdelay-bisimulation-obs* =
FWdelay-bisimulation-final-base - - - - - - - *τmove1 τmove2*
for *τmove1* :: (*′l*,*′t*,*′x1*,*′m1*,*′w*, *′o*) *τmoves*
and *τmove2* :: (*′l*,*′t*,*′x2*,*′m2*,*′w*, *′o*) *τmoves* +
assumes *delay-bisimulation-obs-locale*: *delay-bisimulation-obs* (*r1 t*) (*r2 t*) (*bisim t*) (*ta-bisim bisim*) *τmove1 τmove2*
and *bisim-inv-red-other*:
⟦ *t′* ⊢ (*x*, *m1*) ≈ (*xx*, *m2*); *t* ⊢ (*x1*, *m1*) ≈ (*x2*, *m2*);
r1.silent-moves t (*x1*, *m1*) (*x1′*, *m1*);
t ⊢ (*x1′*, *m1*) −*1*−*ta1*→ (*x1″*, *m1′*); ¬ *τmove1* (*x1′*, *m1*) *ta1* (*x1″*, *m1′*);
r2.silent-moves t (*x2*, *m2*) (*x2′*, *m2*);
t ⊢ (*x2′*, *m2*) −*2*−*ta2*→ (*x2″*, *m2′*); ¬ *τmove2* (*x2′*, *m2*) *ta2* (*x2″*, *m2′*);
t ⊢ (*x1″*, *m1′*) ≈ (*x2″*, *m2′*); *ta-bisim bisim ta1 ta2* ⟧
⟹ *t′* ⊢ (*x*, *m1′*) ≈ (*xx*, *m2′*)
and *bisim-waitI*:
⟦ *t* ⊢ (*x1*, *m1*) ≈ (*x2*, *m2*); *r1.silent-moves t* (*x1*, *m1*) (*x1′*, *m1*);
t ⊢ (*x1′*, *m1*) −*1*−*ta1*→ (*x1″*, *m1′*); ¬ *τmove1* (*x1′*, *m1*) *ta1* (*x1″*, *m1′*);
r2.silent-moves t (*x2*, *m2*) (*x2′*, *m2*);
t ⊢ (*x2′*, *m2*) −*2*−*ta2*→ (*x2″*, *m2′*); ¬ *τmove2* (*x2′*, *m2*) *ta2* (*x2″*, *m2′*);
t ⊢ (*x1″*, *m1′*) ≈ (*x2″*, *m2′*); *ta-bisim bisim ta1 ta2*;
Suspend w ∈ *set* $\{\!|ta1|\!\}_w$; *Suspend w* ∈ *set* $\{\!|ta2|\!\}_w$ ⟧
⟹ *x1″ ≈w x2″*
and *simulation-Wakeup1*:

⟦ t ⊢ (x1, m1) ≈ (x2, m2); x1 ≈w x2; t ⊢ (x1, m1) −1−ta1→ (x1′, m1′); Notified ∈ set ⦃ta1⦄w ∨ WokenUp ∈ set ⦃ta1⦄w ⟧
⟹ ∃ ta2 x2′ m2′. t ⊢ (x2, m2) −2−ta2→ (x2′, m2′) ∧ t ⊢ (x1′, m1′) ≈ (x2′, m2′) ∧ ta-bisim bisim ta1 ta2
and *simulation-Wakeup2*:
⟦ t ⊢ (x1, m1) ≈ (x2, m2); x1 ≈w x2; t ⊢ (x2, m2) −2−ta2→ (x2′, m2′); Notified ∈ set ⦃ta2⦄w ∨ WokenUp ∈ set ⦃ta2⦄w ⟧
⟹ ∃ ta1 x1′ m1′. t ⊢ (x1, m1) −1−ta1→ (x1′, m1′) ∧ t ⊢ (x1′, m1′) ≈ (x2′, m2′) ∧ ta-bisim bisim ta1 ta2
and *ex-final1-conv-ex-final2*:
(∃ x1. final1 x1) ⟷ (∃ x2. final2 x2)

sublocale *FWdelay-bisimulation-obs* <
delay-bisimulation-obs r1 t r2 t bisim t ta-bisim bisim τmove1 τmove2 **for** *t*
⟨*proof*⟩

context *FWdelay-bisimulation-obs* **begin**

lemma *FWdelay-bisimulation-obs-flip*:
FWdelay-bisimulation-obs final2 r2 final1 r1 (λt. flip (bisim t)) (flip bisim-wait) τmove2 τmove1
⟨*proof*⟩

end

lemma *FWdelay-bisimulation-obs-flip-simps* [*flip-simps*]:
FWdelay-bisimulation-obs final2 r2 final1 r1 (λt. flip (bisim t)) (flip bisim-wait) τmove2 τmove1 =

FWdelay-bisimulation-obs final1 r1 final2 r2 bisim bisim-wait τmove1 τmove2
⟨*proof*⟩

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 $\tau 2$: $\neg$ *τmove2* (*x2*, *shr s2*) *ta2* (*x2'*, *m2'*)
and *tbisim*: $\bigwedge t'$. $t \neq t' \Longrightarrow$ *tbisim* (*wset s1 t'* = *None*) *t'* (*thr s1 t'*) (*shr s1*) (*thr s2 t'*) (*shr s2*)
shows *s1'* ≈*m* *s2'*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

context *FWdelay-bisimulation-diverge* **begin**

lemma *FWdelay-bisimulation-diverge-flip*:
FWdelay-bisimulation-diverge final2 r2 final1 r1 (λ*t*. *flip* (*bisim t*)) (*flip bisim-wait*) *τmove2 τmove1*
⟨*proof*⟩

end

lemma *FWdelay-bisimulation-diverge-flip-simps* [*flip-simps*]:
FWdelay-bisimulation-diverge final2 r2 final1 r1 (λ*t*. *flip* (*bisim t*)) (*flip bisim-wait*) *τmove2 τmove1*
=
FWdelay-bisimulation-diverge final1 r1 final2 r2 bisim bisim-wait τmove1 τmove2
⟨*proof*⟩

context *FWdelay-bisimulation-diverge* **begin**

lemma *bisim-inv1*:
assumes *bisim*: *t* ⊢ *s1* ≈ *s2*
and *red*: *t* ⊢ *s1* −*1*−*ta1*→ *s1'*
obtains *s2'* **where** *t* ⊢ *s1'* ≈ *s2'*
⟨*proof*⟩

lemma *bisim-inv2*:
assumes *t* ⊢ *s1* ≈ *s2* *t* ⊢ *s2* −*2*−*ta2*→ *s2'*

obtains *s1'* **where** $t \vdash s1' \approx s2'$
⟨*proof*⟩

lemma *bisim-inv*: *bisim-inv*
⟨*proof*⟩

lemma *bisim-inv-τs1*:
assumes $t \vdash s1 \approx s2$ **and** *r1.silent-moves t s1 s1'*
obtains *s2'* **where** $t \vdash s1' \approx s2'$
⟨*proof*⟩

lemma *bisim-inv-τs2*:
assumes $t \vdash s1 \approx s2$ **and** *r2.silent-moves t s2 s2'*
obtains *s1'* **where** $t \vdash s1' \approx s2'$
⟨*proof*⟩

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 $thr\ s1\ t = \lfloor(x1, no\text{-}wait\text{-}locks)\rfloor$ $wset\ s1\ t = None$
shows *τmRed1 s1 (redT-upd-ε s1 t x1' m1')*
⟨*proof*⟩

lemma *red2-rtrancl-τ-into-RedT-τ*:
assumes *r2.silent-moves t (x2, shr s2) (x2', m2')*
and $t \vdash (x1, m1) \approx (x2, shr\ s2)$ $thr\ s2\ t = \lfloor(x2, no\text{-}wait\text{-}locks)\rfloor$ $wset\ s2\ t = None$
shows *τmRed2 s2 (redT-upd-ε s2 t x2' m2')*
⟨*proof*⟩

lemma *red1-rtrancl-τ-heapD*:
⟦ *r1.silent-moves t s1 s1'*; $t \vdash s1 \approx s2$ ⟧ $\Longrightarrow$ *snd s1' = snd s1*
⟨*proof*⟩

lemma *red2-rtrancl-τ-heapD*:
⟦ *r2.silent-moves t s2 s2'*; $t \vdash s1 \approx s2$ ⟧ $\Longrightarrow$ *snd s2' = snd s2*
⟨*proof*⟩

lemma *mbisim-simulation-silent1*:
assumes *mτ'*: *r1.mthr.silent-move s1 s1'* **and** *mbisim*: $s1 \approx m\ s2$
shows $\exists s2'.$ *r2.mthr.silent-moves s2 s2'* $\wedge\ s1' \approx m\ s2'$
⟨*proof*⟩

lemma *mbisim-simulation-silent2*:
⟦ *mbisim s1 s2*; *r2.mthr.silent-move s2 s2'* ⟧
$\Longrightarrow \exists s1'.$ *r1.mthr.silent-moves s1 s1'* $\wedge$ *mbisim s1' s2'*
⟨*proof*⟩

lemma *mbisim-simulation1'*:
assumes *mbisim*: *mbisim s1 s2* **and** ¬ *mτmove1 s1 tl1 s1' r1.redT s1 tl1 s1'*
shows $\exists s2'\ s2''\ tl2.$ *r2.mthr.silent-moves s2 s2'* $\wedge$ *r2.redT s2' tl2 s2''* $\wedge$
¬ *mτmove2 s2' tl2 s2''* $\wedge$ *mbisim s1' s2''* $\wedge$ *mta-bisim tl1 tl2*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *mτdiverge-simulation1*:
assumes *s1* ≈*m s2*
and *r1.mthr.τdiverge s1*
shows *r2.mthr.τdiverge s2*
⟨*proof*⟩

lemma *τdiverge-mbisim-inv*:
s1 ≈*m s2* ⟹ *r1.mthr.τdiverge s1* ⟷ *r2.mthr.τdiverge s2*
⟨*proof*⟩

lemma *mbisim-delay-bisimulation*:
delay-bisimulation-diverge r1.redT r2.redT mbisim mta-bisim mτmove1 mτmove2
⟨*proof*⟩

theorem *mdelay-bisimulation-final-base*:
delay-bisimulation-final-base r1.redT r2.redT mbisim mτmove1 mτmove2 r1.mfinal r2.mfinal
⟨*proof*⟩

end

sublocale *FWdelay-bisimulation-diverge* < *mthr*: *delay-bisimulation-diverge r1.redT r2.redT mbisim mta-bisim mτmove1 mτmove2*
⟨*proof*⟩

sublocale *FWdelay-bisimulation-diverge* <
mthr: *delay-bisimulation-final-base r1.redT r2.redT mbisim mta-bisim mτmove1 mτmove2 r1.mfinal r2.mfinal*
⟨*proof*⟩

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
⟨*proof*⟩

end

sublocale *FWdelay-bisimulation-diverge* <
mthr: *delay-bisimulation-diverge-final r1.redT r2.redT mbisim mta-bisim mτmove1 mτmove2 r1.mfinal r2.mfinal*
⟨*proof*⟩

1.18.3 Strong bisimulation as corollary

locale *FWbisimulation* = *FWbisimulation-base - - - r2 convert-RA bisim λx1 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* :: $'t \Rightarrow ('x1 \times 'm1, 'x2 \times 'm2)$ *bisim* (‹- ⊢ -/ ≈ -› [*50*, *50*, *50*] *60*) +
assumes *bisimulation-locale*: *bisimulation* (*r1 t*) (*r2 t*) (*bisim t*) (*ta-bisim bisim*)
and *bisim-final*: $t \vdash (x1, m1) \approx (x2, m2) \Longrightarrow$ *final1 x1* $\longleftrightarrow$ *final2 x2*
and *bisim-inv-red-other*:
⟦ $t' \vdash (x, m1) \approx (xx, m2)$; $t \vdash (x1, m1) \approx (x2, m2)$;
$t \vdash (x1, m1)$ −*1*−*ta1*→ $(x1', m1')$; $t \vdash (x2, m2)$ −*2*−*ta2*→ $(x2', m2')$;
$t \vdash (x1', m1') \approx (x2', m2')$; *ta-bisim bisim ta1 ta2* ⟧
$\Longrightarrow t' \vdash (x, m1') \approx (xx, m2')$
and *ex-final1-conv-ex-final2*:
($\exists$ *x1*. *final1 x1*) $\longleftrightarrow$ ($\exists$ *x2*. *final2 x2*)

sublocale *FWbisimulation* < *bisim?*: *bisimulation r1 t r2 t bisim t ta-bisim bisim* **for** *t*
⟨*proof*⟩

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*⟩

context *FWbisimulation* **begin**

lemma *FWbisimulation-flip*: *FWbisimulation final2 r2 final1 r1* (λ*t*. *flip* (*bisim t*))
⟨*proof*⟩

end

lemma *FWbisimulation-flip-simps* [*flip-simps*]:
FWbisimulation final2 r2 final1 r1 (λ*t*. *flip* (*bisim t*)) = *FWbisimulation final1 r1 final2 r2 bisim*
⟨*proof*⟩

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*⟩

lemma *mbisim-wset-eq*:
s1 ≈*m* *s2* $\Longrightarrow$ *wset s1* = *wset s2*
⟨*proof*⟩

lemma *mbisim-mfinal*:
s1 ≈*m* *s2* $\Longrightarrow$ *r1.mfinal s1* $\longleftrightarrow$ *r2.mfinal s2*
⟨*proof*⟩

end

sublocale *FWbisimulation* < *mthr*: *bisimulation r1.redT r2.redT mbisim mta-bisim*
⟨*proof*⟩

sublocale *FWbisimulation* < *mthr*: *bisimulation-final r1.redT r2.redT mbisim mta-bisim r1.mfinal r2.mfinal*
⟨*proof*⟩

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*⟩

lemma *actions-ok2-ex-actions-ok1*:
assumes *r2.actions-ok s2 t ta2*
and *ta1 ∼m ta2*
obtains *s1* **where** *r1.actions-ok s1 t ta1*
⟨*proof*⟩

lemma *ex-actions-ok1-conv-ex-actions-ok2*:
ta1 ∼m ta2 ⟹ (∃ *s1*. *r1.actions-ok s1 t ta1*) ⟷ (∃ *s2*. *r2.actions-ok s2 t ta2*)
⟨*proof*⟩

end

context *FWdelay-bisimulation-diverge* **begin**

lemma *no-τMove1-τs-to-no-τMove2*:
fixes *no-τmoves1 no-τmoves2*
defines *no-τmoves1* ≡ λ*s1 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* ≡ λ*s2 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 *mbisim*: *s1* ≈*m* (*ls2*, (*ts2*, *m2*), *ws2*, *is2*)

shows ∃ *ts2′*. *r2.mthr.silent-moves* (*ls2*, (*ts2*, *m2*), *ws2*, *is2*) (*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*)
⟨*proof*⟩

lemma *no-τMove2-τs-to-no-τMove1*:
fixes *no-τmoves1 no-τmoves2*
defines *no-τmoves1* ≡ λ*s1 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 ∃ *ts1′*. *r1.mthr.silent-moves* (*ls1*, (*ts1*, *m1*), *ws1*, *is1*) (*ls1*, (*ts1′*, *m1*), *ws1*, *is1*) ∧
(∀ *t*. *no-τmoves2 s2 t* ⟶ *no-τmoves1* (*ls1*, (*ts1′*, *m1*), *ws1*, *is1*) *t*) ∧ (*ls1*, (*ts1′*, *m1*), *ws1*, *is1*) ≈*m s2*
⟨*proof*⟩

lemma *deadlock-mbisim-not-final-thread-pres*:
assumes *dead*: *t* ∈ *r1.deadlocked s1* ∨ *r1.deadlock s1*
and *nfin*: *r1.not-final-thread s1 t*
and *fin*: *r1.final-thread s1 t* ⟹ *r2.final-thread s2 t*
and *mbisim*: *s1* ≈*m s2*
shows *r2.not-final-thread s2 t*
⟨*proof*⟩

lemma *deadlocked1-imp-τs-deadlocked2*:
assumes *mbisim*: *s1* ≈*m s2*
and *dead*: *t* ∈ *r1.deadlocked s1*
shows ∃ *s2′*. *r2.mthr.silent-moves s2 s2′* ∧ *t* ∈ *r2.deadlocked s2′* ∧ *s1* ≈*m s2′*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *deadlock2-imp-τs-deadlock1*:
⟦ *s1* ≈*m s2*; *r2.deadlock s2* ⟧
⟹ ∃ *s1′*. *r1.mthr.silent-moves s1 s1′* ∧ *r1.deadlock s1′* ∧ *s1′* ≈*m s2*
⟨*proof*⟩

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′*
⟨*proof*⟩

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*
⟨*proof*⟩

end

context *FWbisimulation* **begin**

lemma *mbisim-final-thread-preserve1*:

assumes *mbisim*: *s1* ≈*m s2* **and** *fin*: *r1.final-thread s1 t*
shows *r2.final-thread s2 t*
⟨*proof*⟩

lemma *mbisim-final-thread-preserve2*:
⟦ *s1* ≈*m s2*; *r2.final-thread s2 t* ⟧ ⟹ *r1.final-thread s1 t*
⟨*proof*⟩

lemma *mbisim-final-thread-inv*:
s1 ≈*m s2* ⟹ *r1.final-thread s1 t* ⟷ *r2.final-thread s2 t*
⟨*proof*⟩

lemma *mbisim-not-final-thread-inv*:
assumes *bisim*: *mbisim s1 s2*
shows *r1.not-final-thread s1* = *r2.not-final-thread s2*
⟨*proof*⟩

lemma *mbisim-deadlocked-preserve1*:
assumes *mbisim*: *s1* ≈*m s2* **and** *dead*: *t* ∈ *r1.deadlocked s1*
shows *t* ∈ *r2.deadlocked s2*
⟨*proof*⟩

lemma *mbisim-deadlocked-preserve2*:
⟦ *s1* ≈*m s2*; *t* ∈ *r2.deadlocked s2* ⟧ ⟹ *t* ∈ *r1.deadlocked s1*
⟨*proof*⟩

lemma *mbisim-deadlocked-inv*:
s1 ≈*m s2* ⟹ *r1.deadlocked s1* = *r2.deadlocked s2*
⟨*proof*⟩

lemma *mbisim-deadlocked′-inv*:
s1 ≈*m s2* ⟹ *r1.deadlocked′ s1* ⟷ *r2.deadlocked′ s2*
⟨*proof*⟩

lemma *mbisim-deadlock-inv*:
s1 ≈*m s2* ⟹ *r1.deadlock s1* = *r2.deadlock s2*
⟨*proof*⟩

end

context *FWbisimulation* **begin**

lemma *bisim-can-sync-preserve1*:
assumes *bisim*: *t* ⊢ (*x1*, *m1*) ≈ (*x2*, *m2*) **and** *cs*: *t* ⊢ ⟨*x1*, *m1*⟩ *LT* ≀*1*
shows *t* ⊢ ⟨*x2*, *m2*⟩ *LT* ≀*2*
⟨*proof*⟩

lemma *bisim-can-sync-preserve2*:
⟦ *t* ⊢ (*x1*, *m1*) ≈ (*x2*, *m2*); *t* ⊢ ⟨*x2*, *m2*⟩ *LT* ≀*2* ⟧ ⟹ *t* ⊢ ⟨*x1*, *m1*⟩ *LT* ≀*1*
⟨*proof*⟩

lemma *bisim-can-sync-inv*:

$t \vdash (x1, m1) \approx (x2, m2) \Longrightarrow t \vdash \langle x1, m1\rangle\ LT \wr 1 \longleftrightarrow t \vdash \langle x2, m2\rangle\ LT \wr 2$
⟨*proof*⟩

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*⟩

lemma *bisim-must-sync-preserve2*:
$[\![\ t \vdash (x1, m1) \approx (x2, m2);\ t \vdash \langle x2, m2\rangle \wr 2\ ]\!] \Longrightarrow t \vdash \langle x1, m1\rangle \wr 1$
⟨*proof*⟩

lemma *bisim-must-sync-inv*:
$t \vdash (x1, m1) \approx (x2, m2) \Longrightarrow t \vdash \langle x1, m1\rangle \wr 1 \longleftrightarrow t \vdash \langle x2, m2\rangle \wr 2$
⟨*proof*⟩

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*:
$[\![\ s\ -t\triangleright ta\rightarrow\ s';\ ts\text{-}inv\text{-}ok\ (thr\ s)\ I\ ]\!]$
$\Longrightarrow ts\text{-}inv\text{-}ok\ (thr\ s')\ (upd\text{-}invs\ I\ P\ \{\!|ta|\!\}_t)$
⟨*proof*⟩

lemma *RedT-preserves-ts-inv-ok*:
$[\![\ s\ -\triangleright ttas\rightarrow*\ s';\ ts\text{-}inv\text{-}ok\ (thr\ s)\ I\ ]\!]$
$\Longrightarrow ts\text{-}inv\text{-}ok\ (thr\ s')\ (upd\text{-}invs\ I\ Q\ (concat\ (map\ (thr\text{-}a \circ snd)\ ttas)))$
⟨*proof*⟩

lemma *redT-upd-inv-ext*:
fixes $I :: 't \rightharpoonup 'i$
shows $[\![\ s\ -t\triangleright ta\rightarrow\ s';\ ts\text{-}inv\text{-}ok\ (thr\ s)\ I\ ]\!] \Longrightarrow I \subseteq_m upd\text{-}invs\ I\ P\ \{\!|ta|\!\}_t$
⟨*proof*⟩

lemma *RedT-upd-inv-ext*:
fixes $I :: 't \rightharpoonup 'i$
shows $[\![\ s\ -\triangleright ttas\rightarrow*\ s';\ ts\text{-}inv\text{-}ok\ (thr\ s)\ I\ ]\!]$
$\Longrightarrow I \subseteq_m upd\text{-}invs\ I\ P\ (concat\ (map\ (thr\text{-}a \circ snd)\ ttas))$
⟨*proof*⟩

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* ⊢ ⟨*x*, *m*⟩ −*ta*→ ⟨*x'*, *m'*⟩; *P i t x m* ⟧ ⟹ *P i t x' m'*
  and *invariant-NewThread*: ⟦ *t* ⊢ ⟨*x*, *m*⟩ −*ta*→ ⟨*x'*, *m'*⟩; *P i t x m*; *NewThread t'' x'' m'* ∈ *set* ⦃*ta*⦄$_t$
⟧
  ⟹ ∃ *i''*. *P i'' t'' x'' m'*
  and *invariant-other*: ⟦ *t* ⊢ ⟨*x*, *m*⟩ −*ta*→ ⟨*x'*, *m'*⟩; *P i t x m*; *P i'' t'' x'' m* ⟧ ⟹ *P i'' t'' x'' m'*
begin

lemma *redT-updTs-invariant*:
  fixes *ln*
  assumes *tsiP*: *ts-inv P I ts m*
  and *red*: *t* ⊢ ⟨*x*, *m*⟩ −*ta*→ ⟨*x'*, *m'*⟩
  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*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

lemma *invariant3p-ts-inv*: *invariant3p redT* {*s*. ∃ *I*. *ts-inv P I* (*thr s*) (*shr s*)}
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *redT-updTs-preserves*:
  fixes *ln*
  assumes *esokQ*: *ts-ok P ts m*
  and *red*: $t \vdash \langle x,\ m\rangle\ -ta\rightarrow\ \langle x',\ m'\rangle$
  and *ts t* $= \lfloor(x,\ ln)\rfloor$
  and *thread-oks ts* $\{\!|ta|\!\}_t$
  shows *ts-ok P* $((redT\text{-}updTs\ ts\ \{\!|ta|\!\}_t)(t \mapsto (x',\ ln')))\ m'$
⟨*proof*⟩

theorem *redT-preserves*:
  assumes *redT*: $s\ -t\triangleright ta\rightarrow\ s'$
  and *esokQ*: *ts-ok P* (*thr s*) (*shr s*)
  shows *ts-ok P* (*thr* s') (*shr* s')
⟨*proof*⟩

theorem *RedT-preserves*:
  $[\![\ s\ -\triangleright ttas\rightarrow *\ s';\ ts\text{-}ok\ P\ (thr\ s)\ (shr\ s)\ ]\!] \Longrightarrow ts\text{-}ok\ P\ (thr\ s')\ (shr\ s')$
⟨*proof*⟩

lemma *invariant3p-ts-ok*: *invariant3p redT* {*s. ts-ok P* (*thr s*) (*shr s*)}
⟨*proof*⟩

end

lemma *lifting-wf-Const* [*intro*!]:
  assumes *multithreaded final r*
  shows *lifting-wf final r* ($\lambda t\ x\ m.\ k$)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *convert-TA-initial-inject* [*simp*]:
  convert-TA-initial ta = *convert-TA-initial ta'* ⟷ *ta* = *ta'*

⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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*
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *init-fin-τdivergeD*:
assumes *τtrsys.τdiverge* (*init-fin t*) *init-fin-τmove* ((*status*, *x*), *m*)
shows *τdiverge t* (*x*, *m*) ∧ *status* = *Running*

⟨*proof*⟩

lemma *init-fin-τdiverge-RunningI*:
assumes *τdiverge t* (*x*, *m*)
shows *τtrsys.τdiverge* (*init-fin t*) *init-fin-τmove* ((*Running*, *x*), *m*)
⟨*proof*⟩

lemma *init-fin-τdiverge-conv*:
τtrsys.τdiverge (*init-fin t*) *init-fin-τmove* ((*status*, *x*), *m*) ⟷
τdiverge t (*x*, *m*) ∧ *status* = *Running*
⟨*proof*⟩

end

lemma *init-fin-τmoves-False*:
τmultithreaded.init-fin-τmove (λ- - -. *False*) = (λ- - -. *False*)
⟨*proof*⟩

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
⟨*proof*⟩

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 *τmove* :: (*'l*,*'t*,*'x*,*'m*,*'w*,*'o*) *τmoves*

sublocale *if-τmultithreaded-wf* < *if-multithreaded*
⟨*proof*⟩

sublocale *if-τmultithreaded-wf* < *if-τmultithreaded* ⟨*proof*⟩

context *τmultithreaded-wf* **begin**

lemma *τmultithreaded-wf-init-fin*:
τmultithreaded-wf init-fin-final init-fin init-fin-τmove
⟨*proof*⟩

end

sublocale *if-τmultithreaded-wf* < *if*: *τmultithreaded-wf*
init-fin-final
init-fin
map NormalAction ∘ *convert-RA*

init-fin-τmove
⟨*proof*⟩

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*⟩

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* ⇒ *'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
⟨*proof*⟩

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*⟩

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* ⇒ *'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

⟨*proof*⟩

lemma (**in** *if-lifting-wf*) *if-lifting-inv*:
if-lifting-inv final r (λ-::*unit*. *P*)
⟨*proof*⟩

locale *τlifting-inv* = *τmultithreaded-wf* +
lifting-inv +
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*
and *P* :: *'i* ⇒ *'t* ⇒ *'x* ⇒ *'m* ⇒ *bool*
begin

lemma *redT-silent-move-invariant*:
⟦ *τmredT s s'*; *ts-inv P Is* (*thr s*) (*shr s*) ⟧ ⟹ *ts-inv P Is* (*thr s'*) (*shr s'*)
⟨*proof*⟩

lemma *redT-silent-moves-invariant*:
⟦ *mthr.silent-moves s s'*; *ts-inv P Is* (*thr s*) (*shr s*) ⟧ ⟹ *ts-inv P Is* (*thr s'*) (*shr s'*)
⟨*proof*⟩

lemma *redT-τrtrancl3p-invariant*:
⟦ *mthr.τrtrancl3p s ttas s'*; *ts-inv P Is* (*thr s*) (*shr s*) ⟧
⟹ *ts-inv P* (*upd-invs Is P* (*concat* (*map* (*thr-a* ∘ *snd*) *ttas*))) (*thr s'*) (*shr s'*)
⟨*proof*⟩

end

locale *τlifting-wf* = *τmultithreaded* +
lifting-wf +
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*
and *P* :: *'t* ⇒ *'x* ⇒ *'m* ⇒ *bool*
begin

lemma *redT-silent-move-preserves*:
⟦ *τmredT s s'*; *ts-ok P* (*thr s*) (*shr s*) ⟧ ⟹ *ts-ok P* (*thr s'*) (*shr s'*)
⟨*proof*⟩

lemma *redT-silent-moves-preserves*:
⟦ *mthr.silent-moves s s'*; *ts-ok P* (*thr s*) (*shr s*) ⟧ ⟹ *ts-ok P* (*thr s'*) (*shr s'*)
⟨*proof*⟩

lemma *redT-τrtrancl3p-preserves*:
⟦ *mthr.τrtrancl3p s ttas s'*; *ts-ok P* (*thr s*) (*shr s*) ⟧ ⟹ *ts-ok P* (*thr s'*) (*shr s'*)
⟨*proof*⟩

end

definition *init-fin-lift-state* :: *status* ⇒ (*'l*,*'t*,*'x*,*'m*,*'w*) *state* ⇒ (*'l*,*'t*,*status* × *'x*,*'m*,*'w*) *state*

where *init-fin-lift-state s σ* = (*locks σ*, (*λt. 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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *thr-init-fin-list-state'*:
thr (*init-fin-lift-state s σ*) = *map-option* (*λ(x, ln).* ((*s, x*), *ln*)) ∘ *thr σ*
⟨*proof*⟩

lemma *init-fin-descend-thr-Some-conv* [*simp*]:
⋀*ln. ts t* = ⌊((*status, x*), *ln*)⌋ ⟹ *init-fin-descend-thr ts t* = ⌊(*x, ln*)⌋
⟨*proof*⟩

lemma *init-fin-descend-thr-None-conv* [*simp*]:
ts t = *None* ⟹ *init-fin-descend-thr ts t* = *None*
⟨*proof*⟩

lemma *init-fin-descend-thr-eq-None* [*simp*]:
init-fin-descend-thr ts t = *None* ⟷ *ts t* = *None*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *ts-ok-init-fin-descend-state*:
ts-ok P (*init-fin-descend-thr ts*) = *ts-ok* (*init-fin-lift P*) *ts*
⟨*proof*⟩

lemma *free-thread-id-init-fin-descend-thr* [*simp*]:
free-thread-id (*init-fin-descend-thr ts*) = *free-thread-id ts*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *thread-ok-init-fin-descend-thr* [*simp*]:
thread-ok (*init-fin-descend-thr ts*) *nta* = *thread-ok ts nta*
⟨*proof*⟩

lemma *threads-ok-init-fin-descend-thr* [*simp*]:
thread-oks (*init-fin-descend-thr ts*) *ntas* = *thread-oks ts ntas*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *set-lift-start-obs* [*simp*]:

set (*lift-start-obs t obs*) =
insert (*t*, *InitialThreadAction*) ((*Pair t* ∘ *NormalAction*) ' *set obs*)
⟨*proof*⟩

lemma *distinct-lift-start-obs* [*simp*]: *distinct* (*lift-start-obs t obs*) = *distinct obs*
⟨*proof*⟩

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*)
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *ta-bisim-init-fin-bisim-InitialThreadAction* [*simp*]:
ta-bisim init-fin-bisim ⦃*InitialThreadAction*⦄ ⦃*InitialThreadAction*⦄
⟨*proof*⟩

lemma *ta-bisim-init-fin-bisim-ThreadFinishAction* [*simp*]:
ta-bisim init-fin-bisim ⦃*ThreadFinishAction*⦄ ⦃*ThreadFinishAction*⦄
⟨*proof*⟩

lemma *init-fin-bisim-wait-simps* [*simp*]:
(*status1*, *x1*) ≈*iw* (*status2*, *x2*) ⟷ *status1* = *Running* ∧ *status2* = *Running* ∧ *x1* ≈*w* *x2*
⟨*proof*⟩

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'*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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.* (*τtrsys.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*⟩

lemma *init-fin-simulation2*:
⟦ $t \vdash s1 \approx i\ s2$; *r2.init-fin t s2 tl2 s2'*; $\neg$ *r2.init-fin-τmove s2 tl2 s2'* ⟧
$\Longrightarrow$ $\exists$ *s1' s1'' tl1.* (*τtrsys.silent-move* (*r1.init-fin t*) *r1.init-fin-τmove*)** *s1 s1'* $\wedge$
r1.init-fin t s1' tl1 s1'' $\wedge$ $\neg$ *r1.init-fin-τmove s1' tl1 s1''* $\wedge$
$t \vdash s1'' \approx i\ s2'$ $\wedge$ *ta-bisim init-fin-bisim tl1 tl2*
⟨*proof*⟩

lemma *init-fin-simulation-Wakeup1*:
assumes *bisim*: $t \vdash (sx1, m1) \approx i\ (sx2, m2)$
and *wait*: $sx1 \approx iw\ sx2$
and *red1*: *r1.init-fin t* (*sx1*, *m1*) *ta1* (*sx1'*, *m1'*)
and *wakeup*: *Notified* $\in$ *set* $\{\!|ta1|\!\}_w$ $\vee$ *WokenUp* $\in$ *set* $\{\!|ta1|\!\}_w$
shows $\exists$ *ta2 sx2' m2'.* *r2.init-fin t* (*sx2*, *m2*) *ta2* (*sx2'*, *m2'*) $\wedge$ $t \vdash (sx1', m1') \approx i\ (sx2', m2')$ $\wedge$
ta-bisim init-fin-bisim ta1 ta2
⟨*proof*⟩

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
⟨*proof*⟩

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-τmove r2.init-fin-τmove
⟨*proof*⟩

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-τmove r2.init-fin-τmove
⟨*proof*⟩

end

context *FWdelay-bisimulation-diverge* **begin**

lemma *init-fin-simulation-silent1*:
⟦ $t \vdash sxm1 \approx i\ sxm2$; *τtrsys.silent-move* (*r1.init-fin t*) *r1.init-fin-τmove sxm1 sxm1'* ⟧
$\Longrightarrow$ $\exists$ *sxm2'.* *τtrsys.silent-moves* (*r2.init-fin t*) *r2.init-fin-τmove sxm2 sxm2'* $\wedge$ $t \vdash sxm1' \approx i\ sxm2'$
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *init-fin-τdiverge-bisim-inv*:
t ⊢ *sxm1* ≈*i* *sxm2*
⟹ *τtrsys.τdiverge* (*r1.init-fin t*) *r1.init-fin-τmove sxm1* =
τtrsys.τdiverge (*r2.init-fin t*) *r2.init-fin-τmove sxm2*
⟨*proof*⟩

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
⟨*proof*⟩

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-wa
r1.init-fin-τmove r2.init-fin-τmove
⟨*proof*⟩

end

context *FWbisimulation* **begin**

lemma *init-fin-simulation1*:
assumes *t* ⊢ *s1* ≈*i* *s2* **and** *r1.init-fin t s1 tl1 s1'*
shows ∃ *s2' tl2*. *r2.init-fin t s2 tl2 s2'* ∧ *t* ⊢ *s1'* ≈*i* *s2'* ∧ *ta-bisim init-fin-bisim tl1 tl2*
⟨*proof*⟩

lemma *init-fin-simulation2*:
⟦ *t* ⊢ *s1* ≈*i* *s2*; *r2.init-fin t s2 tl2 s2'* ⟧
⟹ ∃ *s1' tl1*. *r1.init-fin t s1 tl1 s1'* ∧ *t* ⊢ *s1'* ≈*i* *s2'* ∧ *ta-bisim init-fin-bisim tl1 tl2*
⟨*proof*⟩

lemma *init-fin-bisimulation*:
bisimulation (*r1.init-fin t*) (*r2.init-fin t*) (*init-fin-bisim t*) (*ta-bisim init-fin-bisim*)
⟨*proof*⟩

lemma *init-fin-FWbisimulation*:
FWbisimulation r1.init-fin-final r1.init-fin r2.init-fin-final r2.init-fin init-fin-bisim
⟨*proof*⟩

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 $x \sqsubseteq_r y == x \sqsubseteq_r y$

abbreviation (*input*)
  lesssub1 :: *'a* ⇒ *'a ord* ⇒ *'a* ⇒ *bool* (‹(- /⊏- -)› [*50*, *1000*, *51*] *50*)
  where $x \sqsubset_r y == x \sqsubset_r y$

abbreviation (*input*)

plussub1 :: $'a \Rightarrow ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b \Rightarrow 'c$ (‹(- /⊔- -)› [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* $r \Longrightarrow x \sqsubseteq_r x$
⟨*proof*⟩
lemma *order-antisym*: ⟦ *order* r; $x \sqsubseteq_r y$; $y \sqsubseteq_r x$ ⟧ $\Longrightarrow x = y$
⟨*proof*⟩
lemma *order-trans*: ⟦ *order* r; $x \sqsubseteq_r y$; $y \sqsubseteq_r z$ ⟧ $\Longrightarrow x \sqsubseteq_r z$

⟨*proof*⟩
lemma *order-less-irrefl* [*intro*, *simp*]: *order* $r \Longrightarrow \neg\ x \sqsubset_r x$
⟨*proof*⟩
lemma *order-less-trans*: $[\![$ *order* r; $x \sqsubset_r y$; $y \sqsubset_r z$ $]\!] \Longrightarrow x \sqsubset_r z$
⟨*proof*⟩
lemma *topD* [*simp*, *intro*]: *top* $r\ T \Longrightarrow x \sqsubseteq_r T$
⟨*proof*⟩
lemma *top-le-conv* [*simp*]: $[\![$ *order* r; *top* $r\ T$ $]\!] \Longrightarrow (T \sqsubseteq_r x) = (x = T)$
⟨*proof*⟩
lemma *semilat-Def*:
semilat$(A,r,f) \longleftrightarrow$ *order* $r \wedge$ *closed* $A\ f \wedge$
$(\forall x{\in}A.\ \forall y{\in}A.\ x \sqsubseteq_r x \sqcup_f y) \wedge$
$(\forall x{\in}A.\ \forall y{\in}A.\ y \sqsubseteq_r x \sqcup_f y) \wedge$
$(\forall x{\in}A.\ \forall y{\in}A.\ \forall z{\in}A.\ x \sqsubseteq_r z \wedge y \sqsubseteq_r z \longrightarrow x \sqcup_f y \sqsubseteq_r z)$
⟨*proof*⟩
lemma (**in** *Semilat*) *orderI* [*simp*, *intro*]: *order* r
⟨*proof*⟩
lemma (**in** *Semilat*) *closedI* [*simp*, *intro*]: *closed* $A\ f$
⟨*proof*⟩
lemma *closedD*: $[\![$ *closed* $A\ f$; $x{\in}A$; $y{\in}A$ $]\!] \Longrightarrow x \sqcup_f y \in A$
⟨*proof*⟩
lemma *closed-UNIV* [*simp*]: *closed UNIV f*
⟨*proof*⟩
lemma (**in** *Semilat*) *closed-f* [*simp*, *intro*]: $[\![x \in A;\ y \in A]\!] \Longrightarrow x \sqcup_f y \in A$
⟨*proof*⟩
lemma (**in** *Semilat*) *refl-r* [*intro*, *simp*]: $x \sqsubseteq_r x$ ⟨*proof*⟩

lemma (**in** *Semilat*) *antisym-r* [*intro?*]: $[\![\ x \sqsubseteq_r y;\ y \sqsubseteq_r x\ ]\!] \Longrightarrow x = y$
⟨*proof*⟩
lemma (**in** *Semilat*) *trans-r* [*trans*, *intro?*]: $[\![x \sqsubseteq_r y;\ y \sqsubseteq_r z]\!] \Longrightarrow x \sqsubseteq_r z$
⟨*proof*⟩
lemma (**in** *Semilat*) *ub1* [*simp*, *intro?*]: $[\![\ x \in A;\ y \in A\ ]\!] \Longrightarrow x \sqsubseteq_r x \sqcup_f y$
⟨*proof*⟩
lemma (**in** *Semilat*) *ub2* [*simp*, *intro?*]: $[\![\ x \in A;\ y \in A\ ]\!] \Longrightarrow y \sqsubseteq_r x \sqcup_f y$
⟨*proof*⟩
lemma (**in** *Semilat*) *lub* [*simp*, *intro?*]:
$[\![\ x \sqsubseteq_r z;\ y \sqsubseteq_r z;\ x \in A;\ y \in A;\ z \in A\ ]\!] \Longrightarrow x \sqcup_f y \sqsubseteq_r z$
⟨*proof*⟩
lemma (**in** *Semilat*) *plus-le-conv* [*simp*]:
$[\![\ x \in A;\ y \in A;\ z \in A\ ]\!] \Longrightarrow (x \sqcup_f y \sqsubseteq_r z) = (x \sqsubseteq_r z \wedge y \sqsubseteq_r z)$
⟨*proof*⟩
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** $?P \longleftrightarrow ?Q$)⟨*proof*⟩
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** $?P \longleftrightarrow ?Q$)⟨*proof*⟩
lemma (**in** *Semilat*) *plus-assoc* [*simp*]:
assumes a: $a \in A$ **and** b: $b \in A$ **and** c: $c \in A$
shows $a \sqcup_f (b \sqcup_f c) = a \sqcup_f b \sqcup_f c$⟨*proof*⟩
lemma (**in** *Semilat*) *plus-com-lemma*:
$[\![a \in A;\ b \in A]\!] \Longrightarrow a \sqcup_f b \sqsubseteq_r b \sqcup_f a$⟨*proof*⟩
lemma (**in** *Semilat*) *plus-commutative*:
$[\![a \in A;\ b \in A]\!] \Longrightarrow a \sqcup_f b = b \sqcup_f a$

⟨*proof*⟩
lemma *is-lubD*:
$is\text{-}lub\ r\ x\ y\ u \Longrightarrow is\text{-}ub\ r\ x\ y\ u \wedge (\forall z.\ is\text{-}ub\ r\ x\ y\ z \longrightarrow (u,z) \in r)$
⟨*proof*⟩
lemma *is-ubI*:
$[\![\ (x,u) \in r;\ (y,u) \in r\ ]\!] \Longrightarrow is\text{-}ub\ r\ x\ y\ u$
⟨*proof*⟩
lemma *is-ubD*:
$is\text{-}ub\ r\ x\ y\ u \Longrightarrow (x,u) \in r \wedge (y,u) \in r$
⟨*proof*⟩

lemma *is-lub-bigger1* [*iff*]:
$is\text{-}lub\ (r\hat{}*\ )\ x\ y\ y = ((x,y) \in r\hat{}*\ )$⟨*proof*⟩
lemma *is-lub-bigger2* [*iff*]:
$is\text{-}lub\ (r\hat{}*\ )\ x\ y\ x = ((y,x) \in r\hat{}*\ )$⟨*proof*⟩
lemma *extend-lub*:
assumes *single-valued r*
and $is\text{-}lub\ (r^*)\ x\ y\ u$
and $(x',\ x) \in r$
shows $\exists v.\ is\text{-}lub\ (r^*)\ x'\ y\ v$⟨*proof*⟩
lemma *single-valued-has-lubs*:
assumes *single-valued r*
and *in-r*: $(x,\ u) \in r^*\ (y,\ u) \in r^*$
shows $\exists z.\ is\text{-}lub\ (r^*)\ x\ y\ z$⟨*proof*⟩
lemma *some-lub-conv*:
$[\![\ acyclic\ r;\ is\text{-}lub\ (r\hat{}*\ )\ x\ y\ u\ ]\!] \Longrightarrow some\text{-}lub\ (r\hat{}*\ )\ x\ y = u$⟨*proof*⟩
lemma *is-lub-some-lub*:
$[\![\ single\text{-}valued\ r;\ acyclic\ r;\ (x,u) \in r\hat{}*;\ (y,u) \in r\hat{}*\ ]\!]$
$\Longrightarrow is\text{-}lub\ (r\hat{}*\ )\ x\ y\ (some\text{-}lub\ (r\hat{}*\ )\ x\ y)$
⟨*proof*⟩

2.1.1 An executable lub-finder

definition $exec\text{-}lub :: ('a * 'a)\ set \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a\ binop$
where
$exec\text{-}lub\ r\ f\ x\ y = while\ (\lambda z.\ (x,z) \notin r^*)\ f\ y$

lemma *exec-lub-refl*: $exec\text{-}lub\ r\ f\ T\ T = T$
⟨*proof*⟩

lemma *acyclic-single-valued-finite*:
$[\![acyclic\ r;\ single\text{-}valued\ r;\ (x,y) \in r^*]\!]$
$\Longrightarrow finite\ (r \cap \{a.\ (x,\ a) \in r^*\} \times \{b.\ (b,\ y) \in r^*\})$⟨*proof*⟩

lemma *exec-lub-conv*:
$[\![\ acyclic\ r;\ \forall x\ y.\ (x,y) \in r \longrightarrow f\ x = y;\ is\text{-}lub\ (r^*)\ x\ y\ u\ ]\!] \Longrightarrow$
$exec\text{-}lub\ r\ f\ x\ y = u$⟨*proof*⟩
lemma *is-lub-exec-lub*:
$[\![\ single\text{-}valued\ r;\ acyclic\ r;\ (x,u){:}r\hat{}*;\ (y,u){:}r\hat{}*;\ \forall x\ y.\ (x,y) \in r \longrightarrow f\ x = y\ ]\!]$
$\Longrightarrow is\text{-}lub\ (r\hat{}*\ )\ x\ y\ (exec\text{-}lub\ r\ f\ x\ y)$
⟨*proof*⟩
end

2.2 The Error Type

theory *Err*
imports *Semilat*
begin

datatype $'a\ err = Err \mid OK\ 'a$

type-synonym $'a\ ebinop = 'a \Rightarrow 'a \Rightarrow 'a\ err$
type-synonym $'a\ esl = 'a\ set \times 'a\ ord \times 'a\ ebinop$

primrec $ok\text{-}val :: 'a\ err \Rightarrow 'a$
where
$ok\text{-}val\ (OK\ x) = x$

definition $lift :: ('a \Rightarrow 'b\ err) \Rightarrow ('a\ err \Rightarrow 'b\ err)$
where
$lift\ f\ e = (case\ e\ of\ Err \Rightarrow Err \mid OK\ x \Rightarrow f\ x)$

definition $lift2 :: ('a \Rightarrow 'b \Rightarrow 'c\ err) \Rightarrow 'a\ err \Rightarrow 'b\ err \Rightarrow 'c\ err$
where
$lift2\ f\ e_1\ e_2 =$
$(case\ e_1\ of\ Err\ \Rightarrow Err \mid OK\ x \Rightarrow (case\ e_2\ of\ Err \Rightarrow Err \mid OK\ y \Rightarrow f\ x\ y))$

definition $le :: 'a\ ord \Rightarrow 'a\ err\ ord$
where
$le\ r\ e_1\ e_2 =$
$(case\ e_2\ of\ Err \Rightarrow True \mid OK\ y \Rightarrow (case\ e_1\ of\ Err \Rightarrow False \mid OK\ x \Rightarrow x \sqsubseteq_r y))$

definition $sup :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a\ err \Rightarrow 'b\ err \Rightarrow 'c\ err)$
where
$sup\ f = lift2\ (\lambda x\ y.\ OK\ (x \sqcup_f y))$

definition $err :: 'a\ set \Rightarrow 'a\ err\ set$
where
$err\ A = insert\ Err\ \{OK\ x | x.\ x \in A\}$

definition $esl :: 'a\ sl \Rightarrow 'a\ esl$
where
$esl = (\lambda(A,r,f).\ (A,\ r,\ \lambda x\ y.\ OK(f\ x\ y)))$

definition $sl :: 'a\ esl \Rightarrow 'a\ err\ sl$
where
$sl = (\lambda(A,r,f).\ (err\ A,\ le\ r,\ lift2\ f))$

abbreviation
$err\text{-}semilat :: 'a\ esl \Rightarrow bool$ **where**
$err\text{-}semilat\ L == semilat(sl\ L)$

primrec $strict :: ('a \Rightarrow 'b\ err) \Rightarrow ('a\ err \Rightarrow 'b\ err)$
where
$strict\ f\ Err\ \ = Err$
$\mid strict\ f\ (OK\ x) = f\ x$

lemma *err-def'*:
err A = *insert Err* $\{x.\ \exists y{\in}A.\ x = OK\ y\}$⟨*proof*⟩
lemma *strict-Some* [*simp*]:
$(strict\ f\ x = OK\ y) = (\exists z.\ x = OK\ z \wedge f\ z = OK\ y)$⟨*proof*⟩
lemma *not-Err-eq*: $(x \neq Err) = (\exists a.\ x = OK\ a)$⟨*proof*⟩
lemma *not-OK-eq*: $(\forall y.\ x \neq OK\ y) = (x = Err)$⟨*proof*⟩
lemma *unfold-lesub-err*: $e1 \sqsubseteq_{le\ r} e2 = le\ r\ e1\ e2$⟨*proof*⟩
lemma *le-err-refl*: $\forall x.\ x \sqsubseteq_r x \Longrightarrow e \sqsubseteq_{le\ r} e$⟨*proof*⟩
lemma *le-err-trans* [*rule-format*]:
$order\ r \Longrightarrow e1 \sqsubseteq_{le\ r} e2 \longrightarrow e2 \sqsubseteq_{le\ r} e3 \longrightarrow e1 \sqsubseteq_{le\ r} e3$⟨*proof*⟩
lemma *le-err-antisym* [*rule-format*]:
$order\ r \Longrightarrow e1 \sqsubseteq_{le\ r} e2 \longrightarrow e2 \sqsubseteq_{le\ r} e1 \longrightarrow e1{=}e2$⟨*proof*⟩
lemma *OK-le-err-OK*: $(OK\ x \sqsubseteq_{le\ r} OK\ y) = (x \sqsubseteq_r y)$⟨*proof*⟩
lemma *order-le-err* [*iff*]: $order(le\ r) = order\ r$⟨*proof*⟩
lemma *le-Err* [*iff*]: $e \sqsubseteq_{le\ r} Err$⟨*proof*⟩
lemma *Err-le-conv* [*iff*]: $Err \sqsubseteq_{le\ r} e\ = (e = Err)$⟨*proof*⟩
lemma *le-OK-conv* [*iff*]: $e \sqsubseteq_{le\ r} OK\ x\ =\ (\exists y.\ e = OK\ y \wedge y \sqsubseteq_r x)$⟨*proof*⟩
lemma *OK-le-conv*: $OK\ x \sqsubseteq_{le\ r} e = (e = Err \vee (\exists y.\ e = OK\ y \wedge x \sqsubseteq_r y))$⟨*proof*⟩
lemma *top-Err* [*iff*]: $top\ (le\ r)\ Err$⟨*proof*⟩
lemma *OK-less-conv* [*rule-format*, *iff*]:
$OK\ x \sqsubset_{le\ r} e = (e{=}Err \vee (\exists y.\ e = OK\ y \wedge x \sqsubset_r y))$⟨*proof*⟩
lemma *not-Err-less* [*rule-format*, *iff*]: $\neg(Err \sqsubset_{le\ r} x)$⟨*proof*⟩
lemma *semilat-errI* [*intro*]: **assumes** *Semilat A r f*
shows $semilat(err\ A,\ le\ r,\ lift2(\lambda x\ y.\ OK(f\ x\ y)))$⟨*proof*⟩
lemma *err-semilat-eslI-aux*:
assumes *Semilat A r f* **shows** $err\text{-}semilat(esl(A,r,f))$⟨*proof*⟩
lemma *err-semilat-eslI* [*intro*, *simp*]:
$semilat\ L \Longrightarrow err\text{-}semilat\ (esl\ L)$⟨*proof*⟩
lemma *acc-err* [*simp*, *intro*!]: $acc\ A\ r \Longrightarrow acc\ (err\ A)\ (le\ r)$⟨*proof*⟩
lemma *Err-in-err* [*iff*]: $Err : err\ A$⟨*proof*⟩
lemma *Ok-in-err* [*iff*]: $(OK\ x \in err\ A) = (x{\in}A)$⟨*proof*⟩

2.2.1 lift

lemma *lift-in-errI*:
$[\![\ e \in err\ S;\ \forall x{\in}S.\ e = OK\ x \longrightarrow f\ x \in err\ S\ ]\!] \Longrightarrow lift\ f\ e \in err\ S$⟨*proof*⟩
lemma *Err-lift2* [*simp*]: $Err \sqcup_{lift2\ f} x = Err$⟨*proof*⟩
lemma *lift2-Err* [*simp*]: $x \sqcup_{lift2\ f} Err = Err$⟨*proof*⟩
lemma *OK-lift2-OK* [*simp*]: $OK\ x \sqcup_{lift2\ f} OK\ y = x \sqcup_f y$⟨*proof*⟩

2.2.2 sup

lemma *Err-sup-Err* [*simp*]: $Err \sqcup_{sup\ f} x = Err$⟨*proof*⟩
lemma *Err-sup-Err2* [*simp*]: $x \sqcup_{sup\ f} Err = Err$⟨*proof*⟩
lemma *Err-sup-OK* [*simp*]: $OK\ x \sqcup_{sup\ f} OK\ y = OK\ (x \sqcup_f y)$⟨*proof*⟩
lemma *Err-sup-eq-OK-conv* [*iff*]:
$(sup\ f\ ex\ ey = OK\ z) = (\exists x\ y.\ ex = OK\ x \wedge ey = OK\ y \wedge f\ x\ y = z)$⟨*proof*⟩
lemma *Err-sup-eq-Err* [*iff*]: $(sup\ f\ ex\ ey = Err) = (ex{=}Err \vee ey{=}Err)$⟨*proof*⟩

2.2.3 semilat (err A) (le r) f

lemma *semilat-le-err-Err-plus* [*simp*]:
$[\![\ x{\in}\ err\ A;\ semilat(err\ A,\ le\ r,\ f)\ ]\!] \Longrightarrow Err \sqcup_f x = Err$⟨*proof*⟩
lemma *semilat-le-err-plus-Err* [*simp*]:

$[\![x\in err\ A;\ semilat(err\ A,\ le\ r,\ f)\ ]\!] \Longrightarrow x \sqcup_f Err = Err\langle proof\rangle$

lemma *semilat-le-err-OK1*:
$[\![x\in A;\ y\in A;\ semilat(err\ A,\ le\ r,\ f);\ OK\ x \sqcup_f OK\ y = OK\ z\ ]\!]$
$\Longrightarrow x \sqsubseteq_r z\langle proof\rangle$

lemma *semilat-le-err-OK2*:
$[\![x\in A;\ y\in A;\ semilat(err\ A,\ le\ r,\ f);\ OK\ x \sqcup_f OK\ y = OK\ z\ ]\!]$
$\Longrightarrow y \sqsubseteq_r z\langle proof\rangle$

lemma *eq-order-le*:
$[\![x=y;\ order\ r\ ]\!] \Longrightarrow x \sqsubseteq_r y\langle proof\rangle$

lemma *OK-plus-OK-eq-Err-conv* [*simp*]:
assumes $x\in A\quad y\in A\quad semilat(err\ A,\ le\ r,\ fe)$
shows $(OK\ x \sqcup_{fe} OK\ y = Err) = (\neg(\exists z\in A.\ x \sqsubseteq_r z \wedge y \sqsubseteq_r z))\langle proof\rangle$

2.2.4 semilat (err(Union AS))

lemma *all-bex-swap-lemma* [*iff*]:
$(\forall x.\ (\exists y\in A.\ x = f\ y) \longrightarrow P\ x) = (\forall y\in A.\ P(f\ y))\langle proof\rangle$

lemma *closed-err-Union-lift2I*:
$[\![\forall A\in AS.\ closed\ (err\ A)\ (lift2\ f);\ AS \neq \{\};$
$\forall A\in AS.\forall B\in AS.\ A\neq B \longrightarrow (\forall a\in A.\forall b\in B.\ a \sqcup_f b = Err)\ ]\!]$
$\Longrightarrow closed\ (err(Union\ AS))\ (lift2\ f)\langle proof\rangle$

If $AS = \{\}$ the thm collapses to $order\ r \wedge closed\ \{Err\}\ f \wedge Err \sqcup_f Err = Err$ which may not hold

lemma *err-semilat-UnionI*:
$[\![\forall A\in AS.\ err\text{-}semilat(A,\ r,\ f);\ AS \neq \{\};$
$\forall A\in AS.\forall B\in AS.\ A\neq B \longrightarrow (\forall a\in A.\forall b\in B.\ \neg a \sqsubseteq_r b \wedge a \sqcup_f b = Err)\ ]\!]$
$\Longrightarrow err\text{-}semilat(Union\ AS,\ r,\ f)\langle proof\rangle$

end

2.3 More about Options

theory *Opt*
imports
Err
begin

definition $le :: 'a\ ord \Rightarrow 'a\ option\ ord$
where
$le\ r\ o_1\ o_2 =$
$(case\ o_2\ of\ None \Rightarrow o_1{=}None \mid Some\ y \Rightarrow (case\ o_1\ of\ None \Rightarrow True \mid Some\ x \Rightarrow x \sqsubseteq_r y))$

definition $opt :: 'a\ set \Rightarrow 'a\ option\ set$
where
$opt\ A = insert\ None\ \{Some\ y\ |y.\ y \in A\}$

definition $sup :: 'a\ ebinop \Rightarrow 'a\ option\ ebinop$
where
$sup\ f\ o_1\ o_2 =$
$(case\ o_1\ of\ None \Rightarrow OK\ o_2$
$\mid Some\ x \Rightarrow (case\ o_2\ of\ None \Rightarrow OK\ o_1$
$\mid Some\ y \Rightarrow (case\ f\ x\ y\ of\ Err \Rightarrow Err \mid OK\ z \Rightarrow OK\ (Some\ z))))$

definition $esl :: 'a\ esl \Rightarrow 'a\ option\ esl$

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* $\Rightarrow$ o_1=*None* |
Some y $\Rightarrow$ (*case* o_1 *of None* $\Rightarrow$ *True* | *Some x* $\Rightarrow$ $x \sqsubseteq_r y$))⟨*proof*⟩
lemma *le-opt-refl*: *order r* $\Longrightarrow$ $x \sqsubseteq_{le\ r} x$⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨p

2.4 Products as Semilattices

theory *Product*
imports *Err*
begin

definition *le* :: *'a ord* $\Rightarrow$ *'b ord* $\Rightarrow$ (*'a* $\times$ *'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* $\Rightarrow$ *'b ebinop* $\Rightarrow$ (*'a* $\times$ *'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* $\Rightarrow$ *'b esl* $\Rightarrow$ (*'a* $\times$ *'b*) *esl*
where
esl = (λ(*A*,r_A,f_A) (*B*,r_B,f_B). (*A* $\times$ *B*, *le* r_A r_B, *sup* f_A f_B))

abbreviation
lesubprod :: *'a* $\times$ *'b* $\Rightarrow$ (*'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool*) $\Rightarrow$ (*'b* $\Rightarrow$ *'b* $\Rightarrow$ *bool*) $\Rightarrow$ *'a* $\times$ *'b* $\Rightarrow$ *bool*
(‹(- /⊑'(-,-') -)› [*50*, *0*, *0*, *51*] *50*) **where**
p ⊑(*rA*,*rB*) *q* == $p \sqsubseteq_{Product.le\ rA\ rB}\ q$

lemma *unfold-lesub-prod*: *x* ⊑(r_A,r_B) *y* = *le* r_A r_B *x y*⟨*proof*⟩
lemma *le-prod-Pair-conv* [*iff*]: ((a_1,b_1) ⊑(r_A,r_B) (a_2,b_2)) = ($a_1 \sqsubseteq_{r_A} a_2$ & $b_1 \sqsubseteq_{r_B} b_2$)⟨*proof*⟩
lemma *less-prod-Pair-conv*:
((a_1,b_1) $\sqsubset_{Product.le\ r_A\ r_B}$ (a_2,b_2)) =
($a_1 \sqsubset_{r_A} a_2$ & $b_1 \sqsubseteq_{r_B} b_2$ | $a_1 \sqsubseteq_{r_A} a_2$ & $b_1 \sqsubset_{r_B} b_2$)⟨*proof*⟩
lemma *order-le-prod* [*iff*]: *order*(*Product.le* r_A r_B) = (*order* r_A & *order* r_B)⟨*proof*⟩

lemma *acc-le-prodI* [*intro*!]:
⟦ *acc A* r_A; *acc B* r_B ⟧ $\Longrightarrow$ *acc* (*A* $\times$ *B*) (*Product.le* r_A r_B)⟨*proof*⟩

lemma *closed-lift2-sup*:
⟦ *closed* (*err A*) (*lift2 f*); *closed* (*err B*) (*lift2 g*) ⟧ $\Longrightarrow$
closed (*err*(*A*$\times$*B*)) (*lift2*(*sup f g*))⟨*proof*⟩
lemma *unfold-plussub-lift2*: $e_1 \sqcup_{lift2\ f} e_2$ = *lift2 f* e_1 e_2⟨*proof*⟩

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*) = ($\neg$($\exists$ *z*∈*A*. $x \sqsubseteq_r z \wedge y \sqsubseteq_r z$))⟨*proof*⟩
lemma *err-semilat-Product-esl*:
$\bigwedge L_1\ L_2$. ⟦ *err-semilat* L_1; *err-semilat* L_2 ⟧ $\Longrightarrow$ *err-semilat*(*Product.esl* L_1 L_2)⟨*proof*⟩

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*. $x \sqsubseteq_r y$)

abbreviation
  lesublist :: *'a list* ⇒ *'a ord* ⇒ *'a list* ⇒ *bool* (‹(- /[⊑-] -)› [*50*, *0*, *51*] *50*) **where**
  x [$\sqsubseteq_r$] *y* == *x* <=-(*Listn.le r*) *y*

abbreviation
  lesssublist :: *'a list* ⇒ *'a ord* ⇒ *'a list* ⇒ *bool* (‹(- /[⊏-] -)› [*50*, *0*, *51*] *50*) **where**
  x [$\sqsubset_r$] *y* == *x* <-(*Listn.le r*) *y*

abbreviation
  plussublist :: *'a list* ⇒ (*'a* ⇒ *'b* ⇒ *'c*) ⇒ *'b list* ⇒ *'c list*
    (‹(- /[⊔-] -)› [*65*, *0*, *66*] *65*) **where**
  x [$\sqcup_f$] *y* == *x* $\sqcup_{map2\ f}$ *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* [$\sqcup_f$] *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* [$\sqsubseteq_r$] *ys* = *Listn.le r xs ys*⟨*proof*⟩
lemma *Nil-le-conv* [*iff*]: ([] [$\sqsubseteq_r$] *ys*) = (*ys* = [])⟨*proof*⟩
lemma *Cons-notle-Nil* [*iff*]: ¬ *x*#*xs* [$\sqsubseteq_r$] []⟨*proof*⟩

lemma *Cons-le-Cons* [*iff*]: $x\#xs\ [\sqsubseteq_r]\ y\#ys = (x \sqsubseteq_r y \wedge xs\ [\sqsubseteq_r]\ ys)\langle proof\rangle$
lemma *Cons-less-Conss* [*simp*]:
$order\ r \Longrightarrow\ x\#xs\ [\sqsubset_r]\ y\#ys = (x \sqsubset_r y \wedge xs\ [\sqsubseteq_r]\ ys \vee x = y \wedge xs\ [\sqsubset_r]\ ys)\langle proof\rangle$
lemma *list-update-le-cong*:
$[\![\ i<size\ xs;\ xs\ [\sqsubseteq_r]\ ys;\ x \sqsubseteq_r y\ ]\!] \Longrightarrow xs[i:=x]\ [\sqsubseteq_r]\ ys[i:=y]\langle proof\rangle$

lemma *le-listD*: $[\![\ xs\ [\sqsubseteq_r]\ ys;\ p < size\ xs\ ]\!] \Longrightarrow xs!p \sqsubseteq_r ys!p\langle proof\rangle$
lemma *le-list-refl*: $\forall x.\ x \sqsubseteq_r x \Longrightarrow xs\ [\sqsubseteq_r]\ xs\langle proof\rangle$
lemma *le-list-trans*: $[\![\ order\ r;\ xs\ [\sqsubseteq_r]\ ys;\ ys\ [\sqsubseteq_r]\ zs\ ]\!] \Longrightarrow xs\ [\sqsubseteq_r]\ zs\langle proof\rangle$
lemma *le-list-antisym*: $[\![\ order\ r;\ xs\ [\sqsubseteq_r]\ ys;\ ys\ [\sqsubseteq_r]\ xs\ ]\!] \Longrightarrow xs = ys\langle proof\rangle$
lemma *order-listI* [*simp*, *intro*!]: $order\ r \Longrightarrow order(Listn.le\ r)\langle proof\rangle$
lemma *lesub-list-impl-same-size* [*simp*]: $xs\ [\sqsubseteq_r]\ ys \Longrightarrow size\ ys = size\ xs\langle proof\rangle$
lemma *lesssub-lengthD*: $xs\ [\sqsubset_r]\ ys \Longrightarrow size\ ys = size\ xs\langle proof\rangle$
lemma *le-list-appendI*: $a\ [\sqsubseteq_r]\ b \Longrightarrow c\ [\sqsubseteq_r]\ d \Longrightarrow a@c\ [\sqsubseteq_r]\ b@d\langle proof\rangle$
lemma *le-listI*:
assumes $length\ a = length\ b$
assumes $\bigwedge n.\ n < length\ a \Longrightarrow a!n \sqsubseteq_r b!n$
shows $a\ [\sqsubseteq_r]\ b\langle proof\rangle$
lemma *listI*: $[\![\ size\ xs = n;\ set\ xs \subseteq A\ ]\!] \Longrightarrow xs \in list\ n\ A\langle proof\rangle$

lemma *listE-length* [*simp*]: $xs \in list\ n\ A \Longrightarrow size\ xs = n\langle proof\rangle$
lemma *less-lengthI*: $[\![\ xs \in list\ n\ A;\ p < n\ ]\!] \Longrightarrow p < size\ xs\langle proof\rangle$
lemma *listE-set* [*simp*]: $xs \in list\ n\ A \Longrightarrow set\ xs \subseteq A\langle proof\rangle$
lemma *list-0* [*simp*]: $list\ 0\ A = \{[]\}\langle proof\rangle$
lemma *in-list-Suc-iff*:
$(xs \in list\ (Suc\ n)\ A) = (\exists y{\in}A.\ \exists ys \in list\ n\ A.\ xs = y\#ys)\langle proof\rangle$
lemma *Cons-in-list-Suc* [*iff*]:
$(x\#xs \in list\ (Suc\ n)\ A) = (x{\in}A \wedge xs \in list\ n\ A)\langle proof\rangle$
lemma *list-not-empty*:
$\exists a.\ a{\in}A \Longrightarrow \exists xs.\ xs \in list\ n\ A\langle proof\rangle$

lemma *nth-in* [*rule-format*, *simp*]:
$\forall i\ n.\ size\ xs = n \longrightarrow set\ xs \subseteq A \longrightarrow i < n \longrightarrow (xs!i) \in A\langle proof\rangle$
lemma *listE-nth-in*: $[\![\ xs \in list\ n\ A;\ i < n\ ]\!] \Longrightarrow xs!i \in A\langle proof\rangle$
lemma *listn-Cons-Suc* [*elim*!]:
$l\#xs \in list\ n\ A \Longrightarrow (\bigwedge n'.\ n = Suc\ n' \Longrightarrow l \in A \Longrightarrow xs \in list\ n'\ A \Longrightarrow P) \Longrightarrow P\langle proof\rangle$
lemma *listn-appendE* [*elim*!]:
$a@b \in list\ n\ A \Longrightarrow (\bigwedge n1\ n2.\ n{=}n1{+}n2 \Longrightarrow a \in list\ n1\ A \Longrightarrow b \in list\ n2\ A \Longrightarrow P) \Longrightarrow P\langle proof\rangle$

lemma *listt-update-in-list* [*simp*, *intro*!]:
$[\![\ xs \in list\ n\ A;\ x{\in}A\ ]\!] \Longrightarrow xs[i := x] \in list\ n\ A\langle proof\rangle$
lemma *list-appendI* [*intro?*]:
$[\![\ a \in list\ n\ A;\ b \in list\ m\ A\ ]\!] \Longrightarrow a\ @\ b \in list\ (n{+}m)\ A\langle proof\rangle$
lemma *list-map* [*simp*]: $(map\ f\ xs \in list\ (size\ xs)\ A) = (f\ `\ set\ xs \subseteq A)\langle proof\rangle$
lemma *list-replicateI* [*intro*]: $x \in A \Longrightarrow replicate\ n\ x \in list\ n\ A\langle proof\rangle$
lemma *plus-list-Nil* [*simp*]: $[]\ [\sqcup_f]\ xs = []\langle proof\rangle$
lemma *plus-list-Cons* [*simp*]:
$(x\#xs)\ [\sqcup_f]\ ys = (case\ ys\ of\ [] \Rightarrow []\ |\ y\#ys \Rightarrow (x \sqcup_f y)\#(xs\ [\sqcup_f]\ ys))\langle proof\rangle$
lemma *length-plus-list* [*rule-format*, *simp*]:
$\forall ys.\ size(xs\ [\sqcup_f]\ ys) = min(size\ xs)\ (size\ ys)\langle proof\rangle$
lemma *nth-plus-list* [*rule-format*, *simp*]:
$\forall xs\ ys\ i.\ size\ xs = n \longrightarrow size\ ys = n \longrightarrow i{<}n \longrightarrow (xs\ [\sqcup_f]\ ys)!i = (xs!i) \sqcup_f (ys!i)\langle proof\rangle$

lemma (**in** *Semilat*) *plus-list-ub1* [*rule-format*]:

$[\![$ *set xs* $\subseteq A$; *set ys* $\subseteq A$; *size xs* $=$ *size ys* $]\!]$
$\Longrightarrow$ *xs* $[\sqsubseteq_r]$ *xs* $[\sqcup_f]$ *ys*⟨*proof*⟩
lemma (**in** *Semilat*) *plus-list-ub2*:
$[\![$*set xs* $\subseteq A$; *set ys* $\subseteq A$; *size xs* $=$ *size ys* $]\!]$ $\Longrightarrow$ *ys* $[\sqsubseteq_r]$ *xs* $[\sqcup_f]$ *ys*⟨*proof*⟩
lemma (**in** *Semilat*) *plus-list-lub* [*rule-format*]:
shows $\forall$ *xs ys zs. set xs* $\subseteq A \longrightarrow$ *set ys* $\subseteq A \longrightarrow$ *set zs* $\subseteq A$
$\longrightarrow$ *size xs* $= n \wedge$ *size ys* $= n \longrightarrow$
xs $[\sqsubseteq_r]$ *zs* $\wedge$ *ys* $[\sqsubseteq_r]$ *zs* $\longrightarrow$ *xs* $[\sqcup_f]$ *ys* $[\sqsubseteq_r]$ *zs*⟨*proof*⟩
lemma (**in** *Semilat*) *list-update-incr* [*rule-format*]:
$x{\in}A \Longrightarrow$ *set xs* $\subseteq A \longrightarrow$
$(\forall\, i.\ i{<}$*size xs* $\longrightarrow$ *xs* $[\sqsubseteq_r]$ *xs*$[i := x \sqcup_f$ *xs*$!i])$⟨*proof*⟩
lemma *acc-le-listI'* [*intro*!]:
$[\![$ *order r*; *acc A r* $]\!]$ $\Longrightarrow$ *acc* $(\bigcup n.$ *list n A*$)$ $($*Listn.le r*$)$⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*

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* $\Rightarrow$ *'s* $\Rightarrow$ $($*nat* $\times$ *'s*$)$ *list*

definition *stable* :: *'s ord* $\Rightarrow$ *'s step-type* $\Rightarrow$ *'s list* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
where
stable r step τs *p* $\longleftrightarrow$ $(\forall\,(q,\tau) \in$ *set* $($*step p* $(\tau s!p)).\ \tau \sqsubseteq_r \tau s!q)$

definition *stables* :: *'s ord* $\Rightarrow$ *'s step-type* $\Rightarrow$ *'s list* $\Rightarrow$ *bool*
where
stables r step τs $\longleftrightarrow$ $(\forall\, p <$ *size* $\tau s.$ *stable r step* τs *p*$)$

definition *wt-step* :: *'s ord* $\Rightarrow$ *'s* $\Rightarrow$ *'s step-type* $\Rightarrow$ *'s list* $\Rightarrow$ *bool*
where
wt-step r T step τs $\longleftrightarrow$ $(\forall\, p{<}$*size* $\tau s.\ \tau s!p \neq T \wedge$ *stable r step* τs *p*$)$

definition *is-bcv* :: *'s ord* $\Rightarrow$ *'s* $\Rightarrow$ *'s step-type* $\Rightarrow$ *nat* $\Rightarrow$ *'s set* $\Rightarrow$ $($*'s list* $\Rightarrow$ *'s list*$)$ $\Rightarrow$ *bool*
where
is-bcv r T step n A bcv $\longleftrightarrow$ $(\forall\,\tau s_0 \in$ *list n A*.
$(\forall\, p{<}n.\ ($*bcv* $\tau s_0)!p \neq T) = (\exists\,\tau s \in$ *list n A*$.\ \tau s_0\ [\sqsubseteq_r]\ \tau s \wedge$ *wt-step r T step* $\tau s))$

end

2.7 More on Semilattices

theory *SemilatAlg*
imports *Typing-Framework*
begin

definition *lesubstep-type* :: $($*nat* $\times$ *'s*$)$ *set* $\Rightarrow$ *'s ord* $\Rightarrow$ $($*nat* $\times$ *'s*$)$ *set* $\Rightarrow$ *bool*
(‹(- /{⊑-} -)› [*50*, *0*, *51*] *50*)
where $A\ \{\sqsubseteq_r\}\ B \equiv \forall\,(p,\tau) \in A.\ \exists\,\tau'.\ (p,\tau') \in B \wedge \tau \sqsubseteq_r \tau'$

notation (*ASCII*)
lesubstep-type (‹(- /{<=′--} -)› [*50*, *0*, *51*] *50*)

primrec *pluslussub* :: $'a$ *list* ⇒ ($'a$ ⇒ $'a$ ⇒ $'a$) ⇒ $'a$ ⇒ $'a$ (‹(- /$\bigsqcup$_ -)› [*65*, *0*, *66*] *65*)
where
pluslussub [] *f y* = *y*
| *pluslussub* (*x#xs*) *f y* = *pluslussub xs f* ($x \sqcup_f y$)
definition *bounded* :: $'s$ *step-type* ⇒ *nat* ⇒ *bool*
where
bounded step n ⟷ ($\forall p<n.\ \forall \tau.\ \forall (q,\tau') \in$ *set* (*step p* τ). $q<n$)

definition *pres-type* :: $'s$ *step-type* ⇒ *nat* ⇒ $'s$ *set* ⇒ *bool*
where
pres-type step n A ⟷ ($\forall \tau \in A.\ \forall p<n.\ \forall (q,\tau') \in$ *set* (*step p* τ). $\tau' \in A$)

definition *mono* :: $'s$ *ord* ⇒ $'s$ *step-type* ⇒ *nat* ⇒ $'s$ *set* ⇒ *bool*
where
mono r step n A ⟷
($\forall \tau\ p\ \tau'.\ \tau \in A \wedge p<n \wedge \tau \sqsubseteq_r \tau' \longrightarrow$ *set* (*step p* τ) $\{\sqsubseteq_r\}$ *set* (*step p* τ'))

lemma [*iff*]: {} $\{\sqsubseteq_r\}$ *B*
⟨*proof*⟩
lemma [*iff*]: (*A* $\{\sqsubseteq_r\}$ {}) = (*A* = {})
⟨*proof*⟩
lemma *lesubstep-union*:
$[\![\ A_1\ \{\sqsubseteq_r\}\ B_1;\ A_2\ \{\sqsubseteq_r\}\ B_2\ ]\!] \Longrightarrow A_1 \cup A_2\ \{\sqsubseteq_r\}\ B_1 \cup B_2$
⟨*proof*⟩
lemma *pres-typeD*:
$[\![$ *pres-type step n A*; $s \in A$; $p<n$; $(q,s') \in$ *set* (*step p s*) $]\!] \Longrightarrow s' \in A$⟨*proof*⟩
lemma *monoD*:
$[\![$ *mono r step n A*; $p < n$; $s \in A$; $s \sqsubseteq_r t$ $]\!] \Longrightarrow$ *set* (*step p s*) $\{\sqsubseteq_r\}$ *set* (*step p t*)⟨*proof*⟩
lemma *boundedD*:
$[\![$ *bounded step n*; $p < n$; $(q,t) \in$ *set* (*step p xs*) $]\!] \Longrightarrow q < n$⟨*proof*⟩
lemma *lesubstep-type-refl* [*simp*, *intro*]:
$(\bigwedge x.\ x \sqsubseteq_r x) \Longrightarrow A\ \{\sqsubseteq_r\}\ A$⟨*proof*⟩
lemma *lesub-step-typeD*:
$A\ \{\sqsubseteq_r\}\ B \Longrightarrow (x,y) \in A \Longrightarrow \exists y'.\ (x,\ y') \in B \wedge y \sqsubseteq_r y'$⟨*proof*⟩

lemma *list-update-le-listI* [*rule-format*]:
set xs ⊆ *A* ⟶ *set ys* ⊆ *A* ⟶ *xs* $[\sqsubseteq_r]$ *ys* ⟶ $p <$ *size xs* ⟶
$x \sqsubseteq_r$ *ys*!*p* ⟶ *semilat*(*A*,*r*,*f*) ⟶ $x \in A$ ⟶
xs[$p := x \sqcup_f$ *xs*!*p*] $[\sqsubseteq_r]$ *ys*⟨*proof*⟩
lemma *plusplus-closed*: **assumes** *Semilat A r f* **shows**
$\bigwedge y.\ [\![$ *set x* ⊆ *A*; $y \in A]\!] \Longrightarrow x \bigsqcup_f y \in A$⟨*proof*⟩
lemma (**in** *Semilat*) *pp-ub2*:
$\bigwedge y.\ [\![$ *set x* ⊆ *A*; $y \in A]\!] \Longrightarrow y \sqsubseteq_r x \bigsqcup_f y$⟨*proof*⟩

lemma (**in** *Semilat*) *pp-ub1*:
shows $\bigwedge y.\ [\![$*set ls* ⊆ *A*; $y \in A$; $x \in$ *set ls*$]\!] \Longrightarrow x \sqsubseteq_r ls \bigsqcup_f y$⟨*proof*⟩

lemma (**in** *Semilat*) *pp-lub*:
assumes *z*: $z \in A$
shows

$\bigwedge y.\ y \in A \Longrightarrow set\ xs \subseteq A \Longrightarrow \forall x \in set\ xs.\ x \sqsubseteq_r z \Longrightarrow y \sqsubseteq_r z \Longrightarrow xs \bigsqcup_f y \sqsubseteq_r z\langle proof\rangle$

lemma *ub1'*: **assumes** *Semilat A r f*
shows $[\![\forall (p,s) \in set\ S.\ s \in A;\ y \in A;\ (a,b) \in set\ S]\!]$
$\Longrightarrow b \sqsubseteq_r map\ snd\ [(p',\ t') \leftarrow S.\ p' = a] \bigsqcup_f y\langle proof\rangle$

lemma *plusplus-empty*:
$\forall s'.\ (q,\ s') \in set\ S \longrightarrow s' \sqcup_f ss\ !\ q = ss\ !\ q \Longrightarrow$
$(map\ snd\ [(p',\ t') \leftarrow S.\ p' = q] \bigsqcup_f ss\ !\ q) = ss\ !\ q\langle proof\rangle$

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 ⟷
$(\forall p < size\ \tau s.\ app\ p\ (\tau s!p) \wedge (\forall (q,\tau) \in set\ (step\ p\ (\tau s!p)).\ \tau <\!=\!-r\ \tau s!q))$

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 (λx. (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 ⟷
$(\forall s\ p\ t.\ s \in A \wedge p < n \wedge s \sqsubseteq_r t \longrightarrow app\ p\ t \longrightarrow app\ p\ s)$

lemmas *err-step-defs = err-step-def map-snd-def error-def*

lemma *bounded-err-stepD*:
$\llbracket$ *bounded* (*err-step n app step*) *n*;
p < *n*; *app p a*; (*q*,*b*) ∈ *set* (*step p a*) $\rrbracket$ ⟹ *q* < *n*⟨*proof*⟩

lemma *in-map-sndD*: (*a*,*b*) ∈ *set* (*map-snd f xs*) ⟹ ∃ *b'*. (*a*,*b'*) ∈ *set xs*⟨*proof*⟩

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*⟨*proof*⟩

lemma *bounded-lift*:
bounded step n ⟹ *bounded* (*err-step n app step*) *n*⟨*proof*⟩

lemma *le-list-map-OK* [*simp*]:
⋀*b*. (*map OK a* $[\sqsubseteq_{Err.le\ r}]$ *map OK b*) = (*a* $[\sqsubseteq_r]$ *b*)⟨*proof*⟩

lemma *map-snd-lessI*:
set xs $\{\sqsubseteq_r\}$ *set ys* ⟹ *set* (*map-snd OK xs*) $\{\sqsubseteq_{Err.le\ r}\}$ *set* (*map-snd OK ys*)⟨*proof*⟩

lemma *mono-lift*:
$\llbracket$ *order r*; *app-mono r app n A*; *bounded* (*err-step n app step*) *n*;
∀ *s p t*. *s* ∈ *A* ∧ *p* < *n* ∧ *s* $\sqsubseteq_r$ *t* ⟶ *app p t* ⟶ *set* (*step p s*) $\{\sqsubseteq_r\}$ *set* (*step p t*) $\rrbracket$
⟹ *mono* (*Err.le r*) (*err-step n app step*) *n* (*err A*)⟨*proof*⟩
lemma *in-errorD*: (*x*,*y*) ∈ *set* (*error n*) ⟹ *y* = *Err*⟨*proof*⟩
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*)⟨*proof*⟩

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*)⟨*proof*⟩

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*)⟨*proof*⟩
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 = $\tau \sqcup_f \tau s!q$;
w' = (*if* *u* = $\tau s!q$ *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* ($\tau s!p$)) 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* (λp. ¬*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* := $\tau \sqcup_f \tau s!p$]))

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*)
⟨*proof*⟩

lemma *unstables-spec* [*simp*]: *s-*α (*unstables r step* τs) = {*p*. *p* < *size* τs ∧ ¬*stable r step* τs *p*}
⟨*proof*⟩

end

lemmas [*simp*] = *Let-def Semilat.le-iff-plus-unchanged* [*OF Semilat.intro*, *symmetric*]

lemma (**in** *Semilat*) *nth-merges*:
$\bigwedge ss.\ [\![p < length\ ss;\ ss \in list\ n\ A;\ \forall (p,t){\in}set\ ps.\ p{<}n \wedge t{\in}A\ ]\!] \Longrightarrow$
$(merges\ f\ ps\ ss)!p = map\ snd\ [(p',t') \leftarrow ps.\ p'{=}p]\ \bigsqcup_f\ ss!p$
(**is** $\bigwedge ss.\ [\![-; -;\ ?steptype\ ps]\!] \Longrightarrow\ ?P\ ss\ ps)\langle proof\rangle$

lemma *length-merges* [*simp*]:
$\bigwedge ss.\ size(merges\ f\ ps\ ss) = size\ ss\langle proof\rangle$
lemma (**in** *Semilat*) *merges-preserves-type-lemma*:
shows $\forall xs.\ xs \in list\ n\ A \longrightarrow (\forall (p,x) \in set\ ps.\ p{<}n \wedge x{\in}A)$
$\longrightarrow merges\ f\ ps\ xs \in list\ n\ A\langle proof\rangle$
lemma (**in** *Semilat*) *merges-preserves-type* [*simp*]:
$[\![\ xs \in list\ n\ A;\ \forall (p,x) \in set\ ps.\ p{<}n \wedge x{\in}A\ ]\!]$
$\Longrightarrow merges\ f\ ps\ xs \in list\ n\ A$
$\langle proof\rangle$

lemma (**in** *Semilat*) *merges-incr-lemma*:
$\forall xs.\ xs \in list\ n\ A \longrightarrow (\forall (p,x){\in}set\ ps.\ p{<}size\ xs \wedge x \in A) \longrightarrow xs\ [\sqsubseteq_r]\ merges\ f\ ps\ xs\langle proof\rangle$
lemma (**in** *Semilat*) *merges-incr*:
$[\![\ xs \in list\ n\ A;\ \forall (p,x){\in}set\ ps.\ p{<}size\ xs \wedge x \in A\ ]\!]$
$\Longrightarrow xs\ [\sqsubseteq_r]\ merges\ f\ ps\ xs$
$\langle proof\rangle$

lemma (**in** *Semilat*) *merges-same-conv* [*rule-format*]:
$(\forall xs.\ xs \in list\ n\ A \longrightarrow (\forall (p,x){\in}set\ ps.\ p{<}size\ xs \wedge x{\in}A) \longrightarrow$
$(merges\ f\ ps\ xs = xs) = (\forall (p,x){\in}set\ ps.\ x \sqsubseteq_r xs!p))\langle proof\rangle$

lemma (**in** *Semilat*) *list-update-le-listI* [*rule-format*]:
$set\ xs \subseteq A \longrightarrow set\ ys \subseteq A \longrightarrow xs\ [\sqsubseteq_r]\ ys \longrightarrow p < size\ xs \longrightarrow$
$x \sqsubseteq_r ys!p \longrightarrow x{\in}A \longrightarrow xs[p := x \sqcup_f xs!p]\ [\sqsubseteq_r]\ ys\langle proof\rangle$
lemma (**in** *Semilat*) *merges-pres-le-ub*:
assumes $set\ ts \subseteq A\quad set\ ss \subseteq A$
$\forall (p,t){\in}set\ ps.\ t \sqsubseteq_r ts!p \wedge t \in A \wedge p < size\ ts\quad ss\ [\sqsubseteq_r]\ ts$
shows $merges\ f\ ps\ ss\ [\sqsubseteq_r]\ ts\langle proof\rangle$
context *Kildall* **begin**

2.9.1 *propa*

lemma *decomp-propa*:
$\bigwedge ss\ w.\ (\forall (q,t){\in}set\ qs.\ q < size\ ss) \Longrightarrow$
$t\text{-}\alpha\ (propa\ f\ qs\ ss\ w) =$
$(merges\ f\ qs\ ss,\ \{q.\ \exists t.(q,t){\in}set\ qs \wedge t \sqcup_f ss!q \neq ss!q\} \cup s\text{-}\alpha\ w)$
$\langle proof\rangle$

end

lemma (**in** *Semilat*) *stable-pres-lemma*:
shows $[\![pres\text{-}type\ step\ n\ A;\ bounded\ step\ n;$
$ss \in list\ n\ A;\ p \in w;\ \forall q{\in}w.\ q < n;$
$\forall q.\ q < n \longrightarrow q \notin w \longrightarrow stable\ r\ step\ ss\ q;\ q < n;$
$\forall s'.\ (q,s') \in set\ (step\ p\ (ss!p)) \longrightarrow s' \sqcup_f ss!q = ss!q;$
$q \notin w \vee q = p\ ]\!]$

$\Longrightarrow$ *stable r step (merges f (step p (ss!p)) ss) q*⟨*proof*⟩

lemma (**in** *Semilat*) *merges-bounded-lemma*:
$[\![$ *mono r step n A*; *bounded step n*;
$\forall (p',s') \in$ *set* (*step p* (*ss*!*p*)). $s' \in A$; $ss \in$ *list n A*; $ts \in$ *list n A*; $p < n$;
ss $[\sqsubseteq_r]$ *ts*; $\forall p.\ p < n \longrightarrow$ *stable r step ts p* $]\!]$
$\Longrightarrow$ *merges f* (*step p* (*ss*!*p*)) *ss* $[\sqsubseteq_r]$ *ts*⟨*proof*⟩

lemma *termination-lemma*: **assumes** *Semilat A r f*
shows $[\![$ $ss \in$ *list n A*; $\forall (q,t) \in$*set qs*. $q<n \wedge t \in A$; $p \in w$ $]\!] \Longrightarrow$
ss $[\sqsubset_r]$ *merges f qs ss* $\vee$
merges f qs ss = *ss* $\wedge$ $\{q.\ \exists t.\ (q,t) \in$*set qs* $\wedge$ $t \sqcup_f ss!q \neq ss!q\} \cup (w - \{p\}) \subset w$⟨*proof*⟩
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
⟨*proof*⟩

lemma *wf-s-finite-psubset* [*simp*]: *wf s-finite-psubset*
⟨*proof*⟩

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; $\forall p \in$*s-*α *w0*. $p < n$; $ss0 \in$ *list n A*;
$\forall p<n.\ p \notin$ *s-*α *w0* $\longrightarrow$ *stable r step ss0 p* $]\!] \Longrightarrow$
t-α (*iter f step ss0 w0*) = (ss',w')
$\longrightarrow$
$ss' \in$ *list n A* $\wedge$ *stables r step* ss' $\wedge$ *ss0* $[\sqsubseteq_r]$ ss' $\wedge$
($\forall ts \in$*list n A*. *ss0* $[\sqsubseteq_r]$ *ts* $\wedge$ *stables r step ts* $\longrightarrow$ ss' $[\sqsubseteq_r]$ *ts*)⟨*proof*⟩
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; $ss0 \in$ *list n A* $]\!] \Longrightarrow$
kildall r f step ss0 $\in$ *list n A* $\wedge$
stables r step (*kildall r f step ss0*) $\wedge$
ss0 $[\sqsubseteq_r]$ *kildall r f step ss0* $\wedge$
($\forall ts \in$*list n A*. *ss0* $[\sqsubseteq_r]$ *ts* $\wedge$ *stables r step ts* $\longrightarrow$
kildall r f step ss0 $[\sqsubseteq_r]$ *ts*)⟨*proof*⟩⟨*proof*⟩
end

interpretation *Kildall set* [] $\lambda xs.\ xs = []$ *hd removeAll Cons*
⟨*proof*⟩

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* $:: 's\ certificate \Rightarrow 's\ binop \Rightarrow 's\ ord \Rightarrow 's \Rightarrow nat \Rightarrow (nat \times 's)\ list \Rightarrow 's \Rightarrow 's$
where
$merge\ cert\ f\ r\ T\ pc\ []\quad x = x$
$|\ merge\ cert\ f\ r\ T\ pc\ (s\#ss)\ x = merge\ cert\ f\ r\ T\ pc\ ss\ (let\ (pc',s') = s\ in$
$\quad if\ pc'=pc+1\ then\ s' \sqcup_f x$
$\quad else\ if\ s' \sqsubseteq_r cert!pc'\ then\ x$
$\quad else\ T)$

definition *wtl-inst* $:: 's\ certificate \Rightarrow 's\ binop \Rightarrow 's\ ord \Rightarrow 's \Rightarrow$
$'s\ step\text{-}type \Rightarrow nat \Rightarrow 's \Rightarrow 's$
where
$wtl\text{-}inst\ cert\ f\ r\ T\ step\ pc\ s = merge\ cert\ f\ r\ T\ pc\ (step\ pc\ s)\ (cert!(pc+1))$

definition *wtl-cert* $:: 's\ certificate \Rightarrow 's\ binop \Rightarrow 's\ ord \Rightarrow 's \Rightarrow 's \Rightarrow$
$'s\ step\text{-}type \Rightarrow nat \Rightarrow 's \Rightarrow 's$
where
$wtl\text{-}cert\ cert\ f\ r\ T\ B\ step\ pc\ s =$
$(if\ cert!pc = B\ then$
$\quad wtl\text{-}inst\ cert\ f\ r\ T\ step\ pc\ s$
$else$
$\quad if\ s \sqsubseteq_r cert!pc\ then\ wtl\text{-}inst\ cert\ f\ r\ T\ step\ pc\ (cert!pc)\ else\ T)$

primrec *wtl-inst-list* $:: 'a\ list \Rightarrow 's\ certificate \Rightarrow 's\ binop \Rightarrow 's\ ord \Rightarrow 's \Rightarrow 's \Rightarrow$
$'s\ step\text{-}type \Rightarrow nat \Rightarrow 's \Rightarrow 's$
where
$wtl\text{-}inst\text{-}list\ []\quad cert\ f\ r\ T\ B\ step\ pc\ s = s$
$|\ wtl\text{-}inst\text{-}list\ (i\#is)\ cert\ f\ r\ T\ B\ step\ pc\ s =$
$\quad (let\ s' = wtl\text{-}cert\ cert\ f\ r\ T\ B\ step\ pc\ s\ in$
$\quad\quad if\ s' = T \lor s = T\ then\ T\ else\ wtl\text{-}inst\text{-}list\ is\ cert\ f\ r\ T\ B\ step\ (pc+1)\ s')$

definition *cert-ok* $:: 's\ certificate \Rightarrow nat \Rightarrow 's \Rightarrow 's \Rightarrow 's\ set \Rightarrow bool$
where
$cert\text{-}ok\ cert\ n\ T\ B\ A \longleftrightarrow (\forall i < n.\ cert!i \in A \land cert!i \neq T) \land (cert!n = B)$

definition *bottom* $:: 'a\ ord \Rightarrow 'a \Rightarrow bool$
where
$bottom\ r\ B \longleftrightarrow (\forall x.\ B \sqsubseteq_r x)$

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*))
  ⟨*proof*⟩
lemma (**in** *lbv*) *wtc*:
  wtc c pc s = (*if c*!*pc* = ⊥ *then wti c pc s else if s* $\sqsubseteq_r$ *c*!*pc then wti c pc* (*c*!*pc*) *else* ⊤)
  ⟨*proof*⟩
lemma *cert-okD1* [*intro?*]:
  cert-ok c n T B A ⟹ *pc* < *n* ⟹ *c*!*pc* ∈ A
  ⟨*proof*⟩
lemma *cert-okD2* [*intro?*]:
  cert-ok c n T B A ⟹ *c*!*n* = B
  ⟨*proof*⟩
lemma *cert-okD3* [*intro?*]:
  cert-ok c n T B A ⟹ B ∈ A ⟹ *pc* < *n* ⟹ *c*!*Suc pc* ∈ A
  ⟨*proof*⟩
lemma *cert-okD4* [*intro?*]:
  cert-ok c n T B A ⟹ *pc* < *n* ⟹ *c*!*pc* ≠ T
  ⟨*proof*⟩
declare *Let-def* [*simp*]

2.10.1 more semilattice lemmas

lemma (**in** *lbv*) *sup-top* [*simp*, *elim*]:
  assumes *x*: $x \in A$
  shows $x \sqcup_f \top = \top$⟨*proof*⟩
lemma (**in** *lbv*) *plusplussup-top* [*simp*, *elim*]:
  set xs ⊆ A ⟹ *xs* $\bigsqcup_f$ ⊤ = ⊤
  ⟨*proof*⟩

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* $[(p',\ t') \leftarrow S\ .\ p' = a] \bigsqcup_f y$⟨*proof*⟩
lemma (**in** *lbv*) *bottom-le* [*simp*, *intro*!]: $\bot \sqsubseteq_r x$
⟨*proof*⟩

lemma (**in** *lbv*) *le-bottom* [*simp*]: $x \sqsubseteq_r \bot = (x = \bot)$
⟨*proof*⟩

2.10.2 merge

lemma (**in** *lbv*) *merge-Nil* [*simp*]:
merge c pc [] $x = x$ ⟨*proof*⟩

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 $\top$)
⟨*proof*⟩

lemma (**in** *lbv*) *merge-Err* [*simp*]:
snd`set ss $\subseteq A \Longrightarrow$ *merge c pc ss* $\top = \top$
⟨*proof*⟩

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*)⟨*proof*⟩

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** $\bigwedge x$. - $\Longrightarrow$ - $\Longrightarrow$ *?merge ss x* = *?if ss x* **is** $\bigwedge x$. - $\Longrightarrow$ - $\Longrightarrow$ *?P ss x*)⟨*proof*⟩
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* $[(p',t') \leftarrow ss.\ p'=pc+1] \bigsqcup_f x$)⟨*proof*⟩

2.10.3 wtl-inst-list

lemmas [*iff*] = *not-Err-eq*

lemma (**in** *lbv*) *wtl-Nil* [*simp*]: *wtl* [] *c pc s* = *s*
⟨*proof*⟩

lemma (**in** *lbv*) *wtl-Cons* [*simp*]:
wtl (*i*#*is*) *c pc s* =
(*let* s' = *wtc c pc s in if* $s' = \top \vee s = \top$ *then* $\top$ *else wtl is c* (*pc+1*) s')
⟨*proof*⟩

lemma (**in** *lbv*) *wtl-Cons-not-top*:
$wtl\ (i\#is)\ c\ pc\ s \neq \top =$
$(wtc\ c\ pc\ s \neq \top \wedge s \neq T \wedge wtl\ is\ c\ (pc+1)\ (wtc\ c\ pc\ s) \neq \top)$
⟨*proof*⟩

lemma (**in** *lbv*) *wtl-top* [*simp*]: $wtl\ ls\ c\ pc\ \top = \top$
⟨*proof*⟩

lemma (**in** *lbv*) *wtl-not-top*:
$wtl\ ls\ c\ pc\ s \neq \top \Longrightarrow s \neq \top$
⟨*proof*⟩

lemma (**in** *lbv*) *wtl-append* [*simp*]:
$\bigwedge pc\ s.\ wtl\ (a@b)\ c\ pc\ s = wtl\ b\ c\ (pc+length\ a)\ (wtl\ a\ c\ pc\ s)$
⟨*proof*⟩

lemma (**in** *lbv*) *wtl-take*:
$wtl\ is\ c\ pc\ s \neq \top \Longrightarrow wtl\ (take\ pc'\ is)\ c\ pc\ s \neq \top$
(**is** $?wtl\ is \neq -\ \Longrightarrow -$)⟨*proof*⟩
lemma *take-Suc*:
$\forall n.\ n < length\ l \longrightarrow take\ (Suc\ n)\ l = (take\ n\ l)@[l!n]$ (**is** $?P\ l$)⟨*proof*⟩
lemma (**in** *lbv*) *wtl-Suc*:
assumes *suc*: $pc+1 < length\ is$
assumes *wtl*: $wtl\ (take\ pc\ is)\ c\ 0\ s \neq \top$
shows $wtl\ (take\ (pc+1)\ is)\ c\ 0\ s = wtc\ c\ pc\ (wtl\ (take\ pc\ is)\ c\ 0\ s)$⟨*proof*⟩
lemma (**in** *lbv*) *wtl-all*:
assumes *all*: $wtl\ is\ c\ 0\ s \neq \top$ (**is** $?wtl\ is \neq -$)
assumes *pc*: $pc < length\ is$
shows $wtc\ c\ pc\ (wtl\ (take\ pc\ is)\ c\ 0\ s) \neq \top$⟨*proof*⟩

2.10.4 preserves-type

lemma (**in** *lbv*) *merge-pres*:
assumes *s0*: $snd`set\ ss \subseteq A$ **and** *x*: $x \in A$
shows $merge\ c\ pc\ ss\ x \in A$⟨*proof*⟩
lemma *pres-typeD2*:
$pres\text{-}type\ step\ n\ A \Longrightarrow s \in A \Longrightarrow p < n \Longrightarrow snd`set\ (step\ p\ s) \subseteq A$
⟨*proof*⟩

lemma (**in** *lbv*) *wti-pres* [*intro?*]:
assumes *pres*: $pres\text{-}type\ step\ n\ A$
assumes *cert*: $c!(pc+1) \in A$
assumes *s-pc*: $s \in A\ pc < n$
shows $wti\ c\ pc\ s \in A$⟨*proof*⟩
lemma (**in** *lbv*) *wtc-pres*:
assumes $pres\text{-}type\ step\ n\ A$
assumes $c!pc \in A$ **and** $c!(pc+1) \in A$
assumes $s \in A$ **and** $pc < n$
shows $wtc\ c\ pc\ s \in A$⟨*proof*⟩
lemma (**in** *lbv*) *wtl-pres*:
assumes *pres*: $pres\text{-}type\ step\ (length\ is)\ A$
assumes *cert*: $cert\text{-}ok\ c\ (length\ is)\ \top\ \bot\ A$
assumes *s*: $s \in A$
assumes *all*: $wtl\ is\ c\ 0\ s \neq \top$

shows *pc* < *length is* ⟹ *wtl* (*take pc is*) *c 0 s* ∈ *A*
(**is** *?len pc* ⟹ *?wtl pc* ∈ *A*)⟨*proof*⟩
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*:
τs ≡ *map* (*λpc. if c*!*pc* = ⊥ *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*) ⊤ ⊥ *A*
assumes *pres*: *pres-type step* (*size ins*) *A*

lemma (**in** *lbvs*) *phi-None* [*intro?*]:
⟦ *pc* < *size ins*; *c*!*pc* = ⊥ ⟧ ⟹ *τs*!*pc* = *wtl* (*take pc ins*) *c 0* s_0⟨*proof*⟩
lemma (**in** *lbvs*) *phi-Some* [*intro?*]:
⟦ *pc* < *size ins*; *c*!*pc* ≠ ⊥ ⟧ ⟹ *τs*!*pc* = *c*!*pc*⟨*proof*⟩
lemma (**in** *lbvs*) *phi-len* [*simp*]: *size τs* = *size ins*⟨*proof*⟩
lemma (**in** *lbvs*) *wtl-suc-pc*:
assumes *all*: *wtl ins c 0* s_0 ≠ ⊤
assumes *pc*: *pc*+*1* < *size ins*
shows *wtl* (*take* (*pc*+*1*) *ins*) *c 0* $s_0 \sqsubseteq_r$ *τs*!(*pc*+*1*)⟨*proof*⟩
lemma (**in** *lbvs*) *wtl-stable*:
assumes *wtl*: *wtl ins c 0* s_0 ≠ ⊤
assumes s_0: s_0 ∈ *A* **and** *pc*: *pc* < *size ins*
shows *stable r step τs pc*⟨*proof*⟩
lemma (**in** *lbvs*) *phi-not-top*:
assumes *wtl*: *wtl ins c 0* s_0 ≠ ⊤ **and** *pc*: *pc* < *size ins*
shows *τs*!*pc* ≠ ⊤⟨*proof*⟩
lemma (**in** *lbvs*) *phi-in-A*:
assumes *wtl*: *wtl ins c 0* s_0 ≠ ⊤ **and** s_0: s_0 ∈ *A*
shows *τs* ∈ *list* (*size ins*) *A*⟨*proof*⟩
lemma (**in** *lbvs*) *phi0*:
assumes *wtl*: *wtl ins c 0* s_0 ≠ ⊤ **and** *0*: *0* < *size ins*
shows $s_0 \sqsubseteq_r$ *τs*!*0*⟨*proof*⟩

theorem (**in** *lbvs*) *wtl-sound*:
assumes *wtl*: *wtl ins c 0* s_0 ≠ ⊤ **and** s_0: s_0 ∈ *A*
shows ∃ *τs*. *wt-step r* ⊤ *step τs*⟨*proof*⟩

theorem (**in** *lbvs*) *wtl-sound-strong*:
assumes *wtl*: *wtl ins c 0* s_0 ≠ ⊤
assumes s_0: s_0 ∈ *A* **and** *ins*: *0* < *size ins*

shows $\exists\tau s \in list\ (size\ ins)\ A.\ wt\text{-}step\ r\ \top\ step\ \tau s \wedge s_0 \sqsubseteq_r \tau s!0\langle proof\rangle$
end

2.12 Completeness of the LBV

theory *LBVComplete*
imports *LBVSpec Typing-Framework*
begin

definition $is\text{-}target :: 's\ step\text{-}type \Rightarrow 's\ list \Rightarrow nat \Rightarrow bool$ **where**
$is\text{-}target\ step\ \tau s\ pc' \longleftrightarrow (\exists pc\ s'.\ pc' \neq pc{+}1 \wedge pc < size\ \tau s \wedge (pc',s') \in set\ (step\ pc\ (\tau s!pc)))$

definition $make\text{-}cert :: 's\ step\text{-}type \Rightarrow 's\ list \Rightarrow 's \Rightarrow 's\ certificate$ **where**
$make\text{-}cert\ step\ \tau s\ B = map\ (\lambda pc.\ if\ is\text{-}target\ step\ \tau s\ pc\ then\ \tau s!pc\ else\ B)\ [0..{<}size\ \tau s]\ @\ [B]$

lemma [*code*]:
$is\text{-}target\ step\ \tau s\ pc' =$
$list\text{-}ex\ (\lambda pc.\ pc' \neq pc{+}1 \wedge List.member\ (map\ fst\ (step\ pc\ (\tau s!pc)))\ pc')\ [0..{<}size\ \tau s]\langle proof\rangle$
locale $lbvc = lbv +$
fixes $\tau s :: 'a\ list$
fixes $c :: 'a\ list$
defines *cert-def*: $c \equiv make\text{-}cert\ step\ \tau s\ \bot$

assumes *mono*: $mono\ r\ step\ (size\ \tau s)\ A$
assumes *pres*: $pres\text{-}type\ step\ (size\ \tau s)\ A$
assumes τs: $\forall pc < size\ \tau s.\ \tau s!pc \in A \wedge \tau s!pc \neq \top$
assumes *bounded*: $bounded\ step\ (size\ \tau s)$

assumes *B-neq-T*: $\bot \neq \top$

lemma (**in** *lbvc*) *cert*: $cert\text{-}ok\ c\ (size\ \tau s)\ \top\ \bot\ A\langle proof\rangle$
lemmas [*simp del*] = *split-paired-Ex*

lemma (**in** *lbvc*) *cert-target* [*intro?*]:
$[\![\ (pc',s') \in set\ (step\ pc\ (\tau s!pc));$
$pc' \neq pc{+}1;\ pc < size\ \tau s;\ pc' < size\ \tau s\ ]\!]$
$\Longrightarrow c!pc' = \tau s!pc'\langle proof\rangle$
lemma (**in** *lbvc*) *cert-approx* [*intro?*]:
$[\![\ pc < size\ \tau s;\ c!pc \neq \bot\ ]\!] \Longrightarrow c!pc = \tau s!pc\langle proof\rangle$
lemma (**in** *lbv*) *le-top* [*simp*, *intro*]: $x <=\text{-}r\ \top\langle proof\rangle$
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** $?s_2 \sqsubseteq_r ?s_1)\langle proof\rangle$
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** $?s_2' \sqsubseteq_r ?s_1')\langle proof\rangle$
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** $?s_2{}' \sqsubseteq_r\ ?s_1{}'$)⟨*proof*⟩
lemma (**in** *lbv*) *top-le-conv* [*simp*]: $\top \sqsubseteq_r x = (x = \top)$⟨*proof*⟩
lemma (**in** *lbv*) *neq-top* [*simp*, *elim*]: $[\![\ x \sqsubseteq_r y;\ y \neq \top\ ]\!] \Longrightarrow x \neq \top$⟨*proof*⟩
lemma (**in** *lbvc*) *stable-wti*:
assumes *stable*: *stable r step* τs *pc* **and** *pc*: $pc < size\ \tau s$
shows $wti\ c\ pc\ (\tau s!pc) \neq \top$⟨*proof*⟩
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** $?wti \sqsubseteq_r$ -)⟨*proof*⟩
lemma (**in** *lbvc*) *stable-wtc*:
assumes *stable*: *stable r step* τs *pc* **and** *pc*: $pc < size\ \tau s$
shows $wtc\ c\ pc\ (\tau s!pc) \neq \top$⟨*proof*⟩
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** $?wtc \sqsubseteq_r$ -)⟨*proof*⟩
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** $\bigwedge pc\ s.$ - $\Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ - $\Longrightarrow$ $?wtl\ ls\ pc\ s \neq$ -)⟨*proof*⟩
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 $wtl\ ins\ c\ 0\ s \neq \top$⟨*proof*⟩
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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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 ($T\lfloor\rceil$) = *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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *class-type-of-eq*:
class-type-of hT =
(*case hT of Class-type C* ⇒ *C* | *Array-type T n* ⇒ *Object*)
⟨*proof*⟩

lemma *class-type-of'-ty-of-htype* [*simp*]:

class-type-of' (*ty-of-htype hT*) = ⌊*class-type-of hT*⌋
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

3.1.1 Code generator setup

code-pred *is-refT* ⟨*proof*⟩

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*}⟨*proof*⟩
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*
⟨*proof*⟩

lemma *is-type-ground-type*:
is-type P T ⟹ *is-type P* (*ground-type T*)
⟨*proof*⟩

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*
⟨*proof*⟩

code-pred
(*modes*: *i* ⇒ *i* ⇒ *bool*)
is-class
⟨*proof*⟩

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* (‹- ⊢ - $\prec^1$ -› [*71*, *71*, *71*] *70*)
for *P* :: *'m prog*
where *subcls1I*: ⟦ *class P C* = *Some* (*D*, *rest*); *C* ≠ *Object* ⟧ ⟹ *P* ⊢ *C* $\prec^1$ *D*

abbreviation *subcls* :: *'m prog* ⇒ *cname* ⇒ *cname* ⇒ *bool* (‹- ⊢ - $\preceq^*$ -› [*71*,*71*,*71*] *70*)
where *P* ⊢ *C* $\preceq^*$ *D* ≡ (*subcls1 P*)** *C D*

lemma *subcls1D*:
P ⊢ *C* $\prec^1$ *D* ⟹ *C* ≠ *Object* ∧ (∃ *fs ms*. *class P C* = *Some* (*D*,*fs*,*ms*))
⟨*proof*⟩

lemma *Object-subcls1* [*iff*]: ¬ *P* ⊢ *Object* $\prec^1$ *C*
⟨*proof*⟩

lemma *Object-subcls-conv* [*iff*]: (*P* ⊢ *Object* $\preceq^*$ *C*) = (*C* = *Object*)
⟨*proof*⟩

lemma *finite-subcls1*: *finite* {(*C*, *D*). *P* ⊢ *C* $\prec^1$ *D*}
⟨*proof*⟩

lemma *finite-subcls1'*:
finite ({(*D*, *C*). *P* ⊢ *C* $\prec^1$ *D*})
⟨*proof*⟩

lemma *subcls-is-class*: (*subcls1 P*)$^{++}$ *C D* ⟹ *is-class P C*
⟨*proof*⟩

lemma *subcls-is-class1*: ⟦ *P* ⊢ *C* $\preceq^*$ *D*; *is-class P D* ⟧ ⟹ *is-class P C*
⟨*proof*⟩

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* ⊢ *T* ≤ *T*
| *widen-subcls*: *P* ⊢ *C* $\preceq^*$ *D* ⟹ *P* ⊢ *Class C* ≤ *Class D*
| *widen-null*[*iff*]: *P* ⊢ *NT* ≤ *Class C*
| *widen-null-array*[*iff*]: *P* ⊢ *NT* ≤ *Array A*
| *widen-array-object*: *P* ⊢ *Array A* ≤ *Class Object*
| *widen-array-array*: *P* ⊢ *A* ≤ *B* ⟹ *P* ⊢ *Array A* ≤ *Array B*

abbreviation
widens :: *'m prog* ⇒ *ty list* ⇒ *ty list* ⇒ *bool* (‹- ⊢ - [≤] -› [*71*,*71*,*71*] *70*)
where
$P \vdash Ts\ [\le]\ Ts' == list\text{-}all2\ (widen\ P)\ Ts\ Ts'$

lemma [*iff*]: $(P \vdash T \le Void) = (T = Void)$⟨*proof*⟩
lemma [*iff*]: $(P \vdash T \le Boolean) = (T = Boolean)$⟨*proof*⟩
lemma [*iff*]: $(P \vdash T \le Integer) = (T = Integer)$⟨*proof*⟩
lemma [*iff*]: $(P \vdash Void \le T) = (T = Void)$⟨*proof*⟩
lemma [*iff*]: $(P \vdash Boolean \le T) = (T = Boolean)$⟨*proof*⟩
lemma [*iff*]: $(P \vdash Integer \le T) = (T = Integer)$⟨*proof*⟩
lemma *Class-widen*: $P \vdash Class\ C \le T \implies \exists D.\ T = Class\ D$
⟨*proof*⟩

lemma *Array-Array-widen*:
$P \vdash Array\ T \le Array\ U \Longrightarrow P \vdash T \le U$
⟨*proof*⟩

lemma *widen-Array*: $(P \vdash T \le U\lfloor\rceil) \longleftrightarrow (T = NT \lor (\exists V.\ T = V\lfloor\rceil \land P \vdash V \le U))$
⟨*proof*⟩

lemma *Array-widen*: $P \vdash Array\ A \le T \Longrightarrow (\exists B.\ T = Array\ B \land P \vdash A \le B) \lor T = Class\ Object$
⟨*proof*⟩

lemma [*iff*]: $(P \vdash T \le NT) = (T = NT)$
⟨*proof*⟩

lemma *Class-widen-Class* [*iff*]: $(P \vdash Class\ C \le Class\ D) = (P \vdash C \preceq^* D)$
⟨*proof*⟩

lemma *widen-Class*: $(P \vdash T \le Class\ C) = (T = NT \lor (\exists D.\ T = Class\ D \land P \vdash D \preceq^* C) \lor (C = Object \land (\exists A.\ T = Array\ A)))$
⟨*proof*⟩

lemma *NT-widen*:
$P \vdash NT \le T = (T = NT \lor (\exists C.\ T = Class\ C) \lor (\exists U.\ T = U\lfloor\rceil))$
⟨*proof*⟩

lemma *Class-widen2*: $P \vdash Class\ C \le T = (\exists D.\ T = Class\ D \land P \vdash C \preceq^* D)$
⟨*proof*⟩

lemma *Object-widen*: $P \vdash Class\ Object \le T \Longrightarrow T = Class\ Object$
⟨*proof*⟩

lemma *NT-Array-widen-Object*:
$is\text{-}NT\text{-}Array\ T \implies P \vdash T \le Class\ Object$
⟨*proof*⟩

lemma *widen-trans*[*trans*]:
assumes $P \vdash S \le U$ $P \vdash U \le T$
shows $P \vdash S \le T$
⟨*proof*⟩

lemma *widens-trans*: $[\![P \vdash Ss\ [\leq]\ Ts;\ P \vdash Ts\ [\leq]\ Us]\!] \Longrightarrow P \vdash Ss\ [\leq]\ Us$
⟨*proof*⟩

lemma *class-type-of′-widenD*:
$class\text{-}type\text{-}of'\ T = \lfloor C \rfloor \Longrightarrow P \vdash T \leq Class\ C$
⟨*proof*⟩

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** $class\text{-}type\text{-}of'\ T' = \lfloor C' \rfloor$ $P \vdash C' \preceq^* C$
⟨*proof*⟩

lemma *widens-refl*: $P \vdash Ts\ [\leq]\ Ts$
⟨*proof*⟩

lemma *widen-append1*:
$P \vdash (xs\ @\ ys)\ [\leq]\ Ts = (\exists\, Ts1\ Ts2.\ Ts = Ts1\ @\ Ts2 \wedge length\ xs = length\ Ts1 \wedge length\ ys = length\ Ts2 \wedge P \vdash xs\ [\leq]\ Ts1 \wedge P \vdash ys\ [\leq]\ Ts2)$
⟨*proof*⟩

lemmas *widens-Cons* [*iff*] = *list-all2-Cons1* [*of widen P*] **for** P

lemma *widens-lengthD*:
$P \vdash xs\ [\leq]\ ys \Longrightarrow length\ xs = length\ ys$
⟨*proof*⟩

lemma *widen-refT*: $[\![\ is\text{-}refT\ T;\ P \vdash U \leq T\ ]\!] \Longrightarrow is\text{-}refT\ U$
⟨*proof*⟩

lemma *refT-widen*: $[\![\ is\text{-}refT\ T;\ P \vdash T \leq U\ ]\!] \Longrightarrow is\text{-}refT\ U$
⟨*proof*⟩

inductive *is-lub* :: $'m\ prog \Rightarrow ty \Rightarrow ty \Rightarrow ty \Rightarrow bool$ (‹- ⊢ lub′((-,/ -)′) = -› [*51,51,51,51*] *50*)
for P :: $'m\ prog$ **and** U :: ty **and** V :: ty **and** T :: ty
where
$[\![\ P \vdash U \leq T;\ P \vdash V \leq T;$
$\bigwedge T'.\ [\![\ P \vdash U \leq T';\ P \vdash V \leq T'\ ]\!] \Longrightarrow P \vdash T \leq T'\ ]\!]$
$\Longrightarrow P \vdash lub(U,\ V) = T$

lemma *is-lub-upper*:
$P \vdash lub(U,\ V) = T \Longrightarrow P \vdash U \leq T \wedge P \vdash V \leq T$
⟨*proof*⟩

lemma *is-lub-least*:
$[\![\ P \vdash lub(U,\ V) = T;\ P \vdash U \leq T';\ P \vdash V \leq T'\ ]\!] \Longrightarrow P \vdash T \leq T'$
⟨*proof*⟩

lemma *is-lub-Void* [*iff*]:
$P \vdash lub(Void,\ Void) = T \longleftrightarrow T = Void$
⟨*proof*⟩

lemma *is-lubI* [*code-pred-intro*]:
$[\![P \vdash U \leq T;\ P \vdash V \leq T;\ \forall T'.\ P \vdash U \leq T' \longrightarrow P \vdash V \leq T' \longrightarrow P \vdash T \leq T']\!] \Longrightarrow P \vdash lub(U,\ V) = T$

⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *sees-methods-decl-above*:
assumes *P* ⊢ *C sees-methods Mm*
shows *Mm M* = *Some*(*m*,*D*) ⟹ *P* ⊢ *C* $\preceq^*$ *D*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *sees-methods-decl-mono*:
assumes *sub*: *P* ⊢ *C'* $\preceq^*$ *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* $\preceq^*$ *C*)
(**is** ∃ *Mm' Mm2*. *?Q C' C Mm' Mm2*)
⟨*proof*⟩

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 ⊢ *C sees M*: *Ts*→*T* = *m in D* ≡
∃ *Mm*. *P* ⊢ *C sees-methods Mm* ∧ *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*→T = *m in D*

lemma *has-methodI*:
$P \vdash C$ *sees* M:*Ts*→T = *m in D* ⟹ $P \vdash C$ *has M*
⟨*proof*⟩

lemma *sees-method-fun*:
⟦$P \vdash C$ *sees* M:*TS*→T = *m in D*; $P \vdash C$ *sees* M:*TS′*→T' = *m′ in D′* ⟧
⟹ *TS′* = *TS* ∧ T' = T ∧ m' = m ∧ D' = D
⟨*proof*⟩
lemma *sees-method-decl-above*:
$P \vdash C$ *sees* M:*Ts*→T = *m in D* ⟹ $P \vdash C \preceq^* D$
⟨*proof*⟩
lemma *visible-method-exists*:
$P \vdash C$ *sees* M:*Ts*→T = *m in D* ⟹
∃ *D′ fs ms*. *class P D* = *Some*(*D′*,*fs*,*ms*) ∧ *map-of ms M* = *Some*(*Ts*,*T*,*m*)⟨*proof*⟩

lemma *sees-method-idemp*:
$P \vdash C$ *sees* M:*Ts*→T=*m in D* ⟹ $P \vdash D$ *sees* M:*Ts*→T=*m in D*
⟨*proof*⟩
lemma *sees-method-decl-mono*:
⟦ $P \vdash C' \preceq^* C$; $P \vdash C$ *sees* M:*Ts*→T = *m in D*;
$P \vdash C'$ *sees* M:*Ts′*→T' = *m′ in D′* ⟧ ⟹ $P \vdash D' \preceq^* D$
⟨*proof*⟩

lemma *sees-method-is-class*:
$P \vdash C$ *sees* M:*Ts*→T = *m in D* ⟹ *is-class P C*
⟨*proof*⟩

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*; $P \vdash D$ *has-fields FDTs*;
FDTs′ = *map* (λ(*F*,*Tm*). ((*F*,*C*),*Tm*)) *fs* @ *FDTs* ⟧
⟹ $P \vdash C$ *has-fields FDTs′*

| *has-fields-Object*:
⟦ *class P Object* = *Some*(*D*,*fs*,*ms*); *FDTs* = *map* (λ(*F*,*T*). ((*F*,*Object*),*T*)) *fs* ⟧
⟹ $P \vdash$ *Object has-fields FDTs*

lemma *has-fields-fun*:
assumes $P \vdash C$ *has-fields FDTs* **and** $P \vdash C$ *has-fields FDTs′*
shows *FDTs′* = *FDTs*
⟨*proof*⟩

lemma *all-fields-in-has-fields*:

assumes $P \vdash C$ *has-fields FDTs*
and $P \vdash C \preceq^* D$ *class P D* $=$ *Some*(D',fs,ms) $(F,Tm) \in$ *set fs*
shows $((F,D),Tm) \in$ *set FDTs*
⟨*proof*⟩

lemma *has-fields-decl-above*:
assumes $P \vdash C$ *has-fields FDTs* $((F,D),Tm) \in$ *set FDTs*
shows $P \vdash C \preceq^* D$
⟨*proof*⟩

lemma *subcls-notin-has-fields*:
assumes $P \vdash C$ *has-fields FDTs* $((F,D),Tm) \in$ *set FDTs*
shows $\neg$ (*subcls1 P*)$^{++}$ *D C*
⟨*proof*⟩

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*) $= \{\}$
⟨*proof*⟩

lemma *has-fields-is-class*:
$P \vdash C$ *has-fields FDTs* $\Longrightarrow$ *is-class P C*
⟨*proof*⟩

lemma *Object-has-fields-Object*:
assumes $P \vdash$ *Object has-fields FDTs*
shows *snd ‘ fst ‘ set FDTs* $\subseteq$ {*Object*}
⟨*proof*⟩

definition
has-field :: *'m prog* $\Rightarrow$ *cname* $\Rightarrow$ *vname* $\Rightarrow$ *ty* $\Rightarrow$ *fmod* $\Rightarrow$ *cname* $\Rightarrow$ *bool*
(‹- $\vdash$ - *has* -:- '(-') *in* -› [*51*,*51*,*51*,*51*,*51*,*51*] *50*)
where
$P \vdash C$ *has* $F{:}T$ (*fm*) *in* D $\equiv$
$\exists$ *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*
⟨*proof*⟩

lemma *has-field-is-class*:
$P \vdash C$ *has* $M{:}T$ (*fm*) *in D* $\Longrightarrow$ *is-class P C*
⟨*proof*⟩

lemma *has-field-decl-above*:
$P \vdash C$ *has* $F{:}T$ (*fm*) *in D* $\Longrightarrow$ $P \vdash C \preceq^* D$
⟨*proof*⟩

lemma *has-field-fun*:
⟦$P \vdash C$ *has* $F{:}T$ (*fm*) *in D*; $P \vdash C$ *has* $F{:}T'$ (*fm'*) *in D*⟧ $\Longrightarrow$ $T' = T \wedge$ *fm* $=$ *fm'*
⟨*proof*⟩

definition
sees-field :: *'m prog* $\Rightarrow$ *cname* $\Rightarrow$ *vname* $\Rightarrow$ *ty* $\Rightarrow$ *fmod* $\Rightarrow$ *cname* $\Rightarrow$ *bool*

$(\langle\text{-} \vdash \text{- } sees \text{ -:- } '(\text{-}') \text{ } in \text{ -}\rangle \text{ } [51,51,51,51,51,51] \text{ } 50)$

where
$P \vdash C \; sees \; F{:}T \; (fm) \; in \; D \;\equiv$
$\exists FDTs. \; P \vdash C \; has\text{-}fields \; FDTs \wedge$
$map\text{-}of \; (map \; (\lambda((F,D),Tm). \; (F,(D,Tm))) \; FDTs) \; F = Some(D,T,fm)$

lemma *map-of-remap-SomeD*:
$map\text{-}of \; (map \; (\lambda((k,k'),x). \; (k,(k',x))) \; t) \; k = Some \; (k',x) \Longrightarrow map\text{-}of \; t \; (k, \; k') = Some \; x$
⟨*proof*⟩

lemma *has-visible-field*:
$P \vdash C \; sees \; F{:}T \; (fm) \; in \; D \Longrightarrow P \vdash C \; has \; F{:}T \; (fm) \; in \; D$
⟨*proof*⟩

lemma *sees-field-fun*:
$[\![P \vdash C \; sees \; F{:}T \; (fm) \; in \; D; \; P \vdash C \; sees \; F{:}T' \; (fm') \; in \; D']\!] \Longrightarrow T' = T \wedge D' = D \wedge fm = fm'$
⟨*proof*⟩

lemma *sees-field-decl-above*:
$P \vdash C \; sees \; F{:}T \; (fm) \; in \; D \Longrightarrow P \vdash C \preceq^* D$
⟨*proof*⟩

lemma *sees-field-idemp*:
assumes $P \vdash C \; sees \; F{:}T \; (fm) \; in \; D$
shows $P \vdash D \; sees \; F{:}T \; (fm) \; in \; D$
⟨*proof*⟩

3.3.5 Functional lookup

definition $method :: \; 'm \; prog \Rightarrow cname \Rightarrow mname \Rightarrow cname \times ty \; list \times ty \times \; 'm \; option$
where $method \; P \; C \; M \;\equiv\; THE \; (D,Ts,T,m). \; P \vdash C \; sees \; M{:}Ts \rightarrow T = m \; in \; D$

definition $field \; :: \; 'm \; prog \Rightarrow cname \Rightarrow vname \Rightarrow cname \times ty \times fmod$
where $field \; P \; C \; F \;\equiv\; THE \; (D,T,fm). \; P \vdash C \; sees \; F{:}T \; (fm) \; in \; D$

definition $fields :: \; 'm \; prog \Rightarrow cname \Rightarrow ((vname \times cname) \times (ty \times fmod)) \; list$
where $fields \; P \; C \;\equiv\; THE \; FDTs. \; P \vdash C \; has\text{-}fields \; FDTs$

lemma [*simp*]: $P \vdash C \; has\text{-}fields \; FDTs \Longrightarrow fields \; P \; C = FDTs$⟨*proof*⟩
lemma *field-def2* [*simp*]: $P \vdash C \; sees \; F{:}T \; (fm) \; in \; D \Longrightarrow field \; P \; C \; F = (D,T,fm)$⟨*proof*⟩
lemma *method-def2* [*simp*]: $P \vdash C \; sees \; M{:} \; Ts{\rightarrow}T = m \; in \; D \Longrightarrow method \; P \; C \; M = (D,Ts,T,m)$⟨*proof*⟩
lemma *has-fields-b-fields*:
$P \vdash C \; has\text{-}fields \; FDTs \Longrightarrow fields \; P \; C = FDTs$
⟨*proof*⟩

lemma *has-field-map-of-fields* [*simp*]:
$P \vdash C \; has \; F{:}T \; (fm) \; in \; D \Longrightarrow map\text{-}of \; (fields \; P \; C) \; (F, \; D) = \lfloor(T, \; fm)\rfloor$
⟨*proof*⟩

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*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow bool$)
subcls1
⟨*proof*⟩

Introduce proper constant *subcls′* for *subcls* and generate executable equation for *subcls′*

definition *subcls′* **where** *subcls′* = *subcls*

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow bool$)
[*inductify*]
subcls′
⟨*proof*⟩

lemma *subcls-conv-subcls′* [*code-unfold*]:
(*subcls1 P*)^** = *subcls′ P*
⟨*proof*⟩

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*
⟨*proof*⟩

lemmas [*code-pred-intro*] =
widen-refl widen-subcls widen-null widen-null-array widen-array-object-code widen-array-array
code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$)
widen
⟨*proof*⟩

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*)
⟨*proof*⟩

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$, $i \Rightarrow i \Rightarrow o \Rightarrow bool$)
Methods
⟨*proof*⟩

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$, $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow bool$)
[*inductify*]

Method
⟨*proof*⟩

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$)
[*inductify*]
has-method
⟨*proof*⟩

declare *fun-upd-def* [*code-pred-inline*]

code-pred
(*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$)
Fields
⟨*proof*⟩

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow bool$)
[*inductify*, *skip-proof*]
has-field
⟨*proof*⟩

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$, $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow i \Rightarrow bool$)
[*inductify*, *skip-proof*]
sees-field
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *eval-Fields-conv*:
Predicate.eval (*Fields-i-i-o P C*) = (λ*FDTs*. *P* ⊢ *C has-fields FDTs*)

⟨*proof*⟩

lemma *fields-code* [*code*]:
fields P C = *Predicate.the* (*Fields-i-i-o P C*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Addr-not-default-val*: *Addr a* ≠ *default-val T*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma [*iff*]: *is-Ref Null* ⟨*proof*⟩

definition *undefined-value* :: *'addr val* **where** *undefined-value* = *Unit*

lemma *undefined-value-not-Addr*:
undefined-value ≠ *Addr a Addr a* ≠ *undefined-value*
⟨*proof*⟩

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* $\preceq^*$ *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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *OutOfMemory-not-Object*[*simp*]: *OutOfMemory* ≠ *Object*
⟨*proof*⟩

lemma *ClassCast-not-Object*[*simp*]: *ClassCast* ≠ *Object*
⟨*proof*⟩

lemma *NullPointer-not-Object*[*simp*]: *NullPointer* ≠ *Object*
⟨*proof*⟩

lemma *ArrayIndexOutOfBounds-not-Object*[*simp*]: *ArrayIndexOutOfBounds* ≠ *Object*
⟨*proof*⟩

lemma *ArrayStore-not-Object*[*simp*]: *ArrayStore* ≠ *Object*
⟨*proof*⟩

lemma *NegativeArraySize-not-Object*[*simp*]: *NegativeArraySize* ≠ *Object*
⟨*proof*⟩

lemma *ArithmeticException-not-Object*[*simp*]: *ArithmeticException* ≠ *Object*
⟨*proof*⟩

lemma *IllegalMonitorState-not-Object*[*simp*]: *IllegalMonitorState* ≠ *Object*
⟨*proof*⟩

lemma *IllegalThreadState-not-Object*[*simp*]: *IllegalThreadState* ≠ *Object*
⟨*proof*⟩

lemma *InterruptedException-not-Object*[*simp*]: *InterruptedException* ≠ *Object*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *wf-syscls-xcpt-subcls-Throwable*:
  $[\![\ C \in$ *sys-xcpts*; *wf-syscls P* $]\!] \Longrightarrow P \vdash C \preceq^*$ *Throwable*
⟨*proof*⟩

lemma *wf-syscls-is-class-Throwable*:
  wf-syscls P $\Longrightarrow$ *is-class P Throwable*
⟨*proof*⟩

lemma *wf-syscls-is-class-sub-Throwable*:
  $[\![$ *wf-syscls P*; $P \vdash C \preceq^*$ *Throwable* $]\!] \Longrightarrow$ *is-class P C*
⟨*proof*⟩

lemma *wf-syscls-is-class-xcpt*:
  $[\![\ C \in$ *sys-xcpts*; *wf-syscls P* $]\!] \Longrightarrow$ *is-class P C*
⟨*proof*⟩

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*)
⟨*proof*⟩

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* ≡ (*Throwable*, (*Object*, [], []))

definition *NullPointerC* :: *'m cdecl*
where [*code-unfold*]: *NullPointerC* ≡ (*NullPointer*, (*Throwable*,[],[]))

definition *ClassCastC* :: *'m cdecl*
where [*code-unfold*]: *ClassCastC* ≡ (*ClassCast*, (*Throwable*,[],[]))

definition *OutOfMemoryC* :: *'m cdecl*
where [*code-unfold*]: *OutOfMemoryC* ≡ (*OutOfMemory*, (*Throwable*,[],[]))

definition *ArrayIndexOutOfBoundsC* :: *'m cdecl*
where [*code-unfold*]: *ArrayIndexOutOfBoundsC* ≡ (*ArrayIndexOutOfBounds*, (*Throwable*,[],[]))

definition *ArrayStoreC* :: *'m cdecl*
where [*code-unfold*]: *ArrayStoreC* ≡ (*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*
⟨*proof*⟩

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*$_{\text{-}}$›)
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* $\subseteq_m$ *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)
⟨*proof*⟩

lemma *typeof-lit-eq-Integer* [*simp*]: (*typeof* v = *Some Integer*) = (∃ i. v = *Intg* i)
⟨*proof*⟩

lemma *typeof-lit-eq-NT* [*simp*]: (*typeof* v = *Some NT*) = (v = *Null*)
⟨*proof*⟩

lemma *typeof-lit-eq-Void* [*simp*]: *typeof* v = *Some Void* ⟷ v = *Unit*
⟨*proof*⟩

lemma *typeof-lit-neq-Class* [*simp*]: *typeof* v ≠ *Some* (*Class* C)
⟨*proof*⟩

lemma *typeof-lit-neq-Array* [*simp*]: *typeof* v ≠ *Some* (*Array* T)
⟨*proof*⟩

lemma *typeof-NoneD* [*simp*,*dest*]:
typeof v = *Some* x ⟹ ¬ *is-Addr* v
⟨*proof*⟩

lemma *typeof-lit-is-type*:
typeof v = *Some* T ⟹ *is-type* P T
⟨*proof*⟩

context *heap-base* **begin**

lemma *typeof-h-eq-Boolean* [*simp*]: ($typeof_h$ v = *Some Boolean*) = (∃ b. v = *Bool* b)
⟨*proof*⟩

lemma *typeof-h-eq-Integer* [*simp*]: ($typeof_h$ v = *Some Integer*) = (∃ i. v = *Intg* i)

⟨*proof*⟩

lemma *typeof-h-eq-NT* [*simp*]: (*typeof*$_h$ *v* = *Some NT*) = (*v* = *Null*)
⟨*proof*⟩

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′*
⟨*proof*⟩

lemma *hext-objD*:
assumes *h* ⊴ *h′*
and *typeof-addr h a* = ⌊*Class-type C*⌋
shows *typeof-addr h′ a* = ⌊*Class-type C*⌋
⟨*proof*⟩

lemma *hext-arrD*:
assumes *h* ⊴ *h′ typeof-addr h a* = ⌊*Array-type T n*⌋
shows *typeof-addr h′ a* = ⌊*Array-type T n*⌋
⟨*proof*⟩

lemma *hext-refl* [*iff*]: *h* ⊴ *h*
⟨*proof*⟩

lemma *hext-trans* [*trans*]: ⟦ *h* ⊴ *h′*; *h′* ⊴ *h″* ⟧ ⟹ *h* ⊴ *h″*
⟨*proof*⟩

lemma *typeof-lit-typeof*:
typeof v = ⌊*T*⌋ ⟹ *typeof*$_h$ *v* = ⌊*T*⌋
⟨*proof*⟩

lemma *addr-loc-type-fun*:
⟦ *P*,*h* ⊢ *a*@*al* : *T*; *P*,*h* ⊢ *a*@*al* : *T′* ⟧ ⟹ *T* = *T′*
⟨*proof*⟩

lemma *THE-addr-loc-type*:
P,*h* ⊢ *a*@*al* : *T* ⟹ (*THE T*. *P*,*h* ⊢ *a*@*al* : *T*) = *T*
⟨*proof*⟩

lemma *typeof-addr-locI* [*simp*]:
P,*h* ⊢ *a*@*al* : *T* ⟹ *typeof-addr-loc P h a al* = *T*
⟨*proof*⟩

lemma *deterministic-heap-opsI*:
⟦ ⋀*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* ⟧
⟹ *deterministic-heap-ops*
⟨*proof*⟩

lemma *deterministic-heap-ops-readD*:

$\llbracket$ *deterministic-heap-ops*; *heap-read h ad al v*; *heap-read h ad al v′* $\rrbracket \Longrightarrow v = v'$
⟨*proof*⟩

lemma *deterministic-heap-ops-writeD*:
$\llbracket$ *deterministic-heap-ops*; *heap-write h ad al v h′*; *heap-write h ad al v h′′* $\rrbracket \Longrightarrow h' = h''$
⟨*proof*⟩

lemma *deterministic-heap-ops-allocateD*:
$\llbracket$ *deterministic-heap-ops*; $(h', a) \in$ *allocate h hT*; $(h'', a') \in$ *allocate h hT* $\rrbracket \Longrightarrow h' = h'' \wedge a = a'$
⟨*proof*⟩

lemma *deterministic-heap-ops-no-spurious-wakeups*:
deterministic-heap-ops $\Longrightarrow \neg$ *spurious-wakeups*
⟨*proof*⟩

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*:
$\llbracket$ *typeof-addr h a* = ⌊*Class-type C*⌋; $P \vdash C \preceq^* $ *Thread* $\rrbracket \Longrightarrow$ *thread-id2addr* (*addr2thread-id a*) = *a*
begin

lemma *typeof-addr-thread-id2-addr-addr2thread-id* [*simp*]:
$\llbracket$ *typeof-addr h a* = ⌊*Class-type C*⌋; $P \vdash C \preceq^*$ *Thread* $\rrbracket \Longrightarrow$ *typeof-addr h* (*thread-id2addr* (*addr2thread-id a*)) = ⌊*Class-type C*⌋
⟨*proof*⟩

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*⌋
⟨*proof*⟩

lemma *hext-typeof-mono*:
⟦ *h* ⊴ *h'*; $typeof_h$ *v* = *Some T* ⟧ ⟹ $typeof_{h'}$ *v* = *Some T*
⟨*proof*⟩

lemma *addr-loc-type-hext-mono*:
⟦ *P*,*h* ⊢ *a*@*al* : *T*; *h* ⊴ *h'* ⟧ ⟹ *P*,*h'* ⊢ *a*@*al* : *T*
⟨*proof*⟩

lemma *type-of-hext-type-of*: — FIXME: What's this rule good for?
⟦ $typeof_h$ *w* = ⌊*T*⌋; *hext h h'* ⟧ ⟹ $typeof_{h'}$ *w* = ⌊*T*⌋
⟨*proof*⟩

lemma *hext-None*: ⟦ *h* ⊴ *h'*; *typeof-addr h' a* = *None* ⟧ ⟹ *typeof-addr h a* = *None*
⟨*proof*⟩

lemma *map-typeof-hext-mono*:
⟦ *map* $typeof_h$ *vs* = *map Some Ts*; *h* ⊴ *h'* ⟧ ⟹ *map* $typeof_{h'}$ *vs* = *map Some Ts*
⟨*proof*⟩

lemma *hext-typeof-addr-map-le*:
h ⊴ *h'* ⟹ *typeof-addr h* $\subseteq_m$ *typeof-addr h'*
⟨*proof*⟩

lemma *hext-dom-typeof-addr-subset*:
h ⊴ *h'* ⟹ *dom* (*typeof-addr h*) ⊆ *dom* (*typeof-addr h'*)
⟨*proof*⟩

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*⟩

type-synonym
(*'addr*, *'thread-id*, *'x*, *'heap*) *Jinja-thread-action* =
(*'addr*,*'thread-id*,*'x*,*'heap*,*'addr*,(*'addr*, *'thread-id*) *obs-event*) *thread-action*

⟨*ML*⟩
typ (*'addr*, *'thread-id*, *'x*, *'heap*) *Jinja-thread-action*

lemma *range-ty-of-htype*: *range ty-of-htype* ⊆ *range Class* ∪ *range Array*
⟨*proof*⟩

lemma *some-choice*: (∃ *a*. ∀ *b*. *P b* (*a b*)) ⟷ (∀ *b*. ∃ *a*. *P b a*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *set-convert-RA-not-Read* [*simp*]:
⋀*ln*. *ReadMem ad al v* ∉ *set* (*convert-RA ln*)
⟨*proof*⟩

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 $(h,\ ads,\ b)\ C =$
$(if\ b$
$then\ let\ HA = allocate\ h\ (Class\text{-}type\ C)$
$in\ if\ HA = \{\}\ then\ (h,\ ads,\ False)$
$else\ let\ (h',\ a'') = SOME\ ha.\ ha \in HA\ in\ (h',\ ads\ @\ [a''],\ True)$
$else\ (h,\ ads,\ False))$
⟨*proof*⟩

lemma *create-initial-object-False* [*simp*]:
create-initial-object $(h,\ ads,\ False)\ C = (h,\ ads,\ False)$
⟨*proof*⟩

lemma *foldl-create-initial-object-False* [*simp*]:
foldl create-initial-object $(h,\ ads,\ False)\ Cs = (h,\ ads,\ False)$
⟨*proof*⟩

lemma *NewHeapElem-start-heap-obs-start-addrsD*:
NewHeapElem a CTn $\in$ *set start-heap-obs* $\Longrightarrow a \in$ *set start-addrs*
⟨*proof*⟩

lemma *shr-start-state*: *shr* (*start-state f P C M vs*) = *start-heap*
⟨*proof*⟩

lemma *start-heap-obs-not-Read*:
ReadMem ad al v $\notin$ *set start-heap-obs*
⟨*proof*⟩

lemma *length-initialization-list-le-length-start-addrs*:
length initialization-list $\geq$ *length start-addrs*
⟨*proof*⟩

lemma (**in** −) *distinct-initialization-list*:
distinct initialization-list
⟨*proof*⟩

lemma (**in** −) *wf-syscls-initialization-list-is-class*:
⟦ *wf-syscls P*; $C \in$ *set initialization-list* ⟧ $\Longrightarrow$ *is-class P C*
⟨*proof*⟩

lemma *start-addrs-NewHeapElem-start-heap-obsD*:
$a \in$ *set start-addrs* $\Longrightarrow \exists$ *CTn. NewHeapElem a CTn* $\in$ *set start-heap-obs*
⟨*proof*⟩

lemma *in-set-start-addrs-conv-NewHeapElem*:
$a \in$ *set start-addrs* $\longleftrightarrow$ ($\exists$ *CTn. NewHeapElem a CTn* $\in$ *set start-heap-obs*)
⟨*proof*⟩

3.9.1 *preallocated*

definition *preallocated* :: *'heap* $\Rightarrow$ *bool*
where *preallocated h* $\equiv \forall C \in$ *sys-xcpts. typeof-addr h* (*addr-of-sys-xcpt C*) = ⌊*Class-type C*⌋

lemma *typeof-addr-sys-xcp*:
  $\llbracket$ *preallocated h*; $C \in$ *sys-xcpts* $\rrbracket \Longrightarrow$ *typeof-addr h* (*addr-of-sys-xcpt C*) $= \lfloor$*Class-type C*$\rfloor$
⟨*proof*⟩

lemma *typeof-sys-xcp*:
  $\llbracket$ *preallocated h*; $C \in$ *sys-xcpts* $\rrbracket \Longrightarrow$ *typeof*$_h$ (*Addr* (*addr-of-sys-xcpt C*)) $= \lfloor$*Class C*$\rfloor$
⟨*proof*⟩

lemma *addr-of-sys-xcpt-start-addr*:
  $\llbracket$ *start-heap-ok*; $C \in$ *sys-xcpts* $\rrbracket \Longrightarrow$ *addr-of-sys-xcpt C* $\in$ *set start-addrs*
⟨*proof*⟩

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 Out-OfMemory*)
  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*)
⟨*proof*⟩

lemma *cname-of-xcp* [*simp*]:
  $\llbracket$ *preallocated h*; $C \in$ *sys-xcpts* $\rrbracket \Longrightarrow$ *cname-of h* (*addr-of-sys-xcpt C*) $= C$
⟨*proof*⟩

lemma *preallocated-hext*:
  $\llbracket$ *preallocated h*; $h \trianglelefteq h'$ $\rrbracket \Longrightarrow$ *preallocated* h'
⟨*proof*⟩

end

context *heap* **begin**

lemma *preallocated-heap-ops*:
  assumes *preallocated h*
  shows *preallocated-allocate*: $\bigwedge a.$ $(h', a) \in$ *allocate h hT* $\Longrightarrow$ *preallocated* h'
  and *preallocated-write-field*: *heap-write h a al v* $h' \Longrightarrow$ *preallocated* h'
⟨*proof*⟩

lemma *not-empty-pairE*: $\llbracket$ $A \neq \{\}$; $\bigwedge a\ b.$ $(a, b) \in A \Longrightarrow$ *thesis* $\rrbracket \Longrightarrow$ *thesis*

⟨*proof*⟩

lemma *allocate-not-emptyI*: $(h', a) \in allocate\ h\ hT \Longrightarrow allocate\ h\ hT \neq \{\}$
⟨*proof*⟩

lemma *allocate-Eps*:
$[\![\ (h'', a'') \in allocate\ h\ hT;\ (SOME\ ha.\ ha \in allocate\ h\ hT) = (h', a')\ ]\!] \Longrightarrow (h', a') \in allocate\ h\ hT$
⟨*proof*⟩

lemma *preallocated-start-heap*:
$[\![\ start\text{-}heap\text{-}ok;\ wf\text{-}syscls\ P\ ]\!] \Longrightarrow preallocated\ start\text{-}heap$
⟨*proof*⟩

lemma *start-tid-start-addrs*:
$[\![\ wf\text{-}syscls\ P;\ start\text{-}heap\text{-}ok\ ]\!] \Longrightarrow thread\text{-}id2addr\ start\text{-}tid \in set\ start\text{-}addrs$
⟨*proof*⟩

lemma
assumes *wf-syscls P*
shows *dom-typeof-addr-start-heap*: $set\ start\text{-}addrs \subseteq dom\ (typeof\text{-}addr\ start\text{-}heap)$
and *distinct-start-addrs*: *distinct start-addrs*
⟨*proof*⟩

lemma *NewHeapElem-start-heap-obsD*:
assumes *wf-syscls P*
and $NewHeapElem\ a\ hT \in set\ start\text{-}heap\text{-}obs$
shows $typeof\text{-}addr\ start\text{-}heap\ a = \lfloor hT \rfloor$
⟨*proof*⟩

end

3.9.2 Code generation

definition *pick-addr* :: $('heap \times 'addr)\ set \Rightarrow 'heap \times 'addr$
where $pick\text{-}addr\ HA = (SOME\ ha.\ ha \in HA)$

lemma *pick-addr-code* [*code*]:
$pick\text{-}addr\ (set\ [ha]) = ha$
⟨*proof*⟩

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))
```
⟨*proof*⟩

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* :: $'m\ prog \Rightarrow 'heap \Rightarrow 'addr\ val \Rightarrow ty \Rightarrow bool$ (‹-,- ⊢ - :≤ -› [51,51,51,51] 50)
where $P,h \vdash v :\leq T \equiv \exists T'.\ typeof_h\ v = Some\ T' \wedge P \vdash T' \leq T$

definition *lconf* :: $'m\ prog \Rightarrow 'heap \Rightarrow (vname \rightharpoonup 'addr\ val) \Rightarrow (vname \rightharpoonup ty) \Rightarrow bool$ (‹-,- ⊢ - '(:≤') -› [51,51,51,51] 50)
where $P,h \vdash l\ (:\leq)\ E \equiv \forall V\ v.\ l\ V = Some\ v \longrightarrow (\exists T.\ E\ V = Some\ T \wedge P,h \vdash v :\leq T)$

abbreviation *confs* :: $'m\ prog \Rightarrow 'heap \Rightarrow 'addr\ val\ list \Rightarrow ty\ list \Rightarrow bool$ (‹-,- ⊢ - [:≤] -› [51,51,51,51] 50)
where $P,h \vdash vs\ [:\leq]\ Ts == list\text{-}all2\ (conf\ P\ h)\ vs\ Ts$

definition *tconf* :: $'m\ prog \Rightarrow 'heap \Rightarrow 'thread\text{-}id \Rightarrow bool$ (‹-,- ⊢ - √t› [51,51,51] 50)
where $P,h \vdash t \sqrt{t} \equiv \exists C.\ typeof\text{-}addr\ h\ (thread\text{-}id2addr\ t) = \lfloor Class\text{-}type\ C \rfloor \wedge P \vdash C \preceq^* Thread$

end

locale *heap-conf-base* =
heap-base +
constrains $addr2thread\text{-}id :: ('addr :: addr) \Rightarrow 'thread\text{-}id$
and $thread\text{-}id2addr :: 'thread\text{-}id \Rightarrow 'addr$
and $spurious\text{-}wakeups :: bool$
and $empty\text{-}heap :: 'heap$
and $allocate :: 'heap \Rightarrow htype \Rightarrow ('heap \times 'addr)\ set$
and $typeof\text{-}addr :: 'heap \Rightarrow 'addr \rightharpoonup htype$
and $heap\text{-}read :: 'heap \Rightarrow 'addr \Rightarrow addr\text{-}loc \Rightarrow 'addr\ val \Rightarrow bool$
and $heap\text{-}write :: 'heap \Rightarrow 'addr \Rightarrow addr\text{-}loc \Rightarrow 'addr\ val \Rightarrow 'heap \Rightarrow bool$
fixes $hconf :: 'heap \Rightarrow bool$
and $P :: 'm\ prog$

sublocale *heap-conf-base* < *prog P* ⟨*proof*⟩

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 :≤

context *heap-base* **begin**

lemma *conf-Null* [*simp*]: *P*,*h* ⊢ *Null* :≤ *T* = *P* ⊢ *NT* ≤ *T*
⟨*proof*⟩

lemma *typeof-conf*[*simp*]: $typeof_h$ *v* = *Some T* ⟹ *P*,*h* ⊢ *v* :≤ *T*
⟨*proof*⟩

lemma *typeof-lit-conf*[*simp*]: *typeof v* = *Some T* ⟹ *P*,*h* ⊢ *v* :≤ *T*
⟨*proof*⟩

lemma *defval-conf*[*simp*]: *P*,*h* ⊢ *default-val T* :≤ *T*
⟨*proof*⟩

lemma *conf-widen*: *P*,*h* ⊢ *v* :≤ *T* ⟹ *P* ⊢ *T* ≤ *T'* ⟹ *P*,*h* ⊢ *v* :≤ *T'*
⟨*proof*⟩

lemma *conf-sys-xcpt*:
  $[\![preallocated\ h;\ C \in sys\text{-}xcpts]\!] \Longrightarrow P,h \vdash Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ C) :\leq Class\ C$
⟨*proof*⟩

lemma *conf-NT* [*iff*]: $P,h \vdash v :\leq NT = (v = Null)$
⟨*proof*⟩

lemma *is-IntgI*: $P,h \vdash v :\leq Integer \Longrightarrow is\text{-}Intg\ v$
⟨*proof*⟩

lemma *is-BoolI*: $P,h \vdash v :\leq Boolean \Longrightarrow is\text{-}Bool\ v$
⟨*proof*⟩

lemma *is-RefI*: $P,h \vdash v :\leq T \Longrightarrow is\text{-}refT\ T \Longrightarrow is\text{-}Ref\ v$
⟨*proof*⟩

lemma *non-npD*:
  $[\![\ v \neq Null;\ P,h \vdash v :\leq Class\ C;\ C \neq Object\ ]\!]$
  $\Longrightarrow \exists\, a\ C'.\ v = Addr\ a \wedge typeof\text{-}addr\ h\ a = \lfloor Class\text{-}type\ C' \rfloor \wedge P \vdash C' \preceq^* C$
⟨*proof*⟩

lemma *non-npD2*:
  $[\![v \neq Null;\ P,h \vdash v :\leq Class\ C\ ]\!]$
  $\Longrightarrow \exists\, a\ hT.\ v = Addr\ a \wedge typeof\text{-}addr\ h\ a = \lfloor hT \rfloor \wedge P \vdash class\text{-}type\text{-}of\ hT \preceq^* C$
⟨*proof*⟩

end

context *heap* **begin**

lemma *conf-hext*: $[\![\ h \trianglelefteq h';\ P,h \vdash v :\leq T\ ]\!] \Longrightarrow P,h' \vdash v :\leq T$
⟨*proof*⟩

lemma *conf-heap-ops-mono*:
  assumes $P,h \vdash v :\leq T$
  shows *conf-allocate-mono*: $(h',\ a) \in allocate\ h\ hT \Longrightarrow P,h' \vdash v :\leq T$
  and *conf-heap-write-mono*: $heap\text{-}write\ h\ a\ al\ v'\ h' \Longrightarrow P,h' \vdash v :\leq T$
⟨*proof*⟩

end

3.10.2 Value list conformance [:≤]

context *heap-base* **begin**

lemma *confs-widens* [*trans*]: $[\![P,h \vdash vs\ [:\leq]\ Ts;\ P \vdash Ts\ [\leq]\ Ts']\!] \Longrightarrow P,h \vdash vs\ [:\leq]\ Ts'$
⟨*proof*⟩

lemma *confs-rev*: $P,h \vdash rev\ s\ [:\leq]\ t = (P,h \vdash s\ [:\leq]\ rev\ t)$
⟨*proof*⟩

lemma *confs-conv-map*:
  $P,h \vdash vs\ [:\leq]\ Ts' = (\exists\, Ts.\ map\ typeof_h\ vs = map\ Some\ Ts \wedge P \vdash Ts\ [\leq]\ Ts')$

⟨*proof*⟩

lemma *confs-Cons2*: $P,h \vdash xs\ [:\leq]\ y\#ys = (\exists z\ zs.\ xs = z\#zs \wedge P,h \vdash z :\leq y \wedge P,h \vdash zs\ [:\leq]\ ys)$
⟨*proof*⟩

end

context *heap* **begin**

lemma *confs-hext*: $P,h \vdash vs\ [:\leq]\ Ts \Longrightarrow h \trianglelefteq h' \Longrightarrow P,h' \vdash vs\ [:\leq]\ Ts$
⟨*proof*⟩

end

3.10.3 Local variable conformance

context *heap-base* **begin**

lemma *lconf-upd*:
$[\![P,h \vdash l\ (:\leq)\ E;\ P,h \vdash v :\leq T;\ E\ V = Some\ T]\!] \Longrightarrow P,h \vdash l(V \mapsto v)\ (:\leq)\ E$
⟨*proof*⟩

lemma *lconf-empty* [*iff*]: $P,h \vdash Map.empty\ (:\leq)\ E$
⟨*proof*⟩

lemma *lconf-upd2*: $[\![P,h \vdash l\ (:\leq)\ E;\ P,h \vdash v :\leq T]\!] \Longrightarrow P,h \vdash l(V \mapsto v)\ (:\leq)\ E(V \mapsto T)$
⟨*proof*⟩

end

context *heap* **begin**

lemma *lconf-hext*: $[\![P,h \vdash l\ (:\leq)\ E;\ h \trianglelefteq h']\!] \Longrightarrow P,h' \vdash l\ (:\leq)\ E$
⟨*proof*⟩

end

3.10.4 Thread object conformance

context *heap-base* **begin**

lemma *tconfI*: $[\![typeof\text{-}addr\ h\ (thread\text{-}id2addr\ t) = \lfloor Class\text{-}type\ C \rfloor;\ P \vdash C \preceq^* Thread]\!] \Longrightarrow P,h \vdash t\ \sqrt{t}$
⟨*proof*⟩

lemma *tconfD*: $P,h \vdash t\ \sqrt{t} \Longrightarrow \exists C.\ typeof\text{-}addr\ h\ (thread\text{-}id2addr\ t) = \lfloor Class\text{-}type\ C \rfloor \wedge P \vdash C \preceq^* Thread$
⟨*proof*⟩

end

context *heap* **begin**

lemma *tconf-hext-mono*: $[\![P,h \vdash t\ \sqrt{t};\ h \trianglelefteq h']\!] \Longrightarrow P,h' \vdash t\ \sqrt{t}$

⟨*proof*⟩

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*: $heap\text{-}write\ h\ a\ al\ v\ h' \Longrightarrow P,h' \vdash t \surd t$
⟨*proof*⟩

lemma *tconf-start-heap-start-tid*:
$[\![\ start\text{-}heap\text{-}ok;\ wf\text{-}syscls\ P\ ]\!] \Longrightarrow P,start\text{-}heap \vdash start\text{-}tid \surd t$
⟨*proof*⟩

lemma *start-heap-write-typeable*:
assumes $WriteMem\ ad\ al\ v \in set\ start\text{-}heap\text{-}obs$
shows $\exists T.\ P,start\text{-}heap \vdash ad@al : T \wedge P,start\text{-}heap \vdash v :\leq T$
⟨*proof*⟩

end

3.10.5 Well-formed start state

context *heap-base* **begin**

inductive *wf-start-state* :: $'m\ prog \Rightarrow cname \Rightarrow mname \Rightarrow 'addr\ val\ list \Rightarrow bool$
for P :: $'m\ prog$ **and** C :: *cname* **and** M :: *mname* **and** *vs* :: $'addr\ val\ list$
where
wf-start-state:
$[\![\ P \vdash C\ sees\ M{:}Ts{\rightarrow}T = \lfloor meth \rfloor\ in\ D;\ start\text{-}heap\text{-}ok;\ P,start\text{-}heap \vdash vs\ [:\leq]\ Ts\ ]\!]$
$\Longrightarrow wf\text{-}start\text{-}state\ P\ C\ M\ vs$

end

end

3.11 Semantics of method calls that cannot be defined inside JinjaThreads

theory *ExternalCall*
imports
../Framework/FWSemantics
Conform
begin

type-synonym
$('addr,'thread\text{-}id,'heap)\ external\text{-}thread\text{-}action = ('addr,\ 'thread\text{-}id,\ cname \times mname \times 'addr,'heap)$
$Jinja\text{-}thread\text{-}action$

⟨*ML*⟩
typ $('addr,'thread\text{-}id,'heap)\ external\text{-}thread\text{-}action$

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*
⟨*proof*⟩

inductive (**in** *heap-base*) *external-WT′* :: ′*m prog* ⇒ ′*heap* ⇒ ′*addr* ⇒ *mname* ⇒ ′*addr val list* ⇒ *ty* ⇒ *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*:
$P,h \vdash a{\bullet}M(vs) : U \longleftrightarrow$
$(\exists\, hT\ Ts\ Ts'\ D.\ typeof\text{-}addr\ h\ a = \lfloor hT \rfloor \wedge map\ typeof_h\ vs = map\ Some\ Ts \wedge P \vdash class\text{-}type\text{-}of\ hT$ $sees\ M{:}Ts' \rightarrow U{=}Native\ in\ D \wedge P \vdash Ts\ [\leq]\ Ts')$
⟨*proof*⟩

end

context *heap* **begin**

lemma *external-WT′-hext-mono*:
$[\![\ P,h \vdash a{\bullet}M(vs) : T;\ h \trianglelefteq h'\ ]\!] \Longrightarrow P,h' \vdash a{\bullet}M(vs) : T$
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

| *RedJoin*:
⟦ *typeof-addr h a* = ⌊*Class-type C*⌋; *P* ⊢ *C* ⪯* *Thread* ⟧
⟹ *P*,*t* ⊢ ⟨*a*•*join*([]), *h*⟩ −⦃*Join* (*addr2thread-id a*), *IsInterrupted t False*, *ThreadJoin* (*addr2thread-id a*)⦄→*ext* ⟨*RetVal Unit*, *h*⟩

| *RedJoinInterrupt*:
⟦ *typeof-addr h a* = ⌊*Class-type C*⌋; *P* ⊢ *C* ⪯* *Thread* ⟧
⟹ *P*,*t* ⊢ ⟨*a*•*join*([]), *h*⟩ −⦃*IsInterrupted t True*, *ClearInterrupt t*, *ObsInterrupted t*⦄→*ext* ⟨*RetEXC InterruptedException*, *h*⟩

— 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*:
$\llbracket$ *typeof-addr h a* = $\lfloor$*Class-type C*$\rfloor$; $P \vdash C \preceq^* Thread$ $\rrbracket$
$\Longrightarrow P,t \vdash \langle a{\cdot}interrupt([]),\ h\rangle$
$-\{\!|$ *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*:
$\llbracket$ *typeof-addr h a* = $\lfloor$*Class-type C*$\rfloor$; $P \vdash C \preceq^* Thread$ $\rrbracket$
$\Longrightarrow P,t \vdash \langle a{\cdot}interrupt([]),\ h\rangle$
$-\{\!|$ *ThreadExists* (*addr2thread-id a*) *False*$|\!\}\!\rightarrow ext$
$\langle$*RetVal Unit*, *h*$\rangle$

| *RedIsInterruptedTrue*:
$\llbracket$ *typeof-addr h a* = $\lfloor$*Class-type C*$\rfloor$; $P \vdash C \preceq^* Thread$ $\rrbracket$
$\Longrightarrow P,t \vdash \langle a{\cdot}isInterrupted([]),\ h\rangle$ $-\{\!|$ *IsInterrupted* (*addr2thread-id a*) *True*, *ObsInterrupted* (*addr2thread-id a*)$|\!\}\!\rightarrow ext$
$\langle$*RetVal* (*Bool True*), *h*$\rangle$

| *RedIsInterruptedFalse*:
$\llbracket$ *typeof-addr h a* = $\lfloor$*Class-type C*$\rfloor$; $P \vdash C \preceq^* Thread$ $\rrbracket$
$\Longrightarrow P,t \vdash \langle a{\cdot}isInterrupted([]),\ h\rangle$ $-\{\!|$*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* ⊢ ⟨*a*•*wait*([]), *h*⟩ −⦃*WokenUp*, *ClearInterrupt t*, *ObsInterrupted t*⦄→*ext* ⟨*RetEXC InterruptedException*, *h*⟩

— 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 ⟹
P,*t* ⊢ ⟨*a*•*wait*([]), *h*⟩ −⦃*Unlock*→*a*, *Lock*→*a*, *ReleaseAcquire*→*a*, *IsInterrupted t False*, *SyncUnlock a*⦄ →*ext*
⟨*RetVal Unit*, *h*⟩

— *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* ⊢ ⟨*a*•*notify*([]), *h*⟩ −⦃*Notify a*, *Unlock*→*a*, *Lock*→*a*⦄→*ext* ⟨*RetVal Unit*, *h*⟩

| *RedNotifyFail*:
P,*t* ⊢ ⟨*a*•*notify*([]), *h*⟩ −⦃*UnlockFail*→*a*⦄→*ext* ⟨*RetEXC IllegalMonitorState*, *h*⟩

| *RedNotifyAll*:
P,*t* ⊢ ⟨*a*•*notifyAll*([]), *h*⟩ −⦃*NotifyAll a*, *Unlock*→*a*, *Lock*→*a*⦄→*ext* ⟨*RetVal Unit*, *h*⟩

| *RedNotifyAllFail*:
P,*t* ⊢ ⟨*a*•*notifyAll*([]), *h*⟩ −⦃*UnlockFail*→*a*⦄→*ext* ⟨*RetEXC IllegalMonitorState*, *h*⟩

| *RedClone*:
heap-clone P h a h′ ⌊(*obs*, *a′*)⌋
⟹ *P*,*t* ⊢ ⟨*a*•*clone*([]), *h*⟩ −(*K$* [], [], [], [], [], *obs*)→*ext* ⟨*RetVal* (*Addr a′*), *h′*⟩

| *RedCloneFail*:
heap-clone P h a h′ None ⟹ *P*,*t* ⊢ ⟨*a*•*clone*([]), *h*⟩ −ε→*ext* ⟨*RetEXC OutOfMemory*, *h′*⟩

| *RedHashcode*:
P,*t* ⊢ ⟨*a*•*hashcode*([]), *h*⟩ −⦃⦄→*ext* ⟨*RetVal* (*Intg* (*word-of-int* (*hash-addr a*))), *h*⟩

| *RedPrint*:
P,*t* ⊢ ⟨*a*•*print*(*vs*), *h*⟩ −⦃*ExternalCall a print vs Unit*⦄→*ext* ⟨*RetVal Unit*, *h*⟩

| *RedCurrentThread*:
P,*t* ⊢ ⟨*a*•*currentThread*([]), *h*⟩ −⦃⦄→*ext* ⟨*RetVal* (*Addr* (*thread-id2addr t*)), *h*⟩

| *RedInterruptedTrue*:
P,*t* ⊢ ⟨*a*•*interrupted*([]), *h*⟩ −⦃*IsInterrupted t True*, *ClearInterrupt t*, *ObsInterrupted t*⦄→*ext* ⟨*RetVal* (*Bool True*), *h*⟩

| *RedInterruptedFalse*:
P,*t* ⊢ ⟨*a*•*interrupted*([]), *h*⟩ −⦃*IsInterrupted t False*⦄→*ext* ⟨*RetVal* (*Bool False*), *h*⟩

| *RedYield*:
P,*t* ⊢ ⟨*a*•*yield*([]), *h*⟩ −⦃*Yield*⦄→*ext* ⟨*RetVal Unit*, *h*⟩

3.11.3 Aggressive formulation for external cals

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
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*:

$P,t \vdash \langle a\bullet M(vs),\ h\rangle\ -ta\rightarrow ext\ \langle va,\ h'\rangle \Longrightarrow (ta,\ va,\ h') \in red\text{-}external\text{-}aggr\ P\ t\ a\ M\ vs\ h$
⟨*proof*⟩

end

context *heap* **begin**

lemma *hext-heap-copy-loc*:
$heap\text{-}copy\text{-}loc\ a\ a'\ al\ h\ obs\ h' \Longrightarrow h \trianglelefteq h'$
⟨*proof*⟩

lemma *hext-heap-copies*:
assumes $heap\text{-}copies\ a\ a'\ als\ h\ obs\ h'$
shows $h \trianglelefteq h'$
⟨*proof*⟩

lemma *hext-heap-clone*:
assumes $heap\text{-}clone\ P\ h\ a\ h'\ res$
shows $h \trianglelefteq h'$
⟨*proof*⟩

theorem *red-external-hext*:
assumes $P,t \vdash \langle a\bullet M(vs),\ h\rangle\ -ta\rightarrow ext\ \langle va,\ h'\rangle$
shows $hext\ h\ h'$
⟨*proof*⟩

lemma *red-external-preserves-tconf*:
$[\![\ P,t \vdash \langle a\bullet M(vs),\ h\rangle\ -ta\rightarrow ext\ \langle va,\ h'\rangle;\ P,h \vdash t'\ \surd t\ ]\!] \Longrightarrow P,h' \vdash t'\ \surd t$
⟨*proof*⟩

end

context *heap-conf* **begin**

lemma *typeof-addr-heap-clone*:
assumes $heap\text{-}clone\ P\ h\ a\ h'\ \lfloor(obs,\ a')\rfloor$
and $hconf\ h$
shows $typeof\text{-}addr\ h'\ a' = typeof\text{-}addr\ h\ a$
⟨*proof*⟩

end

context *heap-base* **begin**

lemma *red-ext-new-thread-heap*:
$[\![\ P,t \vdash \langle a\bullet M(vs),\ h\rangle\ -ta\rightarrow ext\ \langle va,\ h'\rangle;\ NewThread\ t'\ ex\ h'' \in set\ \{\!|ta|\!\}_t\ ]\!] \Longrightarrow h'' = h'$
⟨*proof*⟩

lemma *red-ext-aggr-new-thread-heap*:
$[\![\ (ta,\ va,\ h') \in red\text{-}external\text{-}aggr\ P\ t\ a\ M\ vs\ h;\ NewThread\ t'\ ex\ h'' \in set\ \{\!|ta|\!\}_t\ ]\!] \Longrightarrow h'' = h'$
⟨*proof*⟩

end

context *addr-conv* **begin**

lemma *red-external-new-thread-exists-thread-object*:
$\llbracket$ $P,t \vdash \langle a{\cdot}M(vs),\ h\rangle\ -ta\rightarrow ext\ \langle va,\ h'\rangle$; $NewThread\ t'\ x\ h'' \in set\ \{\!|ta|\!\}_t$ $\rrbracket$
$\Longrightarrow \exists\, C.\ typeof\text{-}addr\ h'\ (thread\text{-}id2addr\ t') = \lfloor Class\text{-}type\ C\rfloor \wedge P \vdash C \preceq^* Thread$
⟨proof⟩

lemma *red-external-aggr-new-thread-exists-thread-object*:
$\llbracket$ $(ta,\ va,\ h') \in red\text{-}external\text{-}aggr\ P\ t\ a\ M\ vs\ h$; $typeof\text{-}addr\ h\ a \neq None$;
$NewThread\ t'\ x\ h'' \in set\ \{\!|ta|\!\}_t$ $\rrbracket$
$\Longrightarrow \exists\, C.\ typeof\text{-}addr\ h'\ (thread\text{-}id2addr\ t') = \lfloor Class\text{-}type\ C\rfloor \wedge P \vdash C \preceq^* Thread$
⟨proof⟩

end

context *heap* **begin**

lemma *red-external-aggr-hext*:
$\llbracket$ $(ta,\ va,\ h') \in red\text{-}external\text{-}aggr\ P\ t\ a\ M\ vs\ h$; $is\text{-}native\ P\ (the\ (typeof\text{-}addr\ h\ a))\ M$ $\rrbracket \Longrightarrow h \trianglelefteq h'$
⟨proof⟩

lemma *red-external-aggr-preserves-tconf*:
$\llbracket$ $(ta,\ va,\ h') \in red\text{-}external\text{-}aggr\ P\ t\ a\ M\ vs\ h$; $is\text{-}native\ P\ (the\ (typeof\text{-}addr\ h\ a))\ M$; $P,h \vdash t' \surd t$ $\rrbracket$
$\Longrightarrow P,h' \vdash t' \surd t$
⟨proof⟩

end

context *heap-base* **begin**

lemma *red-external-Wakeup-no-Join-no-Lock-no-Interrupt*:
$\llbracket$ $P,t \vdash \langle a{\cdot}M(vs),\ h\rangle\ -ta\rightarrow ext\ \langle va,\ h'\rangle$; $Notified \in set\ \{\!|ta|\!\}_w \vee WokenUp \in set\ \{\!|ta|\!\}_w$ $\rrbracket \Longrightarrow$
$collect\text{-}locks\ \{\!|ta|\!\}_l = \{\} \wedge collect\text{-}cond\text{-}actions\ \{\!|ta|\!\}_c = \{\} \wedge collect\text{-}interrupts\ \{\!|ta|\!\}_i = \{\}$
⟨proof⟩

lemma *red-external-aggr-Wakeup-no-Join*:
$\llbracket$ $(ta,\ va,\ h') \in red\text{-}external\text{-}aggr\ P\ t\ a\ M\ vs\ h$;
$Notified \in set\ \{\!|ta|\!\}_w \vee WokenUp \in set\ \{\!|ta|\!\}_w$ $\rrbracket$
$\Longrightarrow collect\text{-}locks\ \{\!|ta|\!\}_l = \{\} \wedge collect\text{-}cond\text{-}actions\ \{\!|ta|\!\}_c = \{\} \wedge collect\text{-}interrupts\ \{\!|ta|\!\}_i = \{\}$
⟨proof⟩

lemma *red-external-Suspend-StaySame*:
$\llbracket$ $P,t \vdash \langle a{\cdot}M(vs),\ h\rangle\ -ta\rightarrow ext\ \langle va,\ h'\rangle$; $Suspend\ w \in set\ \{\!|ta|\!\}_w$ $\rrbracket \Longrightarrow va = RetStaySame$
⟨proof⟩

lemma *red-external-aggr-Suspend-StaySame*:
$\llbracket$ $(ta,\ va,\ h') \in red\text{-}external\text{-}aggr\ P\ t\ a\ M\ vs\ h$; $Suspend\ w \in set\ \{\!|ta|\!\}_w$ $\rrbracket \Longrightarrow va = RetStaySame$
⟨proof⟩

lemma *red-external-Suspend-waitD*:
$\llbracket$ $P,t \vdash \langle a{\cdot}M(vs),\ h\rangle\ -ta\rightarrow ext\ \langle va,\ h'\rangle$; $Suspend\ w \in set\ \{\!|ta|\!\}_w$ $\rrbracket \Longrightarrow M = wait$
⟨proof⟩

lemma *red-external-aggr-Suspend-waitD*:

$\llbracket$ *(ta, va, h′)* ∈ *red-external-aggr P t a M vs h*; *Suspend w* ∈ *set* $\{\!|ta|\!\}_w$ $\rrbracket$ ⟹ *M* = *wait*
⟨*proof*⟩

lemma *red-external-new-thread-sub-thread*:
$\llbracket$ *P*,*t* ⊢ ⟨*a*•*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h′*⟩; *NewThread t′ (C, M′, a′) h″* ∈ *set* $\{\!|ta|\!\}_t$ $\rrbracket$
⟹ *typeof-addr h′ a′* = ⌊*Class-type C*⌋ ∧ *P* ⊢ *C* ⪯* *Thread* ∧ *M′* = *run*
⟨*proof*⟩

lemma *red-external-aggr-new-thread-sub-thread*:
$\llbracket$ *(ta, va, h′)* ∈ *red-external-aggr P t a M vs h*; *typeof-addr h a* ≠ *None*;
NewThread t′ (C, M′, a′) h″ ∈ *set* $\{\!|ta|\!\}_t$ $\rrbracket$
⟹ *typeof-addr h′ a′* = ⌊*Class-type C*⌋ ∧ *P* ⊢ *C* ⪯* *Thread* ∧ *M′* = *run*
⟨*proof*⟩

lemma *heap-copy-loc-length*:
assumes *heap-copy-loc a a′ al h obs h′*
shows *length obs* = *2*
⟨*proof*⟩

lemma *heap-copies-length*:
assumes *heap-copies a a′ als h obs h′*
shows *length obs* = *2* ∗ *length als*
⟨*proof*⟩

end

3.11.4 τ-moves

inductive *τexternal-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*:
$\llbracket$ *P*,*t* ⊢ ⟨*a*•*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h′*⟩; *τexternal′ P h a M* $\rrbracket$ ⟹ *h′* = *h*
⟨*proof*⟩

lemma *τexternal′-red-external-aggr-heap-unchanged*:
$\llbracket$ *(ta, va, h′)* ∈ *red-external-aggr P t a M vs h*; *τexternal′ P h a M* $\rrbracket$ ⟹ *h′* = *h*
⟨*proof*⟩

lemma *τexternal′-red-external-TA-empty*:
$\llbracket$ *P*,*t* ⊢ ⟨*a*•*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h′*⟩; *τexternal′ P h a M* $\rrbracket$ ⟹ *ta* = ε
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *bool*)
[*inductify*, *skip-proof*]
is-native
⟨*proof*⟩

declare *heap-base.heap-copy-loc.intros*[*code-pred-intro*]

code-pred
(*modes*: (*i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*) ⇒ (*i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*) ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *o* ⇒ *bool*)
heap-base.heap-copy-loc
⟨*proof*⟩

declare *heap-base.heap-copies.intros* [*code-pred-intro*]

code-pred
(*modes*: (*i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*) => (*i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*) ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *o* ⇒ *bool*)
heap-base.heap-copies
⟨*proof*⟩

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*⟩

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*⟩
⟨*proof*⟩

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*⟩

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*) =
(∀ *D' Ts' T' m'*. *P* ⊢ *D sees M*:*Ts'* → *T'* = *m' in D'* ⟶ *P* ⊢ *Ts'* [≤] *Ts* ∧ *P* ⊢ *T* ≤ *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* $\preceq^*$ *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*)
⟨*proof*⟩

3.12.1 Well-formedness lemmas

lemma *wf-prog-wf-syscls*: *wf-prog wf-md P* ⟹ *wf-syscls P*
⟨*proof*⟩

lemma *class-wf*:
⟦*class P C* = *Some c*; *wf-prog wf-md P*⟧ ⟹ *wf-cdecl wf-md P* (*C*,*c*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma [*simp*]:
assumes *wf-prog wf-md P*
shows *is-class-Object*: *is-class P Object*
and *is-class-Thread*: *is-class P Thread*
⟨*proof*⟩

lemma *xcpt-subcls-Throwable*:
$[\![$ $C \in$ *sys-xcpts*; *wf-prog wf-md P* $]\!] \Longrightarrow P \vdash C \preceq^* $ *Throwable*
⟨*proof*⟩

lemma *is-class-Throwable*:
wf-prog wf-md P $\Longrightarrow$ *is-class P Throwable*
⟨*proof*⟩

lemma *is-class-sub-Throwable*:
$[\![$ *wf-prog wf-md P*; $P \vdash C \preceq^*$ *Throwable* $]\!] \Longrightarrow$ *is-class P C*
⟨*proof*⟩

lemma *is-class-xcpt*:
$[\![$ $C \in$ *sys-xcpts*; *wf-prog wf-md P* $]\!] \Longrightarrow$ *is-class P C*
⟨*proof*⟩

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*⟩

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*⌋ $\wedge$ $P \vdash C \preceq^*$ *Throwable*
⟨*proof*⟩

end

lemma (**in** *heap-conf*) *hconf-start-heap*:
wf-prog wf-md P $\Longrightarrow$ *hconf start-heap*
⟨*proof*⟩

lemma *subcls1-wfD*:
$[\![$ $P \vdash C \prec^1 D$; *wf-prog wf-md P* $]\!] \Longrightarrow D \neq C \wedge \neg$ (*subcls1 P*)$^{++}$ *D C*
⟨*proof*⟩

lemma *wf-cdecl-supD*:
$[\![$*wf-cdecl wf-md P* (*C*,*D*,*r*); $C \neq$ *Object*$]\!] \Longrightarrow$ *is-class P D*⟨*proof*⟩

lemma *subcls-asym*:
$[\![$ *wf-prog wf-md P*; (*subcls1 P*)$^{++}$ *C D* $]\!] \Longrightarrow \neg$ (*subcls1 P*)$^{++}$ *D C*⟨*proof*⟩

lemma *subcls-irrefl*:
$[\![$ *wf-prog wf-md P*; (*subcls1 P*)$^{++}$ *C D*$]\!] \Longrightarrow C \neq D$⟨*proof*⟩
lemma *acyclicP-def*:
acyclicP r $\longleftrightarrow$ ($\forall$ *x*. $\neg$ *r*^++ *x x*)
⟨*proof*⟩

lemma *acyclic-subcls1*:
wf-prog wf-md P $\Longrightarrow$ *acyclicP* (*subcls1 P*)
⟨*proof*⟩

lemma *finite-conversep*: *finite* $\{(x, y).\ r^{-1-1}\ x\ y\}$ = *finite* $\{(x, y).\ r\ x\ y\}$
⟨*proof*⟩

lemma *acyclicP-wf-subcls1*:
acyclicP (*subcls1 P*) $\Longrightarrow$ *wfP* $((subcls1\ P)^{-1-1})$
⟨*proof*⟩

lemma *wf-subcls1*:
wf-prog wf-md P $\Longrightarrow$ *wfP* $((subcls1\ P)^{-1-1})$
⟨*proof*⟩

lemma *single-valued-subcls1*:
wf-prog wf-md G $\Longrightarrow$ *single-valuedp* (*subcls1 G*)⟨*proof*⟩

lemma *subcls-induct*:
$[\![$ *wf-prog wf-md P*; $\bigwedge C.\ \forall D.\ (subcls1\ P)^{++}\ C\ D \longrightarrow Q\ D \Longrightarrow Q\ C$ $]\!] \Longrightarrow Q\ C$⟨*proof*⟩

lemma *subcls1-induct-aux*:
$[\![$ *is-class P C*; *wf-prog wf-md P*; *Q Object*;
$\bigwedge C\ D\ fs\ ms.$
$[\![$ $C \neq Object$; *is-class P C*; *class P C* = *Some* (*D*,*fs*,*ms*) $\wedge$
wf-cdecl wf-md P (*C*,*D*,*fs*,*ms*) $\wedge$ $P \vdash C \prec^1 D$ $\wedge$ *is-class P D* $\wedge$ *Q D*$]\!] \Longrightarrow Q\ C$ $]\!]$
$\Longrightarrow Q\ C$⟨*proof*⟩
lemma *subcls1-induct* [*consumes 2*, *case-names Object Subcls*]:
$[\![$ *wf-prog wf-md P*; *is-class P C*; *Q Object*;
$\bigwedge C\ D.$ $[\![C \neq Object$; $P \vdash C \prec^1 D$; *is-class P D*; $Q\ D]\!] \Longrightarrow Q\ C$ $]\!]$
$\Longrightarrow Q\ C$⟨*proof*⟩

lemma *subcls-C-Object*:
$[\![$ *is-class P C*; *wf-prog wf-md P* $]\!] \Longrightarrow P \vdash C \preceq^* Object$⟨*proof*⟩
lemma *converse-subcls-is-class*:
assumes *wf*: *wf-prog wf-md P*
shows $[\![$ $P \vdash C \preceq^* D$; *is-class P C* $]\!] \Longrightarrow$ *is-class P D*
⟨*proof*⟩

lemma *is-class-is-subcls*:
wf-prog m P $\Longrightarrow$ *is-class P C* = $P \vdash C \preceq^* Object$⟨*proof*⟩
lemma *subcls-antisym*:
$[\![$*wf-prog m P*; $P \vdash C \preceq^* D$; $P \vdash D \preceq^* C]\!] \Longrightarrow C = D$
⟨*proof*⟩

lemma *is-type-pTs*:
$[\![$ *wf-prog wf-md P*; *class P C* = $\lfloor(S,fs,ms)\rfloor$; $(M,Ts,T,m) \in$ *set ms* $]\!] \Longrightarrow$ *set Ts* $\subseteq$ *types P*
⟨*proof*⟩

lemma *widen-asym-1*:
assumes *wfP*: *wf-prog wf-md P*
shows $P \vdash C \leq D \Longrightarrow C = D \vee \neg\ (P \vdash D \leq C)$
⟨*proof*⟩

lemma *widen-asym*: $[\![$ *wf-prog wf-md P*; $P \vdash C \leq D$; $C \neq D$ $]\!] \Longrightarrow \neg (P \vdash D \leq C)$
⟨*proof*⟩

lemma *widen-antisym*:
$[\![$ *wf-prog m P*; $P \vdash T \leq U$; $P \vdash U \leq T$ $]\!] \Longrightarrow T = U$
⟨*proof*⟩

lemma *widen-C-Object*: $[\![$ *wf-prog wf-md P*; *is-class P C* $]\!] \Longrightarrow P \vdash$ *Class C* $\leq$ *Class Object*
⟨*proof*⟩

lemma *is-refType-widen-Object*:
assumes *wfP*: *wf-prog wfmc P*
shows $[\![$ *is-type P A*; *is-refT A* $]\!] \Longrightarrow P \vdash A \leq$ *Class Object*
⟨*proof*⟩

lemma *is-lub-unique*:
assumes *wf*: *wf-prog wf-md P*
shows $[\![$ $P \vdash lub(U,\ V) = T$; $P \vdash lub(U,\ V) = T'$ $]\!] \Longrightarrow T = T'$
⟨*proof*⟩

3.12.2 Well-formedness and method lookup

lemma *sees-wf-mdecl*:
$[\![$ *wf-prog wf-md P*; $P \vdash C$ *sees* $M{:}Ts{\rightarrow}T = m$ *in* D $]\!] \Longrightarrow$ *wf-mdecl wf-md P D* (M,Ts,T,m)⟨*proof*⟩

lemma *sees-method-mono* [*rule-format* (*no-asm*)]:
$[\![$ $P \vdash C' \preceq^* C$; *wf-prog wf-md P* $]\!] \Longrightarrow$
$\forall D\ Ts\ T\ m.\ P \vdash C$ *sees* $M{:}Ts{\rightarrow}T = m$ *in* $D \longrightarrow$
$(\exists D'\ Ts'\ T'\ m'.\ P \vdash C'$ *sees* $M{:}Ts'{\rightarrow}T' = m'$ *in* $D' \wedge P \vdash Ts\ [\leq]\ Ts' \wedge P \vdash T' \leq T)$
⟨*proof*⟩

lemma *sees-method-mono2*:
$[\![$ $P \vdash C' \preceq^* C$; *wf-prog wf-md P*;
$P \vdash C$ *sees* $M{:}Ts{\rightarrow}T = m$ *in* D; $P \vdash C'$ *sees* $M{:}Ts'{\rightarrow}T' = m'$ *in* D' $]\!]$
$\Longrightarrow P \vdash Ts\ [\leq]\ Ts' \wedge P \vdash T' \leq T$⟨*proof*⟩

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)
$\Longrightarrow \exists Mm.\ P \vdash C$ *sees-methods Mm* $\wedge (\forall (M,Ts,T,m) \in$ *set ms*. *Mm M* $=$ *Some*$((Ts,T,m),C))$
⟨*proof*⟩

lemma *mdecl-visible*:
assumes *wf*: *wf-prog wf-md P* **and** *C*: *class P C* $= \lfloor(S,fs,ms)\rfloor$ **and** *m*: $(M,Ts,T,m) \in$ *set ms*
shows $P \vdash C$ *sees* $M{:}Ts{\rightarrow}T = m$ *in* C
⟨*proof*⟩

lemma *sees-wf-native*:
$[\![$ *wf-prog wf-md P*; $P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}Native$ *in* D $]\!] \Longrightarrow D{\bullet}M(Ts) :: T$
⟨*proof*⟩

lemma *Call-lemma*:

$\llbracket$ *P* ⊢ *C sees M*:*Ts*→*T* = *m in D*; *P* ⊢ *C′* $\preceq^*$ *C*; *wf-prog wf-md P* $\rrbracket$
$\Longrightarrow$ ∃ *D′ Ts′ T′ m′*.
P ⊢ *C′ sees M*:*Ts′*→*T′* = *m′ in D′* ∧ *P* ⊢ *Ts* [≤] *Ts′* ∧ *P* ⊢ *T′* ≤ *T* ∧ *P* ⊢ *C′* $\preceq^*$ *D′*
∧ *is-type P T′* ∧ (∀ *T*∈*set Ts′*. *is-type P T*) ∧ (*m′* ≠ *Native* ⟶ *wf-md P D′* (*M*,*Ts′*,*T′*,*the m′*))
⟨*proof*⟩

lemma *sub-Thread-sees-run*:
assumes *wf*: *wf-prog wf-md P*
and *PCThread*: *P* ⊢ *C* $\preceq^*$ *Thread*
shows ∃ *D mthd*. *P* ⊢ *C sees run*: []→*Void* = ⌊*mthd*⌋ *in D*
⟨*proof*⟩

lemma *wf-prog-lift*:
assumes *wf*: *wf-prog* (λ*P C bd*. *A P C bd*) *P*
and *rule*:
⋀*wf-md C M Ts C T m*.
$\llbracket$ *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*) $\rrbracket$
$\Longrightarrow$ *B P C* (*M*,*Ts*,*T*,*m*)
shows *wf-prog* (λ*P C bd*. *B P C bd*) *P*
⟨*proof*⟩

3.12.3 Well-formedness and field lookup

lemma *wf-Fields-Ex*:
$\llbracket$ *wf-prog wf-md P*; *is-class P C* $\rrbracket$ $\Longrightarrow$ ∃ *FDTs*. *P* ⊢ *C has-fields FDTs*⟨*proof*⟩

lemma *has-fields-types*:
$\llbracket$ *P* ⊢ *C has-fields FDTs*; (*FD*, *T*, *fm*) ∈ *set FDTs*; *wf-prog wf-md P* $\rrbracket$ $\Longrightarrow$ *is-type P T*⟨*proof*⟩

lemma *sees-field-is-type*:
$\llbracket$ *P* ⊢ *C sees F*:*T* (*fm*) *in D*; *wf-prog wf-md P* $\rrbracket$ $\Longrightarrow$ *is-type P T*
⟨*proof*⟩

lemma *wf-has-field-mono2*:
assumes *wf*: *wf-prog wf-md P*
and *has*: *P* ⊢ *C has F*:*T* (*fm*) *in E*
shows $\llbracket$ *P* ⊢ *C* $\preceq^*$ *D*; *P* ⊢ *D* $\preceq^*$ *E* $\rrbracket$ $\Longrightarrow$ *P* ⊢ *D has F*:*T* (*fm*) *in E*
⟨*proof*⟩

lemma *wf-has-field-idemp*:
$\llbracket$ *wf-prog wf-md P*; *P* ⊢ *C has F*:*T* (*fm*) *in D* $\rrbracket$ $\Longrightarrow$ *P* ⊢ *D has F*:*T* (*fm*) *in D*
⟨*proof*⟩

lemma *map-of-remap-conv*:
$\llbracket$ *distinct-fst fs*; *map-of* (*map* (λ(*F*, *y*). ((*F*, *D*), *y*)) *fs*) (*F*, *D*) = ⌊*T*⌋ $\rrbracket$
$\Longrightarrow$ *map-of* (*map* (λ((*F*, *D*), *T*). (*F*, *D*, *T*)) (*map* (λ(*F*, *y*). ((*F*, *D*), *y*)) *fs*)) *F* = ⌊(*D*, *T*)⌋
⟨*proof*⟩

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*⟩

lemma *has-fields-distinct*:
assumes *wf*: *wf-prog wf-md P*
and $P \vdash C$ *has-fields FDTs*
shows *distinct* (*map fst FDTs*)
⟨*proof*⟩

3.12.4 Code generation

code-pred
(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow bool$)
[*inductify*]
wf-overriding
⟨*proof*⟩

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* ⇒ *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* ⇒ *'m prog* ⇒ *'m cdecl* ⇒ *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 $P \longleftrightarrow (\forall (C, D, fs, ml) \in set\ (classes\ P).\ C \neq Object \longrightarrow \neg\ subcls'\ P\ D\ C)$
⟨*proof*⟩

lemma *wf-cdecl'-code* [*code*]:
wf-cdecl' wf-md P = $(\lambda(C,(D,fs,ms)).$
$(\forall f{\in}set\ fs.\ wf\text{-}fdecl\ P\ f) \wedge\ distinct\text{-}fst\ fs \wedge$
$(\forall m{\in}set\ ms.\ wf\text{-}mdecl\ wf\text{-}md\ P\ C\ m) \wedge$
$distinct\text{-}fst\ ms \wedge$
$(C \neq Object \longrightarrow is\text{-}class\ P\ D \wedge (\forall m{\in}set\ ms.\ wf\text{-}overriding\ P\ D\ m)) \wedge$
$(C = Thread \longrightarrow ((run, [], Void) \in set\ (map\ (\lambda(M, Ts, T, b).\ (M, Ts, T))\ ms))))$
⟨*proof*⟩

declare *set-append* [*symmetric*, *code-unfold*]

lemma *wf-prog-code* [*code*]:
wf-prog wf-md P $\longleftrightarrow$
acyclic-class-hierarchy P ∧
wf-syscls P ∧ *distinct-fst* (*classes P*) ∧
$(\forall c \in set\ (classes\ P).\ wf\text{-}cdecl'\ wf\text{-}md\ P\ c)$
⟨*proof*⟩

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* ⊆ *types P*
⟨*proof*⟩

context *heap-base* **begin**

lemma *WT-red-external-aggr-imp-red-external*:
  ⟦ *wf-prog wf-md P*; (*ta*, *va*, *h′*) ∈ *red-external-aggr P t a M vs h*; *P*,*h* ⊢ *a*•*M*(*vs*) : *U*; *P*,*h* ⊢ *t* √*t* ⟧
  ⟹ *P*,*t* ⊢ ⟨*a*•*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h′*⟩
⟨*proof*⟩

lemma *WT-red-external-list-conv*:
  ⟦ *wf-prog wf-md P*; *P*,*h* ⊢ *a*•*M*(*vs*) : *U*; *P*,*h* ⊢ *t* √*t* ⟧
  ⟹ *P*,*t* ⊢ ⟨*a*•*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h′*⟩ ⟷ (*ta*, *va*, *h′*) ∈ *red-external-aggr P t a M vs h*
⟨*proof*⟩

lemma *red-external-new-thread-sees*:
  ⟦ *wf-prog wf-md P*; *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*⌋ ∧ (∃ *T meth D*. *P* ⊢ *C sees M′*:[]→*T* = ⌊*meth*⌋ *in D*)
⟨*proof*⟩

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* ⊢ *a*@*al* : *T P*,*h* ⊢ *a′*@*al* : *T*
  shows *hconf h′*
⟨*proof*⟩

lemma *hconf-heap-copies-mono*:
  assumes *heap-copies a a′ als h obs h′*
  and *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 *hconf h′*
⟨*proof*⟩

lemma *hconf-heap-clone-mono*:
  assumes *heap-clone P h a h′ res*
  and *hconf h*

shows *hconf h′*
⟨*proof*⟩

theorem *external-call-hconf*:
assumes *major*: $P,t \vdash \langle a{\bullet}M(vs),\ h\rangle\ {-ta\rightarrow ext}\ \langle va,\ h'\rangle$
and *minor*: $P,h \vdash a{\bullet}M(vs) : U$ *hconf h*
shows *hconf h′*
⟨*proof*⟩

end

context *heap-base* **begin**

primrec *conf-extRet* :: $'m\ prog \Rightarrow\ 'heap \Rightarrow\ 'addr\ extCallRet \Rightarrow ty \Rightarrow bool$ **where**
conf-extRet P h (*RetVal v*) $T = (P,h \vdash v :\leq T)$
| *conf-extRet P h* (*RetExc a*) $T = (P,h \vdash$ *Addr a* $:\leq$ *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 $[\![P,t \vdash \langle a{\bullet}M(vs),\ h\rangle\ {-ta\rightarrow ext}\ \langle va,\ h'\rangle;\ P,h \vdash a{\bullet}M(vs) : U;$ *hconf h*; *preallocated h*; $P,h \vdash t$ $\surd t\ ]\!]$
$\Longrightarrow$ *conf-extRet P h′ va U*
⟨*proof*⟩

end

3.13.2 Progress theorems for external calls

context *heap-progress* **begin**

lemma *heap-copy-loc-progress*:
assumes *hconf*: *hconf h*
and *alconfa*: $P,h \vdash a@al : T$
and *alconfa′*: $P,h \vdash a'@al : T$
shows $\exists v\ h'.$ *heap-copy-loc a a′ al h* ([*ReadMem a al v*, *WriteMem a′ al v*]) $h' \wedge P,h \vdash v :\leq T \wedge$ *hconf h′*
⟨*proof*⟩

lemma *heap-copies-progress*:
assumes *hconf h*
and *list-all2* $(\lambda al\ T.\ P,h \vdash a@al : T)$ *als Ts*
and *list-all2* $(\lambda al\ T.\ P,h \vdash a'@al : T)$ *als Ts*
shows $\exists vs\ h'.$ *heap-copies a a′ als h* (*concat* (*map* $(\lambda(al,\ v).$ [*ReadMem a al v*, *WriteMem a′ al v*]) (*zip als vs*))) $h' \wedge$ *hconf h′*
⟨*proof*⟩

lemma *heap-clone-progress*:
assumes *wf*: *wf-prog wf-md P*
and *typea*: *typeof-addr h a* = $\lfloor hT \rfloor$

and *hconf*: *hconf h*
shows $\exists\, h'$ *res*. *heap-clone P h a h' res*
⟨*proof*⟩

theorem *external-call-progress*:
assumes *wf*: *wf-prog wf-md P*
and *wt*: $P,h \vdash a{\cdot}M(vs) : U$
and *hconf*: *hconf h*
shows $\exists\, ta\ va\ h'.\ P,t \vdash \langle a{\cdot}M(vs),\ h\rangle\ -ta\rightarrow ext\ \langle va,\ h'\rangle$
⟨*proof*⟩

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*: $P,t \vdash \langle a{\cdot}M(vs),h\rangle\ -ta\rightarrow ext\ \langle va,h'\rangle$
and *hext*: $h'' \trianglelefteq h$
and *wt*: $P,h'' \vdash a{\cdot}M(vs) : U$
and *tconf*: $P,h'' \vdash t\ \surd t$
and *hconf*: *hconf* h''
shows $\exists\, ta'\ va'\ h'''.\ P,t \vdash \langle a{\cdot}M(vs),h''\rangle\ -ta'\rightarrow ext\ \langle va',\ h'''\rangle\ \wedge$
$collect\text{-}locks\ \{\!|ta|\!\}_l = collect\text{-}locks\ \{\!|ta'|\!\}_l\ \wedge$
$collect\text{-}cond\text{-}actions\ \{\!|ta|\!\}_c = collect\text{-}cond\text{-}actions\ \{\!|ta'|\!\}_c\ \wedge$
$collect\text{-}interrupts\ \{\!|ta|\!\}_i = collect\text{-}interrupts\ \{\!|ta'|\!\}_i$
⟨*proof*⟩

lemma *red-external-wf-red*:
assumes *wf*: *wf-prog wf-md P*
and *red*: $P,t \vdash \langle a{\cdot}M(vs),\ h\rangle\ -ta\rightarrow ext\ \langle va,\ h'\rangle$
and *tconf*: $P,h \vdash t\ \surd t$
and *hconf*: *hconf h*
and *wst*: $wset\ s\ t = None \vee (M = wait \wedge (\exists\, w.\ wset\ s\ t = \lfloor PostWS\ w\rfloor))$
obtains $ta'\ va'\ h''$
where $P,t \vdash \langle a{\cdot}M(vs),\ h\rangle\ -ta'\rightarrow ext\ \langle va',\ h''\rangle$
and *final-thread.actions-ok final s t ta'* ∨ *final-thread.actions-ok' s t ta'* ∧ *final-thread.actions-subset ta' ta*
⟨*proof*⟩

end

context *heap-base* **begin**

lemma *red-external-ta-satisfiable*:
fixes *final*
assumes $P,t \vdash \langle a{\cdot}M(vs),\ h\rangle\ -ta\rightarrow ext\ \langle va,\ h'\rangle$
shows ∃ *s*. *final-thread.actions-ok final s t ta*
⟨*proof*⟩

lemma *red-external-aggr-ta-satisfiable*:
fixes *final*

assumes $(ta, va, h') \in$ *red-external-aggr P t a M vs h*
shows $\exists$ *s. final-thread.actions-ok final s t ta*
⟨*proof*⟩

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 $ops = ops' \wedge h' = h''$
⟨*proof*⟩

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 $ops = ops' \wedge h' = h''$
⟨*proof*⟩

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 $h' = h'' \wedge obs = obs'$
⟨*proof*⟩

lemma *red-external-deterministic*:
fixes *final*
assumes *det*: *deterministic-heap-ops*
and *red*: $P,t \vdash \langle a{\bullet}M(vs),\ (shr\ s)\rangle\ {-ta{\rightarrow}ext}\ \langle va,\ h'\rangle\ P,t \vdash \langle a{\bullet}M(vs),\ (shr\ s)\rangle\ {-ta'{\rightarrow}ext}\ \langle va',\ h''\rangle$
and *aok*: *final-thread.actions-ok final s t ta final-thread.actions-ok final s t ta′*
shows $ta = ta' \wedge va = va' \wedge h' = h''$
⟨*proof*⟩

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* $(\lambda t\ x\ m.\ P,m \vdash t\ \sqrt{t})$

lemma *thread-confI*:
$(\bigwedge t\ xln.\ ts\ t = \lfloor xln \rfloor \Longrightarrow P,h \vdash t\ \surd t) \Longrightarrow$ *thread-conf P ts h*
⟨*proof*⟩

lemma *thread-confD*:
assumes *thread-conf P ts h ts t* = ⌊*xln*⌋
shows $P,h \vdash t\ \surd t$
⟨*proof*⟩

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*⟩

end

context *heap* **begin**

lemma *thread-conf-hext*:
⟦ *thread-conf P ts h*; *h* ⊴ *h'* ⟧ ⟹ *thread-conf P ts h'*
⟨*proof*⟩

lemma *thread-conf-start-state*:
⟦ *start-heap-ok*; *wf-syscls P* ⟧ ⟹ *thread-conf P* (*thr* (*start-state f P C M vs*)) (*shr* (*start-state f P C M vs*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

end

notepad begin
⟨*proof*⟩
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*]:
  $(\mathit{binop\text{-}LessThan}\ v1\ v2 = \mathit{Some}\ va) \longleftrightarrow$
   $(\exists\, i1\ i2.\ v1 = \mathit{Intg}\ i1 \wedge v2 = \mathit{Intg}\ i2 \wedge va = \mathit{Inl}\ (\mathit{Bool}\ (i1 <s\ i2)))$
⟨*proof*⟩

lemma [*simp*]:
  $(\mathit{binop\text{-}LessOrEqual}\ v1\ v2 = \mathit{Some}\ va) \longleftrightarrow$
   $(\exists\, i1\ i2.\ v1 = \mathit{Intg}\ i1 \wedge v2 = \mathit{Intg}\ i2 \wedge va = \mathit{Inl}\ (\mathit{Bool}\ (i1 <=s\ i2)))$
⟨*proof*⟩

lemma [*simp*]:
  $(\mathit{binop\text{-}GreaterThan}\ v1\ v2 = \mathit{Some}\ va) \longleftrightarrow$
   $(\exists\, i1\ i2.\ v1 = \mathit{Intg}\ i1 \wedge v2 = \mathit{Intg}\ i2 \wedge va = \mathit{Inl}\ (\mathit{Bool}\ (i2 <s\ i1)))$
⟨*proof*⟩

lemma [*simp*]:
  $(\mathit{binop\text{-}GreaterOrEqual}\ v1\ v2 = \mathit{Some}\ va) \longleftrightarrow$
   $(\exists\, i1\ i2.\ v1 = \mathit{Intg}\ i1 \wedge v2 = \mathit{Intg}\ i2 \wedge va = \mathit{Inl}\ (\mathit{Bool}\ (i2 <=s\ i1)))$
⟨*proof*⟩

lemma [*simp*]:
  $(\mathit{binop\text{-}Add}\ v_1\ v_2\ = \mathit{Some}\ va) \longleftrightarrow$
   $(\exists\, i_1\ i_2.\ v_1 = \mathit{Intg}\ i_1 \wedge v_2 = \mathit{Intg}\ i_2 \wedge va = \mathit{Inl}\ (\mathit{Intg}\ (i_1 + i_2)))$
⟨*proof*⟩

lemma [*simp*]:
  $(\mathit{binop\text{-}Subtract}\ v1\ v2 = \mathit{Some}\ va) \longleftrightarrow$
   $(\exists\, i1\ i2.\ v1 = \mathit{Intg}\ i1 \wedge v2 = \mathit{Intg}\ i2 \wedge va = \mathit{Inl}\ (\mathit{Intg}\ (i1 - i2)))$
⟨*proof*⟩

lemma [*simp*]:
  $(\mathit{binop\text{-}Mult}\ v1\ v2 = \mathit{Some}\ va) \longleftrightarrow$
   $(\exists\, i1\ i2.\ v1 = \mathit{Intg}\ i1 \wedge v2 = \mathit{Intg}\ i2 \wedge va = \mathit{Inl}\ (\mathit{Intg}\ (i1 * i2)))$
⟨*proof*⟩

lemma [*simp*]:
  $(\mathit{binop\text{-}BinAnd}\ v1\ v2 = \mathit{Some}\ va) \longleftrightarrow$
   $(\exists\, b1\ b2.\ v1 = \mathit{Bool}\ b1 \wedge v2 = \mathit{Bool}\ b2 \wedge va = \mathit{Inl}\ (\mathit{Bool}\ (b1 \wedge b2))) \vee$
   $(\exists\, i1\ i2.\ v1 = \mathit{Intg}\ i1 \wedge v2 = \mathit{Intg}\ i2 \wedge va = \mathit{Inl}\ (\mathit{Intg}\ (i1\ \mathit{AND}\ i2)))$
⟨*proof*⟩

lemma [*simp*]:
  $(\mathit{binop\text{-}BinOr}\ v1\ v2 = \mathit{Some}\ va) \longleftrightarrow$
   $(\exists\, b1\ b2.\ v1 = \mathit{Bool}\ b1 \wedge v2 = \mathit{Bool}\ b2 \wedge va = \mathit{Inl}\ (\mathit{Bool}\ (b1 \vee b2))) \vee$
   $(\exists\, i1\ i2.\ v1 = \mathit{Intg}\ i1 \wedge v2 = \mathit{Intg}\ i2 \wedge va = \mathit{Inl}\ (\mathit{Intg}\ (i1\ \mathit{OR}\ i2)))$
⟨*proof*⟩

lemma [*simp*]:
  $(\mathit{binop\text{-}BinXor}\ v1\ v2 = \mathit{Some}\ va) \longleftrightarrow$
   $(\exists\, b1\ b2.\ v1 = \mathit{Bool}\ b1 \wedge v2 = \mathit{Bool}\ b2 \wedge va = \mathit{Inl}\ (\mathit{Bool}\ (b1 \neq b2))) \vee$

$(\exists\, i1\ i2.\ v1 = Intg\ i1 \wedge v2 = Intg\ i2 \wedge va = Inl\ (Intg\ (i1\ XOR\ i2)))$
⟨*proof*⟩

lemma [*simp*]:
$(binop\text{-}ShiftLeft\ v1\ v2 = Some\ va) \longleftrightarrow$
$(\exists\, i1\ i2.\ v1 = Intg\ i1 \wedge v2 = Intg\ i2 \wedge va = Inl\ (Intg\ (i1 << unat\ (i2\ AND\ 0x1f))))$
⟨*proof*⟩

lemma [*simp*]:
$(binop\text{-}ShiftRightZeros\ v1\ v2 = Some\ va) \longleftrightarrow$
$(\exists\, i1\ i2.\ v1 = Intg\ i1 \wedge v2 = Intg\ i2 \wedge va = Inl\ (Intg\ (i1 >> unat\ (i2\ AND\ 0x1f))))$
⟨*proof*⟩

lemma [*simp*]:
$(binop\text{-}ShiftRightSigned\ v1\ v2 = Some\ va) \longleftrightarrow$
$(\exists\, i1\ i2.\ v1 = Intg\ i1 \wedge v2 = Intg\ i2 \wedge va = Inl\ (Intg\ (i1 >>> unat\ (i2\ AND\ 0x1f))))$
⟨*proof*⟩

end

context *heap-base*
begin

lemma [*simp*]:
$(binop\text{-}Mod\ v1\ v2 = Some\ va) \longleftrightarrow$
$(\exists\, i1\ i2.\ v1 = Intg\ i1 \wedge v2 = Intg\ i2 \wedge$
$va = (if\ i2 = 0\ then\ Inr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ ArithmeticException)\ else\ Inl\ (Intg(i1\ smod\ i2))))$
⟨*proof*⟩

lemma [*simp*]:
$(binop\text{-}Div\ v1\ v2 = Some\ va) \longleftrightarrow$
$(\exists\, i1\ i2.\ v1 = Intg\ i1 \wedge v2 = Intg\ i2 \wedge$
$va = (if\ i2 = 0\ then\ Inr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ ArithmeticException)\ else\ Inl\ (Intg(i1\ sdiv\ i2))))$
⟨*proof*⟩

end

3.15.2 Typing for binary operators

inductive $WT\text{-}binop :: 'm\ prog \Rightarrow ty \Rightarrow bop \Rightarrow ty \Rightarrow ty \Rightarrow bool$ (‹- ⊢ -«-»- :: -› [51,0,0,0,51] 50)
where
WT-binop-Eq:
$P \vdash T1 \le T2 \vee P \vdash T2 \le T1 \Longrightarrow P \vdash T1 «Eq» T2 :: Boolean$

| *WT-binop-NotEq*:
$P \vdash T1 \le T2 \vee P \vdash T2 \le T1 \Longrightarrow P \vdash T1 «NotEq» T2 :: Boolean$

| *WT-binop-LessThan*:
$P \vdash Integer «LessThan» Integer :: Boolean$

| *WT-binop-LessOrEqual*:
$P \vdash Integer «LessOrEqual» Integer :: Boolean$

| *WT-binop-GreaterThan*:

$P \vdash$ *Integer«GreaterThan»Integer* :: *Boolean*

| *WT-binop-GreaterOrEqual*:
$P \vdash$ *Integer«GreaterOrEqual»Integer* :: *Boolean*

| *WT-binop-Add*:
$P \vdash$ *Integer«Add»Integer* :: *Integer*

| *WT-binop-Subtract*:
$P \vdash$ *Integer«Subtract»Integer* :: *Integer*

| *WT-binop-Mult*:
$P \vdash$ *Integer«Mult»Integer* :: *Integer*

| *WT-binop-Div*:
$P \vdash$ *Integer«Div»Integer* :: *Integer*

| *WT-binop-Mod*:
$P \vdash$ *Integer«Mod»Integer* :: *Integer*

| *WT-binop-BinAnd-Bool*:
$P \vdash$ *Boolean«BinAnd»Boolean* :: *Boolean*

| *WT-binop-BinAnd-Int*:
$P \vdash$ *Integer«BinAnd»Integer* :: *Integer*

| *WT-binop-BinOr-Bool*:
$P \vdash$ *Boolean«BinOr»Boolean* :: *Boolean*

| *WT-binop-BinOr-Int*:
$P \vdash$ *Integer«BinOr»Integer* :: *Integer*

| *WT-binop-BinXor-Bool*:
$P \vdash$ *Boolean«BinXor»Boolean* :: *Boolean*

| *WT-binop-BinXor-Int*:
$P \vdash$ *Integer«BinXor»Integer* :: *Integer*

| *WT-binop-ShiftLeft*:
$P \vdash$ *Integer«ShiftLeft»Integer* :: *Integer*

| *WT-binop-ShiftRightZeros*:
$P \vdash$ *Integer«ShiftRightZeros»Integer* :: *Integer*

| *WT-binop-ShiftRightSigned*:
$P \vdash$ *Integer«ShiftRightSigned»Integer* :: *Integer*

lemma *WT-binopI* [*intro*]:
$P \vdash T1 \leq T2 \lor P \vdash T2 \leq T1 \Longrightarrow P \vdash$ *T1«Eq»T2* :: *Boolean*
$P \vdash T1 \leq T2 \lor P \vdash T2 \leq T1 \Longrightarrow P \vdash$ *T1«NotEq»T2* :: *Boolean*
$bop = Add \lor bop = Subtract \lor bop = Mult \lor bop = Mod \lor bop = Div \lor bop = BinAnd \lor bop = BinOr \lor bop = BinXor \lor$
$bop = ShiftLeft \lor bop = ShiftRightZeros \lor bop = ShiftRightSigned$
$\Longrightarrow P \vdash$ *Integer«bop»Integer* :: *Integer*

bop = *LessThan* ∨ *bop* = *LessOrEqual* ∨ *bop* = *GreaterThan* ∨ *bop* = *GreaterOrEqual* ⟹ *P* ⊢ *Integer*«*bop*»*Integer* :: *Boolean*
bop = *BinAnd* ∨ *bop* = *BinOr* ∨ *bop* = *BinXor* ⟹ *P* ⊢ *Boolean*«*bop*»*Boolean* :: *Boolean*
⟨*proof*⟩

inductive-cases [*elim*]:
P ⊢ *T1*«*Eq*»*T2* :: *T*
P ⊢ *T1*«*NotEq*»*T2* :: *T*
P ⊢ *T1*«*LessThan*»*T2* :: *T*
P ⊢ *T1*«*LessOrEqual*»*T2* :: *T*
P ⊢ *T1*«*GreaterThan*»*T2* :: *T*
P ⊢ *T1*«*GreaterOrEqual*»*T2* :: *T*
P ⊢ *T1*«*Add*»*T2* :: *T*
P ⊢ *T1*«*Subtract*»*T2* :: *T*
P ⊢ *T1*«*Mult*»*T2* :: *T*
P ⊢ *T1*«*Div*»*T2* :: *T*
P ⊢ *T1*«*Mod*»*T2* :: *T*
P ⊢ *T1*«*BinAnd*»*T2* :: *T*
P ⊢ *T1*«*BinOr*»*T2* :: *T*
P ⊢ *T1*«*BinXor*»*T2* :: *T*
P ⊢ *T1*«*ShiftLeft*»*T2* :: *T*
P ⊢ *T1*«*ShiftRightZeros*»*T2* :: *T*
P ⊢ *T1*«*ShiftRightSigned*»*T2* :: *T*

lemma *WT-binop-fun*: ⟦ *P* ⊢ *T1*«*bop*»*T2* :: *T*; *P* ⊢ *T1*«*bop*»*T2* :: *T′* ⟧ ⟹ *T* = *T′*
⟨*proof*⟩

lemma *WT-binop-is-type*:
⟦ *P* ⊢ *T1*«*bop*»*T2* :: *T*; *is-type P T1*; *is-type P T2* ⟧ ⟹ *is-type P T*
⟨*proof*⟩

inductive *WTrt-binop* :: *′m prog* ⇒ *ty* ⇒ *bop* ⇒ *ty* ⇒ *ty* ⇒ *bool* (‹- ⊢ -«-»- : -› [*51*,*0*,*0*,*0*,*51*] *50*)
where
WTrt-binop-Eq:
P ⊢ *T1*«*Eq*»*T2* : *Boolean*

| *WTrt-binop-NotEq*:
P ⊢ *T1*«*NotEq*»*T2* : *Boolean*

| *WTrt-binop-LessThan*:
P ⊢ *Integer*«*LessThan*»*Integer* : *Boolean*

| *WTrt-binop-LessOrEqual*:
P ⊢ *Integer*«*LessOrEqual*»*Integer* : *Boolean*

| *WTrt-binop-GreaterThan*:
P ⊢ *Integer*«*GreaterThan*»*Integer* : *Boolean*

| *WTrt-binop-GreaterOrEqual*:
P ⊢ *Integer*«*GreaterOrEqual*»*Integer* : *Boolean*

| *WTrt-binop-Add*:
P ⊢ *Integer*«*Add*»*Integer* : *Integer*

| *WTrt-binop-Subtract*:
$P \vdash$ *Integer*«*Subtract*»*Integer* : *Integer*

| *WTrt-binop-Mult*:
$P \vdash$ *Integer*«*Mult*»*Integer* : *Integer*

| *WTrt-binop-Div*:
$P \vdash$ *Integer*«*Div*»*Integer* : *Integer*

| *WTrt-binop-Mod*:
$P \vdash$ *Integer*«*Mod*»*Integer* : *Integer*

| *WTrt-binop-BinAnd-Bool*:
$P \vdash$ *Boolean*«*BinAnd*»*Boolean* : *Boolean*

| *WTrt-binop-BinAnd-Int*:
$P \vdash$ *Integer*«*BinAnd*»*Integer* : *Integer*

| *WTrt-binop-BinOr-Bool*:
$P \vdash$ *Boolean*«*BinOr*»*Boolean* : *Boolean*

| *WTrt-binop-BinOr-Int*:
$P \vdash$ *Integer*«*BinOr*»*Integer* : *Integer*

| *WTrt-binop-BinXor-Bool*:
$P \vdash$ *Boolean*«*BinXor*»*Boolean* : *Boolean*

| *WTrt-binop-BinXor-Int*:
$P \vdash$ *Integer*«*BinXor*»*Integer* : *Integer*

| *WTrt-binop-ShiftLeft*:
$P \vdash$ *Integer*«*ShiftLeft*»*Integer* : *Integer*

| *WTrt-binop-ShiftRightZeros*:
$P \vdash$ *Integer*«*ShiftRightZeros*»*Integer* : *Integer*

| *WTrt-binop-ShiftRightSigned*:
$P \vdash$ *Integer*«*ShiftRightSigned*»*Integer* : *Integer*

lemma *WTrt-binopI* [*intro*]:
$P \vdash$ *T1*«*Eq*»*T2* : *Boolean*
$P \vdash$ *T1*«*NotEq*»*T2* : *Boolean*
bop = *Add* ∨ *bop* = *Subtract* ∨ *bop* = *Mult* ∨ *bop* = *Div* ∨ *bop* = *Mod* ∨ *bop* = *BinAnd* ∨ *bop* = *BinOr* ∨ *bop* = *BinXor* ∨
bop = *ShiftLeft* ∨ *bop* = *ShiftRightZeros* ∨ *bop* = *ShiftRightSigned*
$\Longrightarrow P \vdash$ *Integer*«*bop*»*Integer* : *Integer*
bop = *LessThan* ∨ *bop* = *LessOrEqual* ∨ *bop* = *GreaterThan* ∨ *bop* = *GreaterOrEqual* $\Longrightarrow P \vdash$ *Integer*«*bop*»*Integer* : *Boolean*
bop = *BinAnd* ∨ *bop* = *BinOr* ∨ *bop* = *BinXor* $\Longrightarrow P \vdash$ *Boolean*«*bop*»*Boolean* : *Boolean*
⟨*proof*⟩

inductive-cases *WTrt-binop-cases* [*elim*]:
$P \vdash$ *T1*«*Eq*»*T2* : *T*
$P \vdash$ *T1*«*NotEq*»*T2* : *T*

$P \vdash T1$ «*LessThan*» $T2 : T$
$P \vdash T1$ «*LessOrEqual*» $T2 : T$
$P \vdash T1$ «*GreaterThan*» $T2 : T$
$P \vdash T1$ «*GreaterOrEqual*» $T2 : T$
$P \vdash T1$ «*Add*» $T2 : T$
$P \vdash T1$ «*Subtract*» $T2 : T$
$P \vdash T1$ «*Mult*» $T2 : T$
$P \vdash T1$ «*Div*» $T2 : T$
$P \vdash T1$ «*Mod*» $T2 : T$
$P \vdash T1$ «*BinAnd*» $T2 : T$
$P \vdash T1$ «*BinOr*» $T2 : T$
$P \vdash T1$ «*BinXor*» $T2 : T$
$P \vdash T1$ «*ShiftLeft*» $T2 : T$
$P \vdash T1$ «*ShiftRightZeros*» $T2 : T$
$P \vdash T1$ «*ShiftRightSigned*» $T2 : T$

inductive-simps *WTrt-binop-simps* [*simp*]:
$P \vdash T1$ «*Eq*» $T2 : T$
$P \vdash T1$ «*NotEq*» $T2 : T$
$P \vdash T1$ «*LessThan*» $T2 : T$
$P \vdash T1$ «*LessOrEqual*» $T2 : T$
$P \vdash T1$ «*GreaterThan*» $T2 : T$
$P \vdash T1$ «*GreaterOrEqual*» $T2 : T$
$P \vdash T1$ «*Add*» $T2 : T$
$P \vdash T1$ «*Subtract*» $T2 : T$
$P \vdash T1$ «*Mult*» $T2 : T$
$P \vdash T1$ «*Div*» $T2 : T$
$P \vdash T1$ «*Mod*» $T2 : T$
$P \vdash T1$ «*BinAnd*» $T2 : T$
$P \vdash T1$ «*BinOr*» $T2 : T$
$P \vdash T1$ «*BinXor*» $T2 : T$
$P \vdash T1$ «*ShiftLeft*» $T2 : T$
$P \vdash T1$ «*ShiftRightZeros*» $T2 : T$
$P \vdash T1$ «*ShiftRightSigned*» $T2 : T$

fun *binop-relevant-class* :: $bop \Rightarrow {'m}\ prog \Rightarrow cname \Rightarrow bool$
where
binop-relevant-class Div = ($\lambda P\ C.\ P \vdash ArithmeticException \preceq^* C$)
| *binop-relevant-class Mod* = ($\lambda P\ C.\ P \vdash ArithmeticException \preceq^* C$)
| *binop-relevant-class* - = ($\lambda P\ C.\ False$)

lemma *WT-binop-WTrt-binop*:
$P \vdash T1$ «*bop*» $T2 :: T \Longrightarrow P \vdash T1$ «*bop*» $T2 : T$
⟨*proof*⟩

context *heap* **begin**

lemma *binop-progress*:
$[\![\ typeof_h\ v1 = \lfloor T1 \rfloor;\ typeof_h\ v2 = \lfloor T2 \rfloor;\ P \vdash T1$ «*bop*» $T2 : T\ ]\!]$
$\Longrightarrow \exists va.\ binop\ bop\ v1\ v2 = \lfloor va \rfloor$
⟨*proof*⟩

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*
⟨*proof*⟩

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* $\preceq^*$ *C*
shows *binop-relevant-class bop P C*
⟨*proof*⟩

end

lemma *WTrt-binop-fun*: ⟦ *P* ⊢ *T1*«*bop*»*T2* : *T*; *P* ⊢ *T1*«*bop*»*T2* : *T'* ⟧ ⟹ *T* = *T'*
⟨*proof*⟩

lemma *WTrt-binop-THE* [*simp*]: *P* ⊢ *T1*«*bop*»*T2* : *T* ⟹ *The* (*WTrt-binop P T1 bop T2*) = *T*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *WTrt-binop-is-type*:
⟦ *P* ⊢ *T1*«*bop*»*T2* : *T*; *is-type P T1*; *is-type P T2* ⟧ ⟹ *is-type P T*
⟨*proof*⟩

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
⟨*proof*⟩

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*)
WTrt-binop
⟨*proof*⟩

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*
⟨*proof*⟩

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*)

⟨*proof*⟩

end

3.16 The Jinja Type System as a Semilattice

theory *SemiType*
imports
WellForm
../DFA/Semilattices
begin

inductive-set
widen1 :: *'a prog* $\Rightarrow$ (*ty* $\times$ *ty*) *set*
and *widen1-syntax* :: *'a prog* $\Rightarrow$ *ty* $\Rightarrow$ *ty* $\Rightarrow$ *bool* (‹- ⊢ - $<^1$ -› [*71*,*71*,*71*] *70*)
for *P* :: *'a prog*
where
$P \vdash C <^1 D \equiv (C, D) \in$ *widen1 P*

| *widen1-Array-Object*:
$P \vdash$ *Array* (*Class Object*) $<^1$ *Class Object*

| *widen1-Array-Integer*:
$P \vdash$ *Array Integer* $<^1$ *Class Object*

| *widen1-Array-Boolean*:
$P \vdash$ *Array Boolean* $<^1$ *Class Object*

| *widen1-Array-Void*:
$P \vdash$ *Array Void* $<^1$ *Class Object*

| *widen1-Class*:
$P \vdash C \prec^1 D \Longrightarrow P \vdash$ *Class C* $<^1$ *Class D*

| *widen1-Array-Array*:
$[\![\, P \vdash T <^1 U;\ \neg$ *is-NT-Array T* $]\!] \Longrightarrow P \vdash$ *Array T* $<^1$ *Array U*

abbreviation *widen1-trancl* :: *'a prog* $\Rightarrow$ *ty* $\Rightarrow$ *ty* $\Rightarrow$ *bool* (‹- ⊢ - $<^+$ -› [*71*,*71*,*71*] *70*) **where**
$P \vdash T <^+ U \equiv (T, U) \in$ *trancl* (*widen1 P*)

abbreviation *widen1-rtrancl* :: *'a prog* $\Rightarrow$ *ty* $\Rightarrow$ *ty* $\Rightarrow$ *bool* (‹- ⊢ - $<^*$ -› [*71*,*71*,*71*] *70*) **where**
$P \vdash T <^* U \equiv (T, U) \in$ *rtrancl* (*widen1 P*)

inductive-simps *widen1-simps1* [*simp*]:
$P \vdash$ *Integer* $<^1 T$
$P \vdash$ *Boolean* $<^1 T$
$P \vdash$ *Void* $<^1 T$
$P \vdash$ *Class Object* $<^1 T$
$P \vdash$ *NT* $<^1 U$

inductive-simps *widen1-simps* [*simp*]:
$P \vdash$ *Array* (*Class Object*) $<^1 T$
$P \vdash$ *Array Integer* $<^1 T$

$P \vdash$ *Array Boolean* $<^1$ T
$P \vdash$ *Array Void* $<^1$ T
$P \vdash$ *Class* $C <^1 T$
$P \vdash T <^1$ *Array* U

lemma *is-type-widen1*:
assumes *icO*: *is-class P Object*
shows $P \vdash T <^1 U \Longrightarrow$ *is-type P T*
⟨*proof*⟩

lemma *widen1-NT-Array*:
assumes *is-NT-Array T*
shows $\neg P \vdash T\lfloor\rceil <^1 U$
⟨*proof*⟩

lemma *widen1-is-type*:
assumes *wfP*: *wf-prog wfmd P*
shows $(A, B) \in$ *widen1 P* $\Longrightarrow$ *is-type P B*
⟨*proof*⟩

lemma *widen1-trancl-is-type*:
assumes *wfP*: *wf-prog wfmd P*
shows $(A, B) \in$ (*widen1 P*)^+ $\Longrightarrow$ *is-type P B*
⟨*proof*⟩

lemma *single-valued-widen1*:
assumes *wf*: *wf-prog wf-md P*
shows *single-valued* (*widen1 P*)
⟨*proof*⟩

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)
⟨*proof*⟩
termination
⟨*proof*⟩

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*: $P \vdash x <^1 y$
shows *subtype-measure P y* < *subtype-measure P x*
⟨*proof*⟩

lemma *wf-converse-widen1*:
assumes *wfP*: *wf-prog wfmc P*
shows *wf* ((*widen1 P*)^−*1*)
⟨*proof*⟩

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⟩

lemma *Class-widen1-super*:
  P ⊢ Class C' $<^1$ U' ⟷ is-class P C' ∧ C' ≠ Object ∧ U' = super P (Class C')
  (**is** ?lhs ⟷ ?rhs)
⟨proof⟩

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⟩

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)
  ⟨proof⟩

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)
⟨proof⟩

lemma *subcls1-trancl-widen1-trancl*:
  (subcls1 P)^++ C D ⟹ P ⊢ Class C $<^+$ Class D
⟨proof⟩

lemma *subcls-into-widen1-rtrancl*:
$P \vdash C \preceq^* D \Longrightarrow P \vdash \mathit{Class}\ C <^* \mathit{Class}\ D$
⟨*proof*⟩

lemma *not-widen1-NT-Array*:
$P \vdash U <^1 T \Longrightarrow \neg\ \mathit{is\text{-}NT\text{-}Array}\ T$
⟨*proof*⟩

lemma *widen1-trancl-into-Array-widen1-trancl*:
$[\![\ P \vdash A <^+ B;\ \neg\ \mathit{is\text{-}NT\text{-}Array}\ A\ ]\!] \Longrightarrow P \vdash A\lfloor\rceil <^+ B\lfloor\rceil$
⟨*proof*⟩

lemma *widen1-rtrancl-into-Array-widen1-rtrancl*:
$[\![\ P \vdash A <^* B;\ \neg\ \mathit{is\text{-}NT\text{-}Array}\ A\ ]\!] \Longrightarrow P \vdash A\lfloor\rceil <^* B\lfloor\rceil$
⟨*proof*⟩

lemma *Array-Object-widen1-trancl*:
assumes *wf*: *wf-prog wmdc P*
and *itA*: *is-type P* $(A\lfloor\rceil)$
shows $P \vdash A\lfloor\rceil <^+ \mathit{Class\ Object}$
⟨*proof*⟩

lemma *widen-into-widen1-trancl*:
assumes *wf*: *wf-prog wfmd P*
shows $[\![\ P \vdash A \leq B;\ A \neq B;\ A \neq NT;\ \mathit{is\text{-}type}\ P\ A\ ]\!] \Longrightarrow P \vdash A <^+ B$
⟨*proof*⟩

lemma *wf-prog-impl-acc-widen*:
assumes *wfP*: *wf-prog wfmd P*
shows *acc* (*types P*) (*widen P*)
⟨*proof*⟩

lemmas *wf-widen-acc* = *wf-prog-impl-acc-widen*
declare *wf-widen-acc* [*intro*, *simp*]

lemma *acyclic-widen1*:
wf-prog wfmc P $\Longrightarrow$ *acyclic* (*widen1 P*)
⟨*proof*⟩

lemma *widen1-into-widen*:
$(A, B) \in \mathit{widen1}\ P \Longrightarrow P \vdash A \leq B$
⟨*proof*⟩

lemma *widen1-rtrancl-into-widen*:
$P \vdash A <^* B \Longrightarrow P \vdash A \leq B$
⟨*proof*⟩

lemma *widen-eq-widen1-trancl*:
$[\![\ \mathit{wf\text{-}prog}\ \mathit{wf\text{-}md}\ P;\ T \neq NT;\ T \neq U;\ \mathit{is\text{-}type}\ P\ T\ ]\!] \Longrightarrow P \vdash T \leq U \longleftrightarrow P \vdash T <^+ U$
⟨*proof*⟩

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*⟩

lemma *closed-err-types*:
assumes *wfP*: *wf-prog wf-mb P*
shows *closed* (*err* (*types P*)) (*lift2* (*sup P*))
⟨*proof*⟩

lemma *widen-into-widen1-rtrancl*:
$[\![$*wf-prog wfmd P*; *widen P A B*; $A \neq NT$; *is-type P A* $]\!] \Longrightarrow (A,\ B) \in (widen1\ P)^*$
⟨*proof*⟩

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*⟩

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*: $P \vdash T \leq V$
and *UwV*: $P \vdash U \leq V$
and *sup*: *sup P T U* = *OK W*
shows *widen P W V*
⟨*proof*⟩

lemma *sup-exists*:
$[\![$ *widen P a c*; *widen P b c* $]\!] \Longrightarrow \exists\, T.$ *sup P a b* = *OK T*
⟨*proof*⟩

lemma *err-semilat-JType-esl*:
assumes *wf-prog*: *wf-prog wf-mb P*
shows *err-semilat* (*esl P*)
⟨*proof*⟩

3.16.1 Relation between *SemiType.sup P T U* = *OK V* and $P \vdash lub(T,\ U) = V$

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 $P \vdash lub(T,\ U) = V$
⟨*proof*⟩

lemma *is-lub-subD*:
assumes *wf*: *wf-prog wf-md P*

and *it*: *is-type* P T *is-type* P U
and *lub*: $P \vdash lub(T,\ U) = V$
shows *sup* P T U $=$ *OK* V
⟨*proof*⟩

lemma *is-lub-is-type*:
⟦ *wf-prog wf-md* P; *is-type* P T; *is-type* P U; $P \vdash lub(T,\ U) = V$ ⟧ $\Longrightarrow$ *is-type* P V
⟨*proof*⟩

3.16.2 Code generator setup

code-pred *widen1p* ⟨*proof*⟩
lemmas [*code*] = *widen1-def*

lemma *eval-widen1p-i-i-o-conv*:
Predicate.eval (*widen1p-i-i-o* P T) $= (\lambda U.\ P \vdash T <^1 U)$
⟨*proof*⟩

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)$\}$
⟨*proof*⟩

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
$'addr\ locals = vname \rightharpoonup 'addr\ val$ — local vars, incl. params and "this"
type-synonym
$('addr, 'heap)\ Jstate = 'heap \times 'addr\ locals$ — the heap and the local vars

definition $hp :: 'heap \times 'x \Rightarrow 'heap$ **where** $hp \equiv fst$

definition $lcl :: 'heap \times 'x \Rightarrow 'x$ **where** $lcl \equiv snd$

lemma *hp-conv* [*simp*]: $hp\ (h, l) = h$
⟨*proof*⟩

lemma *lcl-conv* [*simp*]: $lcl\ (h, l) = l$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *expr-ineqs* [*simp*]: *Val v* ;; *e* ≠ *e* *if* (*e1*) *e else e2* ≠ *e* *if* (*e1*) *e2 else e* ≠ *e*
⟨*proof*⟩

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*]: $fvs(es\ @\ es') = fvs\ es \cup fvs\ es'$
⟨*proof*⟩

lemma [*simp*]: $fvs(map\ Val\ vs) = \{\}$
⟨*proof*⟩

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-locks $(new\ C) = (\lambda ad.\ 0)$
| *expr-locks* $(newA\ T\lfloor e\rceil)$ = *expr-locks* e
| *expr-locks* $(Cast\ T\ e)$ = *expr-locks* e
| *expr-locks* $(e\ instanceof\ T)$ = *expr-locks* e
| *expr-locks* $(Val\ v) = (\lambda ad.\ 0)$
| *expr-locks* $(Var\ v) = (\lambda ad.\ 0)$
| *expr-locks* $(e\ «bop»\ e') = (\lambda ad.$ *expr-locks* $e\ ad$ + *expr-locks* $e'\ ad)$
| *expr-locks* $(V := e)$ = *expr-locks* e
| *expr-locks* $(a\lfloor i\rceil) = (\lambda ad.$ *expr-locks* $a\ ad$ + *expr-locks* $i\ ad)$
| *expr-locks* $(AAss\ a\ i\ e) = (\lambda ad.$ *expr-locks* $a\ ad$ + *expr-locks* $i\ ad$ + *expr-locks* $e\ ad)$
| *expr-locks* $(a{\cdot}length)$ = *expr-locks* a
| *expr-locks* $(e{\cdot}F\{D\})$ = *expr-locks* e
| *expr-locks* $(FAss\ e\ F\ D\ e') = (\lambda ad.$ *expr-locks* $e\ ad$ + *expr-locks* $e'\ ad)$
| *expr-locks* $(e{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e'')) = (\lambda ad.$ *expr-locks* $e\ ad$ + *expr-locks* $e'\ ad$ + *expr-locks* $e''\ ad)$
| *expr-locks* $(e{\cdot}m(ps)) = (\lambda ad.$ *expr-locks* $e\ ad$ + *expr-lockss* $ps\ ad)$
| *expr-locks* $(\{V : T{=}vo;\ e\})$ = *expr-locks* e
| *expr-locks* $(sync_V\ (o')\ e) = (\lambda ad.$ *expr-locks* $o'\ ad$ + *expr-locks* $e\ ad)$
| *expr-locks* $(insync_V\ (a)\ e) = (\lambda ad.\ if\ (a = ad)\ then\ Suc$ (*expr-locks* $e\ ad$) $else$ *expr-locks* $e\ ad)$
| *expr-locks* $(e;;e') = (\lambda ad.$ *expr-locks* $e\ ad$ + *expr-locks* $e'\ ad)$
| *expr-locks* $(if\ (b)\ e\ else\ e') = (\lambda ad.$ *expr-locks* $b\ ad$ + *expr-locks* $e\ ad$ + *expr-locks* $e'\ ad)$
| *expr-locks* $(while\ (b)\ e) = (\lambda ad.$ *expr-locks* $b\ ad$ + *expr-locks* $e\ ad)$
| *expr-locks* $(throw\ e)$ = *expr-locks* e
| *expr-locks* $(try\ e\ catch(C\ v)\ e') = (\lambda ad.$ *expr-locks* $e\ ad$ + *expr-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*)
⟨*proof*⟩

lemma *expr-lockss-map-Val* [*simp*]: *expr-lockss* (*map Val vs*) = (λ*ad*. *0*)
⟨*proof*⟩

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'*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *contains-insyncs-map-Val* [*simp*]: ¬ *contains-insyncs* (*map Val vs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

code-pred *is-val* ⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *is-vals-conv*: *is-vals es* = (∃ *vs*. *es* = *map Val vs*)
⟨*proof*⟩

lemma *is-vals-map-Vals* [*simp*]: *is-vals* (*map Val vs*) = *True*
⟨*proof*⟩

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*)
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *is-Throws-conv*: *is-Throws es* ⟷ (∃ *vs a es'*. *es* = *map Val vs* @ *Throw a* # *es'*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *expr-locks-blocks*:
⟦ *length vs* = *length pns*; *length Ts* = *length pns* ⟧
⟹ *expr-locks* (*blocks pns Ts vs e*) = *expr-locks e*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *final-locks*: *final e* ⟹ *expr-locks e l* = *0*
⟨*proof*⟩

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* []
⟨*proof*⟩

lemma [*iff*]: *finals* (*Val v* # *es*) = *finals es*
⟨*proof*⟩

lemma *finals-app-map* [*iff*]: *finals* (*map Val vs* @ *es*) = *finals es*
⟨*proof*⟩

lemma [*iff*]: *finals* (*throw e* # *es*) = (∃ *a*. *e* = *addr a*)
⟨*proof*⟩

lemma *not-finals-ConsI*: ¬ *final e* ⟹ ¬ *finals* (*e* # *es*)

⟨*proof*⟩

lemma *finals-iff*: *finals es* ⟷ (∃ *vs*. *es* = *map Val vs*) ∨ (∃ *vs a es*′. *es* = *map Val vs* @ *Throw a* # *es*′)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

code-pred *final* ⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *calls-map-Val* [*simp*]:
calls (*map Val vs*) = *None*
⟨*proof*⟩

lemma *call-not-is-val* [*dest*]: *call e* = ⌊*aMvs*⌋ ⟹ ¬ *is-val e*
⟨*proof*⟩

lemma *is-calls-not-is-vals* [*dest*]: *calls es* = ⌊*aMvs*⌋ ⟹ ¬ *is-vals es*
⟨*proof*⟩

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* ⟨*proof*⟩

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* ⟨*proof*⟩

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*
⟨*proof*⟩

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* ⟨*proof*⟩

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* ⟨*proof*⟩

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*

⟨*ML*⟩
typ (*'addr*,*'thread-id*,*'heap*) *J-thread-action*

⟨*ML*⟩
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*)
⟨*proof*⟩

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
⟨*proof*⟩

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) ∈ allocate h (Class-type C)
⟹ extTA,P,t ⊢ ⟨new C, (h, l)⟩ −⦃NewHeapElem a (Class-type C)⦄→ ⟨addr a, (h', l)⟩

| *RedNewFail*:
allocate h (Class-type C) = {}
⟹ extTA,P,t ⊢ ⟨new C, (h, l)⟩ −ε→ ⟨THROW OutOfMemory, (h, l)⟩

| *NewArrayRed*:
extTA,P,t ⊢ ⟨e, s⟩ −ta→ ⟨e', s'⟩ ⟹ extTA,P,t ⊢ ⟨newA T⌊e⌉, s⟩ −ta→ ⟨newA T⌊e'⌉, s'⟩

| *RedNewArray*:
⟦ 0 <=s i; (h', a) ∈ allocate h (Array-type T (nat (sint i))) ⟧
⟹ extTA,P,t ⊢ ⟨newA T⌊Val (Intg i)⌉, (h, l)⟩ −⦃NewHeapElem a (Array-type T (nat (sint i)))⦄→ ⟨addr a, (h', l)⟩

| *RedNewArrayNegative*:
i <s 0 ⟹ extTA,P,t ⊢ ⟨newA T⌊Val (Intg i)⌉, s⟩ −ε→ ⟨THROW NegativeArraySize, s⟩

| *RedNewArrayFail*:
⟦ 0 <=s i; allocate h (Array-type T (nat (sint i))) = {} ⟧
⟹ extTA,P,t ⊢ ⟨newA T⌊Val (Intg i)⌉, (h, l)⟩ −ε→ ⟨THROW OutOfMemory, (h, l)⟩

| *CastRed*:
extTA,P,t ⊢ ⟨e, s⟩ −ta→ ⟨e', s'⟩ ⟹ extTA,P,t ⊢ ⟨Cast C e, s⟩ −ta→ ⟨Cast C e', s'⟩

| *RedCast*:
⟦ $typeof_{hp\ s}$ v = ⌊U⌋; P ⊢ U ≤ T ⟧
⟹ extTA,P,t ⊢ ⟨Cast T (Val v), s⟩ −ε→ ⟨Val v, s⟩

| *RedCastFail*:
⟦ $typeof_{hp\ s}$ v = ⌊U⌋; ¬ P ⊢ U ≤ T ⟧
⟹ extTA,P,t ⊢ ⟨Cast T (Val v), s⟩ −ε→ ⟨THROW ClassCast, s⟩

| *InstanceOfRed*:
extTA,P,t ⊢ ⟨e, s⟩ −ta→ ⟨e', s'⟩ ⟹ extTA,P,t ⊢ ⟨e instanceof T, s⟩ −ta→ ⟨e' instanceof T, s'⟩

| *RedInstanceOf*:
⟦ $typeof_{hp\ s}$ v = ⌊U⌋; b ⟷ v ≠ Null ∧ P ⊢ U ≤ T ⟧
⟹ extTA,P,t ⊢ ⟨(Val v) instanceof T, s⟩ −ε→ ⟨Val (Bool b), s⟩

| *BinOpRed1*:
extTA,P,t ⊢ ⟨e, s⟩ −ta→ ⟨e', s'⟩ ⟹ extTA,P,t ⊢ ⟨e «bop» e2, s⟩ −ta→ ⟨e' «bop» e2, s'⟩

| *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*:
$[\![\ typeof\text{-}addr\ (hp\ s)\ a = \lfloor Array\text{-}type\ T\ n\rfloor;\ i <s\ 0 \vee sint\ i \geq int\ n\ ]\!]$
$\Longrightarrow extTA,P,t \vdash \langle(addr\ a)\lfloor Val\ (Intg\ i)\rceil, s\rangle -\varepsilon\rightarrow \langle THROW\ ArrayIndexOutOfBounds, s\rangle$

| *RedAAcc*:
$[\![\ typeof\text{-}addr\ h\ a = \lfloor Array\text{-}type\ T\ n\rfloor;\ 0 <=s\ i;\ sint\ i < int\ n;$
$heap\text{-}read\ h\ a\ (ACell\ (nat\ (sint\ i)))\ v\ ]\!]$
$\Longrightarrow extTA,P,t \vdash \langle(addr\ a)\lfloor Val\ (Intg\ i)\rceil, (h, l)\rangle -\{\!|ReadMem\ a\ (ACell\ (nat\ (sint\ i)))\ v|\!\}\rightarrow \langle Val\ v, (h, l)\rangle$

| *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*:
$[\![\ typeof\text{-}addr\ (hp\ s)\ a = \lfloor Array\text{-}type\ T\ n\rfloor;\ i <s\ 0 \vee sint\ i \geq int\ n\ ]\!]$

$\Longrightarrow$ *extTA,P,t* ⊢ ⟨(*addr a*)⌊*Val* (*Intg i*)⌉ := (*Val e*::′*addr expr*), *s*⟩ −ε→ ⟨*THROW ArrayIndexOutOfBounds*, *s*⟩

| *RedAAssStore*:
⟦ *typeof-addr* (*hp s*) *a* = ⌊*Array-type T n*⌋; *0* <=*s i*; *sint i* < *int n*;
typeof$_{hp\ s}$ *w* = ⌊*U*⌋; ¬ (*P* ⊢ *U* ≤ *T*) ⟧
$\Longrightarrow$ *extTA,P,t* ⊢ ⟨(*addr a*)⌊*Val* (*Intg i*)⌉ := (*Val w*::′*addr expr*), *s*⟩ −ε→ ⟨*THROW ArrayStore*, *s*⟩

| *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*′ ⟧
$\Longrightarrow$ *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*′⟩ $\Longrightarrow$ *extTA,P,t* ⊢ ⟨*a*•*length*, *s*⟩ −*ta*→ ⟨*a*′•*length*, *s*′⟩

| *RedALength*:
typeof-addr h a = ⌊*Array-type T n*⌋
$\Longrightarrow$ *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*′⟩ $\Longrightarrow$ *extTA,P,t* ⊢ ⟨*e*•*F*{*D*}, *s*⟩ −*ta*→ ⟨*e*′•*F*{*D*}, *s*′⟩

| *RedFAcc*:
heap-read h a (*CField D F*) *v*
$\Longrightarrow$ *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*′⟩ $\Longrightarrow$ *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*′⟩ $\Longrightarrow$ *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*′ $\Longrightarrow$
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*′⟩ $\Longrightarrow$
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*′⟩ $\Longrightarrow$

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 ⊢ ⟨*null*∙*compareAndSwap*(*D*∙*F*, *Val v*, *Val v*′), *s*⟩ −ε→ ⟨*THROW NullPointer*, *s*⟩

| *RedCASSucceed*:
⟦ *heap-read h a* (*CField D F*) *v*; *heap-write h a* (*CField D F*) *v*′ *h*′ ⟧ ⟹
extTA,P,t ⊢ ⟨*addr a*∙*compareAndSwap*(*D*∙*F*, *Val v*, *Val v*′), (*h*, *l*)⟩
−⦃*ReadMem a* (*CField D F*) *v*, *WriteMem a* (*CField D F*) *v*′⦄→
⟨*true*, (*h*′, *l*)⟩

| *RedCASFail*:
⟦ *heap-read h a* (*CField D F*) *v*″; *v* ≠ *v*″ ⟧ ⟹
extTA,P,t ⊢ ⟨*addr a*∙*compareAndSwap*(*D*∙*F*, *Val v*, *Val v*′), (*h*, *l*)⟩
−⦃*ReadMem a* (*CField D F*) *v*″⦄→
⟨*false*, (*h*, *l*)⟩

| *CallObj*:
extTA,P,t ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩ ⟹ *extTA,P,t* ⊢ ⟨*e*∙*M*(*es*), *s*⟩ −*ta*→ ⟨*e*′∙*M*(*es*), *s*′⟩

| *CallParams*:
extTA,P,t ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*′,*s*′⟩ ⟹
extTA,P,t ⊢ ⟨(*Val v*)∙*M*(*es*),*s*⟩ −*ta*→ ⟨(*Val v*)∙*M*(*es*′),*s*′⟩

| *RedCall*:
⟦ *typeof-addr* (*hp s*) *a* = ⌊*hU*⌋; *P* ⊢ *class-type-of hU sees M*:*Ts*→*T* = ⌊(*pns*,*body*)⌋ *in D*;
size vs = *size pns*; *size Ts* = *size pns* ⟧
⟹ *extTA,P,t* ⊢ ⟨(*addr a*)∙*M*(*map Val vs*), *s*⟩ −ε→ ⟨*blocks* (*this* # *pns*) (*Class D* # *Ts*) (*Addr a* # *vs*) *body*, *s*⟩

| *RedCallExternal*:
⟦ *typeof-addr* (*hp s*) *a* = ⌊*hU*⌋; *P* ⊢ *class-type-of hU sees M*:*Ts*→*T* = *Native in D*;
P,*t* ⊢ ⟨*a*∙*M*(*vs*), *hp s*⟩ −*ta*→*ext* ⟨*va*, *h*′⟩;
ta′ = *extTA ta*; *e*′ = *extRet2J* ((*addr a*)∙*M*(*map Val vs*)) *va*; *s*′ = (*h*′, *lcl s*) ⟧
⟹ *extTA,P,t* ⊢ ⟨(*addr a*)∙*M*(*map Val vs*), *s*⟩ −*ta*′→ ⟨*e*′, *s*′⟩

| *RedCallNull*:
extTA,P,t ⊢ ⟨*null*∙*M*(*map Val vs*), *s*⟩ −ε→ ⟨*THROW NullPointer*, *s*⟩

| *BlockRed*:
extTA,P,t ⊢ ⟨*e*, (*h*, *l*(*V*:=*vo*))⟩ −*ta*→ ⟨*e*′, (*h*′, *l*′)⟩
⟹ *extTA,P,t* ⊢ ⟨{*V*:*T*=*vo*; *e*}, (*h*, *l*)⟩ −*ta*→ ⟨{*V*:*T*=*l*′ *V*; *e*′}, (*h*′, *l*′(*V* := *l V*))⟩

| *RedBlock*:
extTA,P,t ⊢ ⟨{*V*:*T*=*vo*; *Val u*}, *s*⟩ −ε→ ⟨*Val u*, *s*⟩

| *SynchronizedRed1*:
extTA,P,t ⊢ ⟨*o*′, *s*⟩ −*ta*→ ⟨*o*″, *s*′⟩ ⟹ *extTA,P,t* ⊢ ⟨*sync*(*o*′) *e*, *s*⟩ −*ta*→ ⟨*sync*(*o*″) *e*, *s*′⟩

| *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 ⊢ ⟨*e*#*es*,*s*⟩ [−*ta*→] ⟨*e′*#*es*,*s′*⟩

| *ListRed2*:
extTA,P,t ⊢ ⟨*es*,*s*⟩ [−*ta*→] ⟨*es′*,*s′*⟩ ⟹
extTA,P,t ⊢ ⟨*Val v* # *es*,*s*⟩ [−*ta*→] ⟨*Val v* # *es′*,*s′*⟩

— Exception propagation

| *NewArrayThrow*: *extTA,P,t* ⊢ ⟨*newA T*⌊*Throw a*⌉, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *CastThrow*: *extTA,P,t* ⊢ ⟨*Cast C* (*Throw a*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *InstanceOfThrow*: *extTA,P,t* ⊢ ⟨(*Throw a*) *instanceof T*, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *BinOpThrow1*: *extTA,P,t* ⊢ ⟨(*Throw a*) «*bop*» e_2, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *BinOpThrow2*: *extTA,P,t* ⊢ ⟨(*Val* v_1) «*bop*» (*Throw a*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *LAssThrow*: *extTA,P,t* ⊢ ⟨*V*:=(*Throw a*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *AAccThrow1*: *extTA,P,t* ⊢ ⟨(*Throw a*)⌊*i*⌉, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *AAccThrow2*: *extTA,P,t* ⊢ ⟨(*Val v*)⌊*Throw a*⌉, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *AAssThrow1*: *extTA,P,t* ⊢ ⟨(*Throw a*)⌊*i*⌉ := *e*, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *AAssThrow2*: *extTA,P,t* ⊢ ⟨(*Val v*)⌊*Throw a*⌉ := *e*, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *AAssThrow3*: *extTA,P,t* ⊢ ⟨(*Val v*)⌊*Val i*⌉ := *Throw a* :: *′addr expr*, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *ALengthThrow*: *extTA,P,t* ⊢ ⟨(*Throw a*)·*length*, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *FAccThrow*: *extTA,P,t* ⊢ ⟨(*Throw a*)·*F*{*D*}, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *FAssThrow1*: *extTA,P,t* ⊢ ⟨(*Throw a*)·*F*{*D*}:=e_2, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *FAssThrow2*: *extTA,P,t* ⊢ ⟨*Val v*·*F*{*D*}:=(*Throw a*::*′addr expr*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *CASThrow*: *extTA,P,t* ⊢ ⟨*Throw a*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *CASThrow2*: *extTA,P,t* ⊢ ⟨*Val v*·*compareAndSwap*(*D*·*F*, *Throw a*, *e3*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *CASThrow3*: *extTA,P,t* ⊢ ⟨*Val v*·*compareAndSwap*(*D*·*F*, *Val v′*, *Throw a*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *CallThrowObj*: *extTA,P,t* ⊢ ⟨(*Throw a*)·*M*(*es*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *CallThrowParams*: ⟦ *es* = *map Val vs* @ *Throw a* # *es′* ⟧ ⟹ *extTA,P,t* ⊢ ⟨(*Val v*)·*M*(*es*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *BlockThrow*: *extTA,P,t* ⊢ ⟨{*V*:*T*=*vo*; *Throw a*}, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *SynchronizedThrow1*: *extTA,P,t* ⊢ ⟨*sync*(*Throw a*) *e*, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *SynchronizedThrow2*: *extTA,P,t* ⊢ ⟨*insync*(*a*) *Throw ad*, *s*⟩ −⦃*Unlock*→*a*, *SyncUnlock a*⦄→ ⟨*Throw ad*, *s*⟩
| *SeqThrow*: *extTA,P,t* ⊢ ⟨(*Throw a*);;e_2, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *CondThrow*: *extTA,P,t* ⊢ ⟨*if* (*Throw a*) e_1 *else* e_2, *s*⟩ −ε→ ⟨*Throw a*, *s*⟩
| *ThrowThrow*: *extTA,P,t* ⊢ ⟨*throw*(*Throw a*), *s*⟩ −ε→ ⟨*Throw a*, *s*⟩

inductive-cases *red-cases*:
extTA,P,t ⊢ ⟨*new C*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
extTA,P,t ⊢ ⟨*newA T*⌊*e*⌉, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
extTA,P,t ⊢ ⟨*Cast T e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
extTA,P,t ⊢ ⟨*e instanceof T*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
extTA,P,t ⊢ ⟨*e* «*bop*» *e′*, *s*⟩ −*ta*→ ⟨*e″*, *s′*⟩
extTA,P,t ⊢ ⟨*Var V*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
extTA,P,t ⊢ ⟨*V*:=*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
extTA,P,t ⊢ ⟨*a*⌊*i*⌉, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
extTA,P,t ⊢ ⟨*a*⌊*i*⌉ := *e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
extTA,P,t ⊢ ⟨*a*·*length*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
extTA,P,t ⊢ ⟨*e*·*F*{*D*}, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
extTA,P,t ⊢ ⟨*e*·*F*{*D*} := *e′*, *s*⟩ −*ta*→ ⟨*e″*, *s′*⟩
extTA,P,t ⊢ ⟨*e*·*compareAndSwap*(*D*·*F*, *e′*, *e″*), *s*⟩ −*ta*→ ⟨*e‴*, *s′*⟩
extTA,P,t ⊢ ⟨*e*·*M*(*es*), *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
extTA,P,t ⊢ ⟨{*V*:*T*=*vo*; *e*}, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
extTA,P,t ⊢ ⟨*sync*(*o′*) *e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩

extTA,P,t ⊢ ⟨*insync*(*a*) *e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
extTA,P,t ⊢ ⟨*e*;;*e′*, *s*⟩ −*ta*→ ⟨*e′′*, *s′*⟩
extTA,P,t ⊢ ⟨*if* (*b*) *e1* *else* *e2*, *s* ⟩ −*ta*→ ⟨*e′*, *s′*⟩
extTA,P,t ⊢ ⟨*while* (*b*) *e*, *s* ⟩ −*ta*→ ⟨*e′*, *s′*⟩
extTA,P,t ⊢ ⟨*throw* *e*, *s* ⟩ −*ta*→ ⟨*e′*, *s′*⟩
extTA,P,t ⊢ ⟨*try* *e* *catch*(*C* *V*) *e′*, *s*⟩ −*ta*→ ⟨*e′′*, *s′*⟩

inductive-cases *reds-cases*:
extTA,P,t ⊢ ⟨*e* # *es*, *s*⟩ [−*ta*→] ⟨*es′*, *s′*⟩

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′*⟩
⟨*proof*⟩

lemma *red-no-val* [*dest*]:
⟦ *extTA,P,t* ⊢ ⟨*e*, *s*⟩ −*tas*→ ⟨*e′*, *s′*⟩; *is-val e* ⟧ ⟹ *False*
⟨*proof*⟩

lemma [*iff*]: ¬ *extTA,P,t* ⊢ ⟨*Throw a*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩
⟨*proof*⟩

lemma *reds-map-Val-Throw*:
extTA,P,t ⊢ ⟨*map Val vs* @ *Throw a* # *es*, *s*⟩ [−*ta*→] ⟨*es′*, *s′*⟩ ⟷ *False*
⟨*proof*⟩

lemma *reds-preserves-len*:
extTA,P,t ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es′*, *s′*⟩ ⟹ *length es′* = *length es*
⟨*proof*⟩

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′*)
⟨*proof*⟩

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*⟩

lemma *red-lcl-add*: $extTA,P,t \vdash \langle e, (h, l)\rangle -ta\rightarrow \langle e', (h', l')\rangle \Longrightarrow extTA,P,t \vdash \langle e, (h, l0 ++ l)\rangle -ta\rightarrow \langle e', (h', l0 ++ l')\rangle$
and *reds-lcl-add*: $extTA,P,t \vdash \langle es, (h, l)\rangle [-ta\rightarrow] \langle es', (h', l')\rangle \Longrightarrow extTA,P,t \vdash \langle es, (h, l0 ++ l)\rangle [-ta\rightarrow] \langle es', (h', l0 ++ l')\rangle$
⟨*proof*⟩

lemma *reds-no-val* [*dest*]:
$[\![extTA,P,t \vdash \langle es, s\rangle [-ta\rightarrow] \langle es', s'\rangle; is\text{-}vals\ es]\!] \Longrightarrow False$
⟨*proof*⟩

lemma *red-no-Throw* [*dest*!]:
$extTA,P,t \vdash \langle Throw\ a, s\rangle -ta\rightarrow \langle e', s'\rangle \Longrightarrow False$
⟨*proof*⟩

lemma *red-lcl-sub*:
$[\![extTA,P,t \vdash \langle e, s\rangle -ta\rightarrow \langle e', s'\rangle; fv\ e \subseteq W]\!]$
$\Longrightarrow extTA,P,t \vdash \langle e, (hp\ s, (lcl\ s)|`W)\rangle -ta\rightarrow \langle e', (hp\ s', (lcl\ s')|`W)\rangle$

and *reds-lcl-sub*:
$[\![extTA,P,t \vdash \langle es, s\rangle [-ta\rightarrow] \langle es', s'\rangle; fvs\ es \subseteq W]\!]$
$\Longrightarrow extTA,P,t \vdash \langle es, (hp\ s, (lcl\ s)|`W)\rangle [-ta\rightarrow] \langle es', (hp\ s', (lcl\ s')|`W)\rangle$
⟨*proof*⟩

lemma *red-notfree-unchanged*: $[\![extTA,P,t \vdash \langle e, s\rangle -ta\rightarrow \langle e', s'\rangle; V \notin fv\ e]\!] \Longrightarrow lcl\ s'\ V = lcl\ s\ V$
and *reds-notfree-unchanged*: $[\![extTA,P,t \vdash \langle es, s\rangle [-ta\rightarrow] \langle es', s'\rangle; V \notin fvs\ es]\!] \Longrightarrow lcl\ s'\ V = lcl\ s\ V$
⟨*proof*⟩

lemma *red-dom-lcl*: $extTA,P,t \vdash \langle e, s\rangle -ta\rightarrow \langle e', s'\rangle \Longrightarrow dom\ (lcl\ s') \subseteq dom\ (lcl\ s) \cup fv\ e$
and *reds-dom-lcl*: $extTA,P,t \vdash \langle es, s\rangle [-ta\rightarrow] \langle es', s'\rangle \Longrightarrow dom\ (lcl\ s') \subseteq dom\ (lcl\ s) \cup fvs\ es$
⟨*proof*⟩

lemma *red-Suspend-is-call*:
$[\![convert\text{-}extTA\ extNTA,P,t \vdash \langle e, s\rangle -ta\rightarrow \langle e', s'\rangle; Suspend\ w \in set\ \{\!|ta|\!\}_w]\!]$
$\Longrightarrow \exists\, a\ vs\ hT\ Ts\ Tr\ D.\ call\ e' = \lfloor(a, wait, vs)\rfloor \wedge typeof\text{-}addr\ (hp\ s)\ a = \lfloor hT\rfloor \wedge P \vdash class\text{-}type\text{-}of\ hT\ sees\ wait{:}Ts{\rightarrow}Tr = Native\ in\ D$
and *reds-Suspend-is-calls*:
$[\![convert\text{-}extTA\ extNTA,P,t \vdash \langle es, s\rangle [-ta\rightarrow] \langle es', s'\rangle; Suspend\ w \in set\ \{\!|ta|\!\}_w]\!]$
$\Longrightarrow \exists\, a\ vs\ hT\ Ts\ Tr\ D.\ calls\ es' = \lfloor(a, wait, vs)\rfloor \wedge typeof\text{-}addr\ (hp\ s)\ a = \lfloor hT\rfloor \wedge P \vdash class\text{-}type\text{-}of\ hT\ sees\ wait{:}Ts{\rightarrow}Tr = Native\ in\ D$
⟨*proof*⟩

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*: $extTA,P,t \vdash \langle es, s\rangle [-ta\rightarrow] \langle es', s'\rangle \Longrightarrow hp\ s \trianglelefteq hp\ s'$
⟨*proof*⟩

lemma *red-preserves-tconf*: $[\![extTA,P,t \vdash \langle e, s\rangle -ta\rightarrow \langle e', s'\rangle; P,hp\ s \vdash t \sqrt{t}]\!] \Longrightarrow P,hp\ s' \vdash t \sqrt{t}$
⟨*proof*⟩

lemma *reds-preserves-tconf*: ⟦ *extTA,P,t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es′*, *s′*⟩; *P,hp s* ⊢ *t* √*t* ⟧ ⟹ *P,hp s′* ⊢ *t* √*t*
⟨*proof*⟩

end

4.4.2 Code generation

context *J-heap-base* **begin**

lemma *RedCall-code*:
⟦ *is-vals es*; *typeof-addr* (*hp s*) *a* = ⌊*hU*⌋; *P* ⊢ *class-type-of hU sees M*:*Ts*→*T* = ⌊(*pns*,*body*)⌋ *in D*;
size es = *size pns*; *size Ts* = *size pns* ⟧
⟹ *extTA,P,t* ⊢ ⟨(*addr a*)•*M*(*es*), *s*⟩ −ε→ ⟨*blocks* (*this* # *pns*) (*Class D* # *Ts*) (*Addr a* # *map the-Val es*) *body*, *s*⟩

and *RedCallExternal-code*:
⟦ *is-vals es*; *typeof-addr* (*hp s*) *a* = ⌊*hU*⌋; *P* ⊢ *class-type-of hU sees M*:*Ts*→*T* = *Native in D*;
P,t ⊢ ⟨*a*•*M*(*map the-Val es*), *hp s*⟩ −*ta*→*ext* ⟨*va*, *h′*⟩ ⟧
⟹ *extTA,P,t* ⊢ ⟨(*addr a*)•*M*(*es*), *s*⟩ −*extTA ta*→ ⟨*extRet2J* ((*addr a*)•*M*(*es*)) *va*, (*h′*, *lcl s*)⟩

and *RedCallNull-code*:
is-vals es ⟹ *extTA,P,t* ⊢ ⟨*null*•*M*(*es*), *s*⟩ −ε→ ⟨*THROW NullPointer*, *s*⟩

and *CallThrowParams-code*:
is-Throws es ⟹ *extTA,P,t* ⊢ ⟨(*Val v*)•*M*(*es*), *s*⟩ −ε→ ⟨*hd* (*dropWhile is-val es*), *s*⟩

⟨*proof*⟩

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.AAs
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.RedF
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.RedCASSucc 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: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow (i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool) \Rightarrow (i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool) \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$
and
J-heap-base.reds: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow (i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool) \Rightarrow (i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool) \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$)
[*detect-switches*, *skip-proof*] — proofs are possible, but take veeerry long
J-heap-base.red
⟨*proof*⟩

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*)⟨*proof*⟩
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*

| $WTInstanceOf$:
$[\![$ $is\text{-}lub,P,E \vdash e :: T;\ P \vdash U \leq T \vee P \vdash T \leq U;\ is\text{-}type\ P\ U;\ is\text{-}refT\ U$ $]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash e\ instanceof\ U :: Boolean$

| $WTVal$:
$typeof\ v = Some\ T \Longrightarrow$
$is\text{-}lub,P,E \vdash Val\ v :: T$

| $WTVar$:
$E\ V = Some\ T \Longrightarrow$
$is\text{-}lub,P,E \vdash Var\ V :: T$

| $WTBinOp$:
$[\![$ $is\text{-}lub,P,E \vdash e1 :: T1;\ is\text{-}lub,P,E \vdash e2 :: T2;\ P \vdash T1 «bop» T2 :: T$ $]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash e1 «bop» e2 :: T$

| $WTLAss$:
$[\![$ $E\ V = Some\ T;\ \ is\text{-}lub,P,E \vdash e :: T';\ \ P \vdash T' \leq T;\ \ V \neq this$ $]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash V{:=}e :: Void$

| $WTAAcc$:
$[\![$ $is\text{-}lub,P,E \vdash a :: T\lfloor\rceil;\ is\text{-}lub,P,E \vdash i :: Integer$ $]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash a\lfloor i\rceil :: T$

| $WTAAss$:
$[\![$ $is\text{-}lub,P,E \vdash a :: T\lfloor\rceil;\ is\text{-}lub,P,E \vdash i :: Integer;\ is\text{-}lub,P,E \vdash e :: T';\ P \vdash T' \leq T$ $]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash a\lfloor i\rceil := e :: Void$

| $WTALength$:
$is\text{-}lub,P,E \vdash a :: T\lfloor\rceil \Longrightarrow is\text{-}lub,P,E \vdash a{\cdot}length :: Integer$

| $WTFAcc$:
$[\![$ $is\text{-}lub,P,E \vdash e :: U;\ class\text{-}type\text{-}of'\ U = \lfloor C\rfloor;\ P \vdash C\ sees\ F{:}T\ (fm)\ in\ D$ $]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash e{\cdot}F\{D\} :: T$

| $WTFAss$:
$[\![$ $is\text{-}lub,P,E \vdash e_1 :: U;\ class\text{-}type\text{-}of'\ U = \lfloor C\rfloor;\ P \vdash C\ sees\ F{:}T\ (fm)\ in\ D;\ is\text{-}lub,P,E \vdash e_2 :: T';\ P \vdash T' \leq T$ $]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash e_1{\cdot}F\{D\}{:=}e_2 :: Void$

| $WTCAS$:
$[\![$ $is\text{-}lub,P,E \vdash e1 :: U;\ class\text{-}type\text{-}of'\ U = \lfloor C\rfloor;\ P \vdash C\ sees\ F{:}T\ (fm)\ in\ D;\ volatile\ fm;$
$is\text{-}lub,P,E \vdash e2 :: T';\ P \vdash T' \leq T;\ is\text{-}lub,P,E \vdash e3 :: T'';\ P \vdash T'' \leq T$ $]\!]$
$\Longrightarrow\ \ is\text{-}lub,P,E \vdash e1{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3) :: Boolean$

| $WTCall$:
$[\![$ $is\text{-}lub,P,E \vdash e :: U;\ class\text{-}type\text{-}of'\ U = \lfloor C\rfloor;\ P \vdash C\ sees\ M{:}Ts \rightarrow T = meth\ in\ D;$
$is\text{-}lub,P,E \vdash es\ [::]\ Ts';\ P \vdash Ts'\ [\leq]\ Ts$ $]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash e{\cdot}M(es) :: T$

| $WTBlock$:
$[\![$ $is\text{-}type\ P\ T;\ \ is\text{-}lub,P,E(V \mapsto T) \vdash e :: T';\ case\ vo\ of\ None \Rightarrow True \mid \lfloor v\rfloor \Rightarrow \exists T'.\ typeof\ v = \lfloor T'\rfloor \wedge P \vdash T' \leq T$ $]\!]$
$\Longrightarrow\ \ is\text{-}lub,P,E \vdash \{V{:}T{=}vo;\ e\} :: T'$

| *WTSynchronized*:
$[\![$ *is-lub*,P,E $\vdash$ o' :: T; *is-refT* T; $T \neq NT$; *is-lub*,P,E $\vdash$ e :: T' $]\!]$
$\Longrightarrow$ *is-lub*,P,E $\vdash$ *sync*(o') e :: T'

— Note that insync is not statically typable.

| *WTSeq*:
$[\![$ *is-lub*,P,E $\vdash$ e_1::T_1; *is-lub*,P,E $\vdash$ e_2::T_2 $]\!]$
$\Longrightarrow$ *is-lub*,P,E $\vdash$ e_1;;e_2 :: T_2

| *WTCond*:
$[\![$ *is-lub*,P,E $\vdash$ e :: *Boolean*; *is-lub*,P,E $\vdash$ e_1::T_1; *is-lub*,P,E $\vdash$ e_2::T_2; $\vdash$ *lub*(T_1, T_2) = T $]\!]$
$\Longrightarrow$ *is-lub*,P,E $\vdash$ *if* (e) e_1 *else* e_2 :: T

| *WTWhile*:
$[\![$ *is-lub*,P,E $\vdash$ e :: *Boolean*; *is-lub*,P,E $\vdash$ c::T $]\!]$
$\Longrightarrow$ *is-lub*,P,E $\vdash$ *while* (e) c :: *Void*

| *WTThrow*:
$[\![$ *is-lub*,P,E $\vdash$ e :: *Class* C; P $\vdash$ C $\preceq^*$ *Throwable* $]\!]$ $\Longrightarrow$
is-lub,P,E $\vdash$ *throw* e :: *Void*

| *WTTry*:
$[\![$ *is-lub*,P,E $\vdash$ e_1 :: T; *is-lub*,P,E($V \mapsto$ *Class* C) $\vdash$ e_2 :: T; P $\vdash$ C $\preceq^*$ *Throwable* $]\!]$
$\Longrightarrow$ *is-lub*,P,E $\vdash$ *try* e_1 *catch*(C V) e_2 :: T

| *WTNil*: *is-lub*,P,E $\vdash$ [] [::] []

| *WTCons*: $[\![$ *is-lub*,P,E $\vdash$ e :: T; *is-lub*,P,E $\vdash$ *es* [::] *Ts* $]\!]$ $\Longrightarrow$ *is-lub*,P,E $\vdash$ *e#es* [::] *T#Ts*

abbreviation *WT'* :: *'addr J-prog* $\Rightarrow$ *env* $\Rightarrow$ *'addr expr* $\Rightarrow$ *ty* $\Rightarrow$ *bool* (‹-,- $\vdash$ - :: -› [*51*,*51*,*51*] *50*)
where *WT'* P $\equiv$ *WT* (*TypeRel.is-lub* P) P

abbreviation *WTs'* :: *'addr J-prog* $\Rightarrow$ *env* $\Rightarrow$ *'addr expr list* $\Rightarrow$ *ty list* $\Rightarrow$ *bool* (‹-,- $\vdash$ - [::] -› [*51*,*51*,*51*] *50*)
where *WTs'* P $\equiv$ *WTs* (*TypeRel.is-lub* P) P

declare *WT-WTs.intros*[*intro*!]

inductive-simps *WTs-iffs* [*iff*]:
is-lub',P,E $\vdash$ [] [::] *Ts*
is-lub',P,E $\vdash$ *e#es* [::] *T#Ts*
is-lub',P,E $\vdash$ *e#es* [::] *Ts*

lemma *WTs-conv-list-all2*:
fixes *is-lub*
shows *is-lub*,P,E $\vdash$ *es* [::] *Ts* = *list-all2* (*WT is-lub* P E) *es* *Ts*
⟨*proof*⟩

lemma *WTs-append* [*iff*]: $\bigwedge$*is-lub* *Ts*. (*is-lub*,P,E $\vdash$ es_1 @ es_2 [::] *Ts*) =
($\exists$ Ts_1 Ts_2. *Ts* = Ts_1 @ Ts_2 $\wedge$ *is-lub*,P,E $\vdash$ es_1 [::] Ts_1 $\wedge$ *is-lub*,P,E $\vdash$ es_2[::]Ts_2)
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma fixes *is-lub*
shows *wt-env-mono*: *is-lub*,*P*,*E* ⊢ *e* :: *T* ⟹ (⋀*E'*. *E* $\subseteq_m$ *E'* ⟹ *is-lub*,*P*,*E'* ⊢ *e* :: *T*)
and *wts-env-mono*: *is-lub*,*P*,*E* ⊢ *es* [::] *Ts* ⟹ (⋀*E'*. *E* $\subseteq_m$ *E'* ⟹ *is-lub*,*P*,*E'* ⊢ *es* [::] *Ts*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*′
⟨*proof*⟩

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*⟩

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
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
[*inductify*]
WT-code
⟨*proof*⟩

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
[*inductify*]
WTs-code
⟨*proof*⟩

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*
$\mid \lfloor A \rfloor \Rightarrow$ (*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** ‹⊓› *70*)
where
$A \sqcap B \ \equiv\ $ *case A of None* $\Rightarrow$ *B*
$\mid \lfloor A \rfloor \Rightarrow$ (*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** ‹⊖› *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** ‹∈∈› *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** ‹⊑› *50*)
where
$A \sqsubseteq B \ \equiv\ $ *case B of None* $\Rightarrow$ *True*
$\mid \lfloor B \rfloor \Rightarrow$ (*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*]: $\lfloor \{\} \rfloor \sqcup A = A \ \wedge\ A \sqcup \lfloor \{\} \rfloor = A$⟨*proof*⟩
lemma [*simp*]: $\lfloor A \rfloor \sqcup \lfloor B \rfloor = \lfloor A \cup B \rfloor \wedge \lfloor A \rfloor \ominus a = \lfloor A - \{a\} \rfloor$⟨*proof*⟩
lemma [*simp*]: *None* $\sqcup A =$ *None* $\wedge A \sqcup$ *None* $=$ *None*⟨*proof*⟩
lemma [*simp*]: $a \in\in$ *None* $\wedge$ *None* $\ominus a =$ *None*⟨*proof*⟩
lemma *hyperUn-assoc*: $(A \sqcup B) \sqcup C = A \sqcup (B \sqcup C)$⟨*proof*⟩
lemma *hyper-insert-comm*: $A \sqcup \lfloor \{a\} \rfloor = \lfloor \{a\} \rfloor \sqcup A \wedge A \sqcup (\lfloor \{a\} \rfloor \sqcup B) = \lfloor \{a\} \rfloor \sqcup (A \sqcup B)$⟨*proof*⟩
lemma *sqSub-mem-lem* [*elim*]: $[\![A \sqsubseteq A'; a \in\in A]\!] \Longrightarrow a \in\in A'$
⟨*proof*⟩

lemma [*iff*]: $A \sqsubseteq$ *None*
⟨*proof*⟩

lemma [*simp*]: $A \sqsubseteq A$
⟨*proof*⟩

lemma [*iff*]: $\lfloor A \rfloor \sqsubseteq \lfloor B \rfloor \longleftrightarrow A \subseteq B$
⟨*proof*⟩

lemma *sqUn-lem2*: $A \sqsubseteq A' \Longrightarrow B \sqcup A \sqsubseteq B \sqcup A'$
⟨*proof*⟩

lemma *sqSub-trans* [*trans*, *intro*]: $[\![A \sqsubseteq B; B \sqsubseteq C]\!] \Longrightarrow A \sqsubseteq C$
⟨*proof*⟩

lemma *hyperUn-comm*: $A \sqcup B = B \sqcup A$
⟨*proof*⟩

lemma *hyperUn-leftComm*: $A \sqcup (B \sqcup C) = B \sqcup (A \sqcup C)$
⟨*proof*⟩

lemmas *hyperUn-ac* = *hyperUn-comm hyperUn-leftComm hyperUn-assoc*

lemma [*simp*]: $\lfloor\{\}\rfloor \sqcup B = B$
⟨*proof*⟩

lemma [*simp*]: $\lfloor\{\}\rfloor \sqsubseteq A$
⟨*proof*⟩

lemma *sqInt-lem*: $A \sqsubseteq A' \Longrightarrow A \sqcap B \sqsubseteq A' \sqcap B$
⟨*proof*⟩

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) = $\mathcal{A}\ e_1 \sqcup \mathcal{A}\ e_2$
| $\mathcal{A}$ (*Var V*) = $\lfloor\{\}\rfloor$
| $\mathcal{A}$ (*LAss V e*) = $\lfloor\{V\}\rfloor \sqcup \mathcal{A}\ e$
| $\mathcal{A}$ ($a\lfloor i\rceil$) = $\mathcal{A}\ a \sqcup \mathcal{A}\ i$
| $\mathcal{A}$ ($a\lfloor i\rceil$:= *e*) = $\mathcal{A}\ a \sqcup \mathcal{A}\ i \sqcup \mathcal{A}\ e$
| $\mathcal{A}$ (*a*·*length*) = $\mathcal{A}$ *a*
| $\mathcal{A}$ (*e*·*F*{*D*}) = $\mathcal{A}$ *e*
| $\mathcal{A}$ (e_1·*F*{*D*}:=e_2) = $\mathcal{A}\ e_1 \sqcup \mathcal{A}\ e_2$
| $\mathcal{A}$ (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) = $\mathcal{A}$ *e1* ⊔ $\mathcal{A}$ *e2* ⊔ $\mathcal{A}$ *e3*
| $\mathcal{A}$ (*e*·*M*(*es*)) = $\mathcal{A}\ e \sqcup \mathcal{A}s\ es$
| $\mathcal{A}$ ({*V*:*T*=*vo*; *e*}) = $\mathcal{A}\ e \ominus V$
| $\mathcal{A}$ ($sync_V$ (*o'*) *e*) = $\mathcal{A}\ o' \sqcup \mathcal{A}\ e$
| $\mathcal{A}$ ($insync_V$ (*a*) *e*) = $\mathcal{A}$ *e*
| $\mathcal{A}$ (e_1;;e_2) = $\mathcal{A}\ e_1 \sqcup \mathcal{A}\ e_2$
| $\mathcal{A}$ (*if* (*e*) e_1 *else* e_2) = $\mathcal{A}\ e \sqcup (\mathcal{A}\ e_1 \sqcap \mathcal{A}\ e_2)$
| $\mathcal{A}$ (*while* (*b*) *e*) = $\mathcal{A}$ *b*
| $\mathcal{A}$ (*throw e*) = *None*
| $\mathcal{A}$ (*try* e_1 *catch*(*C V*) e_2) = $\mathcal{A}\ e_1 \sqcap (\mathcal{A}\ e_2 \ominus V)$

| $\mathcal{A}s$ ([]) = $\lfloor\{\}\rfloor$
| $\mathcal{A}s$ (*e*#*es*) = $\mathcal{A}\ e \sqcup \mathcal{A}s\ es$

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* = ($\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \sqcup \mathcal{A}\ e_1)$)
| $\mathcal{D}$ (*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 ∧ $\mathcal{D}$ i (A ⊔ $\mathcal{A}$ a))
| $\mathcal{D}$ $(a\lfloor i\rceil$:= $e)$ A = ($\mathcal{D}$ a A ∧ $\mathcal{D}$ i (A ⊔ $\mathcal{A}$ a) ∧ $\mathcal{D}$ e (A ⊔ $\mathcal{A}$ a ⊔ $\mathcal{A}$ i))
| $\mathcal{D}$ (*a*·*length*) A = $\mathcal{D}$ a A
| $\mathcal{D}$ (*e*·*F*{*D*}) A = $\mathcal{D}$ e A
| $\mathcal{D}$ (e_1·*F*{*D*}:=e_2) A = ($\mathcal{D}$ e_1 A ∧ $\mathcal{D}$ e_2 (A ⊔ $\mathcal{A}$ e_1))
| $\mathcal{D}$ (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) A = ($\mathcal{D}$ *e1* A ∧ $\mathcal{D}$ *e2* (A ⊔ $\mathcal{A}$ *e1*) ∧ $\mathcal{D}$ *e3* (A ⊔ $\mathcal{A}$ *e1* ⊔ $\mathcal{A}$ *e2*))
| $\mathcal{D}$ (*e*·*M*(*es*)) A = ($\mathcal{D}$ e A ∧ $\mathcal{D}s$ *es* (A ⊔ $\mathcal{A}$ e))
| $\mathcal{D}$ ({*V*:*T*=*vo*; *e*}) A = (*if* *vo* = *None* *then* $\mathcal{D}$ e (A ⊖ V) *else* $\mathcal{D}$ e (A ⊔ ⌊{*V*}⌋))
| $\mathcal{D}$ ($sync_V$ (o') e) A = ($\mathcal{D}$ o' A ∧ $\mathcal{D}$ e (A ⊔ $\mathcal{A}$ o'))
| $\mathcal{D}$ ($insync_V$ (a) e) A = $\mathcal{D}$ e A
| $\mathcal{D}$ (e_1;;e_2) A = ($\mathcal{D}$ e_1 A ∧ $\mathcal{D}$ e_2 (A ⊔ $\mathcal{A}$ e_1))
| $\mathcal{D}$ (*if* (e) e_1 *else* e_2) A = ($\mathcal{D}$ e A ∧ $\mathcal{D}$ e_1 (A ⊔ $\mathcal{A}$ e) ∧ $\mathcal{D}$ e_2 (A ⊔ $\mathcal{A}$ e))
| $\mathcal{D}$ (*while* (e) c) A = ($\mathcal{D}$ e A ∧ $\mathcal{D}$ c (A ⊔ $\mathcal{A}$ 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 ∧ $\mathcal{D}$ e_2 (A ⊔ ⌊{*V*}⌋))

| $\mathcal{D}s$ ([]) A = *True*
| $\mathcal{D}s$ (*e*#*es*) A = ($\mathcal{D}$ e A ∧ $\mathcal{D}s$ *es* (A ⊔ $\mathcal{A}$ e))

lemma *As-map-Val*[*simp*]: $\mathcal{A}s$ (*map* *Val* *vs*) = ⌊{}⌋⟨*proof*⟩
lemma *As-append* [*simp*]: $\mathcal{A}s$ (*xs* @ *ys*) = ($\mathcal{A}s$ *xs*) ⊔ ($\mathcal{A}s$ *ys*)
⟨*proof*⟩

lemma *Ds-map-Val*[*simp*]: $\mathcal{D}s$ (*map* *Val* *vs*) A⟨*proof*⟩
lemma *D-append*[*iff*]: ⋀*A*. $\mathcal{D}s$ (*es* @ *es'*) A = ($\mathcal{D}s$ *es* A ∧ $\mathcal{D}s$ *es'* (A ⊔ $\mathcal{A}s$ *es*))⟨*proof*⟩

lemma fixes e :: ('a,'b,'addr) *exp* **and** *es* :: ('a,'b,'addr) *exp list*
  shows *A-fv*: ⋀*A*. $\mathcal{A}$ e = ⌊A⌋ ⟹ A ⊆ *fv* e
  and ⋀*A*. $\mathcal{A}s$ *es* = ⌊A⌋ ⟹ A ⊆ *fvs* *es*
⟨*proof*⟩

lemma *sqUn-lem*: A ⊑ A' ⟹ A ⊔ B ⊑ A' ⊔ B⟨*proof*⟩
lemma *diff-lem*: A ⊑ A' ⟹ A ⊖ b ⊑ A' ⊖ b⟨*proof*⟩

lemma fixes e :: ('a, 'b, 'addr) *exp* **and** *es* :: ('a, 'b, 'addr) *exp list*
  shows *D-mono*: ⋀*A A'*. A ⊑ A' ⟹ $\mathcal{D}$ e A ⟹ $\mathcal{D}$ e A'
  and *Ds-mono*: ⋀*A A'*. A ⊑ A' ⟹ $\mathcal{D}s$ *es* A ⟹ $\mathcal{D}s$ *es* A'⟨*proof*⟩

lemma *D-mono'*: $\mathcal{D}$ e A ⟹ A ⊑ A' ⟹ $\mathcal{D}$ e A'
and *Ds-mono'*: $\mathcal{D}s$ *es* A ⟹ A ⊑ A' ⟹ $\mathcal{D}s$ *es* A'⟨*proof*⟩
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*) ∧
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*) ∧
𝒟 *body* ⌊{*this*} ∪ *set pns*⌋)⟨*proof*⟩

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*⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*ML*⟩

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*
⟨*proof*⟩

end

abbreviation *final-expr* :: *'addr expr* × *'addr locals* ⇒ *bool***where**
final-expr ≡ λ(*e*, *x*). *final e*

lemma *final-locks*: *final e* ⟹ *expr-locks e l* = *0*
⟨*proof*⟩

context *J-heap-base* **begin**

abbreviation *mred*
:: *'addr J-prog* ⇒ (*'addr*, *'thread-id*, *'addr expr* × *'addr locals*, *'heap*, *'addr*, (*'addr*, *'thread-id*) *obs-event*) *semantics*
where
mred P t ≡ (λ((*e*, *l*), *h*) *ta* ((*e'*, *l'*), *h'*). *P*,*t* ⊢ ⟨*e*, (*h*, *l*)⟩ −*ta*→ ⟨*e'*, (*h'*, *l'*)⟩)

lemma *red-new-thread-heap*:
⟦ *convert-extTA extNTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; *NewThread t'' ex'' h''* ∈ *set* $\{\!|ta|\!\}_t$ ⟧ ⟹ *h''* = *hp s'*
and *reds-new-thread-heap*:
⟦ *convert-extTA extNTA*,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *NewThread t'' ex'' h''* ∈ *set* $\{\!|ta|\!\}_t$ ⟧ ⟹ *h''* = *hp s'*
⟨*proof*⟩

lemma *red-ta-Wakeup-no-Join-no-Lock-no-Interrupt*:
⟦ *convert-extTA extNTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*, *s'*⟩; *Notified* ∈ *set* $\{\!|ta|\!\}_w$ ∨ *WokenUp* ∈ *set* $\{\!|ta|\!\}_w$ ⟧
⟹ *collect-locks* $\{\!|ta|\!\}_l$ = {} ∧ *collect-cond-actions* $\{\!|ta|\!\}_c$ = {} ∧ *collect-interrupts* $\{\!|ta|\!\}_i$ = {}
and *reds-ta-Wakeup-no-Join-no-Lock-no-Interrupt*:
⟦ *convert-extTA extNTA*,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *Notified* ∈ *set* $\{\!|ta|\!\}_w$ ∨ *WokenUp* ∈ *set* $\{\!|ta|\!\}_w$ ⟧
⟹ *collect-locks* $\{\!|ta|\!\}_l$ = {} ∧ *collect-cond-actions* $\{\!|ta|\!\}_c$ = {} ∧ *collect-interrupts* $\{\!|ta|\!\}_i$ = {}
⟨*proof*⟩

lemma *final-no-red*:
final e ⟹ ¬ *P*,*t* ⊢ ⟨*e*, (*h*, *l*)⟩ −*ta*→ ⟨*e'*, (*h'*, *l'*)⟩
⟨*proof*⟩

lemma *red-mthr*: *multithreaded final-expr* (*mred P*)
⟨*proof*⟩

end

sublocale *J-heap-base* < *red-mthr*: *multithreaded*
final-expr
mred P
convert-RA
for *P*
⟨*proof*⟩

context *J-heap-base* **begin**

abbreviation
mredT ::
'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*
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′
⟨proof⟩

lemma RedT-hext-incr:
assumes P ⊢ s −▷tta→∗ s′
shows shr s ⊴ shr s′
⟨proof⟩

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
⟨proof⟩

lemma lifting-wf-tconf:
lifting-wf final-expr (mred P) (λt ex h. P,h ⊢ t √t)
⟨proof⟩

end

sublocale J-heap < red-tconf: lifting-wf final-expr mred P convert-RA λt ex h. P,h ⊢ t √t
⟨proof⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *expr-locks-sync-ok*: (⋀*ad*. *expr-locks e ad* = *0*) ⟹ *sync-ok e*
  and *expr-lockss-sync-oks*: (⋀*ad*. *expr-lockss es ad* = *0*) ⟹ *sync-oks es*
⟨*proof*⟩

lemma *sync-ok-extRet2J* [*simp*, *intro*!]: *sync-ok e* ⟹ *sync-ok* (*extRet2J e va*)
⟨*proof*⟩

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* ⟧ ⟹ *sync-ok* (*blocks pns Ts vs e*) = *sync-ok e*
⟨*proof*⟩

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* ⟧ ⟹ *sync-ok e'*
and *reds-preserve-sync-oks*: ⟦ *extTA*,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es'*, *s'*⟩; *sync-oks es* ⟧ ⟹ *sync-oks es'*
⟨*proof*⟩

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*⦄$_t$ ⟧ ⟹ *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*⦄$_t$ ⟧ ⟹ *expr-locks e''* = (λ*ad*. *0*)
⟨*proof*⟩

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''*
⟨*proof*⟩

lemma *lifting-wf-sync-ok*: *wf-J-prog P* ⟹ *lifting-wf final-expr* (*mred P*) (λ*t* (*e*, *x*) *m*. *sync-ok e*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

lemma *lock-okD1*:
⟦ lock-ok ls ts; ts t = None ⟧ ⟹ ∀ l. has-locks (ls $ l) t = 0
⟨*proof*⟩

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
⟨*proof*⟩

lemma *lock-ok-lock-thread-ok*:
assumes *lock*: *lock-ok ls ts*
shows *lock-thread-ok ls ts*
⟨*proof*⟩

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))
⟨*proof*⟩

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'*
⟨*proof*⟩

definition *upd-expr-locks* :: ('l ⇒ *int*) ⇒ 'l *lock-actions* ⇒ 'l ⇒ *int*
where *upd-expr-locks els las* ≡ λl. *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*)
⟨*proof*⟩

lemma *upd-expr-lock-action-add* [*simp*]:
upd-expr-lock-action $(l + l')$ *L* = *upd-expr-lock-action l L* + l'
⟨*proof*⟩

lemma *upd-expr-lock-actions-add* [*simp*]:
upd-expr-lock-actions $(l + l')$ *Ls* = *upd-expr-lock-actions l Ls* + l'
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *expr-locks-extRet2J* [*simp*, *intro*!]: *expr-locks e* = (λ*ad*. *0*) ⟹ *expr-locks* (*extRet2J e va*) = (λ*ad*. *0*)
⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *ls-els-okD*:

$\bigwedge ln.$ *ls-els-ok ls t ln els* $\Longrightarrow$ *lock-expr-locks-ok* (*ls* \$ *l*) *t* (*ln* \$ *l*) (*els l*)
⟨*proof*⟩

lemma *redT-updLs-upd-expr-locks-preserves-ls-els-ok*:
$\bigwedge ln.$ ⟦ *ls-els-ok ls t ln els*; *lock-ok-las ls t las* ⟧
$\Longrightarrow$ *ls-els-ok* (*redT-updLs ls t las*) *t* (*redT-updLns ls t ln las*) (*upd-expr-locks els las*)
⟨*proof*⟩

lemma *sync-ok-redT-updT*:
assumes *sync-es-ok ts h*
and *nt*: $\bigwedge t\ e\ x\ h''$. *ta* = *NewThread t* (*e*, *x*) h'' $\Longrightarrow$ *sync-ok e*
shows *sync-es-ok* (*redT-updT ts ta*) h'
⟨*proof*⟩

lemma *sync-ok-redT-updTs*:
⟦ *sync-es-ok ts h*; $\bigwedge t\ e\ x\ h$. *NewThread t* (*e*, *x*) *h* ∈ *set tas* $\Longrightarrow$ *sync-ok e* ⟧
$\Longrightarrow$ *sync-es-ok* (*redT-updTs ts tas*) h'
⟨*proof*⟩

lemma *lock-ok-thr-updI*:
$\bigwedge ln.$ ⟦ *lock-ok ls ts*; *ts t* = ⌊((*e*, *xs*), *ln*)⌋; *expr-locks e* = *expr-locks* e' ⟧
$\Longrightarrow$ *lock-ok ls* (*ts*(*t* ↦ ((e', xs'), *ln*)))
⟨*proof*⟩

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*⟩

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*)}
⟨*proof*⟩

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')
⟨*proof*⟩

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*⟩

lemma *red-mthr-deterministic*:
assumes *det*: *deterministic-heap-ops*
shows *red-mthr.deterministic P UNIV*
⟨*proof*⟩

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*:

$typeof_h\ v = Some\ T \Longrightarrow WTrt\ P\ h\ E\ (Val\ v)\ T$

| $WTrtVar$:
$E\ V = Some\ T \Longrightarrow WTrt\ P\ h\ E\ (Var\ V)\ T$

| $WTrtBinOp$:
$[\![\ WTrt\ P\ h\ E\ e1\ T1;\ WTrt\ P\ h\ E\ e2\ T2;\ P \vdash T1 «bop» T2 : T\ ]\!]$
$\Longrightarrow WTrt\ P\ h\ E\ (e1\ «bop»\ e2)\ T$

| $WTrtLAss$:
$[\![\ E\ V = Some\ T;\ WTrt\ P\ h\ E\ e\ T';\ P \vdash T' \leq T\ ]\!]$
$\Longrightarrow WTrt\ P\ h\ E\ (V{:=}e)\ Void$

| $WTrtAAcc$:
$[\![\ WTrt\ P\ h\ E\ a\ (T\lfloor\rceil);\ WTrt\ P\ h\ E\ i\ Integer\ ]\!]$
$\Longrightarrow WTrt\ P\ h\ E\ (a\lfloor i\rceil)\ T$

| $WTrtAAccNT$:
$[\![\ WTrt\ P\ h\ E\ a\ NT;\ WTrt\ P\ h\ E\ i\ Integer\ ]\!]$
$\Longrightarrow WTrt\ P\ h\ E\ (a\lfloor i\rceil)\ T$

| $WTrtAAss$:
$[\![\ WTrt\ P\ h\ E\ a\ (T\lfloor\rceil);\ WTrt\ P\ h\ E\ i\ Integer;\ WTrt\ P\ h\ E\ e\ T'\ ]\!]$
$\Longrightarrow WTrt\ P\ h\ E\ (a\lfloor i\rceil := e)\ Void$

| $WTrtAAssNT$:
$[\![\ WTrt\ P\ h\ E\ a\ NT;\ WTrt\ P\ h\ E\ i\ Integer;\ WTrt\ P\ h\ E\ e\ T'\ ]\!]$
$\Longrightarrow WTrt\ P\ h\ E\ (a\lfloor i\rceil := e)\ Void$

| $WTrtALength$:
$WTrt\ P\ h\ E\ a\ (T\lfloor\rceil) \Longrightarrow WTrt\ P\ h\ E\ (a{\cdot}length)\ Integer$

| $WTrtALengthNT$:
$WTrt\ P\ h\ E\ a\ NT \Longrightarrow WTrt\ P\ h\ E\ (a{\cdot}length)\ T$

| $WTrtFAcc$:
$[\![\ WTrt\ P\ h\ E\ e\ U;\ class\text{-}type\text{-}of'\ U = \lfloor C\rfloor;\ P \vdash C\ has\ F{:}T\ (fm)\ in\ D\ ]\!] \Longrightarrow$
$WTrt\ P\ h\ E\ (e{\cdot}F\{D\})\ T$

| $WTrtFAccNT$:
$WTrt\ P\ h\ E\ e\ NT \Longrightarrow WTrt\ P\ h\ E\ (e{\cdot}F\{D\})\ T$

| $WTrtFAss$:
$[\![\ WTrt\ P\ h\ E\ e1\ U;\ class\text{-}type\text{-}of'\ U = \lfloor C\rfloor;\ P \vdash C\ has\ F{:}T\ (fm)\ in\ D;\ WTrt\ P\ h\ E\ e2\ T2;\ P \vdash T2 \leq T\ ]\!]$
$\Longrightarrow WTrt\ P\ h\ E\ (e1{\cdot}F\{D\}{:=}e2)\ Void$

| $WTrtFAssNT$:
$[\![\ WTrt\ P\ h\ E\ e1\ NT;\ WTrt\ P\ h\ E\ e2\ T2\ ]\!]$
$\Longrightarrow WTrt\ P\ h\ E\ (e1{\cdot}F\{D\}{:=}e2)\ Void$

| $WTrtCAS$:
$[\![\ WTrt\ P\ h\ E\ e1\ U;\ class\text{-}type\text{-}of'\ U = \lfloor C\rfloor;\ P \vdash C\ has\ F{:}T\ (fm)\ in\ D;\ volatile\ fm;$
$WTrt\ P\ h\ E\ e2\ T2;\ P \vdash T2 \leq T;\ WTrt\ P\ h\ E\ e3\ T3;\ P \vdash T3 \leq T\ ]\!]$

$\Longrightarrow$ *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* ⟧
$\Longrightarrow$ *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* ⟧
$\Longrightarrow$ *WTrt P h E* (*e*·*M*(*es*)) *T*

| *WTrtCallNT*:
⟦ *WTrt P h E e NT*; *WTrts P h E es Ts* ⟧
$\Longrightarrow$ *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* ⟧
$\Longrightarrow$ *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′* ⟧
$\Longrightarrow$ *WTrt P h E* (*sync*(*o′*) *e*) *T′*

| *WTrtInSynchronized*:
⟦ *WTrt P h E* (*addr a*) *T*; *WTrt P h E e T′* ⟧
$\Longrightarrow$ *WTrt P h E* (*insync*(*a*) *e*) *T′*

| *WTrtSeq*:
⟦ *WTrt P h E e1 T1*; *WTrt P h E e2 T2* ⟧
$\Longrightarrow$ *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* ⟧
$\Longrightarrow$ *WTrt P h E* (*if* (*e*) *e1 else e2*) *T*

| *WTrtWhile*:
⟦ *WTrt P h E e Boolean*; *WTrt P h E c T* ⟧
$\Longrightarrow$ *WTrt P h E* (*while*(*e*) *c*) *Void*

| *WTrtThrow*:
⟦ *WTrt P h E e T*; *P* ⊢ *T* ≤ *Class Throwable* ⟧
$\Longrightarrow$ *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* ⟧
$\Longrightarrow$ *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* ⟧ $\Longrightarrow$ *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
$P,E,h \vdash e : T \equiv WTrt\ P\ h\ E\ e\ T$

abbreviation
WTrts-syntax :: *'addr J-prog* ⇒ *env* ⇒ *'heap* ⇒ *'addr expr list* ⇒ *ty list* ⇒ *bool* (‹-,-,- ⊢ - [:] -› [*51*,*51*,*51*]*50*)
where
$P,E,h \vdash es\ [:]\ Ts \equiv WTrts\ P\ h\ E\ es\ Ts$

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*: $P,E,h \vdash es\ [:]\ Ts = list\text{-}all2\ (WTrt\ P\ h\ E)\ es\ Ts$
⟨*proof*⟩

lemma [*simp*]: $(P,E,h \vdash es_1\ @\ es_2\ [:]\ Ts) =$
$(\exists\ Ts_1\ Ts_2.\ Ts = Ts_1\ @\ Ts_2 \wedge P,E,h \vdash es_1\ [:]\ Ts_1\ \&\ P,E,h \vdash es_2[:]Ts_2)$
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*[↦]*Ts*),$h \vdash e{:}T \wedge (\exists$ *Ts'*. *map* (*typeof*$_h$) *vs* = *map Some Ts'* $\wedge P \vdash$ *Ts'* [≤] *Ts*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*]:
⟦ $P,E,h \vdash e : T$; *class-type-of'* T = ⌊U⌋; *final e*;
$\bigwedge a$. e = *addr a* $\Longrightarrow R$;
$\bigwedge a$. e = *Throw a* $\Longrightarrow R$ ⟧ $\Longrightarrow R$

⟨*proof*⟩

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*
⟨*proof*⟩

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*; 𝒟*s es* ⌊*dom l*⌋; ¬ *finals es* ⟧ ⟹ ∃ *es′ s′ ta*. *extTA*,*P*,*t* ⊢ ⟨*es*,(*h*,*l*)⟩ [−*ta*→] ⟨*es′*,*s′*⟩
⟨*proof*⟩

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*!]: 𝒟 *e A* ⟹ 𝒟 (*extRet2J e va*) *A*
⟨*proof*⟩

lemma *blocks-defass* [*iff*]: ⋀*A*. ⟦ *length Vs* = *length Ts*; *length vs* = *length Ts*⟧ ⟹
𝒟 (*blocks Vs Ts vs e*) *A* = 𝒟 *e* (*A* ⊔ ⌊*set Vs*⌋)⟨*proof*⟩
context *J-heap-base* **begin**

lemma *red-lA-incr*: *extTA*,*P*,*t* ⊢ ⟨*e*,*s*⟩ −*ta*→ ⟨*e′*,*s′*⟩ ⟹ ⌊*dom* (*lcl s*)⌋ ⊔ 𝒜 *e* ⊑ ⌊*dom* (*lcl s′*)⌋ ⊔ 𝒜 *e′*
and *reds-lA-incr*: *extTA*,*P*,*t* ⊢ ⟨*es*,*s*⟩ [−*ta*→] ⟨*es′*,*s′*⟩ ⟹ ⌊*dom* (*lcl s*)⌋ ⊔ 𝒜*s es* ⊑ ⌊*dom* (*lcl s′*)⌋ ⊔ 𝒜*s es′*
⟨*proof*⟩

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*: $extTA,P,t \vdash \langle e,s\rangle \ -ta\rightarrow \langle e',s'\rangle \Longrightarrow \mathcal{D}\ e\ \lfloor dom\ (lcl\ s)\rfloor \Longrightarrow \mathcal{D}\ e'\ \lfloor dom\ (lcl\ s')\rfloor$
and *reds-preserves-defass*: $extTA,P,t \vdash \langle es,s\rangle\ [-ta\rightarrow]\ \langle es',s'\rangle \Longrightarrow \mathcal{D}s\ es\ \lfloor dom\ (lcl\ s)\rfloor \Longrightarrow \mathcal{D}s\ es'\ \lfloor dom\ (lcl\ s')\rfloor$
⟨*proof*⟩

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*:
$[\![\ extTA,P,t \vdash \langle e,s\rangle\ -ta\rightarrow \langle e',s'\rangle;\ P,E,hp\ s \vdash e : T;\ hconf\ (hp\ s)\ ]\!] \Longrightarrow hconf\ (hp\ s')$
and *reds-preserves-hconf*:
$[\![\ extTA,P,t \vdash \langle es,s\rangle\ [-ta\rightarrow]\ \langle es',s'\rangle;\ P,E,hp\ s \vdash es\ [:]\ Ts;\ hconf\ (hp\ s)\ ]\!] \Longrightarrow hconf\ (hp\ s')$
⟨*proof*⟩

theorem (**in** *J-heap*) *red-preserves-lconf*:
$[\![\ extTA,P,t \vdash \langle e,s\rangle\ -ta\rightarrow \langle e',s'\rangle;\ P,E,hp\ s \vdash e{:}T;\ P,hp\ s \vdash lcl\ s\ (:\leq)\ E\ ]\!] \Longrightarrow P,hp\ s' \vdash lcl\ s'\ (:\leq)\ E$
and *reds-preserves-lconf*:
$[\![\ extTA,P,t \vdash \langle es,s\rangle\ [-ta\rightarrow]\ \langle es',s'\rangle;\ P,E,hp\ s \vdash es[:]Ts;\ P,hp\ s \vdash lcl\ s\ (:\leq)\ E\ ]\!] \Longrightarrow P,hp\ s' \vdash lcl\ s'\ (:\leq)\ E$
⟨*proof*⟩

Combining conformance of heap and local variables:

definition (**in** *J-heap-conf-base*) *sconf* :: $env \Rightarrow ('addr, 'heap)\ Jstate \Rightarrow bool$ (‹- ⊢ - √› [51,51]50)
where $E \vdash s\ \surd\ \equiv\ let\ (h,l) = s\ in\ hconf\ h \wedge P,h \vdash l\ (:\leq)\ E \wedge preallocated\ h$

context *J-conf-read* **begin**

lemma *red-preserves-sconf*:
$[\![\ extTA,P,t \vdash \langle e,s\rangle\ -tas\rightarrow \langle e',s'\rangle;\ P,E,hp\ s \vdash e : T;\ E \vdash s\ \surd\ ]\!] \Longrightarrow E \vdash s'\ \surd$
⟨*proof*⟩

lemma *reds-preserves-sconf*:
$[\![\ extTA,P,t \vdash \langle es,s\rangle\ [-ta\rightarrow]\ \langle es',s'\rangle;\ P,E,hp\ s \vdash es\ [:]\ Ts;\ E \vdash s\ \surd\ ]\!] \Longrightarrow E \vdash s'\ \surd$
⟨*proof*⟩

end

lemma (**in** *J-heap-base*) *wt-external-call*:
$[\![\ conf\text{-}extRet\ P\ h\ va\ T;\ P,E,h \vdash e : T\ ]\!] \Longrightarrow \exists T'.\ P,E,h \vdash extRet2J\ e\ va : T' \wedge P \vdash T' \leq T$

⟨*proof*⟩

4.13.2 Subject reduction

theorem (**in** *J-conf-read*) **assumes** *wf*: *wf-J-prog P*
shows *subject-reduction*:
$[\![\ extTA,P,t \vdash \langle e,s\rangle\ -ta\rightarrow\ \langle e',s'\rangle;\ E \vdash s\ \surd;\ P,E,hp\ s \vdash e{:}T;\ P,hp\ s \vdash t\ \surd t\ ]\!]$
$\Longrightarrow \exists\, T'.\ P,E,hp\ s' \vdash e'{:}T' \wedge P \vdash T' \leq T$
and *subjects-reduction*:
$[\![\ extTA,P,t \vdash \langle es,s\rangle\ [-ta\rightarrow]\ \langle es',s'\rangle;\ E \vdash s\ \surd;\ P,E,hp\ s \vdash es[:]Ts;\ P,hp\ s \vdash t\ \surd t\ ]\!]$
$\Longrightarrow \exists\, Ts'.\ P,E,hp\ s' \vdash es'[:]Ts' \wedge P \vdash Ts'\ [\leq]\ Ts$
⟨*proof*⟩

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 $\exists\, e\ x\ ln.\ ts\ t = \lfloor((e,\ x),\ ln)\rfloor \wedge \neg\ final\ e$
⟨*proof*⟩

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
$def\text{-}ass\text{-}ts\text{-}ok \equiv ts\text{-}ok\ (\lambda t\ (e,\ x)\ h.\ \mathcal{D}\ e\ \lfloor dom\ x\rfloor)$

context *J-heap-base* **begin**

lemma assumes *wf*: *wf-J-prog P*
shows *red-def-ass-new-thread*:
$[\![\ P,t \vdash \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle;\ NewThread\ t''\ (e'',\ x'')\ c'' \in set\ \{\!|ta|\!\}_t\ ]\!] \Longrightarrow \mathcal{D}\ e''\ \lfloor dom\ x''\rfloor$

and *reds-def-ass-new-thread*:
$[\![\ P,t \vdash \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle;\ NewThread\ t''\ (e'',\ x'')\ c'' \in set\ \{\!|ta|\!\}_t\ ]\!] \Longrightarrow \mathcal{D}\ e''\ \lfloor dom\ x''\rfloor$
⟨*proof*⟩

lemma *lifting-wf-def-ass*: $wf\text{-}J\text{-}prog\ P \Longrightarrow lifting\text{-}wf\ final\text{-}expr\ (mred\ P)\ (\lambda t\ (e,\ x)\ m.\ \mathcal{D}\ e\ \lfloor dom\ x\rfloor)$
⟨*proof*⟩

lemma *def-ass-ts-ok-J-start-state*:

$\llbracket$ *wf-J-prog P*; $P \vdash C$ *sees* $M{:}Ts{\rightarrow}T = \lfloor(pns,\ body)\rfloor$ *in D*; *length vs* = *length Ts* $\rrbracket \Longrightarrow$
def-ass-ts-ok (*thr* (*J-start-state P C M vs*)) *h*
⟨*proof*⟩

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* $P \equiv (\lambda(E,\ T)\ e\ c.\ (\exists\ T'.\ (P,E,c \vdash e : T' \wedge P \vdash T' \leq T)))$

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*:
$\llbracket$ $P,t \vdash \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$; $P,E,hp\ s \vdash e : T$; *NewThread* $t''\ (e'',\ x'')\ (hp\ s') \in set\ \{\!|ta|\!\}_t$ $\rrbracket$
$\Longrightarrow \exists\,E\ T.\ P,E,hp\ s' \vdash e'' : T \wedge P,hp\ s' \vdash x''\ (:\leq)\ E$
and *reds-type-newthread*:
$\llbracket$ $P,t \vdash \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle$; *NewThread* $t''\ (e'',\ x'')\ (hp\ s') \in set\ \{\!|ta|\!\}_t$; $P,E,hp\ s \vdash es\ [:]\ Ts$ $\rrbracket$
$\Longrightarrow \exists\,E\ T.\ P,E,hp\ s' \vdash e'' : T \wedge P,hp\ s' \vdash x''\ (:\leq)\ E$
⟨*proof*⟩

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*) $\surd$ ∧ *type-ok P ET* (*fst ex*) *h* ∧ $P,h \vdash t\ \surd t$

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*)
⟨*proof*⟩

end

lemma (**in** *J-heap*) *red-preserve-welltype*:
$\llbracket$ $extTA,P,t \vdash \langle e,\ (h,\ x)\rangle\ -ta\rightarrow\ \langle e',\ (h',\ x')\rangle$; $P,E,h \vdash e'' : T$ $\rrbracket \Longrightarrow P,E,h' \vdash e'' : T$
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

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'*)
⟨*proof*⟩

lemma *lifting-inv-sconf-subject-ok*:
assumes *wf*: *wf-J-prog P*
shows *lifting-inv final-expr* (*mred P*) *sconf-type-ok*
⟨*proof*⟩

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* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e'*,*s'*⟩; ¬ *red-mthr.actions-ok'* (*ls*, (*ts*, *m*), *ws*, *is*) *t ta*;
sync-ok e; *hconf* (*hp s*); *P*,*hp s* ⊢ *t* √*t*;
∀ *l*. *has-locks* (*ls* $ *l*) *t* ≥ *expr-locks e l*;
ws t = *None* ∨
(∃ *a vs w T Ts Tr D*. *call e* = ⌊(*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*⌋) ⟧
⟹ ∃ *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*;

$ws\ t = None \lor$
$(\exists\, a\ vs\ w\ T\ Ts\ T\ Tr\ D.\ calls\ es = \lfloor(a,\ wait,\ vs)\rfloor \land typeof\text{-}addr\ (hp\ s)\ a = \lfloor T\rfloor \land P \vdash class\text{-}type\text{-}of$
$T\ sees\ wait\colon Ts{\rightarrow}Tr = Native\ in\ D \land ws\ t = \lfloor PostWS\ w\rfloor)\ ]\!]$
$\Longrightarrow \exists\, es''\ s''\ ta'.\ P,t \vdash \langle es,\ s\rangle\ [-ta'\rightarrow]\ \langle es'',s''\rangle \land$
$(red\text{-}mthr.actions\text{-}ok\ (ls,\ (ts,\ m),\ ws,\ is)\ t\ ta' \lor$
$red\text{-}mthr.actions\text{-}ok'\ (ls,\ (ts,\ m),\ ws,\ is)\ t\ ta' \land red\text{-}mthr.actions\text{-}subset\ ta'\ ta)$
⟨*proof*⟩

end

end

context *J-heap-base* **begin**

lemma shows *red-ta-satisfiable*:
$P,t \vdash \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle \Longrightarrow \exists\, s.\ red\text{-}mthr.actions\text{-}ok\ s\ t\ ta$
and *reds-ta-satisfiable*:
$P,t \vdash \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle \Longrightarrow \exists\, s.\ red\text{-}mthr.actions\text{-}ok\ s\ t\ ta$
⟨*proof*⟩

end

context *J-typesafe* **begin**

lemma *wf-progress*:
assumes *wf*: *wf-J-prog P*
shows *progress final-expr* (*mred P*)
$(red\text{-}mthr.wset\text{-}Suspend\text{-}ok\ P\ (\{s.\ sync\text{-}es\text{-}ok\ (thr\ s)\ (shr\ s) \land lock\text{-}ok\ (locks\ s)\ (thr\ s)\} \cap \{s.$
$\exists\, Es.\ sconf\text{-}type\text{-}ts\text{-}ok\ Es\ (thr\ s)\ (shr\ s)\} \cap \{s.\ def\text{-}ass\text{-}ts\text{-}ok\ (thr\ s)\ (shr\ s)\}))$
(**is** *progress - - ?wf-state*)
⟨*proof*⟩

lemma *redT-progress-deadlock*:
assumes *wf*: *wf-J-prog P*
and *wf-start*: *wf-start-state P C M vs*
and *Red*: $P \vdash J\text{-}start\text{-}state\ P\ C\ M\ vs\ {-}{\triangleright}ttas{\rightarrow}{*}\ s$
and *ndead*: $\neg\ red\text{-}mthr.deadlock\ P\ s$
shows $\exists\, t'\ ta'\ s'.\ P \vdash s\ -t'{\triangleright}ta'\rightarrow\ s'$
⟨*proof*⟩

lemma *redT-progress-deadlocked*:
assumes *wf*: *wf-J-prog P*
and *wf-start*: *wf-start-state P C M vs*
and *Red*: $P \vdash J\text{-}start\text{-}state\ P\ C\ M\ vs\ {-}{\triangleright}ttas{\rightarrow}{*}\ s$
and *ndead*: $red\text{-}mthr.not\text{-}final\text{-}thread\ s\ t\ \neg\ t \in red\text{-}mthr.deadlocked\ P\ s$
shows $\exists\, t'\ ta'\ s'.\ P \vdash s\ -t'{\triangleright}ta'\rightarrow\ s'$
⟨*proof*⟩

4.14.5 Type safety proof

theorem *TypeSafetyT*:
fixes *C* **and** *M* **and** *ttas* **and** *Es*
defines $Es\ == J\text{-}sconf\text{-}type\text{-}ET\text{-}start\ P\ C\ M$
and $Es' == upd\text{-}invs\ Es\ sconf\text{-}type\text{-}ok\ (concat\ (map\ (thr\text{-}a \circ snd)\ ttas))$

assumes *wf*: *wf-J-prog P*
and *start-wf*: *wf-start-state P C M vs*
and *RedT*: $P \vdash$ *J-start-state P C M vs* $-\triangleright ttas \rightarrow *$ s'
and *nored*: $\neg$ ($\exists$ *t ta* s''. $P \vdash s'$ $-t\triangleright ta\rightarrow$ s'')
shows *thread-conf P* (*thr* s') (*shr* s')
and *thr* s' $t = \lfloor((e', x'), ln')\rfloor \Longrightarrow$
($\exists v$. e' = *Val v* $\wedge$ ($\exists E$ T. Es' $t = \lfloor(E, T)\rfloor \wedge P$,*shr* $s' \vdash v :\leq T$) $\wedge$ ln' = *no-wait-locks*)
$\vee$ ($\exists$ *a C*. e' = *Throw a* $\wedge$ *typeof-addr* (*shr* s') $a = \lfloor$*Class-type C*$\rfloor \wedge P \vdash C \preceq^*$ *Throwable* $\wedge$ ln' = *no-wait-locks*)
$\vee$ ($t \in$ *red-mthr.deadlocked P* s' $\wedge$ ($\exists E$ T. Es' $t = \lfloor(E, T)\rfloor \wedge$ ($\exists T'$. P,E,*shr* $s' \vdash e' : T' \wedge P \vdash T' \leq T$)))
(**is** - $\Longrightarrow$ *?thesis2*)
and $Es \subseteq_m Es'$
⟨*proof*⟩

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,$H \vdash t \sqrt{t}$
shows ⟦ *convert-extTA extNTA*,P,$t \vdash \langle e, s\rangle$ $-ta\rightarrow$ $\langle e', s'\rangle$; P,E,$H \vdash e : T$; *hext H* (*hp s*) ⟧
$\Longrightarrow \exists$ ta' e' s'. *convert-extTA extNTA*,P,$t \vdash \langle e, (H, \textit{lcl } s)\rangle$ $-ta'\rightarrow$ $\langle e', s'\rangle$ $\wedge$
collect-locks $\{|ta|\}_l$ = *collect-locks* $\{|ta'|\}_l$ $\wedge$ *collect-cond-actions* $\{|ta|\}_c$ = *collect-cond-actions* $\{|ta'|\}_c$ $\wedge$
collect-interrupts $\{|ta|\}_i$ = *collect-interrupts* $\{|ta'|\}_i$
and ⟦ *convert-extTA extNTA*,P,$t \vdash \langle es, s\rangle$ $[-ta\rightarrow]$ $\langle es', s'\rangle$; P,E,$H \vdash es$ [:] *Ts*; *hext H* (*hp s*) ⟧
$\Longrightarrow \exists$ ta' es' s'. *convert-extTA extNTA*,P,$t \vdash \langle es, (H, \textit{lcl } s)\rangle$ $[-ta'\rightarrow]$ $\langle es', s'\rangle$ $\wedge$
collect-locks $\{|ta|\}_l$ = *collect-locks* $\{|ta'|\}_l$ $\wedge$ *collect-cond-actions* $\{|ta|\}_c$ = *collect-cond-actions* $\{|ta'|\}_c$ $\wedge$
collect-interrupts $\{|ta|\}_i$ = *collect-interrupts* $\{|ta'|\}_i$
⟨*proof*⟩

lemma *can-lock-devreserp*:
⟦ *wf-J-prog P*; *red-mthr.can-sync P t* (*e*, *l*) h' *L*; P,E,$h \vdash e : T$; P,$h \vdash t \sqrt{t}$; *hconf h*; $h \trianglelefteq h'$ ⟧
$\Longrightarrow$ *red-mthr.can-sync P t* (*e*, *l*) *h L*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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.
$[\![\ E\ V = None;\ V \neq super;\ E\ this = \lfloor Class\ C \rfloor;\ P \vdash C\ sees\ V{:}T\ (fm)\ in\ D\ ]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash Var\ V \leadsto Var\ this{\cdot}V\{D\}$
| *AnnoBinOp*:
$[\![\ is\text{-}lub,P,E \vdash e1 \leadsto e1';\ \ is\text{-}lub,P,E \vdash e2 \leadsto e2'\ ]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash e1\ «bop»\ e2 \leadsto e1'\ «bop»\ e2'$
| *AnnoLAssVar*:
$[\![\ E\ V = \lfloor T \rfloor;\ V \neq super;\ is\text{-}lub,P,E \vdash e \leadsto e'\ ]\!] \Longrightarrow is\text{-}lub,P,E \vdash V{:=}e \leadsto V{:=}e'$
| *AnnoLAssField*:
$[\![\ E\ V = None;\ V \neq super;\ E\ this = \lfloor Class\ C \rfloor;\ P \vdash C\ sees\ V{:}T\ (fm)\ in\ D;\ is\text{-}lub,P,E \vdash e \leadsto e'\ ]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash V{:=}e \leadsto Var\ this{\cdot}V\{D\} := e'$
| *AnnoAAcc*:
$[\![\ is\text{-}lub,P,E \vdash a \leadsto a';\ is\text{-}lub,P,E \vdash i \leadsto i'\ ]\!] \Longrightarrow is\text{-}lub,P,E \vdash a\lfloor i \rceil \leadsto a'\lfloor i' \rceil$
| *AnnoAAss*:
$[\![\ is\text{-}lub,P,E \vdash a \leadsto a';\ is\text{-}lub,P,E \vdash i \leadsto i';\ is\text{-}lub,P,E \vdash e \leadsto e'\ ]\!] \Longrightarrow is\text{-}lub,P,E \vdash a\lfloor i \rceil := e \leadsto a'\lfloor i' \rceil := e'$
| *AnnoALength*:
$is\text{-}lub,P,E \vdash a \leadsto a' \Longrightarrow is\text{-}lub,P,E \vdash a{\cdot}length \leadsto a'{\cdot}length$
| — 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:
$[\![\ is\text{-}lub,P,E \vdash e \leadsto e';\ \ is\text{-}lub,P,E \vdash e' :: U;\ class\text{-}type\text{-}of'\ U = \lfloor C \rfloor;\ P \vdash C\ sees\ F{:}T\ (fm)\ in\ D;$
$is\text{-}Array\ U \longrightarrow F \neq array\text{-}length\text{-}field\text{-}name\ ]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash e{\cdot}F\{STR\ ''''\} \leadsto e'{\cdot}F\{D\}$
| *AnnoFAccALength*:
$[\![\ is\text{-}lub,P,E \vdash e \leadsto e';\ is\text{-}lub,P,E \vdash e' :: T\lfloor\rceil\ ]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash e{\cdot}array\text{-}length\text{-}field\text{-}name\{STR\ ''''\} \leadsto e'{\cdot}length$
| *AnnoFAccSuper*:
— In class C with super class D, "super" is syntactic sugar for "((D) this)" (cf. JLS, 15.11.2)
$[\![\ E\ this = \lfloor Class\ C \rfloor;\ C \neq Object;\ class\ P\ C = \lfloor (D,\ fs,\ ms) \rfloor;$
$P \vdash D\ sees\ F{:}T\ (fm)\ in\ D'\ ]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash Var\ super{\cdot}F\{STR\ ''''\} \leadsto (Cast\ (Class\ D)\ (Var\ this)){\cdot}F\{D'\}$
| *AnnoFAss*:
$[\![\ is\text{-}lub,P,E \vdash e1 \leadsto e1';\ \ is\text{-}lub,P,E \vdash e2 \leadsto e2';$
$is\text{-}lub,P,E \vdash e1' :: U;\ class\text{-}type\text{-}of'\ U = \lfloor C \rfloor;\ P \vdash C\ sees\ F{:}T\ (fm)\ in\ D;$
$is\text{-}Array\ U \longrightarrow F \neq array\text{-}length\text{-}field\text{-}name\ ]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash e1{\cdot}F\{STR\ ''''\} := e2 \leadsto e1'{\cdot}F\{D\} := e2'$
| *AnnoFAssSuper*:
$[\![\ E\ this = \lfloor Class\ C \rfloor;\ C \neq Object;\ class\ P\ C = \lfloor (D,\ fs,\ ms) \rfloor;$
$P \vdash D\ sees\ F{:}T\ (fm)\ in\ D';\ is\text{-}lub,P,E \vdash e \leadsto e'\ ]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash Var\ super{\cdot}F\{STR\ ''''\} := e \leadsto (Cast\ (Class\ D)\ (Var\ this)){\cdot}F\{D'\} := e'$
| *AnnoCAS*:
$[\![\ is\text{-}lub,P,E \vdash e1 \leadsto e1';\ is\text{-}lub,P,E \vdash e2 \leadsto e2';\ is\text{-}lub,P,E \vdash e3 \leadsto e3'\ ]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash e1{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3) \leadsto e1'{\cdot}compareAndSwap(D{\cdot}F,\ e2',\ e3')$
| *AnnoCall*:
$[\![\ is\text{-}lub,P,E \vdash e \leadsto e';\ \ is\text{-}lub,P,E \vdash es\ [\leadsto]\ es'\ ]\!]$
$\Longrightarrow is\text{-}lub,P,E \vdash Call\ e\ M\ es \leadsto Call\ e'\ M\ es'$
| *AnnoBlock*:
$is\text{-}lub,P,E(V \mapsto T) \vdash e \leadsto e' \ \Longrightarrow\ is\text{-}lub,P,E \vdash \{V{:}T{=}vo;\ e\} \leadsto \{V{:}T{=}vo;\ e'\}$
| *AnnoSync*:
$[\![\ is\text{-}lub,P,E \vdash e1 \leadsto e1';\ is\text{-}lub,P,E \vdash e2 \leadsto e2'\ ]\!]$

⟹ *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*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*′
⟨*proof*⟩

code-pred
(*modes*: (*i* ⇒ *i* ⇒ *o* ⇒ *bool*) ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
[*detect-switches*, *skip-proof*]
Anno
⟨*proof*⟩

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
⟨*proof*⟩

lemma *eval-Anno-i-i-i-o-conv*:

Predicate.eval (*Anno-code-i-i-i-o* P E e) = ($\lambda e'$. P,E $\vdash$ e $\leadsto'$ e')
⟨*proof*⟩

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)
⟨*proof*⟩

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

⟨*ML*⟩
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

⟨*ML*⟩
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* ⟨*proof*⟩

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* ⟨*proof*⟩

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*)
⟨*proof*⟩

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* C_0 M_0 *pc frs* =
 {(ε, (*None*, *h*, (*v* # *stk*, *loc*, C_0, M_0, *pc*+*1*)#*frs*))}

| *exec-instr-New*:
 exec-instr (*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, ⌊*addr-of-sys-xcpt OutOfMemory*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*) # *frs*)}
 else (λ(*h'*, *a*). (⦃*NewHeapElem a* (*Class-type C*)⦄, *None*, *h'*, (*Addr a* # *stk*, *loc*, C_0, M_0, *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 C_0 M_0 pc frs =
(let v = hd stk
in if v = Null then {(ε, ⌊addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C_0, M_0, pc) # frs)}
else let a = the-Addr v
in {(⦃ReadMem a (CField C F) v′⦄, None, h, (v′ # (tl stk), loc, C_0, M_0, pc + 1) # frs) | v′.
heap-read h a (CField C F) v′})

| exec-instr (Putfield F C) P t h stk loc C_0 M_0 pc frs =
(let v = hd stk;
r = hd (tl stk)
in if r = Null then {(ε, ⌊addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C_0, M_0, pc) # frs)}
else let a = the-Addr r
in {(⦃WriteMem a (CField C F) v⦄, None, h′, (tl (tl stk), loc, C_0, M_0, 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 C_0 M_0 pc frs =
{(ε, let U = the ($typeof_h$ (hd stk))

in if $P \vdash U \leq T$ *then* (*None*, *h*, (*stk*, *loc*, C_0, M_0, *pc* + *1*) # *frs*)
else (⌊*addr-of-sys-xcpt ClassCast*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*) # *frs*))}

| *exec-instr* (*Instanceof T*) *P t h stk loc* C_0 M_0 *pc frs* =
{(ε, *None*, *h*, (*Bool* (*hd stk* ≠ *Null* ∧ $P \vdash$ *the* (*typeof* $_h$ (*hd stk*)) ≤ *T*) # *tl stk*, *loc*, C_0, M_0, *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*,mxl_0,*ins*,*xt*)⌋ ⇒
let f' = ([],[*r*]@*ps*@(*replicate* mxl_0 *undefined-value*),*D*,*M*,*0*)
in {(ε, *None*, *h*, *f'* # (*stk*, *loc*, *C0*, *M0*, *pc*) # *frs*)}))

| *exec-instr Return P t h* stk_0 loc_0 C_0 M_0 *pc frs* =
{(ε, (*if frs*=[] *then* (*None*, *h*, []) *else*
let v = *hd* stk_0;
(*stk*,*loc*,*C*,*m*,*pc*) = *hd frs*;
n = *length* (*fst* (*snd* (*method P* C_0 M_0)))
in (*None*, *h*, (*v*#(*drop* (*n*+*1*) *stk*),*loc*,*C*,*m*,*pc*+*1*)#*tl frs*)))}

| *exec-instr Pop P t h stk loc* C_0 M_0 *pc frs* =
{(ε, (*None*, *h*, (*tl stk*, *loc*, C_0, M_0, *pc*+*1*)#*frs*))}

| *exec-instr Dup P t h stk loc* C_0 M_0 *pc frs* =
{(ε, (*None*, *h*, (*hd stk* # *stk*, *loc*, C_0, M_0, *pc*+*1*)#*frs*))}

| *exec-instr Swap P t h stk loc* C_0 M_0 *pc frs* =
{(ε, (*None*, *h*, (*hd* (*tl stk*) # *hd stk* # *tl* (*tl stk*), *loc*, C_0, M_0, *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* C_0 M_0 *pc frs* =
{(ε, (*let pc'* = *if hd stk* = *Bool False then nat*(*int pc*+*i*) *else pc*+*1*
in (*None*, *h*, (*tl stk*, *loc*, C_0, M_0, *pc'*)#*frs*)))}

| *exec-instr-Goto*:
exec-instr (*Goto i*) *P t h stk loc* C_0 M_0 *pc frs* =
{(ε, (*None*, *h*, (*stk*, *loc*, C_0, M_0, *nat*(*int pc*+*i*))#*frs*))}

| *exec-instr ThrowExc P t h stk loc* C_0 M_0 *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*, C_0, M_0, *pc*)#*frs*)))}

| *exec-instr-MEnter*:
exec-instr MEnter P t h stk loc C_0 M_0 *pc frs* =
{*let v = hd stk*
in if v = Null
then (ε, ⌊*addr-of-sys-xcpt NullPointer*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*) # *frs*)
else (⦃*Lock*→*the-Addr v*, *SyncLock* (*the-Addr v*)⦄, *None*, *h*, (*tl stk*, *loc*, C_0, M_0, *pc* + *1*) # *frs*)}

| *exec-instr-MExit*:
exec-instr MExit P t h stk loc C_0 M_0 *pc frs* =
(*let v = hd stk*
in if v = Null
then {(ε, ⌊*addr-of-sys-xcpt NullPointer*⌋, *h*, (*stk*, *loc*, C_0, M_0, *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* $\preceq^*$ *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′*
⟨*proof*⟩

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 =
(λ*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*))
⟨*proof*⟩

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:
$(ta, \sigma') \in exec\ P\ t\ \sigma \Longrightarrow P,t \vdash \sigma\ -ta-jvm\rightarrow \sigma'$

lemma *exec-1-iff*:
$P,t \vdash \sigma\ -ta-jvm\rightarrow \sigma' \longleftrightarrow (ta, \sigma') \in exec\ P\ t\ \sigma$
⟨*proof*⟩

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* < *length stk* ∧ *n* < *length loc*)

| *check-instr-Push*:
check-instr (*Push v*) *P h stk loc* C_0 M_0 *pc frs* =
(¬*is-Addr v*)

| *check-instr-New*:
check-instr (*New C*) *P h stk loc* C_0 M_0 *pc frs* =
is-class P C

| *check-instr-NewArray*:
check-instr (*NewArray T*) *P h stk loc C0 M0 pc frs* =
(*is-type P* (*T*⌊⌉) ∧ *0* < *length stk* ∧ *is-Intg* (*hd stk*))

| *check-instr-ALoad*:
check-instr ALoad P h stk loc C0 M0 pc frs =
(*1* < *length stk* ∧ *is-Intg* (*hd stk*) ∧ *is-Array-ref* (*hd* (*tl stk*)) *h*)

| *check-instr-AStore*:
check-instr AStore P h stk loc C0 M0 pc frs =
(*2* < *length stk* ∧ *is-Intg* (*hd* (*tl stk*)) ∧ *is-Array-ref* (*hd* (*tl* (*tl stk*))) *h* ∧ $typeof_h$ (*hd stk*) ≠ *None*)

| *check-instr-ALength*:
check-instr ALength P h stk loc C0 M0 pc frs =
(*0* < *length stk* ∧ *is-Array-ref* (*hd stk*) *h*)

| *check-instr-Getfield*:
check-instr (*Getfield F C*) *P h stk loc* C_0 M_0 *pc frs* =
(*0* < *length stk* ∧ (∃ *C' T fm*. *P* ⊢ *C sees F*:*T* (*fm*) *in C'*) ∧
(*let* (*C'*, *T*, *fm*) = *field P C F*; *ref* = *hd stk in*
C' = *C* ∧ *is-Ref ref* ∧ (*ref* ≠ *Null* ⟶
(∃ *T*. *typeof-addr h* (*the-Addr ref*) = ⌊*T*⌋ ∧ *P* ⊢ *class-type-of T* ⪯* *C*))))

| *check-instr-Putfield*:
check-instr (*Putfield F C*) *P h stk loc* C_0 M_0 *pc frs* =
(*1* < *length stk* ∧ (∃ *C' T fm*. *P* ⊢ *C sees F*:*T* (*fm*) *in C'*) ∧
(*let* (*C'*, *T*, *fm*) = *field P C F*; *v* = *hd stk*; *ref* = *hd* (*tl stk*) *in*

$C' = C \wedge$ *is-Ref ref* $\wedge$ (*ref* $\neq$ *Null* $\longrightarrow$
($\exists T'$. *typeof-addr h* (*the-Addr ref*) $= \lfloor T' \rfloor \wedge P \vdash$ *class-type-of* $T' \preceq^* C \wedge P,h \vdash v :\leq T$))))

| *check-instr-CAS*:
check-instr (*CAS F C*) *P h stk loc C0 M0 pc frs* =
(*2* < *length stk* $\wedge$ ($\exists C'$ *T fm*. $P \vdash$ *C sees F*:*T* (*fm*) *in* C') $\wedge$
(*let* (C', *T*, *fm*) = *field P C F*; v'' = *hd stk*; v' = *hd* (*tl stk*); *v* = *hd* (*tl* (*tl stk*)) *in*
$C' = C \wedge$ *is-Ref v* $\wedge$ *volatile fm* $\wedge$ ($v \neq$ *Null* $\longrightarrow$
($\exists T'$. *typeof-addr h* (*the-Addr v*) $= \lfloor T' \rfloor \wedge P \vdash$ *class-type-of* $T' \preceq^* C \wedge P,h \vdash v' :\leq T \wedge P,h \vdash v'' :\leq T$))))

| *check-instr-Checkcast*:
check-instr (*Checkcast T*) *P h stk loc* C_0 M_0 *pc frs* =
(*0* < *length stk* $\wedge$ *is-type P T*)

| *check-instr-Instanceof*:
check-instr (*Instanceof T*) *P h stk loc* C_0 M_0 *pc frs* =
(*0* < *length stk* $\wedge$ *is-type P T* $\wedge$ *is-Ref* (*hd stk*))

| *check-instr-Invoke*:
check-instr (*Invoke M n*) *P h stk loc* C_0 M_0 *pc frs* =
(*n* < *length stk* $\wedge$ *is-Ref* (*stk*!*n*) $\wedge$
(*stk*!*n* $\neq$ *Null* $\longrightarrow$
(*let a* = *the-Addr* (*stk*!*n*);
T = *the* (*typeof-addr h a*);
C = *class-type-of T*;
(*D*, *Ts*, *Tr*, *meth*) = *method P C M*
in typeof-addr h a $\neq$ *None* $\wedge$ $P \vdash$ *C has M* $\wedge$
$P,h \vdash$ *rev* (*take n stk*) [:≤] *Ts* $\wedge$
(*meth* = *None* $\longrightarrow$ *D*•*M*(*Ts*) :: *Tr*))))

| *check-instr-Return*:
check-instr Return P h stk loc C_0 M_0 *pc frs* =
(*0* < *length stk* $\wedge$ ((*0* < *length frs*) $\longrightarrow$
($P \vdash C_0$ *has* M_0) $\wedge$
(*let v* = *hd stk*;
T = *fst* (*snd* (*snd* (*method P* C_0 M_0)))
in $P,h \vdash v :\leq T$)))

| *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 C0 M0 pc frs* =
(*1* < *length stk* $\wedge$ ($\exists$ *T1 T2 T*. $typeof_h$ (*hd stk*) = $\lfloor$*T2*$\rfloor$ $\wedge$ $typeof_h$ (*hd* (*tl stk*)) = $\lfloor$*T1*$\rfloor$ $\wedge$ $P \vdash$

T1 «*bop*» *T2* : *T*))

| *check-instr-IfFalse*:
  check-instr (*IfFalse b*) *P h stk loc* C_0 M_0 *pc frs* =
  (*0* < *length stk* ∧ *is-Bool* (*hd 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* ∧ *is-Ref* (*hd stk*) ∧ *P* ⊢ *the* ($typeof_h$ (*hd stk*)) ≤ *Class Throwable*)

| *check-instr-MEnter*:
  check-instr MEnter P h stk loc C_0 M_0 *pc frs* =
  (*0* < *length stk* ∧ *is-Ref* (*hd 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*:
$P,t \vdash$ *Normal* σ $-ta-jvmd\rightarrow$ *Normal* $\sigma' \Longrightarrow$ *check* $P\ \sigma \wedge (ta, \sigma') \in$ *exec* $P\ t\ \sigma \wedge (\exists xcp\ h\ f\ frs.\ \sigma = (xcp, h, f \# frs))$
⟨*proof*⟩

lemma *jvmd-NormalE*:
assumes $P,t \vdash$ *Normal* σ $-ta-jvmd\rightarrow$ *Normal* σ'
obtains *xcp h f frs* **where** *check* $P\ \sigma$ $(ta, \sigma') \in$ *exec* $P\ t\ \sigma$ $\sigma = (xcp, h, f \# frs)$
⟨*proof*⟩

lemma *exec-d-eq-TypeError*: *exec-d* $P\ t\ \sigma$ = *TypeError* $\longleftrightarrow \neg$ *check* $P\ \sigma$
⟨*proof*⟩

lemma *exec-d-eq-Normal*: *exec-d* $P\ t\ \sigma$ = *Normal* (*exec* $P\ t\ \sigma$) $\longleftrightarrow$ *check* $P\ \sigma$
⟨*proof*⟩

end

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

lemma *if-neq* [*dest*!]:
(*if P then A else B*) $\neq B \Longrightarrow P$
⟨*proof*⟩

context *JVM-heap-base* **begin**

lemma *exec-d-no-errorI* [*intro*]:
check $P\ \sigma \Longrightarrow$ *exec-d* $P\ t\ \sigma \neq$ *TypeError*
⟨*proof*⟩

theorem *no-type-error-commutes*:
exec-d $P\ t\ \sigma \neq$ *TypeError* $\Longrightarrow$ *exec-d* $P\ t\ \sigma$ = *Normal* (*exec* $P\ t\ \sigma$)
⟨*proof*⟩

lemma *defensive-imp-aggressive-1*:
$P,t \vdash$ (*Normal* σ) $-tas-jvmd\rightarrow$ (*Normal* σ') $\Longrightarrow P,t \vdash \sigma\ -tas-jvm\rightarrow \sigma'$
⟨*proof*⟩

end

context *JVM-heap* **begin**

lemma *check-exec-hext*:
assumes *exec*: $(ta, xcp', h', frs') \in$ *exec* $P\ t\ (xcp, h, frs)$
and *check*: *check* $P\ (xcp, h, frs)$
shows $h \trianglelefteq h'$
⟨*proof*⟩

lemma *exec-1-d-hext*:
$[\![\ P,t \vdash$ *Normal* (xcp, h, frs) $-ta-jvmd\rightarrow$ *Normal* $(xcp', h', frs')\ ]\!] \Longrightarrow h \trianglelefteq h'$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *NewThread-memory-exec*:
⟦ *P*,*t* ⊢ σ −*ta*−*jvm*→ σ'; *NewThread t' x m* ∈ *set* $\{\!|ta|\!\}_t$ ⟧ ⟹ *m* = (*fst* (*snd* σ'))
⟨*proof*⟩

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* $\{\!|ta|\!\}_i$ = {}
⟨*proof*⟩

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* $\{\!|ta|\!\}_i$ = {}
⟨*proof*⟩

lemma *mexec-final*:
⟦ *mexec P t* (*x*, *m*) *ta* (*x'*, *m'*); *JVM-final x* ⟧ ⟹ *False*
⟨*proof*⟩

lemma *exec-mthr*: *multithreaded JVM-final* (*mexec P*)
⟨*proof*⟩

end

sublocale *JVM-heap-base* < *exec-mthr*:
multithreaded
JVM-final
mexec P
convert-RA
for *P*
⟨*proof*⟩

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*
(‹- ⊢ - −-▷-$\longrightarrow_{jvm}$ -› [*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*
(‹- ⊢ - −▷-$\longrightarrow_{jvm}*$ -› [*50*,*0*,*0*,*50*] *80*)
where
P ⊢ *s* −▷*ttas*$\rightarrow_{jvm}*$ *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*)
⟨*proof*⟩

end

sublocale *JVM-heap-base* < *execd-mthr*:
multithreaded
JVM-final
mexecd P

convert-RA
for *P*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *exec-instr-preserve-tconf*:

$\llbracket$ (*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* $\rrbracket$
⟹ *P*,*h*′ ⊢ *t*′ √*t*
⟨*proof*⟩

lemma *exec-preserve-tconf*:
$\llbracket$ *P*,*t* ⊢ *Normal* (*xcp*, *h*, *frs*) −*ta*−*jvmd*→ *Normal* (*xcp*′, *h*′, *frs*′); *P*,*h* ⊢ *t*′ √*t* $\rrbracket$ ⟹ *P*,*h*′ ⊢ *t*′ √*t*
⟨*proof*⟩

lemma *lifting-wf-thread-conf*: *lifting-wf JVM-final* (*mexecd P*) (λ*t x m*. *P*,*m* ⊢ *t* √*t*)
⟨*proof*⟩

end

sublocale *JVM-heap* < *execd-tconf*: *lifting-wf JVM-final mexecd P convert-RA* λ*t x m*. *P*,*m* ⊢ *t* √*t*
⟨*proof*⟩

context *JVM-heap* **begin**

lemma *execd-hext*:
P ⊢ *s* −*t*▷*ta*→$_{jvmd}$ *s*′ ⟹ *shr s* ⊴ *shr s*′
⟨*proof*⟩

lemma *Execd-hext*:
assumes *P* ⊢ *s* −▷*tta*→$_{jvmd}$∗ *s*′
shows *shr s* ⊴ *shr s*′
⟨*proof*⟩

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* ⇒ *cname* ⇒ ty_m

definition *stk-esl* :: *'c prog* ⇒ *nat* ⇒ ty_s *esl*
where
stk-esl P mxs ≡ *upto-esl mxs* (*SemiType.esl P*)

definition *loc-sl* :: *'c prog* ⇒ *nat* ⇒ ty_l *sl*
where
loc-sl P mxl ≡ *Listn.sl mxl* (*Err.sl* (*SemiType.esl P*))

definition *sl* :: *'c prog* ⇒ *nat* ⇒ *nat* ⇒ $ty_i{}'$ *err sl*
where
sl P mxs mxl ≡
Err.sl(*Opt.esl*(*Product.esl* (*stk-esl P mxs*) (*Err.esl*(*loc-sl P mxl*))))

definition *states* :: *'c prog* ⇒ *nat* ⇒ *nat* ⇒ $ty_i{}'$ *err set*
where
states P mxs mxl ≡ *fst*(*sl P mxs mxl*)

definition *le* :: *'c prog* ⇒ *nat* ⇒ *nat* ⇒ $ty_i{}'$ *err ord*
where
le P mxs mxl ≡ *fst*(*snd*(*sl P mxs mxl*))

definition *sup* :: *'c prog* ⇒ *nat* ⇒ *nat* ⇒ $ty_i{}'$ *err binop*
where
sup P mxs mxl ≡ *snd*(*snd*(*sl P mxs mxl*))

definition *sup-ty-opt* :: $['c\ prog,ty\ err,ty\ err] \Rightarrow bool$
  (‹- ⊢ - $\leq_\top$ -› [71,71,71] 70)
where
  sup-ty-opt P ≡ *Err.le* (*widen P*)

definition *sup-state* :: $['c\ prog,ty_i,ty_i] \Rightarrow bool$
  (‹- ⊢ - $\leq_i$ -› [71,71,71] 70)
where
  sup-state P ≡ *Product.le* (*Listn.le* (*widen P*)) (*Listn.le* (*sup-ty-opt P*))

definition *sup-state-opt* :: $['c\ prog,ty_i{}',ty_i{}'] \Rightarrow bool$
  (‹- ⊢ - $\leq''$ -› [71,71,71] 70)
where
  sup-state-opt P ≡ *Opt.le* (*sup-state P*)

abbreviation *sup-loc* :: $['c\ prog,ty_l,ty_l] \Rightarrow bool$ (‹- ⊢ - [$\leq_\top$] -› [71,71,71] 70)
where *P* ⊢ *LT* [$\leq_\top$] *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*))))
  ⟨*proof*⟩

lemma *JVM-le-unfold*:
 le P m n ≡
  Err.le(*Opt.le*(*Product.le*(*Listn.le*(*widen P*))(*Listn.le*(*Err.le*(*widen P*)))))
  ⟨*proof*⟩

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*)
 ⟨*proof*⟩

lemma *JVM-le-conv*:
  le P m n (*OK t1*) (*OK t2*) = *P* ⊢ *t1* ≤' *t2*
 ⟨*proof*⟩

lemma *JVM-le-Err-conv*:
  le P m n = *Err.le* (*sup-state-opt P*)
 ⟨*proof*⟩

lemma *err-le-unfold* [*iff*]:
  Err.le r (*OK a*) (*OK b*) = *r a b*
 ⟨*proof*⟩

6.1.2 Semilattice

lemma *order-sup-state-opt* [*intro*, *simp*]:
wf-prog wf-mb P $\Longrightarrow$ *order* (*sup-state-opt P*)
⟨*proof*⟩

lemma *semilat-JVM* [*intro?*]:
wf-prog wf-mb P $\Longrightarrow$ *semilat* (*JVM-SemiType.sl P mxs mxl*)
⟨*proof*⟩

lemma *acc-JVM* [*intro*]:
wf-prog wf-mb P $\Longrightarrow$ *acc* (*JVM-SemiType.states P mxs mxl*) (*JVM-SemiType.le P mxs mxl*)
⟨*proof*⟩

6.1.3 Widening with ⊤

lemma *widen-refl*[*iff*]: *widen P t t* ⟨*proof*⟩

lemma *sup-ty-opt-refl* [*iff*]: $P \vdash T \leq_{\top} T$
⟨*proof*⟩

lemma *Err-any-conv* [*iff*]: $P \vdash Err \leq_{\top} T = (T = Err)$
⟨*proof*⟩

lemma *any-Err* [*iff*]: $P \vdash T \leq_{\top} Err$
⟨*proof*⟩

lemma *OK-OK-conv* [*iff*]:
$P \vdash OK\ T \leq_{\top} OK\ T' = P \vdash T \leq T'$
⟨*proof*⟩

lemma *any-OK-conv* [*iff*]:
$P \vdash X \leq_{\top} OK\ T' = (\exists T.\ X = OK\ T \wedge P \vdash T \leq T')$
⟨*proof*⟩

lemma *OK-any-conv*:
$P \vdash OK\ T \leq_{\top} X = (X = Err \vee (\exists T'.\ X = OK\ T' \wedge P \vdash T \leq T'))$
⟨*proof*⟩

lemma *sup-ty-opt-trans* [*intro?*, *trans*]:
$[\![P \vdash a \leq_{\top} b;\ P \vdash b \leq_{\top} c]\!] \Longrightarrow P \vdash a \leq_{\top} c$
⟨*proof*⟩

6.1.4 Stack and Registers

lemma *stk-convert*:
$P \vdash ST\ [\leq]\ ST' = Listn.le\ (widen\ P)\ ST\ ST'$
⟨*proof*⟩

lemma *sup-loc-refl* [*iff*]: $P \vdash LT\ [\leq_{\top}]\ LT$
⟨*proof*⟩

lemmas *sup-loc-Cons1* [*iff*] = *list-all2-Cons1* [*of sup-ty-opt P*] **for** *P*

lemma *sup-loc-def*:

$P \vdash LT\ [\le_\top]\ LT' \equiv Listn.le\ (sup\text{-}ty\text{-}opt\ P)\ LT\ LT'$
⟨*proof*⟩

lemma *sup-loc-widens-conv* [*iff*]:
$P \vdash map\ OK\ Ts\ [\le_\top]\ map\ OK\ Ts' = P \vdash Ts\ [\le]\ Ts'$
⟨*proof*⟩

lemma *sup-loc-trans* [*intro?*, *trans*]:
$[\![P \vdash a\ [\le_\top]\ b;\ P \vdash b\ [\le_\top]\ c]\!] \Longrightarrow P \vdash a\ [\le_\top]\ c$
⟨*proof*⟩

6.1.5 State Type

lemma *sup-state-conv* [*iff*]:
$P \vdash (ST,LT) \le_i (ST',LT') = (P \vdash ST\ [\le]\ ST' \wedge P \vdash LT\ [\le_\top]\ LT')$
⟨*proof*⟩

lemma *sup-state-conv2*:
$P \vdash s1 \le_i s2 = (P \vdash fst\ s1\ [\le]\ fst\ s2 \wedge P \vdash snd\ s1\ [\le_\top]\ snd\ s2)$
⟨*proof*⟩

lemma *sup-state-refl* [*iff*]: $P \vdash s \le_i s$
⟨*proof*⟩

lemma *sup-state-trans* [*intro?*, *trans*]:
$[\![P \vdash a \le_i b;\ P \vdash b \le_i c]\!] \Longrightarrow P \vdash a \le_i c$
⟨*proof*⟩

lemma *sup-state-opt-None-any* [*iff*]:
$P \vdash None \le' s$
⟨*proof*⟩

lemma *sup-state-opt-any-None* [*iff*]:
$P \vdash s \le' None = (s = None)$
⟨*proof*⟩

lemma *sup-state-opt-Some-Some* [*iff*]:
$P \vdash Some\ a \le' Some\ b = P \vdash a \le_i b$
⟨*proof*⟩

lemma *sup-state-opt-any-Some*:
$P \vdash (Some\ s) \le' X = (\exists s'.\ X = Some\ s' \wedge P \vdash s \le_i s')$
⟨*proof*⟩

lemma *sup-state-opt-refl* [*iff*]: $P \vdash s \le' s$
⟨*proof*⟩

lemma *sup-state-opt-trans* [*intro?*, *trans*]:
$[\![P \vdash a \le' b;\ P \vdash b \le' c]\!] \Longrightarrow P \vdash a \le' c$
⟨*proof*⟩

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* ≡ *1+size Ts+*mxl_0

Program counter of successor instructions:

primrec *succs* :: *'addr instr* ⇒ ty_i ⇒ *pc* ⇒ *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 = []
⟨*proof*⟩

lemma *eff-Some*:
eff i P pc xt (*Some* τ) = *norm-eff i P pc* τ @ *xcpt-eff i P pc* τ *xt*
⟨*proof*⟩

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* ⊢

$T_3 \le T_f$)
| 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* $\preceq^*$ *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 $\tau \equiv \forall (f,t,C,h,d) \in$ *set* (*relevant-entries P i pc xt*). (*case C of None* ⇒ *True* | *Some C'* ⇒ *is-class P C'*) $\wedge$ $d \leq$ *size* (*fst* τ) $\wedge$ $d <$ *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) $\wedge$ *xcpt-app i P pc mxs xt* τ $\wedge$
($\forall (pc',\tau') \in$ *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) $\wedge$ *xcpt-app i P pc mxs xt* τ $\wedge$
($\forall (pc',s') \in$ *set* (*eff i P pc xt* (*Some* τ)). $pc' <$ *mpc*))
⟨*proof*⟩

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 $\bigwedge LT.\ P\ ([],LT)$
assumes $\bigwedge l\ ST\ LT.\ P\ (l\#ST,LT)$
shows *P s*

⟨*proof*⟩

lemma *length-cases3*:
assumes $\bigwedge LT.\ P\ ([],LT)$
assumes $\bigwedge l\ LT.\ P\ ([l],LT)$
assumes $\bigwedge l\ l'\ ST\ LT.\ P\ (l\#l'\#ST,LT)$
shows $P\ s$
⟨*proof*⟩

lemma *length-cases4*:
assumes $\bigwedge LT.\ P\ ([],LT)$
assumes $\bigwedge l\ LT.\ P\ ([l],LT)$
assumes $\bigwedge l\ l'\ LT.\ P\ ([l,l'],LT)$
assumes $\bigwedge l\ l'\ l''\ ST\ LT.\ P\ (l\#l'\#l''\#ST,LT)$
shows $P\ s$
⟨*proof*⟩

lemma *length-cases5*:
assumes $\bigwedge LT.\ P\ ([],LT)$
assumes $\bigwedge l\ LT.\ P\ ([l],LT)$
assumes $\bigwedge l\ l'\ LT.\ P\ ([l,l'],LT)$
assumes $\bigwedge l\ l'\ l''\ LT.\ P\ ([l,l',l''],LT)$
assumes $\bigwedge l\ l'\ l''\ l'''\ ST\ LT.\ P\ (l\#l'\#l''\#l'''\#ST,LT)$
shows $P\ s$
⟨*proof*⟩

simp rules for *app*

lemma *appNone*[*simp*]: $app\ i\ P\ mxs\ T_r\ pc\ mpc\ et\ None = True$
⟨*proof*⟩

lemma *appLoad*[*simp*]:
$app_i\ (Load\ idx,\ P,\ T_r,\ mxs,\ pc,\ s) = (\exists\, ST\ LT.\ s = (ST,LT) \wedge idx < length\ LT \wedge LT!idx \neq Err \wedge length\ ST < mxs)$
⟨*proof*⟩

lemma *appStore*[*simp*]:
$app_i\ (Store\ idx,P,pc,mxs,T_r,s) = (\exists\, ts\ ST\ LT.\ s = (ts\#ST,LT) \wedge idx < length\ LT)$
⟨*proof*⟩

lemma *appPush*[*simp*]:
$app_i\ (Push\ v,P,pc,mxs,T_r,s) =$
$(\exists\, ST\ LT.\ s = (ST,LT) \wedge length\ ST < mxs \wedge typeof\ v \neq None)$
⟨*proof*⟩

lemma *appGetField*[*simp*]:
$app_i\ (Getfield\ F\ C,P,pc,mxs,T_r,s) =$
$(\exists\ oT\ vT\ ST\ LT\ fm.\ s = (oT\#ST,\ LT) \wedge$
$P \vdash C\ sees\ F{:}vT\ (fm)\ in\ C \wedge P \vdash oT \leq (Class\ C))$
⟨*proof*⟩

lemma *appPutField*[*simp*]:
$app_i\ (Putfield\ F\ C,P,pc,mxs,T_r,s) =$

(∃ vT vT' oT ST LT fm. $s = (vT\#oT\#ST,\ LT)$ ∧
$P \vdash C$ *sees* $F{:}vT'$ (fm) *in* C ∧ $P \vdash oT \leq (Class\ C)$ ∧ $P \vdash vT \leq vT'$)
⟨*proof*⟩

lemma *appCAS*[*simp*]:
app_i (*CAS F C*, *P*, *pc*, *mxs*, T_r, *s*) =
(∃ *T1 T2 T3* T' *ST LT fm*. $s = (T3\ \#\ T2\ \#\ T1\ \#\ ST,\ LT)$ ∧
$P \vdash C$ *sees* $F{:}T'$ (fm) *in* C ∧ *volatile fm* ∧ $P \vdash T1 \leq Class\ C$ ∧ $P \vdash T2 \leq T'$ ∧ $P \vdash T3 \leq T'$)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *appNewArray*[*simp*]:
app_i (*NewArray Ty*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *ST LT*. *s*=(*Integer*#*ST*,*LT*) ∧ *is-type P* (*Ty*⌊⌉))
⟨*proof*⟩

lemma *appALoad*[*simp*]:
app_i (*ALoad*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *T ST LT*. *s*=(*Integer*#*T*#*ST*,*LT*) ∧ (*T* ≠ *NT* ⟶ (∃ T'. $T = T'$⌊⌉)))
⟨*proof*⟩

lemma *appAStore*[*simp*]:
app_i (*AStore*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *T U ST LT*. *s*=(*T*#*Integer*#*U*#*ST*,*LT*) ∧ (*U* ≠ *NT* ⟶ (∃ T'. $U = T'$⌊⌉)))
⟨*proof*⟩

lemma *appALength*[*simp*]:
app_i (*ALength*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *T ST LT*. *s*=(*T*#*ST*,*LT*) ∧ (*T* ≠ *NT* ⟶ (∃ T'. $T = T'$⌊⌉)))
⟨*proof*⟩

lemma *appCheckcast*[*simp*]:
app_i (*Checkcast Ty*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *T ST LT*. *s* = (*T*#*ST*,*LT*) ∧ *is-type P Ty*)
⟨*proof*⟩

lemma *appInstanceof*[*simp*]:
app_i (*Instanceof Ty*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *T ST LT*. *s* = (*T*#*ST*,*LT*) ∧ *is-type P Ty* ∧ *is-refT T*)
⟨*proof*⟩

lemma $app_i Pop$[*simp*]:
app_i (*Pop*,*P*,*pc*,*mxs*,T_r,*s*) = (∃ *ts ST LT*. *s* = (*ts*#*ST*,*LT*))
⟨*proof*⟩

lemma *appDup*[*simp*]:
app_i (*Dup*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *T ST LT*. *s* = (*T*#*ST*,*LT*) ∧ *Suc* (*length ST*) < *mxs*)
⟨*proof*⟩

lemma $app_i Swap$[*simp*]:
app_i (*Swap*,*P*,*pc*,*mxs*,T_r,*s*) = (∃ *T1 T2 ST LT*. *s* = (*T1*#*T2*#*ST*,*LT*))
⟨*proof*⟩

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*⟩

lemma *appIfFalse* [*simp*]:
app_i (*IfFalse b*,*P*,*pc*,*mxs*,T_r,*s*) =
(∃ *ST LT*. *s* = (*Boolean*#*ST*,*LT*) ∧ *0* ≤ *int pc* + *b*)
⟨*proof*⟩

lemma *appReturn*[*simp*]:
app_i (*Return*,*P*,*pc*,*mxs*,T_r,*s*) = (∃ *T ST LT*. *s* = (*T*#*ST*,*LT*) ∧ *P* ⊢ *T* ≤ T_r)
⟨*proof*⟩

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* $\preceq^*$ *Throwable*)))
⟨*proof*⟩

lemma *appMEnter*[*simp*]:
app_i (*MEnter*,*P*,*pc*,*mxs*,T_r,*s*) = (∃ *T ST LT*. *s*=(*T*#*ST*,*LT*) ∧ *is-refT T*)
⟨*proof*⟩

lemma *appMExit*[*simp*]:
app_i (*MExit*,*P*,*pc*,*mxs*,T_r,*s*) = (∃ *T ST LT*. *s*=(*T*#*ST*,*LT*) ∧ *is-refT T*)
⟨*proof*⟩

lemma *effNone*:
(*pc*′, *s*′) ∈ *set* (*eff i P pc et None*) ⟹ *s*′ = *None*
⟨*proof*⟩

lemma *relevant-entries-append* [*simp*]:
relevant-entries P i pc (*xt* @ *xt*′) = *relevant-entries P i pc xt* @ *relevant-entries P i pc xt*′
⟨*proof*⟩

lemma *xcpt-app-append* [*iff*]:
xcpt-app i P pc mxs (*xt*@*xt*′) τ = (*xcpt-app i P pc mxs xt* τ ∧ *xcpt-app i P pc mxs xt*′ τ)
⟨*proof*⟩

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*′
⟨*proof*⟩

lemma *app-append* [*simp*]:
app i P pc T mxs mpc (*xt*@*xt*′) τ = (*app i P pc T mxs mpc xt* τ ∧ *app i P pc T mxs mpc xt*′ τ)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)) ⟷
Predicate.holds (*Predicate.bind* (*sees-field-i-i-i-o-o-i P C F C*) (λ*T*. *Predicate.single* ())) ∧ *P* ⊢ *T* ≤ *Class C*
⟨*proof*⟩

lemma app_i*-Putfield-code*:
app_i (*Putfield F C*, *P*, *pc*, *mxs*, T_r, (T_1#T_2#*ST*, *LT*)) ⟷
P ⊢ T_2 ≤ (*Class C*) ∧
Predicate.holds (*Predicate.bind* (*sees-field-i-i-i-o-o-i P C F C*) (λ(*T*, *fm*). *if P* ⊢ T_1 ≤ *T then Predicate.single* () *else bot*))
⟨*proof*⟩

lemma app_i*-CAS-code*:
app_i (*CAS F C*, *P*, *pc*, *mxs*, T_r, (T_3#T_2#T_1#*ST*, *LT*)) ⟷
P ⊢ T_1 ≤ *Class C* ∧
Predicate.holds (*Predicate.bind* (*sees-field-i-i-i-o-o-i P C F C*) (λ(*T*, *fm*). *if P* ⊢ T_2 ≤ *T* ∧ *P* ⊢ T_3 ≤ *T* ∧ *volatile fm then Predicate.single* () *else bot*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma app_i*-ALength-code*:
app_i (*ALength*, *P*, *pc*, *mxs*, T_r, (*T1#ST*,*LT*)) =
(*case T1 of Ty*⌊⌉ ⇒ *True* | *NT* ⇒ *True* | - ⇒ *False*)
⟨*proof*⟩

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* ()))
⟨*proof*⟩

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*⟩

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* $\preceq^*$ *Throwable* | - ⇒ *False*)
⟨*proof*⟩

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* ⇒ *nat* ⇒ *nat* ⇒ ty_i*′ err list* ⇒ *bool*
where
check-types P mxs mxl τs ≡ *set τs* ⊆ *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-table*,*′addr instr*,*pc*,ty_m] ⇒ *bool*
(‹-,-,-,-,- ⊢ -,- :: -› [*60*,*0*,*0*,*0*,*0*,*0*,*0*,*61*] *60*)
where
P,*T*,*mxs*,*mpc*,*xt* ⊢ *i*,*pc* :: *τs* ≡
app i P mxs T pc mpc xt (*τs*!*pc*) ∧
(∀ (*pc′*,*τ′*) ∈ *set* (*eff i P pc xt* (*τs*!*pc*)). *P* ⊢ *τ′* ≤*′* *τs*!*pc′*)

— The type at *pc*=*0* conforms to the method calling convention:
definition *wt-start* :: [*′m prog*,*cname*,*ty list*,*nat*,ty_m] ⇒ *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-table*,ty_m] ⇒ *bool*
where
wt-method P C Ts T_r *mxs* mxl_0 *is xt* $\tau s \equiv$
0 < *size is* ∧ *size* τs = *size is* ∧
check-types P mxs (*1*+*size Ts*+mxl_0) (*map OK* τs) ∧
wt-start P C Ts mxl_0 τs ∧
(∀ *pc* < *size is*. *P*,T_r,*mxs*,*size is*,*xt* ⊢ *is*!*pc*,*pc* :: τs)

— A program is welltyped if it is wellformed and all methods are welltyped
definition *wf-jvm-prog-phi* :: ty_P ⇒ '*addr jvm-prog* ⇒ *bool* (‹*wf*'-*jvm*'-*prog*-›)
where
wf-jvm-prog$_\Phi$ ≡
  wf-prog (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*))

definition *wf-jvm-prog* :: '*addr jvm-prog* ⇒ *bool*
where
wf-jvm-prog P ≡ ∃Φ. *wf-jvm-prog*$_\Phi$ *P*

lemma *wt-jvm-progD*:
wf-jvm-prog$_\Phi$ *P* ⟹ ∃ *wt*. *wf-prog wt P*⟨*proof*⟩
lemma *wt-jvm-prog-impl-wt-instr*:
⟦ *wf-jvm-prog*$_\Phi$ *P*;
  P ⊢ *C sees M*:*Ts* → *T* = ⌊(*mxs*,mxl_0,*ins*,*xt*)⌋ *in C*; *pc* < *size ins* ⟧
⟹ *P*,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc*,*pc* :: Φ *C M*⟨*proof*⟩
lemma *wt-jvm-prog-impl-wt-start*:
⟦ *wf-jvm-prog*$_\Phi$ *P*;
  P ⊢ *C sees M*:*Ts* → *T* = ⌊(*mxs*,mxl_0,*ins*,*xt*)⌋ *in C* ⟧ ⟹
0 < *size ins* ∧ *wt-start P C Ts* mxl_0 (Φ *C M*)⟨*proof*⟩
end

6.4 BV Type Safety Invariant

theory *BVConform*
imports
BVSpec
../JVM/JVMExec
begin

context *JVM-heap-base* **begin**

definition *confT* :: '*c prog* ⇒ '*heap* ⇒ '*addr val* ⇒ *ty err* ⇒ *bool*
  (‹-,- ⊢ - :≤⊤ -› [*51*,*51*,*51*,*51*] *50*)
where
P,*h* ⊢ *v* :≤⊤ *E* ≡ *case E of Err* ⇒ *True* | *OK T* ⇒ *P*,*h* ⊢ *v* :≤ *T*

notation (*ASCII*)
  confT (‹-,- |− - :<=T -› [*51*,*51*,*51*,*51*] *50*)

abbreviation *confTs* :: *'c prog* ⇒ *'heap* ⇒ *'addr val list* ⇒ ty_l ⇒ *bool*
  (‹-,- ⊢ - [:≤⊤] -› [*51*,*51*,*51*,*51*] *50*)
where
  P,*h* ⊢ *vs* [:≤⊤] *Ts* ≡ *list-all2* (*confT P h*) *vs Ts*

notation (*ASCII*)
  confTs (‹-,- |− - [:<=T] -› [*51*,*51*,*51*,*51*] *50*)

definition *conf-f* :: *'addr jvm-prog* ⇒ *'heap* ⇒ ty_i ⇒ *'addr bytecode* ⇒ *'addr frame* ⇒ *bool*
where
  conf-f P h ≡ λ(*ST*,*LT*) *is* (*stk*,*loc*,*C*,*M*,*pc*). *P*,*h* ⊢ *stk* [:≤] *ST* ∧ *P*,*h* ⊢ *loc* [:≤⊤] *LT* ∧ *pc* < *size is*

primrec *conf-fs* :: [*'addr jvm-prog*,*'heap*,ty_P,*mname*,*nat*,*ty*,*'addr frame list*] ⇒ *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*
  (∃ *ST LT Ts T mxs* mxl_0 *is xt*.
    Φ *C M* ! *pc* = *Some* (*ST*,*LT*) ∧
    (*P* ⊢ *C sees M*:*Ts* → *T* = ⌊(*mxs*,mxl_0,*is*,*xt*)⌋ *in C*) ∧
    (∃ *Ts'* *T'* *D m D'*.
      is!*pc* = (*Invoke* M_0 n_0) ∧ *class-type-of'* (*ST*!n_0) = ⌊*D*⌋ ∧ *P* ⊢ *D sees* M_0:*Ts'* → *T'* = *m in D'*
∧ *P* ⊢ T_0 ≤ *T'*) ∧
    conf-f P h (*ST*, *LT*) *is f* ∧ *conf-fs P h* Φ *M* (*size Ts*) *T frs*))

primrec *conf-xcp* :: *'addr jvm-prog* ⇒ *'heap* ⇒ *'addr option* ⇒ *'addr instr* ⇒ *bool* **where**
  conf-xcp P h None i = *True*
| *conf-xcp P h* ⌊*a*⌋ *i* = (∃ *D*. *typeof-addr h a* = ⌊*Class-type D*⌋ ∧ *P* ⊢ *D* ⪯* *Throwable* ∧
        (∀ *D'*. *P* ⊢ *D* ⪯* *D'* ⟶ *is-relevant-class i P D'*))

end

context *JVM-heap-conf-base* **begin**

definition *correct-state* :: [ty_P,*'thread-id*,(*'addr*, *'heap*) *jvm-state*] ⇒ *bool*
where
  correct-state Φ *t* ≡ λ(*xp*,*h*,*frs*).
    P,*h* ⊢ *t* √*t* ∧ *hconf h* ∧ *preallocated h* ∧
    (*case frs of*
      [] ⇒ *True*
      | (*f*#*fs*) ⇒
      (*let* (*stk*,*loc*,*C*,*M*,*pc*) = *f*
       in ∃ *Ts T mxs* mxl_0 *is xt* τ.
         (*P* ⊢ *C sees M*:*Ts*→*T* = ⌊(*mxs*,mxl_0,*is*,*xt*)⌋ *in C*) ∧
         Φ *C M* ! *pc* = *Some* τ ∧
         conf-f P h τ *is f* ∧ *conf-fs P h* Φ *M* (*size Ts*) *T fs* ∧
         conf-xcp P h xp (*is* ! *pc*)))

notation

correct-state (‹- ⊢ -:- √› [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*) ≡
P,*h* ⊢ *stk* [:≤] *ST* ∧ *P*,*h* ⊢ *loc* [:≤⊤] *LT* ∧ *pc* < *size is*
⟨*proof*⟩

6.4.1 Values and ⊤

lemma *confT-Err* [*iff*]: *P*,*h* ⊢ *x* :≤⊤ *Err*
⟨*proof*⟩

lemma *confT-OK* [*iff*]: *P*,*h* ⊢ *x* :≤⊤ *OK T* = (*P*,*h* ⊢ *x* :≤ *T*)
⟨*proof*⟩

lemma *confT-cases*:
P,*h* ⊢ *x* :≤⊤ *X* = (*X* = *Err* ∨ (∃ *T*. *X* = *OK T* ∧ *P*,*h* ⊢ *x* :≤ *T*))
⟨*proof*⟩

lemma *confT-widen* [*intro?*, *trans*]:
⟦ *P*,*h* ⊢ *x* :≤⊤ *T*; *P* ⊢ *T* ≤⊤ *T′* ⟧ ⟹ *P*,*h* ⊢ *x* :≤⊤ *T′*
⟨*proof*⟩

end

context *JVM-heap* **begin**

lemma *confT-hext* [*intro?*, *trans*]:
⟦ *P*,*h* ⊢ *x* :≤⊤ *T*; *h* ⊴ *h′* ⟧ ⟹ *P*,*h′* ⊢ *x* :≤⊤ *T*
⟨*proof*⟩

end

6.4.2 Stack and Registers

context *JVM-heap-base* **begin**

lemma *confTs-Cons1* [*iff*]:
P,*h* ⊢ *x* # *xs* [:≤⊤] *Ts* = (∃ *z zs*. *Ts* = *z* # *zs* ∧ *P*,*h* ⊢ *x* :≤⊤ *z* ∧ *list-all2* (*confT P h*) *xs zs*)
⟨*proof*⟩

lemma *confTs-confT-sup*:
⟦ *P*,*h* ⊢ *loc* [:≤⊤] *LT*; *n* < *size LT*; *LT*!*n* = *OK T*; *P* ⊢ *T* ≤ *T′* ⟧
⟹ *P*,*h* ⊢ (*loc*!*n*) :≤ *T′*
⟨*proof*⟩

lemma *confTs-widen* [*intro?*, *trans*]:

$P,h \vdash loc\ [:\leq_\top]\ LT \Longrightarrow P \vdash LT\ [\leq_\top]\ LT' \Longrightarrow P,h \vdash loc\ [:\leq_\top]\ LT'$
⟨*proof*⟩

lemma *confTs-map* [*iff*]:
$(P,h \vdash vs\ [:\leq_\top]\ map\ OK\ Ts) = (P,h \vdash vs\ [:\leq]\ Ts)$
⟨*proof*⟩

lemma (**in** −) *reg-widen-Err*:
$(P \vdash replicate\ n\ Err\ [\leq_\top]\ LT) = (LT = replicate\ n\ Err)$
⟨*proof*⟩

declare *reg-widen-Err* [*iff*]

lemma *confTs-Err* [*iff*]:
$P,h \vdash replicate\ n\ v\ [:\leq_\top]\ replicate\ n\ Err$
⟨*proof*⟩

end

context *JVM-heap* **begin**

lemma *confTs-hext* [*intro?*]:
$P,h \vdash loc\ [:\leq_\top]\ LT \Longrightarrow h \unlhd h' \Longrightarrow P,h' \vdash loc\ [:\leq_\top]\ LT$
⟨*proof*⟩

6.4.3 correct-frames

declare *fun-upd-apply*[*simp del*]

lemma *conf-f-hext*:
$[\![\ conf\text{-}f\ P\ h\ \Phi\ M\ f;\ h \unlhd h'\ ]\!] \Longrightarrow conf\text{-}f\ P\ h'\ \Phi\ M\ f$
⟨*proof*⟩

lemma *conf-fs-hext*:
$[\![\ conf\text{-}fs\ P\ h\ \Phi\ M\ n\ T_r\ frs;\ h \unlhd h'\ ]\!] \Longrightarrow conf\text{-}fs\ P\ h'\ \Phi\ M\ n\ T_r\ frs$
⟨*proof*⟩

declare *fun-upd-apply*[*simp*]

lemma *conf-xcp-hext*:
$[\![\ conf\text{-}xcp\ P\ h\ xcp\ i;\ h \unlhd h'\ ]\!] \Longrightarrow conf\text{-}xcp\ P\ h'\ xcp\ i$
⟨*proof*⟩

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*:
$\Phi \vdash t\!: (None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$
$\Longrightarrow \exists\, m\ Ts\ T.\ P \vdash C\ sees\ M{:}Ts{\rightarrow}T = \lfloor m \rfloor\ in\ C$
⟨*proof*⟩

end

context *JVM-heap-conf-base'* **begin**

lemma *correct-state-hext-mono*:
$[\![\Phi \vdash t\colon (xcp,\ h,\ frs)\ \surd;\ h \trianglelefteq h';\ hconf\ h']\!] \Longrightarrow \Phi \vdash t\colon (xcp,\ h',\ frs)\ \surd$
⟨*proof*⟩

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*:
$match\text{-}ex\text{-}table\ P\ C\ pc\ xt = Some\ (pc',d') \Longrightarrow$
$\exists (f,t,D,h,d) \in set\ (relevant\text{-}entries\ P\ (Invoke\ n\ M)\ pc\ xt).$
$(case\ D\ of\ None \Rightarrow True \mid Some\ D' \Rightarrow P \vdash C \preceq^* D') \wedge pc \in \{f..<t\} \wedge pc' = h \wedge d' = d$
⟨*proof*⟩

lemma *match-is-relevant*:
assumes *rv*: $\bigwedge D'.\ P \vdash D \preceq^* D' \Longrightarrow is\text{-}relevant\text{-}class\ (ins\ !\ i)\ P\ D'$
assumes *match*: $match\text{-}ex\text{-}table\ P\ D\ pc\ xt = Some\ (pc',d')$
shows $\exists (f,t,D',h,d) \in set\ (relevant\text{-}entries\ P\ (ins\ !\ i)\ pc\ xt).\ (case\ D'\ of\ None \Rightarrow True \mid Some\ D'' \Rightarrow P \vdash D \preceq^* D'') \wedge pc \in \{f..<t\} \wedge pc' = h \wedge d' = d$
⟨*proof*⟩

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*: $\Phi \vdash t$:(⌊*xcp*⌋, *h*, *fr* # *frs*) $\surd$
  shows $\Phi \vdash t$:*exception-step P xcp h fr frs* $\surd$
⟨*proof*⟩

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*$_\Phi$ *P*
  assumes *meth-C*: $P \vdash C$ *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* $\vdash$ *ins*!*pc*,*pc* :: Φ *C M*
  assumes *approx*: $\Phi \vdash t$:(*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$
  assumes *exec*: (*tas*, σ) ∈ *exec-instr* (*ins*!*pc*) *P t h stk loc C M pc frs*
  shows $\Phi \vdash t$:σ $\surd$
⟨*proof*⟩

declare *list-all2-Cons2* [*iff*]

lemma *Return-correct*:
  assumes *wt-prog*: *wf-jvm-prog*$_\Phi$ *P*
  assumes *meth*: $P \vdash C$ *sees* M:*Ts*→T=⌊(*mxs*,mxl_0,*ins*,*xt*)⌋ *in C*
  assumes *ins*: *ins* ! *pc* = *Return*
  assumes *wt*: P,T,*mxs*,*size ins*,*xt* $\vdash$ *ins*!*pc*,*pc* :: Φ *C M*
  assumes *correct*: $\Phi \vdash t$:(*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$
  assumes *s'*: (*tas*, σ') ∈ *exec P t* (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)
  shows $\Phi \vdash t$:$\sigma'$$\surd$
⟨*proof*⟩

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*→T=⌊(*mxs*,mxl_0,*ins*,*xt*)⌋ *in C*;
  ins!*pc* = *Load idx*;
  P,T,*mxs*,*size ins*,*xt* $\vdash$ *ins*!*pc*,*pc* :: Φ *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' $\surd$

⟨*proof*⟩

declare [[*simproc del*: *list-to-set-comprehension*]]

lemma *Store-correct*:
$[\![$ *wf-prog wt P*;
$P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}\lfloor(mxs,mxl_0,ins,xt)\rfloor$ *in* C;
$ins!pc$ = *Store idx*;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
$\Phi \vdash t{:}(None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$;
$(tas,\ \sigma') \in exec\ P\ t\ (None,\ h,\ (stk,loc,C,M,pc)\#frs)$ $]\!]$
$\Longrightarrow \Phi \vdash t{:}\sigma'\surd$
⟨*proof*⟩

lemma *Push-correct*:
$[\![$ *wf-prog wt P*;
$P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}\lfloor(mxs,mxl_0,ins,xt)\rfloor$ *in* C;
$ins!pc$ = *Push v*;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
$\Phi \vdash t{:}(None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$;
$(tas,\ \sigma') \in exec\ P\ t\ (None,\ h,\ (stk,loc,C,M,pc)\#frs)$ $]\!]$
$\Longrightarrow \Phi \vdash t{:}\sigma'\surd$
⟨*proof*⟩

declare [[*simproc add*: *list-to-set-comprehension*]]

lemma *Checkcast-correct*:
$[\![$ $wf\text{-}jvm\text{-}prog_{\Phi}$ P;
$P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}\lfloor(mxs,mxl_0,ins,xt)\rfloor$ *in* C;
$ins!pc$ = *Checkcast D*;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
$\Phi \vdash t{:}(None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$;
$(tas,\ \sigma) \in exec\text{-}instr\ (ins!pc)\ P\ t\ h\ stk\ loc\ C\ M\ pc\ frs$ $]\!]$
$\Longrightarrow \Phi \vdash t{:}\sigma\ \surd$
⟨*proof*⟩

lemma *Instanceof-correct*:
$[\![$ $wf\text{-}jvm\text{-}prog_{\Phi}$ P;
$P \vdash C$ *sees* $M{:}Ts{\rightarrow}T{=}\lfloor(mxs,mxl_0,ins,xt)\rfloor$ *in* C;
$ins!pc$ = *Instanceof Ty*;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
$\Phi \vdash t{:}(None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$;
$(tas,\ \sigma) \in exec\text{-}instr\ (ins!pc)\ P\ t\ h\ stk\ loc\ C\ M\ pc\ frs$ $]\!]$
$\Longrightarrow \Phi \vdash t{:}\sigma\ \surd$
⟨*proof*⟩

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

end

lemma *widens-Cons* [*iff*]:
$P \vdash (T\ \#\ Ts)\ [\leq]\ Us = (\exists z\ zs.\ Us = z\ \#\ zs \wedge P \vdash T \leq z \wedge P \vdash Ts\ [\leq]\ zs)$
⟨*proof*⟩

context *heap-conf-base* **begin**

end

context *JVM-conf-read* **begin**

lemma *Getfield-correct*:
assumes *wf*: *wf-prog wt P*
assumes *mC*: $P \vdash C$ *sees* M:$Ts \rightarrow T = \lfloor(mxs, mxl_0, ins, xt)\rfloor$ *in* C
assumes *i*: *ins*!*pc* = *Getfield 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*, σ') $\in$ *exec-instr* (*ins*!*pc*) *P t h stk loc C M pc frs*

shows $\Phi \vdash t$:$\sigma'\surd$
⟨*proof*⟩

lemma *Putfield-correct*:
assumes *wf*: *wf-prog wt P*
assumes *mC*: $P \vdash C$ *sees* M:$Ts \rightarrow T = \lfloor(mxs, mxl_0, ins, xt)\rfloor$ *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*, σ') $\in$ *exec-instr* (*ins*!*pc*) *P t h stk loc C M pc frs*
shows $\Phi \vdash t$:σ' $\surd$
⟨*proof*⟩

lemma *CAS-correct*:
assumes *wf*: *wf-prog wt P*
assumes *mC*: $P \vdash C$ *sees* M:$Ts \rightarrow T = \lfloor(mxs, mxl_0, ins, xt)\rfloor$ *in* C
assumes *i*: *ins*!*pc* = *CAS 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*, σ') $\in$ *exec-instr* (*ins*!*pc*) *P t h stk loc C M pc frs*
shows $\Phi \vdash t$:σ' $\surd$
⟨*proof*⟩

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 $\Phi \vdash t$:σ $\surd$
⟨*proof*⟩

lemma *Goto-correct*:
⟦ *wf-prog wt P*;
$P \vdash C$ *sees* M:$Ts \rightarrow T = \lfloor(mxs, mxl_0, ins, xt)\rfloor$ *in* C;
ins ! *pc* = *Goto branch*;
$P, T, mxs, size\ ins, xt \vdash ins!pc, pc :: \Phi\ C\ M$;
$\Phi \vdash t$:(*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$;

$(tas, \sigma') \in exec\ P\ t\ (None,\ h,\ (stk,loc,C,M,pc)\#frs)$ ⟧
$\Longrightarrow \Phi \vdash t{:}\sigma'\ \surd$
⟨proof⟩

declare [[*simproc del*: *list-to-set-comprehension*]]

lemma *IfFalse-correct*:
⟦ $wf\text{-}prog\ wt\ P$;
$P \vdash C\ sees\ M{:}Ts{\rightarrow}T{=}\lfloor(mxs,mxl_0,ins,xt)\rfloor\ in\ C$;
$ins\ !\ pc = IfFalse\ branch$;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
$\Phi \vdash t{:}(None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$;
$(tas, \sigma') \in exec\ P\ t\ (None,\ h,\ (stk,loc,C,M,pc)\#frs)$ ⟧
$\Longrightarrow \Phi \vdash t{:}\sigma'\surd$
⟨proof⟩

declare [[*simproc add*: *list-to-set-comprehension*]]

lemma *BinOp-correct*:
⟦ $wf\text{-}prog\ wt\ P$;
$P \vdash C\ sees\ M{:}Ts{\rightarrow}T{=}\lfloor(mxs,mxl_0,ins,xt)\rfloor\ in\ C$;
$ins\ !\ pc = BinOpInstr\ bop$;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
$\Phi \vdash t{:}(None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$;
$(tas, \sigma') \in exec\ P\ t\ (None,\ h,\ (stk,loc,C,M,pc)\#frs)$ ⟧
$\Longrightarrow \Phi \vdash t{:}\sigma'\surd$
⟨proof⟩

lemma *Pop-correct*:
⟦ $wf\text{-}prog\ wt\ P$;
$P \vdash C\ sees\ M{:}Ts{\rightarrow}T{=}\lfloor(mxs,mxl_0,ins,xt)\rfloor\ in\ C$;
$ins\ !\ pc = Pop$;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
$\Phi \vdash t{:}(None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$;
$(tas, \sigma') \in exec\ P\ t\ (None,\ h,\ (stk,loc,C,M,pc)\#frs)$ ⟧
$\Longrightarrow \Phi \vdash t{:}\sigma'\surd$
⟨proof⟩

lemma *Dup-correct*:
⟦ $wf\text{-}prog\ wt\ P$;
$P \vdash C\ sees\ M{:}Ts{\rightarrow}T{=}\lfloor(mxs,mxl_0,ins,xt)\rfloor\ in\ C$;
$ins\ !\ pc = Dup$;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
$\Phi \vdash t{:}(None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$;
$(tas, \sigma') \in exec\ P\ t\ (None,\ h,\ (stk,loc,C,M,pc)\#frs)$ ⟧
$\Longrightarrow \Phi \vdash t{:}\sigma'\surd$
⟨proof⟩

lemma *Swap-correct*:
⟦ $wf\text{-}prog\ wt\ P$;
$P \vdash C\ sees\ M{:}Ts{\rightarrow}T{=}\lfloor(mxs,mxl_0,ins,xt)\rfloor\ in\ C$;
$ins\ !\ pc = Swap$;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
$\Phi \vdash t{:}(None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$;

$(tas,\ \sigma') \in exec\ P\ t\ (None,\ h,\ (stk,loc,C,M,pc)\#frs)$ ⟧
$\Longrightarrow \Phi \vdash t{:}\sigma'\surd$
⟨*proof*⟩

declare [[*simproc del*: *list-to-set-comprehension*]]

lemma *Throw-correct*:
⟦ *wf-prog wt P*;
$P \vdash C\ sees\ M{:}Ts{\rightarrow}T{=}\lfloor(mxs,mxl_0,ins,xt)\rfloor\ in\ C$;
$ins\ !\ pc = ThrowExc$;
$P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$;
$\Phi \vdash t{:}(None,\ h,\ (stk,loc,C,M,pc)\#frs)\surd$;
$(tas,\ \sigma') \in exec\text{-}instr\ (ins!pc)\ P\ t\ h\ stk\ loc\ C\ M\ pc\ frs$ ⟧
$\Longrightarrow \Phi \vdash t{:}\sigma'\surd$
⟨*proof*⟩

declare [[*simproc add*: *list-to-set-comprehension*]]

lemma *NewArray-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*: $ins!pc = NewArray\ X$
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*: $(tas,\ \sigma) \in exec\text{-}instr\ (ins!pc)\ P\ t\ h\ stk\ loc\ C\ M\ pc\ frs$
shows $\Phi \vdash t{:}\sigma\ \surd$
⟨*proof*⟩

lemma *ALoad-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*: $ins!pc = ALoad$
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*: $(tas,\ \sigma) \in exec\text{-}instr\ (ins!pc)\ P\ t\ h\ stk\ loc\ C\ M\ pc\ frs$
shows $\Phi \vdash t{:}\sigma\ \surd$
⟨*proof*⟩

lemma *AStore-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*: $ins!pc = AStore$
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*: $(tas,\ \sigma) \in exec\text{-}instr\ (ins!pc)\ P\ t\ h\ stk\ loc\ C\ M\ pc\ frs$
shows $\Phi \vdash t{:}\ \sigma\ \surd$
⟨*proof*⟩

lemma *ALength-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*: $ins!pc = ALength$
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 $\Phi \vdash t$: σ $\surd$
⟨*proof*⟩

lemma *MEnter-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* = *MEnter*
assumes *wt*: *P*,*T*,*mxs*,*size ins*,*xt* $\vdash$ *ins*!*pc*,*pc* :: Φ *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 $\Phi \vdash t$: σ $\surd$
⟨*proof*⟩

lemma *MExit-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* = *MExit*
assumes *wt*: *P*,*T*,*mxs*,*size ins*,*xt* $\vdash$ *ins*!*pc*,*pc* :: Φ *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 $\Phi \vdash t$: σ $\surd$
⟨*proof*⟩

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*$_\Phi$ *P*;
$P \vdash C$ *sees* M:*Ts*$\rightarrow$*T*=$\lfloor$(*mxs*,mxl_0,*ins*,*xt*)$\rfloor$ *in C*;
(*tas*, σ') $\in$ *exec P t* (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*);
$\Phi \vdash t$: (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*)#*frs*)$\surd$ ⟧
$\Longrightarrow \Phi \vdash t$: $\sigma'\surd$
⟨*proof*⟩

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*→ σ' ⟶ $\Phi \vdash t$: $\sigma'\surd$
⟨*proof*⟩

theorem (**in** *JVM-progress*) *progress*:
assumes *wt*: *wf-jvm-prog*$_\Phi$ *P*
and *cs*: $\Phi \vdash t$: (*xcp*, *h*, *f* # *frs*)$\surd$
shows $\exists$ *ta* σ'. *P*,*t* $\vdash$ (*xcp*, *h*, *f* # *frs*) −*ta*−*jvm*→ σ'
⟨*proof*⟩

lemma (**in** *JVM-heap-conf*) *BV-correct-initial*:
shows $[\![$ *wf-jvm-prog*$_\Phi$ *P*; *start-heap-ok*; $P \vdash C$ *sees* $M{:}Ts{\rightarrow}T = \lfloor m \rfloor$ *in D*; *P*,*start-heap* $\vdash$ *vs* $[:\leq]$ *Ts* $]\!]$
$\Longrightarrow \Phi \vdash$ *start-tid*:*JVM-start-state'* *P C M vs* $\surd$
⟨*proof*⟩

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*; $P \vdash C$ *sees* M: $Ts{\rightarrow}T = \lfloor(mxs, mxl, ins, et)\rfloor$ *in C*;
Φ *C M* ! $pc = \tau$; $pc <$ *size ins* $]\!]$
$\Longrightarrow$ *OK* $\tau \in$ *states P mxs* (*1*+*size Ts*+*mxl*)⟨*proof*⟩
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*: $\Phi \vdash t{:}\sigma\ \surd$
shows *exec-d P t* $\sigma \neq$ *TypeError*⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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*; $\Phi \vdash t$: (*xcp*, *h*, *frs*) $\surd$ $]\!] \Longrightarrow$ *mexec P t* ((*xcp*, *frs*), *h*) = *mexecd P t* ((*xcp*, *frs*), *h*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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 $P,T,mxs,size\ ins,xt \vdash ins!pc,pc :: \Phi\ C\ M$
and $\Phi \vdash t$: (*None*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs*) $\surd$
and *typeof-addr h* (*thread-id2addr a*) = ⌊*Class-type D*⌋
and $P \vdash D \preceq^* Thread$
and $P \vdash D$ *sees run*:[]→*Void*=⌊(*mxs'*, *mxl0'*, *ins'*,*xt'*)⌋ *in D'*
shows $\Phi \vdash a$: (*None*, *h*, [([], *Addr* (*thread-id2addr a*) # *replicate mxl0' undefined-value*, *D'*, *run*, *0*)]) $\surd$
⟨*proof*⟩

lemma *exec-new-threadE*:
assumes *wf-jvm-prog*$_\Phi$ *P*
and $P,t \vdash$ *Normal* σ −*ta*−*jvmd*→ *Normal* σ'
and $\Phi \vdash t$: σ $\surd$
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 $P \vdash Us\ [\leq]\ Us'$
⟨*proof*⟩

end

context *JVM-conf-read* **begin**

lemma *correct-state-new-thread*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *red*: $P,t \vdash$ *Normal* σ −*ta*−*jvmd*→ *Normal* σ'
and *cs*: $\Phi \vdash t$: σ $\surd$
and *nt*: *NewThread t''* (*xcp*, *frs*) *h''* ∈ *set* $\{\!|ta|\!\}_t$
shows $\Phi \vdash t''$: (*xcp*, *h''*, *frs*) $\surd$
⟨*proof*⟩

lemma *correct-state-heap-change*:
assumes *wf*: $\textit{wf-jvm-prog}_\Phi$ *P*
and *red*: *P*,*t* ⊢ *Normal* (*xcp*, *h*, *frs*) −*ta*−*jvmd*→ *Normal* (*xcp*′, *h*′, *frs*′)
and *cs*: Φ ⊢ *t*: (*xcp*, *h*, *frs*) √
and *cs*′′: Φ ⊢ *t*′′: (*xcp*′′, *h*, *frs*′′) √
shows Φ ⊢ *t*′′: (*xcp*′′, *h*′, *frs*′′) √
⟨*proof*⟩

lemma *lifting-wf-correct-state-d*:
$\textit{wf-jvm-prog}_\Phi$ *P* ⟹ *lifting-wf JVM-final* (*mexecd P*) (λ*t* (*xcp*, *frs*) *h*. Φ ⊢ *t*: (*xcp*, *h*, *frs*) √)
⟨*proof*⟩

lemma *lifting-wf-correct-state*:
assumes *wf*: $\textit{wf-jvm-prog}_\Phi$ *P*
shows *lifting-wf JVM-final* (*mexec P*) (λ*t* (*xcp*, *frs*) *h*. Φ ⊢ *t*: (*xcp*, *h*, *frs*) √)
⟨*proof*⟩

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*: $\textit{wf-jvm-prog}_\Phi$ *P*
and *wf-start*: *wf-start-state P C M vs*
shows *JVM-start-state P C M vs* ∈ *correct-jvm-state* Φ
⟨*proof*⟩

end

context *JVM-conf-read* **begin**

lemma *invariant3p-correct-jvm-state-mexecdT*:
assumes *wf*: $\textit{wf-jvm-prog}_\Phi$ *P*
shows *invariant3p* (*mexecdT P*) (*correct-jvm-state* Φ)
⟨*proof*⟩

lemma *invariant3p-correct-jvm-state-mexecT*:
assumes *wf*: $\textit{wf-jvm-prog}_\Phi$ *P*
shows *invariant3p* (*mexecT P*) (*correct-jvm-state* Φ)
⟨*proof*⟩

lemma *correct-jvm-state-preserved*:

assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *correct*: *s* ∈ *correct-jvm-state* Φ
and *red*: *P* ⊢ *s* −▷*ttas*→$_{jvm}$∗ *s*′
shows *s*′ ∈ *correct-jvm-state* Φ
⟨*proof*⟩

theorem *jvm-typesafe*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *start*: *wf-start-state P C M vs*
and *exec*: *P* ⊢ *JVM-start-state P C M vs* −▷*ttas*→$_{jvm}$∗ *s*′
shows *s*′ ∈ *correct-jvm-state* Φ
⟨*proof*⟩

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*: Φ ⊢ *t*′: σ √
and *red*: *P*,*t*′ ⊢ *Normal* σ −*ta*−*jvmd*→ *Normal* σ′
and *nt*: *NewThread t x m* ∈ *set* ⦃*ta*⦄$_t$
shows ∃ *C*. *typeof-addr* (*fst* (*snd* σ′)) (*thread-id2addr t*) = ⌊*Class-type C*⌋ ∧ *P* ⊢ *Class C* ≤ *Class Thread*
⟨*proof*⟩

lemma *mexecdT-NewThread-Thread-Object*:
⟦ *wf-jvm-prog*$_\Phi$ *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*
⟨*proof*⟩

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*⟩

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*⟩

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 $P \vdash s -t\triangleright ta\rightarrow_{jvm} s' = P \vdash s -t\triangleright ta\rightarrow_{jvmd} s'$
⟨*proof*⟩

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*⟩

lemma *mexecT-preserves-thread-conf*:
⟦ *wf-jvm-prog*$_\Phi$ *P*; *correct-state-ts* Φ (*thr s*) (*shr s*);
$P \vdash s -t'\triangleright ta\rightarrow_{jvm} s'$; *thread-conf P* (*thr s*) (*shr s*) ⟧
$\Longrightarrow$ *thread-conf P* (*thr s'*) (*shr s'*)
⟨*proof*⟩

lemma *mExecT-preserves-thread-conf*:
⟦ *wf-jvm-prog*$_\Phi$ *P*; *correct-state-ts* Φ (*thr s*) (*shr s*);
$P \vdash s -\triangleright tta\rightarrow_{jvm}* s'$; *thread-conf P* (*thr s*) (*shr s*) ⟧
$\Longrightarrow$ *thread-conf P* (*thr s'*) (*shr s'*)
⟨*proof*⟩

lemma *wset-Suspend-ok-mexecd-mexec*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
shows *exec-mthr.wset-Suspend-ok P* (*correct-jvm-state* Φ) = *execd-mthr.wset-Suspend-ok P* (*correct-jvm-state* Φ)
⟨*proof*⟩

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*⟩

theorem *mexecd-TypeSafety*:
fixes *ln* :: *'addr* ⇒*f nat*
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *s*: $s \in$ *execd-mthr.wset-Suspend-ok P* (*correct-jvm-state* Φ)
and *Exec*: $P \vdash s -\triangleright ttas\rightarrow_{jvmd}* s'$
and $\neg$ ($\exists$ *t ta s''*. $P \vdash s' -t\triangleright ta\rightarrow_{jvmd} s''$)
and *ts't*: *thr s' t* = ⌊((*xcp*, *frs*), *ln*)⌋
shows *frs* ≠ [] ∨ *ln* ≠ *no-wait-locks* $\Longrightarrow$ *t* ∈ *execd-mthr.deadlocked P s'*
and $\Phi \vdash t$: (*xcp*, *shr s'*, *frs*) $\surd$
⟨*proof*⟩

theorem *mexec-TypeSafety*:
fixes *ln* :: *'addr ⇒f nat*
assumes *wf*: $wf\text{-}jvm\text{-}prog_{\Phi}$ *P*
and *s*: *s* ∈ *exec-mthr.wset-Suspend-ok P* (*correct-jvm-state* Φ)
and *Exec*: $P \vdash s -\triangleright ttas\rightarrow_{jvm}* s'$
and $\neg (\exists t\ ta\ s''.\ P \vdash s' -t\triangleright ta\rightarrow_{jvm} s'')$
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*⟩

lemma *start-mexec-mexecd-commute*:
assumes *wf*: $wf\text{-}jvm\text{-}prog_{\Phi}$ *P*
and *start*: *wf-start-state P C M vs*
shows $P \vdash$ *JVM-start-state P C M vs* $-\triangleright ttas\rightarrow_{jvmd}* s \longleftrightarrow P \vdash$ *JVM-start-state P C M vs* $-\triangleright ttas\rightarrow_{jvm}* s$
⟨*proof*⟩

theorem *mRtrancl-eq-mRtrancld*:
assumes *wf*: $wf\text{-}jvm\text{-}prog_{\Phi}$ *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*⟩

lemma *start-mRtrancl-mRtrancld-commute*:
assumes *wf*: $wf\text{-}jvm\text{-}prog_{\Phi}$ *P*
and *start*: *wf-start-state P C M vs*
shows *exec-mthr.mthr.Rtrancl3p P* (*JVM-start-state P C M vs*) *ttas* ⟷ *execd-mthr.mthr.Rtrancl3p P* (*JVM-start-state P C M vs*) *ttas*
⟨*proof*⟩

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''
⟨*proof*⟩

lemma *exec-1-deterministic*:
assumes *wf*: $wf\text{-}jvm\text{-}prog_{\Phi}$ *P*
and *det*: *deterministic-heap-ops*
and *exec1*: *P*,*t* ⊢ (*xcp*, *shr s*, *frs*) −*ta'*−*jvm*→ σ'

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*⟩

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*)}
⟨*proof*⟩

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*⟩

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*: $\Phi \vdash t{:}\ (xcp,\ h,\ frs)\ \surd$
shows *execd-mthr.must-sync P t* (*xcp*, *frs*) *h′*
⟨*proof*⟩

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*⟩

end

context *JVM-typesafe* **begin**

lemma *execd-preserve-deadlocked*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
shows *preserve-deadlocked JVM-final* (*mexecd P*) *convert-RA* (*correct-jvm-state* Φ)
⟨*proof*⟩

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*$_\Phi$ *P*; Φ ⊢ *t*: (*xcp*, *h*, *frs*) √ ⟧
⟹ *execd-mthr.must-sync P t* (*xcp*, *frs*) *h* = *exec-mthr.must-sync P t* (*xcp*, *frs*) *h*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

lemma *succs-mono*:
assumes *wf*: *wf-prog p P* **and** app_i: app_i (i,P,mxs,mpc,rT,τ')
shows $P \vdash \tau \leq_i \tau' \Longrightarrow set\ (succs\ i\ \tau\ pc) \subseteq set\ (succs\ i\ \tau'\ pc)$
⟨*proof*⟩

lemma *app-mono*:
assumes *wf*: *wf-prog p P*
assumes *less'*: $P \vdash \tau \leq' \tau'$
shows $app\ i\ P\ m\ rT\ pc\ mpc\ xt\ \tau' \Longrightarrow app\ i\ P\ m\ rT\ pc\ mpc\ xt\ \tau$
⟨*proof*⟩

lemma eff_i*-mono*:
assumes *wf*: *wf-prog p P*
assumes *less*: $P \vdash \tau \leq_i \tau'$
assumes app_i: $app\ i\ P\ m\ rT\ pc\ mpc\ xt\ (Some\ \tau')$
assumes *succs*: $succs\ i\ \tau\ pc \neq []$ $\ succs\ i\ \tau'\ pc \neq []$
shows $P \vdash eff_i\ (i,P,\tau) \leq_i eff_i\ (i,P,\tau')$
⟨*proof*⟩

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*) (λ*pc. app* (*bs*!*pc*) *G maxs rT pc* (*size bs*) *et*) (λ*pc. 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* ≡ λ*pc. Effect.app* (*is*!*pc*) *P mxs* T_r *pc* (*size is*) *xt*
defines [*simp*]: *eff* ≡ λ*pc. 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* $\subseteq$ *types P*

fixes *first* :: $ty_i{}'$ **and** *start*
defines [*simp*]:
first $\equiv$ *Some* ([],*OK* (*Class C*) # *map OK Ts* @ *replicate* mxl_0 *Err*)
defines [*simp*]:
start $\equiv$ *OK first* # *replicate* (*size is* − *1*) (*OK None*)

6.10.1 Connecting JVM and Framework

lemma (**in** *JVM-sl*) *step-def-exec*: *step* $\equiv$ *exec P mxs* T_r *xt is*
⟨*proof*⟩

lemma *special-ex-swap-lemma* [*iff*]:
$(\exists X.\ (\exists n.\ X = A\ n \wedge P\ n) \wedge Q\ X) = (\exists n.\ Q(A\ n) \wedge P\ n)$
⟨*proof*⟩

lemma *ex-in-list* [*iff*]:
$(\exists n.\ ST \in list\ n\ A \wedge n \leq mxs) = (set\ ST \subseteq A \wedge size\ ST \leq mxs)$
⟨*proof*⟩

lemma *singleton-list*:
$(\exists n.\ [Class\ C] \in list\ n\ (types\ P) \wedge n \leq mxs) = (is\text{-}class\ P\ C \wedge 0 < mxs)$
⟨*proof*⟩

lemma *set-drop-subset*:
$set\ xs \subseteq A \Longrightarrow set\ (drop\ n\ xs) \subseteq A$
⟨*proof*⟩

lemma *Suc-minus-minus-le*:
$n < mxs \Longrightarrow Suc\ (n - (n - b)) \leq mxs$
⟨*proof*⟩

lemma *in-listE*:
$[\![\ xs \in list\ n\ A;\ [\![size\ xs = n;\ set\ xs \subseteq A]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
⟨*proof*⟩

declare *is-relevant-entry-def* [*simp*]
declare *set-drop-subset* [*simp*]

lemma (**in** *start-context*) [*simp*, *intro*!]: *is-class P Throwable*
⟨*proof*⟩

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*⟨*proof*⟩
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*:
$xs\ [\sqsubseteq_{Product.le\ (=)\ r}]\ ys \Longrightarrow set\ xs\ \{\sqsubseteq_r\}\ set\ ys\langle proof\rangle$
declare *is-relevant-entry-def* [*simp del*]

lemma *conjI2*: $[\![\ A;\ A \Longrightarrow B\ ]\!] \Longrightarrow A \wedge B\ \langle proof\rangle$

lemma (**in** *JVM-sl*) *eff-mono*:
$[\![wf\text{-}prog\ p\ P;\ pc < length\ is;\ s \sqsubseteq_{sup\text{-}state\text{-}opt\ P}\ t;\ app\ pc\ t]\!]$
$\Longrightarrow set\ (eff\ pc\ s)\ \{\sqsubseteq_{sup\text{-}state\text{-}opt\ P}\}\ set\ (eff\ pc\ t)\langle proof\rangle$
lemma (**in** *JVM-sl*) *bounded-step*: *bounded step* (*size is*)⟨*proof*⟩
theorem (**in** *JVM-sl*) *step-mono*:
wf-prog wf-mb P ⟹ *mono r step* (*size is*) *A*⟨*proof*⟩

lemma (**in** *start-context*) *first-in-A* [*iff*]: *OK first* ∈ *A*
⟨*proof*⟩

lemma (**in** *JVM-sl*) *wt-method-def2*:
$wt\text{-}method\ P\ C'\ Ts\ T_r\ mxs\ mxl_0\ is\ xt\ \tau s =$
$(is \neq [] \wedge$
$size\ \tau s = size\ is \wedge$
$OK\ \text{`}\ set\ \tau s \subseteq states\ P\ mxs\ mxl \wedge$
$wt\text{-}start\ P\ C'\ Ts\ mxl_0\ \tau s \wedge$
$wt\text{-}app\text{-}eff\ (sup\text{-}state\text{-}opt\ P)\ app\ eff\ \tau s)\langle proof\rangle$

end

6.11 LBV for the JVM

theory *LBVJVM*
imports
../DFA/Abstract-BV
TF-JVM
begin

type-synonym $prog\text{-}cert = cname \Rightarrow mname \Rightarrow ty_i'\ err\ list$

definition $check\text{-}cert :: 'addr\ jvm\text{-}prog \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ list \Rightarrow bool$
where
$check\text{-}cert\ P\ mxs\ mxl\ n\ cert \equiv check\text{-}types\ P\ mxs\ mxl\ cert \wedge size\ cert = n{+}1 \wedge$
$(\forall\, i{<}n.\ cert!i \neq Err) \wedge cert!n = OK\ None$

definition $lbvjvm :: 'addr\ jvm\text{-}prog \Rightarrow nat \Rightarrow nat \Rightarrow ty \Rightarrow ex\text{-}table \Rightarrow$
$ty_i'\ err\ list \Rightarrow 'addr\ instr\ list \Rightarrow ty_i'\ err \Rightarrow ty_i'\ err$
where
$lbvjvm\ P\ mxs\ maxr\ T_r\ et\ cert\ bs \equiv$
$wtl\text{-}inst\text{-}list\ bs\ cert\ (JVM\text{-}SemiType.sup\ P\ mxs\ maxr)\ (JVM\text{-}SemiType.le\ P\ mxs\ maxr)\ Err\ (OK\ None)\ (exec\ P\ mxs\ T_r\ et\ bs)\ 0$

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*)
⟨*proof*⟩

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* $\sqsubseteq_r$ τ*s*!*0*⟨*proof*⟩

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*⟨*proof*⟩

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*⟨*proof*⟩

theorem *jvm-lbv-correct*:
wt-jvm-prog-lbv P Cert ⟹ *wf-jvm-prog P*⟨*proof*⟩
theorem *jvm-lbv-complete*:
assumes *wt*: *wf-jvm-prog*$_\Phi$ *P*
shows *wt-jvm-prog-lbv P* (*prg-cert P* Φ)⟨*proof*⟩
end

6.12 Kildall for the JVM

theory *BVExec*
imports
../DFA/Abstract-BV
TF-JVM
begin

definition *kiljvm* :: *'addr jvm-prog* ⇒ *nat* ⇒ *nat* ⇒ *ty* ⇒
'addr instr list ⇒ *ex-table* ⇒ ty_i*' err list* ⇒ ty_i*' err list*
where
kiljvm P mxs mxl T_r *is xt* ≡
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* ⇒ *cname* ⇒ *ty list* ⇒ *ty* ⇒ *nat* ⇒ *nat* ⇒
'addr instr list ⇒ *ex-table* ⇒ *bool*
where
wt-kildall P C' Ts T_r *mxs* mxl_0 *is xt* ≡
0 < *size is* ∧
(*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 ∀ *n* < *size is*. *result*!*n* ≠ *Err*)

definition *wf-jvm-prog*$_k$ :: *'addr jvm-prog* ⇒ *bool*
where
wf-jvm-prog$_k$ *P* ≡
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*)⟨*proof*⟩

lemma *subset-replicate* [*intro?*]: *set* (*replicate n x*) ⊆ {*x*}
⟨*proof*⟩

lemma *in-set-replicate*:
assumes *x* ∈ *set* (*replicate n y*)
shows *x* = *y*⟨*proof*⟩
lemma (**in** *start-context*) *start-in-A* [*intro?*]:
0 < *size is* ⟹ *start* ∈ *list* (*size is*) *A*
⟨*proof*⟩

theorem (**in** *start-context*) *wt-kil-correct*:
assumes *wtk*: *wt-kildall P C Ts* T_r *mxs* mxl_0 *is xt*
shows ∃τ*s*. *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* τ*s*⟨*proof*⟩

theorem (**in** *start-context*) *wt-kil-complete*:
assumes *wtm*: *wt-method P C Ts* T_r *mxs* mxl_0 *is xt* τ*s*
shows *wt-kildall P C Ts* T_r *mxs* mxl_0 *is xt*⟨*proof*⟩

theorem *jvm-kildall-correct*:
wf-jvm-prog$_k$ *P* = *wf-jvm-prog P*⟨*proof*⟩

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 ($\lambda x.\ x \in A$))
⟨*proof*⟩

lemma *list-code* [*code*]:
list n A = {*xs. size xs = n* ∧ *list-all* ($\lambda x.\ x \in A$) *xs*}
⟨*proof*⟩

lemma *opt-code* [*code*]:
opt A = Collect (case-option True ($\lambda x.\ x \in A$))
⟨*proof*⟩

lemma *Times-code* [*code-unfold*]:
Sigma A (%-. *B*) = {(*a*, *b*). *a* ∈ *A* ∧ *b* ∈ *B*}
⟨*proof*⟩

lemma *upto-esl-code* [*code*]:
upto-esl m (*A*, *r*, *f*) = (*Union* (($\lambda n.$ *list n A*) ‘ {..*m*}), *Listn.le r*, *Listn.sup f*)
⟨*proof*⟩

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*))))
⟨*proof*⟩

declare *sup-fun-def* [*code*]

lemma [*code*]: *states P mxs mxl* = *fst*(*sl P mxs mxl*)
⟨*proof*⟩

lemma *check-types-code* [*code*]:
check-types P mxs mxl τs = (*list-all* ($\lambda x.\ x \in$ (*states P mxs mxl*)) τs)
⟨*proof*⟩

lemma *wf-jvm-prog-code* [*code-unfold*]:

wf-jvm-prog = $wf\text{-}jvm\text{-}prog_k$
⟨*proof*⟩

definition *wf-jvm-prog'* = *wf-jvm-prog*

⟨*ML*⟩

end
theory *BV-Main*
imports
JVMDeadlocked
LBVJVM
BCVExec
begin

end

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*
⟨*proof*⟩

lemma *inline-call-eq-Val-aux*:
inline-call e E = *Val v* ⟹ *call E* = ⌊*aMvs*⌋ ⟹ *e* = *Val v*
⟨*proof*⟩

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* = []
⟨*proof*⟩

lemma *inline-calls-map-Val* [*simp*]: *inline-calls e* (*map Val vs*) = *map Val vs*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Throw-eq-inline-call-eq* [*dest*]:
inline-call e E = *Throw a* ⟹ *call E* = ⌊*aMvs*⌋ ⟹ *Throw a* = *e* ∨ *addr a* = *e*
⟨*proof*⟩

lemma *is-vals-inline-calls* [*dest*]:
⟦ *is-vals* (*inline-calls e es*); *calls es* = ⌊*aMvs*⌋ ⟧ ⟹ *is-val e*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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′*
⟨*proof*⟩

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′*
⟨*proof*⟩

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′*
⟨*proof*⟩

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′*
⟨*proof*⟩

lemma *collapse-append* [*simp*]:
collapse (*e*, *es* @ *es′*) = *collapse* (*collapse* (*e*, *es*), *es′*)
⟨*proof*⟩

lemma *collapse-conv-foldl*:
collapse (*e*, *es*) = *foldl inline-call e es*
⟨*proof*⟩

lemma *fv-collapse*: ∀ *e* ∈ *set es*. *is-call e* ⟹ *fv* (*collapse* (*e*, *es*)) ⊆ *fvs* (*e* # *es*)
⟨*proof*⟩

lemma *final-inline-callD*: ⟦ *final* (*inline-call E e*); *is-call e* ⟧ ⟹ *final E*
⟨*proof*⟩

lemma *collapse-finalD*: ⟦ *final* (*collapse* (*e*, *es*)); ∀ *e*∈*set es*. *is-call e* ⟧ ⟹ *final e*
⟨*proof*⟩

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)
⟨*proof*⟩

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

⟨*ML*⟩
typ *('addr,'thread-id,'heap) J0-thread-action*

⟨*ML*⟩
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)))))}, [])
⟨*proof*⟩

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*⦄$_o$
⟨*proof*⟩

lemma *extTA2J0-ε*: *extTA2J0 P ε = ε*
⟨*proof*⟩

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*: $extTA,P,t \vdash \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle \Longrightarrow fvs\ es' \subseteq fvs\ es$
$\langle proof\rangle$

end

declare *domIff*[*iff del*]

context *J-heap-base* **begin**

lemma assumes *wwf*: *wwf-J-prog P*
shows *red-fv-ok*: $\llbracket\ extTA,P,t \vdash \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle;\ fv\ e \subseteq dom\ (lcl\ s)\ \rrbracket \Longrightarrow fv\ e' \subseteq dom\ (lcl\ s')$
and *reds-fvs-ok*: $\llbracket\ extTA,P,t \vdash \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle;\ fvs\ es \subseteq dom\ (lcl\ s)\ \rrbracket \Longrightarrow fvs\ es' \subseteq dom\ (lcl\ s')$
$\langle proof\rangle$

lemma *is-call-red-state-unchanged*:
$\llbracket\ extTA,P,t \vdash \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle;\ call\ e = \lfloor aMvs\rfloor;\ \neg\ synthesized\text{-}call\ P\ (hp\ s)\ aMvs\ \rrbracket \Longrightarrow s' = s \wedge ta = \varepsilon$

and *is-calls-reds-state-unchanged*:
$\llbracket\ extTA,P,t \vdash \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle;\ calls\ es = \lfloor aMvs\rfloor;\ \neg\ synthesized\text{-}call\ P\ (hp\ s)\ aMvs\ \rrbracket \Longrightarrow s' = s \wedge ta = \varepsilon$
$\langle proof\rangle$

lemma *called-methodD*:
$\llbracket\ extTA,P,t \vdash \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle;\ call\ e = \lfloor(a,\ M,\ vs)\rfloor;\ \neg\ synthesized\text{-}call\ P\ (hp\ s)\ (a,\ M,\ vs)\ \rrbracket$
$\Longrightarrow \exists\, hT\ D\ Us\ U\ pns\ body.\ hp\ s' = hp\ s \wedge typeof\text{-}addr\ (hp\ s)\ a = \lfloor hT\rfloor \wedge$
$P \vdash class\text{-}type\text{-}of\ hT\ sees\ M{:}\ Us\rightarrow U = \lfloor(pns,\ body)\rfloor\ in\ D \wedge$
$length\ vs = length\ pns \wedge length\ Us = length\ pns$

and *called-methodsD*:
$\llbracket\ extTA,P,t \vdash \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle;\ calls\ es = \lfloor(a,\ M,\ vs)\rfloor;\ \neg\ synthesized\text{-}call\ P\ (hp\ s)\ (a,\ M,\ vs)\ \rrbracket$
$\Longrightarrow \exists\, hT\ D\ Us\ U\ pns\ body.\ hp\ s' = hp\ s \wedge typeof\text{-}addr\ (hp\ s)\ a = \lfloor hT\rfloor \wedge$
$P \vdash class\text{-}type\text{-}of\ hT\ sees\ M{:}\ Us\rightarrow U = \lfloor(pns,\ body)\rfloor\ in\ D \wedge$
$length\ vs = length\ pns \wedge length\ Us = length\ pns$
$\langle proof\rangle$

7.2.1 Silent moves

primrec $\tau move0 :: 'm\ prog \Rightarrow 'heap \Rightarrow ('a,\ 'b,\ 'addr)\ exp \Rightarrow bool$
and $\tau moves0 :: 'm\ prog \Rightarrow 'heap \Rightarrow ('a,\ 'b,\ 'addr)\ exp\ list \Rightarrow bool$
where
$\tau move0\ P\ h\ (new\ C) \longleftrightarrow False$
$|\ \tau move0\ P\ h\ (newA\ T\lfloor e\rceil) \longleftrightarrow \tau move0\ P\ h\ e \vee (\exists\, a.\ e = Throw\ a)$
$|\ \tau move0\ P\ h\ (Cast\ U\ e) \longleftrightarrow \tau move0\ P\ h\ e \vee (\exists\, a.\ e = Throw\ a) \vee (\exists\, v.\ e = Val\ v)$
$|\ \tau move0\ P\ h\ (e\ instanceof\ T) \longleftrightarrow \tau move0\ P\ h\ e \vee (\exists\, a.\ e = Throw\ a) \vee (\exists\, v.\ e = Val\ v)$
$|\ \tau move0\ P\ h\ (e\ «bop»\ e') \longleftrightarrow \tau move0\ P\ h\ e \vee (\exists\, a.\ e = Throw\ a) \vee (\exists\, v.\ e = Val\ v \wedge$
$(\tau move0\ P\ h\ e' \vee (\exists\, a.\ e' = Throw\ a) \vee (\exists\, v.\ e' = Val\ v)))$
$|\ \tau move0\ P\ h\ (Val\ v) \longleftrightarrow False$
$|\ \tau move0\ P\ h\ (Var\ V) \longleftrightarrow True$
$|\ \tau move0\ P\ h\ (V := e) \longleftrightarrow \tau move0\ P\ h\ e \vee (\exists\, a.\ e = Throw\ a) \vee (\exists\, v.\ e = Val\ v)$
$|\ \tau move0\ P\ h\ (a\lfloor i\rceil) \longleftrightarrow \tau move0\ P\ h\ a \vee (\exists\, ad.\ a = Throw\ ad) \vee (\exists\, v.\ a = Val\ v \wedge (\tau move0\ P\ h\ i \vee (\exists\, a.\ i = Throw\ a)))$

| τmove0 P h (AAss a i e) ⟷ τmove0 P h a ∨ (∃ ad. a = Throw ad) ∨ (∃ v. a = Val v ∧
(τmove0 P h i ∨ (∃ a. i = Throw a) ∨ (∃ v. i = Val v ∧ (τmove0 P h e ∨ (∃ a. e = Throw a)))))
| τmove0 P h (a•length) ⟷ τmove0 P h a ∨ (∃ ad. a = Throw ad)
| τmove0 P h (e•F{D}) ⟷ τmove0 P h e ∨ (∃ a. e = Throw a)
| τmove0 P h (FAss e F D e′) ⟷ τmove0 P h e ∨ (∃ a. e = Throw a) ∨ (∃ v. e = Val v ∧ (τmove0 P h e′ ∨ (∃ a. e′ = Throw a)))
| τmove0 P h (e•compareAndSwap(D•F, e′, e″)) ⟷ τmove0 P h e ∨ (∃ a. e = Throw a) ∨ (∃ v. e = Val v ∧
(τmove0 P h e′ ∨ (∃ a. e′ = Throw a) ∨ (∃ v. e′ = Val v ∧ (τmove0 P h e″ ∨ (∃ a. e″ = Throw a)))))
| τmove0 P h (e•M(es)) ⟷ τmove0 P h e ∨ (∃ a. e = Throw a) ∨ (∃ v. e = Val v ∧
((τmoves0 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 ⟶ τexternal-defs D M)))))
| τmove0 P h ({V:T=vo; e}) ⟷ τmove0 P h e ∨ ((∃ a. e = Throw a) ∨ (∃ v. e = Val v))
| τmove0 P h ($sync_{V'}$(e) e′) ⟷ τmove0 P h e ∨ (∃ a. e = Throw a)
| τmove0 P h ($insync_{V'}$(ad) e) ⟷ τmove0 P h e
| τmove0 P h (e;;e′) ⟷ τmove0 P h e ∨ (∃ a. e = Throw a) ∨ (∃ v. e = Val v)
| τmove0 P h (if (e) e′ else e″) ⟷ τmove0 P h e ∨ (∃ a. e = Throw a) ∨ (∃ v. e = Val v)
| τmove0 P h (while (e) e′) = True
| — *Throw a* is no *τmove0* 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 *τmove0*, but would be if *Throw a* was.
τmove0 P h (throw e) ⟷ τmove0 P h e ∨ (∃ a. e = Throw a) ∨ e = null
| τmove0 P h (try e catch(C V) e′) ⟷ τmove0 P h e ∨ (∃ a. e = Throw a) ∨ (∃ v. e = Val v)

| τmoves0 P h [] ⟷ False
| τmoves0 P h (e # es) ⟷ τmove0 P h e ∨ (∃ v. e = Val v ∧ τmoves0 P h es)

abbreviation τMOVE :: ′m prog ⇒ ((′addr expr × ′addr locals) × ′heap, (′addr, ′thread-id, ′heap) J-thread-action) trsys
where τMOVE ≡ λP ((e, x), h) ta s′. τmove0 P h e ∧ ta = ε

primrec τMove0 :: ′m prog ⇒ ′heap ⇒ (′addr expr × ′addr expr list) ⇒ bool
where
τMove0 P h (e, exs) = (τmove0 P h e ∨ final e)

abbreviation τMOVE0 :: ′m prog ⇒ ((′addr expr × ′addr expr list) × ′heap, (′addr, ′thread-id, ′heap) J0-thread-action) trsys
where τMOVE0 ≡ λP (es, h) ta s. τMove0 P h es ∧ ta = ε

definition τred0 ::
((′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
τred0 extTA P t h exs e′xs′ =
(extTA,P,t ⊢ ⟨fst exs, (h, snd exs)⟩ −ε→ ⟨fst e′xs′, (h, snd e′xs′)⟩ ∧ τmove0 P h (fst exs) ∧ no-call P h (fst exs))

definition τreds0 ::
((′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
τreds0 extTA P t h esxs es′xs′ =
(*extTA*,*P*,*t* ⊢ ⟨*fst esxs*, (*h*, *snd esxs*)⟩ [−ε→] ⟨*fst es′xs′*, (*h*, *snd es′xs′*)⟩ ∧ *τmoves0 P h* (*fst esxs*) ∧ *no-calls P h* (*fst esxs*))

abbreviation *τred0t* ::
((′*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 *τred0t extTA P t h* ≡ (*τred0 extTA P t h*)^++

abbreviation *τreds0t* ::
((′*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 $\tau move0BinOp$: $\tau move0\ P\ h\ (Val\ v «bop» Val\ v')$
and $\tau move0Var$: $\tau move0\ P\ h\ (Var\ V)$
and $\tau move0LAss$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (V := e)$
and $\tau move0LAssRed$: $\tau move0\ P\ h\ (V := Val\ v)$
and $\tau move0AAcc1$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (e\lfloor e'\rceil)$
and $\tau move0AAcc2$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (Val\ v\lfloor e\rceil)$
and $\tau move0AAss1$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (e\lfloor e1\rceil := e2)$
and $\tau move0AAss2$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (Val\ v\lfloor e\rceil := e')$
and $\tau move0AAss3$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (Val\ v\lfloor Val\ v'\rceil := e)$
and $\tau move0ALength$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (e{\cdot}length)$
and $\tau move0FAcc$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (e{\cdot}F\{D\})$
and $\tau move0FAss1$: $\tau move0\ P\ h\ e \Longrightarrow \tau move0\ P\ h\ (FAss\ e\ F\ D\ e')$
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 *τmove0CondThrow*: *τmove0 P h* (*if* (*Throw a*) *e1 else e2*)
and *τmove0ThrowThrow*: *τmove0 P h* (*throw* (*Throw a*))

and *τmoves0Hd*: *τmove0 P h e* ⟹ *τmoves0 P h* (*e* # *es*)
and *τmoves0Tl*: *τmoves0 P h es* ⟹ *τmoves0 P h* (*Val v* # *es*)
⟨*proof*⟩

lemma *τmoves0-map-Val* [*iff*]:
¬ *τmoves0 P h* (*map Val vs*)
⟨*proof*⟩

lemma *τmoves0-map-Val-append* [*simp*]:
τmoves0 P h (*map Val vs* @ *es*) = *τmoves0 P h es*
⟨*proof*⟩

lemma *no-reds-map-Val-Throw* [*simp*]:
extTA,*P*,*t* ⊢ ⟨*map Val vs* @ *Throw a* # *es*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩ = *False*
⟨*proof*⟩

lemma assumes [*simp*]: *extTA* ε = ε
shows *red-τ-taD*: ⟦ *extTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩; *τmove0 P* (*hp s*) *e* ⟧ ⟹ *ta* = ε
and *reds-τ-taD*: ⟦ *extTA*,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩; *τmoves0 P* (*hp s*) *es* ⟧ ⟹ *ta* = ε
⟨*proof*⟩

lemma *τmove0-heap-unchanged*: ⟦ *extTA*,*P*,*t* ⊢ ⟨*e*, *s*⟩ −*ta*→ ⟨*e*′, *s*′⟩; *τmove0 P* (*hp s*) *e* ⟧ ⟹ *hp s*′ = *hp s*
and *τmoves0-heap-unchanged*: ⟦ *extTA*,*P*,*t* ⊢ ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*′, *s*′⟩; *τmoves0 P* (*hp s*) *es* ⟧ ⟹ *hp s*′ = *hp s*
⟨*proof*⟩

lemma *τMove0-iff*:
τMove0 P h ees ⟷ (*let* (*e*, -) = *ees in τmove0 P h e* ∨ *final e*)
⟨*proof*⟩

lemma *no-call-simps* [*simp*]:
no-call P h (*new C*) = *True*
no-call P h (*newA T*⌊*e*⌉) = *no-call P h e*
no-call P h (*Cast T e*) = *no-call P h e*
no-call P h (*e instanceof T*) = *no-call P h e*
no-call P h (*Val v*) = *True*
no-call P h (*Var V*) = *True*
no-call P h (*V* := *e*) = *no-call P h e*
no-call P h (*e* «*bop*» *e*′) = (*if is-val e then no-call P h e*′ *else no-call P h e*)
no-call P h (*a*⌊*i*⌉) = (*if is-val a then no-call P h i else no-call P h a*)
no-call P h (*AAss a i e*) = (*if is-val a then* (*if is-val i then no-call P h e else no-call P h i*) *else no-call P h a*)
no-call P h (*a*•*length*) = *no-call P h a*
no-call P h (*e*•*F*{*D*}) = *no-call P h e*
no-call P h (*FAss e F D e*′) = (*if is-val e then no-call P h e*′ *else no-call P h e*)
no-call P h (*e*•*compareAndSwap*(*D*•*F*, *e*′, *e*′′)) = (*if is-val e then* (*if is-val e*′ *then no-call P h e*′′ *else no-call P h e*′) *else no-call P h e*)
no-call P h (*e*•*M*(*es*)) = (*if is-val e then* (*if is-vals es* ∧ *is-addr e then synthesized-call P h* (*THE a*. *e* = *addr a*, *M*, *THE vs*. *es* = *map Val vs*) *else no-calls P h es*) *else no-call P h e*)
no-call P h ({*V*:*T*=*vo*; *e*}) = *no-call P h e*

no-call P h (*sync*$_{V'}$ (*e*) *e'*) = *no-call P h e*
no-call P h (*insync*$_{V'}$ (*ad*) *e*) = *no-call P h e*
no-call P h (*e*;;*e'*) = *no-call P h e*
no-call P h (*if* (*e*) *e1 else e2*) = *no-call P h e*
no-call P h (*while*(*e*) *e'*) = *True*
no-call P h (*throw e*) = *no-call P h e*
no-call P h (*try e catch*(*C V*) *e'*) = *no-call P h e*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *no-calls-map-Val* [*simp*]:
no-calls P h (*map Val vs*)
⟨*proof*⟩

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'*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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''*)
⟨*proof*⟩

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'''*)
⟨*proof*⟩

lemma *τreds0t-1step*:
⟦ *extTA*,*P*,*t* ⊢ ⟨*es*, (*h*, *xs*)⟩ [−ε→] ⟨*es'*, (*h*, *xs'*)⟩; *τmoves0 P h es*; *no-calls P h es* ⟧

$\Longrightarrow$ *τreds0t extTA P t h (es, xs) (es′, xs′)*
⟨*proof*⟩

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′* ⟧
$\Longrightarrow$ *τreds0t extTA P t h (es, xs) (es″, xs″)*
⟨*proof*⟩

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″* ⟧
$\Longrightarrow$ *τreds0t extTA P t h (es, xs) (es‴, xs‴)*
⟨*proof*⟩

lemma *τred0r-1step*:
⟦ *extTA,P,t* ⊢ ⟨*e*, (*h*, *xs*)⟩ −ε→ ⟨*e′*, (*h*, *xs′*)⟩; *τmove0 P h e*; *no-call P h e* ⟧
$\Longrightarrow$ *τred0r extTA P t h (e, xs) (e′, xs′)*
⟨*proof*⟩

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′* ⟧
$\Longrightarrow$ *τred0r extTA P t h (e, xs) (e″, xs″)*
⟨*proof*⟩

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″* ⟧
$\Longrightarrow$ *τred0r extTA P t h (e, xs) (e‴, xs‴)*
⟨*proof*⟩

lemma *τreds0r-1step*:
⟦ *extTA,P,t* ⊢ ⟨*es*, (*h*, *xs*)⟩ [−ε→] ⟨*es′*, (*h*, *xs′*)⟩; *τmoves0 P h es*; *no-calls P h es* ⟧
$\Longrightarrow$ *τreds0r extTA P t h (es, xs) (es′, xs′)*
⟨*proof*⟩

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′* ⟧
$\Longrightarrow$ *τreds0r extTA P t h (es, xs) (es″, xs″)*
⟨*proof*⟩

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″* ⟧
$\Longrightarrow$ *τreds0r extTA P t h (es, xs) (es‴, xs‴)*
⟨*proof*⟩

lemma *τred0t-inj-τreds0t*:
τred0t extTA P t h (e, xs) (e′, xs′)

$\Longrightarrow$ *τreds0t extTA P t h* (*e* # *es*, *xs*) (*e'* # *es*, *xs'*)
⟨*proof*⟩

lemma *τreds0t-cons-τreds0t*:
τreds0t extTA P t h (*es*, *xs*) (*es'*, *xs'*)
$\Longrightarrow$ *τreds0t extTA P t h* (*Val v* # *es*, *xs*) (*Val v* # *es'*, *xs'*)
⟨*proof*⟩

lemma *τred0r-inj-τreds0r*:
τred0r extTA P t h (*e*, *xs*) (*e'*, *xs'*)
$\Longrightarrow$ *τreds0r extTA P t h* (*e* # *es*, *xs*) (*e'* # *es*, *xs'*)
⟨*proof*⟩

lemma *τreds0r-cons-τreds0r*:
τreds0r extTA P t h (*es*, *xs*) (*es'*, *xs'*)
$\Longrightarrow$ *τreds0r extTA P t h* (*Val v* # *es*, *xs*) (*Val v* # *es'*, *xs'*)
⟨*proof*⟩

lemma *NewArray-τred0t-xt*:
τred0t extTA P t h (*e*, *xs*) (*e'*, *xs'*)
$\Longrightarrow$ *τred0t extTA P t h* (*newA T*⌊*e*⌉, *xs*) (*newA T*⌊*e'*⌉, *xs'*)
⟨*proof*⟩

lemma *Cast-τred0t-xt*:
τred0t extTA P t h (*e*, *xs*) (*e'*, *xs'*) $\Longrightarrow$ *τred0t extTA P t h* (*Cast T e*, *xs*) (*Cast T e'*, *xs'*)
⟨*proof*⟩

lemma *InstanceOf-τred0t-xt*:
τred0t extTA P t h (*e*, *xs*) (*e'*, *xs'*) $\Longrightarrow$ *τred0t extTA P t h* (*e instanceof T*, *xs*) (*e' instanceof T*, *xs'*)
⟨*proof*⟩

lemma *BinOp-τred0t-xt1*:
τred0t extTA P t h (*e1*, *xs*) (*e1'*, *xs'*) $\Longrightarrow$ *τred0t extTA P t h* (*e1* «*bop*» *e2*, *xs*) (*e1'* «*bop*» *e2*, *xs'*)
⟨*proof*⟩

lemma *BinOp-τred0t-xt2*:
τred0t extTA P t h (*e2*, *xs*) (*e2'*, *xs'*) $\Longrightarrow$ *τred0t extTA P t h* (*Val v* «*bop*» *e2*, *xs*) (*Val v* «*bop*» *e2'*, *xs'*)
⟨*proof*⟩

lemma *LAss-τred0t*:
τred0t extTA P t h (*e*, *xs*) (*e'*, *xs'*) $\Longrightarrow$ *τred0t extTA P t h* (*V* := *e*, *xs*) (*V* := *e'*, *xs'*)
⟨*proof*⟩

lemma *AAcc-τred0t-xt1*:
τred0t extTA P t h (*a*, *xs*) (*a'*, *xs'*) $\Longrightarrow$ *τred0t extTA P t h* (*a*⌊*i*⌉, *xs*) (*a'*⌊*i*⌉, *xs'*)
⟨*proof*⟩

lemma *AAcc-τred0t-xt2*:
τred0t extTA P t h (*i*, *xs*) (*i'*, *xs'*) $\Longrightarrow$ *τred0t extTA P t h* (*Val a*⌊*i*⌉, *xs*) (*Val a*⌊*i'*⌉, *xs'*)
⟨*proof*⟩

lemma *AAss-τred0t-xt1*:
τred0t extTA P t h (*a*, *xs*) (*a'*, *xs'*) $\Longrightarrow$ *τred0t extTA P t h* (*a*⌊*i*⌉ := *e*, *xs*) (*a'*⌊*i*⌉ := *e*, *xs'*)

$\langle proof \rangle$

lemma *AAss-τred0t-xt2*:
$\tau red0t\ extTA\ P\ t\ h\ (i,\ xs)\ (i',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (Val\ a\lfloor i\rceil := e,\ xs)\ (Val\ a\lfloor i'\rceil := e,\ xs')$
$\langle proof \rangle$

lemma *AAss-τred0t-xt3*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (Val\ a\lfloor Val\ i\rceil := e,\ xs)\ (Val\ a\lfloor Val\ i\rceil := e',\ xs')$
$\langle proof \rangle$

lemma *ALength-τred0t-xt*:
$\tau red0t\ extTA\ P\ t\ h\ (a,\ xs)\ (a',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (a{\cdot}length,\ xs)\ (a'{\cdot}length,\ xs')$
$\langle proof \rangle$

lemma *FAcc-τred0t-xt*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (e{\cdot}F\{D\},\ xs)\ (e'{\cdot}F\{D\},\ xs')$
$\langle proof \rangle$

lemma *FAss-τred0t-xt1*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (e{\cdot}F\{D\} := e2,\ xs)\ (e'{\cdot}F\{D\} := e2,\ xs')$
$\langle proof \rangle$

lemma *FAss-τred0t-xt2*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (Val\ v{\cdot}F\{D\} := e,\ xs)\ (Val\ v{\cdot}F\{D\} := e',\ xs')$
$\langle proof \rangle$

lemma *CAS-τred0t-xt1*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3),\ xs)\ (e'{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3),\ xs')$
$\langle proof \rangle$

lemma *CAS-τred0t-xt2*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ e,\ e3),\ xs)\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e3),\ xs')$
$\langle proof \rangle$

lemma *CAS-τred0t-xt3*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v',\ e),\ xs)\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v',\ e'),\ xs')$
$\langle proof \rangle$

lemma *Call-τred0t-obj*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (e{\cdot}M(ps),\ xs)\ (e'{\cdot}M(ps),\ xs')$
$\langle proof \rangle$

lemma *Call-τred0t-param*:
$\tau reds0t\ extTA\ P\ t\ h\ (es,\ xs)\ (es',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (Val\ v{\cdot}M(es),\ xs)\ (Val\ v{\cdot}M(es'),\ xs')$
$\langle proof \rangle$

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*⟩

lemma *Sync-τred0t-xt*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (sync_V\ (e)\ e2,\ xs)\ (sync_V\ (e')\ e2,\ xs')$
⟨*proof*⟩

lemma *InSync-τred0t-xt*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (insync_V\ (a)\ e,\ xs)\ (insync_V\ (a)\ e',\ xs')$
⟨*proof*⟩

lemma *Seq-τred0t-xt*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (e;;e2,\ xs)\ (e';;e2,\ xs')$
⟨*proof*⟩

lemma *Cond-τred0t-xt*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (if\ (e)\ e1\ else\ e2,\ xs)\ (if\ (e')\ e1\ else\ e2,\ xs')$
⟨*proof*⟩

lemma *Throw-τred0t-xt*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (throw\ e,\ xs)\ (throw\ e',\ xs')$
⟨*proof*⟩

lemma *Try-τred0t-xt*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0t\ extTA\ P\ t\ h\ (try\ e\ catch(C\ V)\ e2,\ xs)\ (try\ e'\ catch(C\ V)\ e2,\ xs')$
⟨*proof*⟩

lemma *NewArray-τred0r-xt*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (newA\ T\lfloor e\rceil,\ xs)\ (newA\ T\lfloor e'\rceil,\ xs')$
⟨*proof*⟩

lemma *Cast-τred0r-xt*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (Cast\ T\ e,\ xs)\ (Cast\ T\ e',\ xs')$
⟨*proof*⟩

lemma *InstanceOf-τred0r-xt*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (e\ instanceof\ T,\ xs)\ (e'\ instanceof\ T,\ xs')$
⟨*proof*⟩

lemma *BinOp-τred0r-xt1*:
$\tau red0r\ extTA\ P\ t\ h\ (e1,\ xs)\ (e1',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (e1\ «bop»\ e2,\ xs)\ (e1'\ «bop»\ e2,\ xs')$
⟨*proof*⟩

lemma *BinOp-τred0r-xt2*:
$\tau red0r\ extTA\ P\ t\ h\ (e2,\ xs)\ (e2',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (Val\ v\ «bop»\ e2,\ xs)\ (Val\ v\ «bop»\ e2',\ xs')$
⟨*proof*⟩

lemma *LAss-τred0r*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (V := e,\ xs)\ (V := e',\ xs')$
⟨*proof*⟩

lemma *AAcc-τred0r-xt1*:

$\tau red0r\ extTA\ P\ t\ h\ (a,\ xs)\ (a',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (a\lfloor i\rceil,\ xs)\ (a'\lfloor i\rceil,\ xs')$
⟨*proof*⟩

lemma *AAcc-τred0r-xt2*:
$\tau red0r\ extTA\ P\ t\ h\ (i,\ xs)\ (i',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (Val\ a\lfloor i\rceil,\ xs)\ (Val\ a\lfloor i'\rceil,\ xs')$
⟨*proof*⟩

lemma *AAss-τred0r-xt1*:
$\tau red0r\ extTA\ P\ t\ h\ (a,\ xs)\ (a',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (a\lfloor i\rceil := e,\ xs)\ (a'\lfloor i\rceil := e,\ xs')$
⟨*proof*⟩

lemma *AAss-τred0r-xt2*:
$\tau red0r\ extTA\ P\ t\ h\ (i,\ xs)\ (i',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (Val\ a\lfloor i\rceil := e,\ xs)\ (Val\ a\lfloor i'\rceil := e,\ xs')$
⟨*proof*⟩

lemma *AAss-τred0r-xt3*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (Val\ a\lfloor Val\ i\rceil := e,\ xs)\ (Val\ a\lfloor Val\ i\rceil := e',\ xs')$
⟨*proof*⟩

lemma *ALength-τred0r-xt*:
$\tau red0r\ extTA\ P\ t\ h\ (a,\ xs)\ (a',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (a{\bullet}length,\ xs)\ (a'{\bullet}length,\ xs')$
⟨*proof*⟩

lemma *FAcc-τred0r-xt*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (e{\bullet}F\{D\},\ xs)\ (e'{\bullet}F\{D\},\ xs')$
⟨*proof*⟩

lemma *FAss-τred0r-xt1*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (e{\bullet}F\{D\} := e2,\ xs)\ (e'{\bullet}F\{D\} := e2,\ xs')$
⟨*proof*⟩

lemma *FAss-τred0r-xt2*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (Val\ v{\bullet}F\{D\} := e,\ xs)\ (Val\ v{\bullet}F\{D\} := e',\ xs')$
⟨*proof*⟩

lemma *CAS-τred0r-xt1*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (e{\bullet}compareAndSwap(D{\bullet}F,\ e2,\ e3),\ xs)\ (e'{\bullet}compareAndSwap(D{\bullet}F,\ e2,\ e3),\ xs')$
⟨*proof*⟩

lemma *CAS-τred0r-xt2*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (Val\ v{\bullet}compareAndSwap(D{\bullet}F,\ e,\ e3),\ xs)\ (Val\ v{\bullet}compareAndSwap(D{\bullet}F,\ e',\ e3),\ xs')$
⟨*proof*⟩

lemma *CAS-τred0r-xt3*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (Val\ v{\bullet}compareAndSwap(D{\bullet}F,\ Val\ v',\ e),\ xs)\ (Val\ v{\bullet}compareAndSwap(D{\bullet}F,\ Val\ v',\ e'),\ xs')$
⟨*proof*⟩

lemma *Call-τred0r-obj*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (e{\bullet}M(ps),\ xs)\ (e'{\bullet}M(ps),\ xs')$

⟨*proof*⟩

lemma *Call-τred0r-param*:
$\tau reds0r\ extTA\ P\ t\ h\ (es,\ xs)\ (es',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (Val\ v{\cdot}M(es),\ xs)\ (Val\ v{\cdot}M(es'),\ xs')$
⟨*proof*⟩

lemma *Block-τred0r-xt*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs(V := vo))\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (\{V{:}T{=}vo;\ e\},\ xs)\ (\{V{:}T{=}xs'\ V;\ e'\},\ xs'(V := xs\ V))$
⟨*proof*⟩

lemma *Sync-τred0r-xt*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (sync_V\ (e)\ e2,\ xs)\ (sync_V\ (e')\ e2,\ xs')$
⟨*proof*⟩

lemma *InSync-τred0r-xt*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (insync_V\ (a)\ e,\ xs)\ (insync_V\ (a)\ e',\ xs')$
⟨*proof*⟩

lemma *Seq-τred0r-xt*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (e;;e2,\ xs)\ (e';;e2,\ xs')$
⟨*proof*⟩

lemma *Cond-τred0r-xt*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (if\ (e)\ e1\ else\ e2,\ xs)\ (if\ (e')\ e1\ else\ e2,\ xs')$
⟨*proof*⟩

lemma *Throw-τred0r-xt*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (throw\ e,\ xs)\ (throw\ e',\ xs')$
⟨*proof*⟩

lemma *Try-τred0r-xt*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red0r\ extTA\ P\ t\ h\ (try\ e\ catch(C\ V)\ e2,\ xs)\ (try\ e'\ catch(C\ V)\ e2,\ xs')$
⟨*proof*⟩

lemma *τRed0-conv* [*iff*]:
$\tau Red0\ P\ t\ h\ (e,\ es)\ (e',\ es') = (P,t \vdash 0\ \langle e/es,\ h\rangle\ -\varepsilon\rightarrow\ \langle e'/es',\ h\rangle \wedge \tau Move0\ P\ h\ (e,\ es))$
⟨*proof*⟩

lemma *τred0r-lcl-incr*:
$\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow dom\ xs \subseteq dom\ xs'$
⟨*proof*⟩

lemma *τred0t-lcl-incr*:
$\tau red0t\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow dom\ xs \subseteq dom\ xs'$
⟨*proof*⟩

lemma *τred0r-dom-lcl*:
assumes *wwf*: *wwf-J-prog P*
shows $\tau red0r\ extTA\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow dom\ xs' \subseteq dom\ xs \cup fv\ e$
⟨*proof*⟩

lemma τ*red0t-dom-lcl*:
assumes *wwf*: *wwf-J-prog P*
shows τ*red0t extTA P t h* (*e*, *xs*) (*e'*, *xs'*) $\Longrightarrow$ *dom xs'* $\subseteq$ *dom xs* $\cup$ *fv e*
⟨*proof*⟩

lemma τ*red0r-fv-subset*:
assumes *wwf*: *wwf-J-prog P*
shows τ*red0r extTA P t h* (*e*, *xs*) (*e'*, *xs'*) $\Longrightarrow$ *fv e'* $\subseteq$ *fv e*
⟨*proof*⟩

lemma τ*red0t-fv-subset*:
assumes *wwf*: *wwf-J-prog P*
shows τ*red0t extTA P t h* (*e*, *xs*) (*e'*, *xs'*) $\Longrightarrow$ *fv e'* $\subseteq$ *fv e*
⟨*proof*⟩

lemma fixes *e* :: (*'a*, *'b*, *'addr*) *exp* **and** *es* :: (*'a*, *'b*, *'addr*) *exp list*
shows τ*move0-callD*: *call e* = ⌊(*a*, *M*, *vs*)⌋ $\Longrightarrow$ τ*move0 P h e* $\longleftrightarrow$ (*synthesized-call P h* (*a*, *M*, *vs*) $\longrightarrow$ τ*external' P h a M*)
and τ*moves0-callsD*: *calls es* = ⌊(*a*, *M*, *vs*)⌋ $\Longrightarrow$ τ*moves0 P h es* $\longleftrightarrow$ (*synthesized-call P h* (*a*, *M*, *vs*) $\longrightarrow$ τ*external' P h a M*)
⟨*proof*⟩

lemma fixes *e* :: (*'a*, *'b*, *'addr*) *exp* **and** *es* :: (*'a*, *'b*, *'addr*) *exp list*
shows τ*move0-not-call*: ⟦ τ*move0 P h e*; *call e* = ⌊(*a*, *M*, *vs*)⌋; *synthesized-call P h* (*a*, *M*, *vs*) ⟧ $\Longrightarrow$ τ*external' P h a M*
and τ*moves0-not-calls*: ⟦ τ*moves0 P h es*; *calls es* = ⌊(*a*, *M*, *vs*)⌋; *synthesized-call P h* (*a*, *M*, *vs*) ⟧ $\Longrightarrow$ τ*external' P h a M*
⟨*proof*⟩

lemma τ*red0-into-*τ*Red0*:
assumes *red*: τ*red0* (*extTA2J0 P*) *P t h* (*e*, *Map.empty*) (*e'*, *xs'*)
shows τ*Red0 P t h* (*e*, *es*) (*e'*, *es*)
⟨*proof*⟩

lemma τ*red0r-into-*τ*Red0r*:
assumes *wwf*: *wwf-J-prog P*
shows
⟦ τ*red0r* (*extTA2J0 P*) *P t h* (*e*, *Map.empty*) (*e''*, *Map.empty*); *fv e* = {} ⟧
$\Longrightarrow$ τ*Red0r P t h* (*e*, *es*) (*e''*, *es*)
⟨*proof*⟩

lemma τ*red0t-into-*τ*Red0t*:
assumes *wwf*: *wwf-J-prog P*
shows
⟦ τ*red0t* (*extTA2J0 P*) *P t h* (*e*, *Map.empty*) (*e''*, *Map.empty*); *fv e* = {} ⟧
$\Longrightarrow$ τ*Red0t P t h* (*e*, *es*) (*e''*, *es*)
⟨*proof*⟩

lemma τ*red0r-Val*:
τ*red0r extTA P t h* (*Val v*, *xs*) *s'* $\longleftrightarrow$ *s'* = (*Val v*, *xs*)
⟨*proof*⟩

lemma τ*red0t-Val*:

$\tau red0t$ *extTA P t h* (*Val v*, *xs*) $s' \longleftrightarrow$ *False*
⟨*proof*⟩

lemma *τreds0r-map-Val*:
$\tau reds0r$ *extTA P t h* (*map Val vs*, *xs*) $s' \longleftrightarrow s' =$ (*map Val vs*, *xs*)
⟨*proof*⟩

lemma *τreds0t-map-Val*:
$\tau reds0t$ *extTA P t h* (*map Val vs*, *xs*) $s' \longleftrightarrow$ *False*
⟨*proof*⟩

lemma *Red-Suspend-is-call*:
⟦ *P*,*t* ⊢*0* ⟨*e*/*exs*, *h*⟩ −*ta*→ ⟨*e'*/*exs'*, *h'*⟩; *Suspend w* ∈ *set* ⦃*ta*⦄$_w$ ⟧ ⟹ *is-call e'*
⟨*proof*⟩

lemma *red0-mthr*: *multithreaded final-expr0* (*mred0 P*)
⟨*proof*⟩

lemma *red0-τmthr-wf*: *τmultithreaded-wf final-expr0* (*mred0 P*) (*τMOVE0 P*)
⟨*proof*⟩

lemma *red-τmthr-wf*: *τmultithreaded-wf final-expr* (*mred P*) (*τMOVE P*)
⟨*proof*⟩

end

sublocale *J-heap-base* < *red-mthr*:
τmultithreaded-wf
final-expr
mred P
convert-RA
τMOVE P
for *P*
⟨*proof*⟩

sublocale *J-heap-base* < *red0-mthr*:
τmultithreaded-wf
final-expr0
mred0 P
convert-RA
τMOVE0 P
for *P*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bisim-red-red0I* [*intro*]:
⟦ *e'* = *collapse ees*; *xs* = *Map.empty*; *h'* = *h*; *wf-state ees* ⟧ ⟹ *bisim-red-red0* ((*e'*, *xs*), *h'*) (*ees*, *h*)
⟨*proof*⟩

lemma *bisim-red-red0-final0D*:
⟦ *bisim-red-red0* (*x1*, *m1*) (*x2*, *m2*); *final-expr0 x2* ⟧ ⟹ *final-expr x1*
⟨*proof*⟩

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'*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*)⟩

$\wedge$ *x* = *x*′
(**is** ⟦ -; - ⟧ ⟹ *?concls Es Es′ x x′*)
⟨*proof*⟩

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*⟩

lemma (**in** −) *wf-state-ConsD*: *wf-state* (*e*, *e′* # *es*) ⟹ *wf-state* (*e′*, *es*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *τRed0r-inline-call-not-final*:
∃ *e′ es′*. *τRed0r P t h* (*e*, *es*) (*e′*, *es′*) ∧ (*final e′* ⟶ *es′* = []) ∧ *collapse* (*e*, *es*) = *collapse* (*e′*, *es′*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

lemma *collapse-τmove0-inv*:
⟦ ∀ *e*∈*set es*. *is-call e*; ¬ *final e* ⟧ ⟹ *τmove0 P h* (*collapse* (*e*, *es*)) = *τmove0 P h e*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *red-red0-FWbisim*:
assumes *wf*: *wwf-J-prog P*
shows *FWdelay-bisimulation-diverge final-expr* (*mred P*) *final-expr0* (*mred0 P*)
(λ*t*. *bisim-red-red0*) (λ*exs* (*e0*, *es0*). ¬ *final e0*) (*τMOVE P*) (*τMOVE0 P*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*

⟨*ML*⟩

typ (*'addr*,*'thread-id*,*'heap*) *J1-thread-action*

⟨*ML*⟩
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* e_1 *F D* e_2) = *max* (*max-vars* e_1) (*max-vars* e_2)
| *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* (e_1;;e_2) = *max* (*max-vars* e_1) (*max-vars* e_2)
| *max-vars* (*if* (*e*) e_1 *else* e_2) =
   max (*max-vars e*) (*max* (*max-vars* e_1) (*max-vars* e_2))
| *max-vars* (*while* (*b*) *e*) = *max* (*max-vars b*) (*max-vars e*)
| *max-vars* (*throw e*) = *max-vars e*
| *max-vars* (*try* e_1 *catch*(*C V*) e_2) = *max* (*max-vars* e_1) (*max-vars* e_2 + *1*)

| *max-varss* [] = *0*
| *max-varss* (*e*#*es*) = *max* (*max-vars e*) (*max-varss es*)

— Indices in blocks increase by 1

primrec $\mathcal{B}$ :: *'addr expr1* ⇒ *nat* ⇒ *bool*
  and $\mathcal{B}s$ :: *'addr expr1 list* ⇒ *nat* ⇒ *bool*
where
  $\mathcal{B}$ (*new C*) *i* = *True*
| $\mathcal{B}$ (*newA T*⌊*e*⌉) *i* = $\mathcal{B}$ *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* ∧ $\mathcal{B}$ *e2* *i*)
| $\mathcal{B}$ (*Var j*) *i* = *True*
| $\mathcal{B}$ (*j*:=*e*) *i* = $\mathcal{B}$ *e i*
| $\mathcal{B}$ (*a*⌊*j*⌉) *i* = ($\mathcal{B}$ *a i* ∧ $\mathcal{B}$ *j i*)
| $\mathcal{B}$ (*a*⌊*j*⌉:=*e*) *i* = ($\mathcal{B}$ *a i* ∧ $\mathcal{B}$ *j i* ∧ $\mathcal{B}$ *e i*)
| $\mathcal{B}$ (*a*·*length*) *i* = $\mathcal{B}$ *a i*
| $\mathcal{B}$ (*e*·*F*{*D*}) *i* = $\mathcal{B}$ *e i*
| $\mathcal{B}$ (*e1*·*F*{*D*} := *e2*) *i* = ($\mathcal{B}$ *e1 i* ∧ $\mathcal{B}$ *e2 i*)
| $\mathcal{B}$ (*e*·*compareAndSwap*(*D*·*F*, *e*′, *e*″)) *i* = ($\mathcal{B}$ *e i* ∧ $\mathcal{B}$ *e*′ *i* ∧ $\mathcal{B}$ *e*″ *i*)
| $\mathcal{B}$ (*e*·*M*(*es*)) *i* = ($\mathcal{B}$ *e i* ∧ $\mathcal{B}$*s es i*)
| $\mathcal{B}$ ({*j*:*T*=*vo*; *e*}) *i* = (*i* = *j* ∧ $\mathcal{B}$ *e* (*i*+*1*))
| $\mathcal{B}$ ($sync_V$ (*o*′) *e*) *i* = (*i* = *V* ∧ $\mathcal{B}$ *o*′ *i* ∧ $\mathcal{B}$ *e* (*i*+*1*))
| $\mathcal{B}$ ($insync_V$ (*a*) *e*) *i* = (*i* = *V* ∧ $\mathcal{B}$ *e* (*i*+*1*))
| $\mathcal{B}$ (*e1*;;*e2*) *i* = ($\mathcal{B}$ *e1 i* ∧ $\mathcal{B}$ *e2 i*)
| $\mathcal{B}$ (*if* (*e*) *e1 else e2*) *i* = ($\mathcal{B}$ *e i* ∧ $\mathcal{B}$ *e1 i* ∧ $\mathcal{B}$ *e2 i*)
| $\mathcal{B}$ (*throw e*) *i* = $\mathcal{B}$ *e i*
| $\mathcal{B}$ (*while* (*e*) *c*) *i* = ($\mathcal{B}$ *e i* ∧ $\mathcal{B}$ *c i*)
| $\mathcal{B}$ (*try e1 catch*(*C j*) *e2*) *i* = ($\mathcal{B}$ *e1 i* ∧ *i*=*j* ∧ $\mathcal{B}$ *e2* (*i*+*1*))

| $\mathcal{B}$*s* [] *i* = *True*
| $\mathcal{B}$*s* (*e*#*es*) *i* = ($\mathcal{B}$ *e i* ∧ $\mathcal{B}$*s es i*)

Variables for monitor addresses do not occur freely in synchonization blocks

primrec *syncvars* :: (′*a*, ′*a*, ′*addr*) *exp* ⇒ *bool*
and *syncvarss* :: (′*a*, ′*a*, ′*addr*) *exp list* ⇒ *bool*
where
syncvars (*new C*) = *True*
| *syncvars* (*newA T*⌊*e*⌉) = *syncvars e*
| *syncvars* (*Cast T e*) = *syncvars e*
| *syncvars* (*e instanceof T*) = *syncvars e*
| *syncvars* (*Val v*) = *True*
| *syncvars* (*e1* «*bop*» *e2*) = (*syncvars e1* ∧ *syncvars e2*)
| *syncvars* (*Var V*) = *True*
| *syncvars* (*V*:=*e*) = *syncvars e*
| *syncvars* (*a*⌊*i*⌉) = (*syncvars a* ∧ *syncvars i*)
| *syncvars* (*a*⌊*i*⌉ := *e*) = (*syncvars a* ∧ *syncvars i* ∧ *syncvars e*)
| *syncvars* (*a*·*length*) = *syncvars a*
| *syncvars* (*e*·*F*{*D*}) = *syncvars e*
| *syncvars* (*e*·*F*{*D*} := *e2*) = (*syncvars e* ∧ *syncvars e2*)
| *syncvars* (*e*·*compareAndSwap*(*D*·*F*, *e*′, *e*″)) = (*syncvars e* ∧ *syncvars e*′ ∧ *syncvars e*″)
| *syncvars* (*e*·*M*(*es*)) = (*syncvars e* ∧ *syncvarss es*)
| *syncvars* {*V*:*T*=*vo*;*e*} = *syncvars e*
| *syncvars* ($sync_V$ (*e1*) *e2*) = (*syncvars e1* ∧ *syncvars e2* ∧ *V* ∉ *fv e2*)
| *syncvars* ($insync_V$ (*a*) *e*) = (*syncvars e* ∧ *V* ∉ *fv e*)
| *syncvars* (*e1*;;*e2*) = (*syncvars e1* ∧ *syncvars e2*)
| *syncvars* (*if* (*b*) *e1 else e2*) = (*syncvars b* ∧ *syncvars e1* ∧ *syncvars e2*)
| *syncvars* (*while* (*b*) *c*) = (*syncvars b* ∧ *syncvars c*)
| *syncvars* (*throw e*) = *syncvars e*
| *syncvars* (*try e1 catch*(*C V*) *e2*) = (*syncvars e1* ∧ *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*
⟨*proof*⟩

lemma *max-varss-append* [*simp*]:
max-varss (*es* @ *es'*) = *max* (*max-varss es*) (*max-varss es'*)
⟨*proof*⟩

lemma *max-varss-map-Val* [*simp*]: *max-varss* (*map Val vs*) = *0*
⟨*proof*⟩

lemma *blocks1-max-vars*:
max-vars (*blocks1 n Ts e*) = *max-vars e* + *length Ts*
⟨*proof*⟩

lemma *blocks-max-vars*:
  $[\![$ *length vs* = *length pns*; *length Ts* = *length pns* $]\!]$
  $\Longrightarrow$ *max-vars* (*blocks pns Ts vs e*) = *max-vars e* + *length pns*
⟨*proof*⟩

lemma *Bs-append* [*simp*]: $\mathcal{B}s$ (*es* @ *es*′) *n* ⟷ $\mathcal{B}s$ *es n* ∧ $\mathcal{B}s$ *es*′ *n*
⟨*proof*⟩

lemma *Bs-map-Val* [*simp*]: $\mathcal{B}s$ (*map Val vs*) *n*
⟨*proof*⟩

lemma *B-blocks1* [*intro*]: $\mathcal{B}$ *body* (*n* + *length Ts*) $\Longrightarrow$ $\mathcal{B}$ (*blocks1 n Ts body*) *n*
⟨*proof*⟩

lemma *B-extRet2J* [*simp*]: $\mathcal{B}$ *e n* $\Longrightarrow$ $\mathcal{B}$ (*extRet2J e va*) *n*
⟨*proof*⟩

lemma *B-inline-call*: $[\![$ $\mathcal{B}$ *e n*; $\bigwedge n$. $\mathcal{B}$ *e*′ *n* $]\!]$ $\Longrightarrow$ $\mathcal{B}$ (*inline-call e*′ *e*) *n*
  and *Bs-inline-calls*: $[\![$ $\mathcal{B}s$ *es n*; $\bigwedge n$. $\mathcal{B}$ *e*′ *n* $]\!]$ $\Longrightarrow$ $\mathcal{B}s$ (*inline-calls e*′ *es*) *n*
⟨*proof*⟩

lemma *syncvarss-append* [*simp*]: *syncvarss* (*es* @ *es*′) ⟷ *syncvarss es* ∧ *syncvarss es*′
⟨*proof*⟩

lemma *syncvarss-map-Val* [*simp*]: *syncvarss* (*map Val vs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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′*
⟨*proof*⟩

lemma *calls1-map-Val-append* [*simp*]:
calls1 (*map Val vs* @ *es*) = *calls1 es*
⟨*proof*⟩

lemma *calls1-map-Val* [*simp*]:
calls1 (*map Val vs*) = *None*
⟨*proof*⟩

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*⌋
⟨*proof*⟩

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′*
⟨*proof*⟩

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* ⟨*proof*⟩

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* ⟨*proof*⟩

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* ⟨*proof*⟩

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*), [])
⟨*proof*⟩

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*
⟨*proof*⟩

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 ⊢1 ⟨e, s⟩ −ta→ ⟨e′, s′⟩ ⟹
  uf,P,t ⊢1 ⟨Val v·compareAndSwap(D·F, e, e3), s⟩ −ta→ ⟨Val v·compareAndSwap(D·F, e′, e3), s′⟩

| *CAS1Red3*:
  uf,P,t ⊢1 ⟨e, s⟩ −ta→ ⟨e′, s′⟩ ⟹

uf,P,t ⊢1 ⟨Val v·compareAndSwap(D·F, Val v′, e), s⟩ −ta→ ⟨Val v·compareAndSwap(D·F, Val v′, e′), s′⟩

| *CAS1Null*:
uf,P,t ⊢1 ⟨null·compareAndSwap(D·F, Val v, Val v′), s⟩ −ε→ ⟨THROW NullPointer, s⟩

| *Red1CASSucceed*:
⟦ heap-read h a (CField D F) v; heap-write h a (CField D F) v′ h′ ⟧ ⟹
uf,P,t ⊢1 ⟨addr a·compareAndSwap(D·F, Val v, Val v′), (h, l)⟩
−⦃ReadMem a (CField D F) v, WriteMem a (CField D F) v′⦄→
⟨true, (h′, l)⟩

| *Red1CASFail*:
⟦ heap-read h a (CField D F) v″; v ≠ v″ ⟧ ⟹
uf,P,t ⊢1 ⟨addr a·compareAndSwap(D·F, Val v, Val v′), (h, l)⟩
−⦃ReadMem a (CField D F) v″⦄→
⟨false, (h, l)⟩

| *Call1Obj*:
uf,P,t ⊢1 ⟨e, s⟩ −ta→ ⟨e′, s′⟩ ⟹ uf,P,t ⊢1 ⟨e·M(es), s⟩ −ta→ ⟨e′·M(es), s′⟩

| *Call1Params*:
uf,P,t ⊢1 ⟨es, s⟩ [−ta→] ⟨es′,s′⟩ ⟹
uf,P,t ⊢1 ⟨(Val v)·M(es),s⟩ −ta→ ⟨(Val v)·M(es′),s′⟩

| *Red1CallExternal*:
⟦ typeof-addr (hp s) a = ⌊T⌋; P ⊢ class-type-of T sees M:Ts→Tr = Native in D; P,t ⊢ ⟨a·M(vs), hp s⟩ −ta→ext ⟨va, h′⟩;
e′ = extRet2J1 ((addr a)·M(map Val vs)) va; s′ = (h′, lcl s) ⟧
⟹ uf,P,t ⊢1 ⟨(addr a)·M(map Val vs), s⟩ −ta→ ⟨e′, s′⟩

| *Red1CallNull*:
uf,P,t ⊢1 ⟨null·M(map Val vs), s⟩ −ε→ ⟨THROW NullPointer, s⟩

| *Block1Some*:
V < length x ⟹ uf,P,t ⊢1 ⟨{V:T=⌊v⌋; e}, (h, x)⟩ −ε→ ⟨{V:T=None; e}, (h, x[V := v])⟩

| *Block1Red*:
uf,P,t ⊢1 ⟨e, (h, x)⟩ −ta→ ⟨e′, (h′, x′)⟩
⟹ uf,P,t ⊢1 ⟨{V:T=None; e}, (h, x)⟩ −ta→ ⟨{V:T=None; e′}, (h′, x′)⟩

| *Red1Block*:
uf,P,t ⊢1 ⟨{V:T=None; Val u}, s⟩ −ε→ ⟨Val u, s⟩

| *Synchronized1Red1*:
uf,P,t ⊢1 ⟨o′, s⟩ −ta→ ⟨o″, s′⟩ ⟹ uf,P,t ⊢1 ⟨$sync_V$ *(o′) e, s⟩ −ta→ ⟨*$sync_V$ *(o″) e, s′⟩*

| *Synchronized1Null*:
V < length xs ⟹ uf,P,t ⊢1 ⟨$sync_V$ *(null) e, (h, xs)⟩ −ε→ ⟨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*)
⟨*proof*⟩

lemma *reds1-preserves-elen*: $uf,P,t \vdash 1\ \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle \Longrightarrow length\ es' = length\ es$
⟨*proof*⟩

lemma *red1-Val-iff* [*iff*]:
$\neg\ uf,P,t \vdash 1\ \langle Val\ v,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
⟨*proof*⟩

lemma *red1-Throw-iff* [*iff*]:
$\neg\ uf,P,t \vdash 1\ \langle Throw\ a,\ xs\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$
⟨*proof*⟩

lemma *reds1-Nil-iff* [*iff*]:
$\neg\ uf,P,t \vdash 1\ \langle [],\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle$
⟨*proof*⟩

lemma *reds1-Val-iff* [*iff*]:
$\neg\ uf,P,t \vdash 1\ \langle map\ Val\ vs,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle$
⟨*proof*⟩

lemma *reds1-map-Val-Throw-iff* [*iff*]:
$\neg\ uf,P,t \vdash 1\ \langle map\ Val\ vs\ @\ Throw\ a\ \#\ es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle$
⟨*proof*⟩

lemma *red1-max-vars-decr*: $uf,P,t \vdash 1\ \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle \Longrightarrow max\text{-}vars\ e' \leq max\text{-}vars\ e$
and *reds1-max-varss-decr*: $uf,P,t \vdash 1\ \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle \Longrightarrow max\text{-}varss\ es' \leq max\text{-}varss\ es$
⟨*proof*⟩

lemma *red1-new-thread-heap*: $[\![uf,P,t \vdash 1\ \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle;\ NewThread\ t'\ ex\ h \in set\ \{\!|ta|\!\}_t\ ]\!] \Longrightarrow h = hp\ s'$
and *reds1-new-thread-heap*: $[\![uf,P,t \vdash 1\ \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle;\ NewThread\ t'\ ex\ h \in set\ \{\!|ta|\!\}_t\ ]\!] \Longrightarrow h = hp\ s'$
⟨*proof*⟩

lemma *red1-new-threadD*:
$[\![\ uf,P,t \vdash 1\ \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle;\ NewThread\ t'\ x\ H \in set\ \{\!|ta|\!\}_t\ ]\!]$
$\Longrightarrow \exists\, a\ M\ vs\ va\ T\ Ts\ Tr\ D.\ P,t \vdash \langle a{\cdot}M(vs),\ hp\ s\rangle\ -ta\rightarrow ext\ \langle va,\ hp\ s'\rangle \wedge typeof\text{-}addr\ (hp\ s)\ a = \lfloor T\rfloor \wedge P \vdash class\text{-}type\text{-}of\ T\ sees\ M{:}Ts{\rightarrow}Tr = Native\ in\ D$
and *reds1-new-threadD*:
$[\![\ uf,P,t \vdash 1\ \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle;\ NewThread\ t'\ x\ H \in set\ \{\!|ta|\!\}_t\ ]\!]$
$\Longrightarrow \exists\, a\ M\ vs\ va\ T\ Ts\ Tr\ D.\ P,t \vdash \langle a{\cdot}M(vs),\ hp\ s\rangle\ -ta\rightarrow ext\ \langle va,\ hp\ s'\rangle \wedge typeof\text{-}addr\ (hp\ s)\ a = \lfloor T\rfloor \wedge P \vdash class\text{-}type\text{-}of\ T\ sees\ M{:}Ts{\rightarrow}Tr = Native\ in\ D$
⟨*proof*⟩

lemma *red1-call-synthesized*: $[\![\ uf,P,t \vdash 1\ \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle;\ call1\ e = \lfloor aMvs\rfloor\ ]\!] \Longrightarrow synthesized\text{-}call\ P\ (hp\ s)\ aMvs$
and *reds1-calls-synthesized*: $[\![\ uf,P,t \vdash 1\ \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle;\ calls1\ es = \lfloor aMvs\rfloor\ ]\!] \Longrightarrow synthesized\text{-}call\ P\ (hp\ s)\ aMvs$
⟨*proof*⟩

lemma *red1-preserves-B*: $[\![\ uf,P,t \vdash 1\ \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle;\ \mathcal{B}\ e\ n]\!] \Longrightarrow \mathcal{B}\ e'\ n$
and *reds1-preserves-Bs*: $[\![\ uf,P,t \vdash 1\ \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle;\ \mathcal{B}s\ es\ n]\!] \Longrightarrow \mathcal{B}s\ es'\ n$
⟨*proof*⟩

end

context *J1-heap* **begin**

lemma *red1-hext-incr*: $uf,P,t \vdash 1\ \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle \Longrightarrow hext\ (hp\ s)\ (hp\ s')$
and *reds1-hext-incr*: $uf,P,t \vdash 1\ \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle \Longrightarrow hext\ (hp\ s)\ (hp\ s')$
⟨*proof*⟩

lemma *Red1-hext-incr*: $uf,P,t \vdash 1\ \langle ex/exs,h\rangle\ -ta\rightarrow\ \langle ex'/exs',h'\rangle \Longrightarrow h \trianglelefteq h'$
⟨*proof*⟩

end

7.6.1 Silent moves

context *J1-heap-base* **begin**

primrec $\tau move1$ :: $'m\ prog \Rightarrow 'heap \Rightarrow ('a,\ 'b,\ 'addr)\ exp \Rightarrow bool$
and $\tau moves1$ :: $'m\ prog \Rightarrow 'heap \Rightarrow ('a,\ 'b,\ 'addr)\ exp\ list \Rightarrow bool$
where
$\tau move1\ P\ h\ (new\ C) \longleftrightarrow False$
$|\ \tau move1\ P\ h\ (newA\ T\lfloor e\rceil) \longleftrightarrow \tau move1\ P\ h\ e \lor (\exists a.\ e = Throw\ a)$
$|\ \tau move1\ P\ h\ (Cast\ U\ e) \longleftrightarrow \tau move1\ P\ h\ e \lor final\ e$
$|\ \tau move1\ P\ h\ (e\ instanceof\ T) \longleftrightarrow \tau move1\ P\ h\ e \lor final\ e$
$|\ \tau move1\ P\ h\ (e\ «bop»\ e') \longleftrightarrow \tau move1\ P\ h\ e \lor (\exists a.\ e = Throw\ a) \lor (\exists v.\ e = Val\ v \land (\tau move1\ P\ h\ e' \lor final\ e'))$
$|\ \tau move1\ P\ h\ (Val\ v) \longleftrightarrow False$
$|\ \tau move1\ P\ h\ (Var\ V) \longleftrightarrow True$
$|\ \tau move1\ P\ h\ (V := e) \longleftrightarrow \tau move1\ P\ h\ e \lor final\ e$
$|\ \tau move1\ P\ h\ (a\lfloor i\rceil) \longleftrightarrow \tau move1\ P\ h\ a \lor (\exists ad.\ a = Throw\ ad) \lor (\exists v.\ a = Val\ v \land (\tau move1\ P\ h\ i \lor (\exists a.\ i = Throw\ a)))$
$|\ \tau move1\ P\ h\ (AAss\ a\ i\ e) \longleftrightarrow \tau move1\ P\ h\ a \lor (\exists ad.\ a = Throw\ ad) \lor (\exists v.\ a = Val\ v \land (\tau move1\ P\ h\ i \lor (\exists a.\ i = Throw\ a) \lor (\exists v.\ i = Val\ v \land (\tau move1\ P\ h\ e \lor (\exists a.\ e = Throw\ a)))))$
$|\ \tau move1\ P\ h\ (a{\cdot}length) \longleftrightarrow \tau move1\ P\ h\ a \lor (\exists ad.\ a = Throw\ ad)$
$|\ \tau move1\ P\ h\ (e{\cdot}F\{D\}) \longleftrightarrow \tau move1\ P\ h\ e \lor (\exists a.\ e = Throw\ a)$
$|\ \tau move1\ P\ h\ (FAss\ e\ F\ D\ e') \longleftrightarrow \tau move1\ P\ h\ e \lor (\exists a.\ e = Throw\ a) \lor (\exists v.\ e = Val\ v \land (\tau move1\ P\ h\ e' \lor (\exists a.\ e' = Throw\ a)))$
$|\ \tau move1\ P\ h\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e'')) \longleftrightarrow \tau move1\ P\ h\ e \lor (\exists a.\ e = Throw\ a) \lor (\exists v.\ e = Val\ v \land$
$(\tau move1\ P\ h\ e' \lor (\exists a.\ e' = Throw\ a) \lor (\exists v.\ e' = Val\ v \land (\tau move1\ P\ h\ e'' \lor (\exists a.\ e'' = Throw\ a)))))$
$|\ \tau move1\ P\ h\ (e{\cdot}M(es)) \longleftrightarrow \tau move1\ P\ h\ e \lor (\exists a.\ e = Throw\ a) \lor (\exists v.\ e = Val\ v \land$
$(\tau moves1\ P\ h\ es \lor (\exists vs\ a\ es'.\ es = map\ Val\ vs\ @\ Throw\ a\ \#\ es') \lor$
$(\exists vs.\ es = map\ Val\ vs \land (v = Null \lor (\forall T\ C\ Ts\ Tr\ D.\ typeof_h\ v = \lfloor T\rfloor \longrightarrow class\text{-}type\text{-}of'\ T = \lfloor C\rfloor \longrightarrow P \vdash C\ sees\ M{:}Ts{\rightarrow}Tr = Native\ in\ D \longrightarrow \tau external\text{-}defs\ D\ M)))))$
$|\ \tau move1\ P\ h\ (\{V{:}T{=}vo;\ e\}) \longleftrightarrow vo \neq None \lor \tau move1\ P\ h\ e \lor final\ e$
$|\ \tau move1\ P\ h\ (sync_{V'}(e)\ e') \longleftrightarrow \tau move1\ P\ h\ e \lor (\exists a.\ e = Throw\ a)$
$|\ \tau move1\ P\ h\ (insync_{V'}(ad)\ e) \longleftrightarrow \tau move1\ P\ h\ e$
$|\ \tau move1\ P\ h\ (e;;e') \longleftrightarrow \tau move1\ P\ h\ e \lor final\ e$
$|\ \tau move1\ P\ h\ (if\ (e)\ e'\ else\ e'') \longleftrightarrow \tau move1\ P\ h\ e \lor final\ e$
$|\ \tau move1\ P\ h\ (while\ (e)\ e') = True$
$|\ \tau move1\ P\ h\ (throw\ e) \longleftrightarrow \tau move1\ P\ h\ e \lor (\exists a.\ e = Throw\ a) \lor e = null$
$|\ \tau move1\ P\ h\ (try\ e\ catch(C\ V)\ e') \longleftrightarrow \tau move1\ P\ h\ e \lor final\ e$

| τ*moves1 P h* [] $\longleftrightarrow$ *False*
| τ*moves1 P h* (*e* # *es*) $\longleftrightarrow$ τ*move1 P h e* $\vee$ ($\exists$ *v*. *e* = *Val v* $\wedge$ τ*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*) = (τ*move1 P h e* $\vee$ *final e*)

definition τ*red1g* :: *bool* $\Rightarrow$ *'addr J1-prog* $\Rightarrow$ *'thread-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*)$\rangle$ $-\varepsilon\rightarrow$ $\langle$*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-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*)$\rangle$ [$-\varepsilon\rightarrow$] $\langle$*fst es'xs'*, (*h*, *snd es'xs'*)$\rangle$ $\wedge$ τ*moves1 P h* (*fst esxs*))

abbreviation τ*red1gt* ::
bool $\Rightarrow$ *'addr J1-prog* $\Rightarrow$ *'thread-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-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-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-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-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*$\rangle$ $-\varepsilon\rightarrow$ $\langle$*fst ex'exs'*/*snd ex'exs'*, *h*$\rangle$ $\wedge$ τ*Move1 P h exexs*)

abbreviation τ*Red1gt* ::
bool $\Rightarrow$ *'addr J1-prog* $\Rightarrow$ *'thread-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*Red1gt uf P t h* $\equiv$ (τ*Red1g uf P t h*)^++

abbreviation τ*Red1gr* ::
bool $\Rightarrow$ *'addr J1-prog* $\Rightarrow$ *'thread-id* $\Rightarrow$ *'heap* $\Rightarrow$ (*'addr expr1* $\times$ *'addr locals1*) $\times$ ((*'addr expr1* $\times$ *'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 *τRed1t* ≡ *τRed1gt True*

abbreviation *τRed1r* ::
'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 *τRed1r* ≡ *τRed1gr True*

abbreviation *τred1'* ::
'addr J1-prog ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr1* × *'addr locals1*) ⇒ (*'addr expr1* × *'addr locals1*)

$\Rightarrow$ *bool*
where *τred1'* ≡ *τred1g False*

abbreviation *τreds1'* ::
'addr J1-prog ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr1 list* × *'addr locals1*) ⇒ (*'addr expr1 list* × *'addr locals1*) ⇒ *bool*
where *τreds1'* ≡ *τreds1g False*

abbreviation *τred1't* ::
'addr J1-prog ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr1* × *'addr locals1*) ⇒ (*'addr expr1* × *'addr locals1*) ⇒ *bool*
where *τred1't* ≡ *τred1gt False*

abbreviation *τreds1't* ::
'addr J1-prog ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr1 list* × *'addr locals1*) ⇒ (*'addr expr1 list* × *'addr locals1*) ⇒ *bool*
where *τreds1't* ≡ *τreds1gt False*

abbreviation *τred1'r* ::
'addr J1-prog ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr1* × *'addr locals1*) ⇒ (*'addr expr1* × *'addr locals1*) ⇒ *bool*
where *τred1'r* ≡ *τred1gr False*

abbreviation *τreds1'r* ::
'addr J1-prog ⇒ *'thread-id* ⇒ *'heap* ⇒ (*'addr expr1 list* × *'addr locals1*) ⇒ (*'addr expr1 list* × *'addr locals1*) ⇒ *bool*
where *τreds1'r* ≡ *τreds1gr False*

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 False*

abbreviation *τRed1't* ::
'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't* ≡ *τRed1gt False*

abbreviation *τRed1'r* ::
'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'r* ≡ *τRed1gr False*

abbreviation *τMOVE1* ::
'm prog ⇒ (((*'addr expr1* × *'addr locals1*) × (*'addr expr1* × *'addr locals1*) *list*) × *'heap*, (*'addr*, *'thread-id*, *'heap*) *J1-thread-action*) *trsys*
where *τMOVE1 P* ≡ λ(*exexs*, *h*) *ta s*. *τMove1 P h exexs* ∧ *ta* = ε

lemma *τmove1-τmoves1-intros*:
fixes *e* :: (*'a*, *'b*, *'addr*) *exp* **and** *es* :: (*'a*, *'b*, *'addr*) *exp list*
shows *τmove1NewArray*: *τmove1 P h e* ⟹ *τmove1 P h* (*newA T*⌊*e*⌉)

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 $\tau move1BinOp1$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (e«bop»e')$
and $\tau move1BinOp2$: $\tau move1\ P\ h\ e \Longrightarrow \tau move1\ P\ h\ (Val\ v«bop»e)$
and $\tau move1BinOp$: $\tau move1\ P\ h\ (Val\ v«bop»\ Val\ v')$
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 $\tau move1Call$: $(\bigwedge T\ C\ Ts\ Tr\ D.\ [\![\ typeof_h\ v = \lfloor T\rfloor;\ class\text{-}type\text{-}of'\ T = \lfloor C\rfloor;\ P \vdash C\ sees\ M{:}Ts{\rightarrow}Tr = Native\ in\ D\ ]\!] \Longrightarrow \tau external\text{-}defs\ D\ M) \Longrightarrow \tau move1\ P\ h\ (Val\ v{\cdot}M(map\ Val\ vs))$
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 $\tau move1BinOpThrow1$: $\tau move1\ P\ h\ (Throw\ a\ «bop»\ e2)$
and $\tau move1BinOpThrow2$: $\tau move1\ P\ h\ (Val\ v\ «bop»\ Throw\ a)$
and $\tau move1LAssThrow$: $\tau move1\ P\ h\ (V{:=}(Throw\ a))$
and $\tau move1AAccThrow1$: $\tau move1\ P\ h\ (Throw\ a\lfloor e\rceil)$
and $\tau move1AAccThrow2$: $\tau move1\ P\ h\ (Val\ v\lfloor Throw\ a\rceil)$
and $\tau move1AAssThrow1$: $\tau move1\ P\ h\ (AAss\ (Throw\ a)\ e\ e')$
and $\tau move1AAssThrow2$: $\tau move1\ P\ h\ (AAss\ (Val\ v)\ (Throw\ a)\ e')$
and $\tau move1AAssThrow3$: $\tau move1\ P\ h\ (AAss\ (Val\ v)\ (Val\ v')\ (Throw\ a))$
and $\tau move1ALengthThrow$: $\tau move1\ P\ h\ (Throw\ a{\cdot}length)$
and $\tau move1FAccThrow$: $\tau move1\ P\ h\ (Throw\ a{\cdot}F\{D\})$

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*)
⟨*proof*⟩

lemma *τmoves1-map-Val* [*dest*!]:
τmoves1 P h (*map Val es*) ⟹ *False*
⟨*proof*⟩

lemma *τmoves1-map-Val-ThrowD* [*simp*]: *τmoves1 P h* (*map Val vs* @ *Throw a* # *es*) = *False*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *τMove1-iff*:
τMove1 P h exexs ⟷ (*let* ((*e*, -), -) = *exexs in τmove1 P h e* ∨ *final e*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)

⟨proof⟩

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*′)
⟨proof⟩

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*″)
⟨proof⟩

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*‴)
⟨proof⟩

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*′)
⟨proof⟩

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*″)
⟨proof⟩

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*″ ⟧
⟹ *τreds1gt uf P t h* (*es*, *xs*) (*es*‴, *xs*‴)
⟨proof⟩

lemma *τred1r-1step*:
⟦ *uf*,*P*,*t* ⊢*1* ⟨*e*, (*h*, *xs*)⟩ −*ε*→ ⟨*e*′, (*h*, *xs*′)⟩; *τmove1 P h e* ⟧
⟹ *τred1gr uf P t h* (*e*, *xs*) (*e*′, *xs*′)
⟨proof⟩

lemma *τred1r-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*′ ⟧
⟹ *τred1gr uf P t h* (*e*, *xs*) (*e*″, *xs*″)
⟨proof⟩

lemma *τred1r-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*″ ⟧
⟹ *τred1gr uf P t h* (*e*, *xs*) (*e*‴, *xs*‴)

⟨*proof*⟩

lemma *τreds1r-1step*:
⟦ *uf*,*P*,*t* ⊢*1* ⟨*es*, (*h*, *xs*)⟩ [−ε→] ⟨*es'*, (*h*, *xs'*)⟩; *τmoves1 P h es* ⟧
⟹ *τreds1gr uf P t h* (*es*, *xs*) (*es'*, *xs'*)
⟨*proof*⟩

lemma *τreds1r-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'* ⟧
⟹ *τreds1gr uf P t h* (*es*, *xs*) (*es''*, *xs''*)
⟨*proof*⟩

lemma *τreds1r-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''* ⟧
⟹ *τreds1gr uf P t h* (*es*, *xs*) (*es'''*, *xs'''*)
⟨*proof*⟩

lemma *τred1t-preserves-len*: *τred1gt uf P t h* (*e*, *xs*) (*e'*, *xs'*) ⟹ *length xs'* = *length xs*
⟨*proof*⟩

lemma *τred1r-preserves-len*: *τred1gr uf P t h* (*e*, *xs*) (*e'*, *xs'*) ⟹ *length xs'* = *length xs*
⟨*proof*⟩

lemma *τred1t-inj-τreds1t*: *τred1gt uf P t h* (*e*, *xs*) (*e'*, *xs'*) ⟹ *τreds1gt uf P t h* (*e* # *es*, *xs*) (*e'* # *es*, *xs'*)
⟨*proof*⟩

lemma *τreds1t-cons-τreds1t*: *τreds1gt uf P t h* (*es*, *xs*) (*es'*, *xs'*) ⟹ *τreds1gt uf P t h* (*Val v* # *es*, *xs*) (*Val v* # *es'*, *xs'*)
⟨*proof*⟩

lemma *τred1r-inj-τreds1r*: *τred1gr uf P t h* (*e*, *xs*) (*e'*, *xs'*) ⟹ *τreds1gr uf P t h* (*e* # *es*, *xs*) (*e'* # *es*, *xs'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

lemma *BinOp-τred1t-xt1*:
$\tau red1gt\ uf\ P\ t\ h\ (e1,\ xs)\ (e1',\ xs') \Longrightarrow \tau red1gt\ uf\ P\ t\ h\ (e1\ «bop»\ e2,\ xs)\ (e1'\ «bop»\ e2,\ xs')$
⟨*proof*⟩

lemma *BinOp-τred1t-xt2*:
$\tau red1gt\ uf\ P\ t\ h\ (e2,\ xs)\ (e2',\ xs') \Longrightarrow \tau red1gt\ uf\ P\ t\ h\ (Val\ v\ «bop»\ e2,\ xs)\ (Val\ v\ «bop»\ e2',\ xs')$
⟨*proof*⟩

lemma *LAss-τred1t*:
$\tau red1gt\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gt\ uf\ P\ t\ h\ (V := e,\ xs)\ (V := e',\ xs')$
⟨*proof*⟩

lemma *AAcc-τred1t-xt1*:
$\tau red1gt\ uf\ P\ t\ h\ (a,\ xs)\ (a',\ xs') \Longrightarrow \tau red1gt\ uf\ P\ t\ h\ (a\lfloor i\rceil,\ xs)\ (a'\lfloor i\rceil,\ xs')$
⟨*proof*⟩

lemma *AAcc-τred1t-xt2*:
$\tau red1gt\ uf\ P\ t\ h\ (i,\ xs)\ (i',\ xs') \Longrightarrow \tau red1gt\ uf\ P\ t\ h\ (Val\ a\lfloor i\rceil,\ xs)\ (Val\ a\lfloor i'\rceil,\ xs')$
⟨*proof*⟩

lemma *AAss-τred1t-xt1*:
$\tau red1gt\ uf\ P\ t\ h\ (a,\ xs)\ (a',\ xs') \Longrightarrow \tau red1gt\ uf\ P\ t\ h\ (a\lfloor i\rceil := e,\ xs)\ (a'\lfloor i\rceil := e,\ xs')$
⟨*proof*⟩

lemma *AAss-τred1t-xt2*:
$\tau red1gt\ uf\ P\ t\ h\ (i,\ xs)\ (i',\ xs') \Longrightarrow \tau red1gt\ uf\ P\ t\ h\ (Val\ a\lfloor i\rceil := e,\ xs)\ (Val\ a\lfloor i'\rceil := e,\ xs')$
⟨*proof*⟩

lemma *AAss-τred1t-xt3*:
$\tau red1gt\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gt\ uf\ P\ t\ h\ (Val\ a\lfloor Val\ i\rceil := e,\ xs)\ (Val\ a\lfloor Val\ i\rceil := e',\ xs')$
⟨*proof*⟩

lemma *ALength-τred1t-xt*:
$\tau red1gt\ uf\ P\ t\ h\ (a,\ xs)\ (a',\ xs') \Longrightarrow \tau red1gt\ uf\ P\ t\ h\ (a{\cdot}length,\ xs)\ (a'{\cdot}length,\ xs')$
⟨*proof*⟩

lemma *FAcc-τred1t-xt*:
$\tau red1gt\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gt\ uf\ P\ t\ h\ (e{\cdot}F\{D\},\ xs)\ (e'{\cdot}F\{D\},\ xs')$
⟨*proof*⟩

lemma *FAss-τred1t-xt1*:
$\tau red1gt\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gt\ uf\ P\ t\ h\ (e{\cdot}F\{D\} := e2,\ xs)\ (e'{\cdot}F\{D\} := e2,\ xs')$
⟨*proof*⟩

lemma *FAss-τred1t-xt2*:
$\tau red1gt\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gt\ uf\ P\ t\ h\ (Val\ v{\cdot}F\{D\} := e,\ xs)\ (Val\ v{\cdot}F\{D\} := e',\ xs')$
⟨*proof*⟩

lemma *CAS-τred1t-xt1*:
$\tau red1gt\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gt\ uf\ P\ t\ h\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3),\ xs)\ (e'{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3),\ xs')$
⟨*proof*⟩

lemma *CAS-τred1t-xt2*:

τred1gt uf P t h (e, xs) (e′, xs′) ⟹ τred1gt uf P t h (Val v∙compareAndSwap(D∙F, e, e3), xs) (Val v∙compareAndSwap(D∙F, e′, e3), xs′)
⟨*proof*⟩

lemma *CAS-τred1t-xt3*:
τred1gt uf P t h (e, xs) (e′, xs′) ⟹ τred1gt uf P t h (Val v∙compareAndSwap(D∙F, Val v′, e), xs) (Val v∙compareAndSwap(D∙F, Val v′, e′), xs′)
⟨*proof*⟩

lemma *Call-τred1t-obj*:
τred1gt uf P t h (e, xs) (e′, xs′) ⟹ τred1gt uf P t h (e∙M(ps), xs) (e′∙M(ps), xs′)
⟨*proof*⟩

lemma *Call-τred1t-param*:
τreds1gt uf P t h (es, xs) (es′, xs′) ⟹ τred1gt uf P t h (Val v∙M(es), xs) (Val v∙M(es′), xs′)
⟨*proof*⟩

lemma *Block-None-τred1t-xt*:
τred1gt uf P t h (e, xs) (e′, xs′) ⟹ τred1gt uf P t h ({V:T=None; e}, xs) ({V:T=None; e′}, xs′)
⟨*proof*⟩

lemma *Block-τred1t-Some*:
⟦ *τred1gt uf P t h (e, xs[V := v]) (e′, xs′); V < length xs* ⟧
⟹ *τred1gt uf P t h ({V:Ty=⌊v⌋; e}, xs) ({V:Ty=None; e′}, xs′)*
⟨*proof*⟩

lemma *Sync-τred1t-xt*:
τred1gt uf P t h (e, xs) (e′, xs′) ⟹ τred1gt uf P t h (sync$_V$ (e) e2, xs) (sync$_V$ (e′) e2, xs′)
⟨*proof*⟩

lemma *InSync-τred1t-xt*:
τred1gt uf P t h (e, xs) (e′, xs′) ⟹ τred1gt uf P t h (insync$_V$ (a) e, xs) (insync$_V$ (a) e′, xs′)
⟨*proof*⟩

lemma *Seq-τred1t-xt*:
τred1gt uf P t h (e, xs) (e′, xs′) ⟹ τred1gt uf P t h (e;;e2, xs) (e′;;e2, xs′)
⟨*proof*⟩

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′)
⟨*proof*⟩

lemma *Throw-τred1t-xt*:
τred1gt uf P t h (e, xs) (e′, xs′) ⟹ τred1gt uf P t h (throw e, xs) (throw e′, xs′)
⟨*proof*⟩

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′)
⟨*proof*⟩

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′)

⟨*proof*⟩

lemma *Cast-τred1r-xt*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (Cast\ T\ e,\ xs)\ (Cast\ T\ e',\ xs')$
⟨*proof*⟩

lemma *InstanceOf-τred1r-xt*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (e\ instanceof\ T,\ xs)\ (e'\ instanceof\ T,\ xs')$
⟨*proof*⟩

lemma *BinOp-τred1r-xt1*:
$\tau red1gr\ uf\ P\ t\ h\ (e1,\ xs)\ (e1',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (e1\ «bop»\ e2,\ xs)\ (e1'\ «bop»\ e2,\ xs')$
⟨*proof*⟩

lemma *BinOp-τred1r-xt2*:
$\tau red1gr\ uf\ P\ t\ h\ (e2,\ xs)\ (e2',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (Val\ v\ «bop»\ e2,\ xs)\ (Val\ v\ «bop»\ e2',\ xs')$
⟨*proof*⟩

lemma *LAss-τred1r*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (V := e,\ xs)\ (V := e',\ xs')$
⟨*proof*⟩

lemma *AAcc-τred1r-xt1*:
$\tau red1gr\ uf\ P\ t\ h\ (a,\ xs)\ (a',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (a\lfloor i\rceil,\ xs)\ (a'\lfloor i\rceil,\ xs')$
⟨*proof*⟩

lemma *AAcc-τred1r-xt2*:
$\tau red1gr\ uf\ P\ t\ h\ (i,\ xs)\ (i',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (Val\ a\lfloor i\rceil,\ xs)\ (Val\ a\lfloor i'\rceil,\ xs')$
⟨*proof*⟩

lemma *AAss-τred1r-xt1*:
$\tau red1gr\ uf\ P\ t\ h\ (a,\ xs)\ (a',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (a\lfloor i\rceil := e,\ xs)\ (a'\lfloor i\rceil := e,\ xs')$
⟨*proof*⟩

lemma *AAss-τred1r-xt2*:
$\tau red1gr\ uf\ P\ t\ h\ (i,\ xs)\ (i',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (Val\ a\lfloor i\rceil := e,\ xs)\ (Val\ a\lfloor i'\rceil := e,\ xs')$
⟨*proof*⟩

lemma *AAss-τred1r-xt3*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (Val\ a\lfloor Val\ i\rceil := e,\ xs)\ (Val\ a\lfloor Val\ i\rceil := e',\ xs')$
⟨*proof*⟩

lemma *ALength-τred1r-xt*:
$\tau red1gr\ uf\ P\ t\ h\ (a,\ xs)\ (a',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (a{\cdot}length,\ xs)\ (a'{\cdot}length,\ xs')$
⟨*proof*⟩

lemma *FAcc-τred1r-xt*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (e{\cdot}F\{D\},\ xs)\ (e'{\cdot}F\{D\},\ xs')$
⟨*proof*⟩

lemma *FAss-τred1r-xt1*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (e{\cdot}F\{D\} := e2,\ xs)\ (e'{\cdot}F\{D\} := e2,\ xs')$
⟨*proof*⟩

lemma *FAss-τred1r-xt2*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (Val\ v{\cdot}F\{D\} := e,\ xs)\ (Val\ v{\cdot}F\{D\} := e',\ xs')$
⟨*proof*⟩

lemma *CAS-τred1r-xt1*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (e{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3),\ xs)\ (e'{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3),\ xs')$
⟨*proof*⟩

lemma *CAS-τred1r-xt2*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ e,\ e3),\ xs)\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ e',\ e3),\ xs')$
⟨*proof*⟩

lemma *CAS-τred1r-xt3*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v',\ e),\ xs)\ (Val\ v{\cdot}compareAndSwap(D{\cdot}F,\ Val\ v',\ e'),\ xs')$
⟨*proof*⟩

lemma *Call-τred1r-obj*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (e{\cdot}M(ps),\ xs)\ (e'{\cdot}M(ps),\ xs')$
⟨*proof*⟩

lemma *Call-τred1r-param*:
$\tau reds1gr\ uf\ P\ t\ h\ (es,\ xs)\ (es',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (Val\ v{\cdot}M(es),\ xs)\ (Val\ v{\cdot}M(es'),\ xs')$
⟨*proof*⟩

lemma *Block-None-τred1r-xt*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (\{V{:}T{=}None;\ e\},\ xs)\ (\{V{:}T{=}None;\ e'\},\ xs')$
⟨*proof*⟩

lemma *Block-τred1r-Some*:
$[\![\ \tau red1gr\ uf\ P\ t\ h\ (e,\ xs[V := v])\ (e',\ xs');\ V < length\ xs\ ]\!]$
$\Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (\{V{:}Ty{=}\lfloor v\rfloor;\ e\},\ xs)\ (\{V{:}Ty{=}None;\ e'\},\ xs')$
⟨*proof*⟩

lemma *Sync-τred1r-xt*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (sync_V\ (e)\ e2,\ xs)\ (sync_V\ (e')\ e2,\ xs')$
⟨*proof*⟩

lemma *InSync-τred1r-xt*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (insync_V\ (a)\ e,\ xs)\ (insync_V\ (a)\ e',\ xs')$
⟨*proof*⟩

lemma *Seq-τred1r-xt*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (e;;e2,\ xs)\ (e';;e2,\ xs')$
⟨*proof*⟩

lemma *Cond-τred1r-xt*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (if\ (e)\ e1\ else\ e2,\ xs)\ (if\ (e')\ e1\ else\ e2,\ xs')$
⟨*proof*⟩

lemma *Throw-τred1r-xt*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (throw\ e,\ xs)\ (throw\ e',\ xs')$

⟨proof⟩

lemma *Try-τred1r-xt*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow \tau red1gr\ uf\ P\ t\ h\ (try\ e\ catch(C\ V)\ e2,\ xs)\ (try\ e'\ catch(C\ V)\ e2,\ xs')$
⟨proof⟩

lemma *τred1t-ThrowD* [*dest*]: $\tau red1gt\ uf\ P\ t\ h\ (Throw\ a,\ xs)\ (e'',\ xs'') \Longrightarrow e'' = Throw\ a \wedge xs'' = xs$
⟨proof⟩

lemma *τred1r-ThrowD* [*dest*]: $\tau red1gr\ uf\ P\ t\ h\ (Throw\ a,\ xs)\ (e'',\ xs'') \Longrightarrow e'' = Throw\ a \wedge xs'' = xs$
⟨proof⟩

lemma *τRed1-conv* [*iff*]:
$\tau Red1g\ uf\ P\ t\ h\ (ex,\ exs)\ (ex',\ exs') = (uf,P,t \vdash 1\ \langle ex/exs,\ h\rangle\ -\varepsilon\rightarrow\ \langle ex'/exs',\ h\rangle \wedge \tau Move1\ P\ h\ (ex,\ exs))$
⟨proof⟩

lemma *τred1t-into-τRed1t*:
$\tau red1gt\ uf\ P\ t\ h\ (e,\ xs)\ (e'',\ xs'') \Longrightarrow \tau Red1gt\ uf\ P\ t\ h\ ((e,\ xs),\ exs)\ ((e'',\ xs''),\ exs)$
⟨proof⟩

lemma *τred1r-into-τRed1r*:
$\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e'',\ xs'') \Longrightarrow \tau Red1gr\ uf\ P\ t\ h\ ((e,\ xs),\ exs)\ ((e'',\ xs''),\ exs)$
⟨proof⟩

lemma *red1-max-vars*: $uf,P,t \vdash 1\ \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle \Longrightarrow max\text{-}vars\ e' \leq max\text{-}vars\ e$
and *reds1-max-varss*: $uf,P,t \vdash 1\ \langle es,\ s\rangle\ [-ta\rightarrow]\ \langle es',\ s'\rangle \Longrightarrow max\text{-}varss\ es' \leq max\text{-}varss\ es$
⟨proof⟩

lemma *τred1t-max-vars*: $\tau red1gt\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow max\text{-}vars\ e' \leq max\text{-}vars\ e$
⟨proof⟩

lemma *τred1r-max-vars*: $\tau red1gr\ uf\ P\ t\ h\ (e,\ xs)\ (e',\ xs') \Longrightarrow max\text{-}vars\ e' \leq max\text{-}vars\ e$
⟨proof⟩

lemma *τred1r-Val*:
$\tau red1gr\ uf\ P\ t\ h\ (Val\ v,\ xs)\ s' \longleftrightarrow s' = (Val\ v,\ xs)$
⟨proof⟩

lemma *τred1t-Val*:
$\tau red1gt\ uf\ P\ t\ h\ (Val\ v,\ xs)\ s' \longleftrightarrow False$
⟨proof⟩

lemma *τreds1r-map-Val*:
$\tau reds1gr\ uf\ P\ t\ h\ (map\ Val\ vs,\ xs)\ s' \longleftrightarrow s' = (map\ Val\ vs,\ xs)$
⟨proof⟩

lemma *τreds1t-map-Val*:
$\tau reds1gt\ uf\ P\ t\ h\ (map\ Val\ vs,\ xs)\ s' \longleftrightarrow False$
⟨proof⟩

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*⟩

lemma *τreds1t-map-Val-Throw*:
τreds1gt uf P t h (map Val vs @ Throw a # es, xs) s′ ⟷ *False*
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *τred1r-Throw*:
τred1gr uf P t h (Throw a, xs) s′ ⟷ *s′* = *(Throw a, xs)* (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *τred1t-Throw*:
τred1gt uf P t h (Throw a, xs) s′ ⟷ *False* (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

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′*⟩
⟨*proof*⟩

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′*⟩
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Red1-mthr*: *multithreaded final-expr1 (mred1g uf P)*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Red1-τmthr-wf*: *τmultithreaded-wf final-expr1 (mred1g uf P) (τMOVE1 P)*
⟨*proof*⟩

end

sublocale *J1-heap-base* < *Red1-mthr*:
τmultithreaded-wf

final-expr1
mred1g uf P
convert-RA
τMOVE1 P
for *uf P*
⟨*proof*⟩

context *J1-heap-base* **begin**

lemma *τRed1′t-into-Red1′-τmthr-silent-movet*:
τRed1gt uf P t h (*ex2*, *exs2*) (*ex2′′*, *exs2′′*)
$\Longrightarrow$ *Red1-mthr.silent-movet uf P t* ((*ex2*, *exs2*), *h*) ((*ex2′′*, *exs2′′*), *h*)
⟨*proof*⟩

lemma *τRed1t-into-Red1′-τmthr-silent-moves*:
τRed1gt uf P t h (*ex2*, *exs2*) (*ex2′′*, *exs2′′*)
$\Longrightarrow$ *Red1-mthr.silent-moves uf P t* ((*ex2*, *exs2*), *h*) ((*ex2′′*, *exs2′′*), *h*)
⟨*proof*⟩

lemma *τRed1′r-into-Red1′-τmthr-silent-moves*:
τRed1gr uf P t h (*ex*, *exs*) (*ex′*, *exs′*) $\Longrightarrow$ *Red1-mthr.silent-moves uf P t* ((*ex*, *exs*), *h*) ((*ex′*, *exs′*), *h*)
⟨*proof*⟩

lemma *τRed1r-rtranclpD*:
τRed1gr uf P t h s s′ $\Longrightarrow$ *τtrsys.silent-moves* (*mred1g uf P t*) (*τMOVE1 P*) (*s*, *h*) (*s′*, *h*)
⟨*proof*⟩

lemma *τRed1t-tranclpD*:
τRed1gt uf P t h s s′ $\Longrightarrow$ *τtrsys.silent-movet* (*mred1g uf P t*) (*τMOVE1 P*) (*s*, *h*) (*s′*, *h*)
⟨*proof*⟩

lemma *τmreds1-Val-Nil*: *τtrsys.silent-moves* (*mred1g uf P t*) (*τMOVE1 P*) (((*Val v*, *xs*), []), *h*) *s* $\longleftrightarrow$ *s* = (((*Val v*, *xs*), []), *h*)
⟨*proof*⟩

lemma *τmreds1-Throw-Nil*:
τtrsys.silent-moves (*mred1g uf P t*) (*τMOVE1 P*) (((*Throw a*, *xs*), []), *h*) *s* $\longleftrightarrow$ *s* = (((*Throw a*, *xs*), []), *h*)
⟨*proof*⟩

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*⟩

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*′)
⟨*proof*⟩

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*⟩

lemma *Red1-Red1*′*-deadlock-inv*:
Red1-mthr.deadlock True P s = *Red1-mthr.deadlock False P s*
⟨*proof*⟩

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))
⟨*proof*⟩

lemma *class-compP*:
class P C = Some (D, fs, ms)
⟹ class (compP f P) C = Some (D, fs, map (compM (f C)) ms)
⟨*proof*⟩

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
⟨*proof*⟩

lemma [*simp*]: *is-class (compP f P) C = is-class P C*⟨*proof*⟩

lemma [*simp*]: *class (compP f P) C = map-option (λc. snd(compC f (C,c))) (class P C)*⟨*proof*⟩
lemma *sees-methods-compP*:
P ⊢ C sees-methods Mm ⟹
compP f P ⊢ C sees-methods (λM. map-option (λ((Ts,T,m),D). ((Ts,T,map-option (f D M Ts T) m),D)) (Mm M))⟨*proof*⟩

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⟨*proof*⟩

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)⟨*proof*⟩

lemma *sees-methods-compPD*:
⟦ *cP ⊢ C sees-methods Mm′; cP = compP f P* ⟧ *⟹*
∃Mm. P ⊢ C sees-methods Mm ∧
Mm′ = (λM. map-option (λ((Ts,T,m),D). ((Ts,T,map-option (f D M Ts T) m),D)) (Mm M))⟨*proof*⟩

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⟨*proof*⟩
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
⟨*proof*⟩

lemma [*simp*]: *subcls1(compP f P) = subcls1 P*
⟨*proof*⟩

lemma [*simp*]: *is-type (compP f P) T = is-type P T*
⟨*proof*⟩

lemma *is-type-compP* [*simp*]: *is-type* (*compP f P*) = *is-type P*
⟨*proof*⟩

lemma *compP-widen*[*simp*]:
(*compP f P* ⊢ *T* ≤ *T′*) = (*P* ⊢ *T* ≤ *T′*)
⟨*proof*⟩

lemma [*simp*]: (*compP f P* ⊢ *Ts* [≤] *Ts′*) = (*P* ⊢ *Ts* [≤] *Ts′*)⟨*proof*⟩
lemma *is-lub-compP* [*simp*]:
is-lub (*compP f P*) = *is-lub P*
⟨*proof*⟩

lemma [*simp*]:
fixes *f* :: *cname* ⇒ *mname* ⇒ *ty list* ⇒ *ty* ⇒ *′a* ⇒ *′b*
shows (*compP f P* ⊢ *C has-fields FDTs*) = (*P* ⊢ *C has-fields FDTs*)⟨*proof*⟩

lemma [*simp*]: *fields* (*compP f P*) *C* = *fields P C*⟨*proof*⟩

lemma [*simp*]: (*compP f P* ⊢ *C sees F*:*T* (*fm*) *in D*) = (*P* ⊢ *C sees F*:*T* (*fm*) *in D*)⟨*proof*⟩

lemma [*simp*]: *field* (*compP f P*) *F D* = *field P F D*⟨*proof*⟩

7.8.1 Invariance of *wf-prog* under compilation

lemma [*iff*]: *distinct-fst* (*classes* (*compP f P*)) = *distinct-fst* (*classes P*)⟨*proof*⟩

lemma [*iff*]: *distinct-fst* (*map* (*compM f*) *ms*) = *distinct-fst ms*⟨*proof*⟩

lemma [*iff*]: *wf-syscls* (*compP f P*) = *wf-syscls P*
⟨*proof*⟩

lemma [*iff*]: *wf-fdecl* (*compP f P*) = *wf-fdecl P*⟨*proof*⟩

lemma *set-compP*:
(*class* (*compP f P*) *C* = ⌊(*D*,*fs*,*ms′*)⌋) ⟷
(∃ *ms*. *class P C* = ⌊(*D*,*fs*,*ms*)⌋ ∧ *ms′* = *map* (*compM* (*f C*)) *ms*)
⟨*proof*⟩

lemma *compP-has-method*: *compP f P* ⊢ *C has M* ⟷ *P* ⊢ *C has M*
⟨*proof*⟩

lemma *is-native-compP* [*simp*]: *is-native* (*compP f P*) = *is-native P*
⟨*proof*⟩

lemma *τexternal-compP* [*simp*]:
τexternal (*compP f P*) = *τexternal P*
⟨*proof*⟩

context *heap-base* **begin**

lemma *heap-clone-compP* [*simp*]:
heap-clone (*compP f P*) = *heap-clone P*
⟨*proof*⟩

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*′⟩
⟨*proof*⟩

lemma *τexternal′-compP* [*simp*]:
τexternal′ (*compP f P*) = *τexternal′ P*
⟨*proof*⟩

end

lemma *wf-overriding-compP* [*simp*]: *wf-overriding* (*compP f P*) *D* (*compM* (*f C*) *m*) = *wf-overriding P D m*
⟨*proof*⟩

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* wf_2 (*compP f P*) *C* (*M*,*Ts*,*T*, ⌊*f C M Ts T m*⌋)
and *wfcP1*: ∀ *C rest*. *class P C* = ⌊*rest*⌋ ⟶ *wf-cdecl* wf_1 *P* (*C*, *rest*)
and *xcomp*: *class* (*compP f P*) *C* = ⌊*rest′*⌋
and *wf*: *wf-prog p P*
shows *wf-cdecl* wf_2 (*compP f P*) (*C*, *rest′*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *WT-binop-compP* [*simp*]: *compP f P* ⊢ *T1* «*bop*» *T2* :: *T* ⟷ *P* ⊢ *T1* «*bop*» *T2* :: *T*
⟨*proof*⟩

lemma *WTrt-binop-compP* [*simp*]: *compP f P* ⊢ *T1* «*bop*» *T2* : *T* ⟷ *P* ⊢ *T1* «*bop*» *T2* : *T*
⟨*proof*⟩

lemma *binop-relevant-class-compP* [*simp*]: *binop-relevant-class bop* (*compP f P*) = *binop-relevant-class bop P*
⟨*proof*⟩

lemma *is-class-compP* [*simp*]:
is-class (*compP f P*) = *is-class P*
⟨*proof*⟩

lemma *has-field-compP* [*simp*]:
compP f P ⊢ *C has F*:*T* (*fm*) *in D* ⟷ *P* ⊢ *C has F*:*T* (*fm*) *in D*
⟨*proof*⟩

context *heap-base* **begin**

lemma *compP-addr-loc-type* [*simp*]:
addr-loc-type (*compP f P*) = *addr-loc-type P*
⟨*proof*⟩

lemma *conf-compP* [*simp*]:
compP f P,*h* ⊢ *v* :≤ *T* ⟷ *P*,*h* ⊢ *v* :≤ *T*
⟨*proof*⟩

lemma *compP-conf*: *conf* (*compP f P*) = *conf P*
⟨*proof*⟩

lemma *compP-confs*: *compP f P*,*h* ⊢ *vs* [:≤] *Ts* ⟷ *P*,*h* ⊢ *vs* [:≤] *Ts*
⟨*proof*⟩

lemma *tconf-compP* [*simp*]: *compP f P*, *h* ⊢ *t* √*t* ⟷ *P*,*h* ⊢ *t* √*t*
⟨*proof*⟩

lemma *wf-start-state-compP* [*simp*]:
wf-start-state (*compP f P*) = *wf-start-state P*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

compiler composition

lemma *compM-compM*:
compM f (*compM g md*) = *compM* (λ*M Ts T*. *f M Ts T* ∘ *g M Ts T*) *md*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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) e₁ else e₂) =
      (let cnd   = compE2 e;
           thn   = compE2 e₁;
           els   = compE2 e₂;
           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) e_1 else e_2) pc d =*
(*let pc_1 = pc + size(compE2 e) + 1;*
pc_2 = pc_1 + size(compE2 e_1) + 1
in compxE2 e pc d @ compxE2 e_1 pc_1 d @ compxE2 e_2 pc_2 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 ≠ []*
⟨*proof*⟩

declare *compE2-neq-Nil*[*symmetric*, *simp*]

lemma *compEs2-append* [*simp*]: *compEs2 (es @ es′) = compEs2 es @ compEs2 es′*
⟨*proof*⟩

lemma *compEs2-eq-Nil-conv* [*simp*]: *compEs2 es = [] ⟷ es = []*
⟨*proof*⟩

lemma *compEs2-map-Val*: *compEs2 (map Val vs) = map Push vs*
⟨*proof*⟩

lemma *compE2-0th-neq-Invoke* [*simp*]:
compE2 e ! 0 ≠ Invoke M n
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *compxEs2-map-Val* [*simp*]: *compxEs2 (map Val vs) pc d = []*

⟨*proof*⟩

lemma *compE2-blocks1* [*simp*]:
compE2 (*blocks1 n Ts body*) = *compE2 body*
⟨*proof*⟩

lemma *compxE2-blocks1* [*simp*]:
compxE2 (*blocks1 n Ts body*) = *compxE2 body*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟨*proof*⟩
lemma *max-stacks-ge-length*: *max-stacks es* ≥ *length es*
⟨*proof*⟩

lemma *max-stack-blocks1* [*simp*]:
max-stack (*blocks1 n Ts body*) = *max-stack body*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)}
⟨*proof*⟩

lemma *pcs-Nil* [*simp*]: *pcs* [] = {}
⟨*proof*⟩

lemma *pcs-Cons* [*simp*]: *pcs* (*x*#*xt*) = {*fst x* ..< *fst*(*snd x*)} ∪ *pcs xt*
⟨*proof*⟩

lemma *pcs-append* [*simp*]: $pcs(xt_1 \ @\ xt_2) = pcs\ xt_1 \cup pcs\ xt_2$
⟨*proof*⟩

lemma [*simp*]: $pc < pc_0 \vee pc_0 + size(compE2\ e) \leq pc \Longrightarrow pc \notin pcs(compxE2\ e\ pc_0\ d)$
⟨*proof*⟩

lemma [*simp*]: *pc* < *pc0* ∨ *pc0*+*size*(*compEs2 es*) ≤ *pc* ⟹ *pc* ∉ *pcs*(*compxEs2 es pc0 d*)
⟨*proof*⟩

lemma [*simp*]: *pc1* + *size*(*compE2 e1*) ≤ *pc2* ⟹ *pcs*(*compxE2 e1 pc1 d1*) ∩ *pcs*(*compxE2 e2 pc2 d2*) = {}
⟨*proof*⟩

lemma [*simp*]: pc_1 + *size*(*compE2 e*) ≤ pc_2 ⟹ *pcs*(*compxE2 e* pc_1 d_1) ∩ *pcs*(*compxEs2 es* pc_2 d_2) = {}
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *outside-pcs-not-matches-entry* [*simp*]:
⟦ *x* ∈ *set xt*; *pc* ∉ *pcs xt* ⟧ ⟹ ¬ *matches-ex-entry P D pc x*
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *match-ex-table-not-pcs-None*:
pc ∉ *pcs xt* ⟹ *match-ex-table P C pc xt* = *None*
⟨*proof*⟩

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* $\preceq^*$ *C′*))
⟨*proof*⟩

lemma *pcs-compxE2D* [*dest*]:
pc ∈ *pcs* (*compxE2 e pc′ d*) ⟹ *pc′* ≤ *pc* ∧ *pc* < *pc′* + *length* (*compE2 e*)
⟨*proof*⟩

lemma *pcs-compxEs2D* [*dest*]:
pc ∈ *pcs* (*compxEs2 es pc′ d*) ⟹ *pc′* ≤ *pc* ∧ *pc* < *pc′* + *length* (*compEs2 es*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *shift-Nil* [*simp*]: *shift n* [] = []
⟨*proof*⟩

lemma *shift-Cons-tuple* [*simp*]:
shift n ((*from*, *to*, *C*, *handler*, *depth*) # *xt*) = (*from* + *n*, *to* + *n*, *C*, *handler* + *n*, *depth*) # *shift n xt*
⟨*proof*⟩

lemma *shift-append* [*simp*]: *shift n* (xt_1 @ xt_2) = *shift n* xt_1 @ *shift n* xt_2
⟨*proof*⟩

lemma *shift-shift* [*simp*]: *shift m* (*shift n xt*) = *shift* (*m*+*n*) *xt*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *pcs-shift-conv* [*simp*]: *pcs* (*shift n xt*) = (+) *n* ‘ *pcs xt*
⟨*proof*⟩

lemma *image-plus-const-conv* [*simp*]:
fixes *m* :: *nat*
shows *m* ∈ (+) *n* ‘ *A* ⟷ *m* ≥ *n* ∧ *m* − *n* ∈ *A*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)⌋
⟨*proof*⟩

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*)⌋
⟨*proof*⟩

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*)⌋
⟨*proof*⟩

lemma *match-ex-table-pcsD*: *match-ex-table P C pc xt* = ⌊(*pc*′, *D*)⌋ ⟹ *pc* ∈ *pcs xt*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *stack-xlift-Nil* [*simp*]: *stack-xlift n* [] = []
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *stack-xlift-append* [*simp*]: *stack-xlift n* (*xt* @ *xt*′) = *stack-xlift n xt* @ *stack-xlift n xt*′
⟨*proof*⟩

lemma *stack-xlift-stack-xlift* [*simp*]: *stack-xlift n* (*stack-xlift m xt*) = *stack-xlift* (*n* + *m*) *xt*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *stack-xlift-shift* [*simp*]: *stack-xlift d* (*shift n xt*) = *shift n* (*stack-xlift d xt*)
⟨*proof*⟩

lemma *pcs-stack-xlift-conv* [*simp*]: *pcs* (*stack-xlift n xt*) = *pcs xt*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)⌋
⟨*proof*⟩

lemma *match-ex-table-stack-xliftD*:
$match\text{-}ex\text{-}table\ P\ C\ pc\ (stack\text{-}xlift\ n\ xt) = \lfloor(pc',\ d)\rfloor \Longrightarrow d \geq n \wedge match\text{-}ex\text{-}table\ P\ C\ pc\ xt = \lfloor(pc',\ d - n)\rfloor$
⟨proof⟩

lemma *match-ex-table-stack-xlift*:
$match\text{-}ex\text{-}table\ P\ C\ pc\ xt = \lfloor(pc',\ d)\rfloor \Longrightarrow match\text{-}ex\text{-}table\ P\ C\ pc\ (stack\text{-}xlift\ n\ xt) = \lfloor(pc',\ n + d)\rfloor$
⟨proof⟩

lemma *pcs-stack-xlift*: $pcs\ (stack\text{-}xlift\ n\ xt) = pcs\ xt$
⟨proof⟩

lemma *match-ex-table-None-append* [*simp*]:
$match\text{-}ex\text{-}table\ P\ C\ pc\ xt = None$
$\Longrightarrow match\text{-}ex\text{-}table\ P\ C\ pc\ (xt\ @\ xt') = match\text{-}ex\text{-}table\ P\ C\ pc\ xt'$
⟨proof⟩

lemma *match-ex-table-Some-append* [*simp*]:
$match\text{-}ex\text{-}table\ P\ C\ pc\ xt = \lfloor(pc',\ d)\rfloor \Longrightarrow match\text{-}ex\text{-}table\ P\ C\ pc\ (xt\ @\ xt') = \lfloor(pc',\ d)\rfloor$
⟨proof⟩

lemma *match-ex-table-append*:
$match\text{-}ex\text{-}table\ P\ C\ pc\ (xt\ @\ xt') = (case\ match\text{-}ex\text{-}table\ P\ C\ pc\ xt\ of\ None \Rightarrow match\text{-}ex\text{-}table\ P\ C\ pc\ xt'$
$\mid Some\ pcd \Rightarrow Some\ pcd)$
⟨proof⟩

lemma *match-ex-table-pc-length-compE2*:
$match\text{-}ex\text{-}table\ P\ a\ pc\ (compxE2\ e\ pc'\ d) = \lfloor pcd\rfloor \Longrightarrow pc' \leq pc \wedge pc < length\ (compE2\ e) + pc'$

and *match-ex-table-pc-length-compEs2*:
$match\text{-}ex\text{-}table\ P\ a\ pc\ (compxEs2\ es\ pc'\ d) = \lfloor pcd\rfloor \Longrightarrow pc' \leq pc \wedge pc < length\ (compEs2\ es) + pc'$
⟨proof⟩

lemma *match-ex-table-compxE2-shift-conv*:
$f > 0 \Longrightarrow match\text{-}ex\text{-}table\ P\ C\ pc\ (compxE2\ e\ f\ d) = \lfloor(pc',\ d')\rfloor \longleftrightarrow pc \geq f \wedge pc' \geq f \wedge match\text{-}ex\text{-}table\ P\ C\ (pc - f)\ (compxE2\ e\ 0\ d) = \lfloor(pc' - f,\ d')\rfloor$
⟨proof⟩

lemma *match-ex-table-compxEs2-shift-conv*:
$f > 0 \Longrightarrow match\text{-}ex\text{-}table\ P\ C\ pc\ (compxEs2\ es\ f\ d) = \lfloor(pc',\ d')\rfloor \longleftrightarrow pc \geq f \wedge pc' \geq f \wedge match\text{-}ex\text{-}table\ P\ C\ (pc - f)\ (compxEs2\ es\ 0\ d) = \lfloor(pc' - f,\ d')\rfloor$
⟨proof⟩

lemma *match-ex-table-compxE2-stack-conv*:
$d > 0 \Longrightarrow match\text{-}ex\text{-}table\ P\ C\ pc\ (compxE2\ e\ 0\ d) = \lfloor(pc',\ d')\rfloor \longleftrightarrow d' \geq d \wedge match\text{-}ex\text{-}table\ P\ C\ pc\ (compxE2\ e\ 0\ 0) = \lfloor(pc',\ d' - d)\rfloor$
⟨proof⟩

lemma *match-ex-table-compxEs2-stack-conv*:
$d > 0 \Longrightarrow match\text{-}ex\text{-}table\ P\ C\ pc\ (compxEs2\ es\ 0\ d) = \lfloor(pc',\ d')\rfloor \longleftrightarrow d' \geq d \wedge match\text{-}ex\text{-}table\ P\ C\ pc\ (compxEs2\ es\ 0\ 0) = \lfloor(pc',\ d' - d)\rfloor$
⟨proof⟩

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*) = ⌊(*pc'*, *d'*)⌋ ⟹ *pc* ≠ *pc'*
and *match-ex-table-compxEs2-not-same*:*match-ex-table P C pc* (*compxEs2 es n d*) = ⌊(*pc'*, *d'*)⌋ ⟹ *pc* ≠ *pc'*
⟨*proof*⟩

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* ⇒ *env1* ⇒ *'addr expr1* ⇒ *ty* ⇒ *bool* (‹-,- ⊢1 - :: -› [*51*,*0*,*0*,*51*] *50*)
and *WTs1* :: *'addr J1-prog* ⇒ *env1* ⇒ *'addr expr1 list* ⇒ *ty list* ⇒ *bool* (‹-,- ⊢1 - [::] -› [*51*,*0*,*0*,*51*]*50*)
for *P* :: *'addr J1-prog*
where

WT1New:
is-class P C ⟹
P,*E* ⊢*1 new C* :: *Class C*

| *WT1NewArray*:
⟦ *P*,*E* ⊢*1 e* :: *Integer*; *is-type P* (*T*⌊⌉) ⟧ ⟹
P,*E* ⊢*1 newA T*⌊*e*⌉ :: *T*⌊⌉

| *WT1Cast*:
⟦ *P*,*E* ⊢*1 e* :: *T*; *P* ⊢ *U* ≤ *T* ∨ *P* ⊢ *T* ≤ *U*; *is-type P U* ⟧
⟹ *P*,*E* ⊢*1 Cast U e* :: *U*

| *WT1InstanceOf*:
⟦ *P*,*E* ⊢*1 e* :: *T*; *P* ⊢ *U* ≤ *T* ∨ *P* ⊢ *T* ≤ *U*; *is-type P U*; *is-refT U* ⟧
⟹ *P*,*E* ⊢*1 e instanceof U* :: *Boolean*

| *WT1Val*:
typeof v = *Some T* ⟹
P,*E* ⊢*1 Val v* :: *T*

| *WT1Var*:
⟦ *E*!*V* = *T*; *V* < *size E* ⟧ ⟹
P,*E* ⊢*1 Var V* :: *T*

| *WT1BinOp*:

$\llbracket$ $P,E \vdash 1\ e1 :: T1$; $P,E \vdash 1\ e2 :: T2$; $P \vdash T1 «bop» T2 :: T$ $\rrbracket$
$\Longrightarrow$ $P,E \vdash 1\ e1 «bop» e2 :: T$

| *WT1LAss*:
$\llbracket$ $E!i = T$; $i < size\ E$; $P,E \vdash 1\ e :: T'$; $P \vdash T' \leq T$ $\rrbracket$
$\Longrightarrow$ $P,E \vdash 1\ i{:=}e :: Void$

| *WT1AAcc*:
$\llbracket$ $P,E \vdash 1\ a :: T\lfloor\rceil$; $P,E \vdash 1\ i :: Integer$ $\rrbracket$
$\Longrightarrow$ $P,E \vdash 1\ a\lfloor i\rceil :: T$

| *WT1AAss*:
$\llbracket$ $P,E \vdash 1\ a :: T\lfloor\rceil$; $P,E \vdash 1\ i :: Integer$; $P,E \vdash 1\ e :: T'$; $P \vdash T' \leq T$ $\rrbracket$
$\Longrightarrow$ $P,E \vdash 1\ a\lfloor i\rceil := e :: Void$

| *WT1ALength*:
$P,E \vdash 1\ a :: T\lfloor\rceil \Longrightarrow P,E \vdash 1\ a{\cdot}length :: Integer$

| *WTFAcc1*:
$\llbracket$ $P,E \vdash 1\ e :: U$; $class\text{-}type\text{-}of'\ U = \lfloor C\rfloor$; $P \vdash C\ sees\ F{:}T\ (fm)\ in\ D$ $\rrbracket$
$\Longrightarrow$ $P,E \vdash 1\ e{\cdot}F\{D\} :: T$

| *WTFAss1*:
$\llbracket$ $P,E \vdash 1\ e1 :: U$; $class\text{-}type\text{-}of'\ U = \lfloor C\rfloor$; $P \vdash C\ sees\ F{:}T\ (fm)\ in\ D$; $P,E \vdash 1\ e2 :: T'$; $P \vdash T'$
$\leq T$ $\rrbracket$
$\Longrightarrow$ $P,E \vdash 1\ e1{\cdot}F\{D\} := e2 :: Void$

| *WTCAS1*:
$\llbracket$ $P,E \vdash 1\ e1 :: U$; $class\text{-}type\text{-}of'\ U = \lfloor C\rfloor$; $P \vdash C\ sees\ F{:}T\ (fm)\ in\ D$; $volatile\ fm$;
$P,E \vdash 1\ e2 :: T'$; $P \vdash T' \leq T$; $P,E \vdash 1\ e3 :: T''$; $P \vdash T'' \leq T$ $\rrbracket$
$\Longrightarrow$ $P,E \vdash 1\ e1{\cdot}compareAndSwap(D{\cdot}F,\ e2,\ e3) :: Boolean$

| *WT1Call*:
$\llbracket$ $P,E \vdash 1\ e :: U$; $class\text{-}type\text{-}of'\ U = \lfloor C\rfloor$; $P \vdash C\ sees\ M{:}Ts \rightarrow T = m\ in\ D$;
$P,E \vdash 1\ es\ [::]\ Ts'$; $P \vdash Ts'\ [\leq]\ Ts$ $\rrbracket$
$\Longrightarrow$ $P,E \vdash 1\ e{\cdot}M(es) :: T$

| *WT1Block*:
$\llbracket$ $is\text{-}type\ P\ T$; $P,E@[T] \vdash 1\ e :: T'$; $case\ vo\ of\ None \Rightarrow True \mid \lfloor v\rfloor \Rightarrow \exists\, T'.\ typeof\ v = \lfloor T'\rfloor \wedge P \vdash$
$T' \leq T$ $\rrbracket$
$\Longrightarrow$ $P,E \vdash 1\ \{V{:}T{=}vo;\ e\} :: T'$

| *WT1Synchronized*:
$\llbracket$ $P,E \vdash 1\ o' :: T$; $is\text{-}refT\ T$; $T \neq NT$; $P,E@[Class\ Object] \vdash 1\ e :: T'$ $\rrbracket$
$\Longrightarrow$ $P,E \vdash 1\ sync_V\ (o')\ e :: T'$

| *WT1Seq*:
$\llbracket$ $P,E \vdash 1\ e_1{::}T_1$; $P,E \vdash 1\ e_2{::}T_2$ $\rrbracket$
$\Longrightarrow$ $P,E \vdash 1\ e_1;;e_2 :: T_2$

| *WT1Cond*:
$\llbracket$ $P,E \vdash 1\ e :: Boolean$; $P,E \vdash 1\ e_1{::}T_1$; $P,E \vdash 1\ e_2{::}T_2$; $P \vdash lub(T_1,T_2) = T$ $\rrbracket$
$\Longrightarrow$ $P,E \vdash 1\ if\ (e)\ e_1\ else\ e_2 :: T$

| *WT1While*:
  ⟦ $P,E \vdash 1\ e :: Boolean$; $P,E \vdash 1\ c::T$ ⟧
  $\Longrightarrow$ $P,E \vdash 1$ *while* (e) c :: *Void*

| *WT1Throw*:
  ⟦ $P,E \vdash 1\ e :: Class\ C$; $P \vdash C \preceq^* Throwable$ ⟧ $\Longrightarrow$
  $P,E \vdash 1$ *throw* e :: *Void*

| *WT1Try*:
  ⟦ $P,E \vdash 1\ e_1 :: T$; $P,E@[Class\ C] \vdash 1\ e_2 :: T$; *is-class P C* ⟧
  $\Longrightarrow$ $P,E \vdash 1$ *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* ⟧ $\Longrightarrow$ $P,E \vdash 1$ *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∙F*{*D*} :: T
  $P,E \vdash 1$ *e1∙F*{*D*}:=*e2* :: T
  $P,E \vdash 1$ *e∙compareAndSwap*(*D∙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'*⌊*e*⌉ :: T
  $P,E \vdash 1$ *a*⌊*i*⌉ := *e* :: T
  $P,E \vdash 1$ *a*⌊*i*⌉ :: T
  $P,E \vdash 1$ *a∙length* :: T
  $P,E \vdash 1$ *e∙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*
⟨*proof*⟩

lemma *WTs1-snoc-cases*:
  assumes *wt*: $P,E \vdash 1$ *es* @ [*e*] [::] *Ts*
  obtains *T Ts'* **where** $P,E \vdash 1$ *es* [::] *Ts'* $P,E \vdash 1\ e :: T$
⟨*proof*⟩

lemma *WTs1-append*:
assumes *wt*: $P,Env \vdash 1\ es\ @\ es'$ [::] *Ts*
obtains $Ts'\ Ts''$ **where** $P,Env \vdash 1\ es$ [::] Ts' $P,Env \vdash 1\ es'$ [::] Ts''
⟨*proof*⟩

lemma *WT1-not-contains-insync*: $P,E \vdash 1\ e :: T \Longrightarrow \neg$ *contains-insync e*
and *WTs1-not-contains-insyncs*: $P,E \vdash 1\ es$ [::] $Ts \Longrightarrow \neg$ *contains-insyncs es*
⟨*proof*⟩

lemma *WT1-expr-locks*: $P,E \vdash 1\ e :: T \Longrightarrow$ *expr-locks e* $= (\lambda a.\ 0)$
and *WTs1-expr-lockss*: $P,E \vdash 1\ es$ [::] $Ts \Longrightarrow$ *expr-lockss es* $= (\lambda a.\ 0)$
⟨*proof*⟩

lemma assumes *wf*: *wf-prog wfmd P*
shows *WT1-unique*: $P,E \vdash 1\ e :: T1 \Longrightarrow P,E \vdash 1\ e :: T2 \Longrightarrow T1 = T2$
and *WTs1-unique*: $P,E \vdash 1\ es$ [::] $Ts1 \Longrightarrow P,E \vdash 1\ es$ [::] $Ts2 \Longrightarrow Ts1 = Ts2$
⟨*proof*⟩

lemma assumes *wf*: *wf-prog p P*
shows *WT1-is-type*: $P,E \vdash 1\ e :: T \Longrightarrow set\ E \subseteq types\ P \Longrightarrow$ *is-type P T*
and *WTs1-is-type*: $P,E \vdash 1\ es$ [::] $Ts \Longrightarrow set\ E \subseteq types\ P \Longrightarrow set\ Ts \subseteq types\ P$
⟨*proof*⟩

lemma *blocks1-WT*:
$[\![\ P,Env\ @\ Ts \vdash 1\ body :: T;\ set\ Ts \subseteq types\ P\ ]\!] \Longrightarrow P,Env \vdash 1$ *blocks1* (*length Env*) *Ts body* :: *T*
⟨*proof*⟩

lemma *WT1-fv*: $[\![\ P,E \vdash 1\ e :: T;\ \mathcal{B}\ e\ (length\ E);\ syncvars\ e\ ]\!] \Longrightarrow fv\ e \subseteq \{0..<length\ E\}$
and *WTs1-fvs*: $[\![\ P,E \vdash 1\ es$ [::] $Ts;\ \mathcal{B}s\ es\ (length\ E);\ syncvarss\ es\ ]\!] \Longrightarrow fvs\ es \subseteq \{0..<length\ E\}$
⟨*proof*⟩

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 $\equiv$ $\lambda(M,Ts,T,body)$.
$(\exists T'.\ P,Class\ C\#Ts \vdash 1\ body :: T' \wedge P \vdash T' \leq T) \wedge$
$\mathcal{D}\ body\ \lfloor\{..size\ Ts\}\rfloor \wedge \mathcal{B}\ body\ (size\ Ts + 1) \wedge syncvars\ body$

lemma *wf-J1-mdecl*[*simp*]:
wf-J1-mdecl P C $(M,Ts,T,body) \equiv$
$((\exists T'.\ P,Class\ C\#Ts \vdash 1\ body :: T' \wedge P \vdash T' \leq T) \wedge$
$\mathcal{D}\ body\ \lfloor\{..size\ Ts\}\rfloor \wedge \mathcal{B}\ body\ (size\ Ts + 1)) \wedge syncvars\ body$
⟨*proof*⟩

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*
⟨*proof*⟩

end

context *TC0* **begin**

lemma ty_i'*-None* [*simp*]: ty_i' *ST E None* = *None*
⟨*proof*⟩

lemma ty_l*-app-diff* [*simp*]:
ty_l (*E*@[*T*]) (*A* − {*size E*}) = ty_l *E A*
⟨*proof*⟩

lemma $ty_i{}'$-*app-diff* [*simp*]:
$ty_i{}'\ ST\ (E\ @\ [T])\ (A \ominus size\ E) = ty_i{}'\ ST\ E\ A$
⟨*proof*⟩

lemma ty_l-*antimono*:
$A \subseteq A' \Longrightarrow P \vdash ty_l\ E\ A'\ [\le_\top]\ ty_l\ E\ A$
⟨*proof*⟩

lemma $ty_i{}'$-*antimono*:
$A \subseteq A' \Longrightarrow P \vdash ty_i{}'\ ST\ E\ \lfloor A' \rfloor \le' ty_i{}'\ ST\ E\ \lfloor A \rfloor$
⟨*proof*⟩

lemma ty_l-*env-antimono*:
$P \vdash ty_l\ (E@[T])\ A\ [\le_\top]\ ty_l\ E\ A$
⟨*proof*⟩

lemma $ty_i{}'$-*env-antimono*:
$P \vdash ty_i{}'\ ST\ (E@[T])\ A \le' ty_i{}'\ ST\ E\ A$
⟨*proof*⟩

lemma $ty_i{}'$-*incr*:
$P \vdash ty_i{}'\ ST\ (E\ @\ [T])\ \lfloor insert\ (size\ E)\ A \rfloor \le' ty_i{}'\ ST\ E\ \lfloor A \rfloor$
⟨*proof*⟩

lemma ty_l-*incr*:
$P \vdash ty_l\ (E\ @\ [T])\ (insert\ (size\ E)\ A)\ [\le_\top]\ ty_l\ E\ A$
⟨*proof*⟩

lemma ty_l-*in-types*:
$set\ E \subseteq types\ P \Longrightarrow ty_l\ E\ A \in list\ mxl\ (err\ (types\ P))$
⟨*proof*⟩

function $compT :: ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow 'addr\ expr1 \Rightarrow ty_i{}'\ list$
  and $compTs :: ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow 'addr\ expr1\ list \Rightarrow ty_i{}'\ list$
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⌊i⌉ := e) =
(let ST1 = ty E a # ST; A1 = A ⊔ 𝒜 a;
ST2 = ty E i # ST1; A2 = A1 ⊔ 𝒜 i; A3 = A2 ⊔ 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*]) ($A \sqcup \lfloor\{i\}\rfloor$)] @
compT (*E@*[*Class C*]) ($A \sqcup \lfloor\{i\}\rfloor$) *ST e2*

| *compTs E A ST* [] = []
| *compTs E A ST* (*e#es*) = *compT E A ST e* @ [*after E A ST e*] @
compTs E ($A \sqcup (\mathcal{A}\ e)$) (*ty E e # ST*) *es*
⟨*proof*⟩
termination
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *compT-None-not-Some* [*simp*]: $\lfloor\tau\rfloor \notin$ *set* (*compT E None ST e*)
and *compTs-None-not-Some* [*simp*]: $\lfloor\tau\rfloor \notin$ *set* (*compTs E None ST es*)
⟨*proof*⟩

lemma *pair-eq-*ty_i'*-conv*:
($\lfloor(ST, LT)\rfloor = ty_i'\ ST_0\ E\ A$) = (*case A of None* ⇒ *False* | *Some A* ⇒ ($ST = ST_0 \wedge LT = ty_l\ E\ A$))
⟨*proof*⟩

lemma *pair-conv-*ty_i': $\lfloor(ST,\ ty_l\ E\ A)\rfloor = ty_i'\ ST\ E\ \lfloor A\rfloor$
⟨*proof*⟩

lemma ty_i'*-antimono2*:
$[\![\ E \le E';\ A \subseteq A'\ ]\!] \Longrightarrow P \vdash ty_i'\ ST\ E'\ \lfloor A'\rfloor \le' ty_i'\ ST\ E\ \lfloor A\rfloor$
⟨*proof*⟩

declare ty_i'*-antimono* [*intro*!] *after-def*[*simp*] *pair-conv-*ty_i'[*simp*] *pair-eq-*ty_i'*-conv*[*simp*]

lemma *compT-LT-prefix*:
$[\![\ \lfloor(ST,LT)\rfloor \in set(compT\ E\ A\ ST0\ e);\ \mathcal{B}\ e\ (size\ E)\ ]\!] \Longrightarrow P \vdash \lfloor(ST,LT)\rfloor \le' ty_i'\ ST\ E\ A$
and *compTs-LT-prefix*:
$[\![\ \lfloor(ST,LT)\rfloor \in set(compTs\ E\ A\ ST0\ es);\ \mathcal{B}s\ es\ (size\ E)\ ]\!] \Longrightarrow P \vdash \lfloor(ST,LT)\rfloor \le' ty_i'\ ST\ E\ A$
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

end

lemma *is-class-type-aux*: *is-class P C* ⟹ *is-type P* (*Class C*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

end

locale *TC3* = *TC1* + *TC2*

lemma *eff-None* [*simp*]: *eff i P pc et None* = []
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *relevant-entries-shift* [*simp*]:
relevant-entries P i (*pc*+*n*) (*shift n xt*) = *shift n* (*relevant-entries P i pc xt*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *xcpt-app-shift* [*simp*]:
xcpt-app i P (*pc*+*n*) *m* (*shift n xt*) *τ* = *xcpt-app i P pc m xt τ*
⟨*proof*⟩

lemma *wt-instr-appL*:
⟦ *P*,*T*,*m*,*mpc*,*xt* ⊢ *i*,*pc* :: *τs*; *pc* < *size τs*; *mpc* ≤ *size τs* ⟧

$\Longrightarrow$ $P,T,m,mpc + size\ \tau s',shift\ (size\ \tau s')\ xt \vdash i,pc+size\ \tau s' :: \tau s'@\tau s$
⟨proof⟩

lemma *wt-instr-Cons*:
$[\![\ P,T,m,mpc - 1,[] \vdash i,pc - 1 :: \tau s;$
$0 < pc;\ 0 < mpc;\ pc < size\ \tau s + 1;\ mpc \leq size\ \tau s + 1\ ]\!]$
$\Longrightarrow P,T,m,mpc,[] \vdash i,pc :: \tau\#\tau s$
⟨proof⟩

lemma *wt-instr-append*:
$[\![\ P,T,m,mpc - size\ \tau s',[] \vdash i,pc - size\ \tau s' :: \tau s;$
$size\ \tau s' \leq pc;\ size\ \tau s' \leq mpc;\ pc < size\ \tau s + size\ \tau s';\ mpc \leq size\ \tau s + size\ \tau s'\ ]\!]$
$\Longrightarrow P,T,m,mpc,[] \vdash i,pc :: \tau s'@\tau s$
⟨proof⟩

lemma *xcpt-app-pcs*:
$pc \notin pcs\ xt \Longrightarrow xcpt\text{-}app\ i\ P\ pc\ mxs\ xt\ \tau$
⟨proof⟩

lemma *xcpt-eff-pcs*:
$pc \notin pcs\ xt \Longrightarrow xcpt\text{-}eff\ i\ P\ pc\ \tau\ xt = []$
⟨proof⟩

lemma *pcs-shift*:
$pc < n \Longrightarrow pc \notin pcs\ (shift\ n\ xt)$
⟨proof⟩

lemma *xcpt-eff-shift-pc-ge-n*: **assumes** $x \in set\ (xcpt\text{-}eff\ i\ P\ pc\ \tau\ (shift\ n\ xt))$
shows $n \leq pc$
⟨proof⟩

lemma *wt-instr-appRx*:
$[\![\ P,T,m,mpc,xt \vdash is!pc,pc :: \tau s;\ pc < size\ is;\ size\ is < size\ \tau s;\ mpc \leq size\ \tau s\ ]\!]$
$\Longrightarrow P,T,m,mpc,xt\ @\ shift\ (size\ is)\ xt' \vdash is!pc,pc :: \tau s$
⟨proof⟩

lemma *wt-instr-appLx*:
$[\![\ P,T,m,mpc,xt \vdash i,pc :: \tau s;\ pc \notin pcs\ xt'\ ]\!]$
$\Longrightarrow P,T,m,mpc,xt'@xt \vdash i,pc :: \tau s$
⟨proof⟩

context *TC2* **begin**

lemma *wt-instrs-extR*:
$\vdash is,xt\ [::]\ \tau s \Longrightarrow \vdash is,xt\ [::]\ \tau s\ @\ \tau s'$
⟨proof⟩

lemma *wt-instrs-ext*:
$[\![\vdash is_1,xt_1$ [::] $\tau s_1 @ \tau s_2$; $\vdash is_2,xt_2$ [::] τs_2; *size* τs_1 = *size* $is_1]\!]$
$\Longrightarrow \vdash is_1 @ is_2$, xt_1 @ *shift* (*size* is_1) xt_2 [::] $\tau s_1 @ \tau s_2$
⟨*proof*⟩

corollary *wt-instrs-ext2*:
$[\![\vdash is_2,xt_2$ [::] τs_2; $\vdash is_1,xt_1$ [::] $\tau s_1 @ \tau s_2$; *size* τs_1 = *size* $is_1]\!]$
$\Longrightarrow \vdash is_1 @ is_2$, xt_1 @ *shift* (*size* is_1) xt_2 [::] $\tau s_1 @ \tau s_2$
⟨*proof*⟩

corollary *wt-instrs-ext-prefix* [*trans*]:
$[\![\vdash is_1,xt_1$ [::] $\tau s_1 @ \tau s_2$; $\vdash is_2,xt_2$ [::] τs_3;
size τs_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 [::] $\tau s_1 @ \tau s_2$
⟨*proof*⟩

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* τs_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*⟩

corollary *wt-instrs-app-last*[*trans*]:
$[\![\vdash is_2,xt_2$ [::] $\tau \# \tau s_2$; $\vdash is_1,xt_1$ [::] τs_1;
last $\tau s_1 = \tau$; *size* τs_1 = *size* is_1+*1* $]\!]$
$\Longrightarrow \vdash is_1 @ is_2$, xt_1@*shift* (*size* is_1) xt_2 [::] $\tau s_1 @ \tau s_2$
⟨*proof*⟩

corollary *wt-instrs-append-last*[*trans*]:
$[\![\vdash is,xt$ [::] τs; P,T_r,*mxs*,*mpc*,[] $\vdash i,pc :: \tau s$;
pc = *size is*; *mpc* = *size* τs; *size is* + *1* < *size* $\tau s]\!]$
$\Longrightarrow \vdash is@[i],xt$ [::] τs
⟨*proof*⟩

corollary *wt-instrs-app2*:
$[\![\vdash (is_2$ :: *'b instr list*)$,xt_2$ [::] $\tau' \# \tau s_2$; $\vdash is_1,xt_1$ [::] $\tau \# \tau s_1 @ [\tau']$;
$xt' = xt_1$ @ *shift* (*size* is_1) xt_2; *size* τs_1+*1* = *size* $is_1]\!]$
$\Longrightarrow \vdash is_1 @ is_2,xt'$ [::] $\tau \# \tau s_1 @ \tau' \# \tau s_2$
⟨*proof*⟩

corollary *wt-instrs-app2-simp*[*trans*,*simp*]:
$[\![\vdash (is_2$ :: *'b instr list*)$,xt_2$ [::] $\tau' \# \tau s_2$; $\vdash is_1,xt_1$ [::] $\tau \# \tau s_1 @ [\tau']$; *size* τs_1+*1* = *size* $is_1]\!]$
$\Longrightarrow \vdash is_1 @ is_2$, xt_1@*shift* (*size* is_1) xt_2 [::] $\tau \# \tau s_1 @ \tau' \# \tau s_2$
⟨*proof*⟩

corollary *wt-instrs-Cons*[*simp*]:
⟦ $\tau s \neq$ []; ⊢ [*i*],[] [::] [τ,τ']; ⊢ *is*,*xt* [::] $\tau'\#\tau s$ ⟧
⟹ ⊢ *i*#*is*,*shift 1 xt* [::] $\tau\#\tau'\#\tau s$
⟨*proof*⟩

corollary *wt-instrs-Cons2*[*trans*]:
assumes τs: ⊢ *is*,*xt* [::] τs
assumes *i*: P,T_r,*mxs*,*mpc*,[] ⊢ *i*,*0* :: $\tau\#\tau s$
assumes *mpc*: *mpc* = *size* τs + *1*
shows ⊢ *i*#*is*,*shift 1 xt* [::] $\tau\#\tau s$
⟨*proof*⟩

lemma *wt-instrs-last-incr*[*trans*]:
⟦ ⊢ *is*,*xt* [::] τs@[τ]; $P \vdash \tau \leq' \tau'$ ⟧ ⟹ ⊢ *is*,*xt* [::] τs@[τ']
⟨*proof*⟩

end

lemma [*iff*]: *xcpt-app i P pc mxs* [] τ
⟨*proof*⟩

lemma [*simp*]: *xcpt-eff i P pc* τ [] = []
⟨*proof*⟩

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*]
⟨*proof*⟩

lemma *wt-Cast*:
is-type P T ⟹
⊢ [*Checkcast T*],[] [::] [$ty_i{}'$ (*U* # *ST*) *E A*, $ty_i{}'$ (*T* # *ST*) *E A*]
⟨*proof*⟩

lemma *wt-Instanceof*:
⟦ *is-type P T*; *is-refT U* ⟧ ⟹
⊢ [*Instanceof T*],[] [::] [$ty_i{}'$ (*U* # *ST*) *E A*, $ty_i{}'$ (*Boolean* # *ST*) *E A*]
⟨*proof*⟩

lemma *wt-Push*:
⟦ *size ST* < *mxs*; *typeof v* = *Some T* ⟧
⟹ ⊢ [*Push v*],[] [::] [$ty_i{}'$ *ST E A*, $ty_i{}'$ (*T*#*ST*) *E A*]
⟨*proof*⟩

lemma *wt-Pop*:
⊢ [*Pop*],[] [::] ($ty_i{}'$ (*T*#*ST*) *E A* # $ty_i{}'$ *ST E A* # τs)
⟨*proof*⟩

lemma *wt-BinOpInstr*:
$P \vdash T1 «bop» T2 :: T \Longrightarrow \vdash [BinOpInstr\ bop],[]\ [::]\ [ty_i{}'\ (T2\ \#\ T1\ \#\ ST)\ E\ A,\ ty_i{}'\ (T\ \#\ ST)\ E\ A]$
⟨proof⟩

lemma *wt-Load*:
$[\![\ size\ ST < mxs;\ size\ E \leq mxl;\ i \in\in A;\ i < size\ E\ ]\!]$
$\Longrightarrow \vdash [Load\ i],[]\ [::]\ [ty_i{}'\ ST\ E\ A,\ ty_i{}'\ (E!i\ \#\ ST)\ E\ A]$
⟨proof⟩

lemma *wt-Store*:
$[\![\ P \vdash T \leq E!i;\ i < size\ E;\ size\ E \leq mxl\ ]\!] \Longrightarrow$
$\vdash [Store\ i],[]\ [::]\ [ty_i{}'\ (T\#ST)\ E\ A,\ ty_i{}'\ ST\ E\ (\lfloor\{i\}\rfloor \sqcup A)]$
⟨proof⟩

lemma *wt-Get*:
$[\![\ P \vdash C\ sees\ F{:}T\ (fm)\ in\ D;\ class\text{-}type\text{-}of'\ U = \lfloor C\rfloor\ ]\!] \Longrightarrow$
$\vdash [Getfield\ F\ D],[]\ [::]\ [ty_i{}'\ (U\ \#\ ST)\ E\ A,\ ty_i{}'\ (T\ \#\ ST)\ E\ A]$
⟨proof⟩

lemma *wt-Put*:
$[\![\ P \vdash C\ sees\ F{:}T\ (fm)\ in\ D;\ class\text{-}type\text{-}of'\ U = \lfloor C\rfloor;\ P \vdash T' \leq T\ ]\!] \Longrightarrow$
$\vdash [Putfield\ F\ D],[]\ [::]\ [ty_i{}'\ (T'\ \#\ U\ \#\ ST)\ E\ A,\ ty_i{}'\ ST\ E\ A]$
⟨proof⟩

lemma *wt-CAS*:
$[\![\ P \vdash C\ sees\ F{:}T\ (fm)\ in\ D;\ class\text{-}type\text{-}of'\ U' = \lfloor C\rfloor;\ volatile\ fm;\ P \vdash T2 \leq T;\ P \vdash T3 \leq T\ ]\!] \Longrightarrow$
$\vdash [CAS\ F\ D],[]\ [::]\ [ty_i{}'\ (T3\ \#\ T2\ \#\ U'\ \#\ ST)\ E\ A,\ ty_i{}'\ (Boolean\ \#\ ST)\ E\ A]$
⟨proof⟩

lemma *wt-Throw*:
$P \vdash C \preceq^* Throwable \Longrightarrow \vdash [ThrowExc],[]\ [::]\ [ty_i{}'\ (Class\ C\ \#\ ST)\ E\ A,\ \tau']$
⟨proof⟩

lemma *wt-IfFalse*:
$[\![\ 2 \leq i;\ nat\ i < size\ \tau s + 2;\ P \vdash ty_i{}'\ ST\ E\ A \leq'\ \tau s\ !\ nat(i - 2)\ ]\!]$
$\Longrightarrow \vdash [IfFalse\ i],[]\ [::]\ ty_i{}'\ (Boolean\ \#\ ST)\ E\ A\ \#\ ty_i{}'\ ST\ E\ A\ \#\ \tau s$
⟨proof⟩

lemma *wt-Goto*:
$[\![\ 0 \leq int\ pc + i;\ nat\ (int\ pc + i) < size\ \tau s;\ size\ \tau s \leq mpc;$
$P \vdash \tau s!pc \leq'\ \tau s\ !\ nat\ (int\ pc + i)\ ]\!]$
$\Longrightarrow P,T,mxs,mpc,[] \vdash Goto\ i,pc :: \tau s$
⟨proof⟩

end

context *TC3* **begin**

lemma *wt-Invoke*:

$\llbracket$ *size es* = *size Ts'*; *class-type-of' U* = $\lfloor C \rfloor$; $P \vdash C$ *sees M*: *Ts*→*T* = *m in D*; $P \vdash$ *Ts'* [≤] *Ts* $\rrbracket$
$\Longrightarrow \vdash$ [*Invoke M* (*size es*)],[] [::] [*ty*$_i$' (*rev Ts'* @ *U* # *ST*) *E A*, *ty*$_i$' (*T*#*ST*) *E A*]
⟨*proof*⟩

end

declare *nth-append*[*simp del*]
declare [[*simproc del*: *list-to-set-comprehension*]]

context *TC2* **begin**

corollary *wt-instrs-app3*[*simp*]:
$\llbracket \vdash$ (*is*$_2$ :: *'b instr list*),[] [::] (τ' # τs_2); $\vdash$ *is*$_1$,*xt*$_1$ [::] τ # τs_1 @ [τ']; *size* τs_1+*1* = *size is*$_1$$\rrbracket$
$\Longrightarrow \vdash$ (*is*$_1$ @ *is*$_2$),*xt*$_1$ [::] τ # τs_1 @ τ' # τs_2
⟨*proof*⟩

corollary *wt-instrs-Cons3*[*simp*]:
$\llbracket$ $\tau s \neq$ []; $\vdash$ [*i*],[] [::] [τ,τ']; $\vdash$ *is*,[] [::] τ'#τs $\rrbracket$
$\Longrightarrow \vdash$ (*i* # *is*),[] [::] τ # τ' # τs
⟨*proof*⟩

lemma *wt-instrs-xapp*:
$\llbracket \vdash$ *is*$_1$ @ *is*$_2$, *xt* [::] τs_1 @ *ty*$_i$' (*Class D* # *ST*) *E A* # τs_2;
$\forall \tau \in$ *set* τs_1. $\forall$ *ST' LT'*. τ = *Some*(*ST'*,*LT'*) ⟶
size ST ≤ *size ST'* ∧ $P \vdash$ *Some* (*drop* (*size ST'* − *size ST*) *ST'*,*LT'*) ≤' *ty*$_i$' *ST E A*;
size is$_1$ = *size* τs_1; *size ST* < *mxs*; *case Co of None* ⇒ *D* = *Throwable* | *Some C* ⇒ *D* = *C* ∧ *is-class P C* $\rrbracket \Longrightarrow$
$\vdash$ *is*$_1$ @ *is*$_2$, *xt* @ [(*0*,*size is*$_1$ − *Suc n*,*Co*,*size is*$_1$,*size ST*)] [::] τs_1 @ *ty*$_i$' (*Class D* # *ST*) *E A* # τs_2
⟨*proof*⟩

lemma *wt-instrs-xapp-Some*[*trans*]:
$\llbracket \vdash$ *is*$_1$ @ *is*$_2$, *xt* [::] τs_1 @ *ty*$_i$' (*Class C* # *ST*) *E A* # τs_2;
$\forall \tau \in$ *set* τs_1. $\forall$ *ST' LT'*. τ = *Some*(*ST'*,*LT'*) ⟶
size ST ≤ *size ST'* ∧ $P \vdash$ *Some* (*drop* (*size ST'* − *size ST*) *ST'*,*LT'*) ≤' *ty*$_i$' *ST E A*;
size is$_1$ = *size* τs_1; *is-class P C*; *size ST* < *mxs* $\rrbracket \Longrightarrow$
$\vdash$ *is*$_1$ @ *is*$_2$, *xt* @ [(*0*,*size is*$_1$ − *Suc n*,*Some C*,*size is*$_1$,*size ST*)] [::] τs_1 @ *ty*$_i$' (*Class C* # *ST*) *E A* # τs_2
⟨*proof*⟩

lemma *wt-instrs-xapp-Any*:
$\llbracket \vdash$ *is*$_1$ @ *is*$_2$, *xt* [::] τs_1 @ *ty*$_i$' (*Class Throwable* # *ST*) *E A* # τs_2;
$\forall \tau \in$ *set* τs_1. $\forall$ *ST' LT'*. τ = *Some*(*ST'*,*LT'*) ⟶
size ST ≤ *size ST'* ∧ $P \vdash$ *Some* (*drop* (*size ST'* − *size ST*) *ST'*,*LT'*) ≤' *ty*$_i$' *ST E A*;
size is$_1$ = *size* τs_1; *size ST* < *mxs* $\rrbracket \Longrightarrow$
$\vdash$ *is*$_1$ @ *is*$_2$, *xt* @ [(*0*,*size is*$_1$ − *Suc n*,*None*,*size is*$_1$,*size ST*)] [::] τs_1 @ *ty*$_i$' (*Class Throwable* # *ST*) *E A* # τs_2
⟨*proof*⟩

end

declare [[*simproc add*: *list-to-set-comprehension*]]
declare *nth-append*[*simp*]

lemma *drop-Cons-Suc*:
$\bigwedge xs.\ drop\ n\ xs = y\#ys \Longrightarrow drop\ (Suc\ n)\ xs = ys$
⟨*proof*⟩

lemma *drop-mess*:
$[\![Suc\ (length\ xs_0) \leq length\ xs;\ drop\ (length\ xs - Suc\ (length\ xs_0))\ xs = x\ \#\ xs_0]\!]$
$\Longrightarrow drop\ (length\ xs - length\ xs_0)\ xs = xs_0$
⟨*proof*⟩

lemma *drop-mess2*:
assumes *len*: $Suc\ (Suc\ (length\ xs0)) \leq length\ xs$
and *drop*: $drop\ (length\ xs - Suc\ (Suc\ (length\ xs0)))\ xs = x1\ \#\ x2\ \#\ xs0$
shows $drop\ (length\ xs - length\ xs0)\ xs = xs0$
⟨*proof*⟩

abbreviation *postfix* :: $'a\ list \Rightarrow\ 'a\ list \Rightarrow bool$ (‹(-/ ⋙= -)› [51, 50] 50) **where**
$postfix\ xs\ ys \equiv suffix\ ys\ xs$

lemma *postfix-conv-eq-length-drop*:
$ST' \ggg= ST \longleftrightarrow length\ ST \leq length\ ST' \wedge drop\ (length\ ST' - length\ ST)\ ST' = ST$
⟨*proof*⟩

declare *suffix-ConsI*[*simp*]

context *TC0* **begin**

declare *after-def*[*simp*] $pair\text{-}eq\text{-}ty_i{}'\text{-}conv$[*simp*]

lemma
assumes $ST0 \ggg= ST'$
shows *compT-ST-prefix*:
$\lfloor(ST,LT)\rfloor \in set(compT\ E\ A\ ST0\ e) \Longrightarrow ST \ggg= ST'$

and *compTs-ST-prefix*:
$\lfloor(ST,LT)\rfloor \in set(compTs\ E\ A\ ST0\ es) \Longrightarrow ST \ggg= ST'$
⟨*proof*⟩

declare *after-def*[*simp del*] $pair\text{-}eq\text{-}ty_i{}'\text{-}conv$[*simp del*]

end
declare *suffix-ConsI*[*simp del*]

lemma *fun-of-simp* [*simp*]: $fun\text{-}of\ S\ x\ y = ((x,y) \in S)$
⟨*proof*⟩

declare *widens-refl* [*iff*]

context *TC3* **begin**

theorem *compT-wt-instrs*:
$[\![\ P,E \vdash 1\ e :: T;\ \mathcal{D}\ e\ A;\ \mathcal{B}\ e\ (size\ E);\ size\ ST + max\text{-}stack\ e \leq mxs;\ size\ E + max\text{-}vars\ e \leq mxl;$
$set\ E \subseteq types\ P\ ]\!]$

$\Longrightarrow \vdash$ *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*; $\mathcal{D}$*s es A*; $\mathcal{B}$*s es* (*size E*); *size ST* + *max-stacks es* ≤ *mxs*; *size E* + *max-varss es* ≤ *mxl*; *set E* ⊆ *types P* ⟧
$\Longrightarrow$ *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* ⊔ $\mathcal{A}$*s es*)
(**is** *PROP ?Ps es E Ts A ST*)
⟨*proof*⟩

end

lemma *states-compP* [*simp*]: *states* (*compP f P*) *mxs mxl* = *states P mxs mxl*
⟨*proof*⟩

lemma [*simp*]: app_i (*i*, *compP f P*, *pc*, *mpc*, *T*, τ) = app_i (*i*, *P*, *pc*, *mpc*, *T*, τ)
⟨*proof*⟩

lemma [*simp*]: *is-relevant-entry* (*compP f P*) *i* = *is-relevant-entry P i*
⟨*proof*⟩

lemma [*simp*]: *relevant-entries* (*compP f P*) *i pc xt* = *relevant-entries P i pc xt*
⟨*proof*⟩

lemma [*simp*]: *app i* (*compP f P*) *mpc T pc mxl xt* τ = *app i P mpc T pc mxl xt* τ
⟨*proof*⟩

lemma [*simp*]: *app i P mpc T pc mxl xt* τ $\Longrightarrow$ *eff i* (*compP f P*) *pc xt* τ = *eff i P pc xt* τ
⟨*proof*⟩

lemma [*simp*]: *widen* (*compP f P*) = *widen P*
⟨*proof*⟩

lemma [*simp*]: *compP f P* ⊢ τ ≤' τ' = *P* ⊢ τ ≤' τ'
⟨*proof*⟩

lemma [*simp*]: *compP f P*,*T*,*mpc*,*mxl*,*xt* ⊢ *i*,*pc* :: τ*s* = *P*,*T*,*mpc*,*mxl*,*xt* ⊢ *i*,*pc* :: τ*s*
⟨*proof*⟩

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'* ⟧ $\Longrightarrow$
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*)
⟨*proof*⟩

definition *compTP* :: *′addr J1-prog* ⇒ *ty*$_P$
where
compTP P C M ≡
let (*D*,*Ts*,*T*,*meth*) = *method P C M*;
e = *the meth*;
E = *Class C* # *Ts*;
A = ⌊{..*size Ts*}⌋;
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 ⟹ *wf-jvm-prog*$_{compTP\ P}$ (*compP2 P*)
⟨*proof*⟩

theorem *wt-compP2*:
wf-J1-prog P ⟹ *wf-jvm-prog* (*compP2 P*)
⟨*proof*⟩

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 *τinstr* :: *′m prog* ⇒ *′heap* ⇒ *′addr val list* ⇒ *′addr instr* ⇒ *bool*
where
τinstr P h stk (*Load n*) = *True*
| *τinstr P h stk* (*Store n*) = *True*
| *τinstr P h stk* (*Push v*) = *True*
| *τinstr P h stk* (*New C*) = *False*
| *τinstr P h stk* (*NewArray T*) = *False*
| *τinstr 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
$\tau move2xcp$: $pc < length\ (compE2\ e) \Longrightarrow \tau move2\ P\ h\ stk\ e\ pc\ \lfloor xcp \rfloor$

| $\tau move2NewArray$: $\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (newA\ T\lfloor e \rceil)\ pc\ xcp$

| $\tau move2Cast$: $\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (Cast\ T\ e)\ pc\ xcp$
| $\tau move2CastRed$: $\tau move2\ P\ h\ stk\ (Cast\ T\ e)\ (length\ (compE2\ e))\ None$

| $\tau move2InstanceOf$: $\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (e\ instanceof\ T)\ pc\ xcp$
| $\tau move2InstanceOfRed$: $\tau move2\ P\ h\ stk\ (e\ instanceof\ T)\ (length\ (compE2\ e))\ None$

| $\tau move2Val$: $\tau move2\ P\ h\ stk\ (Val\ v)\ 0\ None$

| $\tau move2BinOp1$:
$\tau move2\ P\ h\ stk\ e1\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (e1 «bop» e2)\ pc\ xcp$
| $\tau move2BinOp2$:
$\tau move2\ P\ h\ stk\ e2\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (e1 «bop» e2)\ (length\ (compE2\ e1) + pc)\ xcp$
| $\tau move2BinOp$:
$\tau move2\ P\ h\ stk\ (e1 «bop» e2)\ (length\ (compE2\ e1) + length\ (compE2\ e2))\ None$

| $\tau move2Var$:
$\tau move2\ P\ h\ stk\ (Var\ V)\ 0\ None$

| $\tau move2LAss$:
$\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow \tau move2\ P\ h\ stk\ (V{:=}e)\ pc\ xcp$
| $\tau move2LAssRed1$:
$\tau move2\ P\ h\ stk\ (V{:=}e)\ (length\ (compE2\ e))\ None$
| $\tau move2LAssRed2$: $\tau move2\ P\ h\ stk\ (V{:=}e)\ (Suc\ (length\ (compE2\ e)))\ None$

| τ*move2AAcc1*: τ*move2 P h stk a pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*a*⌊*i*⌉) *pc xcp*
| τ*move2AAcc2*: τ*move2 P h stk i pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*a*⌊*i*⌉) (*length* (*compE2 a*) + *pc*) *xcp*

| τ*move2AAss1*: τ*move2 P h stk a pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*a*⌊*i*⌉ := *e*) *pc xcp*
| τ*move2AAss2*: τ*move2 P h stk i pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *pc*) *xcp*
| τ*move2AAss3*: τ*move2 P h stk e pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*a*⌊*i*⌉ := *e*) (*length* (*compE2 a*) + *length* (*compE2 i*) + *pc*) *xcp*
| τ*move2AAssRed*: τ*move2 P h stk* (*a*⌊*i*⌉ := *e*) (*Suc* (*length* (*compE2 a*) + *length* (*compE2 i*) + *length* (*compE2 e*))) *None*

| τ*move2ALength*: τ*move2 P h stk a pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*a*·*length*) *pc xcp*

| τ*move2FAcc*: τ*move2 P h stk e pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*e*·*F*{*D*}) *pc xcp*

| τ*move2FAss1*: τ*move2 P h stk e pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*e*·*F*{*D*} := *e'*) *pc xcp*
| τ*move2FAss2*: τ*move2 P h stk e' pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*e*·*F*{*D*} := *e'*) (*length* (*compE2 e*) + *pc*) *xcp*
| τ*move2FAssRed*: τ*move2 P h stk* (*e*·*F*{*D*} := *e'*) (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e'*))) *None*

| τ*move2CAS1*: τ*move2 P h stk e pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*e*·*compareAndSwap*(*D*·*F*, *e'*, *e''*)) *pc xcp*
| τ*move2CAS2*: τ*move2 P h stk e' pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*e*·*compareAndSwap*(*D*·*F*, *e'*, *e''*)) (*length* (*compE2 e*) + *pc*) *xcp*
| τ*move2CAS3*: τ*move2 P h stk e'' pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*e*·*compareAndSwap*(*D*·*F*, *e'*, *e''*)) (*length* (*compE2 e*) + *length* (*compE2 e'*) + *pc*) *xcp*

| τ*move2CallObj*:
τ*move2 P h stk obj pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*obj*·*M*(*ps*)) *pc xcp*
| τ*move2CallParams*:
τ*moves2 P h stk ps pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*obj*·*M*(*ps*)) (*length* (*compE2 obj*) + *pc*) *xcp*
| τ*move2Call*:
⟦ *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*)⟧
$\Longrightarrow$ τ*move2 P h stk* (*obj*·*M*(*ps*)) (*length* (*compE2 obj*) + *length* (*compEs2 ps*)) *None*

| τ*move2BlockSome1*:
τ*move2 P h stk* {*V*:*T*=⌊*v*⌋; *e*} *0 None*
| τ*move2BlockSome2*:
τ*move2 P h stk* {*V*:*T*=⌊*v*⌋; *e*} (*Suc 0*) *None*
| τ*move2BlockSome*:
τ*move2 P h stk e pc xcp* $\Longrightarrow$ τ*move2 P h stk* {*V*:*T*=⌊*v*⌋; *e*} (*Suc* (*Suc pc*)) *xcp*
| τ*move2BlockNone*:
τ*move2 P h stk e pc xcp* $\Longrightarrow$ τ*move2 P h stk* {*V*:*T*=*None*; *e*} *pc xcp*

| τ*move2Sync1*:
τ*move2 P h stk o' pc xcp* $\Longrightarrow$ τ*move2 P h stk* ($sync_V$ (*o'*) *e*) *pc xcp*
| τ*move2Sync2*:
τ*move2 P h stk* ($sync_V$ (*o'*) *e*) (*length* (*compE2 o'*)) *None*
| τ*move2Sync3*:

τ*move2 P h stk* (*sync*$_V$ (*o'*) *e*) (*Suc* (*length* (*compE2 o'*))) *None*
| τ*move2Sync4*:
τ*move2 P h stk e pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*sync*$_V$ (*o'*) *e*) (*Suc* (*Suc* (*Suc* (*length* (*compE2 o'*) + *pc*)))) *xcp*
| τ*move2Sync5*:
τ*move2 P h stk* (*sync*$_V$ (*o'*) *e*) (*Suc* (*Suc* (*Suc* (*length* (*compE2 o'*) + *length* (*compE2 e*))))) *None*
| τ*move2Sync6*:
τ*move2 P h stk* (*sync*$_V$ (*o'*) *e*) (*5* + *length* (*compE2 o'*) + *length* (*compE2 e*)) *None*
| τ*move2Sync7*:
τ*move2 P h stk* (*sync*$_V$ (*o'*) *e*) (*6* + *length* (*compE2 o'*) + *length* (*compE2 e*)) *None*
| τ*move2Sync8*:
τ*move2 P h stk* (*sync*$_V$ (*o'*) *e*) (*8* + *length* (*compE2 o'*) + *length* (*compE2 e*)) *None*

| τ*move2InSync*: τ*move2 P h stk* (*insync*$_V$ (*a*) *e*) *0 None*

| τ*move2Seq1*:
τ*move2 P h stk e pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*e*;;*e'*) *pc xcp*
| τ*move2SeqRed*:
τ*move2 P h stk* (*e*;;*e'*) (*length* (*compE2 e*)) *None*
| τ*move2Seq2*:
τ*move2 P h stk e' pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*e*;;*e'*) (*Suc* (*length* (*compE2 e*) + *pc*)) *xcp*

| τ*move2Cond*:
τ*move2 P h stk e pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*if* (*e*) *e1 else e2*) *pc xcp*
| τ*move2CondRed*:
τ*move2 P h stk* (*if* (*e*) *e1 else e2*) (*length* (*compE2 e*)) *None*
| τ*move2CondThen*:
τ*move2 P h stk e1 pc xcp*
$\Longrightarrow$ τ*move2 P h stk* (*if* (*e*) *e1 else e2*) (*Suc* (*length* (*compE2 e*) + *pc*)) *xcp*
| τ*move2CondThenExit*:
τ*move2 P h stk* (*if* (*e*) *e1 else e2*) (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*))) *None*
| τ*move2CondElse*:
τ*move2 P h stk e2 pc xcp*
$\Longrightarrow$ τ*move2 P h stk* (*if* (*e*) *e1 else e2*) (*Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + *pc*))) *xcp*

| τ*move2While1*:
τ*move2 P h stk c pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*while* (*c*) *e*) *pc xcp*
| τ*move2While2*:
τ*move2 P h stk e pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*while* (*c*) *e*) (*Suc* (*length* (*compE2 c*) + *pc*)) *xcp*
| τ*move2While3*: — Jump back to condition
τ*move2 P h stk* (*while* (*c*) *e*) (*Suc* (*Suc* (*length* (*compE2 c*) + *length* (*compE2 e*)))) *None*
| τ*move2While4*: — last instruction: Push Unit
τ*move2 P h stk* (*while* (*c*) *e*) (*Suc* (*Suc* (*Suc* (*length* (*compE2 c*) + *length* (*compE2 e*))))) *None*
| τ*move2While5*: — IfFalse instruction
τ*move2 P h stk* (*while* (*c*) *e*) (*length* (*compE2 c*)) *None*
| τ*move2While6*: — Pop instruction
τ*move2 P h stk* (*while* (*c*) *e*) (*Suc* (*length* (*compE2 c*) + *length* (*compE2 e*))) *None*

| τ*move2Throw1*:
τ*move2 P h stk e pc xcp* $\Longrightarrow$ τ*move2 P h stk* (*throw e*) *pc xcp*
| τ*move2Throw2*:
τ*move2 P h stk* (*throw e*) (*length* (*compE2 e*)) *None*

| *τmove2Try1*:
τmove2 P h stk e pc xcp ⟹ *τmove2 P h stk* (*try e catch*(*C V*) *e′*) *pc xcp*
| *τmove2TryJump*:
τmove2 P h stk (*try e catch*(*C V*) *e′*) (*length* (*compE2 e*)) *None*
| *τmove2TryCatch2*:
τmove2 P h stk (*try e catch*(*C V*) *e′*) (*Suc* (*length* (*compE2 e*))) *None*
| *τmove2Try2*:
τmove2 P h stk {*V*:*T*=*None*; *e′*} *pc xcp*
⟹ *τmove2 P h stk* (*try e catch*(*C V*) *e′*) (*Suc* (*Suc* (*length* (*compE2 e*) + *pc*))) *xcp*

| *τmoves2Hd*:
τmove2 P h stk e pc xcp ⟹ *τmoves2 P h stk* (*e* # *es*) *pc xcp*
| *τmoves2Tl*:
τmoves2 P h stk es pc xcp ⟹ *τmoves2 P h stk* (*e* # *es*) (*length* (*compE2 e*) + *pc*) *xcp*

inductive-cases *τmove2-cases*:
τmove2 P h stk (*new C*) *pc xcp*
τmove2 P h stk (*newA T*⌊*e*⌉) *pc xcp*
τmove2 P h stk (*Cast T e*) *pc xcp*
τmove2 P h stk (*e instanceof T*) *pc xcp*
τmove2 P h stk (*Val v*) *pc xcp*
τmove2 P h stk (*Var V*) *pc xcp*
τmove2 P h stk (*e1* «*bop*» *e2*) *pc xcp*
τmove2 P h stk (*V* := *e*) *pc xcp*
τmove2 P h stk (*e1*⌊*e2*⌉) *pc xcp*
τmove2 P h stk (*e1*⌊*e2*⌉ := *e3*) *pc xcp*
τmove2 P h stk (*e1*·*length*) *pc xcp*
τmove2 P h stk (*e1*·*F*{*D*}) *pc xcp*
τmove2 P h stk (*e1*·*F*{*D*} := *e3*) *pc xcp*
τmove2 P h stk (*e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*)) *pc xcp*
τmove2 P h stk (*e*·*M*(*ps*)) *pc xcp*
τmove2 P h stk {*V*:*T*=*vo*; *e*} *pc xcp*
τmove2 P h stk ($sync_V$ (*e1*) *e2*) *pc xcp*
τmove2 P h stk (*e1*;;*e2*) *pc xcp*
τmove2 P h stk (*if* (*e1*) *e2 else e3*) *pc xcp*
τmove2 P h stk (*while* (*e1*) *e2*) *pc xcp*
τmove2 P h stk (*try e1 catch*(*C V*) *e2*) *pc xcp*
τmove2 P h stk (*throw e*) *pc xcp*

lemma *τmoves2xcp*: *pc* < *length* (*compEs2 es*) ⟹ *τmoves2 P h stk es pc* ⌊*xcp*⌋
⟨*proof*⟩

lemma *τmove2-intros′*:
shows *τmove2CastRed′*: *pc* = *length* (*compE2 e*) ⟹ *τmove2 P h stk* (*Cast T e*) *pc None*
and *τmove2InstanceOfRed′*: *pc* = *length* (*compE2 e*) ⟹ *τmove2 P h stk* (*e instanceof T*) *pc None*
and *τmove2BinOp2′*: ⟦ *τmove2 P h stk e2 pc xcp*; *pc′* = *length* (*compE2 e1*) + *pc* ⟧ ⟹ *τmove2 P h stk* (*e1* «*bop*» *e2*) *pc′ xcp*
and *τmove2BinOp′*: *pc* = *length* (*compE2 e1*) + *length* (*compE2 e2*) ⟹ *τmove2 P h stk* (*e1* «*bop*» *e2*) *pc None*
and *τmove2LAssRed1′*: *pc* = *length* (*compE2 e*) ⟹ *τmove2 P h stk* (*V*:=*e*) *pc None*
and *τmove2LAssRed2′*: *pc* = *Suc* (*length* (*compE2 e*)) ⟹ *τmove2 P h stk* (*V*:=*e*) *pc None*
and *τmove2AAcc2′*: ⟦ *τmove2 P h stk i pc xcp*; *pc′* = *length* (*compE2 a*) + *pc* ⟧ ⟹ *τmove2 P h stk* (*a*⌊*i*⌉) *pc′ xcp*

and *τmove2AAss2'*: ⟦ *τmove2 P h stk i pc xcp*; *pc' = length (compE2 a) + pc* ⟧ ⟹ *τmove2 P h stk (a⌊i⌉ := e) pc' xcp*
and *τmove2AAss3'*: ⟦ *τmove2 P h stk e pc xcp*; *pc' = length (compE2 a) + length (compE2 i) + pc* ⟧ ⟹ *τmove2 P h stk (a⌊i⌉ := e) pc' xcp*
and *τmove2AAssRed'*: *pc = Suc (length (compE2 a) + length (compE2 i) + length (compE2 e))* ⟹ *τmove2 P h stk (a⌊i⌉ := e) pc None*
and *τmove2FAss2'*: ⟦ *τmove2 P h stk e' pc xcp*; *pc' = length (compE2 e) + pc* ⟧ ⟹ *τmove2 P h stk (e•F{D} := e') pc' xcp*
and *τmove2FAssRed'*: *pc = Suc (length (compE2 e) + length (compE2 e'))* ⟹ *τmove2 P h stk (e•F{D} := e') pc None*
and *τmove2CAS2'*: ⟦ *τmove2 P h stk e2 pc xcp*; *pc' = length (compE2 e1) + pc* ⟧ ⟹ *τmove2 P h stk (e1•compareAndSwap(D•F, e2, e3)) pc' xcp*
and *τmove2CAS3'*: ⟦ *τmove2 P h stk e3 pc xcp*; *pc' = length (compE2 e1) + length (compE2 e2) + pc* ⟧ ⟹ *τmove2 P h stk (e1•compareAndSwap(D•F, e2, e3)) pc' xcp*
and *τmove2CallParams'*: ⟦ *τmoves2 P h stk ps pc xcp*; *pc' = length (compE2 obj) + pc* ⟧ ⟹ *τmove2 P h stk (obj•M(ps)) pc' xcp*
and *τmove2Call'*: ⟦ *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* ⟶ *τexternal-defs D M*) ⟧
⟹ *τmove2 P h stk (obj•M(ps)) pc None*
and *τmove2BlockSome2*: *pc = Suc 0* ⟹ *τmove2 P h stk {V:T=⌊v⌋; e} pc None*
and *τmove2BlockSome'*: ⟦ *τmove2 P h stk e pc xcp*; *pc' = Suc (Suc pc)* ⟧ ⟹ *τmove2 P h stk {V:T=⌊v⌋; e} pc' xcp*
and *τmove2Sync2'*: *pc = length (compE2 o')* ⟹ *τmove2 P h stk (sync$_V$ (o') e) pc None*
and *τmove2Sync3'*: *pc = Suc (length (compE2 o'))* ⟹ *τmove2 P h stk (sync$_V$ (o') e) pc None*
and *τmove2Sync4'*: ⟦ *τmove2 P h stk e pc xcp*; *pc' = Suc (Suc (Suc (length (compE2 o') + pc)))* ⟧ ⟹ *τmove2 P h stk (sync$_V$ (o') e) pc' xcp*
and *τmove2Sync5'*: *pc = Suc (Suc (Suc (length (compE2 o') + length (compE2 e))))* ⟹ *τmove2 P h stk (sync$_V$ (o') e) pc None*
and *τmove2Sync6'*: *pc = 5 + length (compE2 o') + length (compE2 e)* ⟹ *τmove2 P h stk (sync$_V$ (o') e) pc None*
and *τmove2Sync7'*: *pc = 6 + length (compE2 o') + length (compE2 e)* ⟹ *τmove2 P h stk (sync$_V$ (o') e) pc None*
and *τmove2Sync8'*: *pc = 8 + length (compE2 o') + length (compE2 e)* ⟹ *τmove2 P h stk (sync$_V$ (o') e) pc None*
and *τmove2SeqRed'*: *pc = length (compE2 e)* ⟹ *τmove2 P h stk (e;;e') pc None*
and *τmove2Seq2'*: ⟦ *τmove2 P h stk e' pc xcp*; *pc' = Suc (length (compE2 e) + pc)* ⟧ ⟹ *τmove2 P h stk (e;;e') pc' xcp*
and *τmove2CondRed'*: *pc = length (compE2 e)* ⟹ *τmove2 P h stk (if (e) e1 else e2) pc None*
and *τmove2CondThen'*: ⟦ *τmove2 P h stk e1 pc xcp*; *pc' = Suc (length (compE2 e) + pc)* ⟧ ⟹ *τmove2 P h stk (if (e) e1 else e2) pc' xcp*
and *τmove2CondThenExit'*: *pc = Suc (length (compE2 e) + length (compE2 e1))* ⟹ *τmove2 P h stk (if (e) e1 else e2) pc None*
and *τmove2CondElse'*: ⟦ *τmove2 P h stk e2 pc xcp*; *pc' = Suc (Suc (length (compE2 e) + length (compE2 e1) + pc))* ⟧
⟹ *τmove2 P h stk (if (e) e1 else e2) pc' xcp*
and *τmove2While2'*: ⟦ *τmove2 P h stk e pc xcp*; *pc' = Suc (length (compE2 c) + pc)* ⟧ ⟹ *τmove2 P h stk (while (c) e) pc' xcp*
and *τmove2While3'*: *pc = Suc (Suc (length (compE2 c) + length (compE2 e)))* ⟹ *τmove2 P h stk (while (c) e) pc None*
and *τmove2While4'*: *pc = Suc (Suc (Suc (length (compE2 c) + length (compE2 e))))* ⟹ *τmove2 P h stk (while (c) e) pc None*
and *τmove2While5'*: *pc = length (compE2 c)* ⟹ *τmove2 P h stk (while (c) e) pc None*

and *τmove2While6'*: $pc = Suc\ (length\ (compE2\ c) + length\ (compE2\ e)) \Longrightarrow \tau move2\ P\ h\ stk\ (while\ (c)\ e)\ pc\ None$
and *τmove2Throw2'*: $pc = length\ (compE2\ e) \Longrightarrow \tau move2\ P\ h\ stk\ (throw\ e)\ pc\ None$
and *τmove2TryJump'*: $pc = length\ (compE2\ e) \Longrightarrow \tau move2\ P\ h\ stk\ (try\ e\ catch(C\ V)\ e')\ pc\ None$
and *τmove2TryCatch2'*: $pc = Suc\ (length\ (compE2\ e)) \Longrightarrow \tau move2\ P\ h\ stk\ (try\ e\ catch(C\ V)\ e')\ pc\ None$
and *τmove2Try2'*: $[\![\ \tau move2\ P\ h\ stk\ \{V{:}T{=}None;\ e'\}\ pc\ xcp;\ pc' = Suc\ (Suc\ (length\ (compE2\ e) + pc))\ ]\!]$
$\Longrightarrow \tau move2\ P\ h\ stk\ (try\ e\ catch(C\ V)\ e')\ pc'\ xcp$
and *τmoves2Tl'*: $[\![\ \tau moves2\ P\ h\ stk\ es\ pc\ xcp;\ pc' = length\ (compE2\ e) + pc\ ]\!] \Longrightarrow \tau moves2\ P\ h\ stk\ (e\ \#\ es)\ pc'\ xcp$
⟨*proof*⟩

lemma *τmove2-iff*: $\tau move2\ P\ h\ stk\ e\ pc\ xcp \longleftrightarrow pc < length\ (compE2\ e) \land (xcp = None \longrightarrow \tau instr\ P\ h\ stk\ (compE2\ e\ !\ pc))$ (**is** *?lhs1* ⟷ *?rhs1*)
and *τmoves2-iff*: $\tau moves2\ P\ h\ stk\ es\ pc\ xcp \longleftrightarrow pc < length\ (compEs2\ es) \land (xcp = None \longrightarrow \tau instr\ P\ h\ stk\ (compEs2\ es\ !\ pc))$ (**is** *?lhs2* ⟷ *?rhs2*)
⟨*proof*⟩

lemma *τmove2-pc-length-compE2*: $\tau move2\ P\ h\ stk\ e\ pc\ xcp \Longrightarrow pc < length\ (compE2\ e)$
and *τmoves2-pc-length-compEs2*: $\tau moves2\ P\ h\ stk\ es\ pc\ xcp \Longrightarrow pc < length\ (compEs2\ es)$
⟨*proof*⟩

lemma *τmove2-pc-length-compE2-conv*: $pc \geq length\ (compE2\ e) \Longrightarrow \neg\ \tau move2\ P\ h\ stk\ e\ pc\ xcp$
⟨*proof*⟩

lemma *τmoves2-pc-length-compEs2-conv*: $pc \geq length\ (compEs2\ es) \Longrightarrow \neg\ \tau moves2\ P\ h\ stk\ es\ pc\ xcp$
⟨*proof*⟩

lemma *τmoves2-append* [*elim*]:
$\tau moves2\ P\ h\ stk\ es\ pc\ xcp \Longrightarrow \tau moves2\ P\ h\ stk\ (es\ @\ es')\ pc\ xcp$
⟨*proof*⟩

lemma *append-τmoves2*:
$\tau moves2\ P\ h\ stk\ es\ pc\ xcp \Longrightarrow \tau moves2\ P\ h\ stk\ (es'\ @\ es)\ (length\ (compEs2\ es') + pc)\ xcp$
⟨*proof*⟩

lemma [*dest*]:
shows *τmove2-NewArrayD*: $[\![\ \tau move2\ P\ h\ stk\ (newA\ T\lfloor e\rceil)\ pc\ xcp;\ pc < length\ (compE2\ e)\ ]\!] \Longrightarrow \tau move2\ P\ h\ stk\ e\ pc\ xcp$
and *τmove2-CastD*: $[\![\ \tau move2\ P\ h\ stk\ (Cast\ T\ e)\ pc\ xcp;\ pc < length\ (compE2\ e)\ ]\!] \Longrightarrow \tau move2\ P\ h\ stk\ e\ pc\ xcp$
and *τmove2-InstanceOfD*: $[\![\ \tau move2\ P\ h\ stk\ (e\ instanceof\ T)\ pc\ xcp;\ pc < length\ (compE2\ e)\ ]\!] \Longrightarrow \tau move2\ P\ h\ stk\ e\ pc\ xcp$
and *τmove2-BinOp1D*: $[\![\ \tau move2\ P\ h\ stk\ (e1\ «bop»\ e2)\ pc'\ xcp';\ pc' < length\ (compE2\ e1)\ ]\!] \Longrightarrow \tau move2\ P\ h\ stk\ e1\ pc'\ xcp'$
and *τmove2-BinOp2D*:
$[\![\ \tau move2\ P\ h\ stk\ (e1\ «bop»\ e2)\ (length\ (compE2\ e1) + pc')\ xcp';\ pc' < length\ (compE2\ e2)\ ]\!] \Longrightarrow \tau move2\ P\ h\ stk\ e2\ pc'\ xcp'$
and *τmove2-LAssD*: $[\![\ \tau move2\ P\ h\ stk\ (V := e)\ pc\ xcp;\ pc < length\ (compE2\ e)\ ]\!] \Longrightarrow \tau move2\ P\ h\ stk\ e\ pc\ xcp$
and *τmove2-AAccD1*: $[\![\ \tau move2\ P\ h\ stk\ (a\lfloor i\rceil)\ pc\ xcp;\ pc < length\ (compE2\ a)\ ]\!] \Longrightarrow \tau move2\ P\ h\ stk\ a\ pc\ xcp$
and *τmove2-AAccD2*: $[\![\ \tau move2\ P\ h\ stk\ (a\lfloor i\rceil)\ (length\ (compE2\ a) + pc)\ xcp;\ pc < length\ (compE2$

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
⟨proof⟩

lemma τmove2-Invoke:

⟦τmove2 P h stk e pc None; compE2 e ! pc = Invoke M n ⟧
⟹ n < length stk ∧ (stk ! n = Null ∨ (∀ T Ts Tr D. typeof-addr h (the-Addr (stk ! n)) = ⌊T⌋ ⟶ P ⊢ class-type-of T sees M:Ts→Tr = Native in D ⟶ τexternal-defs D M))

and τmoves2-Invoke:
⟦τmoves2 P h stk es pc None; compEs2 es ! pc = Invoke M n ⟧
⟹ n < length stk ∧ (stk ! n = Null ∨ (∀ T C Ts Tr D. typeof-addr h (the-Addr (stk ! n)) = ⌊T⌋ ⟶ P ⊢ class-type-of T sees M:Ts→Tr = Native in D ⟶ τexternal-defs D M))
⟨proof⟩

lemmas τmove2-compE2-not-Invoke = τmove2-Invoke
lemmas τmoves2-compEs2-not-Invoke = τmoves2-Invoke

lemma τmove2-blocks1 [simp]:
τmove2 P h stk (blocks1 n Ts body) pc′ xcp′ = τmove2 P h stk body pc′ xcp′
⟨proof⟩

lemma τinstr-stk-append:
τinstr P h stk i ⟹ τinstr P h (stk @ vs) i
⟨proof⟩

lemma τmove2-stk-append:
τmove2 P h stk e pc xcp ⟹ τmove2 P h (stk @ vs) e pc xcp
⟨proof⟩

lemma τmoves2-stk-append:
τmoves2 P h stk es pc xcp ⟹ τmoves2 P h (stk @ vs) es pc xcp
⟨proof⟩

fun τMove2 :: ′addr jvm-prog ⇒ (′addr, ′heap) jvm-state ⇒ bool
where
τMove2 P (xcp, h, []) = False
| τMove2 P (xcp, h, (stk, loc, C, M, pc) # frs) =
(let (-,-,-,meth) = method P C M; (-,-,ins,xt) = the meth
in (pc < length ins ∧ (xcp = None ⟶ τinstr P h stk (ins ! pc))))

lemma τMove2-iff:
τMove2 P σ = (let (xcp, h, frs) = σ
in case frs of [] ⇒ False
| (stk, loc, C, M, pc) # frs′ ⇒
(let (-,-,-,meth) = method P C M; (-,-,ins,xt) = the meth
in (pc < length ins ∧ (xcp = None ⟶ τinstr P h stk (ins ! pc)))))
⟨proof⟩

lemma τinstr-compP [simp]: τinstr (compP f P) h stk i ⟷ τinstr P h stk i
⟨proof⟩

lemma [simp]: **fixes** e :: ′addr expr1 **and** es :: ′addr expr1 list
shows τmove2-compP: τmove2 (compP f P) h stk e = τmove2 P h stk e
and τmoves2-compP: τmoves2 (compP f P) h stk es = τmoves2 P h stk es
⟨proof⟩

lemma τMove2-compP2:
P ⊢ C sees M:Ts→T=⌊body⌋ in D ⟹

τMove2 (*compP2 P*) (*xcp*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs*) =
(*case xcp of None* ⇒ *τmove2 P h stk body pc xcp* ∨ *pc* = *length* (*compE2 body*) | *Some a* ⇒ *pc* < *Suc* (*length* (*compE2 body*)))
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *mexecd-τmthr-wf*:
τmultithreaded-wf JVM-final (*mexecd P*) (*τMOVE2 P*)
⟨*proof*⟩

end

sublocale *JVM-heap-base* < *execd-mthr*:
τmultithreaded-wf
JVM-final
mexecd P
convert-RA
τMOVE2 P
for *P*
⟨*proof*⟩

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
⟨*proof*⟩

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*:
$(\bigwedge$*ins P h stk stk′ loc C M pc frs. ci ins P h stk loc C M pc frs* $\Longrightarrow$ *ci ins P h* (*stk* @ *stk′*) *loc C M pc frs*) $\Longrightarrow$ *ci-stk-offer ci*
⟨*proof*⟩

lemma *ci-stk-offerD*:
⟦ *ci-stk-offer ci*; *ci ins P h stk loc C M pc frs* ⟧ $\Longrightarrow$ *ci ins P h* (*stk* @ *stk′*) *loc C M pc frs*
⟨*proof*⟩

lemma *check-instr′-stk-offer*:
ci-stk-offer check-instr′
⟨*proof*⟩

context *JVM-heap-base* **begin**

lemma *check-instr-imp-check-instr′*:
check-instr ins P h stk loc C M pc frs $\Longrightarrow$ *check-instr′ ins P h stk loc C M pc frs*
⟨*proof*⟩

lemma *check-instr-stk-offer*:
ci-stk-offer check-instr
⟨*proof*⟩

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′*
⟨*proof*⟩

lemma *jump-ok-GotoD*:
⟦ *jump-ok ins n n′*; *ins* ! *pc* = *Goto m*; *pc* < *length ins* ⟧ $\Longrightarrow$ − *int* (*pc* + *n*) ≤ *m* ∧ *m* < *int* (*length ins* − *pc* + *n′*)
⟨*proof*⟩

lemma *jump-ok-IfFalseD*:
⟦ *jump-ok ins n n′*; *ins* ! *pc* = *IfFalse m*; *pc* < *length ins* ⟧ $\Longrightarrow$ − *int* (*pc* + *n*) ≤ *m* ∧ *m* < *int* (*length ins* − *pc* + *n′*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *wf-ci-check-instr'* [*iff*]:
wf-ci check-instr'
⟨*proof*⟩

lemma *jump-ok-ins-jump-ok*:
⟦ *jump-ok ins n n'*; *pc* < *length ins* ⟧ ⟹ *ins-jump-ok* (*ins* ! *pc*) (*pc* + *n*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *wf-ci-check-instr* [*iff*]:
wf-ci check-instr
⟨*proof*⟩

end

lemma *wf-ciD1*: *wf-ci ci* $\Longrightarrow$ *ci-stk-offer ci*
⟨*proof*⟩

lemma *wf-ciD2*: $\llbracket$ *wf-ci ci*; *ci ins P h stk loc C M pc frs* $\rrbracket \Longrightarrow$ *check-instr′ ins P h stk loc C M pc frs*
⟨*proof*⟩

lemma *wf-ciD3*: $\llbracket$ *wf-ci ci*; *ci ins P h stk loc C M pc frs*; *ins-jump-ok ins pc′* $\rrbracket \Longrightarrow$ *ci ins P h stk loc C M pc′ frs*
⟨*proof*⟩

lemma *check-instr′-ins-jump-ok*: *check-instr′ ins P h stk loc C M pc frs* $\Longrightarrow$ *ins-jump-ok ins pc*
⟨*proof*⟩
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′*: $pc \leq pc'$
  shows *ins-jump-ok ins pc′*
⟨*proof*⟩

lemma *wf-ciD3′*: $\llbracket$ *wf-ci ci*; *ci ins P h stk loc C M pc frs*; $pc \leq pc'$ $\rrbracket \Longrightarrow$ *ci ins P h stk loc C M pc′ frs*
⟨*proof*⟩

typedef (*′addr*, *′heap*) *check-instr* = *Collect wf-ci* :: (*′addr*, *′heap*) *check-instr′ set*
  morphisms *ci-app Abs-check-instr*
⟨*proof*⟩

lemma *ci-app-check-instr′* [*simp*]: *ci-app* (*Abs-check-instr check-instr′*) = *check-instr′*
⟨*proof*⟩

lemma (**in** *JVM-heap-base*) *ci-app-check-instr* [*simp*]: *ci-app* (*Abs-check-instr check-instr*) = *check-instr*
⟨*proof*⟩

lemma *wf-ci-stk-offerD*:
  ci-app ci ins P h stk loc C M pc frs $\Longrightarrow$ *ci-app ci ins P h* (*stk* @ *stk′*) *loc C M pc frs*
⟨*proof*⟩

lemma *wf-ciD2-ci-app*:
  ci-app ci ins P h stk loc C M pc frs $\Longrightarrow$ *check-instr′ ins P h stk loc C M pc frs*
⟨*proof*⟩

lemma *wf-ciD3-ci-app*:
  $\llbracket$ *ci-app ci ins P h stk loc C M pc frs*; *ins-jump-ok ins pc′* $\rrbracket \Longrightarrow$ *ci-app ci ins P h stk loc C M pc′ frs*
⟨*proof*⟩

lemma *wf-ciD3′-ci-app*: $\llbracket$ *ci-app ci ins P h stk loc C M pc frs*; $pc \leq pc'$ $\rrbracket \Longrightarrow$ *ci-app ci ins P h stk loc C M pc′ frs*
⟨*proof*⟩

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* []
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*)
⟨*proof*⟩

lemma *exec-meth-drop*:
$\llbracket$ *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 $\rrbracket$
$\Longrightarrow$ *exec-meth ci P ins' xt t h* (*stk*, *loc*, *pc*, *xcp*) *ta h'* (*stk'*, *loc'*, (*pc'* − *length ins*), *xcp'*)
⟨*proof*⟩

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*
⟨*proof*⟩

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' $\Longrightarrow$ *exec-move ci P t* (*newA T*⌊*e*⌉) *h s ta h' s'*
⟨*proof*⟩

lemma *exec-move-newArray*:
pc < *length* (*compE2 e*) $\Longrightarrow$ *exec-move ci P t* (*newA T*⌊*e*⌉) *h* (*stk*, *loc*, *pc*, *xcp*) = *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*)
⟨*proof*⟩

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′*
⟨*proof*⟩

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*)
⟨*proof*⟩

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′*
⟨*proof*⟩

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*)
⟨*proof*⟩

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′*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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′*
⟨*proof*⟩

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*)
⟨*proof*⟩

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′*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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′*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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′*
⟨*proof*⟩

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)*
⟨*proof*⟩

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′*
⟨*proof*⟩

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)*
⟨*proof*⟩

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′*
⟨*proof*⟩

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)*
⟨*proof*⟩

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*⟩

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′*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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′*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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′*
⟨*proof*⟩

lemma *exec-move-BlockNone*:
exec-move ci P t {*V*:*T*=*None*; *e*} = *exec-move ci P t e*
⟨*proof*⟩

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*⟩

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*⟩

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'*
⟨*proof*⟩

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*⟩

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*⟩

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'*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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'*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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'*
⟨*proof*⟩

lemma (**in** *ab-group-add*) *uminus-minus-left-commute*:
− *a* − (*b* + *c*) = − *b* − (*a* + *c*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *exec-move-While2*:
assumes *pc*: $pc < length\ (compE2\ e')$
shows $exec\text{-}move\ ci\ P\ t\ (while\ (e)\ e')\ h\ (stk,\ loc,\ (Suc\ (length\ (compE2\ e) + pc)),\ xcp)\ ta$
$h'\ (stk',\ loc',\ (Suc\ (length\ (compE2\ e) + pc')),\ xcp') =$
$exec\text{-}move\ ci\ P\ t\ e'\ h\ (stk,\ loc,\ pc,\ xcp)\ ta\ h'\ (stk',\ loc',\ pc',\ xcp')$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *exec-move-ThrowI*:
$exec\text{-}move\ ci\ P\ t\ e\ h\ s\ ta\ h'\ s' \Longrightarrow exec\text{-}move\ ci\ P\ t\ (throw\ e)\ h\ s\ ta\ h'\ s'$
⟨*proof*⟩

lemma *exec-move-Throw*:
$pc < length\ (compE2\ e) \Longrightarrow exec\text{-}move\ ci\ P\ t\ (throw\ e)\ h\ (stk,\ loc,\ pc,\ xcp) = exec\text{-}move\ ci\ P\ t\ e\ h\ (stk,\ loc,\ pc,\ xcp)$
⟨*proof*⟩

lemma *exec-move-TryI1*:
$exec\text{-}move\ ci\ P\ t\ e\ h\ s\ ta\ h'\ s' \Longrightarrow exec\text{-}move\ ci\ P\ t\ (try\ e\ catch(C\ V)\ e')\ h\ s\ ta\ h'\ s'$
⟨*proof*⟩

lemma *exec-move-TryI2*:
assumes *exec*: $exec\text{-}move\ ci\ P\ t\ e\ h\ (stk,\ loc,\ pc,\ xcp)\ ta\ h'\ (stk',\ loc',\ pc',\ xcp')$
shows $exec\text{-}move\ ci\ P\ t\ (try\ e'\ catch(C\ V)\ e)\ h\ (stk,\ loc,\ Suc\ (Suc\ (length\ (compE2\ e') + pc)),\ xcp)\ ta\ h'\ (stk',\ loc',\ Suc\ (Suc\ (length\ (compE2\ e') + pc')),\ xcp')$
⟨*proof*⟩

lemma *exec-move-Try2*:
$exec\text{-}move\ ci\ P\ t\ (try\ e\ catch(C\ V)\ e')\ h\ (stk,\ loc,\ Suc\ (Suc\ (length\ (compE2\ e) + pc)),\ xcp)\ ta$
$h'\ (stk',\ loc',\ Suc\ (Suc\ (length\ (compE2\ e) + pc')),\ xcp') =$
$exec\text{-}move\ ci\ P\ t\ e'\ h\ (stk,\ loc,\ pc,\ xcp)\ ta\ h'\ (stk',\ loc',\ pc',\ xcp')$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *exec-move-raise-xcp-pcD*:
$exec\text{-}move\ ci\ P\ t\ E\ h\ (stk,\ loc,\ pc,\ None)\ ta\ h'\ (stk',\ loc',\ pc',\ Some\ a) \Longrightarrow pc' = pc$
⟨*proof*⟩

definition $\tau exec\text{-}meth$ ::
$('addr,\ 'heap)\ check\text{-}instr \Rightarrow 'addr\ jvm\text{-}prog \Rightarrow 'addr\ instr\ list \Rightarrow ex\text{-}table \Rightarrow 'thread\text{-}id \Rightarrow 'heap$
$\Rightarrow ('addr\ val\ list \times 'addr\ val\ list \times pc \times 'addr\ option)$
$\Rightarrow ('addr\ val\ list \times 'addr\ val\ list \times pc \times 'addr\ option) \Rightarrow bool$
where
$\tau exec\text{-}meth\ ci\ P\ ins\ xt\ t\ h\ s\ s' \longleftrightarrow$
$exec\text{-}meth\ ci\ P\ ins\ xt\ t\ h\ s\ \varepsilon\ h\ s' \wedge (snd\ (snd\ (snd\ s)) = None \longrightarrow \tau instr\ P\ h\ (fst\ s)\ (ins\ !\ fst\ (snd\ (snd\ s))))$

abbreviation $\tau exec\text{-}meth\text{-}a$
where $\tau exec\text{-}meth\text{-}a \equiv \tau exec\text{-}meth\ (Abs\text{-}check\text{-}instr\ check\text{-}instr')$

abbreviation $\tau exec\text{-}meth\text{-}d$
where $\tau exec\text{-}meth\text{-}d \equiv \tau exec\text{-}meth\ (Abs\text{-}check\text{-}instr\ check\text{-}instr)$

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'*
⟨*proof*⟩

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*)
⟨*proof*⟩

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* ⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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* $\Longrightarrow$ τ*instr P h stk'* (*ins* ! *pc'*)
⟨*proof*⟩

lemma τ*Exec-methr-induct* [*consumes 1*, *case-names refl step*]:
  ⟦ τ*Exec-methr ci P ins xt t h s s'*;
    Q s;
    ⋀*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* $\Longrightarrow$ τ*instr P h stk* (*ins* ! *pc*); *Q* (*stk*, *loc*, *pc*, *xcp*) ⟧ $\Longrightarrow$ *Q s'* ⟧
  $\Longrightarrow$ *Q s'*
⟨*proof*⟩

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''* ⟧ $\Longrightarrow$ τ*Exec-methr ci P ins xt t h s s''*
⟨*proof*⟩

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* $\Longrightarrow$ τ*instr P h stk* (*ins* ! *pc*)
      τ*Exec-methr ci P ins xt t h s'' s'*
⟨*proof*⟩

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 =
$(\lambda(stk, loc, pc, xcp)\ s'.\ exec\text{-}moves\ ci\ P\ t\ es\ h\ (stk, loc, pc, xcp)\ \varepsilon\ h\ s' \wedge \tau moves2\ P\ h\ stk\ es\ pc\ xcp)$

lemma *τexec-moveI*:
$[\![$ *exec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*) ε *h s'*; *τmove2 P h stk e pc xcp* $]\!]$
$\Longrightarrow$ *τexec-move ci P t e h* (*stk*, *loc*, *pc*, *xcp*) *s'*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *τexec-movesI*:
$[\![$ *exec-moves ci P t es h* (*stk*, *loc*, *pc*, *xcp*) ε *h s'*; *τmoves2 P h stk es pc xcp* $]\!]$
$\Longrightarrow$ *τexec-moves ci P t es h* (*stk*, *loc*, *pc*, *xcp*) *s'*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *τexec-move-conv-τexec-meth*:
τexec-move ci P t e = τexec-meth ci (*compP2 P*) (*compE2 e*) (*compxE2 e 0 0*) *t*
⟨*proof*⟩

lemma *τexec-moves-conv-τexec-meth*:
τexec-moves ci P t es = τexec-meth ci (*compP2 P*) (*compEs2 es*) (*compxEs2 es 0 0*) *t*
⟨*proof*⟩

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 *τExec-movesr*
where *τExec-movesr ci P t e h ==* (*τexec-moves ci P t e h*)^**

abbreviation *τExec-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*
⟨*proof*⟩

lemma τ*Execsr-refl*: τ*Exec-movesr ci P t e h s s*
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

lemmas τ*Execr1step* = τ*Execr-step*[*OF* τ*Execr-refl*]
lemmas τ*Execr2step* = τ*Execr-step*[*OF* τ*Execr-step*, *OF* τ*Execr-refl*]
lemmas τ*Execr3step* = τ*Execr-step*[*OF* τ*Execr-step*, *OF* τ*Execr-step*, *OF* τ*Execr-refl*]

lemmas τ*Execsr1step* = τ*Execsr-step*[*OF* τ*Execsr-refl*]
lemmas τ*Execsr2step* = τ*Execsr-step*[*OF* τ*Execsr-step*, *OF* τ*Execsr-refl*]

lemmas τ*Execsr3step* = τ*Execsr-step*[*OF* τ*Execsr-step*, *OF* τ*Execsr-step*, *OF* τ*Execsr-refl*]

lemma τ*Exect1step*:
⟦ *exec-move ci P t e h s ε h s′*;
τ*move2 P h* (*fst s*) *e* (*fst* (*snd* (*snd s*))) (*snd* (*snd* (*snd s*))) ⟧
⟹ τ*Exec-movet ci P t e h s s′*
⟨*proof*⟩

lemmas τ*Exect2step* = τ*Exect-step*[*OF* τ*Exect1step*]
lemmas τ*Exect3step* = τ*Exect-step*[*OF* τ*Exect-step*, *OF* τ*Exect1step*]

lemma τ*Execst1step*:
⟦ *exec-moves ci P t es h s ε h s′*;
τ*moves2 P h* (*fst s*) *es* (*fst* (*snd* (*snd s*))) (*snd* (*snd* (*snd s*))) ⟧
⟹ τ*Exec-movest ci P t es h s s′*
⟨*proof*⟩

lemmas τ*Execst2step* = τ*Execst-step*[*OF* τ*Execst1step*]
lemmas τ*Execst3step* = τ*Execst-step*[*OF* τ*Execst-step*, *OF* τ*Execst1step*]

lemma τ*Execr-induct* [*consumes 1*, *case-names refl step*]:
assumes *major*: τ*Exec-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″*.
⟦ τ*Exec-mover ci P t e h* (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*);
τ*exec-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″*
⟨*proof*⟩

lemma τ*Execsr-induct* [*consumes 1*, *case-names refl step*]:
assumes *major*: τ*Exec-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″*.
⟦ τ*Exec-movesr 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′* ⟧
⟹ *Q stk″ loc″ pc″ xcp″*
shows *Q stk″ loc″ pc″ xcp″*
⟨*proof*⟩

lemma τ*Exect-induct* [*consumes 1*, *case-names base step*]:
assumes *major*: τ*Exec-movet ci P t e h* (*stk*, *loc*, *pc*, *xcp*) (*stk″*, *loc″*, *pc″*, *xcp″*)
and *base*: ⋀*stk′ loc′ pc′ xcp′*. τ*exec-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″*.
⟦ τ*Exec-movet ci P t e h* (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*);
τ*exec-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″*
⟨*proof*⟩

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*: ⋀*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′
and *step*: $\bigwedge$*stk′ loc′ pc′ xcp′ stk′′ loc′′ pc′′ xcp′′*.
⟦ *τExec-movest 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′′*
⟨*proof*⟩

lemma *τExec-mover-τExec-methr*:
τExec-mover ci P t e = τExec-methr ci (compP2 P) (compE2 e) (compxE2 e 0 0) t
⟨*proof*⟩

lemma *τExec-movesr-τExec-methr*:
τExec-movesr ci P t es = τExec-methr ci (compP2 P) (compEs2 es) (compxEs2 es 0 0) t
⟨*proof*⟩

lemma *τExec-movet-τExec-metht*:
τExec-movet ci P t e = τExec-metht ci (compP2 P) (compE2 e) (compxE2 e 0 0) t
⟨*proof*⟩

lemma *τExec-movest-τExec-metht*:
τExec-movest ci P t es = τExec-metht ci (compP2 P) (compEs2 es) (compxEs2 es 0 0) t
⟨*proof*⟩

lemma *τExec-mover-trans*:
⟦ *τExec-mover ci P t e h s s′*; *τExec-mover ci P t e h s′ s′′* ⟧ ⟹ *τExec-mover ci P t e h s s′′*
⟨*proof*⟩

lemma *τExec-movesr-trans*:
⟦ *τExec-movesr ci P t es h s s′*; *τExec-movesr ci P t es h s′ s′′* ⟧ ⟹ *τExec-movesr ci P t es h s s′′*
⟨*proof*⟩

lemma *τExec-movet-trans*:
⟦ *τExec-movet ci P t e h s s′*; *τExec-movet ci P t e h s′ s′′* ⟧ ⟹ *τExec-movet ci P t e h s s′′*
⟨*proof*⟩

lemma *τExec-movest-trans*:
⟦ *τExec-movest ci P t es h s s′*; *τExec-movest ci P t es h s′ s′′* ⟧ ⟹ *τExec-movest ci P t es h s s′′*
⟨*proof*⟩

lemma *τexec-move-into-τexec-moves*:
τexec-move ci P t e h s s′ ⟹ *τexec-moves ci P t (e # es) h s s′*
⟨*proof*⟩

lemma *τExec-mover-τExec-movesr*:
τExec-mover ci P t e h s s′ ⟹ *τExec-movesr ci P t (e # es) h s s′*
⟨*proof*⟩

lemma *τExec-movet-τExec-movest*:
τExec-movet ci P t e h s s′ ⟹ *τExec-movest ci P t (e # es) h s s′*
⟨*proof*⟩

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′*

⟨*proof*⟩

lemma *τexec-moves-append*: *τexec-moves ci P t es h s s′* ⟹ *τexec-moves ci P t* (*es* @ *es′*) *h s s′*
⟨*proof*⟩

lemma *τExec-movesr-append* [*intro*]:
τExec-movesr ci P t es h s s′ ⟹ *τExec-movesr ci P t* (*es* @ *es′*) *h s s′*
⟨*proof*⟩

lemma *τExec-movest-append* [*intro*]:
τExec-movest ci P t es h s s′ ⟹ *τExec-movest ci P t* (*es* @ *es′*) *h s s′*
⟨*proof*⟩

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*⟩

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′*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

lemma *append-τExec-movest*:
assumes *len*: *length vs* = *length es′*
shows *τExec-movest ci P t es h* (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*)
⟹ *τExec-movest ci P t* (*es′* @ *es*) *h* ((*stk* @ *vs*), *loc*, (*length* (*compEs2 es′*) + *pc*), *xcp*) ((*stk′* @ *vs*), *loc′*, (*length* (*compEs2 es′*) + *pc′*), *xcp′*)
⟨*proof*⟩

lemma *NewArray-τexecI*:
τexec-move ci P t e h s s′ ⟹ *τexec-move ci P t* (*newA T*⌊*e*⌉) *h s s′*
⟨*proof*⟩

lemma *Cast-τexecI*:
τexec-move ci P t e h s s′ ⟹ *τexec-move ci P t* (*Cast T e*) *h s s′*
⟨*proof*⟩

lemma *InstanceOf-τexecI*:
τexec-move ci P t e h s s′ ⟹ *τexec-move ci P t* (*e instanceof T*) *h s s′*
⟨*proof*⟩

lemma *BinOp-τexecI1*:
  τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t (e «bop» e') h s s'*
⟨*proof*⟩

lemma *BinOp-τexecI2*:
  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')*
⟨*proof*⟩

lemma *LAss-τexecI*:
  τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t (V := e) h s s'*
⟨*proof*⟩

lemma *AAcc-τexecI1*:
  τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t (e⌊i⌉) h s s'*
⟨*proof*⟩

lemma *AAcc-τexecI2*:
  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')*
⟨*proof*⟩

lemma *AAss-τexecI1*:
  τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t (e⌊i⌉ := e') h s s'*
⟨*proof*⟩

lemma *AAss-τexecI2*:
  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')*
⟨*proof*⟩

lemma *AAss-τexecI3*:
  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')*
⟨*proof*⟩

lemma *ALength-τexecI*:
  τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t (e•length) h s s'*
⟨*proof*⟩

lemma *FAcc-τexecI*:
  τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t (e•F{D}) h s s'*
⟨*proof*⟩

lemma *FAss-τexecI1*:
  τ*exec-move ci P t e h s s'* $\Longrightarrow$ τ*exec-move ci P t (e•F{D} := e') h s s'*
⟨*proof*⟩

lemma *FAss-τexecI2*:

τexec-move ci P t e′ h (stk, loc, pc, xcp) (stk′, loc′, pc′, xcp′)
⟹ τexec-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′)
⟨*proof*⟩

lemma *CAS-τexecI1*:
τexec-move ci P t e h s s′ ⟹ τexec-move ci P t (e•compareAndSwap(D•F, e′, e′′)) h s s′
⟨*proof*⟩

lemma *CAS-τexecI2*:
τexec-move ci P t e′ h (stk, loc, pc, xcp) (stk′, loc′, pc′, xcp′)
⟹ τexec-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′)
⟨*proof*⟩

lemma *CAS-τexecI3*:
τexec-move ci P t e′′ h (stk, loc, pc, xcp) (stk′, loc′, pc′, xcp′)
⟹ τexec-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′)
⟨*proof*⟩

lemma *Call-τexecI1*:
τexec-move ci P t e h s s′ ⟹ τexec-move ci P t (e•M(es)) h s s′
⟨*proof*⟩

lemma *Call-τexecI2*:
τexec-moves ci P t es h (stk, loc, pc, xcp) (stk′, loc′, pc′, xcp′)
⟹ τexec-move ci P t (e•M(es)) h ((stk @ [v]), loc, (length (compE2 e) + pc), xcp) ((stk′ @ [v]), loc′, (length (compE2 e) + pc′), xcp′)
⟨*proof*⟩

lemma *Block-τexecI-Some*:
τexec-move ci P t e h (stk, loc, pc, xcp) (stk′, loc′, pc′, xcp′)
⟹ τexec-move ci P t {V:T=⌊v⌋; e} h (stk, loc, Suc (Suc pc), xcp) (stk′, loc′, Suc (Suc pc′), xcp′)
⟨*proof*⟩

lemma *Block-τexecI-None*:
τexec-move ci P t e h s s′ ⟹ τexec-move ci P t {V:T=None; e} h s s′
⟨*proof*⟩

lemma *Sync-τexecI*:
τexec-move ci P t e h s s′ ⟹ τexec-move ci P t (sync$_V$ (e) e′) h s s′
⟨*proof*⟩

lemma *Insync-τexecI*:
τexec-move ci P t e′ h (stk, loc, pc, xcp) (stk′, loc′, pc′, xcp′)
⟹ τexec-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′)
⟨*proof*⟩

lemma *Seq-τexecI1*:
τexec-move ci P t e h s s′ ⟹ τexec-move ci P t (e;; e′) h s s′
⟨*proof*⟩

lemma *Seq-τexecI2*:
τexec-move ci P t e′ h (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*)
⟹ *τexec-move ci P t* (*e*;; *e′*) *h* (*stk*, *loc*, *Suc* (*length* (*compE2 e*) + *pc*), *xcp*) (*stk′*, *loc′*, *Suc* (*length* (*compE2 e*) + *pc′*), *xcp′*)
⟨*proof*⟩

lemma *Cond-τexecI1*:
τexec-move ci P t e h s s′ ⟹ *τexec-move ci P t* (*if* (*e*) *e′ else e″*) *h s s′*
⟨*proof*⟩

lemma *Cond-τexecI2*:
τexec-move ci P t e′ h (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*)
⟹ *τexec-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′*)
⟨*proof*⟩

lemma *Cond-τexecI3*:
τexec-move ci P t e″ h (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*)
⟹ *τexec-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′*)
⟨*proof*⟩

lemma *While-τexecI1*:
τexec-move ci P t e h s s′ ⟹ *τexec-move ci P t* (*while* (*e*) *e′*) *h s s′*
⟨*proof*⟩

lemma *While-τexecI2*:
τexec-move ci P t e′ h (*stk*, *loc*, *pc*, *xcp*) (*stk′*, *loc′*, *pc′*, *xcp′*)
⟹ *τexec-move ci P t* (*while* (*e*) *e′*) *h* (*stk*, *loc*, *Suc* (*length* (*compE2 e*) + *pc*), *xcp*) (*stk′*, *loc′*, *Suc* (*length* (*compE2 e*) + *pc′*), *xcp′*)
⟨*proof*⟩

lemma *Throw-τexecI*:
τexec-move ci P t e h s s′ ⟹ *τexec-move ci P t* (*throw e*) *h s s′*
⟨*proof*⟩

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′*
⟨*proof*⟩

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′*)
⟨*proof*⟩

lemma *NewArray-τExecrI*:
τExec-mover ci P t e h s s′ ⟹ *τExec-mover ci P t* (*newA T*⌊*e*⌉) *h s s′*
⟨*proof*⟩

lemma *Cast-τExecrI*:

τ*Exec-mover ci P t e h s s'* $\Longrightarrow$ τ*Exec-mover ci P t (Cast T e) h s s'*
⟨*proof*⟩

lemma *InstanceOf-*τ*ExecrI*:
τ*Exec-mover ci P t e h s s'* $\Longrightarrow$ τ*Exec-mover ci P t (e instanceof T) h s s'*
⟨*proof*⟩

lemma *BinOp-*τ*ExecrI1*:
τ*Exec-mover ci P t e1 h s s'* $\Longrightarrow$ τ*Exec-mover ci P t (e1 «bop» e2) h s s'*
⟨*proof*⟩

lemma *BinOp-*τ*ExecrI2*:
τ*Exec-mover ci P t e2 h (stk, loc, pc, xcp) (stk', loc', pc', xcp')*
$\Longrightarrow$ τ*Exec-mover ci P t (e «bop» e2) h ((stk @ [v]), loc, (length (compE2 e) + pc), xcp) ((stk' @ [v]), loc', (length (compE2 e) + pc'), xcp')*
⟨*proof*⟩

lemma *LAss-*τ*ExecrI*:
τ*Exec-mover ci P t e h s s'* $\Longrightarrow$ τ*Exec-mover ci P t (V := e) h s s'*
⟨*proof*⟩

lemma *AAcc-*τ*ExecrI1*:
τ*Exec-mover ci P t e h s s'* $\Longrightarrow$ τ*Exec-mover ci P t (e⌊i⌉) h s s'*
⟨*proof*⟩

lemma *AAcc-*τ*ExecrI2*:
τ*Exec-mover ci P t i h (stk, loc, pc, xcp) (stk', loc', pc', xcp')*
$\Longrightarrow$ τ*Exec-mover ci P t (a⌊i⌉) h ((stk @ [v]), loc, (length (compE2 a) + pc), xcp) ((stk' @ [v]), loc', (length (compE2 a) + pc'), xcp')*
⟨*proof*⟩

lemma *AAss-*τ*ExecrI1*:
τ*Exec-mover ci P t e h s s'* $\Longrightarrow$ τ*Exec-mover ci P t (e⌊i⌉ := e') h s s'*
⟨*proof*⟩

lemma *AAss-*τ*ExecrI2*:
τ*Exec-mover ci P t i h (stk, loc, pc, xcp) (stk', loc', pc', xcp')*
$\Longrightarrow$ τ*Exec-mover ci P t (a⌊i⌉ := e) h ((stk @ [v]), loc, (length (compE2 a) + pc), xcp) ((stk' @ [v]), loc', (length (compE2 a) + pc'), xcp')*
⟨*proof*⟩

lemma *AAss-*τ*ExecrI3*:
τ*Exec-mover ci P t e h (stk, loc, pc, xcp) (stk', loc', pc', xcp')*
$\Longrightarrow$ τ*Exec-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')*
⟨*proof*⟩

lemma *ALength-*τ*ExecrI*:
τ*Exec-mover ci P t e h s s'* $\Longrightarrow$ τ*Exec-mover ci P t (e·length) h s s'*
⟨*proof*⟩

lemma *FAcc-*τ*ExecrI*:
τ*Exec-mover ci P t e h s s'* $\Longrightarrow$ τ*Exec-mover ci P t (e·F{D}) h s s'*
⟨*proof*⟩

lemma *FAss-τExecrI1*:
$\tau Exec\text{-}mover\ ci\ P\ t\ e\ h\ s\ s' \Longrightarrow \tau Exec\text{-}mover\ ci\ P\ t\ (e\cdot F\{D\} := e')\ h\ s\ s'$
⟨*proof*⟩

lemma *FAss-τExecrI2*:
$\tau Exec\text{-}mover\ ci\ P\ t\ e'\ h\ (stk, loc, pc, xcp)\ (stk', loc', pc', xcp')$
$\Longrightarrow \tau Exec\text{-}mover\ ci\ P\ t\ (e\cdot F\{D\} := e')\ h\ ((stk\ @\ [v]), loc, (length\ (compE2\ e) + pc), xcp)\ ((stk'\ @\ [v]), loc', (length\ (compE2\ e) + pc'), xcp')$
⟨*proof*⟩

lemma *CAS-τExecrI1*:
$\tau Exec\text{-}mover\ ci\ P\ t\ e\ h\ s\ s' \Longrightarrow \tau Exec\text{-}mover\ ci\ P\ t\ (e\cdot compareAndSwap(D\cdot F, e', e''))\ h\ s\ s'$
⟨*proof*⟩

lemma *CAS-τExecrI2*:
$\tau Exec\text{-}mover\ ci\ P\ t\ e'\ h\ (stk, loc, pc, xcp)\ (stk', loc', pc', xcp')$
$\Longrightarrow \tau Exec\text{-}mover\ ci\ P\ t\ (e\cdot compareAndSwap(D\cdot F, e', e''))\ h\ ((stk\ @\ [v]), loc, (length\ (compE2\ e) + pc), xcp)\ ((stk'\ @\ [v]), loc', (length\ (compE2\ e) + pc'), xcp')$
⟨*proof*⟩

lemma *CAS-τExecrI3*:
$\tau Exec\text{-}mover\ ci\ P\ t\ e''\ h\ (stk, loc, pc, xcp)\ (stk', loc', pc', xcp')$
$\Longrightarrow \tau Exec\text{-}mover\ ci\ P\ t\ (e\cdot compareAndSwap(D\cdot F, e', e''))\ h\ ((stk\ @\ [v, v']), loc, (length\ (compE2\ e) + length\ (compE2\ e') + pc), xcp)\ ((stk'\ @\ [v, v']), loc', (length\ (compE2\ e) + length\ (compE2\ e') + pc'), xcp')$
⟨*proof*⟩

lemma *Call-τExecrI1*:
$\tau Exec\text{-}mover\ ci\ P\ t\ obj\ h\ s\ s' \Longrightarrow \tau Exec\text{-}mover\ ci\ P\ t\ (obj\cdot M'(es))\ h\ s\ s'$
⟨*proof*⟩

lemma *Call-τExecrI2*:
$\tau Exec\text{-}movesr\ ci\ P\ t\ es\ h\ (stk, loc, pc, xcp)\ (stk', loc', pc', xcp')$
$\Longrightarrow \tau Exec\text{-}mover\ ci\ P\ t\ (obj\cdot M'(es))\ h\ ((stk\ @\ [v]), loc, (length\ (compE2\ obj) + pc), xcp)\ ((stk'\ @\ [v]), loc', (length\ (compE2\ obj) + pc'), xcp')$
⟨*proof*⟩

lemma *Block-τExecrI-Some*:
$\tau Exec\text{-}mover\ ci\ P\ t\ e\ h\ (stk, loc, pc, xcp)\ (stk', loc', pc', xcp')$
$\Longrightarrow \tau Exec\text{-}mover\ ci\ P\ t\ \{V{:}T{=}\lfloor v\rfloor;\ e\}\ h\ (stk, loc, (Suc\ (Suc\ pc)), xcp)\ (stk', loc', (Suc\ (Suc\ pc')), xcp')$
⟨*proof*⟩

lemma *Block-τExecrI-None*:
$\tau Exec\text{-}mover\ ci\ P\ t\ e\ h\ s\ s' \Longrightarrow \tau Exec\text{-}mover\ ci\ P\ t\ \{V{:}T{=}None;\ e\}\ h\ s\ s'$
⟨*proof*⟩

lemma *Sync-τExecrI*:
$\tau Exec\text{-}mover\ ci\ P\ t\ e\ h\ s\ s' \Longrightarrow \tau Exec\text{-}mover\ ci\ P\ t\ (sync_V\ (e)\ e')\ h\ s\ s'$
⟨*proof*⟩

lemma *Insync-τExecrI*:
$\tau Exec\text{-}mover\ ci\ P\ t\ e'\ h\ (stk, loc, pc, xcp)\ (stk', loc', pc', xcp')$

$\Longrightarrow$ *τExec-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'*)
⟨*proof*⟩

lemma *Seq-τExecrI1*:
τExec-mover ci P t e h s s' $\Longrightarrow$ *τExec-mover ci P t* (*e*;;*e'*) *h s s'*
⟨*proof*⟩

lemma *Seq-τExecrI2*:
τExec-mover ci P t e h (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'* ,*xcp'*) $\Longrightarrow$
τExec-mover ci P t (*e'*;;*e*) *h* (*stk*, *loc*, (*Suc* (*length* (*compE2 e'*) + *pc*)), *xcp*) (*stk'*, *loc'*, (*Suc* (*length* (*compE2 e'*) + *pc'*)), *xcp'*)
⟨*proof*⟩

lemma *Cond-τExecrI1*:
τExec-mover ci P t e h s s' $\Longrightarrow$ *τExec-mover ci P t* (*if* (*e*) *e1 else e2*) *h s s'*
⟨*proof*⟩

lemma *Cond-τExecrI2*:
τExec-mover ci P t e1 h (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*) $\Longrightarrow$
τExec-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'*)
⟨*proof*⟩

lemma *Cond-τExecrI3*:
τExec-mover ci P t e2 h (*stk*, *loc* ,*pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*) $\Longrightarrow$
τExec-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'*)
⟨*proof*⟩

lemma *While-τExecrI1*:
τExec-mover ci P t c h s s' $\Longrightarrow$ *τExec-mover ci P t* (*while* (*c*) *e*) *h s s'*
⟨*proof*⟩

lemma *While-τExecrI2*:
τExec-mover ci P t E h (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ *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'*)
⟨*proof*⟩

lemma *Throw-τExecrI*:
τExec-mover ci P t e h s s' $\Longrightarrow$ *τExec-mover ci P t* (*throw e*) *h s s'*
⟨*proof*⟩

lemma *Try-τExecrI1*:
τExec-mover ci P t E h s s' $\Longrightarrow$ *τExec-mover ci P t* (*try E catch*(*C' V*) *e*) *h s s'*
⟨*proof*⟩

lemma *Try-τExecrI2*:
τExec-mover ci P t e h (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ *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'*)
⟨*proof*⟩

lemma *NewArray-τExectI*:
τExec-movet ci P t e h s s' ⟹ *τExec-movet ci P t (newA T⌊e⌉) h s s'*
⟨*proof*⟩

lemma *Cast-τExectI*:
τExec-movet ci P t e h s s' ⟹ *τExec-movet ci P t (Cast T e) h s s'*
⟨*proof*⟩

lemma *InstanceOf-τExectI*:
τExec-movet ci P t e h s s' ⟹ *τExec-movet ci P t (e instanceof T) h s s'*
⟨*proof*⟩

lemma *BinOp-τExectI1*:
τExec-movet ci P t e1 h s s' ⟹ *τExec-movet ci P t (e1 «bop» e2) h s s'*
⟨*proof*⟩

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')*
⟨*proof*⟩

lemma *LAss-τExectI*:
τExec-movet ci P t e h s s' ⟹ *τExec-movet ci P t (V := e) h s s'*
⟨*proof*⟩

lemma *AAcc-τExectI1*:
τExec-movet ci P t e h s s' ⟹ *τExec-movet ci P t (e⌊i⌉) h s s'*
⟨*proof*⟩

lemma *AAcc-τExectI2*:
τExec-movet ci P t i h (stk, loc, pc, xcp) (stk', loc', pc', xcp')
⟹ *τExec-movet ci P t (a⌊i⌉) h ((stk @ [v]), loc, (length (compE2 a) + pc), xcp) ((stk' @ [v]), loc', (length (compE2 a) + pc'), xcp')*
⟨*proof*⟩

lemma *AAss-τExectI1*:
τExec-movet ci P t e h s s' ⟹ *τExec-movet ci P t (e⌊i⌉ := e') h s s'*
⟨*proof*⟩

lemma *AAss-τExectI2*:
τExec-movet ci P t i h (stk, loc, pc, xcp) (stk', loc', pc', xcp')
⟹ *τExec-movet ci P t (a⌊i⌉ := e) h ((stk @ [v]), loc, (length (compE2 a) + pc), xcp) ((stk' @ [v]), loc', (length (compE2 a) + pc'), xcp')*
⟨*proof*⟩

lemma *AAss-τExectI3*:
τExec-movet ci P t e h (stk, loc, pc, xcp) (stk', loc', pc', xcp')
⟹ *τExec-movet 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')*
⟨*proof*⟩

lemma *ALength-τExectI*:

τ*Exec-movet ci P t e h s s'* $\Longrightarrow$ τ*Exec-movet ci P t* (*e*•*length*) *h s s'*
⟨*proof*⟩

lemma *FAcc-*τ*ExectI*:
τ*Exec-movet ci P t e h s s'* $\Longrightarrow$ τ*Exec-movet ci P t* (*e*•*F*{*D*}) *h s s'*
⟨*proof*⟩

lemma *FAss-*τ*ExectI1*:
τ*Exec-movet ci P t e h s s'* $\Longrightarrow$ τ*Exec-movet ci P t* (*e*•*F*{*D*} := *e'*) *h s s'*
⟨*proof*⟩

lemma *FAss-*τ*ExectI2*:
τ*Exec-movet ci P t e' h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*Exec-movet ci P t* (*e*•*F*{*D*} := *e'*) *h* ((*stk* @ [*v*]), *loc*, (*length* (*compE2 e*) + *pc*), *xcp*) ((*stk'* @ [*v*]), *loc'*, (*length* (*compE2 e*) + *pc'*), *xcp'*)
⟨*proof*⟩

lemma *CAS-*τ*ExectI1*:
τ*Exec-movet ci P t e h s s'* $\Longrightarrow$ τ*Exec-movet ci P t* (*e*•*compareAndSwap*(*D*•*F*, *e'*, *e''*)) *h s s'*
⟨*proof*⟩

lemma *CAS-*τ*ExectI2*:
τ*Exec-movet ci P t e' h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*Exec-movet 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'*)
⟨*proof*⟩

lemma *CAS-*τ*ExectI3*:
τ*Exec-movet ci P t e'' h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*Exec-movet 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'*)
⟨*proof*⟩

lemma *Call-*τ*ExectI1*:
τ*Exec-movet ci P t obj h s s'* $\Longrightarrow$ τ*Exec-movet ci P t* (*obj*•*M'*(*es*)) *h s s'*
⟨*proof*⟩

lemma *Call-*τ*ExectI2*:
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'*)
⟨*proof*⟩

lemma *Block-*τ*ExectI-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'*)
⟨*proof*⟩

lemma *Block-*τ*ExectI-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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

lemma *While-τExectI1*:
τ*Exec-movet ci P t c h s s'* $\Longrightarrow$ τ*Exec-movet ci P t* (*while* (*c*) *e*) *h s s'*
⟨*proof*⟩

lemma *While-τExectI2*:
τ*Exec-movet ci P t E h* (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ τ*Exec-movet ci P t* (*while* (*c*) *E*) *h* (*stk*, *loc* ,(*Suc* (*length* (*compE2 c*) + *pc*)), *xcp*) (*stk'*, *loc'*, (*Suc* (*length* (*compE2 c*) + *pc'*)), *xcp'*)
⟨*proof*⟩

lemma *Throw-τExectI*:
τ*Exec-movet ci P t e h s s'* $\Longrightarrow$ τ*Exec-movet ci P t* (*throw e*) *h s s'*
⟨*proof*⟩

lemma *Try-τExectI1*:
τ*Exec-movet ci P t E h s s'* $\Longrightarrow$ τ*Exec-movet ci P t* (*try E catch*(*C' V*) *e*) *h s s'*
⟨*proof*⟩

lemma *Try-τExectI2*:
τExec-movet ci P t e h (*stk*, *loc*, *pc*, *xcp*) (*stk'*, *loc'*, *pc'*, *xcp'*)
$\Longrightarrow$ *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'*)
⟨*proof*⟩

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*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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* ⟧ $\Longrightarrow$ *τexec-1 P t σ σ'*
⟨*proof*⟩

lemma *τexec-1E* [*elim*]:
assumes *τexec-1 P t σ σ'*
obtains *exec-1 P t σ ε σ' τMove2 P σ*
⟨*proof*⟩

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* ⟧ $\Longrightarrow$ *τexec-1-d P t σ σ'*
⟨*proof*⟩

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*
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *τexec-move-τexec-1*:
assumes *exec*: *τexec-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 *τexec-1* (*compP2 P*) *t* (*xcp*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs*) (*xcp'*, *h*, (*stk'*, *loc'*, *C*, *M*, *pc'*) # *frs*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *exec-meth-length-pc-xt-Nil*:
exec-meth ci P ins [] *t h* (*stk*, *loc*, *pc*, *xcp*) *ta h′ s′* ⟹ *pc* < *length ins*
⟨*proof*⟩

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*⟩

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*⟩

lemma *exec-move-length-compE2D* [*dest*]:
exec-move ci P t e h (*stk*, *loc*, *pc*, *xcp*) *ta h′ s′* ⟹ *pc* < *length* (*compE2 e*)
⟨*proof*⟩

lemma *exec-moves-length-compEs2D* [*dest*]:
exec-moves ci P t es h (*stk*, *loc*, *pc*, *xcp*) *ta h′ s′* ⟹ *pc* < *length* (*compEs2 es*)
⟨*proof*⟩

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* []

⟨*proof*⟩

lemma *exec-move-ci-appD*:
exec-move ci P t E h (*stk*, *loc*, *pc*, *None*) *ta h′ fr′*
$\Longrightarrow$ *ci-app ci* (*compE2 E* ! *pc*) (*compP2 P*) *h stk loc undefined undefined pc* []
⟨*proof*⟩

lemma *exec-moves-ci-appD*:
exec-moves ci P t Es h (*stk*, *loc*, *pc*, *None*) *ta h′ fr′*
$\Longrightarrow$ *ci-app ci* (*compEs2 Es* ! *pc*) (*compP2 P*) *h stk loc undefined undefined pc* []
⟨*proof*⟩

lemma *τinstr-stk-append-check*:
check-instr′ i P h stk loc C M pc frs $\Longrightarrow$ *τinstr P h* (*stk* @ *vs*) *i* = *τinstr P h stk i*
⟨*proof*⟩

lemma *τinstr-stk-drop-exec-move*:
exec-move ci P t e h (*stk*, *loc*, *pc*, *None*) *ta h′ fr′*
$\Longrightarrow$ *τinstr* (*compP2 P*) *h* (*stk* @ *vs*) (*compE2 e* ! *pc*) = *τinstr* (*compP2 P*) *h stk* (*compE2 e* ! *pc*)
⟨*proof*⟩

lemma *τinstr-stk-drop-exec-moves*:
exec-moves ci P t es h (*stk*, *loc*, *pc*, *None*) *ta h′ fr′*
$\Longrightarrow$ *τinstr* (*compP2 P*) *h* (*stk* @ *vs*) (*compEs2 es* ! *pc*) = *τinstr* (*compP2 P*) *h stk* (*compEs2 es* ! *pc*)
⟨*proof*⟩

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*] ⟧ $\Longrightarrow$ *τexec-1-d P t σ σ′*
⟨*proof*⟩

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*] $\Longrightarrow$ Φ |− *t*:*σ′* [*ok*]
⟨*proof*⟩

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*] $\Longrightarrow$ Φ |− *t*:*σ′* [*ok*]
⟨*proof*⟩

lemma *τExec-1r-τExec-1-dr*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
shows ⟦ *τExec-1r P t σ σ′*; Φ |− *t*:*σ* [*ok*] ⟧ $\Longrightarrow$ *τExec-1-dr P t σ σ′*
⟨*proof*⟩

lemma *τExec-1t-τExec-1-dt*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
shows ⟦ *τExec-1t P t σ σ′*; Φ |− *t*:*σ* [*ok*] ⟧ $\Longrightarrow$ *τExec-1-dt P t σ σ′*
⟨*proof*⟩

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*] $\Longrightarrow$ Φ |− *t*: *σ′* [*ok*]
⟨*proof*⟩

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*] $\Longrightarrow$ Φ |− *t*:*σ′* [*ok*]
⟨*proof*⟩

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*)

$\Rightarrow$ (*'addr val list* $\times$ *'addr val list* $\times$ *pc* $\times$ *'addr option*) $\Rightarrow$ *bool*
(‹-,-,- ⊢ - ↔ -› [*50*, *0*, *0*, *0*, *50*] *100*)

and *bisims1-syntax* ::
'm prog $\Rightarrow$ *'addr expr1 list* $\Rightarrow$ *'heap* $\Rightarrow$ (*'addr expr1 list* $\times$ *'addr locals1*)
$\Rightarrow$ (*'addr val list* $\times$ *'addr val list* $\times$ *pc* $\times$ *'addr option*) $\Rightarrow$ *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*) $\Longrightarrow$ *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*) $\Longrightarrow$ *P*, *newA T*⌊*e*⌉, *h* ⊢ (*newA T*⌊*e'*⌉, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1NewArrayThrow*:
P, *e*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋) $\Longrightarrow$ *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* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*) $\Longrightarrow$ *P*, *Cast T e*, *h* ⊢ (*Cast T e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1CastThrow*:
P, *e*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋) $\Longrightarrow$ *P*, *Cast T e*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)

| *bisim1CastFail*:
P, *Cast T e*, *h* ⊢ (*THROW ClassCast*, *xs*) ↔ ([*v*], *xs*, *length* (*compE2 e*), ⌊*addr-of-sys-xcpt ClassCast*⌋)

| *bisim1InstanceOf*:
P, *e*, *h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*) $\Longrightarrow$ *P*, *e instanceof T*, *h* ⊢ (*e' instanceof T*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1InstanceOfThrow*:
P, *e*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋) $\Longrightarrow$ *P*, *e instanceof T*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)

| *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*:
$P,\ a,\ h \vdash (a',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ a{\bullet}length,\ h \vdash (a'{\bullet}length,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1ALengthThrow*:
$P,\ a,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor ad\rfloor)$
$\Longrightarrow P,\ a{\bullet}length,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor ad\rfloor)$

| *bisim1ALengthNull*:
$P,\ a{\bullet}length,\ h \vdash (THROW\ NullPointer,\ xs) \leftrightarrow ([Null],\ xs,\ length\ (compE2\ a),\ \lfloor addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer\rfloor)$

| *bisim1FAcc*:
$P,\ e,\ h \vdash (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ e{\bullet}F\{D\},\ h \vdash (e'{\bullet}F\{D\},\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1FAccThrow*:
$P,\ e,\ h \vdash (Throw\ ad,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor ad\rfloor)$

$\Longrightarrow$ *P*, *e*•*F*{*D*}, *h* ⊢ (*Throw ad*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*ad*⌋)

| *bisim1FAccNull*:
P, *e*•*F*{*D*}, *h* ⊢ (*THROW NullPointer*, *xs*) ↔ ([*Null*], *xs*, *length* (*compE2 e*), ⌊*addr-of-sys-xcpt NullPointer*⌋)

| *bisim1FAss1*:
P, *e*, *h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
$\Longrightarrow$ *P*, *e*•*F*{*D*} := *e2*, *h* ⊢ (*e'*•*F*{*D*} := *e2*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1FAss2*:
P, *e2*, *h* ⊢ (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
$\Longrightarrow$ *P*, *e*•*F*{*D*} := *e2*, *h* ⊢ (*Val v*•*F*{*D*} := *e'*, *xs*) ↔ (*stk* @ [*v*], *loc*, *length* (*compE2 e*) + *pc*, *xcp*)

| *bisim1FAssThrow1*:
P, *e*, *h* ⊢ (*Throw ad*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*ad*⌋)
$\Longrightarrow$ *P*, *e*•*F*{*D*} := *e2*, *h* ⊢ (*Throw ad*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*ad*⌋)

| *bisim1FAssThrow2*:
P, *e2*, *h* ⊢ (*Throw ad*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*ad*⌋)
$\Longrightarrow$ *P*, *e*•*F*{*D*} := *e2*, *h* ⊢ (*Throw ad*, *xs*) ↔ (*stk* @ [*v*], *loc*, *length* (*compE2 e*) + *pc*, ⌊*ad*⌋)

| *bisim1FAssNull*:
P, *e*•*F*{*D*} := *e2*, *h* ⊢ (*THROW NullPointer*, *xs*) ↔ ([*v*, *Null*], *xs*, *length* (*compE2 e*) + *length* (*compE2 e2*), ⌊*addr-of-sys-xcpt NullPointer*⌋)

| *bisim1FAss3*:
P, *e*•*F*{*D*} := *e2*, *h* ⊢ (*unit*, *xs*) ↔ ([], *xs*, *Suc* (*length* (*compE2 e*) + *length* (*compE2 e2*)), *None*)

| *bisim1CAS1*:
P, *e1*, *h* ⊢ (*e1'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
$\Longrightarrow$ *P*, *e1*•*compareAndSwap*(*D*•*F*, *e2*, *e3*), *h* ⊢ (*e1'*•*compareAndSwap*(*D*•*F*, *e2*, *e3*), *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1CAS2*:
P, *e2*, *h* ⊢ (*e2'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
$\Longrightarrow$ *P*, *e1*•*compareAndSwap*(*D*•*F*, *e2*, *e3*), *h* ⊢ (*Val v*•*compareAndSwap*(*D*•*F*, *e2'*, *e3*), *xs*) ↔ (*stk* @ [*v*], *loc*, *length* (*compE2 e1*) + *pc*, *xcp*)

| *bisim1CAS3*:
P, *e3*, *h* ⊢ (*e3'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
$\Longrightarrow$ *P*, *e1*•*compareAndSwap*(*D*•*F*, *e2*, *e3*), *h* ⊢ (*Val v*•*compareAndSwap*(*D*•*F*, *Val v'*, *e3'*), *xs*) ↔ (*stk* @ [*v'*, *v*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*, *xcp*)

| *bisim1CASThrow1*:
P, *e1*, *h* ⊢ (*Throw ad*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*ad*⌋)
$\Longrightarrow$ *P*, *e1*•*compareAndSwap*(*D*•*F*, *e2*, *e3*), *h* ⊢ (*Throw ad*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*ad*⌋)

| *bisim1CASThrow2*:
P, *e2*, *h* ⊢ (*Throw ad*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*ad*⌋)
$\Longrightarrow$ *P*, *e1*•*compareAndSwap*(*D*•*F*, *e2*, *e3*), *h* ⊢ (*Throw ad*, *xs*) ↔ (*stk* @ [*v*], *loc*, *length* (*compE2 e1*) + *pc*, ⌊*ad*⌋)

| *bisim1CASThrow3*:
P, *e3*, *h* ⊢ (*Throw ad*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*ad*⌋)
⟹ *P*, *e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *h* ⊢ (*Throw ad*, *xs*) ↔ (*stk* @ [*v′*, *v*], *loc*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *pc*, ⌊*ad*⌋)

| *bisim1CASFail*:
P, *e1*·*compareAndSwap*(*D*·*F*, *e2*, *e3*), *h* ⊢ (*Throw ad*, *xs*) ↔ ([*v′*, *v*, *v″*], *xs*, *length* (*compE2 e1*) + *length* (*compE2 e2*) + *length* (*compE2 e3*), ⌊*ad*⌋)

| *bisim1Call1*:
P, *obj*, *h* ⊢ (*obj′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
⟹ *P*, *obj*·*M*(*ps*), *h* ⊢ (*obj′*·*M*(*ps*), *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1CallParams*:
P, *ps*, *h* ⊢ (*ps′*, *xs*) [↔] (*stk*, *loc*, *pc*, *xcp*)
⟹ *P*, *obj*·*M*(*ps*), *h* ⊢ (*Val v*·*M*(*ps′*), *xs*) ↔ (*stk* @ [*v*], *loc*, *length* (*compE2 obj*) + *pc*, *xcp*)

| *bisim1CallThrowObj*:
P, *obj*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
⟹ *P*, *obj*·*M*(*ps*), *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)

| *bisim1CallThrowParams*:
P, *ps*, *h* ⊢ (*map Val vs* @ *Throw a* # *ps′*, *xs*) [↔] (*stk*, *loc*, *pc*, ⌊*a*⌋)
⟹ *P*, *obj*·*M*(*ps*), *h* ⊢ (*Throw a*, *xs*) ↔ (*stk* @ [*v*], *loc*, *length* (*compE2 obj*) + *pc*, ⌊*a*⌋)

| *bisim1CallThrow*:
length ps = *length vs*
⟹ *P*, *obj*·*M*(*ps*), *h* ⊢ (*Throw a*, *xs*) ↔ (*vs* @ [*v*], *xs*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), ⌊*a*⌋)

| *bisim1BlockSome1*:
P, {*V*:*T*=⌊*v*⌋; *e*}, *h* ⊢ ({*V*:*T*=⌊*v*⌋; *e*}, *xs*) ↔ ([], *xs*, *0*, *None*)

| *bisim1BlockSome2*:
P, {*V*:*T*=⌊*v*⌋; *e*}, *h* ⊢ ({*V*:*T*=⌊*v*⌋; *e*}, *xs*) ↔ ([*v*], *xs*, *Suc 0*, *None*)

| *bisim1BlockSome4*:
P, *e*, *h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
⟹ *P*, {*V*:*T*=⌊*v*⌋; *e*}, *h* ⊢ ({*V*:*T*=*None*; *e′*}, *xs*) ↔ (*stk*, *loc*, *Suc* (*Suc pc*), *xcp*)

| *bisim1BlockThrowSome*:
P, *e*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
⟹ *P*, {*V*:*T*=⌊*v*⌋; *e*}, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *Suc* (*Suc pc*), ⌊*a*⌋)

| *bisim1BlockNone*:
P, *e*, *h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
⟹ *P*, {*V*:*T*=*None*; *e*}, *h* ⊢ ({*V*:*T*=*None*; *e′*}, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1BlockThrowNone*:
P, *e*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
⟹ *P*, {*V*:*T*=*None*; *e*}, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)

| *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*:
$P,\ sync_V\ (e1)\ e2,\ h \vdash (Throw\ a,\ xs) \leftrightarrow ([Addr\ a],\ xs,\ 8 + length\ (compE2\ e1) + length\ (compE2\ e2),\ None)$

| *bisim1Sync10*:
$P,\ sync_V\ (e1)\ e2,\ h \vdash (Throw\ a,\ xs) \leftrightarrow ([Addr\ a],\ xs,\ 8 + length\ (compE2\ e1) + length\ (compE2\ e2),\ \lfloor a \rfloor)$

| *bisim1Sync11*:
$P,\ sync_V\ (e1)\ e2,\ h \vdash (THROW\ NullPointer,\ xs) \leftrightarrow ([Null],\ xs,\ Suc\ (Suc\ (length\ (compE2\ e1))),\ \lfloor addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer \rfloor)$

| *bisim1Sync12*:
$P,\ sync_V\ (e1)\ e2,\ h \vdash (Throw\ a,\ xs) \leftrightarrow ([v,\ v'],\ xs,\ 4 + length\ (compE2\ e1) + length\ (compE2\ e2),\ \lfloor a \rfloor)$

| *bisim1Sync14*:
$P,\ sync_V\ (e1)\ e2,\ h \vdash (Throw\ a,\ xs) \leftrightarrow$
$([v,\ Addr\ a'],\ xs,\ 7 + length\ (compE2\ e1) + length\ (compE2\ e2),\ \lfloor a \rfloor)$

| *bisim1SyncThrow*:
$P,\ e1,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$

$\Longrightarrow$ *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*) $\Longrightarrow$ *P*, *e1*;;*e2*, *h* ⊢ (*e′*;;*e2*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1SeqThrow1*:
P, *e1*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋) $\Longrightarrow$ *P*, *e1*;;*e2*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)

| *bisim1Seq2*:
P, *e2*, *h* ⊢ *exs* ↔ (*stk*, *loc*, *pc*, *xcp*)
$\Longrightarrow$ *P*, *e1*;;*e2*, *h* ⊢ *exs* ↔ (*stk*, *loc*, *Suc* (*length* (*compE2 e1*) + *pc*), *xcp*)

| *bisim1Cond1*:
P, *e*, *h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
$\Longrightarrow$ *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*)
$\Longrightarrow$ *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*)
$\Longrightarrow$ *P*, *if* (*e*) *e1 else e2*, *h* ⊢ *exs* ↔ (*stk*, *loc*, *Suc* (*Suc* (*length* (*compE2 e*) + *length* (*compE2 e1*) + *pc*)), *xcp*)

| *bisim1CondThrow*:
P, *e*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
$\Longrightarrow$ *P*, *if* (*e*) *e1 else e2*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)

| *bisim1While1*:
P, *while* (*c*) *e*, *h* ⊢ (*while* (*c*) *e*, *xs*) ↔ ([], *xs*, *0*, *None*)

| *bisim1While3*:
P, *c*, *h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
$\Longrightarrow$ *P*, *while* (*c*) *e*, *h* ⊢ (*if* (*e′*) (*e*;; *while* (*c*) *e*) *else unit*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1While4*:
P, *e*, *h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
$\Longrightarrow$ *P*, *while* (*c*) *e*, *h* ⊢ (*e′*;; *while* (*c*) *e*, *xs*) ↔ (*stk*, *loc*, *Suc* (*length* (*compE2 c*) + *pc*), *xcp*)

| *bisim1While6*:
P, *while* (*c*) *e*, *h* ⊢ (*while* (*c*) *e*, *xs*) ↔ ([], *xs*, *Suc* (*Suc* (*length* (*compE2 c*) + *length* (*compE2 e*))), *None*)

| *bisim1While7*:
P, *while* (*c*) *e*, *h* ⊢ (*unit*, *xs*) ↔ ([], *xs*, *Suc* (*Suc* (*Suc* (*length* (*compE2 c*) + *length* (*compE2 e*)))), *None*)

| *bisim1WhileThrow1*:
P, *c*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
⟹ *P*, *while* (*c*) *e*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)

| *bisim1WhileThrow2*:
P, *e*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)
⟹ *P*, *while* (*c*) *e*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *Suc* (*length* (*compE2 c*) + *pc*), ⌊*a*⌋)

| *bisim1Throw1*:
P, *e*, *h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*) ⟹ *P*, *throw e*, *h* ⊢ (*throw e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1Throw2*:
P, *throw e*, *h* ⊢ (*Throw a*, *xs*) ↔ ([*Addr a*], *xs*, *length* (*compE2 e*), ⌊*a*⌋)

| *bisim1ThrowNull*:
P, *throw e*, *h* ⊢ (*THROW NullPointer*, *xs*) ↔ ([*Null*], *xs*, *length* (*compE2 e*), ⌊*addr-of-sys-xcpt NullPointer*⌋)

| *bisim1ThrowThrow*:
P, *e*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋) ⟹ *P*, *throw e*, *h* ⊢ (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)

| *bisim1Try*:
P, *e*, *h* ⊢ (*e′*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)
⟹ *P*, *try e catch*(*C V*) *e2*, *h* ⊢ (*try e′ catch*(*C V*) *e2*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1TryCatch1*:
⟦ *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* ⊢ ({*V*:*Class C*=*None*; *e2*}, *xs*[*V* := *Addr a*]) ↔ ([*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*)

$\Longrightarrow$ *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*)
⟨*proof*⟩

lemma *bisims1-lengthD*: *P*, *es*, *h* ⊢ (*es'*, *xs*) [↔] *s* $\Longrightarrow$ *length es* = *length es'*
⟨*proof*⟩

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*)
$\Longrightarrow$ *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*⌋)

$\Longrightarrow P,\ newA\ T\lfloor e\rceil,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a\rfloor)$

| *bisim1NewArrayFail'*:
$(\bigwedge xs.\ P,\ e,\ n,\ h \vdash' (e,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None))$
$\Longrightarrow P,\ newA\ T\lfloor e\rceil,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow ([v],\ xs,\ length\ (compE2\ e),\ \lfloor a\rfloor)$

| *bisim1Cast'*:
$P,\ e,\ n,\ h \vdash' (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ Cast\ T\ e,\ n,\ h \vdash' (Cast\ T\ e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1CastThrow'*:
$P,\ e,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a\rfloor)$
$\Longrightarrow P,\ Cast\ T\ e,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a\rfloor)$

| *bisim1CastFail'*:
$(\bigwedge xs.\ P,\ e,\ n,\ h \vdash' (e,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None))$
$\Longrightarrow P,\ Cast\ T\ e,\ n,\ h \vdash' (THROW\ ClassCast,\ xs) \leftrightarrow ([v],\ xs,\ length\ (compE2\ e),\ \lfloor addr\text{-}of\text{-}sys\text{-}xcpt\ ClassCast\rfloor)$

| *bisim1InstanceOf'*:
$P,\ e,\ n,\ h \vdash' (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow P,\ e\ instanceof\ T,\ n,\ h \vdash' (e'\ instanceof\ T,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *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'*:
$[\![\ P,\ e1,\ n,\ h \vdash' (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp);$
$\bigwedge xs.\ P,\ e2,\ n,\ h \vdash' (e2,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ e1 «bop» e2,\ n,\ h \vdash' (e' «bop» e2,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp)$

| *bisim1BinOp2'*:
$[\![\ P,\ e2,\ n,\ h \vdash' (e',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp);$
$\bigwedge xs.\ P,\ e1,\ n,\ h \vdash' (e1,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ e1 «bop» e2,\ n,\ h \vdash' (Val\ v1\ «bop»\ e',\ xs) \leftrightarrow (stk\ @\ [v1],\ loc,\ length\ (compE2\ e1) + pc,\ xcp)$

| *bisim1BinOpThrow1'*:
$[\![\ P,\ e1,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a\rfloor);$
$\bigwedge xs.\ P,\ e2,\ n,\ h \vdash' (e2,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ e1 «bop» e2,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a\rfloor)$

| *bisim1BinOpThrow2'*:
$[\![\ P,\ e2,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a\rfloor);$
$\bigwedge xs.\ P,\ e1,\ n,\ h \vdash' (e1,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None)\ ]\!]$
$\Longrightarrow P,\ e1 «bop» e2,\ n,\ h \vdash' (Throw\ a,\ xs) \leftrightarrow (stk\ @\ [v1],\ loc,\ length\ (compE2\ e1) + pc,\ \lfloor a\rfloor)$

| *bisim1BinOpThrow′*:
⟦ ⋀xs. P, e1, n, h ⊢′ (e1, xs) ↔ ([], xs, 0, None);
⋀xs. P, e2, n, h ⊢′ (e2, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, e1«bop»e2, n, h ⊢′ (Throw a, xs) ↔ ([v1, v2], xs, length (compE2 e1) + length (compE2 e2), ⌊a⌋)

| *bisim1LAss1′*:
P, e, n, h ⊢′ (e′, xs) ↔ (stk, loc, pc, xcp)
⟹ P, V:=e, n, h ⊢′ (V:=e′, xs) ↔ (stk, loc, pc, xcp)

| *bisim1LAss2′*:
(⋀xs. P, e, n, h ⊢′ (e, xs) ↔ ([], xs, 0, None))
⟹ P, V:=e, n, h ⊢′ (unit, xs) ↔ ([], xs, Suc (length (compE2 e)), None)

| *bisim1LAssThrow′*:
P, e, n, h ⊢′ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)
⟹ P, V:=e, n, h ⊢′ (Throw a, xs) ↔ (stk, loc, pc, ⌊a⌋)

| *bisim1AAcc1′*:
⟦ P, a, n, h ⊢′ (a′, xs) ↔ (stk, loc, pc, xcp); ⋀xs. P, i, n, h ⊢′ (i, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, a⌊i⌉, n, h ⊢′ (a′⌊i⌉, xs) ↔ (stk, loc, pc, xcp)

| *bisim1AAcc2′*:
⟦ P, i, n, h ⊢′ (i′, xs) ↔ (stk, loc, pc, xcp); ⋀xs. P, a, n, h ⊢′ (a, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, a⌊i⌉, n, h ⊢′ (Val v⌊i′⌉, xs) ↔ (stk @ [v], loc, length (compE2 a) + pc, xcp)

| *bisim1AAccThrow1′*:
⟦ P, a, n, h ⊢′ (Throw ad, xs) ↔ (stk, loc, pc, ⌊ad⌋);
⋀xs. P, i, n, h ⊢′ (i, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, a⌊i⌉, n, h ⊢′ (Throw ad, xs) ↔ (stk, loc, pc, ⌊ad⌋)

| *bisim1AAccThrow2′*:
⟦ P, i, n, h ⊢′ (Throw ad, xs) ↔ (stk, loc, pc, ⌊ad⌋);
⋀xs. P, a, n, h ⊢′ (a, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, a⌊i⌉, n, h ⊢′ (Throw ad, xs) ↔ (stk @ [v], loc, length (compE2 a) + pc, ⌊ad⌋)

| *bisim1AAccFail′*:
⟦ ⋀xs. P, a, n, h ⊢′ (a, xs) ↔ ([], xs, 0, None); ⋀xs. P, i, n, h ⊢′ (i, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, a⌊i⌉, n, h ⊢′ (Throw ad, xs) ↔ ([v, v′], xs, length (compE2 a) + length (compE2 i), ⌊ad⌋)

| *bisim1AAss1′*:
⟦ P, a, n, h ⊢′ (a′, xs) ↔ (stk, loc, pc, xcp);
⋀xs. P, i, n, h ⊢′ (i, xs) ↔ ([], xs, 0, None);
⋀xs. P, e, n, h ⊢′ (e, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, a⌊i⌉ := e, n, h ⊢′ (a′⌊i⌉ := e, xs) ↔ (stk, loc, pc, xcp)

| *bisim1AAss2′*:
⟦ P, i, n, h ⊢′ (i′, xs) ↔ (stk, loc, pc, xcp);
⋀xs. P, a, n, h ⊢′ (a, xs) ↔ ([], xs, 0, None);
⋀xs. P, e, n, h ⊢′ (e, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, a⌊i⌉ := e, n, h ⊢′ (Val v⌊i′⌉ := e, xs) ↔ (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⌋)

| *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′*:
  $[\![P,\ e1,\ n,\ h \vdash' (e1',\ xs) \leftrightarrow (stk,\ loc,\ pc,\ xcp);$
    $\bigwedge xs.\ P,\ e2,\ n,\ h \vdash' (e2,\ xs) \leftrightarrow ([],\ xs,\ 0,\ None);$

⋀xs. P, e3, n, h ⊢′ (e3, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, e1·compareAndSwap(D·F, e2, e3), n, h ⊢′ (e1′·compareAndSwap(D·F, e2, e3), xs) ↔ (stk, loc, pc, xcp)

| bisim1CAS2′:
⟦ P, e2, n, h ⊢′ (e2′, xs) ↔ (stk, loc, pc, xcp);
⋀xs. P, e1, n, h ⊢′ (e1, xs) ↔ ([], xs, 0, None);
⋀xs. P, e3, n, h ⊢′ (e3, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, e1·compareAndSwap(D·F, e2, e3), n, h ⊢′ (Val v·compareAndSwap(D·F, e2′, e3), xs) ↔ (stk @ [v], loc, length (compE2 e1) + pc, xcp)

| bisim1CAS3′:
⟦ P, e3, n, h ⊢′ (e3′, xs) ↔ (stk, loc, pc, xcp);
⋀xs. P, e1, n, h ⊢′ (e1, xs) ↔ ([], xs, 0, None);
⋀xs. P, e2, n, h ⊢′ (e2, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, e1·compareAndSwap(D·F, e2, e3), n, h ⊢′ (Val v·compareAndSwap(D·F, Val v′, e3′), xs) ↔ (stk @ [v′, v], loc, length (compE2 e1) + length (compE2 e2) + pc, xcp)

| bisim1CASThrow1′:
⟦ P, e1, n, h ⊢′ (Throw ad, xs) ↔ (stk, loc, pc, ⌊ad⌋);
⋀xs. P, e2, n, h ⊢′ (e2, xs) ↔ ([], xs, 0, None);
⋀xs. P, e3, n, h ⊢′ (e3, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, e1·compareAndSwap(D·F, e2, e3), n, h ⊢′ (Throw ad, xs) ↔ (stk, loc, pc, ⌊ad⌋)

| bisim1CASThrow2′:
⟦ P, e2, n, h ⊢′ (Throw ad, xs) ↔ (stk, loc, pc, ⌊ad⌋);
⋀xs. P, e1, n, h ⊢′ (e1, xs) ↔ ([], xs, 0, None);
⋀xs. P, e3, n, h ⊢′ (e3, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, e1·compareAndSwap(D·F, e2, e3), n, h ⊢′ (Throw ad, xs) ↔ (stk @ [v], loc, length (compE2 e1) + pc, ⌊ad⌋)

| bisim1CASThrow3′:
⟦ P, e3, n, h ⊢′ (Throw ad, xs) ↔ (stk, loc, pc, ⌊ad⌋);
⋀xs. P, e1, n, h ⊢′ (e1, xs) ↔ ([], xs, 0, None);
⋀xs. P, e2, n, h ⊢′ (e2, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, e1·compareAndSwap(D·F, e2, e3), n, h ⊢′ (Throw ad, xs) ↔ (stk @ [v′, v], loc, length (compE2 e1) + length (compE2 e2) + pc, ⌊ad⌋)

| bisim1CASFail′:
⟦ ⋀xs. P, e1, n, h ⊢′ (e1, xs) ↔ ([], xs, 0, None);
⋀xs. P, e2, n, h ⊢′ (e2, xs) ↔ ([], xs, 0, None);
⋀xs. P, e3, n, h ⊢′ (e3, xs) ↔ ([], xs, 0, None) ⟧
⟹ P, e1·compareAndSwap(D·F, e2, e3), n, h ⊢′ (Throw ad, xs) ↔ ([v′, v, v″], xs, length (compE2 e1) + length (compE2 e2) + length (compE2 e3), ⌊ad⌋)

| bisim1Call1′:
⟦ P, obj, n, h ⊢′ (obj′, xs) ↔ (stk, loc, pc, xcp);
⋀xs. P, ps, n, h ⊢′ (ps, xs) [↔] ([], xs, 0, None) ⟧
⟹ P, obj·M(ps), n, h ⊢′ (obj′·M(ps), xs) ↔ (stk, loc, pc, xcp)

| bisim1CallParams′:
⟦ P, ps, n, h ⊢′ (ps′, xs) [↔] (stk, loc, pc, xcp); ps ≠ [];
⋀xs. P, obj, n, h ⊢′ (obj, xs) ↔ ([], xs, 0, None) ⟧

$\Longrightarrow$ *P, obj•M(ps), n, h* $\vdash'$ (*Val v•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,* ⌊*a*⌋);
⋀*xs. P, ps, n, h* $\vdash'$ (*ps, xs*) [$\leftrightarrow$] ([], *xs, 0, None*)⟧
$\Longrightarrow$ *P, obj•M(ps), n, h* $\vdash'$ (*Throw a, xs*) $\leftrightarrow$ (*stk, loc, pc,* ⌊*a*⌋)

| *bisim1CallThrowParams′*:
⟦ *P, ps, n, h* $\vdash'$ (*map Val vs* @ *Throw a* # *ps′, xs*) [$\leftrightarrow$] (*stk, loc, pc,* ⌊*a*⌋);
⋀*xs. P, obj, n, h* $\vdash'$ (*obj, xs*) $\leftrightarrow$ ([], *xs, 0, None*) ⟧
$\Longrightarrow$ *P, obj•M(ps), n, h* $\vdash'$ (*Throw a, xs*) $\leftrightarrow$ (*stk* @ [*v*], *loc*, *length* (*compE2 obj*) + *pc*, ⌊*a*⌋)

| *bisim1CallThrow′*:
⟦ *length ps* = *length vs*;
⋀*xs. P, obj, n, h* $\vdash'$ (*obj, xs*) $\leftrightarrow$ ([], *xs, 0, None*); ⋀*xs. P, ps, n, h* $\vdash'$ (*ps, xs*) [$\leftrightarrow$] ([], *xs, 0, None*)
⟧
$\Longrightarrow$ *P, obj•M(ps), n, h* $\vdash'$ (*Throw a, xs*) $\leftrightarrow$ (*vs* @ [*v*], *xs*, *length* (*compE2 obj*) + *length* (*compEs2 ps*), ⌊*a*⌋)

| *bisim1BlockSome1′*:
(⋀*xs. P, e, Suc n, h* $\vdash'$ (*e, xs*) $\leftrightarrow$ ([], *xs, 0, None*))
$\Longrightarrow$ *P*, {*V*:*T*=⌊*v*⌋; *e*}, *n, h* $\vdash'$ ({*V*:*T*=⌊*v*⌋; *e*}, *xs*) $\leftrightarrow$ ([], *xs, 0, None*)

| *bisim1BlockSome2′*:
(⋀*xs. P, e, Suc n, h* $\vdash'$ (*e, xs*) $\leftrightarrow$ ([], *xs, 0, None*))
$\Longrightarrow$ *P*, {*V*:*T*=⌊*v*⌋; *e*}, *n, h* $\vdash'$ ({*V*:*T*=⌊*v*⌋; *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*=⌊*v*⌋; *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,* ⌊*a*⌋)
$\Longrightarrow$ *P*, {*V*:*T*=⌊*v*⌋; *e*}, *n, h* $\vdash'$ (*Throw a, xs*) $\leftrightarrow$ (*stk, loc, Suc* (*Suc pc*), ⌊*a*⌋)

| *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,* ⌊*a*⌋)
$\Longrightarrow$ *P*, {*V*:*T*=*None*; *e*}, *n, h* $\vdash'$ (*Throw a, xs*) $\leftrightarrow$ (*stk, loc, pc,* ⌊*a*⌋)

| *bisim1Sync1′*:
⟦ *P, e1, n, h* $\vdash'$ (*e′, xs*) $\leftrightarrow$ (*stk, loc, pc, xcp*);
⋀*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′*:
⟦ ⋀*xs. P, e1, n, h* $\vdash'$ (*e1, xs*) $\leftrightarrow$ ([], *xs, 0, None*);
⋀*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'*:
$\llbracket \bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e2, Suc\ n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None) \rrbracket$
$\Longrightarrow P, sync_V\ (e1)\ e2, n, h \vdash' (sync_V\ (Val\ v)\ e2, xs) \leftrightarrow ([v], xs[V := v], Suc\ (Suc\ (length\ (compE2\ e1))), None)$

| *bisim1Sync4'*:
$\llbracket P, e2, Suc\ n, h \vdash' (e', xs) \leftrightarrow (stk, loc, pc, xcp);$
$\bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None) \rrbracket$
$\Longrightarrow P, sync_V\ (e1)\ e2, n, h \vdash' (insync_V\ (a)\ e', xs) \leftrightarrow (stk, loc, Suc\ (Suc\ (Suc\ (length\ (compE2\ e1) + pc))), xcp)$

| *bisim1Sync5'*:
$\llbracket \bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e2, Suc\ n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None) \rrbracket$
$\Longrightarrow P, sync_V\ (e1)\ e2, n, h \vdash' (insync_V\ (a)\ Val\ v, xs) \leftrightarrow ([xs\ !\ V, v], xs, 4 + length\ (compE2\ e1) + length\ (compE2\ e2), None)$

| *bisim1Sync6'*:
$\llbracket \bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e2, Suc\ n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None) \rrbracket$
$\Longrightarrow P, sync_V\ (e1)\ e2, n, h \vdash' (Val\ v, xs) \leftrightarrow ([v], xs, 5 + length\ (compE2\ e1) + length\ (compE2\ e2), None)$

| *bisim1Sync7'*:
$\llbracket \bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e2, Suc\ n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None) \rrbracket$
$\Longrightarrow P, sync_V\ (e1)\ e2, n, h \vdash' (insync_V\ (a)\ Throw\ a', xs) \leftrightarrow ([Addr\ a'], xs, 6 + length\ (compE2\ e1) + length\ (compE2\ e2), None)$

| *bisim1Sync8'*:
$\llbracket \bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e2, Suc\ n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None) \rrbracket$
$\Longrightarrow P, sync_V\ (e1)\ e2, n, h \vdash' (insync_V\ (a)\ Throw\ a', xs) \leftrightarrow$
$([xs\ !\ V, Addr\ a'], xs, 7 + length\ (compE2\ e1) + length\ (compE2\ e2), None)$

| *bisim1Sync9'*:
$\llbracket \bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e2, Suc\ n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None) \rrbracket$
$\Longrightarrow P, sync_V\ (e1)\ e2, n, h \vdash' (Throw\ a, xs) \leftrightarrow ([Addr\ a], xs, 8 + length\ (compE2\ e1) + length\ (compE2\ e2), None)$

| *bisim1Sync10'*:
$\llbracket \bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e2, Suc\ n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None) \rrbracket$
$\Longrightarrow P, sync_V\ (e1)\ e2, n, h \vdash' (Throw\ a, xs) \leftrightarrow ([Addr\ a], xs, 8 + length\ (compE2\ e1) + length\ (compE2\ e2), \lfloor a \rfloor)$

| *bisim1Sync11'*:
$\llbracket \bigwedge xs.\ P, e1, n, h \vdash' (e1, xs) \leftrightarrow ([], xs, 0, None);$
$\bigwedge xs.\ P, e2, Suc\ n, h \vdash' (e2, xs) \leftrightarrow ([], xs, 0, None) \rrbracket$
$\Longrightarrow P, sync_V\ (e1)\ e2, n, h \vdash' (THROW\ NullPointer, xs) \leftrightarrow ([Null], xs, Suc\ (Suc\ (length\ (compE2\ e1))), \lfloor addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer \rfloor)$

| *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*) ⟧
⟹ *P*, *e1*;;*e2*, *n*, *h* ⊢' (*e'*;;*e2*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1SeqThrow1'*:
⟦ *P*, *e1*, *n*, *h* ⊢' (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋);
⋀*xs*. *P*, *e2*, *n*, *h* ⊢' (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*) ⟧
⟹ *P*, *e1*;;*e2*, *n*, *h* ⊢' (*Throw a*, *xs*) ↔ (*stk*, *loc*, *pc*, ⌊*a*⌋)

| *bisim1Seq2'*:
⟦ *P*, *e2*, *n*, *h* ⊢' (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*);
⋀*xs*. *P*, *e1*, *n*, *h* ⊢' (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*) ⟧
⟹ *P*, *e1*;;*e2*, *n*, *h* ⊢' (*e'*, *xs*) ↔ (*stk*, *loc*, *Suc* (*length* (*compE2 e1*) + *pc*), *xcp*)

| *bisim1Cond1'*:
⟦ *P*, *e*, *n*, *h* ⊢' (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*);
⋀*xs*. *P*, *e1*, *n*, *h* ⊢' (*e1*, *xs*) ↔ ([], *xs*, *0*, *None*);
⋀*xs*. *P*, *e2*, *n*, *h* ⊢' (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*) ⟧
⟹ *P*, *if* (*e*) *e1 else e2*, *n*, *h* ⊢' (*if* (*e'*) *e1 else e2*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1CondThen'*:
⟦ *P*, *e1*, *n*, *h* ⊢' (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*);
⋀*xs*. *P*, *e*, *n*, *h* ⊢' (*e*, *xs*) ↔ ([], *xs*, *0*, *None*);
⋀*xs*. *P*, *e2*, *n*, *h* ⊢' (*e2*, *xs*) ↔ ([], *xs*, *0*, *None*) ⟧
⟹ *P*, *if* (*e*) *e1 else e2*, *n*, *h* ⊢' (*e'*, *xs*) ↔ (*stk*, *loc*, *Suc* (*length* (*compE2 e*) + *pc*), *xcp*)

| *bisim1CondElse'*:
⟦ *P*, *e2*, *n*, *h* ⊢' (*e'*, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*);
⋀*xs*. *P*, *e*, *n*, *h* ⊢' (*e*, *xs*) ↔ ([], *xs*, *0*, *None*);
⋀*xs*. *P*, *e1*, *n*, *h* ⊢' (*e1*, *xs*) ↔ ([], *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'*:
  $[\![$ $\bigwedge$*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, Suc (Suc (length (compE2 c) + length (compE2 e))), None)*

| *bisim1While7'*:
  $[\![$ $\bigwedge$*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 $\vdash'$ *(unit, xs)* $\leftrightarrow$ *([], xs, Suc (Suc (Suc (length (compE2 c) + length (compE2 e)))), None)*

| *bisim1WhileThrow1'*:
  $[\![$ *P, c, 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, while (c) e, n, h* $\vdash'$ *(Throw a, xs)* $\leftrightarrow$ *(stk, loc, pc,* $\lfloor a \rfloor$*)*

| *bisim1WhileThrow2'*:
  $[\![$ *P, e, n, h* $\vdash'$ *(Throw a, xs)* $\leftrightarrow$ *(stk, loc, pc,* $\lfloor a \rfloor$*)*;
    $\bigwedge$*xs. P, c, n, h* $\vdash'$ *(c, xs)* $\leftrightarrow$ *([], xs, 0, None)* $]\!]$
  $\Longrightarrow$ *P, while (c) e, n, h* $\vdash'$ *(Throw a, xs)* $\leftrightarrow$ *(stk, loc, Suc (length (compE2 c) + pc),* $\lfloor a \rfloor$*)*

| *bisim1Throw1'*:
  P, e, n, h $\vdash'$ *(e', xs)* $\leftrightarrow$ *(stk, loc, pc, xcp)*
  $\Longrightarrow$ *P, throw e, n, h* $\vdash'$ *(throw e', xs)* $\leftrightarrow$ *(stk, loc, pc, xcp)*

| *bisim1Throw2'*:
  ($\bigwedge$*xs. P, e, n, h* $\vdash'$ *(e, xs)* $\leftrightarrow$ *([], xs, 0, None)*)

$\Longrightarrow$ *P*, *throw e*, *n*, *h* $\vdash'$ (*Throw a*, *xs*) $\leftrightarrow$ ([*Addr a*], *xs*, *length* (*compE2 e*), $\lfloor a \rfloor$)

| *bisim1ThrowNull'*:
($\bigwedge$*xs*. *P*, *e*, *n*, *h* $\vdash'$ (*e*, *xs*) $\leftrightarrow$ ([], *xs*, *0*, *None*))
$\Longrightarrow$ *P*, *throw e*, *n*, *h* $\vdash'$ (*THROW NullPointer*, *xs*) $\leftrightarrow$ ([*Null*], *xs*, *length* (*compE2 e*), $\lfloor$*addr-of-sys-xcpt NullPointer*$\rfloor$)

| *bisim1ThrowThrow'*:
P, *e*, *n*, *h* $\vdash'$ (*Throw a*, *xs*) $\leftrightarrow$ (*stk*, *loc*, *pc*, $\lfloor a \rfloor$)
$\Longrightarrow$ *P*, *throw e*, *n*, *h* $\vdash'$ (*Throw a*, *xs*) $\leftrightarrow$ (*stk*, *loc*, *pc*, $\lfloor a \rfloor$)

| *bisim1Try'*:
$[\![$ *P*, *e*, *n*, *h* $\vdash'$ (*e'*, *xs*) $\leftrightarrow$ (*stk*, *loc*, *pc*, *xcp*);
$\bigwedge$*xs*. *P*, *e2*, *Suc n*, *h* $\vdash'$ (*e2*, *xs*) $\leftrightarrow$ ([], *xs*, *0*, *None*) $]\!]$
$\Longrightarrow$ *P*, *try e catch*(*C V*) *e2*, *n*, *h* $\vdash'$ (*try e' catch*(*C V*) *e2*, *xs*) $\leftrightarrow$ (*stk*, *loc*, *pc*, *xcp*)

| *bisim1TryCatch1'*:
$[\![$ *P*, *e*, *n*, *h* $\vdash'$ (*Throw a*, *xs*) $\leftrightarrow$ (*stk*, *loc*, *pc*, $\lfloor a \rfloor$); *typeof-addr h a* = $\lfloor$*Class-type C'*$\rfloor$; *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* $\vdash'$ ({*V*:*Class C*=*None*; *e2*}, *xs*[*V* := *Addr a*]) $\leftrightarrow$ ([*Addr a*], *loc*, *Suc* (*length* (*compE2 e*)), *None*)

| *bisim1TryCatch2'*:
$[\![$ *P*, *e2*, *Suc 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*, *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-addr h a* = $\lfloor$*Class-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*: $P,es,n,h \vdash' (es,xs)$ [↔] $([],xs,0,None)$
⟨*proof*⟩

lemma *bisim1-imp-bisim1′*: $P,\ e,\ h \vdash exs \leftrightarrow s \Longrightarrow P,\ e,\ n,\ h \vdash' exs \leftrightarrow s$
and *bisims1-imp-bisims1′*: $P,\ es,\ h \vdash esxs$ [↔] $s \Longrightarrow P,\ es,\ n,\ h \vdash' esxs$ [↔] s
⟨*proof*⟩

lemma *bisim1′-imp-bisim1*: $P,\ e,\ n,\ h \vdash' exs \leftrightarrow s \Longrightarrow P,\ e,\ h \vdash exs \leftrightarrow s$
and *bisims1′-imp-bisims1*: $P,\ es,\ n,\ h \vdash' esxs$ [↔] $s \Longrightarrow P,\ es,\ h \vdash esxs$ [↔] s
⟨*proof*⟩

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*
⟨*proof*⟩

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*: $P,Es,h \vdash (es, xs)\ [\leftrightarrow]\ (stk, loc, pc, xcp) \Longrightarrow pc \leq length\ (compEs2\ Es)$
⟨*proof*⟩

lemma *bisim1-pc-length-compE2D*:
$P,e,h \vdash (e', xs) \leftrightarrow (stk,loc,length\ (compE2\ e),xcp)$
$\Longrightarrow xcp = None \land call1\ e' = None \land (\exists v.\ stk = [v] \land (is\text{-}val\ e' \longrightarrow e' = Val\ v \land xs = loc))$

and *bisims1-pc-length-compEs2D*:
$P,es,h \vdash (es', xs)\ [\leftrightarrow]\ (stk,loc,length\ (compEs2\ es),xcp)$
$\Longrightarrow xcp = None \land calls1\ es' = None \land (\exists vs.\ stk = rev\ vs \land length\ vs = length\ es \land (is\text{-}vals\ es' \longrightarrow es' = map\ Val\ vs \land xs = loc))$
⟨*proof*⟩

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*⟩

lemma *bisim1-length-xs*: $P,e,h \vdash (e',xs) \leftrightarrow (stk, loc, pc, xcp) \Longrightarrow length\ xs = length\ loc$
and *bisims1-length-xs*: $P,es,h \vdash (es',xs)\ [\leftrightarrow]\ (stk, loc, pc, xcp) \Longrightarrow length\ xs = length\ loc$
⟨*proof*⟩

lemma *bisim1-Val-length-compE2D*:
$P,e,h \vdash (Val\ v,xs) \leftrightarrow (stk, loc, length\ (compE2\ e), xcp) \Longrightarrow stk = [v] \land xs = loc \land xcp = None$

and *bisims1-Val-length-compEs2D*:
$P,es,h \vdash (map\ Val\ vs,xs)\ [\leftrightarrow]\ (stk, loc, length\ (compEs2\ es), xcp) \Longrightarrow stk = rev\ vs \land xs = loc \land xcp = None$
⟨*proof*⟩

lemma *bisim-Val-loc-eq-xcp-None*:
$P,\ e,\ h \vdash (Val\ v,\ xs) \leftrightarrow (stk, loc, pc, xcp) \Longrightarrow xs = loc \land xcp = None$

and *bisims-Val-loc-eq-xcp-None*:
$P,\ es,\ h \vdash (map\ Val\ vs,\ xs)\ [\leftrightarrow]\ (stk, loc, pc, xcp) \Longrightarrow xs = loc \land xcp = None$
⟨*proof*⟩

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*:
$[\![\ P,es,h \vdash (map\ Val\ vs,xs)\ [\leftrightarrow]\ (stk,loc,pc,xcp);\ pc < length\ (compEs2\ es)\ ]\!] \Longrightarrow compEs2\ es\ !\ pc \neq Invoke\ M\ n'$
⟨*proof*⟩

lemma *bisim1-VarD*: $P,\ E,\ h \vdash (Var\ V,xs) \leftrightarrow (stk,loc,pc,xcp) \Longrightarrow xs = loc$
and $P,\ es,\ h \vdash (es',\ xs)\ [\leftrightarrow]\ (stk, loc, pc, xcp) \Longrightarrow True$
⟨*proof*⟩

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*:
$P,\ es,\ h \vdash (map\ Val\ vs\ @\ Throw\ a\ \#\ es',\ xs)\ [\leftrightarrow]\ (stk,\ loc,\ pc,\ xcp)$
$\Longrightarrow pc < length\ (compEs2\ es) \wedge (xcp = \lfloor a \rfloor \vee xcp = None) \wedge xs = loc$
⟨*proof*⟩

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*⟩

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*: $P,es,h \vdash (es',xs)\ [\leftrightarrow]\ (stk,loc,pc,\lfloor a \rfloor) \Longrightarrow calls1\ es' = None$
⟨*proof*⟩

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*⟩

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*⟩

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*:
$P,\ e,\ h \vdash (Throw\ a,\ xs) \leftrightarrow (stk,\ loc,\ pc,\ \lfloor a \rfloor)$
$\Longrightarrow match\text{-}ex\text{-}table\ (compP\ f\ P)\ (cname\text{-}of\ h\ a)\ (pc' + pc)\ (compxE2\ e\ pc'\ d) = None$

and *bisims1-xcp-Some-not-caught*:
$P,\ es,\ h \vdash (map\ Val\ vs\ @\ Throw\ a\ \#\ es',\ xs)\ [\leftrightarrow]\ (stk,\ loc,\ pc,\ \lfloor a \rfloor)$
$\Longrightarrow match\text{-}ex\text{-}table\ (compP\ f\ P)\ (cname\text{-}of\ h\ a)\ (pc' + pc)\ (compxEs2\ es\ pc'\ d) = None$
⟨*proof*⟩

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*: $P,e,h \vdash (e', xs) \leftrightarrow (stk, loc, pc, \lfloor a \rfloor) \Longrightarrow pc < length\ (compE2\ e)$
and *bisims1-xcp-pcD*: $P,es,h \vdash (es', xs)\ [\leftrightarrow]\ (stk, loc, pc, \lfloor a \rfloor) \Longrightarrow pc < length\ (compEs2\ es)$
⟨*proof*⟩

declare *nth-append* [*simp*]

lemma *bisim1-Val-τExec-move*:
$[\![P, E, h \vdash (Val\ v, xs) \leftrightarrow (stk, loc, pc, xcp);\ pc < length\ (compE2\ E)]\!]$
$\Longrightarrow xs = loc \wedge xcp = None \wedge$
$\tau Exec\text{-}mover\text{-}a\ P\ t\ E\ h\ (stk, xs, pc, None)\ ([v], xs, length\ (compE2\ E), None)$

and *bisims1-Val-τExec-moves*:
$[\![P, Es, h \vdash (map\ Val\ vs, xs)\ [\leftrightarrow]\ (stk, loc, pc, xcp);\ pc < length\ (compEs2\ Es)]\!]$
$\Longrightarrow xs = loc \wedge xcp = None \wedge$
$\tau Exec\text{-}movesr\text{-}a\ P\ t\ Es\ h\ (stk, xs, pc, None)\ (rev\ vs, xs, length\ (compEs2\ Es), None)$
⟨*proof*⟩

lemma *bisim1Val2D1*:
assumes *bisim*: $P, e, h \vdash (Val\ v,xs) \leftrightarrow (stk,loc,pc,xcp)$
shows $xcp = None \wedge xs = loc \wedge \tau Exec\text{-}mover\text{-}a\ P\ t\ e\ h\ (stk, loc, pc, xcp)\ ([v], loc, length\ (compE2\ e), None)$
⟨*proof*⟩

lemma *bisim1-Throw-τExec-movet*:
$[\![P, e, h \vdash (Throw\ a,xs) \leftrightarrow (stk,loc,pc,None)]\!]$
$\Longrightarrow \exists pc'.\ \tau Exec\text{-}movet\text{-}a\ P\ t\ e\ h\ (stk, loc, pc, None)\ ([Addr\ a], loc, pc', \lfloor a \rfloor) \wedge$
$P, e, h \vdash (Throw\ a,xs) \leftrightarrow ([Addr\ a], loc, pc', \lfloor a \rfloor) \wedge xs = loc$

and *bisims1-Throw-τExec-movest*:
$[\![P, es, h \vdash (map\ Val\ vs\ @\ Throw\ a\ \#\ es',xs)\ [\leftrightarrow]\ (stk,loc,pc,None)]\!]$
$\Longrightarrow \exists pc'.\ \tau Exec\text{-}movest\text{-}a\ P\ t\ es\ h\ (stk, loc, pc, None)\ (Addr\ a\ \#\ rev\ vs, loc, pc', \lfloor a \rfloor) \wedge$
$P, es, h \vdash (map\ Val\ vs\ @\ Throw\ a\ \#\ es',xs)\ [\leftrightarrow]\ (Addr\ a\ \#\ rev\ vs, loc, pc', \lfloor a \rfloor) \wedge xs = loc$
⟨*proof*⟩

lemma *bisim1-Throw-τExec-mover*:
$[\![P, e, h \vdash (Throw\ a,xs) \leftrightarrow (stk,loc,pc,None)]\!]$
$\Longrightarrow \exists pc'.\ \tau Exec\text{-}mover\text{-}a\ P\ t\ e\ h\ (stk, loc, pc, None)\ ([Addr\ a], loc, pc', \lfloor a \rfloor) \wedge$
$P, e, h \vdash (Throw\ a,xs) \leftrightarrow ([Addr\ a], loc, pc', \lfloor a \rfloor) \wedge xs = loc$
⟨*proof*⟩

lemma *bisims1-Throw-τExec-movesr*:
$[\![P, es, h \vdash (map\ Val\ vs\ @\ Throw\ a\ \#\ es',xs)\ [\leftrightarrow]\ (stk,loc,pc,None)]\!]$
$\Longrightarrow \exists pc'.\ \tau Exec\text{-}movesr\text{-}a\ P\ t\ es\ h\ (stk, loc, pc, None)\ (Addr\ a\ \#\ rev\ vs, loc, pc', \lfloor a \rfloor) \wedge$
$P, es, h \vdash (map\ Val\ vs\ @\ Throw\ a\ \#\ es',xs)\ [\leftrightarrow]\ (Addr\ a\ \#\ rev\ vs, loc, pc', \lfloor a \rfloor) \wedge xs = loc$
⟨*proof*⟩

declare *split-beta* [*simp*]

lemma *bisim1-inline-call-Throw*:
$[\![P,e,h \vdash (e', xs) \leftrightarrow (stk, loc, pc, None);\ call1\ e' = \lfloor (a, M, vs) \rfloor;$
$compE2\ e\ !\ pc = Invoke\ M\ n0;\ pc < length\ (compE2\ e)]\!]$
$\Longrightarrow n0 = length\ vs \wedge P,e,h \vdash (inline\text{-}call\ (Throw\ A)\ e', xs) \leftrightarrow (stk, loc, pc, \lfloor A \rfloor)$
(**is** $[\![\text{-};\ \text{-};\ \text{-};\ \text{-}]\!] \Longrightarrow ?concl\ e\ n\ e'\ xs\ pc\ stk\ loc$)

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⟩

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
⟨proof⟩

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
⟨proof⟩

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
⟨proof⟩

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⟩

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*⟩

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*
⟨*proof*⟩

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′*
⟨*proof*⟩

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*⟩

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′*) ⟧
⟹ ∃ *stk″*. *stk′* = *stk″* @ *STK* ∧ *exec-meth-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′* ⟧ ⟹ *?concl E stk STK loc pc xcp stk′ loc′ pc′ xcp′*)

and *exec-meth-stk-splits*:
⟦ *P*,*Es*,*h* ⊢ (*es*,*xs*) [↔] (*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′*) ⟧
⟹ ∃ *stk″*. *stk′* = *stk″* @ *STK* ∧ *exec-meth-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′* ⟧ ⟹ *?concls Es stk STK loc pc xcp stk′ loc′ pc′ xcp′*)
⟨*proof*⟩

lemma shows *bisim1-callD*:

$\llbracket$ *P*,*e*,*h* ⊢ (*e*′, *xs*) ↔ (*stk*, *loc*, *pc*, *xcp*); *call1 e*′ = ⌊(*a*, *M*, *vs*)⌋;
compE2 e ! *pc* = *Invoke M*′ *n0* $\rrbracket$
⟹ *M* = *M*′

and *bisims1-callD*:
$\llbracket$ *P*,*es*,*h* ⊢ (*es*′,*xs*) [↔] (*stk*,*loc*,*pc*, *xcp*); *calls1 es*′ = ⌊(*a*, *M*, *vs*)⌋;
compEs2 es ! *pc* = *Invoke M*′ *n0* $\rrbracket$
⟹ *M* = *M*′
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bisim1-match-Some-stk-length*:
$\llbracket$ *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*)⌋ $\rrbracket$
⟹ *d* ≤ *length stk*

and *bisims1-match-Some-stk-length*:
$\llbracket$ *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*)⌋ $\rrbracket$
⟹ *d* ≤ *length stk*
⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

end

sublocale *J1-JVM-conf-read* < *heap-conf?*: *J1-JVM-heap-conf*
⟨*proof*⟩

sublocale *J1-JVM-conf-read* < *heap?*: *J1-heap*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *sim12-sizes-append* [*simp*]:
sim12-sizes (*es* @ *es'*) = *sim12-sizes es* + *sim12-sizes es'*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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)
⟨*proof*⟩

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*: $rec\text{-}option = case\text{-}option$
⟨*proof*⟩

context *J1-JVM-heap-base* **begin**

lemma assumes *ha*: $typeof\text{-}addr\ h\ a = \lfloor Class\text{-}type\ D \rfloor$
and *subclsObj*: $P \vdash D \preceq^* Throwable$
shows *bisim1-xcp-τRed*:
$[\![P,e,h \vdash (e', xs) \leftrightarrow (stk, loc, pc, \lfloor a \rfloor);$
$match\text{-}ex\text{-}table\ (compP\ f\ P)\ (cname\text{-}of\ h\ a)\ pc\ (compxE2\ e\ 0\ 0) = None;$
$\exists n.\ n + max\text{-}vars\ e' \leq length\ xs \wedge \mathcal{B}\ e\ n\ ]\!]$
$\Longrightarrow \tau red1r\ P\ t\ h\ (e', xs)\ (Throw\ a, loc) \wedge P,e,h \vdash (Throw\ a, loc) \leftrightarrow (stk, loc, pc, \lfloor a \rfloor)$

and *bisims1-xcp-τReds*:
$[\![P,es,h \vdash (es', xs)\ [\leftrightarrow]\ (stk, loc, pc, \lfloor a \rfloor);$
$match\text{-}ex\text{-}table\ (compP\ f\ P)\ (cname\text{-}of\ h\ a)\ pc\ (compxEs2\ es\ 0\ 0) = None;$
$\exists n.\ n + max\text{-}varss\ es' \leq length\ xs \wedge \mathcal{B}s\ es\ n\ ]\!]$
$\Longrightarrow \exists vs\ es''.\ \tau reds1r\ P\ t\ h\ (es', xs)\ (map\ Val\ vs\ @\ Throw\ a\ \#\ es'', loc) \wedge$
$P,es,h \vdash (map\ Val\ vs\ @\ Throw\ a\ \#\ es'', loc)\ [\leftrightarrow]\ (stk, loc, pc, \lfloor a \rfloor)$
⟨*proof*⟩

primrec *conf-xcp'* :: *'m prog ⇒ 'heap ⇒ 'addr option ⇒ bool* **where**
$conf\text{-}xcp'\ P\ h\ None = True$
$|\ conf\text{-}xcp'\ P\ h\ \lfloor a \rfloor = (\exists D.\ typeof\text{-}addr\ h\ a = \lfloor Class\text{-}type\ D \rfloor \wedge P \vdash D \preceq^* Throwable)$

lemma *conf-xcp-conf-xcp'*:
$conf\text{-}xcp\ P\ h\ xcp\ i \Longrightarrow conf\text{-}xcp'\ P\ h\ xcp$
⟨*proof*⟩

lemma *conf-xcp'-compP* [*simp*]: $conf\text{-}xcp'\ (compP\ f\ P) = conf\text{-}xcp'\ P$
⟨*proof*⟩

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*: $P,h \vdash t\ \surd t$
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 ∃ *ta′ e′ xs′. τred1r P t h* (*e*, *xs*) (*e′*, *xs′*) ∧ *True*,*P*,*t* ⊢*1* ⟨*e′*, (*h*, *xs′*)⟩ −*ta′*→ ⟨*e″*, (*h′*, *xs″*)⟩ ∧ *ta-bisim wbisim1* (*extTA2J1 P ta′*) *ta* ∧ ¬ *τmove1 P h e′* ∧ (*call1 e* = *None* ∨ *no-call2 E pc* ∨ *e′* = *e* ∧ *xs′* = *xs*))
(**is** ⟦ -; *?exec E stk loc pc xcp stk′ loc′ pc′ xcp′*; -; -; -; - ⟧
⟹ ∃ *e″ xs″*. *P*, *E*, *h′* ⊢ (*e″*, *xs″*) ↔ (*stk′*, *loc′*, *pc′*, *xcp′*) ∧ *?red e xs e″ xs″ E stk pc pc′ xcp xcp′*)

and *reds1-simulates-exec-instr*:
⟦ *P*, *Es*, *h* ⊢ (*es*, *xs*) [↔] (*stk*, *loc*, *pc*, *xcp*);
exec-moves-d P t Es h (*stk*, *loc*, *pc*, *xcp*) *ta h′* (*stk′*, *loc′*, *pc′*, *xcp′*);
n + *max-varss es* ≤ *length xs*; *bsoks Es n*; *P*,*h* ⊢ *stk* [:≤] *ST*; *conf-xcp′* (*compP2 P*) *h xcp* ⟧
⟹ ∃ *es″ xs″*. *P*, *Es*, *h′* ⊢ (*es″*, *xs″*) [↔] (*stk′*, *loc′*, *pc′*, *xcp′*) ∧
(*if τmoves2* (*compP2 P*) *h stk Es pc xcp*
then h′ = *h* ∧ (*if xcp′* = *None* ⟶ *pc* < *pc′ then τreds1r else τreds1t*) *P t h* (*es*, *xs*) (*es″*, *xs″*)
else ∃ *ta′ es′ xs′*. *τreds1r P t h* (*es*, *xs*) (*es′*, *xs′*) ∧ *True*,*P*,*t* ⊢*1* ⟨*es′*, (*h*, *xs′*)⟩ [−*ta′*→] ⟨*es″*, (*h′*, *xs″*)⟩ ∧ *ta-bisim wbisim1* (*extTA2J1 P ta′*) *ta* ∧ ¬ *τmoves1 P h es′* ∧ (*calls1 es* = *None* ∨ *no-calls2 Es pc* ∨ *es′* = *es* ∧ *xs′* = *xs*))
(**is** ⟦ -; *?execs Es stk loc pc xcp stk′ loc′ pc′ xcp′*; -; -; -; - ⟧
⟹ ∃ *es″ xs″*. *P*, *Es*, *h′* ⊢ (*es″*, *xs″*) [↔] (*stk′*, *loc′*, *pc′*, *xcp′*) ∧ *?reds es xs es″ xs″ Es stk pc pc′ xcp xcp′*)
⟨*proof*⟩

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*⟩

lemma *Collect-split-mem*: {(*x*, *y*). (*x*, *y*) ∈ *Q*} = *Q* ⟨*proof*⟩

lemma *wfP-sim21-size*: *wfP* (*sim21-size P*)
⟨*proof*⟩

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*⟩

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*: *τMove2 (compP2 P) (xcp, h, frs)*
shows *h = h′ ∧ (∃ e′ xs′ exs′. (if sim21-size (compP2 P) (xcp′, frs′) (xcp, frs) then τRed1r else τRed1t) P t h ((e, xs), exs) ((e′, xs′), exs′) ∧ bisim1-list1 t h (e′, xs′) exs′ xcp′ frs′)*
⟨*proof*⟩

lemma *τRed1-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*: ¬ *τMove2 (compP2 P) (xcp, h, frs)*
shows ∃ *e′ xs′ exs′ ta′ e″ xs″ exs″. τRed1r P t h ((e, xs), exs) ((e′, xs′), exs′) ∧*
True,P,t ⊢1 ⟨(e′, xs′)/exs′, h⟩ −ta′→ ⟨(e″, xs″)/exs″, h′⟩ ∧
¬ *τMove1 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)

⟨*proof*⟩

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*) $\Longrightarrow$ *P* $\vdash$ *C* *has M*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

context *JVM-heap-base* **begin**

lemma *τexec-1-d-silent-move*:
τexec-1-d P t (*xcp*, *h*, *frs*) (*xcp'*, *h'*, *frs'*)
$\Longrightarrow$ *τtrsys.silent-move* (*mexecd P t*) (*τMOVE2 P*) ((*xcp*, *frs*), *h*) ((*xcp'*, *frs'*), *h'*)
⟨*proof*⟩

lemma *silent-move-τexec-1-d*:
τtrsys.silent-move (*mexecd P t*) (*τMOVE2 P*) ((*xcp*, *frs*), *h*) ((*xcp'*, *frs'*), *h'*)
$\Longrightarrow$ *τexec-1-d P t* (*xcp*, *h*, *frs*) (*xcp'*, *h'*, *frs'*)
⟨*proof*⟩

lemma *τExec-1-dr-rtranclpD*:
τExec-1-dr P t (*xcp*, *h*, *frs*) (*xcp'*, *h'*, *frs'*)
$\Longrightarrow$ *τtrsys.silent-moves* (*mexecd P t*) (*τMOVE2 P*) ((*xcp*, *frs*), *h*) ((*xcp'*, *frs'*), *h'*)
⟨*proof*⟩

lemma *τExec-1-dt-tranclpD*:
τExec-1-dt P t (*xcp*, *h*, *frs*) (*xcp'*, *h'*, *frs'*)
$\Longrightarrow$ *τtrsys.silent-movet* (*mexecd P t*) (*τMOVE2 P*) ((*xcp*, *frs*), *h*) ((*xcp'*, *frs'*), *h'*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

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*⟩

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*) (*τMOVE1 P*) (*τMOVE2* (*compP2 P*))
⟨*proof*⟩

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*)
⟨*proof*⟩

sublocale *JVM-heap-base* < *execd-mthr*:
τmultithreaded
JVM-final
mexecd P
convert-RA
τMOVE2 P
for *P*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

lemmas τ*red1-Val-simps* = τ*red1r-Val* τ*red1t-Val* τ*reds1r-map-Val* τ*reds1t-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*⟨*proof*⟩

lemma [*simp*]: (*index* (*xs* @ [*x*]) *y* = *size xs*) = (*x* = *y*)⟨*proof*⟩

lemma [*simp*]: *x* ∈ *set xs* ⟹ *xs* ! *index xs x* = *x*⟨*proof*⟩

lemma [*simp*]: *x* ∉ *set xs* ⟹ *index xs x* = *size xs*⟨*proof*⟩

lemma *index-size-conv*[*simp*]: (*index xs x* = *size xs*) = (*x* ∉ *set xs*)⟨*proof*⟩

lemma *size-index-conv*[*simp*]: (*size xs* = *index xs x*) = (*x* ∉ *set xs*)⟨*proof*⟩

lemma (*index xs x* < *size xs*) = (*x* ∈ *set xs*)⟨*proof*⟩

lemma [*simp*]: ⟦ *y* ∈ *set xs*; *x* ≠ *y* ⟧ ⟹ *index* (*xs* @ [*x*]) *y* = *index xs y*⟨*proof*⟩

lemma *index-less-size*[*simp*]: *x* ∈ *set xs* ⟹ *index xs x* < *size xs*⟨*proof*⟩
lemma *index-less-aux*: ⟦*x* ∈ *set xs*; *size xs* ≤ *n*⟧ ⟹ *index xs x* < *n*⟨*proof*⟩

lemma [*simp*]: *x* ∈ *set xs* ∨ *y* ∈ *set xs* ⟹ (*index xs x* = *index xs y*) = (*x* = *y*)⟨*proof*⟩

lemma *inj-on-index*: *inj-on* (*index xs*) (*set xs*)⟨*proof*⟩

lemma *index-drop*: ⋀*x i*. ⟦ *x* ∈ *set xs*; *index xs x* < *i* ⟧ ⟹ *x* ∉ *set*(*drop i xs*)⟨*proof*⟩

7.20.2 *hidden*

lemma *hidden-index*: $x \in set\ xs \Longrightarrow hidden\ (xs\ @\ [x])\ (index\ xs\ x)\langle proof\rangle$

lemma *hidden-inacc*: $hidden\ xs\ i \Longrightarrow index\ xs\ x \neq i\langle proof\rangle$

lemma [*simp*]: $hidden\ xs\ i \Longrightarrow hidden\ (xs@[x])\ i\langle proof\rangle$

lemma *fun-upds-apply*: $\bigwedge m\ ys.$
$(m(xs[\mapsto]ys))\ x =$
$(let\ xs' = take\ (size\ ys)\ xs$
$in\ if\ x \in set\ xs'\ then\ Some(ys\ !\ index\ xs'\ x)\ else\ m\ x)\langle proof\rangle$

lemma *map-upds-apply-eq-Some*:
$((m(xs[\mapsto]ys))\ x = Some\ y) =$
$(let\ xs' = take\ (size\ ys)\ xs$
$in\ if\ x \in set\ xs'\ then\ ys\ !\ index\ xs'\ x = y\ else\ m\ x = Some\ y)\langle proof\rangle$

lemma *map-upds-upd-conv-index*:
$[\![x \in set\ xs;\ size\ xs \leq size\ ys\ ]\!]$
$\Longrightarrow m(xs[\mapsto]ys,\ x\mapsto y) = m(xs[\mapsto]ys[index\ xs\ x := y])\langle proof\rangle$
lemma *image-index*:
$A \subseteq set(xs@[x]) \Longrightarrow index\ (xs\ @\ [x])\ `\ A =$
$(if\ x \in A\ then\ insert\ (size\ xs)\ (index\ xs\ `\ (A-\{x\}))\ else\ index\ xs\ `\ A)\langle proof\rangle$
lemma *index-le-lengthD*: $index\ xs\ x < length\ xs \Longrightarrow x \in set\ xs$
$\langle proof\rangle$

lemma *not-hidden-index-nth*: $[\![\ i < length\ Vs;\ \neg\ hidden\ Vs\ i\ ]\!] \Longrightarrow index\ Vs\ (Vs\ !\ i) = i$
$\langle proof\rangle$

lemma *hidden-snoc-nth*:
assumes *len*: $i < length\ Vs$
shows $hidden\ (Vs\ @\ [Vs\ !\ i])\ i$
$\langle proof\rangle$

lemma *map-upds-Some-eq-nth-index*:
assumes $[Vs\ [\mapsto]\ vs]\ V = Some\ v\ length\ Vs \leq length\ vs$
shows $vs\ !\ index\ Vs\ V = v$
$\langle proof\rangle$

end

7.21 Compilation Stage 1

theory *Compiler1* **imports**
PCompiler
J1State
ListIndex
begin

definition *fresh-var* :: $vname\ list \Rightarrow vname$
where $fresh\text{-}var\ Vs = sum\text{-}list\ (STR\ ''V''\ \#\ Vs)$

lemma *fresh-var-fresh*: $fresh\text{-}var\ Vs \notin set\ Vs$

⟨*proof*⟩

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*
⟨*proof*⟩
termination
⟨*proof*⟩

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*
⟨*proof*⟩

lemmas *compEs1-map-Val = compEs1-conv-map*

lemma *compE1-eq-Val* [*simp*]: *compE1 Vs e = Val v* ⟷ *e = Val v*
⟨*proof*⟩

lemma *Val-eq-compE1* [*simp*]: *Val v = compE1 Vs e* ⟷ *e = Val v*
⟨*proof*⟩

lemma *compEs1-eq-map-Val* [*simp*]: *compEs1 Vs es = map Val vs* ⟷ *es = map Val vs*

⟨*proof*⟩

lemma *compE1-eq-Var* [*simp*]: *compE1 Vs e* = *Var V* ⟷ (∃ *V'*. *e* = *Var V'* ∧ *V* = *index Vs V'*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *length-compEs2* [*simp*]:
length (*compEs1 Vs es*) = *length es*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma fixes *e* :: *'addr expr* **and** *es* :: *'addr expr list*
shows $\mathcal{B}$: *size Vs* = *n* ⟹ $\mathcal{B}$ (*compE1 Vs e*) *n*
and $\mathcal{B}$*s*: *size Vs* = *n* ⟹ $\mathcal{B}$*s* (*compEs1 Vs es*) *n*
⟨*proof*⟩

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*⟩

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*⟩

lemma (**in** *heap-base*) *synthesized-call-compP* [*simp*]:
synthesized-call (*compP f P*) *h aMvs* = *synthesized-call P h aMvs*
⟨*proof*⟩

primrec *fin1* :: *'addr expr* ⇒ *'addr expr1*
where
fin1 (*Val v*) = *Val v*
| *fin1* (*throw e*) = *throw* (*fin1 e*)

lemma *comp-final*: *final e* $\Longrightarrow$ *compE1 Vs e* = *fin1 e*
⟨*proof*⟩

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*
⟨*proof*⟩

Compiling programs:

definition *compP1* :: *'addr J-prog* ⇒ *'addr J1-prog*
where
  compP1 ≡ *compP* (λ*C 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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *unmod-extRet2J* [*simp*]: *unmod e i* ⟹ *unmod* (*extRet2J e va*) *i*
⟨*proof*⟩

lemma *max-dest*: (*n* :: *nat*) + *max a b* ≤ *c* ⟹ *n* + *a* ≤ *c* ∧ *n* + *b* ≤ *c*
⟨*proof*⟩

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*⟩

lemma *hidden-lengthD*: *hidden Vs i* ⟹ *i* < *length Vs*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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* = {}; *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 (λt. 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)
⟨proof⟩

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*
⟨proof⟩

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*
⟨proof⟩

declare *bisim-list1.simps* [*simp del*]

lemma *bisims-map-Val-conv* [*simp*]: *bisims Vs (map Val vs) es xs = (es = map Val vs)*
⟨proof⟩

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'*
⟨proof⟩

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*
⟨proof⟩

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*⟩

lemma *bisim-hidden-unmod*: ⟦ *bisim Vs e e′ xs*; *hidden Vs i* ⟧ $\Longrightarrow$ *unmod e′ i*
and *bisims-hidden-unmods*: ⟦ *bisims Vs es es′ xs*; *hidden Vs i* ⟧ $\Longrightarrow$ *unmods es′ i*
⟨*proof*⟩

lemma *bisim-fv-unmod*: ⟦ *bisim Vs e e′ xs*; *i* < *length Vs*; *Vs* ! *i* ∉ *fv e* ⟧ $\Longrightarrow$ *unmod e′ i*
and *bisims-fvs-unmods*: ⟦ *bisims Vs es es′ xs*; *i* < *length Vs*; *Vs* ! *i* ∉ *fvs es* ⟧ $\Longrightarrow$ *unmods es′ i*
⟨*proof*⟩

lemma *bisim-extRet2J* [*intro*!]: *bisim Vs e e′ xs* $\Longrightarrow$ *bisim Vs* (*extRet2J e va*) (*extRet2J1 e′ va*) *xs*
⟨*proof*⟩

lemma *bisims-map-Val-conv2* [*simp*]: *bisims Vs es* (*map Val vs*) *xs* = (*es* = *map Val vs*)
⟨*proof*⟩

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″*)
⟨*proof*⟩

lemma *hidden-bisim-unmod*: ⟦ *bisim Vs e e′ xs*; *hidden Vs i* ⟧ $\Longrightarrow$ *unmod e′ i*
and *hidden-bisims-unmods*: ⟦ *bisims Vs es es′ xs*; *hidden Vs i* ⟧ $\Longrightarrow$ *unmods es′ i*
⟨*proof*⟩

lemma *bisim-list-list-all2-conv*:
bisim-list es exs′ ⟷ *list-all2 bisim01 es exs′*
⟨*proof*⟩

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*⟩

lemma *bisim-max-vars*: *bisim Vs e e′ xs* $\Longrightarrow$ *max-vars e* = *max-vars e′*
and *bisims-max-varss*: *bisims Vs es es′ xs* $\Longrightarrow$ *max-varss es* = *max-varss es′*
⟨*proof*⟩

lemma *bisim-call*: *bisim Vs e e′ xs* $\Longrightarrow$ *call e* = *call e′*
and *bisims-calls*: *bisims Vs es es′ xs* $\Longrightarrow$ *calls es* = *calls es′*
⟨*proof*⟩

lemma *bisim-call-None-call1*: ⟦ *bisim Vs e e′ xs*; *call e* = *None* ⟧ $\Longrightarrow$ *call1 e′* = *None*
and *bisims-calls-None-calls1*: ⟦ *bisims Vs es es′ xs*; *calls es* = *None* ⟧ $\Longrightarrow$ *calls1 es′* = *None*
⟨*proof*⟩

lemma *bisim-call1-Some-call*:
⟦ *bisim Vs e e′ xs*; *call1 e′* = ⌊*aMvs*⌋ ⟧ $\Longrightarrow$ *call e* = ⌊*aMvs*⌋

and *bisims-calls1-Some-calls*:
⟦ *bisims Vs es es′ xs*; *calls1 es′* = ⌊*aMvs*⌋ ⟧ $\Longrightarrow$ *calls es* = ⌊*aMvs*⌋
⟨*proof*⟩

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*
⟨*proof*⟩

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′*
⟨*proof*⟩

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*⟩

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*⟩

declare *hyperUn-ac* [*simp del*]

lemma *sqInt-lem3*: ⟦ *A* ⊑ *A′*; *B* ⊑ *B′* ⟧ ⟹ *A* ⊓ *B* ⊑ *A′* ⊓ *B′*
⟨*proof*⟩

lemma *sqUn-lem3*: ⟦ *A* ⊑ *A′*; *B* ⊑ *B′* ⟧ ⟹ *A* ⊔ *B* ⊑ *A′* ⊔ *B′*
⟨*proof*⟩

lemma *A-inline-call*: *call e* = ⌊*aMvs*⌋ ⟹ 𝒜 *e* ⊑ 𝒜 (*inline-call e′ e*)
and *As-inline-calls*: *calls es* = ⌊*aMvs*⌋ ⟹ 𝒜*s es* ⊑ 𝒜*s* (*inline-calls e′ es*)
⟨*proof*⟩

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*⌋; 𝒟*s es A* ⟧ ⟹ 𝒟*s* (*inline-calls e′ es*) *A*
⟨*proof*⟩

lemma *bisim-B*: *bisim Vs e E xs* ⟹ ℬ *E* (*length Vs*)
and *bisims-Bs*: *bisims Vs es Es xs* ⟹ ℬ*s Es* (*length Vs*)
⟨*proof*⟩

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* ⟹ *expr-lockss es* = *expr-lockss es′*
⟨*proof*⟩

lemma *bisim-list-expr-lockss-eq*: *bisim-list es exs′* ⟹ *expr-lockss es* = *expr-lockss* (*map fst exs′*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *τMove1-compP* [*simp*]: *τMove1* (*compP f P*) = *τMove1 P*
⟨*proof*⟩

lemma *red1-preserves-unmod*:
⟦ *uf*,*P*,*t* ⊢*1* ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩; *unmod e i* ⟧ ⟹ (*lcl s′*) ! *i* = (*lcl s*) ! *i*

and *reds1-preserves-unmod*:
⟦ *uf*,*P*,*t* ⊢*1* ⟨*es*, *s*⟩ [−*ta*→] ⟨*es′*, *s′*⟩; *unmods es i* ⟧ ⟹ (*lcl s′*) ! *i* = (*lcl s*) ! *i*
⟨*proof*⟩

lemma *red1-unmod-preserved*:
⟦ *uf*,*P*,*t* ⊢*1* ⟨*e*, *s*⟩ −*ta*→ ⟨*e′*, *s′*⟩; *unmod e i* ⟧ ⟹ *unmod e′ i*
and *reds1-unmods-preserved*:
⟦ *uf*,*P*,*t* ⊢*1* ⟨*es*, *s*⟩ [−*ta*→] ⟨*es′*, *s′*⟩; *unmods es i* ⟧ ⟹ *unmods es′ i*
⟨*proof*⟩

lemma *τred1t-unmod-preserved*:
⟦ *τred1gt uf P t h* (*e*, *xs*) (*e′*, *xs′*); *unmod e i* ⟧ ⟹ *unmod e′ i*
⟨*proof*⟩

lemma *τred1r-unmod-preserved*:
⟦ *τred1gr uf P t h* (*e*, *xs*) (*e′*, *xs′*); *unmod e i* ⟧ ⟹ *unmod e′ i*
⟨*proof*⟩

lemma *τred1t-preserves-unmod*:
⟦*τred1gt uf P t h* (*e*, *xs*) (*e′*, *xs′*); *unmod e i*; *i* < *length xs* ⟧
⟹ *xs′* ! *i* = *xs* ! *i*
⟨*proof*⟩

lemma *τred1′r-preserves-unmod*:
⟦*τred1gr uf P t h* (*e*, *xs*) (*e′*, *xs′*); *unmod e i*; *i* < *length xs* ⟧
⟹ *xs′* ! *i* = *xs* ! *i*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *τMove0-compP* [*simp*]: *τMove0* (*compP f P*) = *τMove0 P*
⟨*proof*⟩

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*; *𝓑s es n* ⟧ ⟹ *els-loc-ok es xs*
⟨*proof*⟩

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*
⟨*proof*⟩

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′*
⟨*proof*⟩

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*: ⟦ *𝓑s es n*; *V* < *n* ⟧ ⟹ *els-loc-ok es* (*xs*[*V* := *v*]) = *els-loc-ok es xs*
⟨*proof*⟩

lemma *els-loc-ok-map-Val* [*simp*]:
els-loc-ok (*map Val vs*) *xs*
⟨*proof*⟩

lemma *els-loc-ok-map-Val-append* [*simp*]:
els-loc-ok (*map Val vs* @ *es*) *xs* = *els-loc-ok es xs*
⟨*proof*⟩

lemma *el-loc-ok-extRet2J* [*simp*]:
el-loc-ok e xs ⟹ *el-loc-ok* (*extRet2J e va*) *xs*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *el-loc-ok-blocks1* [*simp*]:
el-loc-ok (*blocks1 n Ts body*) *xs* = *el-loc-ok body xs*
⟨*proof*⟩

lemma *sync-oks-blocks1* [*simp*]: *sync-ok* (*blocks1 n Ts e*) = *sync-ok e*
⟨*proof*⟩

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*

⟨*proof*⟩

lemma assumes *sync-ok e'*
shows *sync-ok-inline-call*: *sync-ok e* $\Longrightarrow$ *sync-ok* (*inline-call e' e*)
and *sync-oks-inline-calls*: *sync-oks es* $\Longrightarrow$ *sync-oks* (*inline-calls e' es*)
⟨*proof*⟩

lemma *bisim-sync-ok*:
bisim Vs e e' xs $\Longrightarrow$ *sync-ok e*
bisim Vs e e' xs $\Longrightarrow$ *sync-ok e'*

and *bisims-sync-oks*:
bisims Vs es es' xs $\Longrightarrow$ *sync-oks es*
bisims Vs es es' xs $\Longrightarrow$ *sync-oks es'*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *lock-oks1I*:
⟦ ⋀*t l*. *ts t* = *None* $\Longrightarrow$ *has-locks* (*ls* $ *l*) *t* = *0*;
⋀*t e x exs ln l*. *ts t* = ⌊(((*e*, *x*), *exs*), *ln*)⌋ $\Longrightarrow$ *has-locks* (*ls* $ *l*) *t* + *ln* $ *l*= *expr-locks e l* + *expr-lockss* (*map fst exs*) *l* ⟧
$\Longrightarrow$ *lock-oks1 ls ts*
⟨*proof*⟩

lemma *lock-oks1E*:
⟦ *lock-oks1 ls ts*;
∀ *t*. *ts t* = *None* ⟶ (∀ *l*. *has-locks* (*ls* $ *l*) *t* = *0*) $\Longrightarrow$ *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*) $\Longrightarrow$ *Q* ⟧
$\Longrightarrow$ *Q*
⟨*proof*⟩

lemma *lock-oks1D1*:
⟦ *lock-oks1 ls ts*; *ts t* = *None* ⟧ $\Longrightarrow$ ∀ *l*. *has-locks* (*ls* $ *l*) *t* = *0*
⟨*proof*⟩

lemma *lock-oks1D2*:
⋀*ln*. ⟦ *lock-oks1 ls ts*; *ts t* = ⌊(((*e*, *x*), *exs*), *ln*)⌋ ⟧
$\Longrightarrow$ ∀ *l*. *has-locks* (*ls* $ *l*) *t* + *ln* $ *l* = *expr-locks e l* + *expr-lockss* (*map fst exs*) *l*
⟨*proof*⟩

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* ⟧
$\Longrightarrow$ *lock-oks1 ls* (*ts*(*t* ↦ (((*e'*, *xs'*), *exs'*), *ln*)))
⟨*proof*⟩

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*
⟨*proof*⟩

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*) ⟧
⟹ *False*,*P*,*t* ⊢*1* ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*', *s*'⟩ ∨ (∃ *l*. *ta* = ⦃*UnlockFail*→*l*⦄ ∧ *expr-lockss es l* > *0*)
⟨*proof*⟩

lemma *Red1-True-into-Red1-False*:
assumes *True*,*P*,*t* ⊢*1* ⟨*ex*/*exs*,*shr s*⟩ −*ta*→ ⟨*ex*'/*exs*',*m*'⟩
and *el-loc-ok1* (*ex*, *exs*)
shows *False*,*P*,*t* ⊢*1* ⟨*ex*/*exs*,*shr s*⟩ −*ta*→ ⟨*ex*'/*exs*',*m*'⟩ ∨
(∃ *l*. *ta* = ⦃*UnlockFail*→*l*⦄ ∧ *expr-lockss* (*fst ex* # *map fst exs*) *l* > *0*)
⟨*proof*⟩

lemma shows *red1-preserves-el-loc-ok*:
⟦ *uf*,*P*,*t* ⊢*1* ⟨*e*, *s*⟩ −*ta*→ ⟨*e*', *s*'⟩; *sync-ok e*; *el-loc-ok e* (*lcl s*) ⟧ ⟹ *el-loc-ok e*' (*lcl s*')

and *reds1-preserves-els-loc-ok*:
⟦ *uf*,*P*,*t* ⊢*1* ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*', *s*'⟩; *sync-oks es*; *els-loc-ok es* (*lcl s*) ⟧ ⟹ *els-loc-ok es*' (*lcl s*')
⟨*proof*⟩

lemma *red1-preserves-sync-ok*: ⟦ *uf*,*P*,*t* ⊢*1* ⟨*e*, *s*⟩ −*ta*→ ⟨*e*', *s*'⟩; *sync-ok e* ⟧ ⟹ *sync-ok e*'
and *reds1-preserves-sync-oks*: ⟦ *uf*,*P*,*t* ⊢*1* ⟨*es*, *s*⟩ [−*ta*→] ⟨*es*', *s*'⟩; *sync-oks es* ⟧ ⟹ *sync-oks es*'
⟨*proof*⟩

lemma *Red1-preserves-el-loc-ok1*:
assumes *wf*: *wf-J1-prog P*
shows ⟦ *uf*,*P*,*t* ⊢*1* ⟨*ex*/*exs*,*m*⟩ −*ta*→ ⟨*ex*'/*exs*',*m*'⟩; *el-loc-ok1* (*ex*, *exs*) ⟧ ⟹ *el-loc-ok1* (*ex*', *exs*')
⟨*proof*⟩

lemma assumes *wf*: *wf-J1-prog P*
shows *red1-el-loc-ok1-new-thread*:
⟦ *uf*,*P*,*t* ⊢*1* ⟨*e*, *s*⟩ −*ta*→ ⟨*e*', *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*), [])

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*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *Red1-mthr-eq-Red1-mthr*′:
assumes *lok*: *lock-oks1* (*locks s*) (*thr s*)
and *elo*: *ts-ok* (λ*t exexs h*. *el-loc-ok1 exexs*) (*thr s*) (*shr s*)
shows *Red1-mthr*.*redT True P s* = *Red1-mthr*.*redT False P s*
⟨*proof*⟩

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*⟩

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*⟩

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* (λ*t exexs h. el-loc-ok1 exexs*) (*thr s1*) (*shr s1*)
shows *lock-oks1* (*locks s1'*) (*thr s1'*)
⟨*proof*⟩

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*⟩

lemma *Red1'-Red1-bisimulation-final*:
wf-J1-prog P
$\Longrightarrow$ *bisimulation-final* (*Red1-mthr.redT False P*) (*Red1-mthr.redT True P*)
mbisim-Red1'-Red1 (=) *Red1-mthr.mfinal Red1-mthr.mfinal*
⟨*proof*⟩

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*⟩

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.mτmove P*) (*Red1-mthr.mτmove P*)
⟨*proof*⟩

end

sublocale *J1-heap-base* < *Red1-mthr*:
if-τmultithreaded-wf
final-expr1
mred1g uf P
convert-RA
τMOVE1 P
for *uf P*
⟨*proof*⟩

context *J1-heap-base* **begin**

abbreviation *if-lock-oks1* ::
(*'addr,'thread-id*) *locks*
$\Rightarrow$ (*'addr,'thread-id,*(*status* × ((*'a,'b,'addr*) *exp* × *'c*) × ((*'a,'b,'addr*) *exp* × *'c*) *list*)) *thread-info*
$\Rightarrow$ *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*)
⟨*proof*⟩

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′*
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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.m*τ*move P*) (*Red1-mthr.if.m*τ*move P*)
⟨*proof*⟩

lemma *if-Red1′-Red1-bisimulation-final*:
wf-J1-prog P
$\Longrightarrow$ *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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⌋ ⟧ $\Longrightarrow$ $\mathcal{D}$ *e′* ⌊*dom xs′*⌋
⟨*proof*⟩

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*⌋ ⟧ $\Longrightarrow$ $\mathcal{D}$ *e′* ⌊*dom xs′*⌋
⟨*proof*⟩

end

lemma *LAss-lem*:
⟦*x* ∈ *set xs*; *size xs* ≤ *size ys* ⟧
$\Longrightarrow$ *m1* $\subseteq_m$ *m2*(*xs*[↦]*ys*) $\Longrightarrow$ *m1*(*x*↦*y*) $\subseteq_m$ *m2*(*xs*[↦]*ys*[*index xs x* := *y*])
⟨*proof*⟩

lemma *Block-lem*:
fixes *l* :: ′*a* ⇀ ′*b*
assumes *0*: *l* $\subseteq_m$ [*Vs* [↦] *ls*]
and *1*: *l′* $\subseteq_m$ [*Vs* [↦] *ls′*, *V*↦*v*]
and *hidden*: *V* ∈ *set Vs* $\Longrightarrow$ *ls* ! *index Vs V* = *ls′* ! *index Vs V*
and *size*: *size ls* = *size ls′*   *size Vs* < *size ls′*
shows *l′*(*V* := *l V*) $\subseteq_m$ [*Vs* [↦] *ls′*]
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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'*
⟨*proof*⟩

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' $\Longrightarrow$ *sim-moves01 P t ta0 (Val v # es0) (Val v # es) h xs ta (Val v # es') h' xs'*
⟨*proof*⟩

lemma *sim-move01-CallParams*:
sim-moves01 P t ta0 es0 es h xs ta es' h' xs'
$\Longrightarrow$ *sim-move01 P t ta0 (Val v•M(es0)) (Val v•M(es)) h xs ta (Val v•M(es')) h' xs'*
⟨*proof*⟩

lemma *sim-move01-reds*:
⟦ *(h', a)* ∈ *allocate h (Class-type C); ta0 =* {|*NewHeapElem a (Class-type C)*|}*; ta =* {|*NewHeapElem a (Class-type C)*|} ⟧
$\Longrightarrow$ *sim-move01 P t ta0 (new C) (new C) h xs ta (addr a) h' xs*
allocate h (Class-type C) = {} $\Longrightarrow$ *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)))*|} ⟧
$\Longrightarrow$ *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 $\Longrightarrow$ *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* ⟧
$\Longrightarrow$ *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* ⟧
$\Longrightarrow$ *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* ⟧
$\Longrightarrow$ *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* ⟧
$\Longrightarrow$ *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) $\Longrightarrow$ *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) $\Longrightarrow$ *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* ⟧ $\Longrightarrow$ *sim-move01 P t* ε *(Var V') (Var V) h xs* ε *(Val v) h xs*
V < length xs $\Longrightarrow$ *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* ⟧
$\Longrightarrow$ *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*|} ⟧
$\Longrightarrow$ *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 = ⌊Array-type T n⌋; 0 <=s i; sint i < int n;* $typeof_h$ *v = ⌊U⌋;* ¬ *(P ⊢ U ≤ 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 = ⌊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′;
ta0 = ⦃WriteMem a (ACell (nat (sint i))) v⦄; ta = ⦃WriteMem a (ACell (nat (sint i))) v ⦄ ⟧
⟹ *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 = ⌊Array-type T n⌋ ⟹ *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⦄* ⟧
⟹ *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⦄* ⟧
⟹ *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′⦄ ⟧
⟹ *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″ ≠ v;*
ta0 = ⦃ReadMem a (CField D F) v″⦄; ta = ⦃ReadMem a (CField D F) v″⦄ ⟧
⟹ *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 ⟹ *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 = ⌊Class-type D⌋; P ⊢ D ⪯* C; V < length xs* ⟧
⟹ *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⌊Throw a⌉) (newA T⌊Throw a⌉) 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⌊e0⌉) (Throw a⌊e1⌉) h xs ε (Throw a) h xs
sim-move01 P t ε (Val v⌊Throw a⌉) (Val v⌊Throw a⌉) h xs ε (Throw a) h xs
sim-move01 P t ε (Throw a⌊e0⌉ := e0′) (Throw a⌊e1⌉ := e1′) h xs ε (Throw a) h xs
sim-move01 P t ε (Val v⌊Throw a⌉ := e0) (Val v⌊Throw a⌉ := 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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*])
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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 $\subseteq_m$ [*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′* $\subseteq_m$ [*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 $\subseteq_m$ [*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′* $\subseteq_m$ [*vs* [↦] *xs′*]
(**is** ⟦ -; -; -; -; -; *?synths es1 S* ⟧ ⟹ *?concls es1 es2 S XS es1′ S′ TA vs*)
⟨*proof*⟩

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′′. τreds0r (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′′ ∧ ¬ τmoves0 P h es′′)
else extTA2J0 P,P,t ⊢ ⟨es, (h, xs)⟩ [−ta→] ⟨es′, (h′, xs′)⟩ ∧ no-calls P h es ∧ ¬ τmoves0 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′
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemmas *sim-move10-intros* =
sim-move10-expr sim-move10-reds sim-move10-CallNull sim-move10-TryFail sim-move10-Block sim-move10-CallParams

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'*⌋
⟨*proof*⟩

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);
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);
𝒟s 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*⟩

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*⟩

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′*
⟨*proof*⟩

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*
⟨*proof*⟩

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″*)
⟨*proof*⟩

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*⟩

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′*)
⟨*proof*⟩

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*
⟨*proof*⟩

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* (*λt. bisim-red0-Red1*)) (*τMOVE0 P*) (*τMOVE1* (*compP1 P*)) (*λes es′. False*) (*λ*(((*e′*, *xs′*), *exs′*), *h′*) (((*e*, *xs*), *exs*), *h*). *countInitBlock e′*< *countInitBlock e*)
(**is** *delay-bisimulation-measure* - - - - - - *?μ1 ?μ2*)
⟨*proof*⟩

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* ($\lambda t.$ *bisim-red0-Red1*)) (τ*MOVE0 P*) (τ*MOVE1* (*compP1 P*))
⟨*proof*⟩

lemma *red0-Red1′-FWweak-bisim*:
assumes *wf*: *wf-J-prog P*
shows *FWdelay-bisimulation-diverge final-expr0* (*mred0 P*) *final-expr1* (*mred1′* (*compP1 P*))
($\lambda t.$ *bisim-red0-Red1*) *bisim-wait01* (τ*MOVE0 P*) (τ*MOVE1* (*compP1 P*))
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

lemma fixes *e* :: (*'a*, *'b*, *'addr*) *exp* **and** *es* :: (*'a*, *'b*, *'addr*) *exp list*
  shows *D-None* [*iff*]: $\mathcal{D}$ *e None*
  and *Ds-None* [*iff*]: $\mathcal{D}$*s es None*
⟨*proof*⟩

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* ⟧ ⟹ $\mathcal{D}$ *e* ⌊*A*⌋ ⟹ $\mathcal{D}$ (*compE1 Vs e*) ⌊*index Vs* ‘ *A*⌋
  and *Ds-index-compEs1*: ⟦ *A* ⊆ *set Vs*; *fvs es* ⊆ *set Vs* ⟧ ⟹ $\mathcal{D}$*s es* ⌊*A*⌋ ⟹ $\mathcal{D}$*s* (*compEs1 Vs es*) ⌊*index Vs* ‘ *A*⌋
⟨*proof*⟩

declare *Un-ac* [*simp del*]

lemma *index-image-set*: *distinct xs* ⟹ *index xs* ‘ *set xs* = {..<*size xs*}
⟨*proof*⟩

lemma *D-compE1*:
  ⟦ $\mathcal{D}$ *e* ⌊*set Vs*⌋; *fv e* ⊆ *set Vs*; *distinct Vs* ⟧ ⟹ $\mathcal{D}$ (*compE1 Vs e*) ⌊{..<*length Vs*}⌋
⟨*proof*⟩

lemma *D-compE1'*:
  assumes $\mathcal{D}$ *e* ⌊*set*(*V#Vs*)⌋ **and** *fv e* ⊆ *set*(*V#Vs*) **and** *distinct*(*V#Vs*)
  shows $\mathcal{D}$ (*compE1* (*V#Vs*) *e*) ⌊{..*length Vs*}⌋
⟨*proof*⟩

lemma *compP1-pres-wf*: *wf-J-prog P* ⟹ *wf-J1-prog* (*compP1 P*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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* $\circ_B$ *red0-Red1'.mbisim* $\circ_B$ *mbisim-Red1'-Red1* $\circ_B$ *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* $\circ_B$ *red0-Red1'.mta-bisim* $\circ_B$ (=) $\circ_B$ *Red1-execd.mta-bisim*

end

lemma *compP2-has-method* [*simp*]: *compP2 P* ⊢ *C has M* ⟷ *P* ⊢ *C has M*
⟨*proof*⟩

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* ⟨*proof*⟩

theorem *bisimJ2JVM-weak-bisim*:
assumes *wf*: *wf-J-prog P*
shows *delay-bisimulation-diverge-final* (*mredT P*) (*execd-mthr.redT* (*J2JVM P*)) *bisimJ2JVM tl-simJ2JVM*
(*red-mthr.mτmove P*) (*execd-mthr.mτmove* (*J2JVM P*)) *red-mthr.mfinal exec-mthr.mfinal*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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.τRuns P s ξ*
obtains ξ'
where *execd-mthr.mthr.τRuns* (*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*⟩

theorem *J2JVM-correct2*:
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 *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 $\bigwedge cs'$. $[\![$ *tfinite* ξ'; *terminal* $\xi' = \lfloor cs' \rfloor$; *exec-mthr.mfinal* cs' $]\!]$
$\Longrightarrow \exists\, s'$. *tfinite* $\xi \wedge$ *terminal* $\xi = \lfloor s' \rfloor \wedge cs' =$ *mexception* $s' \wedge$ *bisimJ2JVM* s' cs'
and $\bigwedge cs'$. $[\![$ *tfinite* ξ'; *terminal* $\xi' = \lfloor cs' \rfloor$; *execd-mthr.deadlock* (*J2JVM P*) cs' $]\!]$
$\Longrightarrow \exists\, s'$. *tfinite* $\xi \wedge$ *terminal* $\xi = \lfloor s' \rfloor \wedge$ *red-mthr.deadlock P* $s' \wedge$ *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*⟩

end

declare *compP1-def* [*simp*]

theorem *wt-J2JVM*: *wf-J-prog P* $\Longrightarrow$ *wf-jvm-prog* (*J2JVM P*)
⟨*proof*⟩

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*⟩

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′*
⟨*proof*⟩

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*⟩

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
⟨*proof*⟩
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*
⟨*proof*⟩

lemma *new-Addr-SomeI*:
finite (*dom h*) ⟹ ∃ *a*. *new-Addr h* = *Some a*
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

definition *init-fields* :: $('field\text{-}name \times (ty \times fmod))$ *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*
⟨*proof*⟩

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* ⟨*proof*⟩

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 *sc-start-heap-ok*: *sc-start-heap-ok P*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *sc-heap*:
heap addr2thread-id thread-id2addr (*sc-allocate P*) *sc-typeof-addr sc-heap-write P*
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *sc-hext-new*:
h a = *None* ⟹ *h* ⊴*sc h*(*a* ↦ *arrobj*)
⟨*proof*⟩

lemma *sc-hext-upd-obj*: *h a* = *Some* (*Obj C fs*) ⟹ *h* ⊴*sc h*(*a*↦(*Obj C fs*′))
⟨*proof*⟩

lemma *sc-hext-upd-arr*: ⟦ *h a* = *Some* (*Arr T f e*); *length e* = *length e*′ ⟧ ⟹ *h* ⊴*sc h*(*a*↦(*Arr T f*′ *e*′))
⟨*proof*⟩

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* ⟨*proof*⟩

declare *sc.typeof-addr-thread-id2-addr-addr2thread-id* [*simp del*]

lemma *sc-conf-upd-obj*: $h\ a = Some(Obj\ C\ fs) \Longrightarrow (P,h(a \mapsto (Obj\ C\ fs')) \vdash sc\ x :\leq T) = (P,h \vdash sc\ x :\leq T)$
⟨*proof*⟩

lemma *sc-conf-upd-arr*: $h\ a = Some(Arr\ T\ f\ el) \Longrightarrow (P,h(a \mapsto (Arr\ T\ f'\ el')) \vdash sc\ x :\leq T') = (P,h \vdash sc\ x :\leq T')$
⟨*proof*⟩

lemma *sc-oconf-hext*: $P,h \vdash sc\ obj\ \surd \Longrightarrow h \trianglelefteq sc\ h' \Longrightarrow P,h' \vdash sc\ obj\ \surd$
⟨*proof*⟩

lemma *sc-oconf-init-fields*:
assumes $P \vdash C$ *has-fields FDTs*
shows $P,h \vdash sc$ (*Obj C* (*init-fields FDTs*)) $\surd$
⟨*proof*⟩

lemma *sc-oconf-init*:
is-htype P hT $\Longrightarrow P,h \vdash sc$ *blank P hT* $\surd$
⟨*proof*⟩

lemma *sc-oconf-fupd* [*intro?*]:
$[\![\ P \vdash C\ has\ F{:}T\ (fm)\ in\ D;\ P,h \vdash sc\ v :\leq T;\ P,h \vdash sc\ (Obj\ C\ fs)\ \surd\ ]\!]$
$\Longrightarrow P,h \vdash sc\ (Obj\ C\ (fs((F,D) \mapsto v)))\ \surd$
⟨*proof*⟩

lemma *sc-oconf-fupd-arr* [*intro?*]:
$[\![\ P,h \vdash sc\ v :\leq T;\ P,h \vdash sc\ (Arr\ T\ f\ el)\ \surd\ ]\!]$
$\Longrightarrow P,h \vdash sc\ (Arr\ T\ f\ (el[i := v]))\ \surd$
⟨*proof*⟩

lemma *sc-oconf-fupd-arr-fields*:
$[\![\ P \vdash Object\ has\ F{:}T\ (fm)\ in\ Object;\ P,h \vdash sc\ v :\leq T;\ P,h \vdash sc\ (Arr\ T'\ f\ el)\ \surd\ ]\!]$
$\Longrightarrow P,h \vdash sc\ (Arr\ T'\ (f((F,\ Object) \mapsto v))\ el)\ \surd$
⟨*proof*⟩

lemma *sc-oconf-new*: $[\![\ P,h \vdash sc\ obj\ \surd;\ h\ a = None\ ]\!] \Longrightarrow P,h(a \mapsto arrobj) \vdash sc\ obj\ \surd$
⟨*proof*⟩

lemmas *sc-oconf-upd-obj* = *sc-oconf-hext* [*OF - sc-hext-upd-obj*]

lemma *sc-oconf-upd-arr*:
  assumes $P,h \vdash sc\ obj\ \surd$
  and *ha*: $h\ a = \lfloor Arr\ T\ f\ el \rfloor$
  shows $P,h(a \mapsto Arr\ T\ f'\ el') \vdash sc\ obj\ \surd$
⟨*proof*⟩

lemma *sc-hconfD*: $[\![\ P \vdash sc\ h\ \surd;\ h\ a = Some\ obj\ ]\!] \Longrightarrow P,h \vdash sc\ obj\ \surd$
⟨*proof*⟩

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*: $[\![\ P \vdash sc\ h\ \surd;\ h\ a = None;\ P,h \vdash sc\ obj\ \surd\ ]\!] \Longrightarrow P \vdash sc\ h(a \mapsto obj)\ \surd$
⟨*proof*⟩

lemma *sc-hconf-upd-obj*: $[\![\ P \vdash sc\ h\ \surd;\ h\ a = Some\ (Obj\ C\ fs);\ P,h \vdash sc\ (Obj\ C\ fs')\ \surd\ ]\!] \Longrightarrow P \vdash sc\ h(a \mapsto (Obj\ C\ fs'))\ \surd$
⟨*proof*⟩

lemma *sc-hconf-upd-arr*: $[\![\ P \vdash sc\ h\ \surd;\ h\ a = Some(Arr\ T\ f\ el);\ P,h \vdash sc\ (Arr\ T\ f'\ el')\ \surd\ ]\!] \Longrightarrow P \vdash sc\ h(a \mapsto (Arr\ T\ f'\ el'))\ \surd$
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

8.1.4 Code generation

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
sc-heap-read ⟨*proof*⟩

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
sc-heap-write ⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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* ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *sc-start-heap-ok*: *sc-start-heap-ok P*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *sc-heap*:
heap addr2thread-id thread-id2addr (*sc-allocate P*) *sc-typeof-addr sc-heap-write P*
⟨*proof*⟩

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* ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *sc-hext-upd-obj*: *rm-lookup a h* = *Some* (*Obj C fs*) ⟹ *h* ⊴*sc rm-update a* (*Obj C fs*′) *h*
⟨*proof*⟩

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*
⟨*proof*⟩

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* $\Rightarrow$
is-type P $(T\lfloor\rceil) \wedge (\forall n.\ n < si \longrightarrow (\exists v.$ *rm-α el n = Some v* $\wedge$ *P,h* $\vdash$*sc v* $:\leq T)) \wedge$
$(\forall$ *F T fm. P* $\vdash$ *Object has F:T (fm) in Object* $\longrightarrow (\exists v.$ *lm-lookup F f = Some v* $\wedge$ *P,h* $\vdash$*sc v* $:\leq T)))$

definition *sc-hconf* :: *'m prog* $\Rightarrow$ *heap* $\Rightarrow$ *bool* (‹- $\vdash$*sc* - $\surd$› [51,51] 50)
where *P* $\vdash$*sc h* $\surd \longleftrightarrow (\forall$ *a obj. rm-α h a = Some obj* $\longrightarrow$ *P,h* $\vdash$*sc obj* $\surd)$

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*
⟨*proof*⟩

lemma *sc-conf-upd-obj*: *rm-lookup a h = Some(Obj C fs)* $\Longrightarrow$ (*P,rm-update a (Obj C fs') h* $\vdash$*sc x* $:\leq T) = ($*P,h* $\vdash$*sc x* $:\leq T)$
⟨*proof*⟩

lemma *sc-conf-upd-arr*:
rm-lookup a h = Some(Arr T si f el) $\Longrightarrow$ (*P,rm-update a (Arr T si f' el') h* $\vdash$*sc x* $:\leq T') = ($*P,h* $\vdash$*sc x* $:\leq T')$
⟨*proof*⟩

lemma *sc-oconf-hext*: *P,h* $\vdash$*sc obj* $\surd \Longrightarrow h \trianglelefteq$*sc h'* $\Longrightarrow$ *P,h'* $\vdash$*sc obj* $\surd$
⟨*proof*⟩

lemma *map-of-fields-init-fields*:
assumes *map-of FDTs (F, D)* $= \lfloor(T, fm)\rfloor$
shows $\exists$*fs' v. tm-α (init-fields (map (λ(FD, (T, fm)). (FD, T)) FDTs)) (String.explode F)* $= \lfloor fs'\rfloor$ $\wedge$ *lm-α fs' D* $= \lfloor v\rfloor \wedge$ *sc.conf P h v T*
⟨*proof*⟩

lemma *sc-oconf-init-fields*:
assumes *P* $\vdash$ *C has-fields FDTs*
shows *P,h* $\vdash$*sc (Obj C (init-fields (map (λ(FD, (T, fm)). (FD, T)) FDTs)))* $\surd$
⟨*proof*⟩

lemma *sc-oconf-init-arr*:
assumes *type*: *is-type P* $(T\lfloor\rceil)$
shows *P,h* $\vdash$*sc Arr T n (init-fields-array (map (λ((F, D), (T, fm)). (F, T)) (TypeRel.fields P Object))) (init-cells T n)* $\surd$
⟨*proof*⟩

lemma *sc-oconf-fupd* [*intro?*]:
⟦ *P* $\vdash$ *C has F:T (fm) in D*; *P,h* $\vdash$*sc v* $:\leq T$; *P,h* $\vdash$*sc (Obj C fs)* $\surd$;

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))* √
⟨*proof*⟩

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))* √
⟨*proof*⟩

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)* √
⟨*proof*⟩

lemma *sc-oconf-new*: ⟦ *P,h ⊢sc obj* √; *rm-lookup a h = None* ⟧ ⟹ *P,rm-update a arrobj h ⊢sc obj* √
⟨*proof*⟩

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* √
⟨*proof*⟩

lemma *sc-oconf-blank*: *is-htype P hT* ⟹ *P,h ⊢sc blank P hT* √
⟨*proof*⟩

lemma *sc-hconfD*: ⟦ *P ⊢sc h* √; *rm-lookup a h = Some obj* ⟧ ⟹ *P,h ⊢sc obj* √
⟨*proof*⟩

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* √
⟨*proof*⟩

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* √
⟨*proof*⟩

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* √
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

8.2.4 Code generation

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
sc-heap-read ⟨*proof*⟩

code-pred
(*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *bool*, *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
sc-heap-write ⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*: (⋀*a a'*. *r a a'* ⟹ *a* ∈ *A* ∧ *a'* ∈ *A*) ⟹ (⋀*a*. *a* : *A* ⟹ *r a a*) ⟹ *refl-onP A r*
⟨*proof*⟩

lemma *refl-onPD*: *refl-onP A r* ==> *a* : *A* ==> *r a a*
⟨*proof*⟩

lemma *refl-onPD1*: *refl-onP A r* ==> *r a b* ==> *a* : *A*
⟨*proof*⟩

lemma *refl-onPD2*: *refl-onP A r* ==> *r a b* ==> *b* : *A*
⟨*proof*⟩

lemma *refl-onP-Int*: *refl-onP A r* ==> *refl-onP B s* ==> *refl-onP* (*A* ∩ *B*) (λ*a a'*. *r a a'* ∧ *s a a'*)
⟨*proof*⟩

lemma *refl-onP-Un*: *refl-onP A r* ==> *refl-onP B s* ==> *refl-onP* (*A* ∪ *B*) (λ*a a'*. *r a a'* ∨ *s a a'*)
⟨*proof*⟩

lemma *refl-onP-empty*[*simp*]: *refl-onP* {} (λ*a a'*. *False*)
⟨*proof*⟩

lemma *refl-onP-tranclp*:
  assumes *refl-onP A r*
  shows *refl-onP A r*^++
⟨*proof*⟩

lemma *irreflPI*: (⋀*a*. ¬ *r a a*) ⟹ *irreflP r*
⟨*proof*⟩

lemma *irreflPE*:
  assumes *irreflP r*
  obtains ∀ *a*. ¬ *r a a*
⟨*proof*⟩

lemma *irreflPD*: ⟦ *irreflP r*; *r a a* ⟧ ⟹ *False*
⟨*proof*⟩

lemma *irreflclpD*:

$r^{\neq\neq}$ *a b* $\Longrightarrow$ *r a b* $\wedge$ *a* $\neq$ *b*
⟨*proof*⟩

lemma *irreflclpI* [*intro*!]:
⟦ *r a b*; *a* $\neq$ *b* ⟧ $\Longrightarrow$ $r^{\neq\neq}$ *a b*
⟨*proof*⟩

lemma *irreflclpE* [*elim*!]:
assumes $r^{\neq\neq}$ *a b*
obtains *r a b a* $\neq$ *b*
⟨*proof*⟩

lemma *transPI*: ($\bigwedge$*x y z*. ⟦ *r x y*; *r y z* ⟧ $\Longrightarrow$ *r x z*) $\Longrightarrow$ *transp r*
⟨*proof*⟩

lemma *transPD*: ⟦*transp r*; *r x y*; *r y z* ⟧ $\Longrightarrow$ *r x z*
⟨*proof*⟩

lemma *transP-tranclp*: *transp r*^++
⟨*proof*⟩

lemma *antisymPI*: ($\bigwedge$*x y*. ⟦ *r x y*; *r y x* ⟧ $\Longrightarrow$ *x* = *y*) $\Longrightarrow$ *antisymp r*
⟨*proof*⟩

lemma *antisymPD*: ⟦ *antisymp r*; *r a b*; *r b a* ⟧ $\Longrightarrow$ *a* = *b*
⟨*proof*⟩

lemma *antisym-subset*:
⟦ *antisymp r*; $\bigwedge$*a a'*. *s a a'* $\Longrightarrow$ *r a a'* ⟧ $\Longrightarrow$ *antisymp s*
⟨*proof*⟩

lemma *symPI*: ($\bigwedge$*x y*. *r x y* $\Longrightarrow$ *r y x*) $\Longrightarrow$ *symP r*
⟨*proof*⟩

lemma *symPD*: ⟦ *symP r*; *r x y* ⟧ $\Longrightarrow$ *r y x*
⟨*proof*⟩

8.3.2 Easy properties

lemma *porder-onI*:
⟦ *refl-onP A r*; *antisymp r*; *transp r* ⟧ $\Longrightarrow$ *porder-on A r*
⟨*proof*⟩

lemma *porder-onE*:
assumes *porder-on A r*
obtains *refl-onP A r antisymp r transp r*
⟨*proof*⟩

lemma *torder-onI*:
⟦ *porder-on A r*; *total-onP A r* ⟧ $\Longrightarrow$ *torder-on A r*
⟨*proof*⟩

lemma *torder-onE*:
assumes *torder-on A r*

obtains *porder-on A r total-onP A r*
⟨*proof*⟩

lemma *total-onI*:
$(\bigwedge x\ y.\ [\![\ x \in A;\ y \in A\ ]\!] \Longrightarrow (x,\ y) \in r \vee x = y \vee (y,\ x) \in r) \Longrightarrow \textit{total-on}\ A\ r$
⟨*proof*⟩

lemma *total-onPI*:
$(\bigwedge x\ y.\ [\![\ x \in A;\ y \in A\ ]\!] \Longrightarrow r\ x\ y \vee x = y \vee r\ y\ x) \Longrightarrow \textit{total-onP}\ A\ r$
⟨*proof*⟩

lemma *total-onD*:
$[\![\ \textit{total-on}\ A\ r;\ x \in A;\ y \in A\ ]\!] \Longrightarrow (x,\ y) \in r \vee x = y \vee (y,\ x) \in r$
⟨*proof*⟩

lemma *total-onPD*:
$[\![\ \textit{total-onP}\ A\ r;\ x \in A;\ y \in A\ ]\!] \Longrightarrow r\ x\ y \vee x = y \vee r\ y\ x$
⟨*proof*⟩

8.3.3 Order consistency

lemma *order-consistentI*:
$(\bigwedge a\ a'.\ [\![\ r\ a\ a';\ s\ a'\ a\ ]\!] \Longrightarrow a = a') \Longrightarrow \textit{order-consistent}\ r\ s$
⟨*proof*⟩

lemma *order-consistentD*:
$[\![\ \textit{order-consistent}\ r\ s;\ r\ a\ a';\ s\ a'\ a\ ]\!] \Longrightarrow a = a'$
⟨*proof*⟩

lemma *order-consistent-subset*:
$[\![\ \textit{order-consistent}\ r\ s;\ \bigwedge a\ a'.\ r'\ a\ a' \Longrightarrow r\ a\ a';\ \bigwedge a\ a'.\ s'\ a\ a' \Longrightarrow s\ a\ a'\ ]\!] \Longrightarrow \textit{order-consistent}\ r'\ s'$
⟨*proof*⟩

lemma *order-consistent-sym*:
$\textit{order-consistent}\ r\ s \Longrightarrow \textit{order-consistent}\ s\ r$
⟨*proof*⟩

lemma *antisym-order-consistent-self*:
$\textit{antisymp}\ r \Longrightarrow \textit{order-consistent}\ r\ r$
⟨*proof*⟩

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*: $x \in A$ **and** *y*: $y \in A$ **and** *s*: *s x y*
shows *r x y*
⟨*proof*⟩

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*: $B \subseteq A$
and *trancl*: $(\lambda a\ b.\ r\ a\ b \vee s\ a\ b)\hat{}{+}{+}\ x\ y$

obtains *r x y*
| *u v* **where** *r x u s u v r v y*
⟨*proof*⟩

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*: $B \subseteq A$
shows *antisymp* $(\lambda x\ y.\ r\ x\ y \vee s\ x\ y)\hat{}{++}$
⟨*proof*⟩

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*: $B \subseteq A$
shows *porder-on A* $(\lambda a\ b.\ r\ a\ b \vee s\ a\ b)\hat{}{++}$
⟨*proof*⟩

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*: $\bigwedge x\ y.\ t\ x\ y \Longrightarrow s\ x\ y$
and *subset*: $B \subseteq A$
shows *porder-on A* $(\lambda x\ y.\ r\ x\ y \vee t\ x\ y)\hat{}{++}$
⟨*proof*⟩

8.3.4 Order restrictions

lemma *restrictPI* [*intro*!, *simp*]:
$[\![\ r\ a\ b;\ a \in A;\ b \in A\ ]\!] \Longrightarrow (r\ |`\ A)\ a\ b$
⟨*proof*⟩

lemma *restrictPE* [*elim*!]:
assumes $(r\ |`\ A)\ a\ b$
obtains $r\ a\ b\ \ a \in A\ \ b \in A$
⟨*proof*⟩

lemma *restrictP-empty* [*simp*]: $R\ |`\ \{\} = (\lambda\text{-}\ \text{-}.\ \mathit{False})$
⟨*proof*⟩

lemma *refl-on-restrictPI*:
refl-onP A r $\Longrightarrow$ *refl-onP* $(A \cap B)\ (r\ |`\ B)$
⟨*proof*⟩

lemma *refl-on-restrictPI'*:
$[\![$ *refl-onP A r*; $B = A \cap C\ ]\!] \Longrightarrow$ *refl-onP B* $(r\ |`\ C)$
⟨*proof*⟩

lemma *antisym-restrictPI*:
antisymp r $\Longrightarrow$ *antisymp* $(r\ |`\ A)$
⟨*proof*⟩

lemma *trans-restrictPI*:
transp $r \Longrightarrow$ *transp* $(r \mid\text{`}\ A)$
⟨*proof*⟩

lemma *porder-on-restrictPI*:
porder-on $A\ r \Longrightarrow$ *porder-on* $(A \cap B)\ (r \mid\text{`}\ B)$
⟨*proof*⟩

lemma *porder-on-restrictPI'*:
$[\![$ *porder-on* $A\ r;\ B = A \cap C\ ]\!] \Longrightarrow$ *porder-on* $B\ (r \mid\text{`}\ C)$
⟨*proof*⟩

lemma *total-on-restrictPI*:
total-onP $A\ r \Longrightarrow$ *total-onP* $(A \cap B)\ (r \mid\text{`}\ B)$
⟨*proof*⟩

lemma *total-on-restrictPI'*:
$[\![$ *total-onP* $A\ r;\ B = A \cap C\ ]\!] \Longrightarrow$ *total-onP* $B\ (r \mid\text{`}\ C)$
⟨*proof*⟩

lemma *torder-on-restrictPI*:
torder-on $A\ r \Longrightarrow$ *torder-on* $(A \cap B)\ (r \mid\text{`}\ B)$
⟨*proof*⟩

lemma *torder-on-restrictPI'*:
$[\![$ *torder-on* $A\ r;\ B = A \cap C\ ]\!] \Longrightarrow$ *torder-on* $B\ (r \mid\text{`}\ C)$
⟨*proof*⟩

lemma *restrictP-commute*:
fixes $r :: 'a \Rightarrow 'a \Rightarrow bool$
shows $r \mid\text{`}\ A \mid\text{`}\ B = r \mid\text{`}\ B \mid\text{`}\ A$
⟨*proof*⟩

lemma *restrictP-subsume1*:
fixes $r :: 'a \Rightarrow 'a \Rightarrow bool$
assumes $A \subseteq B$
shows $r \mid\text{`}\ A \mid\text{`}\ B = r \mid\text{`}\ A$
⟨*proof*⟩

lemma *restrictP-subsume2*:
fixes $r :: 'a \Rightarrow 'a \Rightarrow bool$
assumes $B \subseteq A$
shows $r \mid\text{`}\ A \mid\text{`}\ B = r \mid\text{`}\ B$
⟨*proof*⟩

lemma *restrictP-idem* [*simp*]:
fixes $r :: 'a \Rightarrow 'a \Rightarrow bool$
shows $r \mid\text{`}\ A \mid\text{`}\ A = r \mid\text{`}\ A$
⟨*proof*⟩

8.3.5 Maximal elements w.r.t. a total order

definition *max-torder* $:: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$

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*
⟨*proof*⟩

lemma *refl-onP-DomainPD*: *refl-onP A r* ⟹ *A* = {*a*. *Domainp r a*}
⟨*proof*⟩

lemma *semilattice-max-torder*:
assumes *tot*: *torder-on A r*
shows *semilattice* (*max-torder r*)
⟨*proof*⟩

lemma *max-torder-ge-conv-disj*:
assumes *tot*: *torder-on A r* **and** *x*: *x* ∈ *A* **and** *y*: *y* ∈ *A*
shows *r z* (*max-torder r x y*) ⟷ *r z x* ∨ *r z y*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *domain*:
Domainp r a ⟷ *a* ∈ *A*
⟨*proof*⟩

lemma *Max-torder-in-Domain*:
assumes *B*: *finite B B* ≠ {} *B* ⊆ *A*
shows *Max-torder r B* ∈ *A*
⟨*proof*⟩

lemma *Max-torder-in-set*:
assumes *B*: *finite B B* ≠ {} *B* ⊆ *A*
shows *Max-torder r B* ∈ *B*
⟨*proof*⟩

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*⟩

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*⟩

lemma *inv-imageP-id* [*simp*]: *inv-imageP R id* = *R*
⟨*proof*⟩

lemma *inv-into-id* [*simp*]: *a* ∈ *A* ⟹ *inv-into A id a* = *a*
⟨*proof*⟩

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* ⟹ *saction P* (*NormalAction* (*ReadMem a al v*))
| *Write*: *is-volatile P al* ⟹ *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* ⇒ (*'addr*, *'thread-id*) *execution* ⇒ *JMM-action set*
where *sactions P E* = {*a*. *a* ∈ *actions E* ∧ *saction P* (*action-obs E a*)}

inductive-set *write-actions* :: (*'addr*, *'thread-id*) *execution* ⇒ *JMM-action set*
for *E* :: (*'addr*, *'thread-id*) *execution*
where
  write-actionsI: ⟦ *a* ∈ *actions E*; *is-write-action* (*action-obs E a*) ⟧ ⟹ *a* ∈ *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* ⇒ *htype* ⇒ *addr-loc set*

where
addr-locs P (*Class-type C*) = {*CField D F*|*D F*. ∃*fm T*. *P* ⊢ *C has F*:*T* (*fm*) *in D*}
| *addr-locs P* (*Array-type T n*) = ({*ACell n*′|*n*′. *n*′ < *n*} ∪ {*CField Object F*|*F*. ∃*fm T*. *P* ⊢ *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* ⇒ (′*addr*, ′*thread-id*) *obs-event action* ⇒ (′*addr* × *addr-loc*) *set*
where
action-loc-aux P (*NormalAction* (*NewHeapElem ad* (*Class-type C*))) =
{(*ad*, *CField D F*)|*D F T fm*. *P* ⊢ *C has F*:*T* (*fm*) *in D*}
| *action-loc-aux P* (*NormalAction* (*NewHeapElem ad* (*Array-type T n*′))) =
{(*ad*, *ACell n*)|*n*. *n* < *n*′} ∪ {(*ad*, *CField D F*)|*D F T fm*. *P* ⊢ *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 ⊢ *class-type-of hT has F*:*T* (*fm*) *in D* ⟹ (*ad*, *CField D F*) ∈ *action-loc-aux P* (*NormalAction* (*NewHeapElem ad hT*))
n < *n*′ ⟹ (*ad*, *ACell n*) ∈ *action-loc-aux P* (*NormalAction* (*NewHeapElem ad* (*Array-type T n*′)))
(*ad*, *al*) ∈ *action-loc-aux P* (*NormalAction* (*WriteMem ad al v*))
(*ad*, *al*) ∈ *action-loc-aux P* (*NormalAction* (*ReadMem ad al v*))
⟨*proof*⟩

lemma *action-loc-aux-cases* [*elim?*, *cases set*: *action-loc-aux*]:
assumes *adal* ∈ *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* ⊢ *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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*) ⟧
⟹ *a* ∈ *external-actions E*

fun *value-written-aux* :: *'m prog* ⇒ (*'addr*, *'thread-id*) *obs-event action* ⇒ *addr-loc* ⇒ *'addr val*
where
value-written-aux P (*NormalAction* (*NewHeapElem ad' hT*)) *al* = *addr-loc-default P hT al*
| *value-written-aux-WriteMem'*:
value-written-aux P (*NormalAction* (*WriteMem ad al' v*)) *al* = (*if al* = *al' then v else undefined*)
| *value-written-aux-undefined*:
value-written-aux P - *al* = *undefined*

primrec *value-written* :: *'m prog* ⇒ (*'addr*, *'thread-id*) *execution* ⇒ *JMM-action* ⇒ (*'addr* × *addr-loc*) ⇒ *'addr val*
where *value-written P E a* (*ad*, *al*) = *value-written-aux P* (*action-obs E a*) *al*

definition *value-read* :: (*'addr*, *'thread-id*) *execution* ⇒ *JMM-action* ⇒ *'addr val*
where
value-read E a =
(*case action-obs E a of*
NormalAction obs ⇒
(*case obs of*
ReadMem ad al v ⇒ *v*
| - ⇒ *undefined*)
| - ⇒ *undefined*)

definition *action-order* :: (*'addr*, *'thread-id*) *execution* ⇒ *JMM-action* ⇒ *JMM-action* ⇒ *bool* (‹- ⊢

- ≤a -› [51,0,50] 50)
where
E ⊢ a ≤a a′ ⟷
a ∈ actions E ∧ a′ ∈ actions E ∧
(if is-new-action (action-obs E a)
then is-new-action (action-obs E a′) ⟶ a ≤ a′
else ¬ is-new-action (action-obs E a′) ∧ a ≤ a′)

definition *program-order* :: ('addr, 'thread-id) execution ⇒ JMM-action ⇒ JMM-action ⇒ bool (‹- ⊢ - ≤po -› [51,0,50] 50)
where
E ⊢ a ≤po a′ ⟷ E ⊢ a ≤a a′ ∧ action-tid E a = action-tid E a′

inductive *synchronizes-with* ::
′m prog
⇒ (′thread-id × (′addr, ′thread-id) obs-event action) ⇒ (′thread-id × (′addr, ′thread-id) obs-event action) ⇒ bool
(‹- ⊢ - ↝sw -› [51, 51, 51] 50)
for P :: ′m prog
where
ThreadStart: P ⊢ (t, NormalAction (ThreadStart t′)) ↝sw (t′, InitialThreadAction)
| ThreadFinish: P ⊢ (t, ThreadFinishAction) ↝sw (t′, NormalAction (ThreadJoin t))
| UnlockLock: P ⊢ (t, NormalAction (SyncUnlock a)) ↝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 ⊢ (t, NormalAction (WriteMem a al v)) ↝sw (t′, NormalAction (ReadMem a al v′))
| VolatileNew:
al ∈ addr-locs P hT
⟹ P ⊢ (t, NormalAction (NewHeapElem a hT)) ↝sw (t′, NormalAction (ReadMem a al v))
| NewHeapElem: P ⊢ (t, NormalAction (NewHeapElem a hT)) ↝sw (t′, InitialThreadAction)
| Interrupt: P ⊢ (t, NormalAction (ObsInterrupt t′)) ↝sw (t″, NormalAction (ObsInterrupted t′))

definition *sync-order* ::
′m prog ⇒ (′addr, ′thread-id) execution ⇒ JMM-action ⇒ JMM-action ⇒ bool
(‹-,- ⊢ - ≤so -› [51,0,0,50] 50)
where
P,E ⊢ a ≤so a′ ⟷ a ∈ sactions P E ∧ a′ ∈ sactions P E ∧ E ⊢ a ≤a a′

definition *sync-with* ::
′m prog ⇒ (′addr, ′thread-id) execution ⇒ JMM-action ⇒ JMM-action ⇒ bool
(‹-,- ⊢ - ≤sw -› [51, 0, 0, 50] 50)
where
P,E ⊢ a ≤sw a′ ⟷
P,E ⊢ a ≤so a′ ∧ P ⊢ (action-tid E a, action-obs E a) ↝sw (action-tid E a′, action-obs E a′)

definition *po-sw* :: ′m prog ⇒ (′addr, ′thread-id) execution ⇒ JMM-action ⇒ JMM-action ⇒ bool
where po-sw P E a a′ ⟷ E ⊢ a ≤po a′ ∨ P,E ⊢ a ≤sw a′

abbreviation *happens-before* ::
′m prog ⇒ (′addr, ′thread-id) execution ⇒ JMM-action ⇒ JMM-action ⇒ bool
(‹-,- ⊢ - ≤hb -› [51, 0, 0, 50] 50)
where happens-before P E ≡ (po-sw P E)^++

type-synonym *write-seen* = JMM-action ⇒ JMM-action

definition *is-write-seen* :: *'m prog* ⇒ (*'addr*, *'thread-id*) *execution* ⇒ *write-seen* ⇒ *bool* **where**
is-write-seen P E ws ⟷
(∀ *a* ∈ *read-actions E*. ∀ *ad al v*. *action-obs E a* = *NormalAction* (*ReadMem ad al v*) ⟶
ws a ∈ *write-actions E* ∧ (*ad*, *al*) ∈ *action-loc P E* (*ws a*) ∧
value-written P E (*ws a*) (*ad*, *al*) = *v* ∧ ¬ *P*,*E* ⊢ *a* ≤*hb ws a* ∧
(*is-volatile P al* ⟶ ¬ *P*,*E* ⊢ *a* ≤*so ws a*) ∧
(∀ *w'* ∈ *write-actions E*. (*ad*, *al*) ∈ *action-loc P E w'* ⟶
(*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*) ⟶
w' = *ws a*))

definition *thread-start-actions-ok* :: (*'addr*, *'thread-id*) *execution* ⇒ *bool*
where
thread-start-actions-ok E ⟷
(∀ *a* ∈ *actions E*. ¬ *is-new-action* (*action-obs E a*) ⟶
(∃ *i*. *i* ≤ *a* ∧ *action-obs E i* = *InitialThreadAction* ∧ *action-tid E i* = *action-tid E a*))

primrec *wf-exec* :: *'m prog* ⇒ (*'addr*, *'thread-id*) *execution* × *write-seen* ⇒ *bool* (‹- ⊢ - √› [*51*, *50*] *51*)
where *P* ⊢ (*E*, *ws*) √ ⟷ *is-write-seen P E ws* ∧ *thread-start-actions-ok E*

inductive *most-recent-write-for* :: *'m prog* ⇒ (*'addr*, *'thread-id*) *execution* ⇒ *JMM-action* ⇒ *JMM-action* ⇒ *bool*
(‹-,- ⊢ - ⇝*mrw* -› [*50*, *0*, *51*] *51*)
for *P* :: *'m prog* **and** *E* :: (*'addr*, *'thread-id*) *execution* **and** *ra* :: *JMM-action* **and** *wa* :: *JMM-action*
where
⟦ *ra* ∈ *read-actions E*; *adal* ∈ *action-loc P E ra*; *E* ⊢ *wa* ≤*a ra*;
wa ∈ *write-actions E*; *adal* ∈ *action-loc P E wa*;
⋀*wa'*. ⟦ *wa'* ∈ *write-actions E*; *adal* ∈ *action-loc P E wa'* ⟧
⟹ *E* ⊢ *wa'* ≤*a wa* ∨ *E* ⊢ *ra* ≤*a wa'* ⟧
⟹ *P*,*E* ⊢ *ra* ⇝*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*
⟨*proof*⟩

lemmas *saction-iff* = *saction.simps*

lemma *actionsD*: $a \in$ *actions E* $\Longrightarrow$ *enat a* < *llength E*
⟨*proof*⟩

lemma *actionsE*:
assumes $a \in$ *actions E*
obtains *enat a* < *llength E*
⟨*proof*⟩

lemma *actions-lappend*:
llength xs = *enat n* $\Longrightarrow$ *actions* (*lappend xs ys*) = *actions xs* $\cup$ ((+) *n*) ‘ *actions ys*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *sactionsI*:
⟦ $a \in$ *actions E*; *saction P* (*action-obs E a*) ⟧ $\Longrightarrow$ $a \in$ *sactions P E*
⟨*proof*⟩

lemma *sactionsE*:
assumes $a \in$ *sactions P E*
obtains $a \in$ *actions E saction P* (*action-obs E a*)
⟨*proof*⟩

lemma *sactions-actions* [*simp*]:
$a \in$ *sactions P E* $\Longrightarrow$ $a \in$ *actions E*
⟨*proof*⟩

lemma *value-written-aux-WriteMem* [*simp*]:
value-written-aux P (*NormalAction* (*WriteMem ad al v*)) *al* = *v*
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

lemma *read-action-action-locE*:
assumes $r \in$ *read-actions* E
obtains *ad al* **where** $(ad, al) \in$ *action-loc* P E r
⟨*proof*⟩

lemma *read-actions-not-write-actions*:
$[\![a \in$ *read-actions* E; $a \in$ *write-actions* $E]\!] \Longrightarrow$ *False*
⟨*proof*⟩

lemma *read-actions-Int-write-actions* [*simp*]:
read-actions $E \cap$ *write-actions* $E = \{\}$
write-actions $E \cap$ *read-actions* $E = \{\}$
⟨*proof*⟩

lemma *action-loc-addr-fun*:
$[\![(ad, al) \in$ *action-loc* P E a; $(ad', al') \in$ *action-loc* P E $a]\!] \Longrightarrow ad = ad'$
⟨*proof*⟩

lemma *value-written-cong* [*cong*]:
$[\![P = P'$; $a = a'$; *action-obs* E a' = *action-obs* E' $a']\!]$
$\Longrightarrow$ *value-written* P E a = *value-written* P' E' a'
⟨*proof*⟩

declare *value-written.simps* [*simp del*]

lemma *new-actionsI*:
$[\![a \in$ *actions* E; *adal* $\in$ *action-loc* P E a; *is-new-action* (*action-obs* E a) $]\!]$
$\Longrightarrow a \in$ *new-actions-for* P E *adal*
⟨*proof*⟩

lemma *new-actionsE*:
assumes $a \in$ *new-actions-for* P E *adal*
obtains $a \in$ *actions* E *adal* $\in$ *action-loc* P E a *is-new-action* (*action-obs* E a)
⟨*proof*⟩

lemma *action-loc-read-action-singleton*:
$[\![r \in$ *read-actions* E; *adal* $\in$ *action-loc* P E r; *adal'* $\in$ *action-loc* P E $r]\!] \Longrightarrow$ *adal* = *adal'*
⟨*proof*⟩

lemma *addr-locsI*:
$P \vdash$ *class-type-of* hT *has* F:T (*fm*) *in* $D \Longrightarrow$ *CField* D $F \in$ *addr-locs* P hT
$[\![hT$ = *Array-type* T n; $n' < n]\!] \Longrightarrow$ *ACell* $n' \in$ *addr-locs* P hT
⟨*proof*⟩

8.4.3 Orders

8.4.4 Action order

lemma *action-orderI*:
assumes $a \in actions\ E\ a' \in actions\ E$
and $[\![$ *is-new-action* (*action-obs* $E\ a$); *is-new-action* (*action-obs* $E\ a'$) $]\!] \Longrightarrow a \leq a'$
and $\neg$ *is-new-action* (*action-obs* $E\ a$) $\Longrightarrow \neg$ *is-new-action* (*action-obs* $E\ a'$) $\wedge\ a \leq a'$
shows $E \vdash a \leq a\ a'$
⟨*proof*⟩

lemma *action-orderE*:
assumes $E \vdash a \leq a\ a'$
obtains $a \in actions\ E\ a' \in actions\ E$
is-new-action (*action-obs* $E\ a$) *is-new-action* (*action-obs* $E\ a'$) $\longrightarrow a \leq a'$
| $a \in actions\ E\ a' \in actions\ E$
$\neg$ *is-new-action* (*action-obs* $E\ a$) $\neg$ *is-new-action* (*action-obs* $E\ a'$) $a \leq a'$
⟨*proof*⟩

lemma *refl-action-order*:
refl-onP (*actions* E) (*action-order* E)
⟨*proof*⟩

lemma *antisym-action-order*:
antisymp (*action-order* E)
⟨*proof*⟩

lemma *trans-action-order*:
transp (*action-order* E)
⟨*proof*⟩

lemma *porder-action-order*:
porder-on (*actions* E) (*action-order* E)
⟨*proof*⟩

lemma *total-action-order*:
total-onP (*actions* E) (*action-order* E)
⟨*proof*⟩

lemma *torder-action-order*:
torder-on (*actions* E) (*action-order* E)
⟨*proof*⟩

lemma *wf-action-order*: *wfP* (*action-order* E)$^{\neq\neq}$
⟨*proof*⟩

lemma *action-order-is-new-actionD*:
$[\![$ $E \vdash a \leq a\ a'$; *is-new-action* (*action-obs* $E\ a'$) $]\!] \Longrightarrow$ *is-new-action* (*action-obs* $E\ a$)
⟨*proof*⟩

8.4.5 Program order

lemma *program-orderI*:
assumes $E \vdash a \leq a\ a'$ **and** *action-tid* $E\ a$ = *action-tid* $E\ a'$
shows $E \vdash a \leq po\ a'$

⟨*proof*⟩

lemma *program-orderE*:
assumes $E \vdash a \leq po\ a'$
obtains *t obs obs'*
where $E \vdash a \leq a\ a'$
and $action\text{-}tid\ E\ a = t\ action\text{-}obs\ E\ a = obs$
and $action\text{-}tid\ E\ a' = t\ action\text{-}obs\ E\ a' = obs'$
⟨*proof*⟩

lemma *refl-on-program-order*:
refl-onP (*actions E*) (*program-order E*)
⟨*proof*⟩

lemma *antisym-program-order*:
antisymp (*program-order E*)
⟨*proof*⟩

lemma *trans-program-order*:
transp (*program-order E*)
⟨*proof*⟩

lemma *porder-program-order*:
porder-on (*actions E*) (*program-order E*)
⟨*proof*⟩

lemma *total-program-order-on-tactions*:
total-onP (*tactions E t*) (*program-order E*)
⟨*proof*⟩

8.4.6 Synchronization order

lemma *sync-orderI*:
$[\![E \vdash a \leq a\ a';\ a \in sactions\ P\ E;\ a' \in sactions\ P\ E]\!] \Longrightarrow P,E \vdash a \leq so\ a'$
⟨*proof*⟩

lemma *sync-orderE*:
assumes $P,E \vdash a \leq so\ a'$
obtains $a \in sactions\ P\ E\ a' \in sactions\ P\ E\ E \vdash a \leq a\ a'$
⟨*proof*⟩

lemma *refl-on-sync-order*:
refl-onP (*sactions P E*) (*sync-order P E*)
⟨*proof*⟩

lemma *antisym-sync-order*:
antisymp (*sync-order P E*)
⟨*proof*⟩

lemma *trans-sync-order*:
transp (*sync-order P E*)
⟨*proof*⟩

lemma *porder-sync-order*:

porder-on (*sactions P E*) (*sync-order P E*)
⟨*proof*⟩

lemma *total-sync-order*:
total-onP (*sactions P E*) (*sync-order P E*)
⟨*proof*⟩

lemma *torder-sync-order*:
torder-on (*sactions P E*) (*sync-order P E*)
⟨*proof*⟩

8.4.7 Synchronizes with

lemma *sync-withI*:
$\llbracket$ $P,E \vdash a \leq so\ a'$; $P \vdash$ (*action-tid E a*, *action-obs E a*) $\leadsto sw$ (*action-tid E a'*, *action-obs E a'*) $\rrbracket$
$\Longrightarrow P,E \vdash a \leq sw\ a'$
⟨*proof*⟩

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*) $\leadsto sw$ (*action-tid E a'*, *action-obs E a'*)
⟨*proof*⟩

lemma *irrefl-synchronizes-with*:
irreflP (*synchronizes-with P*)
⟨*proof*⟩

lemma *irrefl-sync-with*:
irreflP (*sync-with P E*)
⟨*proof*⟩

lemma *anitsym-sync-with*:
antisymp (*sync-with P E*)
⟨*proof*⟩

lemma *consistent-program-order-sync-order*:
order-consistent (*program-order E*) (*sync-order P E*)
⟨*proof*⟩

lemma *consistent-program-order-sync-with*:
order-consistent (*program-order E*) (*sync-with P E*)
⟨*proof*⟩

8.4.8 Happens before

lemma *porder-happens-before*:
porder-on (*actions E*) (*happens-before P E*)
⟨*proof*⟩

lemma *porder-tranclp-po-so*:
porder-on (*actions E*) ($\lambda a\ a'$. *program-order E a a'* $\vee$ *sync-order P E a a'*)^++
⟨*proof*⟩

lemma *happens-before-refl*:

assumes $a \in actions\ E$
shows $P,E \vdash a \leq hb\ a$
⟨*proof*⟩

lemma *happens-before-into-po-so-tranclp*:
assumes $P,E \vdash a \leq hb\ a'$
shows $(\lambda a\ a'.\ E \vdash a \leq po\ a' \vee P,E \vdash a \leq so\ a')\hat{}{++}\ a\ a'$
⟨*proof*⟩

lemma *po-sw-into-action-order*:
$po\text{-}sw\ P\ E\ a\ a' \Longrightarrow E \vdash a \leq a\ a'$
⟨*proof*⟩

lemma *happens-before-into-action-order*:
assumes $P,E \vdash a \leq hb\ a'$
shows $E \vdash a \leq a\ a'$
⟨*proof*⟩

lemma *action-order-consistent-with-happens-before*:
$order\text{-}consistent\ (action\text{-}order\ E)\ (happens\text{-}before\ P\ E)$
⟨*proof*⟩

lemma *happens-before-new-actionD*:
assumes *hb*: $P,E \vdash a \leq hb\ a'$
and *new*: $is\text{-}new\text{-}action\ (action\text{-}obs\ E\ a')$
shows $is\text{-}new\text{-}action\ (action\text{-}obs\ E\ a)\ action\text{-}tid\ E\ a = action\text{-}tid\ E\ a'\ a \leq a'$
⟨*proof*⟩

lemma *external-actions-not-new*:
$[\![\ a \in external\text{-}actions\ E;\ is\text{-}new\text{-}action\ (action\text{-}obs\ E\ a)\ ]\!] \Longrightarrow False$
⟨*proof*⟩

8.4.9 Most recent writes and sequential consistency

lemma *most-recent-write-for-fun*:
$[\![\ P,E \vdash ra \rightsquigarrow mrw\ wa;\ P,E \vdash ra \rightsquigarrow mrw\ wa'\ ]\!] \Longrightarrow wa = wa'$
⟨*proof*⟩

lemma *THE-most-recent-writeI*: $P,E \vdash r \rightsquigarrow mrw\ w \Longrightarrow (THE\ w.\ P,E \vdash r \rightsquigarrow mrw\ w) = w$
⟨*proof*⟩

lemma *most-recent-write-for-write-actionsD*:
assumes $P,E \vdash ra \rightsquigarrow mrw\ wa$
shows $wa \in write\text{-}actions\ E$
⟨*proof*⟩

lemma *most-recent-write-recent*:
$[\![\ P,E \vdash r \rightsquigarrow mrw\ w;\ adal \in action\text{-}loc\ P\ E\ r;\ w' \in write\text{-}actions\ E;\ adal \in action\text{-}loc\ P\ E\ w'\ ]\!]$
$\Longrightarrow E \vdash w' \leq a\ w \vee E \vdash r \leq a\ w'$
⟨*proof*⟩

lemma *is-write-seenI*:
$[\![\ \bigwedge a\ ad\ al\ v.\ [\![\ a \in read\text{-}actions\ E;\ action\text{-}obs\ E\ a = NormalAction\ (ReadMem\ ad\ al\ v)\ ]\!]$
$\Longrightarrow ws\ a \in write\text{-}actions\ E;$

$\bigwedge a\ ad\ al\ v.\ [\![\ a \in \textit{read-actions}\ E;\ \textit{action-obs}\ E\ a = \textit{NormalAction}\ (\textit{ReadMem}\ ad\ al\ v)\ ]\!]$
$\Longrightarrow (ad,\ al) \in \textit{action-loc}\ P\ E\ (ws\ a);$
$\bigwedge a\ ad\ al\ v.\ [\![\ a \in \textit{read-actions}\ E;\ \textit{action-obs}\ E\ a = \textit{NormalAction}\ (\textit{ReadMem}\ ad\ al\ v)\ ]\!]$
$\Longrightarrow \textit{value-written}\ P\ E\ (ws\ a)\ (ad,\ al) = v;$
$\bigwedge a\ ad\ al\ v.\ [\![\ a \in \textit{read-actions}\ E;\ \textit{action-obs}\ E\ a = \textit{NormalAction}\ (\textit{ReadMem}\ ad\ al\ v)\ ]\!]$
$\Longrightarrow \neg\ P,E \vdash a \le hb\ ws\ a;$
$\bigwedge a\ ad\ al\ v.\ [\![\ a \in \textit{read-actions}\ E;\ \textit{action-obs}\ E\ a = \textit{NormalAction}\ (\textit{ReadMem}\ ad\ al\ v);\ \textit{is-volatile}$
$P\ al\ ]\!]$
$\Longrightarrow \neg\ P,E \vdash a \le so\ ws\ a;$
$\bigwedge a\ ad\ al\ v\ a'.\ [\![\ a \in \textit{read-actions}\ E;\ \textit{action-obs}\ E\ a = \textit{NormalAction}\ (\textit{ReadMem}\ ad\ al\ v);$
$a' \in \textit{write-actions}\ E;\ (ad,\ al) \in \textit{action-loc}\ P\ E\ a';\ P,E \vdash ws\ a \le hb\ a';$
$P,E \vdash a' \le hb\ a\ ]\!] \Longrightarrow a' = ws\ a;$
$\bigwedge a\ ad\ al\ v\ a'.\ [\![\ a \in \textit{read-actions}\ E;\ \textit{action-obs}\ E\ a = \textit{NormalAction}\ (\textit{ReadMem}\ ad\ al\ v);$
$a' \in \textit{write-actions}\ E;\ (ad,\ al) \in \textit{action-loc}\ P\ E\ a';\ \textit{is-volatile}\ P\ al;\ P,E \vdash ws\ a \le so\ a';$
$P,E \vdash a' \le so\ a\ ]\!] \Longrightarrow a' = ws\ a\ ]\!]$
$\Longrightarrow \textit{is-write-seen}\ P\ E\ ws$
⟨*proof*⟩

lemma *is-write-seenD*:
$[\![\ \textit{is-write-seen}\ P\ E\ ws;\ a \in \textit{read-actions}\ E;\ \textit{action-obs}\ E\ a = \textit{NormalAction}\ (\textit{ReadMem}\ ad\ al\ v)\ ]\!]$
$\Longrightarrow ws\ a \in \textit{write-actions}\ E \wedge (ad,\ al) \in \textit{action-loc}\ P\ E\ (ws\ a) \wedge \textit{value-written}\ P\ E\ (ws\ a)\ (ad,\ al)$
$= v \wedge \neg\ P,E \vdash a \le hb\ ws\ a \wedge (\textit{is-volatile}\ P\ al \longrightarrow \neg\ P,E \vdash a \le so\ ws\ a) \wedge$
$(\forall\, a' \in \textit{write-actions}\ E.\ (ad,\ al) \in \textit{action-loc}\ P\ E\ a' \wedge (P,E \vdash ws\ a \le hb\ a' \wedge P,E \vdash a' \le hb\ a \vee$
$\textit{is-volatile}\ P\ al \wedge P,E \vdash ws\ a \le so\ a' \wedge P,E \vdash a' \le so\ a) \longrightarrow a' = ws\ a)$
⟨*proof*⟩

lemma *thread-start-actions-okI*:
$(\bigwedge a.\ [\![\ a \in \textit{actions}\ E;\ \neg\ \textit{is-new-action}\ (\textit{action-obs}\ E\ a)\ ]\!]$
$\Longrightarrow \exists\, i.\ i \le a \wedge \textit{action-obs}\ E\ i = \textit{InitialThreadAction} \wedge \textit{action-tid}\ E\ i = \textit{action-tid}\ E\ a)$
$\Longrightarrow \textit{thread-start-actions-ok}\ E$
⟨*proof*⟩

lemma *thread-start-actions-okD*:
$[\![\ \textit{thread-start-actions-ok}\ E;\ a \in \textit{actions}\ E;\ \neg\ \textit{is-new-action}\ (\textit{action-obs}\ E\ a)\ ]\!]$
$\Longrightarrow \exists\, i.\ i \le a \wedge \textit{action-obs}\ E\ i = \textit{InitialThreadAction} \wedge \textit{action-tid}\ E\ i = \textit{action-tid}\ E\ a$
⟨*proof*⟩

lemma *thread-start-actions-ok-prefix*:
$[\![\ \textit{thread-start-actions-ok}\ E';\ \textit{lprefix}\ E\ E'\ ]\!] \Longrightarrow \textit{thread-start-actions-ok}\ E$
⟨*proof*⟩

lemma *wf-execI* [*intro?*]:
$[\![\ \textit{is-write-seen}\ P\ E\ ws;$
$\textit{thread-start-actions-ok}\ E\ ]\!]$
$\Longrightarrow P \vdash (E,\ ws)\ \surd$
⟨*proof*⟩

lemma *wf-exec-is-write-seenD*:
$P \vdash (E,\ ws)\ \surd \Longrightarrow \textit{is-write-seen}\ P\ E\ ws$
⟨*proof*⟩

lemma *wf-exec-thread-start-actions-okD*:
$P \vdash (E,\ ws)\ \surd \Longrightarrow \textit{thread-start-actions-ok}\ E$
⟨*proof*⟩

lemma *sequentially-consistentI*:
($\bigwedge r.\ r \in$ *read-actions* $E \Longrightarrow P,E \vdash r \rightsquigarrow$*mrw ws r*)
$\Longrightarrow$ *sequentially-consistent P* (*E*, *ws*)
⟨*proof*⟩

lemma *sequentially-consistentE*:
assumes *sequentially-consistent P* (*E*, *ws*) $a \in$ *read-actions E*
obtains $P,E \vdash a \rightsquigarrow$*mrw ws a*
⟨*proof*⟩

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*
⟨*proof*⟩

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*

$obs \approx InitialThreadAction$
$obs \approx ThreadFinishAction$
$obs \approx NormalAction\ (NewHeapElem\ a\ hT)$
$obs \approx NormalAction\ (ReadMem\ ad\ al\ v')$
$obs \approx NormalAction\ (WriteMem\ ad\ al\ v')$
$obs \approx NormalAction\ (ThreadStart\ t)$
$obs \approx NormalAction\ (ThreadJoin\ t)$
$obs \approx NormalAction\ (SyncLock\ a)$
$obs \approx NormalAction\ (SyncUnlock\ a)$
$obs \approx NormalAction\ (ObsInterrupt\ t)$
$obs \approx NormalAction\ (ObsInterrupted\ t)$
$obs \approx NormalAction\ (ExternalCall\ a\ M\ vs\ v)$

inductive-simps *sim-action-simps* [*simp*]:
$InitialThreadAction \approx obs$
$ThreadFinishAction \approx obs$
$NormalAction\ (NewHeapElem\ a\ hT) \approx obs$
$NormalAction\ (ReadMem\ ad\ al\ v) \approx obs$
$NormalAction\ (WriteMem\ ad\ al\ v) \approx obs$
$NormalAction\ (ThreadStart\ t) \approx obs$
$NormalAction\ (ThreadJoin\ t) \approx obs$
$NormalAction\ (SyncLock\ a) \approx obs$
$NormalAction\ (SyncUnlock\ a) \approx obs$
$NormalAction\ (ObsInterrupt\ t) \approx obs$
$NormalAction\ (ObsInterrupted\ t) \approx obs$
$NormalAction\ (ExternalCall\ a\ M\ vs\ v) \approx obs$

$obs \approx InitialThreadAction$
$obs \approx ThreadFinishAction$
$obs \approx NormalAction\ (NewHeapElem\ a\ hT)$
$obs \approx NormalAction\ (ReadMem\ ad\ al\ v')$
$obs \approx NormalAction\ (WriteMem\ ad\ al\ v')$
$obs \approx NormalAction\ (ThreadStart\ t)$
$obs \approx NormalAction\ (ThreadJoin\ t)$
$obs \approx NormalAction\ (SyncLock\ a)$
$obs \approx NormalAction\ (SyncUnlock\ a)$
$obs \approx NormalAction\ (ObsInterrupt\ t)$
$obs \approx NormalAction\ (ObsInterrupted\ t)$
$obs \approx NormalAction\ (ExternalCall\ a\ M\ vs\ v)$

lemma *sim-action-trans* [*trans*]:
$[\![\ obs \approx obs';\ obs' \approx obs''\ ]\!] \Longrightarrow obs \approx obs''$
⟨*proof*⟩

lemma *sim-action-sym* [*sym*]:
assumes $obs \approx obs'$
shows $obs' \approx obs$
⟨*proof*⟩

lemma *sim-actions-sym* [*sym*]:
$E\ [\approx]\ E' \Longrightarrow E'\ [\approx]\ E$
⟨*proof*⟩

lemma *sim-actions-action-obsD*:
$E\ [\approx]\ E' \Longrightarrow \textit{action-obs}\ E\ a \approx \textit{action-obs}\ E'\ a$
⟨*proof*⟩

lemma *sim-actions-action-tidD*:
$E\ [\approx]\ E' \Longrightarrow \textit{action-tid}\ E\ a = \textit{action-tid}\ E'\ a$
⟨*proof*⟩

lemma *action-loc-aux-sim-action*:
$a \approx a' \Longrightarrow \textit{action-loc-aux}\ P\ a = \textit{action-loc-aux}\ P\ a'$
⟨*proof*⟩

lemma *eq-into-sim-actions*:
assumes $E = E'$
shows $E\ [\approx]\ E'$
⟨*proof*⟩

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*:
$[\![\ E \in \mathcal{E};\ a \in \textit{new-actions-for}\ P\ E\ \textit{adal};\ a' \in \textit{new-actions-for}\ P\ E\ \textit{adal}\ ]\!] \Longrightarrow a = a'$
and $\mathcal{E}$-*sequential-completion*:
$[\![\ E \in \mathcal{E};\ P \vdash (E,\ \textit{ws})\ \surd;\ \bigwedge a.\ [\![\ a < r;\ a \in \textit{read-actions}\ E\ ]\!] \Longrightarrow P,E \vdash a \leadsto \textit{mrw ws}\ a\ ]\!]$
$\Longrightarrow \exists E' \in \mathcal{E}.\ \exists \textit{ws}'.\ P \vdash (E',\ \textit{ws}')\ \surd \wedge \textit{ltake}\ (\textit{enat}\ r)\ E = \textit{ltake}\ (\textit{enat}\ r)\ E' \wedge \textit{sequentially-consistent}$
$P\ (E',\ \textit{ws}') \wedge$
$\textit{action-tid}\ E\ r = \textit{action-tid}\ E'\ r \wedge \textit{action-obs}\ E\ r \approx \textit{action-obs}\ E'\ r \wedge$
$(r \in \textit{actions}\ E \longrightarrow r \in \textit{actions}\ E')$

locale *executions-aux* =
executions-base $\mathcal{E}$ P
for $\mathcal{E}$ :: ('*addr*, '*thread-id*) *execution set*
and P :: '*m prog* +
assumes *init-before-read*:
$[\![\ \ E \in \mathcal{E};\ P \vdash (E,\ \textit{ws})\ \surd;\ r \in \textit{read-actions}\ E;\ \textit{adal} \in \textit{action-loc}\ P\ E\ r;$
$\bigwedge a.\ [\![\ a < r;\ a \in \textit{read-actions}\ E\ ]\!] \Longrightarrow P,E \vdash a \leadsto \textit{mrw ws}\ a\ ]\!]$
$\Longrightarrow \exists i<r.\ i \in \textit{new-actions-for}\ P\ E\ \textit{adal}$
and $\mathcal{E}$-*new-actions-for-fun*:
$[\![\ E \in \mathcal{E};\ a \in \textit{new-actions-for}\ P\ E\ \textit{adal};\ a' \in \textit{new-actions-for}\ P\ E\ \textit{adal}\ ]\!] \Longrightarrow a = a'$

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*:
$[\![\ E \in \mathcal{E};\ P \vdash (E,\ \textit{ws})\ \surd;\ \bigwedge a.\ [\![\ a < r;\ a \in \textit{read-actions}\ E\ ]\!] \Longrightarrow P,E \vdash a \leadsto \textit{mrw ws}\ a\ ]\!]$
$\Longrightarrow \exists E' \in \mathcal{E}.\ \exists \textit{ws}'.\ P \vdash (E',\ \textit{ws}')\ \surd \wedge \textit{ltake}\ (\textit{enat}\ r)\ E = \textit{ltake}\ (\textit{enat}\ r)\ E' \wedge$

($\forall a \in$ *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* $E\ r$) $\wedge$
($r \in$ *actions* $E \longrightarrow r \in$ *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* $\Rightarrow$ *JMM-action*

type-synonym (*'addr*, *'thread-id*) *justification* = *nat* $\Rightarrow$ (*'addr*, *'thread-id*) *justifying-execution*

definition *wf-action-translation-on* ::
(*'addr*, *'thread-id*) *execution* $\Rightarrow$ (*'addr*, *'thread-id*) *execution* $\Rightarrow$ *JMM-action set* $\Rightarrow$ (*JMM-action* $\Rightarrow$ *JMM-action*) $\Rightarrow$ *bool*
where
wf-action-translation-on $E\ E'\ A\ f \longleftrightarrow$
inj-on f (*actions* E) $\wedge$
($\forall a \in A$. *action-tid* $E\ a$ = *action-tid* E' ($f\ a$) $\wedge$ *action-obs* $E\ a \approx$ *action-obs* E' ($f\ a$))

abbreviation *wf-action-translation* :: (*'addr*, *'thread-id*) *execution* $\Rightarrow$ (*'addr*, *'thread-id*) *justifying-execution* $\Rightarrow$ *bool*
where
wf-action-translation $E\ J \equiv$
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 $\longleftrightarrow$
committed (J *0*) = {} $\wedge$
($\forall n$. *action-translation* ($J\ n$) ‘ *committed* ($J\ n$) $\subseteq$ *action-translation* (J (*Suc* n)) ‘ *committed* (J (*Suc* n))) $\wedge$
actions E = ($\bigcup n$. *action-translation* ($J\ n$) ‘ *committed* ($J\ n$))

definition *justification-well-formed* :: *bool* **where**
justification-well-formed $\longleftrightarrow$ ($\forall$ *n*. *P* $\vdash$ (*justifying-exec* (*J n*), *justifying-ws* (*J n*)) $\surd$)

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 ∈ committed-n ∨
(action-translation (J (Suc n)) (justifying-ws (J (Suc n)) r′) ∈ committed-n ∧ ws r ∈ 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

$P \vdash (E,\ ws)$ *weakly-justified-by* $J \longleftrightarrow$
is-commit-sequence $E\ J \wedge$
justification-well-formed $P\ J \wedge$
committed-subset-actions $J \wedge$
happens-before-committed-weak $P\ E\ ws\ J \wedge$
— no *sync-order* constraint
value-written-committed $P\ E\ J \wedge$
write-seen-committed $ws\ J \wedge$
uncommitted-reads-see-hb $P\ J \wedge$
committed-reads-see-committed-writes-weak $ws\ J \wedge$
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* $\Rightarrow$ $('addr,\ 'thread\text{-}id)$ *execution* $\Rightarrow$ *JMM-action* $\Rightarrow$ *JMM-action* $\Rightarrow$ *bool*
(‹-,- ⊢/(-)†(-)› [*51*,*50*,*50*,*50*] *51*)
where
$P,E \vdash a \dagger a' \longleftrightarrow$
$(a \in \textit{read-actions}\ E \wedge a' \in \textit{write-actions}\ E \vee$
$a \in \textit{write-actions}\ E \wedge a' \in \textit{read-actions}\ E \vee$
$a \in \textit{write-actions}\ E \wedge a' \in \textit{write-actions}\ E) \wedge$
$(\exists\, ad\ al.\ (ad,\ al) \in \textit{action-loc}\ P\ E\ a \cap \textit{action-loc}\ P\ E\ a' \wedge \neg\ \textit{is-volatile}\ P\ al)$

definition *correctly-synchronized* :: $'m$ *prog* $\Rightarrow$ $('addr,\ 'thread\text{-}id)$ *execution set* $\Rightarrow$ *bool*
where
correctly-synchronized $P\ \mathcal{E} \longleftrightarrow$
$(\forall E \in \mathcal{E}.\ \forall ws.\ P \vdash (E,\ ws) \surd \longrightarrow \textit{sequentially-consistent}\ P\ (E,\ ws)$
$\longrightarrow (\forall a \in \textit{actions}\ E.\ \forall a' \in \textit{actions}\ E.\ P,E \vdash a \dagger a'$
$\longrightarrow P,E \vdash a \leq hb\ a' \vee P,E \vdash a' \leq hb\ a))$

primrec *gen-legal-execution* ::
$('m\ \textit{prog} \Rightarrow ('addr,\ 'thread\text{-}id)\ \textit{execution} \times \textit{write-seen} \Rightarrow ('addr,\ 'thread\text{-}id)\ \textit{justification} \Rightarrow \textit{bool})$
$\Rightarrow\ 'm\ \textit{prog} \Rightarrow ('addr,\ 'thread\text{-}id)\ \textit{execution set} \Rightarrow ('addr,\ 'thread\text{-}id)\ \textit{execution} \times \textit{write-seen} \Rightarrow \textit{bool}$
where
gen-legal-execution is-justification $P\ \mathcal{E}\ (E,\ ws) \longleftrightarrow$
$E \in \mathcal{E} \wedge P \vdash (E,\ ws) \surd \wedge$
$(\exists J.\ \textit{is-justification}\ P\ (E,\ ws)\ J \wedge \textit{range}\ (\textit{justifying-exec} \circ J) \subseteq \mathcal{E})$

abbreviation *legal-execution* ::
$'m\ \textit{prog} \Rightarrow ('addr,\ 'thread\text{-}id)\ \textit{execution set} \Rightarrow ('addr,\ 'thread\text{-}id)\ \textit{execution} \times \textit{write-seen} \Rightarrow \textit{bool}$
where
legal-execution $\equiv$ *gen-legal-execution is-justified-by*

abbreviation *weakly-legal-execution* ::
$'m$ *prog* $\Rightarrow$ (*'addr*, *'thread-id*) *execution set* $\Rightarrow$ (*'addr*, *'thread-id*) *execution* $\times$ *write-seen* $\Rightarrow$ *bool*
where
weakly-legal-execution $\equiv$ *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*)
⟨*proof*⟩

lemma *legal-executionI*:
⟦ $E \in \mathcal{E}$; $P \vdash (E, ws) \surd$; *is-justification P* (*E*, *ws*) *J*; *range* (*justifying-exec* $\circ$ *J*) $\subseteq \mathcal{E}$ ⟧
$\Longrightarrow$ *gen-legal-execution is-justification P* $\mathcal{E}$ (*E*, *ws*)
⟨*proof*⟩

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* $\circ$ *J*) $\subseteq \mathcal{E}$
⟨*proof*⟩

lemma *legal-$\mathcal{E}$D*: *gen-legal-execution is-justification P* $\mathcal{E}$ (*E*, *ws*) $\Longrightarrow E \in \mathcal{E}$
⟨*proof*⟩

lemma *legal-wf-execD*:
gen-legal-execution is-justification P $\mathcal{E}$ *Ews* $\Longrightarrow P \vdash$ *Ews* $\surd$
⟨*proof*⟩

lemma *correctly-synchronizedD*:
⟦ *correctly-synchronized P* $\mathcal{E}$; $E \in \mathcal{E}$; $P \vdash (E, ws) \surd$; *sequentially-consistent P* (*E*, *ws*) ⟧
$\Longrightarrow \forall a\ a'.\ a \in$ *actions E* $\longrightarrow a' \in$ *actions E* $\longrightarrow P,E \vdash a \dagger a' \longrightarrow P,E \vdash a \leq hb\ a' \lor P,E \vdash a' \leq hb$
a
⟨*proof*⟩

lemma *wf-action-translation-on-actionD*:
⟦ *wf-action-translation-on E E' A f*; $a \in A$ ⟧
$\Longrightarrow$ *action-tid E a* = *action-tid E'* (*f a*) $\land$ *action-obs E a* $\approx$ *action-obs E'* (*f a*)
⟨*proof*⟩

lemma *wf-action-translation-on-inj-onD*:
wf-action-translation-on E E' A f $\Longrightarrow$ *inj-on f* (*actions E*)
⟨*proof*⟩

lemma *wf-action-translation-on-action-locD*:
⟦ *wf-action-translation-on E E' A f*; $a \in A$ ⟧
$\Longrightarrow$ *action-loc P E a* = *action-loc P E'* (*f a*)
⟨*proof*⟩

lemma *weakly-justified-write-seen-hb-read-committed*:
assumes *J*: $P \vdash (E, ws)$ *weakly-justified-by J*
and *r*: $r \in$ *read-actions* (*justifying-exec* (*J n*)) $r \in$ *committed* (*J n*)
shows *ws* (*action-translation* (*J n*) *r*) $\in$ *action-translation* (*J n*) ‘ *committed* (*J n*)
⟨*proof*⟩

lemma *justified-write-seen-hb-read-committed*:
assumes *J*: $P \vdash (E, ws)$ *justified-by J*
and *r*: $r \in$ *read-actions* (*justifying-exec* (*J n*)) $r \in$ *committed* (*J n*)
shows *justifying-ws* (*J n*) $r \in$ *committed* (*J n*) (**is** *?thesis1*)
and *ws* (*action-translation* (*J n*) *r*) $\in$ *action-translation* (*J n*) ' *committed* (*J n*) (**is** *?thesis2*)
⟨*proof*⟩

lemma *is-justified-by-imp-is-weakly-justified-by*:
assumes *justified*: $P \vdash (E, ws)$ *justified-by J*
and *wf*: $P \vdash (E, ws) \surd$
shows $P \vdash (E, ws)$ *weakly-justified-by J*
⟨*proof*⟩

corollary *legal-imp-weakly-legal-execution*:
legal-execution $P\ \mathcal{E}\ Ews \Longrightarrow$ *weakly-legal-execution* $P\ \mathcal{E}\ Ews$
⟨*proof*⟩

lemma *drop-0th-justifying-exec*:
assumes $P \vdash (E, ws)$ *justified-by J*
and *wf*: $P \vdash (E', ws') \surd$
shows $P \vdash (E, ws)$ *justified-by* (*J*(*0* := (|*committed* = {}, *justifying-exec* = E', *justifying-ws* = ws', *action-translation* = *id*|)))
(**is** - ⊢ - *justified-by ?J*)
⟨*proof*⟩

lemma *drop-0th-weakly-justifying-exec*:
assumes $P \vdash (E, ws)$ *weakly-justified-by J*
and *wf*: $P \vdash (E', ws') \surd$
shows $P \vdash (E, ws)$ *weakly-justified-by* (*J*(*0* := (|*committed* = {}, *justifying-exec* = E', *justifying-ws* = ws', *action-translation* = *id*|)))
(**is** - ⊢ - *weakly-justified-by ?J*)
⟨*proof*⟩

8.4.13 Executions with common prefix

lemma *actions-change-prefix*:
assumes *read*: $a \in$ *actions E*
and *prefix*: *ltake n E* [≈] *ltake n E'*
and *rn*: *enat* $a < n$
shows $a \in$ *actions E'*
⟨*proof*⟩

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*⟩

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*⟩

lemma *read-actions-change-prefix*:
assumes *read*: $r \in$ *read-actions E*
and *prefix*: *ltake n E* [≈] *ltake n E′ enat r* $<$ *n*
shows $r \in$ *read-actions E′*
⟨*proof*⟩

lemma *sim-action-is-write-action-eq*:
assumes *obs* ≈ *obs′*
shows *is-write-action obs* ⟷ *is-write-action obs′*
⟨*proof*⟩

lemma *write-actions-change-prefix*:
assumes *write*: $w \in$ *write-actions E*
and *prefix*: *ltake n E* [≈] *ltake n E′ enat w* $<$ *n*
shows $w \in$ *write-actions E′*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *sim-action-is-new-action-eq*:
assumes *obs* ≈ *obs′*
shows *is-new-action obs* = *is-new-action obs′*
⟨*proof*⟩

lemma *action-order-change-prefix*:
assumes *ao*: $E \vdash a \leq a\ a'$
and *prefix*: *ltake n E* [≈] *ltake n E′*
and *an*: *enat a* $<$ *n*
and *a′n*: *enat a′* $<$ *n*
shows $E' \vdash a \leq a\ a'$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *action-tid-change-prefix*:
assumes *prefix*: *ltake n E* [≈] *ltake n E′*
and *an*: *enat a* $<$ *n*
shows *action-tid E a* = *action-tid E′ a*
⟨*proof*⟩

lemma *program-order-change-prefix*:
assumes *po*: $E \vdash a \leq po\ a'$
and *prefix*: *ltake n E* [≈] *ltake n E′*
and *an*: *enat a* $<$ *n*
and *a′n*: *enat a′* $<$ *n*
shows $E' \vdash a \leq po\ a'$

⟨*proof*⟩

lemma *sim-action-sactionD*:
assumes $obs \approx obs'$
shows $saction\ P\ obs \longleftrightarrow saction\ P\ obs'$
⟨*proof*⟩

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 $a \in sactions\ P\ E'$
⟨*proof*⟩

lemma *sync-order-change-prefix*:
assumes *so*: $P,E \vdash a \leq so\ a'$
and *prefix*: $ltake\ n\ E\ [\approx]\ ltake\ n\ E'$
and *an*: $enat\ a < n$
and *a'n*: $enat\ a' < n$
shows $P,E' \vdash a \leq so\ a'$
⟨*proof*⟩

lemma *sim-action-synchronizes-withD*:
assumes $obs \approx obs'\ obs'' \approx obs'''$
shows $P \vdash (t,\ obs) \leadsto sw\ (t',\ obs'') \longleftrightarrow P \vdash (t,\ obs') \leadsto sw\ (t',\ obs''')$
⟨*proof*⟩

lemma *sync-with-change-prefix*:
assumes *sw*: $P,E \vdash a \leq sw\ a'$
and *prefix*: $ltake\ n\ E\ [\approx]\ ltake\ n\ E'$
and *an*: $enat\ a < n$
and *a'n*: $enat\ a' < n$
shows $P,E' \vdash a \leq sw\ a'$
⟨*proof*⟩

lemma *po-sw-change-prefix*:
assumes *posw*: $po\text{-}sw\ P\ E\ a\ a'$
and *prefix*: $ltake\ n\ E\ [\approx]\ ltake\ n\ E'$
and *an*: $enat\ a < n$
and *a'n*: $enat\ a' < n$
shows $po\text{-}sw\ P\ E'\ a\ a'$
⟨*proof*⟩

lemma *happens-before-new-not-new*:
assumes *tsa-ok*: *thread-start-actions-ok E*
and *a*: $a \in actions\ E$
and *a'*: $a' \in actions\ E$
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*⟩

lemma *happens-before-change-prefix*:
assumes *hb*: $P,E \vdash a \leq hb\ a'$
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 $P,E' \vdash a \leq hb\ a'$
⟨*proof*⟩

lemma *thread-start-actions-ok-change*:
assumes *tsa*: *thread-start-actions-ok E*
and *sim*: E [≈] E'
shows *thread-start-actions-ok* E'
⟨*proof*⟩

context *executions-aux* **begin**

lemma $\mathcal{E}$*-new-same-addr-singleton*:
assumes *E*: $E \in \mathcal{E}$
shows $\exists\, a.$ *new-actions-for P E adal* $\subseteq \{a\}$
⟨*proof*⟩

lemma *new-action-before-read*:
assumes *E*: $E \in \mathcal{E}$
and *wf*: $P \vdash (E,\ ws)\ \surd$
and *ra*: $ra \in$ *read-actions E*
and *adal*: $adal \in$ *action-loc P E ra*
and *new*: $wa \in$ *new-actions-for P E adal*
and *sc*: $\bigwedge a.\ [\![\ a < ra;\ a \in \textit{read-actions}\ E\ ]\!] \Longrightarrow P,E \vdash a \leadsto mrw\ ws\ a$
shows $wa < ra$
⟨*proof*⟩

lemma *mrw-before*:
assumes *E*: $E \in \mathcal{E}$
and *wf*: $P \vdash (E,\ ws)\ \surd$
and *mrw*: $P,E \vdash r \leadsto mrw\ w$
and *sc*: $\bigwedge a.\ [\![\ a < r;\ a \in \textit{read-actions}\ E\ ]\!] \Longrightarrow P,E \vdash a \leadsto mrw\ ws\ a$
shows $w < r$
⟨*proof*⟩

lemma *mrw-change-prefix*:
assumes E': $E' \in \mathcal{E}$
and *mrw*: $P,E \vdash r \leadsto mrw\ w$
and *tsa-ok*: *thread-start-actions-ok* E'
and *prefix*: *ltake n E* [≈] *ltake n* E'
and *an*: *enat* $r < n$
and $a'n$: *enat* $w < n$
shows $P,E' \vdash r \leadsto mrw\ w$
⟨*proof*⟩

lemma *action-order-read-before-write*:
assumes *E*: $E \in \mathcal{E}$ $P \vdash (E,\ ws)\ \surd$
and *ao*: $E \vdash w \leq a\ r$
and *r*: $r \in$ *read-actions E*

and *w*: $w \in$ *write-actions E*
and *adal*: $adal \in$ *action-loc P E r adal* $\in$ *action-loc P E w*
and *sc*: $\bigwedge a.\ [\![\ a < r;\ a \in$ *read-actions E* $]\!] \Longrightarrow P,E \vdash a \leadsto mrw\ ws\ a$
shows $w < r$
⟨*proof*⟩

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*: $P \vdash (E,\ ws)\ \surd$
and *E*: $E \in \mathcal{E}$
and *sync*: *correctly-synchronized P* $\mathcal{E}$
and *read-before*: $\bigwedge r.\ r \in$ *read-actions E* $\Longrightarrow P,E \vdash ws\ r \le hb\ r$
shows *sequentially-consistent P* $(E,\ ws)$
⟨*proof*⟩

lemma *justified-action-committedD*:
assumes *justified*: $P \vdash (E,\ ws)$ *weakly-justified-by J*
and *a*: $a \in$ *actions E*
obtains $n\ a'$ **where** $a =$ *action-translation* $(J\ n)\ a'\ a' \in$ *committed* $(J\ n)$
⟨*proof*⟩

theorem *drf-weak*:
assumes *sync*: *correctly-synchronized P* $\mathcal{E}$
and *legal*: *weakly-legal-execution P* $\mathcal{E}$ $(E,\ ws)$
shows *sequentially-consistent P* $(E,\ ws)$
⟨*proof*⟩

corollary *drf*:
$[\![$ *correctly-synchronized P* $\mathcal{E}$; *legal-execution P* $\mathcal{E}$ $(E,\ ws)\ ]\!]$
$\Longrightarrow$ *sequentially-consistent P* $(E,\ ws)$
⟨*proof*⟩

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*⟩

theorem *SC-is-legal*:
assumes *E*: *E* ∈ ℰ
and *wf*: *P* ⊢ (*E*, *ws*) √
and *sc*: *sequentially-consistent P* (*E*, *ws*)
shows *legal-execution P* ℰ (*E*, *ws*)
⟨*proof*⟩

end

context *jmm-consistent* **begin**

theorem *consistent*:
assumes *E* ∈ ℰ *P* ⊢ (*E*, *ws*) √
shows ∃ *E* ∈ ℰ. ∃ *ws*. *legal-execution P* ℰ (*E*, *ws*)
⟨*proof*⟩

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-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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *w-values-WriteMemD*:
assumes *NormalAction* (*WriteMem ad al v*) ∈ *set obs*
shows *v* ∈ *w-values P vs0 obs* (*ad*, *al*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *w-value-mono*: *vs0 adal* ⊆ *w-value P vs0 ob adal*
⟨*proof*⟩

lemma *w-values-mono*: *vs0 adal* ⊆ *w-values P vs0 obs adal*
⟨*proof*⟩

lemma *w-value-greater*: *vs0* ≤ *w-value P vs0 ob*
⟨*proof*⟩

lemma *w-values-greater*: *vs0* ≤ *w-values P vs0 obs*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *non-speculative-coinduct-append* [*consumes 1*, *case-names non-speculative*, *case-conclusion non-speculative LNil lappend*]:
assumes *major*: *X vs obs*
and *step*: $\bigwedge$*vs obs. X vs obs*
$\Longrightarrow$ *obs* = *LNil* $\vee$
($\exists$ *obs′ obs″*. *obs* = *lappend obs′ obs″* $\wedge$ *obs′* $\neq$ *LNil* $\wedge$ *non-speculative P vs obs′* $\wedge$
(*lfinite obs′* $\longrightarrow$ (*X* (*w-values P vs* (*list-of obs′*)) *obs″* $\vee$
non-speculative P (*w-values P vs* (*list-of obs′*)) *obs″*)))
(**is** $\bigwedge$*vs obs*. - $\Longrightarrow$ - $\vee$ *?step vs obs*)
shows *non-speculative P vs obs*
⟨*proof*⟩

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*: $\bigwedge$*vs obs a. X vs obs a*
$\Longrightarrow$ *obs* = *LNil* $\vee$
($\exists$ *obs′ obs″ a′*. *obs* = *lappend obs′ obs″* $\wedge$ *non-speculative P vs obs′* $\wedge$ (*obs′* = *LNil* $\longrightarrow$ (*a′*, *a*) $\in$ *R*) $\wedge$
(*lfinite obs′* $\longrightarrow$ *X* (*w-values P vs* (*list-of obs′*)) *obs″ a′* $\vee$
non-speculative P (*w-values P vs* (*list-of obs′*)) *obs″*))
(**is** $\bigwedge$*vs obs a*. - $\Longrightarrow$ - $\vee$ *?step vs obs a*)
shows *non-speculative P vs obs*
⟨*proof*⟩

lemma *non-speculative-nthI*:
($\bigwedge$*i ad al v*.
⟦ *enat i* < *llength obs*; *lnth obs i* = *NormalAction* (*ReadMem ad al v*);
non-speculative P vs (*ltake* (*enat i*) *obs*) ⟧
$\Longrightarrow$ *v* $\in$ *w-values P vs* (*list-of* (*ltake* (*enat i*) *obs*)) (*ad*, *al*))
$\Longrightarrow$ *non-speculative P vs obs*
⟨*proof*⟩

locale *executions-sc-hb* =
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 $\mathcal{E}$*-ex-new-action*:
⟦ *E* $\in$ $\mathcal{E}$; *ra* $\in$ *read-actions E*; *adal* $\in$ *action-loc P E ra*; *non-speculative P* (λ-. {}) (*ltake* (*enat ra*) (*lmap snd E*)) ⟧
$\Longrightarrow$ $\exists$ *wa*. *wa* $\in$ *new-actions-for P E adal* $\wedge$ *wa* < *ra*
begin

lemma $\mathcal{E}$*-new-same-addr-singleton*:

assumes *E*: $E \in \mathcal{E}$
shows $\exists\, a.$ *new-actions-for P E adal* $\subseteq \{a\}$
⟨*proof*⟩

lemma *new-action-before-read*:
assumes *E*: $E \in \mathcal{E}$
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*
⟨*proof*⟩

lemma *most-recent-write-exists*:
assumes *E*: $E \in \mathcal{E}$
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*⟩

lemma *mrw-before*:
assumes *E*: $E \in \mathcal{E}$
and *mrw*: *P*,*E* ⊢ *r* ⇝*mrw w*
and *sc*: *non-speculative P* (λ-. {}) (*ltake* (*enat r*) (*lmap snd E*))
shows *w* < *r*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

abbreviation *mrw-values* ::
  'm prog ⇒ ((*'addr* × *addr-loc*) ⇀ (*'addr val* × *bool*)) ⇒ (*'addr*, *'thread-id*) *obs-event action list*
  ⇒ ((*'addr* × *addr-loc*) ⇀ (*'addr val* × *bool*))
where *mrw-values P* ≡ *foldl* (*mrw-value P*)

lemma *mrw-values-eq-SomeD*:
  assumes *mrw*: *mrw-values P vs0 obs* (*ad*, *al*) = ⌊(*v*, *b*)⌋
  and *vs0* (*ad*, *al*) = ⌊(*v*, *b*)⌋ ⟹ ∃ *wa*. *wa* ∈ *set obs* ∧ *is-write-action wa* ∧ (*ad*, *al*) ∈ *action-loc-aux P wa* ∧ (*b* ⟶ ¬ *is-new-action wa*)
  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-new-action wa* ⟷ ¬ *b*) ∧
    (∀ *ob*∈*set obs''*. *is-write-action ob* ⟶ (*ad*, *al*) ∈ *action-loc-aux P ob* ⟶ *is-new-action ob* ∧ *b*)
  (**is** *?concl obs*)
⟨*proof*⟩

lemma *mrw-values-WriteMemD*:
  assumes *NormalAction* (*WriteMem ad al v'*) ∈ *set obs*
  shows ∃ *v*. *mrw-values P vs0 obs* (*ad*, *al*) = *Some* (*v*, *True*)
⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *mrw-value-dom-mono*:
dom vs ⊆ *dom* (*mrw-value P vs ob*)
⟨*proof*⟩

lemma *mrw-values-dom-mono*:
dom vs ⊆ *dom* (*mrw-values P vs obs*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *llist-of-list-of-append*:
lfinite xs ⟹ *llist-of* (*list-of xs* @ *ys*) = *lappend xs* (*llist-of ys*)
⟨*proof*⟩

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*⟩

lemma *ta-seq-consist-mrw-before*:
assumes *sc*: *ta-seq-consist P Map.empty* (*lmap snd E*)
and *new-actions-for-fun*: $\bigwedge$*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*⟩

lemma *ta-seq-consist-imp-sequentially-consistent*:
assumes *tsa-ok*: *thread-start-actions-ok E*
and *new-actions-for-fun*: $\bigwedge$*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*⟩

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*)
(λ*ob 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*))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*: $\bigwedge a.\ \llbracket\ enat\ a < r;\ a \in read\text{-}actions\ E\ \rrbracket \Longrightarrow P,E \vdash a \leadsto mrw\ ws\ a$
  shows *ta-seq-consist P Map.empty (lmap snd (ltake r E))*
⟨*proof*⟩

end

context *jmm-multithreaded* **begin**

definition *complete-sc* :: $('l,'thread\text{-}id,'x,'m,'w)\ state \Rightarrow ('addr \times addr\text{-}loc \rightharpoonup 'addr\ val \times bool) \Rightarrow$
  $('thread\text{-}id \times ('l,\ 'thread\text{-}id,\ 'x,\ 'm,\ 'w,\ ('addr,\ 'thread\text{-}id)\ obs\text{-}event\ action)\ thread\text{-}action)\ llist$
where
  $complete\text{-}sc\ s\ vs = unfold\text{-}llist$
    $(\lambda(s,\ vs).\ \forall\ t\ ta\ s'.\ \neg\ s\ -t\triangleright ta\rightarrow\ s')$
    $(\lambda(s,\ vs).\ fst\ (SOME\ ((t,\ ta),\ s').\ s\ -t\triangleright ta\rightarrow\ s' \wedge ta\text{-}seq\text{-}consist\ P\ vs\ (llist\text{-}of\ \{\!|ta|\!\}_o)))$
    $(\lambda(s,\ vs).\ let\ ((t,\ ta),\ s') = SOME\ ((t,\ ta),\ s').\ s\ -t\triangleright ta\rightarrow\ s' \wedge ta\text{-}seq\text{-}consist\ P\ vs\ (llist\text{-}of\ \{\!|ta|\!\}_o)$
      $in\ (s',\ mrw\text{-}values\ P\ vs\ \{\!|ta|\!\}_o))$
    $(s,\ vs)$

definition *sc-completion* :: $('l,\ 'thread\text{-}id,\ 'x,\ 'm,\ 'w)\ state \Rightarrow ('addr \times addr\text{-}loc \rightharpoonup 'addr\ val \times bool)$
$\Rightarrow bool$
where
  $sc\text{-}completion\ s\ vs \longleftrightarrow$
   $(\forall\ ttas\ s'\ t\ x\ ta\ x'\ m'.$
     $s\ -\triangleright ttas\rightarrow*\ s' \longrightarrow ta\text{-}seq\text{-}consist\ P\ vs\ (llist\text{-}of\ (concat\ (map\ (\lambda(t,\ ta).\ \{\!|ta|\!\}_o)\ ttas))) \longrightarrow$
     $thr\ s'\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor \longrightarrow t \vdash (x,\ shr\ s')\ -ta\rightarrow\ (x',\ m') \longrightarrow actions\text{-}ok\ s'\ t\ ta \longrightarrow$
     $(\exists\ ta'\ x''\ m''.\ t \vdash (x,\ shr\ s')\ -ta'\rightarrow\ (x'',\ m'') \wedge actions\text{-}ok\ s'\ t\ ta' \wedge$
          $ta\text{-}seq\text{-}consist\ P\ (mrw\text{-}values\ P\ vs\ (concat\ (map\ (\lambda(t,\ ta).\ \{\!|ta|\!\}_o)\ ttas)))\ (llist\text{-}of$
$\{\!|ta'|\!\}_o)))$

lemma *sc-completionD*:
  $\llbracket\ sc\text{-}completion\ s\ vs;\ s\ -\triangleright ttas\rightarrow*\ s';\ ta\text{-}seq\text{-}consist\ P\ vs\ (llist\text{-}of\ (concat\ (map\ (\lambda(t,\ ta).\ \{\!|ta|\!\}_o)\ ttas)));$

    $thr\ s'\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor;\ t \vdash (x,\ shr\ s')\ -ta\rightarrow\ (x',\ m');\ actions\text{-}ok\ s'\ t\ ta\ \rrbracket$
  $\Longrightarrow \exists\ ta'\ x''\ m''.\ t \vdash (x,\ shr\ s')\ -ta'\rightarrow\ (x'',\ m'') \wedge actions\text{-}ok\ s'\ t\ ta' \wedge$
       $ta\text{-}seq\text{-}consist\ P\ (mrw\text{-}values\ P\ vs\ (concat\ (map\ (\lambda(t,\ ta).\ \{\!|ta|\!\}_o)\ ttas)))\ (llist\text{-}of\ \{\!|ta'|\!\}_o)$
⟨*proof*⟩

lemma *sc-completionI*:
  $(\bigwedge ttas\ s'\ t\ x\ ta\ x'\ m'.$
    $\llbracket\ s\ -\triangleright ttas\rightarrow*\ s';\ ta\text{-}seq\text{-}consist\ P\ vs\ (llist\text{-}of\ (concat\ (map\ (\lambda(t,\ ta).\ \{\!|ta|\!\}_o)\ ttas)));$
     $thr\ s'\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor;\ t \vdash (x,\ shr\ s')\ -ta\rightarrow\ (x',\ m');\ actions\text{-}ok\ s'\ t\ ta\ \rrbracket$
    $\Longrightarrow \exists\ ta'\ x''\ m''.\ t \vdash (x,\ shr\ s')\ -ta'\rightarrow\ (x'',\ m'') \wedge actions\text{-}ok\ s'\ t\ ta' \wedge$
       $ta\text{-}seq\text{-}consist\ P\ (mrw\text{-}values\ P\ vs\ (concat\ (map\ (\lambda(t,\ ta).\ \{\!|ta|\!\}_o)\ ttas)))\ (llist\text{-}of\ \{\!|ta'|\!\}_o))$
  $\Longrightarrow sc\text{-}completion\ s\ vs$
⟨*proof*⟩

lemma *sc-completion-shift*:
  assumes *sc-c*: *sc-completion s vs*
  and τRed: $s\ -\triangleright ttas\rightarrow*\ s'$
  and *sc*: $ta\text{-}seq\text{-}consist\ P\ vs\ (lconcat\ (lmap\ (\lambda(t,\ ta).\ llist\text{-}of\ \{\!|ta|\!\}_o)\ (llist\text{-}of\ ttas)))$
  shows $sc\text{-}completion\ s'\ (mrw\text{-}values\ P\ vs\ (concat\ (map\ (\lambda(t,\ ta).\ \{\!|ta|\!\}_o)\ ttas)))$

⟨*proof*⟩

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*⟩

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*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

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)))
⟨*proof*⟩

lemma *cut-and-update-imp-sc-completion*:
cut-and-update s vs $\Longrightarrow$ *sc-completion s vs*
⟨*proof*⟩

lemma *sequential-completion*:
assumes *cut-and-update*: *cut-and-update s vs*
and *ta-seq-consist-convert-RA*: $\bigwedge$*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*⟩

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* ⟧
$\Longrightarrow$ *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*⟩

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″*)
⟨*proof*⟩

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*⟩

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*⟩

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″*
⟨*proof*⟩

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*:
$\bigwedge w\ w'\ adal.\ [\![\ w \in new\text{-}actions\text{-}for\ P\ E''\ adal;$
$w' \in new\text{-}actions\text{-}for\ P\ E''\ adal\ ]\!] \Longrightarrow w = w'$
shows $\exists\, ws.\ P \vdash (E'',\ ws) \surd \wedge (\forall\, n.\ n \in read\text{-}actions\ E'' \longrightarrow length\ E' \leq n \longrightarrow P,E'' \vdash ws\ n \leq hb\ n) \wedge$
$(\forall\, n.\ n < length\ E' \longrightarrow ws\ n = ws'\ n)$
⟨*proof*⟩

lemma *ta-hb-consistent-not-ReadI*:
$(\bigwedge t\ ad\ al\ v.\ (t,\ NormalAction\ (ReadMem\ ad\ al\ v)) \notin lset\ E) \Longrightarrow ta\text{-}hb\text{-}consistent\ P\ E'\ E$
⟨*proof*⟩

context *jmm-multithreaded* **begin**

definition *complete-hb* :: $('l,'thread\text{-}id,'x,'m,'w)\ state \Rightarrow ('thread\text{-}id \times ('addr,\ 'thread\text{-}id)\ obs\text{-}event\ action)\ list$
$\Rightarrow ('thread\text{-}id \times ('l,\ 'thread\text{-}id,\ 'x,\ 'm,\ 'w,\ ('addr,\ 'thread\text{-}id)\ obs\text{-}event\ action)\ thread\text{-}action)\ llist$
where
$complete\text{-}hb\ s\ E = unfold\text{-}llist$
$(\lambda(s,\ E).\ \forall\, t\ ta\ s'.\ \neg\ s\ -t\triangleright ta\rightarrow\ s')$
$(\lambda(s,\ E).\ fst\ (SOME\ ((t,\ ta),\ s').\ s\ -t\triangleright ta\rightarrow\ s' \wedge ta\text{-}hb\text{-}consistent\ P\ E\ (llist\text{-}of\ (map\ (Pair\ t)\ \{\!|ta|\!\}_o))))$
$(\lambda(s,\ E).\ let\ ((t,\ ta),\ s') = SOME\ ((t,\ ta),\ s').\ s\ -t\triangleright ta\rightarrow\ s' \wedge ta\text{-}hb\text{-}consistent\ P\ E\ (llist\text{-}of\ (map\ (Pair\ t)\ \{\!|ta|\!\}_o))$
$in\ (s',\ E\ @\ map\ (Pair\ t)\ \{\!|ta|\!\}_o))$
$(s,\ E)$

definition *hb-completion* ::
$('l,\ 'thread\text{-}id,\ 'x,\ 'm,\ 'w)\ state \Rightarrow ('thread\text{-}id \times ('addr,\ 'thread\text{-}id)\ obs\text{-}event\ action)\ list \Rightarrow bool$
where
$hb\text{-}completion\ s\ E \longleftrightarrow$
$(\forall\, ttas\ s'\ t\ x\ ta\ x'\ m'\ i.$
$s\ -\triangleright ttas\rightarrow*\ s' \longrightarrow$
$non\text{-}speculative\ P\ (w\text{-}values\ P\ (\lambda\text{-}.\ \{\})\ (map\ snd\ E))\ (llist\text{-}of\ (concat\ (map\ (\lambda(t,\ ta).\ \{\!|ta|\!\}_o)\ ttas))) \longrightarrow$
$thr\ s'\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor \longrightarrow t \vdash (x,\ shr\ s')\ -ta\rightarrow\ (x',\ m') \longrightarrow actions\text{-}ok\ s'\ t\ ta \longrightarrow$
$non\text{-}speculative\ P\ (w\text{-}values\ P\ (w\text{-}values\ P\ (\lambda\text{-}.\ \{\})\ (map\ snd\ E))\ (concat\ (map\ (\lambda(t,\ ta).\ \{\!|ta|\!\}_o)\ ttas)))\ (llist\text{-}of\ (take\ i\ \{\!|ta|\!\}_o)) \longrightarrow$
$(\exists\, ta'\ x''\ m''.\ t \vdash (x,\ shr\ s')\ -ta'\rightarrow\ (x'',\ m'') \wedge actions\text{-}ok\ s'\ t\ ta' \wedge$
$take\ i\ \{\!|ta'|\!\}_o = take\ i\ \{\!|ta|\!\}_o \wedge$
$ta\text{-}hb\text{-}consistent\ P$
$(E\ @\ concat\ (map\ (\lambda(t,\ ta).\ map\ (Pair\ t)\ \{\!|ta|\!\}_o)\ ttas)\ @\ map\ (Pair\ t)\ (take\ i\ \{\!|ta|\!\}_o))$
$(llist\text{-}of\ (map\ (Pair\ t)\ (drop\ i\ \{\!|ta'|\!\}_o))) \wedge$
$(i < length\ \{\!|ta|\!\}_o \longrightarrow i < length\ \{\!|ta'|\!\}_o) \wedge$
$(if\ \exists\, ad\ al\ v.\ \{\!|ta|\!\}_o\ !\ i = NormalAction\ (ReadMem\ ad\ al\ v)\ then\ sim\text{-}action\ else\ (=))\ (\{\!|ta|\!\}_o\ !\ i)\ (\{\!|ta'|\!\}_o\ !\ i)))$

lemma *hb-completionD*:
$[\![\ hb\text{-}completion\ s\ E;\ s\ -\triangleright ttas\rightarrow*\ s';$
$non\text{-}speculative\ P\ (w\text{-}values\ P\ (\lambda\text{-}.\ \{\})\ (map\ snd\ E))\ (llist\text{-}of\ (concat\ (map\ (\lambda(t,\ ta).\ \{\!|ta|\!\}_o)\ ttas)));$

$thr\ s'\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor;\ t \vdash (x,\ shr\ s')\ -ta\rightarrow\ (x',\ m');\ actions\text{-}ok\ s'\ t\ ta;$
$non\text{-}speculative\ P\ (w\text{-}values\ P\ (w\text{-}values\ P\ (\lambda\text{-}.\ \{\})\ (map\ snd\ E))\ (concat\ (map\ (\lambda(t,\ ta).\ \{\!|ta|\!\}_o)$

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*) $\{\!|ta|\!\}_o$)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

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-. {}) = {}
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma
shows *start-addrs-allocated*: *allocated start-heap* = *set start-addrs*
and *distinct-start-addrs'*: *distinct start-addrs*
⟨*proof*⟩

lemma *w-addrs-start-heap-obs*: *w-addrs* (*w-values P vs* (*map NormalAction start-heap-obs*)) ⊆ *w-addrs vs*
⟨*proof*⟩

end

context *heap-base* **begin**

lemma *addr-loc-default-conf*:
P ⊢ *class-type-of CTn has F*:*T* (*fm*) *in C*
⟹ *P*,*h* ⊢ *addr-loc-default P CTn* (*CField C F*) :≤ *T*
⟨*proof*⟩

definition *vs-conf* :: *'m prog* ⇒ *'heap* ⇒ (*'addr* × *addr-loc* ⇒ *'addr val set*) ⇒ *bool*
where *vs-conf P h vs* ⟷ (∀ *ad al v*. *v* ∈ *vs* (*ad*, *al*) ⟶ (∃ *T*. *P*,*h* ⊢ *ad*@*al* : *T* ∧ *P*,*h* ⊢ *v* :≤ *T*))

lemma *vs-confI*:
(⋀*ad al v*. *v* ∈ *vs* (*ad*, *al*) ⟹ ∃ *T*. *P*,*h* ⊢ *ad*@*al* : *T* ∧ *P*,*h* ⊢ *v* :≤ *T*) ⟹ *vs-conf P h vs*
⟨*proof*⟩

lemma *vs-confD*:
⟦ *vs-conf P h vs*; *v* ∈ *vs* (*ad*, *al*) ⟧ ⟹ ∃ *T*. *P*,*h* ⊢ *ad*@*al* : *T* ∧ *P*,*h* ⊢ *v* :≤ *T*
⟨*proof*⟩

lemma *vs-conf-insert-iff*:
vs-conf P h (*vs*((*ad*, *al*) := *insert v* (*vs* (*ad*, *al*))))
⟷ *vs-conf P h vs* ∧ (∃ *T*. *P*,*h* ⊢ *ad*@*al* : *T* ∧ *P*,*h* ⊢ *v* :≤ *T*)
⟨*proof*⟩

end

context *heap* **begin**

lemma *vs-conf-hext*: ⟦ *vs-conf P h vs*; *h* ⊴ *h'* ⟧ ⟹ *vs-conf P h' vs*

⟨*proof*⟩

lemma *vs-conf-allocate*:
$\llbracket$ *vs-conf P h vs*; $(h', a) \in$ *allocate h hT*; *is-htype P hT* $\rrbracket$
$\Longrightarrow$ *vs-conf P h′* (*w-value P vs* (*NormalAction* (*NewHeapElem a hT*)))
⟨*proof*⟩

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* :: ($'heap \Rightarrow bool$) $\Rightarrow$ $'m$ *prog* $\Rightarrow$ *bool*
where *heap-read-typeable hconf P* $\longleftrightarrow$ ($\forall$ *h ad al v T*. *hconf h* $\longrightarrow$ *P*,*h* $\vdash$ *ad*@*al* : *T* $\longrightarrow$ *P*,*h* $\vdash$ *v* :$\leq$ *T* $\longrightarrow$ *heap-read h ad al v*)

lemma *heap-read-typeableI*:
($\bigwedge$*h ad al v T*. $\llbracket$ *P*,*h* $\vdash$ *ad*@*al* : *T*; *P*,*h* $\vdash$ *v* :$\leq$ *T*; *hconf h* $\rrbracket$ $\Longrightarrow$ *heap-read h ad al v*) $\Longrightarrow$ *heap-read-typeable hconf P*
⟨*proof*⟩

lemma *heap-read-typeableD*:
$\llbracket$ *heap-read-typeable hconf P*; *P*,*h* $\vdash$ *ad*@*al* : *T*; *P*,*h* $\vdash$ *v* :$\leq$ *T*; *hconf h* $\rrbracket$ $\Longrightarrow$ *heap-read h ad al v*
⟨*proof*⟩

end

context *heap-base* **begin**

definition *heap-read-typed* :: $'m$ *prog* $\Rightarrow$ $'heap$ $\Rightarrow$ $'addr$ $\Rightarrow$ *addr-loc* $\Rightarrow$ $'addr$ *val* $\Rightarrow$ *bool*
where *heap-read-typed P h ad al v* $\longleftrightarrow$ *heap-read h ad al v* $\wedge$ ($\forall$ *T*. *P*,*h* $\vdash$ *ad*@*al* : *T* $\longrightarrow$ *P*,*h* $\vdash$ *v* :$\leq$ *T*)

lemma *heap-read-typedI*:
$\llbracket$ *heap-read h ad al v*; $\bigwedge$*T*. *P*,*h* $\vdash$ *ad*@*al* : *T* $\Longrightarrow$ *P*,*h* $\vdash$ *v* :$\leq$ *T* $\rrbracket$ $\Longrightarrow$ *heap-read-typed P h ad al v*
⟨*proof*⟩

lemma *heap-read-typed-into-heap-read*:
heap-read-typed P h ad al v $\Longrightarrow$ *heap-read h ad al v*
⟨*proof*⟩

lemma *heap-read-typed-typed*:
$\llbracket$ *heap-read-typed P h ad al v*; *P*,*h* $\vdash$ *ad*@*al* : *T* $\rrbracket$ $\Longrightarrow$ *P*,*h* $\vdash$ *v* :$\leq$ *T*
⟨*proof*⟩

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*⟩

end

context *heap* **begin**

lemma *start-addrs-dom-w-values*:
assumes *wf*: *wf-syscls P*
and *a*: $a \in$ *set start-addrs*
and *adal*: *P*,*start-heap* ⊢ *a*@*al* : *T*
shows *w-values P* (λ-. {}) (*map NormalAction start-heap-obs*) (*a*, *al*) ≠ {}
⟨*proof*⟩

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
$enat\ m + n = enat\ k \longleftrightarrow k \geq m \wedge n = enat\ (k - m)$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *non-speculative-start-heap-obs*:
non-speculative P vs (*llist-of* (*map snd* (*lift-start-obs start-tid start-heap-obs*)))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*)))
⟨*proof*⟩

end

8.11.1 JMM traces for Jinja semantics

context *multithreaded-base* **begin**

inductive-set $\mathcal{E}$ :: ('*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'*) ∈ $\mathcal{E}$ σ

lemma *actions-*$\mathcal{E}$*E-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*⟩

lemma *actions-*$\mathcal{E}$*E*:
assumes *E*: *E* ∈ $\mathcal{E}$ 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*⟩

end

context *τmultithreaded-wf* **begin**

Alternative characterisation for $\mathcal{E}$

lemma *$\mathcal{E}$-conv-Runs*:
$\mathcal{E}\ \sigma = lconcat\ `\ lmap\ (\lambda(t,\ ta).\ llist\text{-}of\ (map\ (Pair\ t)\ \{\!|ta|\!\}_o))\ `\ llist\text{-}of\text{-}tllist\ `\ \{E.\ mthr.\tau Runs\ \sigma\ E\}$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

end

Running threads have been started before

definition *Status-no-wait-locks* :: $('l,'t,status \times 'x)\ thread\text{-}info \Rightarrow bool$
where
Status-no-wait-locks ts ⟷
$(\forall t\ status\ x\ ln.\ ts\ t = \lfloor((status,\ x),\ ln)\rfloor \longrightarrow status \neq Running \longrightarrow ln = no\text{-}wait\text{-}locks)$

lemma *Status-no-wait-locks-PreStartD*:
$\bigwedge ln.\ [\![\ Status\text{-}no\text{-}wait\text{-}locks\ ts;\ ts\ t = \lfloor((PreStart,\ x),\ ln)\rfloor\ ]\!] \Longrightarrow ln = no\text{-}wait\text{-}locks$
⟨*proof*⟩

lemma *Status-no-wait-locks-FinishedD*:
$\bigwedge ln.\ [\![\ Status\text{-}no\text{-}wait\text{-}locks\ ts;\ ts\ t = \lfloor((Finished,\ x),\ ln)\rfloor\ ]\!] \Longrightarrow ln = no\text{-}wait\text{-}locks$
⟨*proof*⟩

lemma *Status-no-wait-locksI*:
$(\bigwedge t\ status\ x\ ln.\ [\![\ ts\ t = \lfloor((status,\ x),\ ln)\rfloor;\ status = PreStart \vee status = Finished\ ]\!] \Longrightarrow ln = no\text{-}wait\text{-}locks)$
$\Longrightarrow Status\text{-}no\text{-}wait\text{-}locks\ ts$
⟨*proof*⟩

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)))
⟨*proof*⟩

end

context *multithreaded-base* **begin**

lemma *init-fin-preserve-Status-no-wait-locks*:
assumes *ok*: *Status-no-wait-locks (thr s)*
and *redT*: $multithreaded\text{-}base.redT\ init\text{-}fin\text{-}final\ init\text{-}fin\ (map\ NormalAction \circ convert\text{-}RA)\ s\ tta\ s'$
shows $Status\text{-}no\text{-}wait\text{-}locks\ (thr\ s')$
⟨*proof*⟩

lemma *init-fin-Running-InitialThreadAction*:
assumes *redT*: $multithreaded\text{-}base.redT\ init\text{-}fin\text{-}final\ init\text{-}fin\ (map\ NormalAction \circ convert\text{-}RA)\ s\ tta\ s'$
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 $tta = (t,\ \{\!|InitialThreadAction|\!\})$
⟨*proof*⟩

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'*)
⟨*proof*⟩

lemma *init-fin-Trsys-Running-InitialThreadAction*:
assumes *redT*: *if.mthr.Trsys s ttas s'*
and *not-running*: ⋀*x ln*. *thr s t* ≠ ⌊((*Running*, *x*), *ln*)⌋
and *running*: *thr s' t* = ⌊((*Running*, *x'*), *ln'*)⌋
shows (*t*, ⦃*InitialThreadAction*⦄) ∈ *set ttas*
⟨*proof*⟩

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*) ⇒ *'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*

sublocale *heap-multithreaded-base* < *mthr*: *if-multithreaded-base final r convert-RA*
⟨*proof*⟩

context *heap-multithreaded-base* **begin**

abbreviation *ℰ-start* ::
(*cname* ⇒ *mname* ⇒ *ty list* ⇒ *ty* ⇒ *'md* ⇒ *'addr val list* ⇒ *'x*)
⇒ *'md prog* ⇒ *cname* ⇒ *mname* ⇒ *'addr val list* ⇒ *status*
⇒ (*'thread-id* × (*'addr*, *'thread-id*) *obs-event action*) *llist set*
where
ℰ-start f P C M vs status ≡
lappend (*llist-of* (*lift-start-obs start-tid start-heap-obs*)) ‘
mthr.if.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*
⟨*proof*⟩

sublocale *heap-multithreaded* < *if*: *jmm-multithreaded*
mthr.*init-fin-final mthr*.*init-fin map NormalAction* ∘ *convert-RA P*
⟨*proof*⟩

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*⟩

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*⟩

end

In the subsequent locales, *convert-RA* refers to *convert-RA* and is no longer a parameter!

lemma *convert-RA-not-write*:
$\bigwedge ln.\ ob \in set\ (convert\text{-}RA\ ln) \Longrightarrow \neg\ is\text{-}write\text{-}action\ (NormalAction\ ob)$
⟨*proof*⟩

lemma *ta-seq-consist-convert-RA*:
fixes *ln* **shows**
$ta\text{-}seq\text{-}consist\ P\ vs\ (llist\text{-}of\ ((map\ NormalAction \circ convert\text{-}RA)\ ln))$
⟨*proof*⟩

lemma *ta-hb-consistent-convert-RA*:
$\bigwedge ln.\ ta\text{-}hb\text{-}consistent\ P\ E\ (llist\text{-}of\ (map\ (Pair\ t)\ ((map\ NormalAction \circ convert\text{-}RA)\ ln)))$
⟨*proof*⟩

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*: $t \vdash (x, m) -ta\to (x', m') \Longrightarrow allocated\ m \subseteq allocated\ m'$
and *red-New-allocatedD*:
$[\![\ t \vdash (x, m) -ta\to (x', m');\ NewHeapElem\ ad\ CTn \in set\ \{\!|ta|\!\}_o\ ]\!]$
$\Longrightarrow ad \in allocated\ m' \wedge ad \notin allocated\ m$
and *red-allocated-NewD*:
$[\![\ t \vdash (x, m) -ta\to (x', m');\ ad \in allocated\ m';\ ad \notin allocated\ m\ ]\!]$
$\Longrightarrow \exists\, CTn.\ NewHeapElem\ ad\ CTn \in set\ \{\!|ta|\!\}_o$
and *red-New-same-addr-same*:
$[\![\ t \vdash (x, m) -ta\to (x', m');$
$\{\!|ta|\!\}_o\ !\ i = NewHeapElem\ a\ CTn;\ i < length\ \{\!|ta|\!\}_o;$

$\{\!|ta|\!\}_o$! 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
⟨*proof*⟩

context *allocated-multithreaded* **begin**

lemma *redT-allocated-mono*:
assumes *mthr.redT* σ (t, *ta*) σ'
shows *allocated* (*shr* σ) ⊆ *allocated* (*shr* σ')
⟨*proof*⟩

lemma *RedT-allocated-mono*:
assumes *mthr.RedT* σ *ttas* σ'
shows *allocated* (*shr* σ) ⊆ *allocated* (*shr* σ')
⟨*proof*⟩

lemma *init-fin-allocated-mono*:
$t \vdash (x, m)$ −*ta*→*i* (x', m') ⟹ *allocated* m ⊆ *allocated* m'
⟨*proof*⟩

lemma *init-fin-redT-allocated-mono*:
assumes *mthr.if.redT* σ (t, *ta*) σ'
shows *allocated* (*shr* σ) ⊆ *allocated* (*shr* σ')
⟨*proof*⟩

lemma *init-fin-RedT-allocated-mono*:
assumes *mthr.if.RedT* σ *ttas* σ'
shows *allocated* (*shr* σ) ⊆ *allocated* (*shr* σ')
⟨*proof*⟩

lemma *init-fin-red-New-allocatedD*:
assumes $t \vdash (x, m)$ −*ta*→*i* (x', m') *NormalAction* (*NewHeapElem ad CTn*) ∈ *set* $\{\!|ta|\!\}_o$
shows *ad* ∈ *allocated* m' ∧ *ad* ∉ *allocated* m
⟨*proof*⟩

lemma *init-fin-red-allocated-NewD*:
assumes $t \vdash (x, m)$ −*ta*→*i* (x', m') *ad* ∈ *allocated* m' *ad* ∉ *allocated* m
shows ∃ *CTn*. *NormalAction* (*NewHeapElem ad CTn*) ∈ *set* $\{\!|ta|\!\}_o$
⟨*proof*⟩

lemma *init-fin-red-New-same-addr-same*:
assumes $t \vdash (x, m)$ −*ta*→*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 $i = j$
⟨*proof*⟩

lemma *init-fin-redT-allocated-NewHeapElemD*:
  assumes *mthr.if.redT s* (*t*, *ta*) *s'*
  and *ad* ∈ *allocated* (*shr s'*)
  and *ad* ∉ *allocated* (*shr s*)
  shows ∃ *CTn*. *NormalAction* (*NewHeapElem ad CTn*) ∈ *set* $\{\!|ta|\!\}_o$
⟨*proof*⟩

lemma *init-fin-RedT-allocated-NewHeapElemD*:
  assumes *mthr.if.RedT s ttas s'*
  and *ad* ∈ *allocated* (*shr s'*)
  and *ad* ∉ *allocated* (*shr s*)
  shows ∃ *t ta CTn*. (*t*, *ta*) ∈ *set ttas* ∧ *NormalAction* (*NewHeapElem ad CTn*) ∈ *set* $\{\!|ta|\!\}_o$
⟨*proof*⟩

lemma $\mathcal{E}$*-new-actions-for-unique*:
  assumes *E*: *E* ∈ $\mathcal{E}$*-start f P C M vs status*
  and *a*: *a* ∈ *new-actions-for P E adal*
  and *a'*: *a'* ∈ *new-actions-for P E adal*
  shows *a* = *a'*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *typeof-ka*: *typeof v* ≠ *None* ⟹ *ka-Val v* = {}
⟨*proof*⟩

lemma *ka-Val-undefined-value* [*simp*]:
  ka-Val undefined-value = {}
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *known-addrs-stateI*:
  ⋀*ln*. ⟦ *ad* ∈ *known-addrs t x*; *thr s t* = ⌊(*x*, *ln*)⌋ ⟧ ⟹ *ad* ∈ *known-addrs-state s*
⟨*proof*⟩

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* ⟨*proof*⟩

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*:
$t \vdash \langle x,\ m\rangle\ {-ta\rightarrow}\ \langle x',\ m'\rangle$
$\Longrightarrow$ *known-addrs t x*$'$ ⊆ *known-addrs t x* ∪ *new-obs-addrs* $\{\!|ta|\!\}_o$
and *red-known-addrs-new-thread*:
⟦ $t \vdash \langle x,\ m\rangle\ {-ta\rightarrow}\ \langle x',\ m'\rangle$; *NewThread t*$'$ *x*$''$ *m*$''$ ∈ *set* $\{\!|ta|\!\}_t$ ⟧
$\Longrightarrow$ *known-addrs t*$'$ *x*$''$ ⊆ *known-addrs t x*
and *red-read-knows-addr*:
⟦ $t \vdash \langle x,\ m\rangle\ {-ta\rightarrow}\ \langle x',\ m'\rangle$; *ReadMem ad al v* ∈ *set* $\{\!|ta|\!\}_o$ ⟧
$\Longrightarrow$ *ad* ∈ *known-addrs t x*
and *red-write-knows-addr*:
⟦ $t \vdash \langle x,\ m\rangle\ {-ta\rightarrow}\ \langle x',\ m'\rangle$; $\{\!|ta|\!\}_o$! *n* = *WriteMem ad al* (*Addr ad*$'$); *n* < *length* $\{\!|ta|\!\}_o$ ⟧
$\Longrightarrow$ *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 $t \vdash (x,\ m)\ {-ta\rightarrow}i\ (x',\ m')$
shows *known-addrs-if t x*$'$ ⊆ *known-addrs-if t x* ∪ *new-obs-addrs-if* $\{\!|ta|\!\}_o$
⟨*proof*⟩

lemma *if-red-known-addrs-new-thread*:
assumes $t \vdash (x,\ m)\ {-ta\rightarrow}i\ (x',\ m')$ *NewThread t*$'$ *x*$''$ *m*$''$ ∈ *set* $\{\!|ta|\!\}_t$
shows *known-addrs-if t*$'$ *x*$''$ ⊆ *known-addrs-if t x*
⟨*proof*⟩

lemma *if-red-read-knows-addr*:
assumes $t \vdash (x,\ m)\ {-ta\rightarrow}i\ (x',\ m')$ *NormalAction* (*ReadMem ad al v*) ∈ *set* $\{\!|ta|\!\}_o$
shows *ad* ∈ *known-addrs-if t x*
⟨*proof*⟩

lemma *if-red-write-knows-addr*:
assumes $t \vdash (x, m) -ta\rightarrow i\ (x', m')$
and $\{\!|ta|\!\}_o$! $n = NormalAction\ (WriteMem\ ad\ al\ (Addr\ ad'))$ $n < length\ \{\!|ta|\!\}_o$
shows $ad' \in known\text{-}addrs\text{-}if\ t\ x \vee ad' \in new\text{-}obs\text{-}addrs\text{-}if\ (take\ n\ \{\!|ta|\!\}_o)$
⟨*proof*⟩

lemma *if-redT-known-addrs-new*:
assumes *redT*: $mthr.if.redT\ s\ (t, ta)\ s'$
shows $if.known\text{-}addrs\text{-}state\ s' \subseteq if.known\text{-}addrs\text{-}state\ s \cup new\text{-}obs\text{-}addrs\text{-}if\ \{\!|ta|\!\}_o$
⟨*proof*⟩

lemma *if-redT-read-knows-addr*:
assumes *redT*: $mthr.if.redT\ s\ (t, ta)\ s'$
and *read*: $NormalAction\ (ReadMem\ ad\ al\ v) \in set\ \{\!|ta|\!\}_o$
shows $ad \in if.known\text{-}addrs\text{-}state\ s$
⟨*proof*⟩

lemma *init-fin-redT-known-addrs-subset*:
assumes $mthr.if.redT\ s\ (t, ta)\ s'$
shows $if.known\text{-}addrs\text{-}state\ s' \subseteq if.known\text{-}addrs\text{-}state\ s \cup known\text{-}addrs\text{-}if\ t\ (fst\ (the\ (thr\ s'\ t)))$
⟨*proof*⟩

lemma *w-values-no-write-unchanged*:
assumes *no-write*: $\bigwedge w.\ [\![\ w \in set\ obs;\ is\text{-}write\text{-}action\ w;\ adal \in action\text{-}loc\text{-}aux\ P\ w\ ]\!] \Longrightarrow False$
shows $w\text{-}values\ P\ vs\ obs\ adal = vs\ adal$
⟨*proof*⟩

lemma *redT-non-speculative-known-addrs-allocated*:
assumes *red*: $mthr.if.redT\ s\ (t, ta)\ s'$
and *tasc*: $non\text{-}speculative\ P\ vs\ (llist\text{-}of\ \{\!|ta|\!\}_o)$
and *ka*: $if.known\text{-}addrs\text{-}state\ s \subseteq allocated\ (shr\ s)$
and *vs*: $w\text{-}addrs\ vs \subseteq allocated\ (shr\ s)$
shows $if.known\text{-}addrs\text{-}state\ s' \subseteq allocated\ (shr\ s')$ (**is** *?thesis1*)
and $w\text{-}addrs\ (w\text{-}values\ P\ vs\ \{\!|ta|\!\}_o) \subseteq allocated\ (shr\ s')$ (**is** *?thesis2*)
⟨*proof*⟩

lemma *RedT-non-speculative-known-addrs-allocated*:
assumes *red*: $mthr.if.RedT\ s\ ttas\ s'$
and *tasc*: $non\text{-}speculative\ P\ vs\ (llist\text{-}of\ (concat\ (map\ (\lambda(t, ta).\ \{\!|ta|\!\}_o)\ ttas)))$
and *ka*: $if.known\text{-}addrs\text{-}state\ s \subseteq allocated\ (shr\ s)$
and *vs*: $w\text{-}addrs\ vs \subseteq allocated\ (shr\ s)$
shows $if.known\text{-}addrs\text{-}state\ s' \subseteq allocated\ (shr\ s')$ (**is** *?thesis1 s'*)
and $w\text{-}addrs\ (w\text{-}values\ P\ vs\ (concat\ (map\ (\lambda(t, ta).\ \{\!|ta|\!\}_o)\ ttas))) \subseteq allocated\ (shr\ s')$ (**is** *?thesis2 s' ttas*)
⟨*proof*⟩

lemma *read-ex-NewHeapElem* [*consumes 5*, *case-names start Red*]:
assumes *RedT*: $mthr.if.RedT\ (init\text{-}fin\text{-}lift\text{-}state\ status\ (start\text{-}state\ f\ P\ C\ M\ vs))\ ttas\ s$
and *red*: $mthr.if.redT\ s\ (t, ta)\ s'$
and *read*: $NormalAction\ (ReadMem\ ad\ al\ v) \in set\ \{\!|ta|\!\}_o$
and *sc*: $non\text{-}speculative\ P\ (\lambda\text{-}.\ \{\})\ (llist\text{-}of\ (map\ snd\ (lift\text{-}start\text{-}obs\ start\text{-}tid\ start\text{-}heap\text{-}obs)\ @\ concat\ (map\ (\lambda(t, ta).\ \{\!|ta|\!\}_o)\ ttas)))$

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*⟩

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*:
$\llbracket$ $t \vdash (x,\ m)\ -ta\rightarrow\ (x',\ m')$; *wfx t x m*; *ReadMem ad al v* $\in$ *set* $\{\!\!\{ta\}\!\!\}_o$ $\rrbracket$
$\Longrightarrow \exists\, T.\ P{,}m \vdash ad@al : T$
and *red-NewHeapElemD*:
$\llbracket$ $t \vdash (x,\ m)\ -ta\rightarrow\ (x',\ m')$; *wfx t x m*; *NewHeapElem ad hT* $\in$ *set* $\{\!\!\{ta\}\!\!\}_o$ $\rrbracket$
$\Longrightarrow$ *typeof-addr* m' *ad* $= \lfloor hT \rfloor$
and *red-hext-incr*:
$\llbracket$ $t \vdash (x,\ m)\ -ta\rightarrow\ (x',\ m')$; *wfx t x m*;
vs-conf P m vs; *non-speculative P vs* (*llist-of* (*map NormalAction* $\{\!\!\{ta\}\!\!\}_o$)) $\rrbracket$
$\Longrightarrow m \trianglelefteq m'$
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')
⟨*proof*⟩

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* $\{\!\!\{ta\}\!\!\}_o$)))
⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *init-fin-hext-incr*:

assumes $t \vdash (x, m) -ta\rightarrow i (x', m')$
and *init-fin-lift wfx t x m*
and *non-speculative P vs* (*llist-of* $\{\!|ta|\!\}_o$)
and *vs-conf P m vs*
shows $m \trianglelefteq m'$
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *init-fin-red-read-typeable*:
assumes $t \vdash (x, m) -ta\rightarrow i (x', m')$
and *init-fin-lift wfx t x m NormalAction* (*ReadMem ad al v*) ∈ *set* $\{\!|ta|\!\}_o$
shows ∃ *T*. *P*,*m* ⊢ *ad*@*al* : *T*
⟨*proof*⟩

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* ∈ 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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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* ≠ {}}
⟨*proof*⟩

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* ≠ {}}
⟨*proof*⟩

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' ∈ *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*⦄$_o$))
  ⟹ *non-speculative-read n s vs*
⟨*proof*⟩

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' ∈ *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*⦄$_o$)
⟨*proof*⟩

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 $E \equiv$ *lift-start-obs start-tid start-heap-obs*
and *vs* $\equiv$ *w-values P* (λ-. {}) (*map snd E*)
and *s* $\equiv$ *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*) $\in$ *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*) $\subseteq$ *allocated start-heap*
and *wt*: *ts-ok* (*init-fin-lift wfx*) (*thr s*) (*shr s*)
and *type-adal*: *P*,*shr s'* $\vdash$ *ad*@*al* : *T*
shows $\exists$ *w*. *w* $\in$ *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*)
⟨*proof*⟩

lemma *non-speculative-read-into-cut-and-update*:
fixes *status f C M params E*
defines $E \equiv$ *lift-start-obs start-tid start-heap-obs*
and *vs* $\equiv$ *w-values P* (λ-. {}) (*map snd E*)
and *s* $\equiv$ *init-fin-lift-state status* (*start-state f P C M params*)
and *vs'* $\equiv$ *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*) $\subseteq$ *allocated start-heap*
shows *if.cut-and-update s vs'*
⟨*proof*⟩

lemma *non-speculative-read-into-hb-completion*:
fixes *status f C M params E*
defines $E \equiv$ *lift-start-obs start-tid start-heap-obs*
and *vs* $\equiv$ *w-values P* (λ-. {}) (*map snd E*)
and *s* $\equiv$ *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*) $\subseteq$ *allocated start-heap*
shows *if.hb-completion s E*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩
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*
⟨*proof*⟩

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*
⟨*proof*⟩
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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *w-addrs-vs-type-all*: *w-addrs* (*vs-type-all P*) ⊆ *dom typeof-addr*
⟨*proof*⟩

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* ≠ {}}
⟨*proof*⟩

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'* ⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

end

lemma *lprefix-lappend2I*: *lprefix xs ys* ⟹ *lprefix xs* (*lappend ys zs*)
⟨*proof*⟩

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'* ⟨*proof*⟩

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* ≠ {}}
⟨*proof*⟩

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*⌋
⟨*proof*⟩

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*⌋
⟨*proof*⟩

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.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*⟩

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*
⟨*proof*⟩

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*⟩

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.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*⟩

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.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*
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *heap-copies-not-New*:
assumes *heap-copies a a′ als h obs h′*
and *NewHeapElem a″ x* ∈ *set obs*
shows *False*
⟨*proof*⟩

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 $i = j$
⟨*proof*⟩

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*⦄$_o$ ⟧
⟹ $i = j$
⟨*proof*⟩

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*⦄$_o$ ⟧
⟹ $i = j$
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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′*)
⟨*proof*⟩

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′*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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* (λ*al T*. *P*,*h* ⊢ *ad*@*al* : *T*) *als Ts list-all2* (λ*al 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′*
⟨*proof*⟩

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*)))
⟨*proof*⟩

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′*)⌋
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*
⟨*proof*⟩

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′*
⟨*proof*⟩

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* $\{\!|ta|\!\}_o$)))
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *red-external-aggr-non-speculative-vs-conf*:
assumes *red*: $(ta, va, h') \in$ *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* $\{\!|ta|\!\}_o$)))
⟨*proof*⟩

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*: $P,h \vdash a@al : T$ $P,h \vdash a'@al : T$
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*: $v' \in$ *w-values P vs* (*map NormalAction* (*take i obs*)) (a'', al'')
shows $\exists$ *obs' h''*. *heap-copy-loc a a' al 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* $\wedge$
non-speculative P vs (*llist-of* (*map NormalAction obs'*))
⟨*proof*⟩

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* ($\lambda al\ T.\ P,h \vdash a@al : T$) *als Ts*
and *type2*: *list-all2* ($\lambda al\ T.\ P,h \vdash a'@al : T$) *als Ts*
and *hconf*: *hconf h*
and *i*: $i <$ *length obs*
and *read*: *obs* ! i = *ReadMem a'' al'' v*
and *v*: $v' \in$ *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 $\exists$ *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*
(**is** *?concl als h obs vs i*)
⟨*proof*⟩

lemma *heap-clone-non-speculative-read*:
assumes *hrt*: *heap-read-typeable hconf P*
and *clone*: *heap-clone P h a h'* $\lfloor(obs, a')\rfloor$
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*: $v' \in w\text{-}values\ P\ vs\ (map\ NormalAction\ (take\ i\ obs))\ (a'',\ al'')$
and *ns*: $non\text{-}speculative\ P\ vs\ (llist\text{-}of\ (map\ NormalAction\ (take\ i\ obs)))$
shows $\exists\, obs'\ h''.\ heap\text{-}clone\ P\ h\ a\ h''\ \lfloor(obs',\ a')\rfloor \wedge i < length\ obs' \wedge take\ i\ obs' = take\ i\ obs \wedge obs'\ !\ i = ReadMem\ a''\ al''\ v' \wedge length\ obs' \leq length\ obs$
⟨*proof*⟩

lemma *red-external-non-speculative-read*:
assumes *hrt*: *heap-read-typeable hconf P*
and *vs*: *vs-conf P* (*shr s*) *vs*
and *red*: $P,t \vdash \langle a{\bullet}M(vs'),\ shr\ s\rangle\ -ta\rightarrow ext\ \langle va,h'\rangle$
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*: $v' \in w\text{-}values\ P\ vs\ (map\ NormalAction\ (take\ i\ \{\!|ta|\!\}_o))\ (a'',\ al'')$
and *ns*: $non\text{-}speculative\ P\ vs\ (llist\text{-}of\ (map\ NormalAction\ (take\ i\ \{\!|ta|\!\}_o)))$
shows $\exists\, ta''\ va''\ h''.\ P,t \vdash \langle a{\bullet}M(vs'),\ shr\ s\rangle\ -ta''\rightarrow ext\ \langle va'',\ h''\rangle \wedge final\text{-}thread.actions\text{-}ok\ final\ s\ t\ ta'' \wedge$
$i < length\ \{\!|ta''|\!\}_o \wedge take\ i\ \{\!|ta''|\!\}_o = take\ i\ \{\!|ta|\!\}_o \wedge$
$\{\!|ta''|\!\}_o\ !\ i = ReadMem\ a''\ al''\ v' \wedge length\ \{\!|ta''|\!\}_o \leq length\ \{\!|ta|\!\}_o$
⟨*proof*⟩

lemma *red-external-aggr-non-speculative-read*:
assumes *hrt*: *heap-read-typeable hconf P*
and *vs*: *vs-conf P* (*shr s*) *vs*
and *red*: $(ta,\ va,\ h') \in red\text{-}external\text{-}aggr\ P\ t\ a\ M\ vs'\ (shr\ s)$
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*: $v' \in w\text{-}values\ P\ vs\ (map\ NormalAction\ (take\ i\ \{\!|ta|\!\}_o))\ (a'',\ al'')$
and *ns*: $non\text{-}speculative\ P\ vs\ (llist\text{-}of\ (map\ NormalAction\ (take\ i\ \{\!|ta|\!\}_o)))$
shows $\exists\, ta''\ va''\ h''.\ (ta'',\ va'',\ h'') \in red\text{-}external\text{-}aggr\ P\ t\ a\ M\ vs'\ (shr\ s) \wedge final\text{-}thread.actions\text{-}ok\ final\ s\ t\ ta'' \wedge$
$i < length\ \{\!|ta''|\!\}_o \wedge take\ i\ \{\!|ta''|\!\}_o = take\ i\ \{\!|ta|\!\}_o \wedge$
$\{\!|ta''|\!\}_o\ !\ i = ReadMem\ a''\ al''\ v' \wedge length\ \{\!|ta''|\!\}_o \leq length\ \{\!|ta|\!\}_o$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *heap-copy-loc-allocated-mono*:

heap-copy-loc a a′ al h obs h′ $\Longrightarrow$ *allocated h* $\subseteq$ *allocated h′*
⟨*proof*⟩

lemma *heap-copies-allocated-same*:
assumes *heap-copies a a′ al h obs h′*
shows *allocated h′* = *allocated h*
⟨*proof*⟩

lemma *heap-copies-allocated-mono*:
heap-copies a a′ al h obs h′ $\Longrightarrow$ *allocated h* $\subseteq$ *allocated h′*
⟨*proof*⟩

lemma *heap-clone-allocated-mono*:
assumes *heap-clone P h a h′ aobs*
shows *allocated h* $\subseteq$ *allocated h′*
⟨*proof*⟩

lemma *red-external-allocated-mono*:
assumes *P*,*t* ⊢ ⟨*a*•*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h′*⟩
shows *allocated h* $\subseteq$ *allocated h′*
⟨*proof*⟩

lemma *red-external-aggr-allocated-mono*:
⟦ (*ta*, *va*, *h′*) ∈ *red-external-aggr P t a M vs h*; *is-native P* (*the* (*typeof-addr h a*)) *M* ⟧
$\Longrightarrow$ *allocated h* $\subseteq$ *allocated h′*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *red-external-allocatedD*:
⟦ *P*,*t* ⊢ ⟨*a*•*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h′*⟩; *NewHeapElem a′ x* ∈ *set* ⦃*ta*⦄$_o$ ⟧
$\Longrightarrow$ *a′* ∈ *allocated h′* ∧ *a′* ∉ *allocated h*
⟨*proof*⟩

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* ⟧
$\Longrightarrow$ *a′* ∈ *allocated h′* ∧ *a′* ∉ *allocated h*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *heap-clone-fail-allocated-same*:
assumes *heap-clone P h a h′ None*
shows *allocated h′* = *allocated h*

⟨*proof*⟩

lemma *red-external-NewHeapElemD*:
  $[\![P,t \vdash \langle a{\cdot}M(vs),\ h\rangle\ {-ta\rightarrow}ext\ \langle va,\ h'\rangle;\ a' \in allocated\ h';\ a' \notin allocated\ h\ ]\!]$
  $\Longrightarrow \exists\, CTn.\ NewHeapElem\ a'\ CTn \in set\ \{\!|ta|\!\}_o$
⟨*proof*⟩

lemma *red-external-aggr-NewHeapElemD*:
  $[\![(ta,\ va,\ h') \in red\text{-}external\text{-}aggr\ P\ t\ a\ M\ vs\ h;\ a' \in allocated\ h';\ a' \notin allocated\ h;$
    $is\text{-}native\ P\ (the\ (typeof\text{-}addr\ h\ a))\ M\ ]\!]$
  $\Longrightarrow \exists\, CTn.\ NewHeapElem\ a'\ CTn \in set\ \{\!|ta|\!\}_o$
⟨*proof*⟩

end

context *heap-base* **begin**

lemma *binop-known-addrs*:
  assumes *ok*: *start-heap-ok*
  shows $binop\ bop\ v1\ v2 = \lfloor Inl\ v\rfloor \Longrightarrow ka\text{-}Val\ v \subseteq ka\text{-}Val\ v1 \cup ka\text{-}Val\ v2 \cup set\ start\text{-}addrs$
  and $binop\ bop\ v1\ v2 = \lfloor Inr\ a\rfloor \Longrightarrow a \in ka\text{-}Val\ v1 \cup ka\text{-}Val\ v2 \cup set\ start\text{-}addrs$
⟨*proof*⟩

lemma *heap-copy-loc-known-addrs-ReadMem*:
  assumes $heap\text{-}copy\text{-}loc\ a\ a'\ al\ h\ ob\ h'$
  and $ReadMem\ ad\ al'\ v \in set\ ob$
  shows $ad = a$
⟨*proof*⟩

lemma *heap-copies-known-addrs-ReadMem*:
  assumes $heap\text{-}copies\ a\ a'\ als\ h\ obs\ h'$
  and $ReadMem\ ad\ al\ v \in set\ obs$
  shows $ad = a$
⟨*proof*⟩

lemma *heap-clone-known-addrs-ReadMem*:
  assumes $heap\text{-}clone\ P\ h\ a\ h'\ \lfloor(obs,\ a')\rfloor$
  and $ReadMem\ ad\ al\ v \in set\ obs$
  shows $ad = a$
⟨*proof*⟩

lemma *red-external-known-addrs-ReadMem*:
  $[\![P,t \vdash \langle a{\cdot}M(vs),\ h\rangle\ {-ta\rightarrow}ext\ \langle va,h'\rangle;\ ReadMem\ ad\ al\ v \in set\ \{\!|ta|\!\}_o\ ]\!]$
  $\Longrightarrow ad \in \{thread\text{-}id2addr\ t,\ a\} \cup (\bigcup (ka\text{-}Val\ `\ set\ vs)) \cup set\ start\text{-}addrs$
⟨*proof*⟩

lemma *red-external-aggr-known-addrs-ReadMem*:
  $[\![(ta,\ va,\ h') \in red\text{-}external\text{-}aggr\ P\ t\ a\ M\ vs\ h;\ ReadMem\ ad\ al\ v \in set\ \{\!|ta|\!\}_o\ ]\!]$
  $\Longrightarrow ad \in \{thread\text{-}id2addr\ t,\ a\} \cup (\bigcup (ka\text{-}Val\ `\ set\ vs)) \cup set\ start\text{-}addrs$
⟨*proof*⟩

lemma *heap-copy-loc-known-addrs-WriteMem*:
  assumes $heap\text{-}copy\text{-}loc\ a\ a'\ al\ h\ ob\ h'$
  and $ob\ !\ n = WriteMem\ ad\ al'\ (Addr\ a'')\ n < length\ ob$

shows $a'' \in$ *new-obs-addrs* (*take n ob*)
⟨*proof*⟩

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 $a'' \in$ *new-obs-addrs* (*take n obs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *red-external-known-addrs-WriteMem*:
⟦ *P*,*t* ⊢ ⟨*a*•*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*,*h′*⟩; ⦃*ta*⦄$_o$! *n* = *WriteMem ad al* (*Addr a′*); *n* < *length* ⦃*ta*⦄$_o$ ⟧
⟹ *a′* ∈ {*thread-id2addr t*, *a*} ∪ (⋃(*ka-Val* ‘ *set vs*)) ∪ *set start-addrs* ∪ *new-obs-addrs* (*take n* ⦃*ta*⦄$_o$)
⟨*proof*⟩

lemma *red-external-aggr-known-addrs-WriteMem*:
⟦ (*ta*, *va*, *h′*) ∈ *red-external-aggr P t a M vs h*;
⦃*ta*⦄$_o$! *n* = *WriteMem ad al* (*Addr a′*); *n* < *length* ⦃*ta*⦄$_o$ ⟧
⟹ *a′* ∈ {*thread-id2addr t*, *a*} ∪ (⋃(*ka-Val* ‘ *set vs*)) ∪ *set start-addrs* ∪ *new-obs-addrs* (*take n* ⦃*ta*⦄$_o$)
⟨*proof*⟩

lemma *red-external-known-addrs-mono*:
assumes *ok*: *start-heap-ok*
and *red*: *P*,*t* ⊢ ⟨*a*•*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h′*⟩
shows (*case va of RetVal v* ⇒ *ka-Val v* | *RetExc a* ⇒ {*a*} | *RetStaySame* ⇒ {}) ⊆ {*thread-id2addr t*, *a*} ∪ (⋃(*ka-Val* ‘ *set vs*)) ∪ *set start-addrs* ∪ *new-obs-addrs* ⦃*ta*⦄$_o$
⟨*proof*⟩

lemma *red-external-aggr-known-addrs-mono*:
assumes *ok*: *start-heap-ok*
and *red*: (*ta*, *va*, *h′*) ∈ *red-external-aggr P t a M vs h is-native P* (*the* (*typeof-addr h a*)) *M*
shows (*case va of RetVal v* ⇒ *ka-Val v* | *RetExc a* ⇒ {*a*} | *RetStaySame* ⇒ {}) ⊆ {*thread-id2addr t*, *a*} ∪ (⋃(*ka-Val* ‘ *set vs*)) ∪ *set start-addrs* ∪ *new-obs-addrs* ⦃*ta*⦄$_o$
⟨*proof*⟩

lemma *red-external-NewThread-idD*:
⟦ *P*,*t* ⊢ ⟨*a*•*M*(*vs*), *h*⟩ −*ta*→*ext* ⟨*va*, *h′*⟩; *NewThread t′* (*C*, *M′*, *a′*) *h′′* ∈ *set* ⦃*ta*⦄$_t$ ⟧
⟹ *t′* = *addr2thread-id a* ∧ *a′* = *a*
⟨*proof*⟩

lemma *red-external-aggr-NewThread-idD*:
⟦ (*ta*, *va*, *h′*) ∈ *red-external-aggr P t a M vs h*;
NewThread t′ (*C*, *M′*, *a′*) *h′′* ∈ *set* ⦃*ta*⦄$_t$ ⟧
⟹ *t′* = *addr2thread-id a* ∧ *a′* = *a*
⟨*proof*⟩

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*⌋
⟨*proof*⟩

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*⌋
⟨*proof*⟩

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*⌋
⟨*proof*⟩

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*⌋
⟨*proof*⟩

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*⌋
⟨*proof*⟩

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*
⟨*proof*⟩

context *J-heap-base* **begin**

abbreviation *J-ℰ* ::
'addr J-prog ⇒ *cname* ⇒ *mname* ⇒ *'addr val list* ⇒ *status*
⇒ (*'thread-id* × (*'addr*, *'thread-id*) *obs-event action*) *llist set*
where
J-ℰ P ≡ *red-mthr.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*
⟨*proof*⟩

lemma *ka-locals-update-subset*:
  ka-locals (*xs*(*V* := *None*)) ⊆ *ka-locals xs*
  ka-locals (*xs*(*V* ↦ *v*)) ⊆ *ka-Val v* ∪ *ka-locals xs*
⟨*proof*⟩

lemma *ka-locals-empty* [*simp*]: *ka-locals Map.empty* = {}
⟨*proof*⟩

lemma *kas-append* [*simp*]: *kas* (*es* @ *es'*) = *kas es* ∪ *kas es'*
⟨*proof*⟩

lemma *kas-map-Val* [*simp*]: *kas* (*map Val vs*) = ⋃(*ka-Val* ‘ *set vs*)
⟨*proof*⟩

lemma *ka-blocks*:
  ⟦ *length pns* = *length Ts*; *length vs* = *length Ts* ⟧
  ⟹ *ka* (*blocks pns Ts vs body*) = ⋃(*ka-Val* ‘ *set vs*) ∪ *ka body*
⟨*proof*⟩

lemma *WT-ka*: *P*,*E* ⊢ *e* :: *T* ⟹ *ka e* = {}
  and *WTs-kas*: *P*,*E* ⊢ *es* [::] *Ts* ⟹ *kas es* = {}
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*′)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⦄$_o$
⟨*proof*⟩

lemma *mred-allocated-multithreaded*:
allocated-multithreaded addr2thread-id thread-id2addr empty-heap allocate typeof-addr heap-write allocated final-expr (*mred P*) *P*
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

end

context *J-heap* **begin**

lemma *red-read-typeable*:
  $\llbracket$ *convert-extTA extTA,P,t* $\vdash \langle e,\ s\rangle\ -ta\rightarrow\ \langle e',\ s'\rangle$; *P,E,hp s* $\vdash$ *e* : *T*; *ReadMem ad al v* $\in$ *set* $\{\!\!\{ta\}\!\!\}_o$ $\rrbracket$
  $\Longrightarrow \exists\, T'.$ *P,hp s* $\vdash$ *ad@al* : T'
  and *reds-read-typeable*:
  $\llbracket$ *convert-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* $\{\!\!\{ta\}\!\!\}_o$ $\rrbracket$
  $\Longrightarrow \exists\, T'.$ *P,hp s* $\vdash$ *ad@al* : T'
⟨*proof*⟩

end

primrec *new-types* :: (*'a*, *'b*, *'addr*) *exp* ⇒ *ty set*
  and *new-typess* :: (*'a*, *'b*, *'addr*) *exp list* ⇒ *ty set*
where
  new-types (*new C*) = {*Class C*}
| *new-types* (*newA T*⌊*e*⌉) = *insert* (*T*⌊⌉) (*new-types e*)
| *new-types* (*Cast T e*) = *new-types e*
| *new-types* (*e instanceof T*) = *new-types e*
| *new-types* (*Val v*) = {}
| *new-types* (*Var V*) = {}
| *new-types* (*e1* «*bop*» *e2*) = *new-types e1* ∪ *new-types e2*
| *new-types* (*V* := *e*) = *new-types e*
| *new-types* (*a*⌊*e*⌉) = *new-types a* ∪ *new-types e*
| *new-types* (*a*⌊*e*⌉ := *e'*) = *new-types a* ∪ *new-types e* ∪ *new-types e'*
| *new-types* (*a*•*length*) = *new-types a*
| *new-types* (*e*•*F*{*D*}) = *new-types e*
| *new-types* (*e*•*F*{*D*} := *e'*) = *new-types e* ∪ *new-types e'*
| *new-types* (*e*•*compareAndSwap*(*D*•*F*, *e'*, *e''*)) = *new-types e* ∪ *new-types e'* ∪ *new-types e''*
| *new-types* (*e*•*M*(*es*)) = *new-types e* ∪ *new-typess es*
| *new-types* {*V*:*T*=*vo*; *e*} = *new-types e*
| *new-types* (*Synchronized x e e'*) = *new-types e* ∪ *new-types e'*
| *new-types* (*InSynchronized x a e*) = *new-types e*
| *new-types* (*e*;; *e'*) = *new-types e* ∪ *new-types e'*
| *new-types* (*if* (*e*) *e1 else e2*) = *new-types e* ∪ *new-types e1* ∪ *new-types e2*
| *new-types* (*while* (*b*) *e*) = *new-types b* ∪ *new-types e*
| *new-types* (*throw e*) = *new-types e*
| *new-types* (*try e catch*(*C V*) *e'*) = *new-types e* ∪ *new-types e'*

| *new-typess* [] = {}
| *new-typess* (*e* # *es*) = *new-types e* ∪ *new-typess es*

lemma *new-types-blocks*:
  ⟦ *length pns* = *length Ts*; *length vs* = *length Ts* ⟧ ⟹ *new-types* (*blocks pns vs Ts e*) = *new-types e*
⟨*proof*⟩

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*
⟨*proof*⟩

end

lemma *WT-new-types-types*: $P,E \vdash e :: T \Longrightarrow$ *new-types* $e \subseteq$ *types* P
and *WTs-new-typess-types*: $P,E \vdash es$ [::] $Ts \Longrightarrow$ *new-typess* $es \subseteq$ *types* P
⟨*proof*⟩

context *J-heap-conf* **begin**

lemma *red-New-typeof-addrD*:
⟦ *convert-extTA extTA*,P,$t \vdash \langle e, s\rangle$ −*ta*→ $\langle e', s'\rangle$; *new-types* $e \subseteq$ *types* P; *hconf* (*hp* s); *NewHeapElem* a $x \in$ *set* ⦃*ta*⦄$_o$ ⟧
$\Longrightarrow$ *typeof-addr* (*hp* s') a = *Some* x
and *reds-New-typeof-addrD*:
⟦ *convert-extTA extTA*,P,$t \vdash \langle es, s\rangle$ [−*ta*→] $\langle es', s'\rangle$; *new-typess* $es \subseteq$ *types* P; *hconf* (*hp* s); *NewHeapElem* a $x \in$ *set* ⦃*ta*⦄$_o$ ⟧
$\Longrightarrow$ *typeof-addr* (*hp* s') a = *Some* x
⟨*proof*⟩

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*⟩

lemma *red-non-speculative-vs-conf*:
⟦ *convert-extTA extTA*,P,$t \vdash \langle e, s\rangle$ −*ta*→ $\langle e', s'\rangle$; $P,E,hp\ s \vdash e : T$;
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*⦄$_o$)))
and *reds-non-speculative-vs-conf*:
⟦ *convert-extTA extTA*,P,$t \vdash \langle es, s\rangle$ [−*ta*→] $\langle es', s'\rangle$; $P,E,hp\ s \vdash es$ [:] 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*⟩

lemma *red-non-speculative-typeable*:
⟦ *convert-extTA extTA*,P,$t \vdash \langle e, s\rangle$ −*ta*→ $\langle e', s'\rangle$; $P,E,hp\ s \vdash e : T$;
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,$t \vdash \langle es, s\rangle$ [−*ta*→] $\langle es', s'\rangle$; $P,E,hp\ s \vdash es$ [:] 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*⟩

end

sublocale *J-heap-base* < *red-mthr*:
if-multithreaded
final-expr
mred P
convert-RA

for *P*
⟨*proof*⟩

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*⟩

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*: ⋃(*ka-Val* ‘ *set vs*) ⊆ *set start-addrs*
shows *executions-sc-hb* (*J-ℰ P C M vs status*) *P*
(**is** *executions-sc-hb ?E P*)
⟨*proof*⟩

end

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

context *J-progress* **begin**

lemma *ex-WTrt-simps*:
$P,E,h \vdash e : T \Longrightarrow \exists E\ T.\ P,E,h \vdash e : T$
⟨*proof*⟩

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*:
$[\![P,t \vdash \langle e, (shr\ s,\ xs)\rangle -ta\rightarrow \langle e', (h', xs')\rangle; \exists E\ T.\ P,E,shr\ s \vdash e : T;$
red-mthr.mthr.if.actions-ok s t ta;
$I < length\ \{\!|ta|\!\}_o;\ \{\!|ta|\!\}_o\ !\ I = ReadMem\ a''\ al''\ v;\ v' \in w\text{-}values\ P\ vs\ (map\ NormalAction\ (take\ I\ \{\!|ta|\!\}_o))\ (a'',\ al'');$
$non\text{-}speculative\ P\ vs\ (llist\text{-}of\ (map\ NormalAction\ (take\ I\ \{\!|ta|\!\}_o)))\ ]\!]$
$\Longrightarrow \exists ta'\ e''\ xs''\ h''.\ P,t \vdash \langle e, (shr\ s,\ xs)\rangle -ta'\rightarrow \langle e'', (h'', xs'')\rangle \wedge$
red-mthr.mthr.if.actions-ok s t ta' ∧
$I < length\ \{\!|ta'|\!\}_o \wedge take\ I\ \{\!|ta'|\!\}_o = take\ I\ \{\!|ta|\!\}_o \wedge$
$\{\!|ta'|\!\}_o\ !\ I = ReadMem\ a''\ al''\ v' \wedge length\ \{\!|ta'|\!\}_o \leq max\ J\text{-}non\text{-}speculative\text{-}read\text{-}bound\ (length\ \{\!|ta|\!\}_o)$
and *reds-non-speculative-read*:
$[\![P,t \vdash \langle es, (shr\ s,\ xs)\rangle\ [-ta\rightarrow]\ \langle es', (h', xs')\rangle; \exists E\ Ts.\ P,E,shr\ s \vdash es\ [:]\ Ts;$
red-mthr.mthr.if.actions-ok s t ta;
$I < length\ \{\!|ta|\!\}_o;\ \{\!|ta|\!\}_o\ !\ I = ReadMem\ a''\ al''\ v;\ v' \in w\text{-}values\ P\ vs\ (map\ NormalAction\ (take\ I\ \{\!|ta|\!\}_o))\ (a'',\ al'');$
$non\text{-}speculative\ P\ vs\ (llist\text{-}of\ (map\ NormalAction\ (take\ I\ \{\!|ta|\!\}_o)))\ ]\!]$
$\Longrightarrow \exists ta'\ es''\ xs''\ h''.\ P,t \vdash \langle es, (shr\ s,\ xs)\rangle\ [-ta'\rightarrow]\ \langle es'', (h'', xs'')\rangle \wedge$
red-mthr.mthr.if.actions-ok s t ta' ∧
$I < length\ \{\!|ta'|\!\}_o \wedge take\ I\ \{\!|ta'|\!\}_o = take\ I\ \{\!|ta|\!\}_o \wedge$
$\{\!|ta'|\!\}_o\ !\ I = ReadMem\ a''\ al''\ v' \wedge length\ \{\!|ta'|\!\}_o \leq max\ J\text{-}non\text{-}speculative\text{-}read\text{-}bound\ (length\ \{\!|ta|\!\}_o)$
⟨*proof*⟩

end

sublocale *J-allocated-heap-conf* < *if-known-addrs-base*
J-known-addrs
final-expr mred P convert-RA
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*: $\bigcup$(*ka-Val ‘ set vs*) $\subseteq$ *set start-addrs*
shows *jmm-consistent* (*J-$\mathcal{E}$ P C M vs status*) *P*
(**is** *jmm-consistent ?$\mathcal{E}$ P*)
⟨*proof*⟩

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*: $\bigcup$(*ka-Val ‘ set vs*) $\subseteq$ *set start-addrs*
shows $\exists$ *E ws*. *E* $\in$ *J-$\mathcal{E}$ P C M vs status* $\wedge$ *P* $\vdash$ (*E*, *ws*) $\surd$ $\wedge$ *sequentially-consistent P* (*E*, *ws*)
(**is** $\exists$ *E ws*. - $\in$ *?$\mathcal{E}$* $\wedge$ -)
⟨*proof*⟩

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*: $\bigcup$(*ka-Val ‘ set vs*) $\subseteq$ *set start-addrs*
shows $\exists$ *E ws*. *legal-execution P* (*J-$\mathcal{E}$ P C M vs status*) (*E*, *ws*)
⟨*proof*⟩

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*
⟨*proof*⟩

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'*
⟨*proof*⟩

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*⟩

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*⦄$_o$
⟨*proof*⟩

lemma *exec-instr-known-addrs-ReadMem*:
assumes *exec*: $(ta, xcp', h', frs') \in exec\text{-}instr\ i\ P\ t\ h\ stk\ loc\ C\ M\ pc\ frs$
and *check*: $check\text{-}instr\ i\ P\ h\ stk\ loc\ C\ M\ pc\ frs$
and *read*: $ReadMem\ ad\ al\ v \in set\ \{\!|ta|\!\}_o$
shows $ad \in jvm\text{-}known\text{-}addrs\ t\ (None, (stk, loc, C, M, pc)\ \#\ frs)$
⟨*proof*⟩

lemma *mexecd-known-addrs-ReadMem*:
$[\![\ mexecd\ P\ t\ (xcpfrs, h)\ ta\ (xcpfrs', h');\ ReadMem\ ad\ al\ v \in set\ \{\!|ta|\!\}_o\ ]\!]$
$\Longrightarrow ad \in jvm\text{-}known\text{-}addrs\ t\ xcpfrs$
⟨*proof*⟩

lemma *exec-instr-known-addrs-WriteMem*:
assumes *exec*: $(ta, xcp', h', frs') \in exec\text{-}instr\ i\ P\ t\ h\ stk\ loc\ C\ M\ pc\ frs$
and *check*: $check\text{-}instr\ i\ P\ h\ stk\ loc\ C\ M\ pc\ frs$
and *write*: $\{\!|ta|\!\}_o\ !\ n = WriteMem\ ad\ al\ (Addr\ a)\ n < length\ \{\!|ta|\!\}_o$
shows $a \in jvm\text{-}known\text{-}addrs\ t\ (None, (stk, loc, C, M, pc)\ \#\ frs) \vee a \in new\text{-}obs\text{-}addrs\ (take\ n\ \{\!|ta|\!\}_o)$
⟨*proof*⟩

lemma *mexecd-known-addrs-WriteMem*:
$[\![\ mexecd\ P\ t\ (xcpfrs, h)\ ta\ (xcpfrs', h');\ \{\!|ta|\!\}_o\ !\ n = WriteMem\ ad\ al\ (Addr\ a);\ n < length\ \{\!|ta|\!\}_o\ ]\!]$
$\Longrightarrow a \in jvm\text{-}known\text{-}addrs\ t\ xcpfrs \vee a \in new\text{-}obs\text{-}addrs\ (take\ n\ \{\!|ta|\!\}_o)$
⟨*proof*⟩

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*: $check\text{-}instr\ i\ P\ h\ stk\ loc\ C\ M\ pc\ frs$
and *new*: $NewThread\ t'\ x'\ h'' \in set\ \{\!|ta|\!\}_t$
shows $jvm\text{-}known\text{-}addrs\ t'\ x' \subseteq jvm\text{-}known\text{-}addrs\ t\ (None, (stk, loc, C, M, pc)\ \#\ frs)$
⟨*proof*⟩

lemma *mexecd-known-addrs-new-thread*:
$[\![\ mexecd\ P\ t\ (xcpfrs, h)\ ta\ (xcpfrs', h');\ NewThread\ t'\ x'\ h'' \in set\ \{\!|ta|\!\}_t\ ]\!]$
$\Longrightarrow jvm\text{-}known\text{-}addrs\ t'\ x' \subseteq jvm\text{-}known\text{-}addrs\ t\ xcpfrs$
⟨*proof*⟩

lemma *exec-instr-New-same-addr-same*:
$[\![\ (ta, xcp', h', frs') \in exec\text{-}instr\ ins\ P\ t\ h\ stk\ loc\ C\ M\ pc\ frs;$
$\{\!|ta|\!\}_o\ !\ i = NewHeapElem\ a\ x;\ i < length\ \{\!|ta|\!\}_o;$
$\{\!|ta|\!\}_o\ !\ j = NewHeapElem\ a\ x';\ j < length\ \{\!|ta|\!\}_o\ ]\!]$
$\Longrightarrow i = j$
⟨*proof*⟩

lemma *exec-New-same-addr-same*:
$[\![\ (ta, xcp', h', frs') \in exec\ P\ t\ (xcp, h, frs);$
$\{\!|ta|\!\}_o\ !\ i = NewHeapElem\ a\ x;\ i < length\ \{\!|ta|\!\}_o;$
$\{\!|ta|\!\}_o\ !\ j = NewHeapElem\ a\ x';\ j < length\ \{\!|ta|\!\}_o\ ]\!]$
$\Longrightarrow i = j$
⟨*proof*⟩

lemma *exec-1-d-New-same-addr-same*:
$[\![\ P,t \vdash Normal\ (xcp, h, frs)\ -ta{-}jvmd{\rightarrow}\ Normal\ (xcp', h', frs');$
$\{\!|ta|\!\}_o\ !\ i = NewHeapElem\ a\ x;\ i < length\ \{\!|ta|\!\}_o;$
$\{\!|ta|\!\}_o\ !\ j = NewHeapElem\ a\ x';\ j < length\ \{\!|ta|\!\}_o\ ]\!]$

$\Longrightarrow i = j$
⟨*proof*⟩

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*
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *mexecd-allocated-mono*:
mexecd P t (*xcpfrs*, *h*) *ta* (*xcpfrs'*, *h'*) ⟹ *allocated h* ⊆ *allocated h'*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *mexecd-allocatedD*:
⟦ *mexecd P t* (*xcpfrs*, *h*) *ta* (*xcpfrs'*, *h'*); *NewHeapElem ad CTn* ∈ *set* $\{\!|ta|\!\}_o$ ⟧
⟹ *ad* ∈ *allocated h'* ∧ *ad* ∉ *allocated h*
⟨*proof*⟩

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* $\{\!|ta|\!\}_o$
⟨*proof*⟩

lemma *mexecd-NewHeapElemD*:
⟦ *mexecd P t* (*xcpfrs*, *h*) *ta* (*xcpfrs'*, *h'*); *ad* ∈ *allocated h'*; *ad* ∉ *allocated h* ⟧
⟹ ∃ *CTn*. *NewHeapElem ad CTn* ∈ *set* $\{\!|ta|\!\}_o$
⟨*proof*⟩

lemma *mexecd-allocated-multithreaded*:
allocated-multithreaded addr2thread-id thread-id2addr empty-heap allocate typeof-addr heap-write allocated JVM-final (*mexecd P*) *P*
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

end

context *JVM-heap* **begin**

lemma *exec-instr-read-typeable*:
assumes *exec*: $(ta,\ xcp',\ h',\ frs') \in$ *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* $\in$ *set* $\{\!|ta|\!\}_o$
shows $\exists\ T'.\ P,h \vdash ad@al : T'$
⟨*proof*⟩

lemma *exec-1-d-read-typeable*:
$[\![$ $P,t \vdash$ *Normal* $(xcp,\ h,\ frs)$ $-ta-jvmd\rightarrow$ *Normal* $(xcp',\ h',\ frs')$;
ReadMem ad al v $\in$ *set* $\{\!|ta|\!\}_o$ $]\!]$
$\Longrightarrow \exists\ T'.\ P,h \vdash ad@al : T'$
⟨*proof*⟩

end

sublocale *JVM-heap-base* < *execd-mthr*:
if-multithreaded
JVM-final
mexecd P
convert-RA
for *P*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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'*))
⟨*proof*⟩

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* $\{\!|ta|\!\}_o$)))
⟨*proof*⟩

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*) (λt (*xcp*, *frstls*) *h*. $\Phi \vdash t$: (*xcp*, *h*, *frstls*) $\surd$) *P*
⟨*proof*⟩

lemma *executions-sc*:
assumes *wf*: *wf-jvm-prog*$_\Phi$ *P*
and *wf-start*: *wf-start-state P C M vs*
and *vs2*: $\bigcup$(*ka-Val* ‘ *set vs*) ⊆ *set start-addrs*
shows *executions-sc-hb* (*JVMd-*$\mathcal{E}$ *P C M vs status*) *P*
(**is** *executions-sc-hb ?E P*)
⟨*proof*⟩

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'*: $v' \in w\text{-}values\ P\ vs\ (map\ NormalAction\ (take\ I\ \{\!|ta|\!\}_o))\ (a'', al'')$
and *ns*: $non\text{-}speculative\ P\ vs\ (llist\text{-}of\ (map\ NormalAction\ (take\ I\ \{\!|ta|\!\}_o)))$
shows $\exists\, ta'\ xcp''\ h''\ frs''.\ (ta', xcp'', h'', frs'') \in exec\text{-}instr\ i\ P\ t\ (shr\ s)\ stk\ loc\ C\ M\ pc\ frs \wedge$
$execd\text{-}mthr.mthr.if.actions\text{-}ok\ s\ t\ ta' \wedge$
$I < length\ \{\!|ta'|\!\}_o \wedge take\ I\ \{\!|ta'|\!\}_o = take\ I\ \{\!|ta|\!\}_o \wedge$
$\{\!|ta'|\!\}_o\ !\ I = ReadMem\ a''\ al''\ v' \wedge$
$length\ \{\!|ta'|\!\}_o \leq max\ jvm\text{-}non\text{-}speculative\text{-}read\text{-}bound\ (length\ \{\!|ta|\!\}_o)$
⟨*proof*⟩

lemma *exec-1-d-non-speculative-read*:
assumes *hrt*: *heap-read-typeable hconf P*
and *vs*: *vs-conf P* (*shr s*) *vs*
and *exec*: $P,t \vdash Normal\ (xcp, shr\ s, frs)\ -ta-jvmd\rightarrow\ Normal\ (xcp', h', frs')$
and *aok*: *execd-mthr.mthr.if.actions-ok s t ta*
and *hconf*: *hconf* (*shr s*)
and *i*: $I < length\ \{\!|ta|\!\}_o$
and *read*: $\{\!|ta|\!\}_o\ !\ I = ReadMem\ a''\ al''\ v$
and *v'*: $v' \in w\text{-}values\ P\ vs\ (map\ NormalAction\ (take\ I\ \{\!|ta|\!\}_o))\ (a'', al'')$
and *ns*: $non\text{-}speculative\ P\ vs\ (llist\text{-}of\ (map\ NormalAction\ (take\ I\ \{\!|ta|\!\}_o)))$
shows $\exists\, ta'\ xcp''\ h''\ frs''.\ P,t \vdash Normal\ (xcp, shr\ s, frs)\ -ta'-jvmd\rightarrow\ Normal\ (xcp'', h'', frs'') \wedge$
$execd\text{-}mthr.mthr.if.actions\text{-}ok\ s\ t\ ta' \wedge$
$I < length\ \{\!|ta'|\!\}_o \wedge take\ I\ \{\!|ta'|\!\}_o = take\ I\ \{\!|ta|\!\}_o \wedge$
$\{\!|ta'|\!\}_o\ !\ I = ReadMem\ a''\ al''\ v' \wedge$
$length\ \{\!|ta'|\!\}_o \leq max\ jvm\text{-}non\text{-}speculative\text{-}read\text{-}bound\ (length\ \{\!|ta|\!\}_o)$
⟨*proof*⟩

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*⟩

lemma *JVM-cut-and-update*:
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.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*⟩

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-ℰ P C M vs status*) *P*
⟨*proof*⟩

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-ℰ P C M vs status*) *P*
⟨*proof*⟩

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-ℰ P C M vs status*) *P*
(**is** *jmm-consistent ?ℰ P*)
⟨*proof*⟩

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*: $\bigcup$(*ka-Val* ‘ *set vs*) $\subseteq$ *set start-addrs*
shows $\exists$ *E ws*. *E* $\in$ *JVMd-*$\mathcal{E}$ *P C M vs status* $\wedge$ *P* $\vdash$ (*E*, *ws*) $\surd$ $\wedge$ *sequentially-consistent P* (*E*, *ws*)
(**is** $\exists$ *E ws*. - $\in$ *?*$\mathcal{E}$ $\wedge$ -)
⟨*proof*⟩

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*: $\bigcup$(*ka-Val* ‘ *set vs*) $\subseteq$ *set start-addrs*
shows $\exists$ *E ws*. *legal-execution P* (*JVMd-*$\mathcal{E}$ *P C M vs status*) (*E*, *ws*)
⟨*proof*⟩

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* = (λ*a*. (*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
⟨*proof*⟩

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* ⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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* ⟨*proof*⟩

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* ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *jmm′-heap-read-typeable*: *jmm′-heap-read-typeable tytok P*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *action-loc-compP*: *action-loc (compP f P) = action-loc P*

⟨*proof*⟩

lemma *is-volatile-compP* [*simp*]: *is-volatile* (*compP f P*) = *is-volatile P*
⟨*proof*⟩

lemma *saction-compP* [*simp*]: *saction* (*compP f P*) = *saction P*
⟨*proof*⟩

lemma *sactions-compP* [*simp*]: *sactions* (*compP f P*) = *sactions P*
⟨*proof*⟩

lemma *addr-locs-compP* [*simp*]: *addr-locs* (*compP f P*) = *addr-locs P*
⟨*proof*⟩

lemma *syncronizes-with-compP* [*simp*]: *synchronizes-with* (*compP f P*) = *synchronizes-with P*
⟨*proof*⟩

lemma *sync-order-compP* [*simp*]: *sync-order* (*compP f P*) = *sync-order P*
⟨*proof*⟩

lemma *sync-with-compP* [*simp*]: *sync-with* (*compP f P*) = *sync-with P*
⟨*proof*⟩

lemma *po-sw-compP* [*simp*]: *po-sw* (*compP f P*) = *po-sw P*
⟨*proof*⟩

lemma *happens-before-compP*: *happens-before* (*compP f P*) = *happens-before P*
⟨*proof*⟩

lemma *addr-loc-default-compP* [*simp*]: *addr-loc-default* (*compP f P*) = *addr-loc-default P*
⟨*proof*⟩

lemma *value-written-aux-compP* [*simp*]: *value-written-aux* (*compP f P*) = *value-written-aux P*
⟨*proof*⟩

lemma *value-written-compP* [*simp*]: *value-written* (*compP f P*) = *value-written P*
⟨*proof*⟩

lemma *is-write-seen-compP* [*simp*]: *is-write-seen* (*compP f P*) = *is-write-seen P*
⟨*proof*⟩

lemma *justification-well-formed-compP* [*simp*]:
justification-well-formed (*compP f P*) = *justification-well-formed P*
⟨*proof*⟩

lemma *happens-before-committed-compP* [*simp*]:
happens-before-committed (*compP f P*) = *happens-before-committed P*
⟨*proof*⟩

lemma *happens-before-committed-weak-compP* [*simp*]:
happens-before-committed-weak (*compP f P*) = *happens-before-committed-weak P*
⟨*proof*⟩

lemma *sync-order-committed-compP* [*simp*]:

sync-order-committed (*compP f P*) = *sync-order-committed P*
⟨*proof*⟩

lemma *value-written-committed-compP* [*simp*]:
value-written-committed (*compP f P*) = *value-written-committed P*
⟨*proof*⟩

lemma *uncommitted-reads-see-hb-compP* [*simp*]:
uncommitted-reads-see-hb (*compP f P*) = *uncommitted-reads-see-hb P*
⟨*proof*⟩

lemma *external-actions-committed-compP* [*simp*]:
external-actions-committed (*compP f P*) = *external-actions-committed P*
⟨*proof*⟩

lemma *is-justified-by-compP* [*simp*]: *is-justified-by* (*compP f P*) = *is-justified-by P*
⟨*proof*⟩

lemma *is-weakly-justified-by-compP* [*simp*]: *is-weakly-justified-by* (*compP f P*) = *is-weakly-justified-by P*
⟨*proof*⟩

lemma *legal-execution-compP*: *legal-execution* (*compP f P*) = *legal-execution P*
⟨*proof*⟩

lemma *weakly-legal-execution-compP*: *weakly-legal-execution* (*compP f P*) = *weakly-legal-execution P*
⟨*proof*⟩

lemma *most-recent-write-for-compP* [*simp*]:
most-recent-write-for (*compP f P*) = *most-recent-write-for P*
⟨*proof*⟩

lemma *sequentially-consistent-compP* [*simp*]:
sequentially-consistent (*compP f P*) = *sequentially-consistent P*
⟨*proof*⟩

lemma *conflict-compP* [*simp*]: *non-volatile-conflict* (*compP f P*) = *non-volatile-conflict P*
⟨*proof*⟩

lemma *correctly-synchronized-compP* [*simp*]:
correctly-synchronized (*compP f P*) = *correctly-synchronized P*
⟨*proof*⟩

lemma (**in** *heap-base*) *heap-read-typed-compP* [*simp*]:
heap-read-typed (*compP f P*) = *heap-read-typed P*
⟨*proof*⟩

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 $\circ_B$
FWbisimulation-base.mbisim red0-Red1'.init-fin-bisim red0-Red1'.init-fin-bisim-wait $\circ_B$
if-mbisim-Red1'-Red1 $\circ_B$
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*
⟨*proof*⟩

sublocale *J-JVM-conf-read* < *execd-mthr*:
if-τmultithreaded-wf
JVM-final
mexecd (*compP2* (*compP1 P*))
convert-RA
τMOVE2 (*compP2* (*compP1 P*))
⟨*proof*⟩

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
⟨*proof*⟩

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*))
⟨*proof*⟩

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.*$\mathcal{E}$ *P s* = *execd-mthr.mthr.if.*$\mathcal{E}$ (*J2JVM P*) *cs*
⟨*proof*⟩

lemma *red-ℰ-eq-mexecd-ℰ*:
  ⟦ *wf-J-prog P*; *wf-start-state P C M vs* ⟧
  ⟹ *J-ℰ P C M vs Running* = *JVMd-ℰ* (*J2JVM P*) *C M vs Running*
⟨*proof*⟩

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-ℰ P C M vs Running*) (*E*, *ws*) ⟷
      legal-execution (*J2JVM P*) (*JVMd-ℰ* (*J2JVM P*) *C M vs Running*) (*E*, *ws*)
⟨*proof*⟩

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-ℰ P C M vs Running*) (*E*, *ws*) ⟷
      weakly-legal-execution (*J2JVM P*) (*JVMd-ℰ* (*J2JVM P*) *C M vs Running*) (*E*, *ws*)
⟨*proof*⟩

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-ℰ* (*J2JVM P*) *C M vs Running*) ⟷
      correctly-synchronized P (*J-ℰ P C M vs Running*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩
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*
⟨*proof*⟩

lemma *jmm-typeof-addr*′*-conv-jmm-typeof-addr*: (λ-. *jmm-typeof-addr*′ *P*) = *jmm-typeof-addr P*
⟨*proof*⟩

lemma *jmm-typeof-addr-conv-jmm-typeof-addr*′: *jmm-typeof-addr* = (λ*P* -. *jmm-typeof-addr*′ *P*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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* ⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *jmm-allocate-Eps*:
$(SOME\ ha.\ ha \in jmm\text{-}allocate\ h\ hT) = (h', a')$
$\Longrightarrow jmm\text{-}allocate\ h\ hT \neq \{\} \longrightarrow (h', a') \in jmm\text{-}allocate\ h\ hT$
⟨*proof*⟩

lemma *jmm-allocate-eq-empty*: $jmm\text{-}allocate\ h\ hT = \{\} \longleftrightarrow h\ hT = UNIV$
⟨*proof*⟩

lemma *jmm-allocate-otherD*:
$(h', a) \in jmm\text{-}allocate\ h\ hT \Longrightarrow \forall hT'.\ hT' \neq hT \longrightarrow h'\ hT' = h\ hT'$
⟨*proof*⟩

lemma *jmm-start-heap-ok*: *jmm.start-heap-ok*
⟨*proof*⟩

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* ⟨*proof*⟩

abbreviation (*input*) *jmm′-hconf* :: *JMM-heap* ⇒ *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* ⟨*proof*⟩

abbreviation *jmm-heap-read-typeable* :: *′m prog* ⇒ *bool*
where *jmm-heap-read-typeable P* ≡ *jmm.heap-read-typeable TYPE(′m) P jmm-hconf P*

abbreviation *jmm′-heap-read-typeable* :: *′m prog* ⇒ *bool*
where *jmm′-heap-read-typeable P* ≡ *jmm′.heap-read-typeable TYPE(′m) P jmm-hconf P*

lemma *jmm-heap-read-typeable*: *jmm-heap-read-typeable P*
⟨*proof*⟩

lemma *jmm′-heap-read-typeable*: *jmm′-heap-read-typeable P*
⟨*proof*⟩

lemma *jmm-heap-conf*:
heap-conf addr2thread-id thread-id2addr jmm-empty jmm-allocate (*jmm-typeof-addr P*) *jmm-heap-write jmm-hconf P*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *jmm-typeof-addr-compP* [*simp*]:
jmm-typeof-addr (*compP f P*) = *jmm-typeof-addr P*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *jmm-heap′′*: *heap′′ addr2thread-id thread-id2addr jmm-allocate* (*jmm-typeof-addr′ P*) *jmm-heap-write P*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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′* - -)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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'*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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* ⟧
⟹ *typeof-addr ad* = ⌊*CTn*⌋
and *reds-New-type-match*:
⟦ *h.reds′ P t es s ta es′ s′*; *NewHeapElem ad CTn* ∈ *set* $\{\!|ta|\!\}_o$; *typeof-addr ad* ≠ *None* ⟧
⟹ *typeof-addr ad* = ⌊*CTn*⌋
⟨*proof*⟩

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*) ($\lambda t\ x\ h.\ \exists ET.$ *h.sconf-type-ok ET t x h*) *P*
⟨*proof*⟩

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-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*⟩

end

8.20.1 Specific part for JMM implementation 2

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 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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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′*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *J-ℰ-heap-read-typedD*:
E ∈ *J-heap-base.J-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-ℰ addr2thread-id thread-id2addr spurious-wakeups empty-heap allocate* (λ-. *typeof-addr*) *jmm-heap-read jmm-heap-write P C M vs status*
⟨*proof*⟩

lemma *J-ℰ-typesafe-subset*: *jmm'-J-ℰ P C M vs status* ⊆ *jmm-J-ℰ P C M vs status*
⟨*proof*⟩

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-ℰ P C M vs status*) (*E*, *ws*)
shows *legal-execution P* (*jmm'-J-ℰ P C M vs status*) (*E*, *ws*)
⟨*proof*⟩

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*⟩

lemma *J-legal-typesafe2*:
assumes *legal*: *legal-execution P* (*jmm'-J-ℰ P C M vs status*) (*E*, *ws*)
shows *legal-execution P* (*jmm-J-ℰ P C M vs status*) (*E*, *ws*)
⟨*proof*⟩

lemma *J-weakly-legal-typesafe2*:
assumes *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*⟩

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-ℰ P C M vs status*) = *weakly-legal-execution P* (*jmm′-J-ℰ P C M vs status*)
⟨*proof*⟩

theorem *J-legal-typesafe*:
  assumes *wf-J-prog P*
  and *jmm-wf-start-state P C M vs*
  shows *legal-execution P* (*jmm-J-ℰ P C M vs status*) = *legal-execution P* (*jmm′-J-ℰ P C M vs status*)
⟨*proof*⟩

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
⟨*proof*⟩

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*⌋
⟨*proof*⟩

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*⌋
⟨*proof*⟩

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*⟩

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*⟩

end

abbreviation *jmm-JVMd-*$\mathcal{E}$
:: *addr jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒ *status* ⇒ (*addr* × (*addr*, *addr*) *obs-event action*) *llist set*
where
jmm-JVMd-$\mathcal{E}$ *P* ≡
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* ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒ *status* ⇒ (*addr* × (*addr*, *addr*) *obs-event*

action) *llist set*
where
jmm′-JVMd-ℰ P ≡
JVM-heap-base.JVMd-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* ⇒ *cname* ⇒ *mname* ⇒ *addr val list* ⇒ (*addr*,*thread-id*,*addr jvm-thread-state*,*JMM-heap*,*addr*) *state*
where *jmm-JVM-start-state* ≡ *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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *exec-instr-heap-read-typed*:
(*ta*, *xcphfrs′*) ∈ *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* ⟷
(*ta*, *xcphfrs′*) ∈ *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* ∧
(∀ *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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *JVMd-ℰ-heap-read-typedI*:
⟦ *E* ∈ *JVM-heap-base.JVMd-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-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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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'*
⟹ ∃ *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'*
⟨*proof*⟩

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-ℰ P C M vs status*) (*E*, *ws*)
shows *legal-execution P* (*jmm'-JVMd-ℰ P C M vs status*) (*E*, *ws*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *JVMd-ℰ-typesafe-subset*: *jmm'-JVMd-ℰ P C M vs status* ⊆ *jmm-JVMd-ℰ P C M vs status*
⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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-ℰ P C M vs Running*) (*E*, *ws*) ⟷
legal-execution (*J2JVM P*) (*jmm-JVMd-ℰ* (*J2JVM P*) *C M vs Running*) (*E*, *ws*)
⟨*proof*⟩

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-ℰ P C M vs Running*) (*E*, *ws*) ⟷
weakly-legal-execution (*J2JVM P*) (*jmm-JVMd-ℰ* (*J2JVM P*) *C M vs Running*) (*E*, *ws*)
⟨*proof*⟩

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*⟩

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)*
⟨*proof*⟩

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*)
⟨*proof*⟩

end

sublocale *state-refine-base* < α: *final-thread final* ⟨*proof*⟩
sublocale *state-refine-base* < α:
  multithreaded-base
    final
    $\lambda t\ xm\ ta\ x'm'.$ *Predicate.eval* (*r t xm*) (*ta*, *x'm'*)
⟨*proof*⟩

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 (*λa*. ∃ *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′*))
⟨*proof*⟩

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* ⟧

$\Longrightarrow$ *α.active-threads* (*state-α s*) = {}
and *schedule-Some-NoneD*:
⟦ *schedule σ s* = ⌊(*t*, *None*, *σ'*)⌋; *state-invar s*; *σ-invar σ* (*dom* (*thr-α* (*thr s*))); *state-α s* ∈ *invariant* ⟧
$\Longrightarrow$ ∃ *x ln n*. *thr-α* (*thr s*) *t* = ⌊(*x*, *ln*)⌋ ∧ *ln* $ *n* > *0* ∧ ¬ *waiting* (*ws-α* (*wset s*) *t*) ∧ *may-acquire-all* (*locks s*) *t ln*
and *schedule-Some-SomeD*:
⟦ *schedule σ s* = ⌊(*t*, ⌊(*ta*, *x'*, *m'*)⌋, *σ'*)⌋; *state-invar s*; *σ-invar σ* (*dom* (*thr-α* (*thr s*))); *state-α s* ∈ *invariant* ⟧
$\Longrightarrow$ ∃ *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*
and *schedule-invar-None*:
⟦ *schedule σ s* = ⌊(*t*, *None*, *σ'*)⌋; *state-invar s*; *σ-invar σ* (*dom* (*thr-α* (*thr s*))); *state-α s* ∈ *invariant* ⟧
$\Longrightarrow$ *σ-invar σ'* (*dom* (*thr-α* (*thr s*)))
and *schedule-invar-Some*:
⟦ *schedule σ s* = ⌊(*t*, ⌊(*ta*, *x'*, *m'*)⌋, *σ'*)⌋; *state-invar s*; *σ-invar σ* (*dom* (*thr-α* (*thr s*))); *state-α s* ∈ *invariant* ⟧
$\Longrightarrow$ *σ-invar σ'* (*dom* (*thr-α* (*thr s*)) ∪ {*t*. ∃ *x m*. *NewThread t x m* ∈ *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* ⇒ *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 *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 *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 *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* ⇒ *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 *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 *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*
+
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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *thread-ok-correct* [*simp*]:
thr-invar ts ⟹ *thread-ok ts nta* ⟷ *FWThread.thread-ok* (*thr-α ts*) *nta*
⟨*proof*⟩

lemma *thread-oks-correct* [*simp*]:
thr-invar ts ⟹ *thread-oks ts ntas* ⟷ *FWThread.thread-oks* (*thr-α ts*) *ntas*
⟨*proof*⟩

lemma *wset-actions-ok-correct* [*simp*]:
ws-invar ws ⟹ *wset-actions-ok ws t was* ⟷ *FWWait.wset-actions-ok* (*ws-α ws*) *t was*
⟨*proof*⟩

lemma *cond-action-ok-correct* [*simp*]:
state-invar s ⟹ *cond-action-ok s t cta* ⟷ *α.cond-action-ok* (*state-α s*) *t cta*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *interrupt-action-ok-correct* [*simp*]:
is-invar is ⟹ *interrupt-action-ok is ia* ⟷ *FWInterrupt.interrupt-action-ok* (*is-α is*) *ia*
⟨*proof*⟩

lemma *interrupt-actions-ok-correct* [*simp*]:
is-invar is $\Longrightarrow$ *interrupt-actions-ok is ias* $\longleftrightarrow$ *FWInterrupt.interrupt-actions-ok* (*is-α is*) *ias*
⟨*proof*⟩

lemma *actions-ok-correct* [*simp*]:
state-invar s $\Longrightarrow$ *actions-ok s t ta* $\longleftrightarrow$ *α.actions-ok* (*state-α s*) *t ta*
⟨*proof*⟩

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* ⇒ *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* :: $'m\text{-}w \Rightarrow$ *bool*
and *ws-lookup* :: $'t \Rightarrow 'm\text{-}w \rightharpoonup 'w$ *wait-set-status*
and *ws-update* :: $'t \Rightarrow 'w$ *wait-set-status* $\Rightarrow 'm\text{-}w \Rightarrow 'm\text{-}w$
and *ws-delete* :: $'t \Rightarrow 'm\text{-}w \Rightarrow 'm\text{-}w$
and *ws-iterate* :: $'m\text{-}w \Rightarrow ('t \times 'w$ *wait-set-status*, $'m\text{-}w)$ *set-iterator*
and *is-α* :: $'s\text{-}i \Rightarrow 't$ *interrupts*
and *is-invar* :: $'s\text{-}i \Rightarrow$ *bool*
and *is-memb* :: $'t \Rightarrow 's\text{-}i \Rightarrow$ *bool*
and *is-ins* :: $'t \Rightarrow 's\text{-}i \Rightarrow 's\text{-}i$
and *is-delete* :: $'t \Rightarrow 's\text{-}i \Rightarrow 's\text{-}i$
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*) $\subseteq T$ $t \in T$
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*⟩

lemma *exec-updWs-correct*:
assumes *ws-invar ws σ-invar σ T dom* (*ws-α ws*) $\subseteq T$ $t \in T$
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*⟩

lemma *exec-upd-correct*:
assumes *state-invar s σ-invar σ* (*dom* (*thr-α* (*thr s*))) $t \in$ (*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′*)
⟨*proof*⟩

lemma *execT-None*:
assumes *invar*: *state-invar s σ-invar σ* (*dom* (*thr-α* (*thr s*))) *state-α s* $\in$ *invariant*
and *exec*: *execT σ s* = *None*
shows *α.active-threads* (*state-α s*) = {}
⟨*proof*⟩

lemma *execT-Some*:
assumes *invar*: *state-invar s σ-invar σ* (*dom* (*thr-α* (*thr s*))) *state-α s* $\in$ *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*⟩

lemma *exec-step-into-redT*:
assumes *invar*: *state-invar s σ-invar σ* (*dom* (*thr-α* (*thr s*))) *state-α s* $\in$ *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*⟩

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*
⟨*proof*⟩

lemma (**in** *multithreaded-base*) *red-in-active-threads*:
assumes *s* −*t*▹*ta*→ *s′*
shows *t* ∈ *active-threads s*
⟨*proof*⟩

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*⟩

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*⟩

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* :: $'t \Rightarrow 'm\text{-}w \rightharpoonup 'w$ *wait-set-status*
and *ws-update* :: $'t \Rightarrow 'w$ *wait-set-status* $\Rightarrow 'm\text{-}w \Rightarrow 'm\text{-}w$
and *ws-delete* :: $'t \Rightarrow 'm\text{-}w \Rightarrow 'm\text{-}w$
and *ws-iterate* :: $'m\text{-}w \Rightarrow ('t \times 'w$ *wait-set-status*, $'m\text{-}w)$ *set-iterator*
and *ws-sel* :: $'m\text{-}w \Rightarrow ('t \times 'w$ *wait-set-status* $\Rightarrow$ *bool*$) \rightharpoonup ('t \times 'w$ *wait-set-status*$)$
and *is-α* :: $'s\text{-}i \Rightarrow 't$ *interrupts*
and *is-invar* :: $'s\text{-}i \Rightarrow$ *bool*
and *is-memb* :: $'t \Rightarrow 's\text{-}i \Rightarrow$ *bool*
and *is-ins* :: $'t \Rightarrow 's\text{-}i \Rightarrow 's\text{-}i$
and *is-delete* :: $'t \Rightarrow 's\text{-}i \Rightarrow 's\text{-}i$
and *thr′-α* :: $'s\text{-}t \Rightarrow 't$ *set*
and *thr′-invar* :: $'s\text{-}t \Rightarrow$ *bool*
and *thr′-empty* :: *unit* $\Rightarrow 's\text{-}t$
and *thr′-ins-dj* :: $'t \Rightarrow 's\text{-}t \Rightarrow 's\text{-}t$
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
⟨*proof*⟩

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
⟨*proof*⟩

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* $\Rightarrow 's) \Rightarrow ('l,'t,'x,'m,'w)$ *state* $\Rightarrow 't$
$\Rightarrow ('t \times (('l,'t,'x,'m,'w,'o)$ *thread-action* $\times 'x \times 'm)$ *option* $\times 's)$ *option*
where
$\bigwedge ln.$ *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* ⟹ *t* ∉ *active-threads s*
⟨*proof*⟩

lemma *inactive-step-thread-eq-NoneI*:
t ∉ *active-threads s* ⟹ *step-thread update-state s t* = *None*
⟨*proof*⟩

lemma *step-thread-eq-None-conv*:
step-thread update-state s t = *None* ⟷ *t* ∉ *active-threads s*
⟨*proof*⟩

lemma *step-thread-eq-Some-activeD*:
step-thread update-state s t = ⌊(*t′*, *taxmσ′*)⌋
⟹ *t′* = *t* ∧ *t* ∈ *active-threads s*
⟨*proof*⟩

declare *actions-ok-iff* [*simp del*]
declare *actions-ok.cases* [*rule del*]

lemma *step-thread-Some-NoneD*:
step-thread update-state s t′ = ⌊(*t*, *None*, *σ′*)⌋
⟹ ∃ *x ln n. thr s t* = ⌊(*x*, *ln*)⌋ ∧ *ln $ n* > *0* ∧ ¬ *waiting* (*wset s t*) ∧ *may-acquire-all* (*locks s*) *t ln* ∧ *σ′* = *update-state* (*K$* [], [], [], [], [], *convert-RA ln*)
⟨*proof*⟩

lemma *step-thread-Some-SomeD*:
⟦ *deterministic I*; *step-thread update-state s t′* = ⌊(*t*, ⌊(*ta*, *x′*, *m′*)⌋, *σ′*)⌋; *s* ∈ *I* ⟧
⟹ ∃ *x. thr s t* = ⌊(*x*, *no-wait-locks*)⌋ ∧ *t* ⊢ ⟨*x*, *shr s*⟩ −*ta*→ ⟨*x′*, *m′*⟩ ∧ *actions-ok s t ta* ∧ *σ′* = *update-state ta*
⟨*proof*⟩

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*
⌊(*x*, *ln*)⌋ ⇒
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*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *step-thread-Some-NoneD*:
assumes *det*: *α.deterministic I*
and *step*: *step-thread update-state s t′* = ⌊(*t*, *None*, *σ′*)⌋
and *invar*: *state-invar s state-α s* ∈ *I*
shows ∃ *x ln n*. *thr-α* (*thr s*) *t* = ⌊(*x*, *ln*)⌋ ∧ *ln* $ *n* > *0* ∧ ¬ *waiting* (*ws-α* (*wset s*) *t*) ∧ *may-acquire-all* (*locks s*) *t ln* ∧ *σ′* = *update-state* (*K*$ [], [], [], [], [], *convert-RA ln*)
⟨*proof*⟩

lemma *step-thread-Some-SomeD*:
assumes *det*: *α.deterministic I*
and *step*: *step-thread update-state s t′* = ⌊(*t*, ⌊(*ta*, *x′*, *m′*)⌋, *σ′*)⌋
and *invar*: *state-invar s state-α s* ∈ *I*
shows ∃ *x*. *thr-α* (*thr s*) *t* = ⌊(*x*, *no-wait-locks*)⌋ ∧ *Predicate.eval* (*r t* (*x*, *shr s*)) (*ta*, *x′*, *m′*) ∧ *actions-ok s t ta* ∧ *σ′* = *update-state ta*
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *the2-eq* [*code*]:
  the2 A = *singleton2* (λ*x*. *Code.abort* (*STR ''not-unique''*) (λ-. *the2 A*)) *A*
⟨*proof*⟩

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 (λ*ta. 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* = {}
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

end

sublocale *random-scheduler-base* <
  scheduler-base
    final r convert-RA
    random-scheduler output pick-wakeup-via-sel (λs *P*. *ws-sel s* ($\lambda(k,v)$. *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* ⟨*proof*⟩

sublocale *random-scheduler* <
  pick-wakeup-spec
    final r convert-RA
    pick-wakeup-via-sel (λs *P*. *ws-sel s* ($\lambda(k,v)$. *P k v*)) *random-scheduler-invar*
    thr-α thr-invar
    ws-α ws-invar
    is-α is-invar
⟨*proof*⟩

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* (λs *P*. *ws-sel s* ($\lambda(k,v)$. *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*⟩

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*}
⟨*proof*⟩

lemma *set-enqueue-new-threads*:
set (*enqueue-new-threads queue ntas*) = *set queue* ∪ {*t*. ∃ *x m*. *NewThread t x m* ∈ *set ntas*}
⟨*proof*⟩

lemma *enqueue-new-thread-eq-Nil* [*simp*]:
enqueue-new-thread queue nta = [] ⟷ *queue* = [] ∧ (∃ *t b*. *nta* = *ThreadExists t b*)
⟨*proof*⟩

lemma *enqueue-new-threads-eq-Nil* [*simp*]:
enqueue-new-threads queue ntas = [] ⟷ *queue* = [] ∧ *set ntas* ⊆ {*ThreadExists t b*|*t b*. *True*}
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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* $\{\!|ta|\!\}_t$})
⟨*proof*⟩

lemma *round-robin-reschedule-Cons*:
round-robin-reschedule t0 (*t0* # *queue*) *n0 s* = *None*

$t \neq t0 \Longrightarrow$ *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*)
⟨*proof*⟩

lemma *round-robin-reschedule-NoneD*:
assumes *rrr*: *round-robin-reschedule t0 queue n0 s* = *None*
and *t0*: $t0 \in$ *set queue*
shows *set* (*takeWhile* ($\lambda t'.\ t' \neq t0$) *queue*) ∩ *active-threads s* = {}
⟨*proof*⟩

lemma *round-robin-reschedule-Some-NoneD*:
assumes *rrr*: *round-robin-reschedule t0 queue n0 s* = ⌊(*t*, *None*, σ′)⌋
and *t0*: $t0 \in$ *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*
⟨*proof*⟩

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*: $t0 \in$ *set queue*
and *I*: $s \in I$
shows ∃ *x*. *thr s t* = ⌊(*x*, *no-wait-locks*)⌋ ∧ *t* ⊢ ⟨*x*, *shr s*⟩ −*ta*→ ⟨*x*′, *m*′⟩ ∧ *actions-ok s t ta*
⟨*proof*⟩

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*: $t0 \in$ *set queue*
shows *round-robin-invar* σ′ (*dom* (*thr s*))
⟨*proof*⟩

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*: $t0 \in$ *set queue*
and $s \in I$
shows *round-robin-invar* σ′ (*dom* (*thr s*) ∪ {*t*. ∃ *x m*. *NewThread t x m* ∈ *set* $\{\!|ta|\!\}_t$})
⟨*proof*⟩

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*⟩

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*⟩

lemma *round-robin-Some-SomeD*:
assumes *deterministic I*
and *rr*: $round\text{-}robin\ n0\ \sigma\ s = \lfloor(t,\ \lfloor(ta,\ x',\ m')\rfloor,\ \sigma')\rfloor$
and $s \in I$
shows $\exists x.\ thr\ s\ t = \lfloor(x,\ no\text{-}wait\text{-}locks)\rfloor \wedge t \vdash \langle x,\ shr\ s\rangle\ -ta\rightarrow\ \langle x',\ m'\rangle \wedge actions\text{-}ok\ s\ t\ ta$
⟨*proof*⟩

lemma *round-robin-invar-None*:
assumes *rr*: $round\text{-}robin\ n0\ \sigma\ s = \lfloor(t,\ None,\ \sigma')\rfloor$
and *invar*: $round\text{-}robin\text{-}invar\ \sigma\ (dom\ (thr\ s))$
shows $round\text{-}robin\text{-}invar\ \sigma'\ (dom\ (thr\ s))$
⟨*proof*⟩

lemma *round-robin-invar-Some*:
assumes *deterministic I*
and *rr*: $round\text{-}robin\ n0\ \sigma\ s = \lfloor(t,\ \lfloor(ta,\ x',\ m')\rfloor,\ \sigma')\rfloor$
and *invar*: $round\text{-}robin\text{-}invar\ \sigma\ (dom\ (thr\ s))\ s \in I$
shows $round\text{-}robin\text{-}invar\ \sigma'\ (dom\ (thr\ s) \cup \{t.\ \exists x\ m.\ NewThread\ t\ x\ m \in set\ \{\!|ta|\!\}_t\})$
⟨*proof*⟩

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* :: $'x \Rightarrow bool$
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 *output* :: $'queue\ round\text{-}robin \Rightarrow 't \Rightarrow ('l,'t,'x,'m,'w,'o)\ thread\text{-}action \Rightarrow 'q\ option$
and *thr-α* :: $'m\text{-}t \Rightarrow ('l,'t,'x)\ thread\text{-}info$
and *thr-invar* :: $'m\text{-}t \Rightarrow bool$
and *thr-lookup* :: $'t \Rightarrow 'm\text{-}t \rightharpoonup ('x \times 'l\ released\text{-}locks)$
and *thr-update* :: $'t \Rightarrow 'x \times 'l\ released\text{-}locks \Rightarrow 'm\text{-}t \Rightarrow 'm\text{-}t$
and *ws-α* :: $'m\text{-}w \Rightarrow ('w,'t)\ wait\text{-}sets$
and *ws-invar* :: $'m\text{-}w \Rightarrow bool$
and *ws-lookup* :: $'t \Rightarrow 'm\text{-}w \rightharpoonup 'w\ wait\text{-}set\text{-}status$
and *ws-update* :: $'t \Rightarrow 'w\ wait\text{-}set\text{-}status \Rightarrow 'm\text{-}w \Rightarrow 'm\text{-}w$
and *ws-delete* :: $'t \Rightarrow 'm\text{-}w \Rightarrow 'm\text{-}w$
and *ws-iterate* :: $'m\text{-}w \Rightarrow ('t \times 'w\ wait\text{-}set\text{-}status,\ 'm\text{-}w)\ set\text{-}iterator$
and *ws-sel* :: $'m\text{-}w \Rightarrow ('t \times 'w\ wait\text{-}set\text{-}status \Rightarrow bool) \rightharpoonup ('t \times 'w\ wait\text{-}set\text{-}status)$
and *is-α* :: $'s\text{-}i \Rightarrow 't\ interrupts$
and *is-invar* :: $'s\text{-}i \Rightarrow bool$
and *is-memb* :: $'t \Rightarrow 's\text{-}i \Rightarrow bool$
and *is-ins* :: $'t \Rightarrow 's\text{-}i \Rightarrow 's\text{-}i$
and *is-delete* :: $'t \Rightarrow 's\text{-}i \Rightarrow 's\text{-}i$
+
fixes *queue-α* :: $'queue \Rightarrow 't\ list$
and *queue-invar* :: $'queue \Rightarrow bool$
and *queue-empty* :: $unit \Rightarrow 'queue$
and *queue-isEmpty* :: $'queue \Rightarrow bool$

and *queue-enqueue* :: $'t \Rightarrow 'queue \Rightarrow 'queue$
and *queue-dequeue* :: $'queue \Rightarrow 't \times 'queue$
and *queue-push* :: $'t \Rightarrow 'queue \Rightarrow 'queue$
begin

definition *queue-rotate1* :: $'queue \Rightarrow 'queue$
where *queue-rotate1* = *case-prod queue-enqueue* ∘ *queue-dequeue*

primrec *enqueue-new-thread* :: $'queue \Rightarrow ('t,'x,'m)$ *new-thread-action* $\Rightarrow 'queue$
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* :: $'queue \Rightarrow ('t,'x,'m)$ *new-thread-action list* $\Rightarrow 'queue$
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 $n = 0 \vee$ *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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *round-robin-start-invar*:
round-robin-invar (*round-robin-start n0 t0*) {*t0*}
⟨*proof*⟩

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* ⟨*proof*⟩

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
⟨*proof*⟩

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* (*λs 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*⟩

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*
⟨*proof*⟩

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* ⟨*proof*⟩

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* ⟨*proof*⟩

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^** x y*
obtains (*refl*) *x* = *y*
| (*trancl*) *r^++ x y*
⟨*proof*⟩

lemma *map-of-map2*: *map-of* (*map* (λ(*k*, *v*). (*k*, *f k v*)) *xs*) *k* = *map-option* (*f k*) (*map-of xs k*)
⟨*proof*⟩

lemma *map-of-map-K*: *map-of* (*map* (λ*k*. (*k*, *c*)) *xs*) *k* = (*if k* ∈ *set xs then Some c else None*)
⟨*proof*⟩

lift-definition *map-values* :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ (*'a*, *'b*) *mapping* ⇒ (*'a*, *'c*) *mapping*
is λ*f m k*. *map-option* (*f k*) (*m k*) ⟨*proof*⟩

lemma *map-values-Mapping* [*simp*]:
map-values f (*Mapping.Mapping m*) = *Mapping.Mapping* (λ*k*. *map-option* (*f k*) (*m k*))

⟨*proof*⟩

lemma *map-Mapping*: *Mapping.map f g* (*Mapping.Mapping m*) = *Mapping.Mapping* (*map-option g* ∘ *m* ∘ *f*)
⟨*proof*⟩

abbreviation *subclst* :: *'m prog* ⇒ *cname* ⇒ *cname* ⇒ *bool*
where *subclst P* ≡ (*subcls1 P*)^++

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* ⟨*proof*⟩

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* $\preceq^*$ *D*} *else None* ⟨*proof*⟩

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 ⟨*proof*⟩

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 ⟨*proof*⟩

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*⟩

lemma *impl-of-ProgImpl* [*simp*]:

wf-prog-impl′ Pfsm ⟹ *impl-of* (*ProgRefine Pfsm*) = *Pfsm*
⟨*proof*⟩

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′*
⟨*proof*⟩

lemma *wf-prog-impl′-impl-of* [*simp*, *intro*!]:
wf-prog-impl′ (*impl-of Pi*) **for** *Pi*
⟨*proof*⟩

lemma *ProgImpl-impl-of* [*simp*, *code abstype*]:
ProgRefine (*impl-of Pi*) = *Pi* **for** *Pi*
⟨*proof*⟩

lemma *program-ProgRefine* [*simp*]: *wf-prog-impl′ Psfm* ⟹ *program* (*ProgRefine Psfm*) = *Program* (*fst Psfm*)
⟨*proof*⟩

lemma *classes-program* [*code*]: *classes* (*program P*) = *fst* (*impl-of P*)
⟨*proof*⟩

lemma *class-program* [*code*]: *class* (*program Pi*) = *Mapping.lookup* (*fst* (*snd* (*impl-of Pi*))) **for** *Pi*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *subcls′-i-i-i-program* [*code*]:
subcls′-i-i-i P C D = (*if subcls′ P C D then Predicate.single* () *else bot*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *Program-code* [*code*]:
Program = *program* ∘ *tabulate-program*
⟨*proof*⟩

class

lemma *tabulate-class-code* [*code*]:
tabulate-class = *Mapping.of-alist*
⟨*proof*⟩

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′ ⟨*proof*⟩

lemma *subcls1-into-subcls1′*:
assumes *subcls1* (*Program P*) *C D*
shows *subcls1′ P C D*

⟨*proof*⟩

lemma *subcls1′-into-subcls1*:
  assumes *subcls1′ P C D*
  shows *subcls1* (*Program P*) *C D*
⟨*proof*⟩

lemma *subcls1-eq-subcls1′*:
  subcls1 (*Program P*) = *subcls1′ P*
⟨*proof*⟩

definition *subcls′′* :: *′m cdecl list* ⇒ *cname* ⇒ *cname* ⇒ *bool*
where *subcls′′ P* = (*subcls1′ P*)^**

code-pred
  (*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *bool*)
  [*inductify*]
  subcls′′ ⟨*proof*⟩

lemma *subcls′′-eq-subcls*: *subcls′′ P* = *subcls* (*Program P*)
⟨*proof*⟩

lemma *subclst-snd-classD*:
  assumes *subclst* (*Program P*) *C D*
  shows *D* ∈ *fst* ‘ *snd* ‘ *set P*
⟨*proof*⟩

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 ())
⟨*proof*⟩

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
)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Fields-into-Fields'*:
assumes *Program P* ⊢ *C has-fields FDTs*
shows *Fields' P C FDTs*
⟨*proof*⟩

lemma *Fields'-eq-Fields*:
Fields' P = *Fields* (*Program P*)
⟨*proof*⟩

code-pred
(*modes*: *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
Fields' ⟨*proof*⟩

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*
⟨*proof*⟩

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* [])

⟨*proof*⟩

lemma *The-Fields* [*simp*]:
$P \vdash C$ *has-fields FDTs* $\Longrightarrow$ *The* (*Fields P C*) = *FDTs*
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *Methods-into-Methods′*:
assumes *Program P* ⊢ *C sees-methods Mm*
shows ∃ *Mm′*. *Methods′ P C Mm′* ∧ *Mm* = *map-of Mm′*
⟨*proof*⟩

code-pred
(*modes*: *i* ⇒ *i* ⇒ *o* ⇒ *bool*)
Methods′
⟨*proof*⟩

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* [])
⟨*proof*⟩

lemma *Methods′-fun*:
assumes *Methods′ P C Mm*
shows *Methods′ P C Mm′* ⟹ *Mm* = *Mm′*
⟨*proof*⟩

lemma *The-Methods'* [*simp*]: *Methods' P C Mm* ⟹ *The* (*Methods' P C*) = *Mm*
⟨*proof*⟩

lemma *methods-def2* [*simp*]: *Methods' P C Mm* ⟹ *methods' P C* = *Mm*
⟨*proof*⟩

lemma *tabulate-Method-code* [*code*]:
tabulate-Method P =
Mapping.tabulate (*map fst P*) (λ*C*. *Mapping.of-alist* (*map* (λ(*M*, (*rest*, *D*)). (*M*, *D*, *rest*)) (*methods' P C*)))
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

declare *compP.simps* [*simp*] *compP.simps*[*symmetric*, *simp del*]

lemma *compP-program* [*code*]:
compP f (*program P*) = *program* (*compP-code f P*)
⟨*proof*⟩

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*⟩

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* ⟨*proof*⟩

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-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-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-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-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′*)⟩)
⟨*proof*⟩

lemma *sc-J-start-state-invar*: (λ-. *True*) (*sc-state-α* (*sc-J-start-state-refine P C M vs*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

9.6.2 Random scheduler

interpretation *J-rnd*:
sc-random-scheduler-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*
⟨*proof*⟩

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-Fii-i-oB-Fii-i-i-oB-i-i-i-i-i-o-o-o P convert-RA Jinja-output
for *P*
⟨*proof*⟩

interpretation *J-rnd*:
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-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*
⟨*proof*⟩

⟨*ML*⟩

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
$\lambda h\ a\ ad\ v.\ v \in$ *heap-read h a ad* $\lambda h\ a\ ad\ v\ h'.\ h' \in$ *heap-write h a ad v*
⟨*proof*⟩

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'*)}
})
⟨*proof*⟩

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''*)}
})
⟨*proof*⟩

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'*)⌋)}
  }
| - ⇒ {})
⟨*proof*⟩

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 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)})
⟨proof⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *frames′-ok-inverse* [*simp*]:
∀ *x*∈*set frs*. *frame′-ok P x* ⟹ *map* (*frame′-of-frame P* ∘ *frame-of-frame′*) *frs* = *frs*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *jvm-thread-action′-ok-ε* [*simp*]: *jvm-thread-action′-ok P ε*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *frame-of-frame′-inverse* [*simp*]: *frame-of-frame′* ∘ *frame′-of-frame P* = *id*
⟨*proof*⟩

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-jvm-thread-*
P) = *id*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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* C_0 M_0 *pc frs* =
(*let v* = *hd stk*
in if v = *Null then* {(ε, ⌊*execute.addr-of-sys-xcpt NullPointer*⌋, *h*, ((*ins′*, *ins*, *xt*), *stk*, *loc*, C_0, M_0, *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*, C_0, M_0, *pc* + *1*) # *frs*)}
})
| *exec-instr ins′ ins xt* (*Putfield F C*) *P t h stk loc* C_0 M_0 *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*, C_0, M_0, *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*, C_0, M_0, *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 C_0 M_0 pc frs =
{(ε, let U = the ($typeof_h$ (hd stk))
in if P ⊢ U ≤ T then (None, h, ((tl ins', ins, xt), stk, loc, C_0, M_0, pc + 1) # frs)
else (⌊execute.addr-of-sys-xcpt ClassCast⌋, h, ((ins', ins, xt), stk, loc, C_0, M_0, pc) # frs))}
| exec-instr ins' ins xt (Instanceof T) P t h stk loc C_0 M_0 pc frs =
{(ε, None, h, ((tl ins', ins, xt), Bool (hd stk ≠ Null ∧ P ⊢ the ($typeof_h$ (hd stk)) ≤ T) # tl stk, loc, C_0, M_0, 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,mxl_0,ins'',xt'')⌋ ⇒
let f' = ((ins'', ins'', xt''), [],[r]@ps@(replicate mxl_0 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 stk_0 loc_0 C_0 M_0 pc frs =
{(ε, (if frs=[] then (None, h, [])
else
let v = hd stk_0;
((ins', ins, xt), stk,loc,C,m,pc) = hd frs;
n = length (fst (snd (method P C_0 M_0)))
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 C_0 M_0 pc frs =
{(ε, (None, h, ((tl ins', ins, xt), tl stk, loc, C_0, M_0, pc+1)#frs))}
| exec-instr ins' ins xt Dup P t h stk loc C_0 M_0 pc frs =
{(ε, (None, h, ((tl ins', ins, xt), hd stk # stk, loc, C_0, M_0, pc+1)#frs))}
| exec-instr ins' ins xt Swap P t h stk loc C_0 M_0 pc frs =
{(ε, (None, h, ((tl ins', ins, xt), hd (tl stk) # hd stk # tl (tl stk), loc, C_0, M_0, 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* C_0 M_0 *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*, C_0, M_0, *pc'*)#*frs*)))}
| *exec-instr ins' ins xt* (*Goto i*) *P t h stk loc* C_0 M_0 *pc frs* =
{*let pc'* = *nat*(*int pc*+*i*)
in (ε, (*None*, *h*, ((*drop pc' ins*, *ins*, *xt*), *stk*, *loc*, C_0, M_0, *pc'*)#*frs*))}
| *exec-instr ins' ins xt ThrowExc P t h stk loc* C_0 M_0 *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*, C_0, M_0, *pc*)#*frs*)))}
| *exec-instr ins' ins xt MEnter P t h stk loc* C_0 M_0 *pc frs* =
{*let v* = *hd stk*
in if v = *Null*
then (ε, ⌊*execute.addr-of-sys-xcpt NullPointer*⌋, *h*, ((*ins'*, *ins*, *xt*), *stk*, *loc*, C_0, M_0, *pc*) # *frs*)
else (⦃*Lock*→*the-Addr v*, *SyncLock* (*the-Addr v*)⦄, *None*, *h*, ((*tl ins'*, *ins*, *xt*), *tl stk*, *loc*, C_0, M_0, *pc* + *1*) # *frs*)}
| *exec-instr ins' ins xt MExit P t h stk loc* C_0 M_0 *pc frs* =
(*let v* = *hd stk*
in if v = *Null*
then {(ε, ⌊*execute.addr-of-sys-xcpt NullPointer*⌋, *h*, ((*ins'*, *ins*, *xt*), *stk*, *loc*, C_0, M_0, *pc*)#*frs*)}
else {(⦃*Unlock*→*the-Addr v*, *SyncUnlock* (*the-Addr v*)⦄, *None*, *h*, ((*tl ins'*, *ins*, *xt*), *tl stk*, *loc*, C_0, M_0, *pc* + *1*) # *frs*),
(⦃*UnlockFail*→*the-Addr v*⦄, ⌊*execute.addr-of-sys-xcpt IllegalMonitorState*⌋, *h*, ((*ins'*, *ins*, *xt*), *stk*, *loc*, C_0, M_0, *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*⟩

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*⟩

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*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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 $\bigwedge h\ ad\ al\ v.\ v \in$ *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*⟩

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*⟩

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, (λt. 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 (λt (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*
⟨*proof*⟩

lemma *jvm-mstate-of-jvm-mstate′-sc-start-state* [*simp*]:
jvm-mstate-of-jvm-mstate′
(sc-start-state (λC 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
⟨*proof*⟩

lemma *sc-jvm-start-state-invar*:
assumes $wf\text{-}jvm\text{-}prog_{\Phi}$ *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*
⟨*proof*⟩

lemma *invariant3p-sc-jvm-state-invar*:
assumes $wf\text{-}jvm\text{-}prog_{\Phi}$ *P*
shows *invariant3p (multithreaded-base.redT 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*⟩

lemma *sc-exec-deterministic*:
assumes $wf\text{-}jvm\text{-}prog_{\Phi}$ *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*⟩

9.8.1 Round-robin scheduler

interpretation *JVM-rr*:
sc-round-robin-base
JVM-final′ sc-mexec P convert-RA Jinja-output
for *P*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *JVM-rr*:
assumes $wf\text{-}jvm\text{-}prog_{\Phi}$ *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)
⟨*proof*⟩

9.8.2 Random scheduler

interpretation *JVM-rnd*:
sc-random-scheduler-base
JVM-final' sc-mexec P convert-RA Jinja-output
for *P*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *JVM-rnd*:
assumes $wf\text{-}jvm\text{-}prog_{\Phi}$ *P*

shows
sc-scheduler
JVM-final′ (*sc-mexec P*) *convert-RA*
(*JVM-rnd.random-scheduler P*) (*pick-wakeup-via-sel* (*λs P. rm-sel s* (*λ*(*k*,*v*). *P k v*))) (*λ- -. True*)
(*sc-jvm-state-invar P* Φ)
⟨*proof*⟩

⟨*ML*⟩

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*)
⟨*proof*⟩

equations on predicate operations for code inlining

lemma *eq-i-o-conv-single*: *eq-i-o* = *Predicate.single*
⟨*proof*⟩

lemma *eq-o-i-conv-single*: *eq-o-i* = *Predicate.single*
⟨*proof*⟩

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′*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *sup-bot1*: *sup-class.sup bot a* = (*a* :: *'a* :: {*semilattice-sup*, *order-bot*})
⟨*proof*⟩

lemma *sup-bot2*: *sup-class.sup a bot* = (*a* :: *'a* :: {*semilattice-sup*, *order-bot*})
⟨*proof*⟩

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*))
⟨*proof*⟩

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'*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

```
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,mxl_0,ins,xt)⌋ ⇒
let f' = ([],[r]@ps@(replicate mxl_0 undefined-value),D,M,0)
in {(ε, None, h, f' # (stk, loc, C0, M0, pc) # frs)})))
exec-instr Return P t h stk_0 loc_0 C_0 M_0 pc frs =
{(ε, (if frs=[] then (None, h, [])
else
let v = hd stk_0;
(stk,loc,C,m,pc) = hd frs;
n = length (fst (snd (method P C_0 M_0)))
in (None, h, (v#(drop (n+1) stk),loc,C,m,pc+1)#tl frs)))}
exec-instr Pop P t h stk loc C_0 M_0 pc frs = {(ε, (None, h, (tl stk, loc, C_0, M_0, pc+1)#frs))}
exec-instr Dup P t h stk loc C_0 M_0 pc frs = {(ε, (None, h, (hd stk # stk, loc, C_0, M_0, pc+1)#frs))}
exec-instr Swap P t h stk loc C_0 M_0 pc frs = {(ε, (None, h, (hd (tl stk) # hd stk # tl (tl stk), loc, C_0, M_0, 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 C_0 M_0 pc frs =
{(ε, (let pc' = if hd stk = Bool False then nat(int pc+i) else pc+1
in (None, h, (tl stk, loc, C_0, M_0, pc')#frs)))}
exec-instr (Goto i) P t h stk loc C_0 M_0 pc frs = {(ε, (None, h, (stk, loc, C_0, M_0, nat(int pc+i))#frs))}
exec-instr ThrowExc P t h stk loc C_0 M_0 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, C_0, M_0, pc)#frs)))}
exec-instr MEnter P t h stk loc C_0 M_0 pc frs =
{(let v = hd stk
in if v = Null
then (ε, ⌊execute.addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C_0, M_0, pc) # frs)
else (⦃Lock→the-Addr v, SyncLock (the-Addr v)⦄, None, h, (tl stk, loc, C_0, M_0, pc + 1) # frs))}
exec-instr MExit P t h stk loc C_0 M_0 pc frs =
(let v = hd stk
in if v = Null
then {(ε, ⌊execute.addr-of-sys-xcpt NullPointer⌋, h, (stk, loc, C_0, M_0, 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⦄, ⌊execute.addr-of-sys-xcpt IllegalMonitorState⌋, h, (stk, loc, C_0, M_0, pc) # frs)})
⟨proof⟩

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 (⌊a⌋, h, fr # frs) = {(ε, execute.exception-step P a h fr frs)}
⟨proof⟩

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'*)
⟨*proof*⟩

lemma *exec-1E*:
assumes *Predicate.eval* (*exec-1 P t σ*) (*ta*, *σ'*)
obtains *execute.exec-1 P t σ ta σ'*
⟨*proof*⟩

lemma *exec-1-eq* [*simp*]:
Predicate.eval (*exec-1 P t σ*) (*ta*, *σ'*) ⟷ *execute.exec-1 P t σ ta σ'*
⟨*proof*⟩

lemma *exec-1-eq'*:
Predicate.eval (*exec-1 P t σ*) = (λ(*ta*, *σ'*). *execute.exec-1 P t σ ta σ'*)
⟨*proof*⟩

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 $\bigwedge h\ ad\ al\ v.\ v \in$ *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*⟩

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*
⟨*proof*⟩

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'*))
⟨*proof*⟩

lemma *sc-jvm-start-state-invar*:
assumes $wf\text{-}jvm\text{-}prog_\Phi$ *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
⟨*proof*⟩

9.9.2 Round-robin scheduler

interpretation *JVM-rr*:
  sc-round-robin-base
    JVM-final sc-mexec P convert-RA Jinja-output
  for *P*
⟨*proof*⟩

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*
⟨*proof*⟩

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)
⟨*proof*⟩

9.9.3 Random scheduler

interpretation *JVM-rnd*:
  sc-random-scheduler-base
    JVM-final sc-mexec P convert-RA Jinja-output
  for *P*
⟨*proof*⟩

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*
⟨*proof*⟩

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)
⟨*proof*⟩

⟨*ML*⟩

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*⟩
end

instantiation *char* :: *toString* **begin**
definition [*code*]: *toString* (*c* :: *char*) = *String.implode* [*c*]
instance ⟨*proof*⟩
end

instantiation *String.literal* :: *toString* **begin**
definition [*code*]: *toString* (*s* :: *String.literal*) = *s*
instance ⟨*proof*⟩
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⟩
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)])
⟨proof⟩
termination ⟨proof⟩

instantiation int :: toString begin
definition [code]: toString i = sum-list (int-toString i)
instance ⟨proof⟩
end

instantiation nat :: toString begin
definition [code]: toString n = toString (int n)
instance ⟨proof⟩
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⟩
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 (λx. [STR '','', toString x, STR ''|−>'', toString (f $ x)]) (finfun-to-list f))
     @ [STR '')''])
instance ⟨proof⟩
end
```

instantiation *word* :: (*len*) *toString* **begin**
definition [*code*]: *toString* (*w* :: *'a word*) = *toString* (*sint w*)
instance ⟨*proof*⟩
end

instantiation *fun* :: (*type*, *type*) *toString* **begin**
definition [*code*]: *toString* (*f* :: *'a* ⇒ *'b*) = *STR ''fn''*
instance ⟨*proof*⟩
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*⟩
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*⟩
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*⟩

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*⟩
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*⟩
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*⟩
end

instantiation *prod* :: (*toString*, *toString*) *toString* **begin**
definition *toString* = (λ(*a*, *b*). *sum-list* [*STR* ''('', *toString a*, *STR* '', '', *toString b*, *STR* '')''])
instance ⟨*proof*⟩
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*⟩
end

instantiation *unit* :: *toString* **begin**
definition *toString* (*u* :: *unit*) = *STR* ''()''
instance ⟨*proof*⟩
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⟩
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*⟩
end

instantiation *trie* :: (*toString*, *toString*) *toString* **begin**
definition [*code*]: *toString* (*t* :: (*'a*, *'b*) *trie*) = *toString* (*tm-to-list t*)
instance ⟨*proof*⟩
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*⟩
end

instantiation *assoc-list* :: (*toString*, *toString*) *toString* **begin**
definition [*code*]: *toString* = *toString* ∘ *Assoc-List.impl-of*
instance ⟨*proof*⟩
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*⟩
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

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*, *ApprenticeC*])

definition *ApprenticeChallenge-annotated*
where *ApprenticeChallenge-annotated* = *annotate-prog-code ApprenticeChallenge*

lemma *wf-J-prog ApprenticeChallenge-annotated*
⟨*proof*⟩

lemmas [*code-unfold*] =
Container-def Job-def String-def Apprentice-def

definition *main* :: *String.literal* **where** *main* = *STR* ''*main*''

⟨*ML*⟩

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*)
⟨*proof*⟩

definition *main* **where** *main = STR ''main''*
definition *five* :: *int* **where** *five = 5*

⟨*ML*⟩

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.