Regression Test Selection over JVM

Susannah Mansky

March 17, 2025

Abstract

This development provides a general definition for safe Regression Test Selection (RTS) algorithms. RTS algorithms select which tests to rerun on revised code, reducing the time required to check for newly introduced errors. An RTS algorithm is considered safe if and only if all deselected tests would have unchanged results.

This definition is instantiated with two class-collection-based RTS algorithms run over the JVM as modeled by JinjaDCI. This is achieved with a general definition for Collection Semantics, small-step semantics instrumented to collect information during execution. As the RTS definition mandates safety, these instantiations include proofs of safety.

This work is described in Mansky and Gunter's LSFA 2020 paper [1] and Mansky's doctoral thesis [2].

Contents

1 Theory Dependencies **3**

2 Regression Test Selection algorithm model **4**

3 Semantics model **5**
3.1 Extending *small* to multiple steps 6
3.2 Extending *small* to a big-step semantics 7

4 Collection Semantics **8**
4.1 Small-Step Collection Semantics 9
4.2 Extending *csmall* to multiple steps 9
4.3 Extending *csmall* to a big-step semantics 12

5 Collection-based RTS **14**

6 Instantiating *Semantics* with Jinja JVM **15**

7 *classes-changed* theory **15**

8 *subcls* theory **17**

9 *classes-above* theory **18**
9.1 Methods . . . 20
9.2 Fields . . . 21
9.3 Other . . . 24

10 Instantiating *CollectionSemantics* with Jinja JVM **24**
10.1 JVM-specific *classes-above* theory . . . 24
10.2 Naive RTS algorithm . . . 26
10.3 Smarter RTS algorithm . . . 27
10.4 A few lemmas using the instantiations . . . 27

11 Inductive JVM execution **29**
11.1 Equivalence of *exec-step* and *exec-step-input* . . . 37

12 Instantiating *CollectionBasedRTS* with Jinja JVM **49**
12.1 Some *classes-above* lemmas . . . 49
12.2 JVM next-step lemmas for initialization calling . . . 50
12.3 Definitions . . . 52
12.4 Definition lemmas . . . 53
12.5 Naive RTS algorithm . . . 54
12.5.1 Definitions . . . 54
12.5.2 Naive algorithm correctness . . . 54
12.6 Smarter RTS algorithm . . . 66
12.6.1 Definitions and helper lemmas . . . 66
12.6.2 Additional well-formedness conditions . . . 67
12.6.3 Proving naive $\subseteq$ smart . . . 69
12.6.4 Proving smart $\subseteq$ naive . . . 98
12.6.5 Safety of the smart algorithm . . . 100

1 Theory Dependencies

Figure 1 shows the dependencies between the Isabelle theories in the following sections.

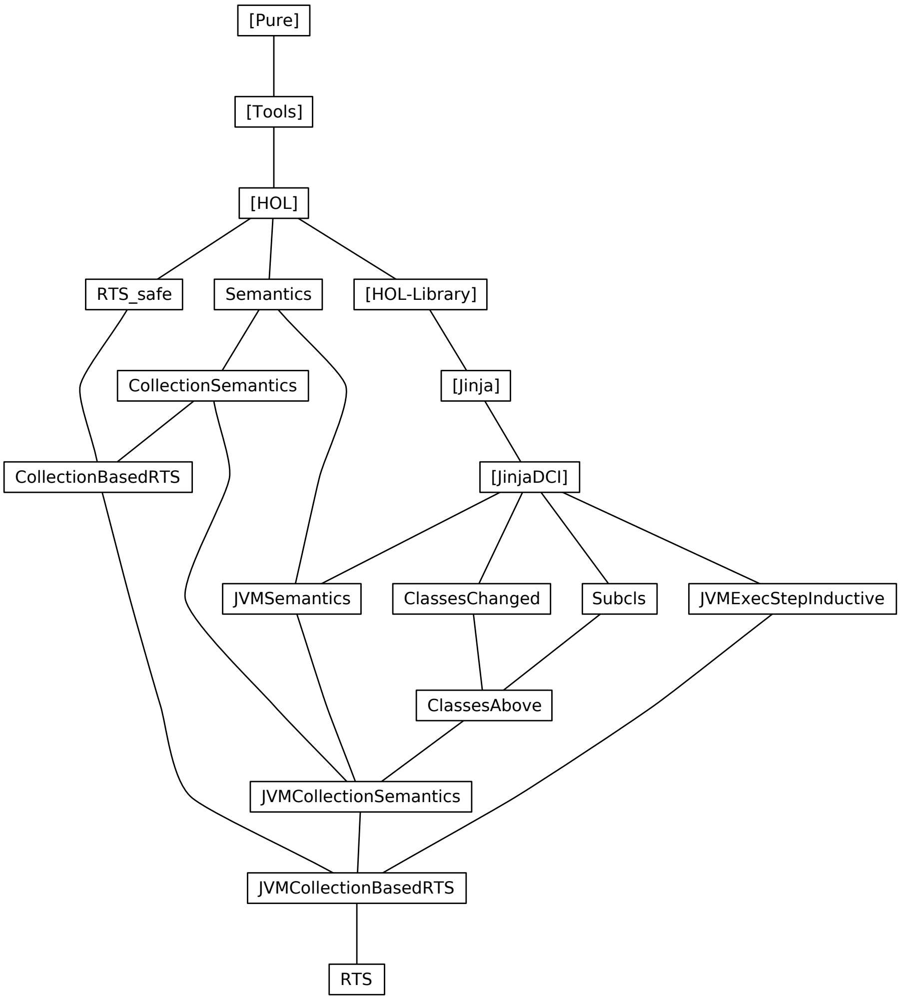

Figure 1: Theory Dependency Graph

2 Regression Test Selection algorithm model

theory *RTS-safe*
imports *Main*
begin

This describes an *existence safe* RTS algorithm: if a test is deselected based on an output, there is SOME equivalent output under the changed program.

locale *RTS-safe* =
fixes
out :: $'prog \Rightarrow 'test \Rightarrow 'prog\text{-}out\ set$ **and**
equiv-out :: $'prog\text{-}out \Rightarrow 'prog\text{-}out \Rightarrow bool$ **and**
deselect :: $'prog \Rightarrow 'prog\text{-}out \Rightarrow 'prog \Rightarrow bool$ **and**
progs :: $'prog\ set$ **and**
tests :: $'test\ set$
assumes
existence-safe: $[\![P \in progs;\ P' \in progs;\ t \in tests;\ o1 \in out\ P\ t;\ deselect\ P\ o1\ P']\!]$
$\Longrightarrow (\exists\, o2 \in out\ P'\ t.\ equiv\text{-}out\ o1\ o2)$ **and**
equiv-out-equiv: $equiv\ UNIV\ \{(x,y).\ equiv\text{-}out\ x\ y\}$ **and**
equiv-out-deselect: $[\![equiv\text{-}out\ o1\ o2;\ deselect\ P\ o1\ P']\!] \Longrightarrow deselect\ P\ o2\ P'$

context *RTS-safe* **begin**

lemma *equiv-out-refl*: *equiv-out a a*
using *equiv-class-eq-iff equiv-out-equiv* **by** *fastforce*

lemma *equiv-out-trans*: $[\![equiv\text{-}out\ a\ b;\ equiv\text{-}out\ b\ c]\!] \Longrightarrow equiv\text{-}out\ a\ c$
using *equiv-class-eq-iff equiv-out-equiv* **by** *fastforce*

This shows that it is safe to continue deselecting a test based on its output under a previous program, to an arbitrary number of program changes, as long as the test is continually deselected. This is useful because it means changed programs don't need to generate new outputs for deselected tests to ensure safety of future deselections.

lemma *existence-safe-trans*:
assumes *Pst-in*: $Ps \neq []\ set\ Ps \subseteq progs\ t \in tests$ **and**
o0: $o_0 \in out\ (Ps!0)\ t$ **and**
des: $\forall\, n < (length\ Ps) - 1.\ deselect\ (Ps!n)\ o_0\ (Ps!(Suc\ n))$
shows $\exists\, o_n \in out\ (last\ Ps)\ t.\ equiv\text{-}out\ o_0\ o_n$
using *assms* **proof**(*induct length Ps arbitrary*: *Ps*)
  case *0* **with** *Pst-in* **show** *?case* **by** *simp*
next
  case (*Suc x*) **then show** *?case*
  proof(*induct x*)
  case *z*: *0*
    from *z.prems*(*2*,*3*) **have** $Ps\ !\ (length\ Ps - 2) = last\ Ps$
      by (*simp add*: *last-conv-nth numeral-2-eq-2*)
    with *equiv-out-refl z.prems*(*2*,*6*) **show** *?case* **by** *auto*
  next

case *Suc′*:(*Suc x′*)
let *?Ps* = *take* (*Suc x′*) *Ps*
have *len′*: *Suc x′* = *length* (*take* (*Suc x′*) *Ps*) **using** *Suc′.prems*(*2*) **by** *auto*
moreover have *nmt′*: *take* (*Suc x′*) *Ps* ≠ [] **using** *len′* **by** *auto*
moreover have *sub′*: *set* (*take* (*Suc x′*) *Ps*) ⊆ *progs* **using** *Suc.prems*(*2*)
by (*meson order-trans set-take-subset*)
moreover have *t* ∈ *tests* **using** *Pst-in*(*3*) **by** *simp*
moreover have o_0 ∈ *out* (*take* (*Suc x′*) *Ps* ! *0*) *t* **using** *Suc.prems*(*4*) **by** *simp*
moreover have ∀ *n*<*length* (*take* (*Suc x′*) *Ps*) − *1*.
deselect (*take* (*Suc x′*) *Ps* ! *n*) o_0 (*take* (*Suc x′*) *Ps* ! (*Suc n*))
using *Suc.prems*(*5*) *len′* **by** *simp*
ultimately have ∃ *o′*∈*out* (*last ?Ps*) *t*. *equiv-out* o_0 *o′* **by**(*rule Suc′.prems*(*1*)[*of ?Ps*])
then obtain *o′* **where** *o′*: *o′* ∈ *out* (*last ?Ps*) *t* **and** *eo*: *equiv-out* o_0 *o′* **by** *clarify*
from *Suc.prems*(*1*) *Suc′.prems*(*2*) *len′* *nmt′*
have *last* (*take* (*Suc x′*) *Ps*) = *Ps*!*x′* *last Ps* = *Ps*!(*Suc x′*)
by (*metis diff-Suc-1 last-conv-nth lessI nth-take*)+
moreover have *x′* < *length Ps* − *1* **using** *Suc′.prems*(*2*) **by** *linarith*
ultimately have *des′*:*deselect* (*last* (*take* (*Suc x′*) *Ps*)) o_0 (*last Ps*)
using *Suc.prems*(*5*) **by** *simp*
from *Suc.prems*(*1*,*2*) *sub′* *nmt′* *last-in-set*
have *Ps-in*: *last* (*take* (*Suc x′*) *Ps*) ∈ *progs* *last Ps* ∈ *progs* **by** *blast*+
have ∃ o_n ∈ *out* (*last Ps*) *t*. *equiv-out o′* o_n
by(*rule existence-safe*[**where** *P*=*last* (*take* (*Suc x′*) *Ps*) **and** *P′*=*last Ps* **and** *t*=*t*,
OF Ps-in Pst-in(*3*) *o′* *equiv-out-deselect*[*OF eo des′*]])
then obtain o_n **where** *oN*: o_n ∈ *out* (*last Ps*) *t* **and** *eo′*: *equiv-out o′* o_n **by** *clarify*
moreover from *eo eo′* **have** *equiv-out* o_0 o_n **by**(*rule equiv-out-trans*)
ultimately show *?case* **by** *auto*
qed
qed

end — *RTS-safe*

end

3 Semantics model

theory *Semantics*
imports *Main*
begin

General model for small-step semantics:

locale *Semantics* =
fixes
small :: *′prog* ⇒ *′state* ⇒ *′state set* **and**
endset :: *′state set*

assumes
endset-final: $\sigma \in endset \Longrightarrow \forall P.\ small\ P\ \sigma = \{\}$

context *Semantics* **begin**

3.1 Extending *small* to multiple steps

primrec *small-nstep* :: *'prog* ⇒ *'state* ⇒ *nat* ⇒ *'state set* **where**
small-nstep-base:
small-nstep P σ 0 = {σ} |
small-nstep-Rec:
small-nstep P σ (*Suc n*) =
{ *σ2*. ∃*σ1*. *σ1* ∈ *small-nstep P σ n* ∧ *σ2* ∈ *small P σ1* }

lemma *small-nstep-Rec2*:
small-nstep P σ (*Suc n*) =
{ *σ2*. ∃*σ1*. *σ1* ∈ *small P σ* ∧ *σ2* ∈ *small-nstep P σ1 n* }
proof(*induct n arbitrary*: *σ*)
case (*Suc n*)
have *right*: ⋀*σ'*. *σ'* ∈ *small-nstep P σ* (*Suc*(*Suc n*))
⟹ ∃*σ1*. *σ1* ∈ *small P σ* ∧ *σ'* ∈ *small-nstep P σ1* (*Suc n*)
proof −
fix *σ'*
assume *σ'* ∈ *small-nstep P σ* (*Suc*(*Suc n*))
then obtain *σ1* **where** *Sucnstep*: *σ1* ∈ *small-nstep P σ* (*Suc n*) *σ'* ∈ *small P*
σ1 **by** *fastforce*
obtain *σ12* **where** *nstep*: *σ12* ∈ *small P σ* ∧ *σ1* ∈ *small-nstep P σ12 n*
using *Suc Sucnstep*(*1*) **by** *fastforce*
then show ∃*σ1*. *σ1* ∈ *small P σ* ∧ *σ'* ∈ *small-nstep P σ1* (*Suc n*)
using *Sucnstep* **by** *fastforce*
qed
have *left*: ⋀*σ'* . ∃*σ1*. *σ1* ∈ *small P σ* ∧ *σ'* ∈ *small-nstep P σ1* (*Suc n*)
⟹ *σ'* ∈ *small-nstep P σ* (*Suc*(*Suc n*))
proof −
fix *σ'*
assume ∃*σ1*. *σ1* ∈ *small P σ* ∧ *σ'* ∈ *small-nstep P σ1* (*Suc n*)
then obtain *σ1* **where** *Sucnstep*: *σ1* ∈ *small P σ σ'* ∈ *small-nstep P σ1* (*Suc*
n)
by *fastforce*
obtain *σ12* **where** *nstep*: *σ12* ∈ *small-nstep P σ1 n* ∧ *σ'* ∈ *small P σ12*
using *Sucnstep*(*2*) **by** *auto*
then show *σ'* ∈ *small-nstep P σ* (*Suc*(*Suc n*)) **using** *Suc Sucnstep* **by** *fastforce*
qed
show *?case* **using** *right left* **by** *fast*
qed(*simp*)

lemma *small-nstep-SucD*:
assumes *σ'* ∈ *small-nstep P σ* (*Suc n*)
shows ∃*σ1*. *σ1* ∈ *small P σ* ∧ *σ'* ∈ *small-nstep P σ1 n*

using *small-nstep-Rec2 case-prodD assms* **by** *fastforce*

lemma *small-nstep-Suc-nend*: $\sigma' \in$ *small-nstep P* σ (*Suc n1*) $\Longrightarrow \sigma \notin$ *endset*
using *endset-final small-nstep-SucD* **by** *fastforce*

3.2 Extending *small* to a big-step semantics

definition *big* :: *'prog* $\Rightarrow$ *'state* $\Rightarrow$ *'state set* **where**
big P $\sigma \equiv \{ \sigma'. \exists n. \sigma' \in$ *small-nstep P* σ *n* $\wedge \sigma' \in$ *endset* $\}$

lemma *bigI*:
$[\![\sigma' \in$ *small-nstep P* σ *n*; $\sigma' \in$ *endset* $]\!] \Longrightarrow \sigma' \in$ *big P* σ
by (*fastforce simp add*: *big-def*)

lemma *bigD*:
$[\![\sigma' \in$ *big P* σ $]\!] \Longrightarrow \exists n. \sigma' \in$ *small-nstep P* σ *n* $\wedge \sigma' \in$ *endset*
by (*simp add*: *big-def*)

lemma *big-def2*:
$\sigma' \in$ *big P* $\sigma \longleftrightarrow (\exists n. \sigma' \in$ *small-nstep P* σ *n* $\wedge \sigma' \in$ *endset*)
proof(*rule iffI*)
assume $\sigma' \in$ *big P* σ
then show $\exists n. \sigma' \in$ *small-nstep P* σ *n* $\wedge \sigma' \in$ *endset* **using** *bigD big-def* **by** *auto*
next
assume $\exists n. \sigma' \in$ *small-nstep P* σ *n* $\wedge \sigma' \in$ *endset*
then show $\sigma' \in$ *big P* σ **using** *big-def big-def* **by** *auto*
qed

lemma *big-stepD*:
assumes *big*: $\sigma' \in$ *big P* σ **and** *nend*: $\sigma \notin$ *endset*
shows $\exists \sigma 1. \sigma 1 \in$ *small P* $\sigma \wedge \sigma' \in$ *big P* $\sigma 1$
proof –
obtain *n* **where** *n*: $\sigma' \in$ *small-nstep P* σ *n* $\sigma' \in$ *endset*
using *big-def2 big* **by** *auto*
then show *?thesis* **using** *small-nstep-SucD nend big-def2* **by**(*cases n*, *simp*) *blast*
qed

lemma *small-nstep-det-last-eq*:
assumes *det*: $\forall \sigma$. *small P* $\sigma = \{\} \vee (\exists \sigma'$. *small P* $\sigma = \{\sigma'\})$
shows $[\![\sigma' \in$ *big P* σ; $\sigma' \in$ *small-nstep P* σ *n*; $\sigma' \in$ *small-nstep P* σ n' $]\!] \Longrightarrow n = n'$
proof(*induct n arbitrary*: n' σ σ')
case *0*
have $\sigma' = \sigma$ **using** *0.prems*(*2*) *small-nstep-base* **by** *blast*
then have *endset*: $\sigma \in$ *endset* **using** *0.prems*(*1*) *bigD* **by** *blast*
show *?case*

proof(*cases n′*)
case *Suc* **then show** *?thesis* **using** *0.prems*(*3*) *small-nstep-SucD endset-final*[*OF endset*] **by** *blast*
qed(*simp*)
next
case (*Suc n1*)
then have *endset*: $\sigma' \in$ *endset* **using** *Suc.prems*(*1*) *bigD* **by** *blast*
have *nend*: $\sigma \notin$ *endset* **using** *small-nstep-Suc-nend*[*OF Suc.prems*(*2*)] **by** *simp*
then have *neq*: $\sigma' \neq \sigma$ **using** *endset* **by** *auto*
obtain *σ1* **where** *σ1*: *σ1* ∈ *small P σ σ′* ∈ *small-nstep P σ1 n1*
using *small-nstep-SucD*[*OF Suc.prems*(*2*)] **by** *clarsimp*
then have *big*: *σ′* ∈ *big P σ1* **using** *endset* **by**(*auto simp*: *big-def*)
show *?case*
proof(*cases n′*)
case *0* **then show** *?thesis* **using** *neq Suc.prems*(*3*) **using** *small-nstep-base* **by** *blast*
next
case *Suc′*: (*Suc n1′*)
then obtain *σ1′* **where** *σ1′*: *σ1′* ∈ *small P σ σ′* ∈ *small-nstep P σ1′ n1′*
using *small-nstep-SucD*[**where** *σ*=*σ* **and** *σ′*=*σ′* **and** *n*=*n1′*] *Suc.prems*(*3*) **by** *blast*
then have *σ1*=*σ1′* **using** *σ1*(*1*) *det* **by** *auto*
then show *?thesis* **using** *Suc.hyps*(*1*)[*OF big σ1*(*2*)] *σ1′*(*2*) *Suc′* **by** *blast*
qed
qed

end — Semantics

end

4 Collection Semantics

theory *CollectionSemantics*
imports *Semantics*
begin

General model for small step semantics instrumented with an information collection mechanism:

locale *CollectionSemantics* = *Semantics* +
constrains
small :: *′prog* ⇒ *′state* ⇒ *′state set* **and**
endset :: *′state set*
fixes
collect :: *′prog* ⇒ *′state* ⇒ *′state* ⇒ *′coll* **and**
combine :: *′coll* ⇒ *′coll* ⇒ *′coll* **and**
collect-id :: *′coll*
assumes
combine-assoc: *combine* (*combine c1 c2*) *c3* = *combine c1* (*combine c2 c3*) **and**

collect-idl[*simp*]: *combine collect-id c* = *c* **and**
collect-idr[*simp*]: *combine c collect-id* = *c*

context *CollectionSemantics* **begin**

4.1 Small-Step Collection Semantics

definition *csmall* :: *'prog* ⇒ *'state* ⇒ (*'state* × *'coll*) *set* **where**
csmall P σ ≡ { (*σ'*, *coll*). *σ'* ∈ *small P σ* ∧ *collect P σ σ'* = *coll* }

lemma *small-det-csmall-det*:
assumes ∀*σ*. *small P σ* = {} ∨ (∃*σ'*. *small P σ* = {*σ'*})
shows ∀*σ*. *csmall P σ* = {} ∨ (∃*o'*. *csmall P σ* = {*o'*})
using *assms* **by**(*fastforce simp*: *csmall-def*)

4.2 Extending *csmall* to multiple steps

primrec *csmall-nstep* :: *'prog* ⇒ *'state* ⇒ *nat* ⇒ (*'state* × *'coll*) *set* **where**
csmall-nstep-base:
csmall-nstep P σ 0 = {(*σ*, *collect-id*)} |
csmall-nstep-Rec:
csmall-nstep P σ (*Suc n*) =
{ (*σ2*, *coll*). ∃*σ1 coll1 coll2*. (*σ1*, *coll1*) ∈ *csmall-nstep P σ n*
∧ (*σ2*, *coll2*) ∈ *csmall P σ1* ∧ *combine coll1 coll2* = *coll* }

lemma *small-nstep-csmall-nstep-equiv*:
small-nstep P σ n
= { *σ'*. ∃*coll*. (*σ'*, *coll*) ∈ *csmall-nstep P σ n* }
proof (*induct n*) **qed**(*simp-all add*: *csmall-def*)

lemma *csmall-nstep-exists*:
σ' ∈ *big P σ* ⟹ ∃*n coll*. (*σ'*, *coll*) ∈ *csmall-nstep P σ n* ∧ *σ'* ∈ *endset*
proof(*drule bigD*) **qed**(*clarsimp simp*: *small-nstep-csmall-nstep-equiv*)

lemma *csmall-det-csmall-nstep-det*:
assumes ∀*σ*. *csmall P σ* = {} ∨ (∃*o'*. *csmall P σ* = {*o'*})
shows ∀*σ*. *csmall-nstep P σ n* = {} ∨ (∃*o'*. *csmall-nstep P σ n* = {*o'*})
using *assms*
proof(*induct n*)
case (*Suc n*) **then show** *?case* **by** *fastforce*
qed(*simp*)

lemma *csmall-nstep-Rec2*:
csmall-nstep P σ (*Suc n*) =
{ (*σ2*, *coll*). ∃*σ1 coll1 coll2*. (*σ1*, *coll1*) ∈ *csmall P σ*
∧ (*σ2*, *coll2*) ∈ *csmall-nstep P σ1 n* ∧ *combine coll1 coll2* = *coll*
}
proof(*induct n arbitrary*: *σ*)
case (*Suc n*)
have *right*: ⋀*σ' coll'*. (*σ'*, *coll'*) ∈ *csmall-nstep P σ* (*Suc*(*Suc n*))

$\Longrightarrow \exists \sigma 1\ coll1\ coll2.\ (\sigma 1,\ coll1) \in csmall\ P\ \sigma$
$\wedge\ (\sigma',\ coll2) \in csmall\text{-}nstep\ P\ \sigma 1\ (Suc\ n) \wedge combine\ coll1\ coll2 = coll'$
proof −
fix $\sigma'\ coll'$
assume $(\sigma',\ coll') \in csmall\text{-}nstep\ P\ \sigma\ (Suc(Suc\ n))$
then obtain $\sigma 1\ coll1\ coll2$ **where** *Sucnstep*: $(\sigma 1,\ coll1) \in csmall\text{-}nstep\ P\ \sigma\ (Suc\ n)$
$(\sigma',\ coll2) \in csmall\ P\ \sigma 1\ combine\ coll1\ coll2 = coll'$ **by** *fastforce*
obtain $\sigma 12\ coll12\ coll22$ **where** *nstep*: $(\sigma 12,\ coll12) \in csmall\ P\ \sigma$
$\wedge\ (\sigma 1,\ coll22) \in csmall\text{-}nstep\ P\ \sigma 12\ n \wedge combine\ coll12\ coll22 = coll1$
using *Suc Sucnstep*(*1*) **by** *fastforce*
then show $\exists \sigma 1\ coll1\ coll2.\ (\sigma 1,\ coll1) \in csmall\ P\ \sigma$
$\wedge\ (\sigma',\ coll2) \in csmall\text{-}nstep\ P\ \sigma 1\ (Suc\ n) \wedge combine\ coll1\ coll2 = coll'$
using *combine-assoc*[*of coll12 coll22 coll2*] *Sucnstep* **by** *fastforce*
qed
have *left*: $\bigwedge\sigma'\ coll'.\ \exists \sigma 1\ coll1\ coll2.\ (\sigma 1,\ coll1) \in csmall\ P\ \sigma$
$\wedge\ (\sigma',\ coll2) \in csmall\text{-}nstep\ P\ \sigma 1\ (Suc\ n) \wedge combine\ coll1\ coll2 = coll'$
$\Longrightarrow (\sigma',\ coll') \in csmall\text{-}nstep\ P\ \sigma\ (Suc(Suc\ n))$
proof −
fix $\sigma'\ coll'$
assume $\exists \sigma 1\ coll1\ coll2.\ (\sigma 1,\ coll1) \in csmall\ P\ \sigma$
$\wedge\ (\sigma',\ coll2) \in csmall\text{-}nstep\ P\ \sigma 1\ (Suc\ n) \wedge combine\ coll1\ coll2 = coll'$
then obtain $\sigma 1\ coll1\ coll2$ **where** *Sucnstep*: $(\sigma 1,\ coll1) \in csmall\ P\ \sigma$
$(\sigma',\ coll2) \in csmall\text{-}nstep\ P\ \sigma 1\ (Suc\ n)\ combine\ coll1\ coll2 = coll'$
by *fastforce*
obtain $\sigma 12\ coll12\ coll22$ **where** *nstep*: $(\sigma 12,\ coll12) \in csmall\text{-}nstep\ P\ \sigma 1\ n$
$\wedge\ (\sigma',\ coll22) \in csmall\ P\ \sigma 12 \wedge combine\ coll12\ coll22 = coll2$
using *Sucnstep*(*2*) **by** *auto*
then show $(\sigma',\ coll') \in csmall\text{-}nstep\ P\ \sigma\ (Suc(Suc\ n))$
using *combine-assoc*[*of coll1 coll12 coll22*] *Suc Sucnstep* **by** *fastforce*
qed
show *?case* **using** *right left* **by** *fast*
qed(*simp*)

lemma *csmall-nstep-SucD*:
assumes $(\sigma',coll') \in csmall\text{-}nstep\ P\ \sigma\ (Suc\ n)$
shows $\exists \sigma 1\ coll1.\ (\sigma 1,\ coll1) \in csmall\ P\ \sigma$
$\wedge\ (\exists coll.\ coll' = combine\ coll1\ coll \wedge (\sigma',coll) \in csmall\text{-}nstep\ P\ \sigma 1\ n)$
using *csmall-nstep-Rec2 CollectionSemantics-axioms case-prodD assms* **by** *fastforce*

lemma *csmall-nstep-Suc-nend*: $o' \in csmall\text{-}nstep\ P\ \sigma\ (Suc\ n1) \Longrightarrow \sigma \notin endset$
using *endset-final csmall-nstep-SucD CollectionSemantics.csmall-def CollectionSemantics-axioms*

by *fastforce*

lemma *small-to-csmall-nstep-pres*:
assumes *Qpres*: $\bigwedge P\ \sigma\ \sigma'.\ Q\ P\ \sigma \Longrightarrow \sigma' \in small\ P\ \sigma \Longrightarrow Q\ P\ \sigma'$
shows $Q\ P\ \sigma \Longrightarrow (\sigma', coll) \in csmall\text{-}nstep\ P\ \sigma\ n \Longrightarrow Q\ P\ \sigma'$
proof(*induct n arbitrary*: $\sigma\ \sigma'$ *coll*)
  case (*Suc n*)
  then obtain $\sigma 1$ *coll1 coll2* **where** *nstep*: $(\sigma 1, coll1) \in csmall\text{-}nstep\ P\ \sigma\ n$
    $\wedge (\sigma', coll2) \in csmall\ P\ \sigma 1 \wedge combine\ coll1\ coll2 = coll$ **by** *clarsimp*
  then show *?case* **using** *Suc Qpres*[**where** $P=P$ **and** $\sigma=\sigma 1$ **and** $\sigma'=\sigma'$] **by**(*auto simp*: *csmall-def*)
qed(*simp*)

lemma *csmall-to-csmall-nstep-prop*:
assumes *cond*: $\bigwedge P\ \sigma\ \sigma'\ coll.\ (\sigma', coll) \in csmall\ P\ \sigma \Longrightarrow R\ P\ coll \Longrightarrow Q\ P\ \sigma \Longrightarrow R'\ P\ \sigma\ \sigma'\ coll$
  and *Rcomb*: $\bigwedge P\ coll1\ coll2.\ R\ P\ (combine\ coll1\ coll2) = (R\ P\ coll1 \wedge R\ P\ coll2)$
  and *Qpres*: $\bigwedge P\ \sigma\ \sigma'.\ Q\ P\ \sigma \Longrightarrow \sigma' \in small\ P\ \sigma \Longrightarrow Q\ P\ \sigma'$
  and *Rtrans'*: $\bigwedge P\ \sigma\ \sigma 1\ \sigma'\ coll1\ coll2.$
    $R'\ P\ \sigma\ \sigma 1\ coll1 \wedge R'\ P\ \sigma 1\ \sigma'\ coll2 \Longrightarrow R'\ P\ \sigma\ \sigma'\ (combine\ coll1\ coll2)$
  and *base*: $\bigwedge \sigma.\ R'\ P\ \sigma\ \sigma\ collect\text{-}id$
shows $(\sigma', coll) \in csmall\text{-}nstep\ P\ \sigma\ n \Longrightarrow R\ P\ coll \Longrightarrow Q\ P\ \sigma \Longrightarrow R'\ P\ \sigma\ \sigma'\ coll$
proof(*induct n arbitrary*: $\sigma\ \sigma'$ *coll*)
  case (*Suc n*)
  then obtain $\sigma 1$ *coll1 coll2* **where** *nstep*: $(\sigma 1, coll1) \in csmall\text{-}nstep\ P\ \sigma\ n$
    $\wedge (\sigma', coll2) \in csmall\ P\ \sigma 1 \wedge combine\ coll1\ coll2 = coll$ **by** *clarsimp*
  then have $Q\ P\ \sigma 1$ **using** *small-to-csmall-nstep-pres*[**where** $Q=Q$] *Qpres Suc* **by** *blast*
  then show *?case* **using** *nstep assms Suc* **by** *auto blast*
qed(*simp add*: *base*)

lemma *csmall-to-csmall-nstep-prop2*:
assumes *cond*: $\bigwedge P\ P'\ \sigma\ \sigma'\ coll.\ (\sigma', coll) \in csmall\ P\ \sigma$
    $\Longrightarrow R\ P\ P'\ coll \Longrightarrow Q\ \sigma \Longrightarrow (\sigma', coll) \in csmall\ P'\ \sigma$
  and *Rcomb*: $\bigwedge P\ P'\ coll1\ coll2.\ R\ P\ P'\ (combine\ coll1\ coll2) = (R\ P\ P'\ coll1 \wedge R\ P\ P'\ coll2)$
  and *Qpres*: $\bigwedge P\ \sigma\ \sigma'.\ Q\ \sigma \Longrightarrow \sigma' \in small\ P\ \sigma \Longrightarrow Q\ \sigma'$
shows $(\sigma', coll) \in csmall\text{-}nstep\ P\ \sigma\ n \Longrightarrow R\ P\ P'\ coll \Longrightarrow Q\ \sigma \Longrightarrow (\sigma', coll) \in csmall\text{-}nstep\ P'\ \sigma\ n$
proof(*induct n arbitrary*: $\sigma\ \sigma'$ *coll*)
  case (*Suc n*)
  then obtain $\sigma 1$ *coll1 coll2* **where** *nstep*: $(\sigma 1, coll1) \in csmall\text{-}nstep\ P\ \sigma\ n$
    $\wedge (\sigma', coll2) \in csmall\ P\ \sigma 1 \wedge combine\ coll1\ coll2 = coll$ **by** *clarsimp*
  then have $Q\ \sigma 1$ **using** *small-to-csmall-nstep-pres*[**where** $Q=\lambda P.\ Q$] *Qpres Suc* **by** *blast*

then show *?case* **using** *nstep assms Suc* **by** *auto blast*
qed(*simp*)

4.3 Extending *csmall* to a big-step semantics

definition *cbig* :: $'prog \Rightarrow 'state \Rightarrow ('state \times 'coll)\ set$ **where**
$cbig\ P\ \sigma \equiv$
$\{\ (\sigma', coll).\ \exists n.\ (\sigma', coll) \in csmall\text{-}nstep\ P\ \sigma\ n \wedge \sigma' \in endset\ \}$

lemma *cbigD*:
$[\![\ (\sigma',coll') \in cbig\ P\ \sigma\ ]\!] \Longrightarrow \exists n.\ (\sigma',coll') \in csmall\text{-}nstep\ P\ \sigma\ n \wedge \sigma' \in endset$
by(*simp add*: *cbig-def*)

lemma *cbigD'*:
$[\![\ o' \in cbig\ P\ \sigma\ ]\!] \Longrightarrow \exists n.\ o' \in csmall\text{-}nstep\ P\ \sigma\ n \wedge fst\ o' \in endset$
by(*cases o'*, *simp add*: *cbig-def*)

lemma *cbig-def2*:
$(\sigma', coll) \in cbig\ P\ \sigma \longleftrightarrow (\exists n.\ (\sigma', coll) \in csmall\text{-}nstep\ P\ \sigma\ n \wedge \sigma' \in endset)$
proof(*rule iffI*)
assume $(\sigma', coll) \in cbig\ P\ \sigma$
then show $\exists n.\ (\sigma', coll) \in csmall\text{-}nstep\ P\ \sigma\ n \wedge \sigma' \in endset$ **using** *bigD cbig-def*
by *auto*
next
assume $\exists n.\ (\sigma', coll) \in csmall\text{-}nstep\ P\ \sigma\ n \wedge \sigma' \in endset$
then show $(\sigma', coll) \in cbig\ P\ \sigma$ **using** *big-def cbig-def small-nstep-csmall-nstep-equiv*
by *auto*
qed

lemma *cbig-big-equiv*:
$(\exists coll.\ (\sigma', coll) \in cbig\ P\ \sigma) \longleftrightarrow \sigma' \in big\ P\ \sigma$
proof(*rule iffI*)
assume $\exists coll.\ (\sigma', coll) \in cbig\ P\ \sigma$
then show $\sigma' \in big\ P\ \sigma$ **by** (*auto simp*: *big-def cbig-def small-nstep-csmall-nstep-equiv*)
next
assume $\sigma' \in big\ P\ \sigma$
then show $\exists coll.\ (\sigma', coll) \in cbig\ P\ \sigma$ **by** (*fastforce simp*: *cbig-def dest*: *csmall-nstep-exists*)
qed

lemma *cbig-big-implies*:
$(\sigma', coll) \in cbig\ P\ \sigma \Longrightarrow \sigma' \in big\ P\ \sigma$
using *cbig-big-equiv* **by** *blast*

lemma *csmall-to-cbig-prop*:
assumes $\bigwedge P\ \sigma\ \sigma'\ coll.\ (\sigma', coll) \in csmall\ P\ \sigma \Longrightarrow R\ P\ coll \Longrightarrow Q\ P\ \sigma \Longrightarrow R'\ P\ \sigma\ \sigma'\ coll$
and $\bigwedge P\ coll1\ coll2.\ R\ P\ (combine\ coll1\ coll2) = (R\ P\ coll1 \wedge R\ P\ coll2)$
and $\bigwedge P\ \sigma\ \sigma'.\ Q\ P\ \sigma \Longrightarrow \sigma' \in small\ P\ \sigma \Longrightarrow Q\ P\ \sigma'$

and $\bigwedge P\ \sigma\ \sigma 1\ \sigma'\ coll1\ coll2.$
$R'\ P\ \sigma\ \sigma 1\ coll1 \wedge R'\ P\ \sigma 1\ \sigma'\ coll2 \Longrightarrow R'\ P\ \sigma\ \sigma'\ (combine\ coll1\ coll2)$
and $\bigwedge\sigma.\ R'\ P\ \sigma\ \sigma\ collect\text{-}id$
shows $(\sigma',\ coll) \in cbig\ P\ \sigma \Longrightarrow R\ P\ coll \Longrightarrow Q\ P\ \sigma \Longrightarrow R'\ P\ \sigma\ \sigma'\ coll$
using *assms csmall-to-csmall-nstep-prop*[**where** $R{=}R$ **and** $Q{=}Q$ **and** $R'{=}R'$ **and** $\sigma{=}\sigma$]
by(*auto simp*: *cbig-def2*)

lemma *csmall-to-cbig-prop2*:
assumes $\bigwedge P\ P'\ \sigma\ \sigma'\ coll.\ (\sigma',\ coll) \in csmall\ P\ \sigma \Longrightarrow R\ P\ P'\ coll \Longrightarrow Q\ \sigma \Longrightarrow (\sigma',\ coll) \in csmall\ P'\ \sigma$
and $\bigwedge P\ P'\ coll1\ coll2.\ R\ P\ P'\ (combine\ coll1\ coll2) = (R\ P\ P'\ coll1 \wedge R\ P\ P'\ coll2)$
and *Qpres*: $\bigwedge P\ \sigma\ \sigma'.\ Q\ \sigma \Longrightarrow \sigma' \in small\ P\ \sigma \Longrightarrow Q\ \sigma'$
shows $(\sigma',\ coll) \in cbig\ P\ \sigma \Longrightarrow R\ P\ P'\ coll \Longrightarrow Q\ \sigma \Longrightarrow (\sigma',\ coll) \in cbig\ P'\ \sigma$
using *assms csmall-to-csmall-nstep-prop2*[**where** $R{=}R$ **and** $Q{=}Q$] **by**(*auto simp*: *cbig-def2*) *blast*

lemma *cbig-stepD*:
assumes *cbig*: $(\sigma',coll') \in cbig\ P\ \sigma$ **and** *nend*: $\sigma \notin endset$
shows $\exists \sigma 1\ coll1.\ (\sigma 1,\ coll1) \in csmall\ P\ \sigma$
$\wedge (\exists\ coll.\ coll' = combine\ coll1\ coll \wedge (\sigma',coll) \in cbig\ P\ \sigma 1)$
proof −
obtain n **where** n: $(\sigma',\ coll') \in csmall\text{-}nstep\ P\ \sigma\ n\ \sigma' \in endset$
using *cbig-def2 cbig* **by** *auto*
then show *?thesis* **using** *csmall-nstep-SucD nend cbig-def2* **by**(*cases n*, *simp*) *blast*
qed

lemma *csmall-nstep-det-last-eq*:
assumes *det*: $\forall \sigma.\ small\ P\ \sigma = \{\} \vee (\exists \sigma'.\ small\ P\ \sigma = \{\sigma'\})$
shows $[\![\ (\sigma',coll') \in cbig\ P\ \sigma;\ (\sigma',coll') \in csmall\text{-}nstep\ P\ \sigma\ n;\ (\sigma',coll'') \in csmall\text{-}nstep\ P\ \sigma\ n'\ ]\!]$
$\Longrightarrow n = n'$
proof(*induct n arbitrary*: $n'\ \sigma\ \sigma'\ coll'\ coll''$)
case *0*
have $\sigma' = \sigma$ **using** *0.prems*(*2*) *csmall-nstep-base* **by** *blast*
then have *endset*: $\sigma \in endset$ **using** *0.prems*(*1*) *cbigD* **by** *blast*
show *?case*
proof(*cases* n')
case *Suc* **then show** *?thesis* **using** *0.prems*(*3*) *csmall-nstep-Suc-nend endset* **by** *blast*
qed(*simp*)
next
case (*Suc n1*)
then have *endset*: $\sigma' \in endset$ **using** *Suc.prems*(*1*) *cbigD* **by** *blast*

have *nend*: $\sigma \notin$ *endset* **using** *csmall-nstep-Suc-nend*[*OF Suc.prems*(*2*)] **by** *simp*
then have *neq*: $\sigma' \neq \sigma$ **using** *endset* **by** *auto*
obtain *σ1 coll coll1* **where** *σ1*: (*σ1*,*coll1*) ∈ *csmall P σ coll′* = *combine coll1 coll*
(*σ′*,*coll*) ∈ *csmall-nstep P σ1 n1*
using *csmall-nstep-SucD*[*OF Suc.prems*(*2*)] **by** *clarsimp*
then have *cbig*: (*σ′*,*coll*) ∈ *cbig P σ1* **using** *endset* **by**(*auto simp*: *cbig-def*)
show *?case*
proof(*cases n′*)
case *0* **then show** *?thesis* **using** *neq Suc.prems*(*3*) **using** *csmall-nstep-base* **by** *simp*
next
case *Suc′*: (*Suc n1′*)
then obtain *σ1′ coll2 coll1′* **where** *σ1′*: (*σ1′*,*coll1′*) ∈ *csmall P σ coll″* = *combine coll1′ coll2*
(*σ′*,*coll2*) ∈ *csmall-nstep P σ1′ n1′*
using *csmall-nstep-SucD*[**where** *σ*=*σ* **and** *σ′*=*σ′* **and** *coll′*=*coll″* **and** *n*=*n1′*] *Suc.prems*(*3*) **by** *blast*
then have *σ1*=*σ1′* **using** *σ1 det csmall-def* **by** *auto*
then show *?thesis* **using** *Suc.hyps*(*1*)[*OF cbig σ1*(*3*)] *σ1′*(*3*) *Suc′* **by** *blast*
qed
qed

end — CollectionSemantics

end

5 Collection-based RTS

theory *CollectionBasedRTS*
imports *RTS-safe CollectionSemantics*
begin

locale *CollectionBasedRTS-base* = *RTS-safe* + *CollectionSemantics*

General model for Regression Test Selection based on *CollectionSemantics*:

locale *CollectionBasedRTS* = *CollectionBasedRTS-base* **where**
small = *small* :: *′prog* ⇒ *′state* ⇒ *′state set* **and**
collect = *collect* :: *′prog* ⇒ *′state* ⇒ *′state* ⇒ *′coll* **and**
out = *out* :: *′prog* ⇒ *′test* ⇒ (*′state* × *′coll*) *set*
for *small collect out*
\+
fixes
make-test-prog :: *′prog* ⇒ *′test* ⇒ *′prog* **and**
collect-start :: *′prog* ⇒ *′coll*
assumes
out-cbig:
∃ *i*. *out P t* = {(*σ′*,*coll′*). ∃ *coll*. (*σ′*,*coll*) ∈ *cbig* (*make-test-prog P t*) *i*
∧ *coll′* = *combine coll* (*collect-start P*) }

context *CollectionBasedRTS* **begin**

end — CollectionBasedRTS

end

6 Instantiating *Semantics* with Jinja JVM

theory *JVMSemantics*
imports ../*Common*/*Semantics* *JinjaDCI*.*JVMExec*
begin

fun *JVMsmall* :: *jvm-prog* ⇒ *jvm-state* ⇒ *jvm-state set* **where**
JVMsmall P σ = { σ'. *exec* (*P*, σ) = *Some* σ' }

lemma *JVMsmall-prealloc-pres*:
assumes *pre*: *preallocated* (*fst*(*snd* σ))
  and $\sigma' \in$ *JVMsmall P* σ
shows *preallocated* (*fst*(*snd* σ'))
using *exec-prealloc-pres*[*OF pre*] *assms* **by**(*cases* σ, *cases* σ', *auto*)

lemma *JVMsmall-det*: *JVMsmall P* σ = {} ∨ (∃ σ'. *JVMsmall P* σ = {σ'})
by *auto*

definition *JVMendset* :: *jvm-state set* **where**
JVMendset ≡ { (*xp*,*h*,*frs*,*sh*). *frs* = [] ∨ (∃ *x*. *xp* = *Some x*) }

lemma *JVMendset-final*: $\sigma \in$ *JVMendset* ⟹ ∀ *P*. *JVMsmall P* σ = {}
 by(*auto simp*: *JVMendset-def*)

lemma *start-state-nend*:
 start-state P ∉ *JVMendset*
 by(*simp add*: *start-state-def JVMendset-def*)

interpretation *JVMSemantics*: *Semantics JVMsmall JVMendset*
 by *unfold-locales* (*auto dest*: *JVMendset-final*)

end

7 *classes-changed* theory

theory *ClassesChanged*
imports *JinjaDCI*.*Decl*
begin

A class is considered changed if it exists only in one program or the other, or exists in both but is different.

definition *classes-changed* :: *'m prog* ⇒ *'m prog* ⇒ *cname set* **where**
classes-changed P1 P2 = {*cn. class P1 cn* ≠ *class P2 cn*}

definition *class-changed* :: *'m prog* ⇒ *'m prog* ⇒ *cname* ⇒ *bool* **where**
class-changed P1 P2 cn = (*class P1 cn* ≠ *class P2 cn*)

lemma *classes-changed-class-changed*[*simp*]: *cn* ∈ *classes-changed P1 P2* = *class-changed P1 P2 cn*
by (*simp add*: *classes-changed-def class-changed-def*)

lemma *classes-changed-self*[*simp*]: *classes-changed P P* = {}
by (*auto simp*: *class-changed-def*)

lemma *classes-changed-sym*: *classes-changed P P'* = *classes-changed P' P*
by (*auto simp*: *class-changed-def*)

lemma *classes-changed-class*: ⟦ *cn* ∉ *classes-changed P P'*⟧ ⟹ *class P cn* = *class P' cn*
by (*clarsimp simp*: *class-changed-def*)

lemma *classes-changed-class-set*: ⟦ *S* ∩ *classes-changed P P'* = {} ⟧
⟹ ∀ *C* ∈ *S*. *class P C* = *class P' C*
by (*fastforce simp*: *disjoint-iff-not-equal dest*: *classes-changed-class*)

We now relate *classes-changed* over two programs to those over programs with an added class (such as a test class).

lemma *classes-changed-cons-eq*:
classes-changed (*t* # *P*) *P'* = (*classes-changed P P'* − {*fst t*})
∪ (*if class-changed* [*t*] *P'* (*fst t*) *then* {*fst t*} *else* {})
by (*auto simp*: *classes-changed-def class-changed-def class-def*)

lemma *class-changed-cons*:
fst t ∉ *classes-changed* (*t*#*P*) (*t*#*P'*)
by (*simp add*: *class-changed-def class-def*)

lemma *classes-changed-cons*:
classes-changed (*t* # *P*) (*t* # *P'*) = *classes-changed P P'* − {*fst t*}
proof(*cases fst t* ∈ *classes-changed P P'*)
case *True*
then show *?thesis* **using** *class-changed-cons*[**where** *t*=*t* **and** *P*=*P* **and** *P'*=*P'*]
classes-changed-cons-eq[**where** *t*=*t*] **by** (*auto simp*: *class-changed-def class-cons*)
next
case *False*
then show *?thesis* **using** *class-changed-cons*[**where** *t*=*t* **and** *P*=*P* **and** *P'*=*P'*]
by (*auto simp*: *class-changed-def*) (*metis* (*no-types*, *lifting*) *class-cons*)+
qed

lemma *classes-changed-int-Cons*:
assumes *coll* ∩ *classes-changed P P'* = {}

shows *coll* ∩ *classes-changed* (*t* # *P*) (*t* # *P'*) = {}
proof(*cases fst t* ∈ *classes-changed P P'*)
  case *True*
  then have *classes-changed P P'* = *classes-changed* (*t* # *P*) (*t* # *P'*) ∪ {*fst t*}
    using *classes-changed-cons*[**where** *t*=*t* **and** *P*=*P* **and** *P'*=*P'*] **by** *fastforce*
  then show *?thesis* **using** *assms* **by** *simp*
next
  case *False*
  then have *classes-changed P P'* = *classes-changed* (*t* # *P*) (*t* # *P'*)
    using *classes-changed-cons*[**where** *t*=*t* **and** *P*=*P* **and** *P'*=*P'*] **by** *fastforce*
  then show *?thesis* **using** *assms* **by** *simp*
qed

end

8 *subcls* theory

theory *Subcls*
imports *JinjaDCI.TypeRel*
begin

lemma *subcls-class-ex*: ⟦ $P \vdash C \preceq^* C'$; $C \neq C'$ ⟧
⟹ ∃ *D' fs ms. class P C* = ⌊(*D'*, *fs*, *ms*)⌋
proof(*induct rule*: *converse-rtrancl-induct*)
  case (*step y z*) **then show** *?case* **by**(*auto dest*: *subcls1D*)
qed(*simp*)

lemma *class-subcls1*:
⟦ *class P y* = *class P' y*; $P \vdash y \prec^1 z$ ⟧ ⟹ $P' \vdash y \prec^1 z$
 by(*auto dest*!: *subcls1D intro*!: *subcls1I intro*: *sym*)

lemma *subcls1-single-valued*: *single-valued* (*subcls1 P*)
by (*clarsimp simp*: *single-valued-def subcls1.simps*)

lemma *subcls-confluent*:
  ⟦ $P \vdash C \preceq^* C'$; $P \vdash C \preceq^* C''$ ⟧ ⟹ $P \vdash C' \preceq^* C''$ ∨ $P \vdash C'' \preceq^* C'$
 by (*simp add*: *single-valued-confluent subcls1-single-valued*)

lemma *subcls1-confluent*: ⟦ $P \vdash a \prec^1 b$; $P \vdash a \preceq^* c$; $a \neq c$ ⟧ ⟹ $P \vdash b \preceq^* c$
using *subcls1-single-valued*
 by (*auto elim*!: *converse-rtranclE*[**where** *x*=*a*] *simp*: *single-valued-def*)

lemma *subcls-self-superclass*: ⟦ $P \vdash C \prec^1 C$; $P \vdash C \preceq^* D$ ⟧ ⟹ *D* = *C*
using *subcls1-single-valued*
 by (*auto elim*!: *rtrancl-induct*[**where** *b*=*D*] *simp*: *single-valued-def*)

lemma *subcls-of-Obj-acyclic*:

$[\![P \vdash C \preceq^* Object; C \neq D]\!] \Longrightarrow \neg(P \vdash C \preceq^* D \wedge P \vdash D \preceq^* C)$
proof(*induct arbitrary*: *D rule*: *converse-rtrancl-induct*)
  case *base* **then show** *?case* **by** *simp*
next
  case (*step y z*) **show** *?case*
  proof(*cases y=z*)
    case *True* **with** *step* **show** *?thesis* **by** *simp*
  next
    case *False* **show** *?thesis*
    proof(*cases z = D*)
      case *True* **with** *False step.hyps*(*3*)[*of y*] **show** *?thesis* **by** *clarsimp*
    next
      case *neq*: *False*
      with *step.hyps*(*3*) **have** $\neg(P \vdash z \preceq^* D \wedge P \vdash D \preceq^* z)$ **by** *simp*
      moreover from *step.hyps*(*1*)
      have $P \vdash D \preceq^* y \Longrightarrow P \vdash D \preceq^* z$ **by**(*simp add*: *rtrancl-into-rtrancl*)
      moreover from *step.hyps*(*1*) *step.prems*(*1*)
      have $P \vdash y \preceq^* D \Longrightarrow P \vdash z \preceq^* D$ **by**(*simp add*: *subcls1-confluent*)
      ultimately show *?thesis* **by** *clarsimp*
    qed
  qed
qed

lemma *subcls-of-Obj*: $[\![P \vdash C \preceq^* Object; P \vdash C \preceq^* D]\!] \Longrightarrow P \vdash D \preceq^* Object$
 by(*auto dest*: *subcls-confluent*)

end

9 *classes-above* theory

This section contains theory around the classes above (superclasses of) a class in the class structure, in particular noting that if their contents have not changed, then much of what that class sees (methods, fields) stays the same.

theory *ClassesAbove*
imports *ClassesChanged Subcls JinjaDCI.Exceptions*
begin

abbreviation *classes-above* :: *'m prog* ⇒ *cname* ⇒ *cname set* **where**
classes-above P c ≡ $\{ cn.\ P \vdash c \preceq^* cn \}$

abbreviation *classes-between* :: *'m prog* ⇒ *cname* ⇒ *cname* ⇒ *cname set* **where**
classes-between P c d ≡ $\{ cn.\ (P \vdash c \preceq^* cn \wedge P \vdash cn \preceq^* d) \}$

abbreviation *classes-above-xcpts* :: *'m prog* ⇒ *cname set* **where**
classes-above-xcpts P ≡ $\bigcup x{\in}sys\text{-}xcpts.$ *classes-above P x*

lemma *classes-above-def2*:
$P \vdash C \prec^1 D \Longrightarrow$ *classes-above* P $C = \{C\} \cup$ *classes-above* P D
using *subcls1-confluent* **by** *auto*

lemma *classes-above-class*:
$[\![$ *classes-above* P $C \cap$ *classes-changed* P $P' = \{\}$; $P \vdash C \preceq^* C'$ $]\!]$
$\Longrightarrow$ *class* P C' = *class* P' C'
by (*drule classes-changed-class-set*, *simp*)

lemma *classes-above-subset*:
assumes *classes-above* P $C \cap$ *classes-changed* P $P' = \{\}$
shows *classes-above* P $C \subseteq$ *classes-above* P' C
proof –
have *ind*: $\bigwedge x.$ $P \vdash C \preceq^* x \Longrightarrow P' \vdash C \preceq^* x$
proof –
fix x **assume** *sub*: $P \vdash C \preceq^* x$
then show $P' \vdash C \preceq^* x$
proof(*induct rule*: *rtrancl-induct*)
case *base* **then show** *?case* **by** *simp*
next
case (*step y z*)
have $P' \vdash y \prec^1 z$ **by**(*rule class-subcls1*[*OF classes-above-class*[*OF assms step*(*1*)] *step*(*2*)])
then show *?case* **using** *step*(*3*) **by** *simp*
qed
qed
with *classes-changed-class-set*[*OF assms*] **show** *?thesis* **by** *clarsimp*
qed

lemma *classes-above-subcls*:
$[\![$ *classes-above* P $C \cap$ *classes-changed* P $P' = \{\}$; $P \vdash C \preceq^* C'$ $]\!]$
$\Longrightarrow P' \vdash C \preceq^* C'$
by (*fastforce dest*: *classes-above-subset*)

lemma *classes-above-subset2*:
assumes *classes-above* P $C \cap$ *classes-changed* P $P' = \{\}$
shows *classes-above* P' $C \subseteq$ *classes-above* P C
proof –
have *ind*: $\bigwedge x.$ $P' \vdash C \preceq^* x \Longrightarrow P \vdash C \preceq^* x$
proof –
fix x **assume** *sub*: $P' \vdash C \preceq^* x$
then show $P \vdash C \preceq^* x$
proof(*induct rule*: *rtrancl-induct*)
case *base* **then show** *?case* **by** *simp*
next
case (*step y z*)
with *class-subcls1 classes-above-class*[*OF assms*] *rtrancl-into-rtrancl* **show** *?case* **by** *metis*

qed
qed
with *classes-changed-class-set*[*OF assms*] **show** *?thesis* **by** *clarsimp*
qed

lemma *classes-above-subcls2*:
$[\![$ *classes-above* P C $\cap$ *classes-changed* P $P' = \{\}$; $P' \vdash C \preceq^* C'$ $]\!]$
$\Longrightarrow P \vdash C \preceq^* C'$
by (*fastforce dest*: *classes-above-subset2*)

lemma *classes-above-set*:
$[\![$ *classes-above* P C $\cap$ *classes-changed* P $P' = \{\}$ $]\!]$
$\Longrightarrow$ *classes-above* P C = *classes-above* P' C
by(*fastforce dest*: *classes-above-subset classes-above-subset2*)

lemma *classes-above-classes-changed-sym*:
assumes *classes-above* P C $\cap$ *classes-changed* P $P' = \{\}$
shows *classes-above* P' C $\cap$ *classes-changed* P' $P = \{\}$
proof −
have *classes-above* P C = *classes-above* P' C **by**(*rule classes-above-set*[*OF assms*])
with *classes-changed-sym*[**where** $P{=}P$] *assms* **show** *?thesis* **by** *simp*
qed

lemma *classes-above-sub-classes-between-eq*:
$P \vdash C \preceq^* D \Longrightarrow$ *classes-above* P C = (*classes-between* P C D − $\{D\}$) $\cup$ *classes-above* P D
using *subcls-confluent* **by** *auto*

lemma *classes-above-subcls-subset*:
$[\![$ $P \vdash C \preceq^* C'$ $]\!] \Longrightarrow$ *classes-above* P C' $\subseteq$ *classes-above* P C
by *auto*

9.1 Methods

lemma *classes-above-sees-methods*:
assumes *int*: *classes-above* P C $\cap$ *classes-changed* P $P' = \{\}$ **and** *ms*: $P \vdash C$ *sees-methods* Mm
shows $P' \vdash C$ *sees-methods* Mm
proof −
have *cls*: $\forall C' \in$ *classes-above* P C. *class* P C' = *class* P' C'
by(*rule classes-changed-class-set*[*OF int*])

have $\bigwedge C\ Mm.$ $P \vdash C$ *sees-methods* $Mm \Longrightarrow$
$\forall C' \in$ *classes-above* P C. *class* P C' = *class* P' C' $\Longrightarrow P' \vdash C$ *sees-methods* Mm
proof −
fix C Mm **assume** $P \vdash C$ *sees-methods* Mm **and** $\forall C' \in$ *classes-above* P C. *class* P C' = *class* P' C'
then show $P' \vdash C$ *sees-methods* Mm

proof(*induct rule*: *Methods.induct*)
case *Obj*: (*sees-methods-Object D fs ms Mm*)
with *cls* **have** *class P′ Object* = ⌊(*D*, *fs*, *ms*)⌋ **by** *simp*
with *Obj* **show** *?case* **by**(*auto intro*!: *sees-methods-Object*)
next
case *rec*: (*sees-methods-rec C D fs ms Mm Mm′*)
then have $P \vdash C \preceq^* D$ **by** (*simp add*: *r-into-rtrancl*[*OF subcls1I*])
with *converse-rtrancl-into-rtrancl* **have** $\bigwedge x.\ P \vdash D \preceq^* x \Longrightarrow P \vdash C \preceq^* x$ **by** *simp*
with *rec.prems*(*1*) **have** ∀ *C′*∈*classes-above P D*. *class P C′* = *class P′ C′* **by** *simp*
with *rec* **show** *?case* **by**(*auto intro*: *sees-methods-rec*)
qed
qed
with *ms cls* **show** *?thesis* **by** *simp*
qed

lemma *classes-above-sees-method*:
⟦ *classes-above P C* ∩ *classes-changed P P′* = {};
P ⊢ *C sees M*,*b*: *Ts*→*T* = *m in C′* ⟧
⟹ *P′* ⊢ *C sees M*,*b*: *Ts*→*T* = *m in C′*
by (*auto dest*: *classes-above-sees-methods simp*: *Method-def*)

lemma *classes-above-sees-method2*:
⟦ *classes-above P C* ∩ *classes-changed P P′* = {};
P′ ⊢ *C sees M*,*b*: *Ts*→*T* = *m in C′* ⟧
⟹ *P* ⊢ *C sees M*,*b*: *Ts*→*T* = *m in C′*
by (*auto dest*: *classes-above-classes-changed-sym intro*: *classes-above-sees-method*)

lemma *classes-above-method*:
assumes *classes-above P C* ∩ *classes-changed P P′* = {}
shows *method P C M* = *method P′ C M*
proof(*cases* ∃ *Ts T m D b*. *P* ⊢ *C sees M*,*b* : *Ts*→*T* = *m in D*)
case *True*
with *assms* **show** *?thesis* **by** (*auto dest*: *classes-above-sees-method*)
next
case *False*
with *assms* **have** ¬(∃ *Ts T m D b*. *P′* ⊢ *C sees M*,*b* : *Ts*→*T* = *m in D*)
by (*auto dest*: *classes-above-sees-method2*)
with *False* **show** *?thesis* **by**(*simp add*: *method-def*)
qed

9.2 Fields

lemma *classes-above-has-fields*:
assumes *int*: *classes-above P C* ∩ *classes-changed P P′* = {} **and** *fs*: *P* ⊢ *C has-fields FDTs*
shows *P′* ⊢ *C has-fields FDTs*
proof −

have *cls*: $\forall$ $C'$$\in$*classes-above P C*. *class P* C' = *class* P' C'
by(*rule classes-changed-class-set*[*OF int*])

have $\bigwedge$*C Mm*. $P \vdash C$ *has-fields FDTs* $\Longrightarrow$
$\forall$ $C'$$\in$*classes-above P C*. *class P* C' = *class* P' C' $\Longrightarrow$ $P' \vdash C$ *has-fields FDTs*
proof –
fix *C Mm* **assume** $P \vdash C$ *has-fields FDTs* **and** $\forall$ $C'$$\in$*classes-above P C*. *class P* C' = *class* P' C'
then show $P' \vdash C$ *has-fields FDTs*
proof(*induct rule*: *Fields.induct*)
case *Obj*: (*has-fields-Object D fs ms FDTs*)
with *cls* **have** *class* P' *Object* = $\lfloor$(*D*, *fs*, *ms*)$\rfloor$ **by** *simp*
with *Obj* **show** *?case* **by**(*auto intro*!: *has-fields-Object*)
next
case *rec*: (*has-fields-rec C D fs ms FDTs FDTs'*)
then have $P \vdash C \preceq^* D$ **by** (*simp add*: *r-into-rtrancl*[*OF subcls1I*])
with *converse-rtrancl-into-rtrancl* **have** $\bigwedge$*x*. $P \vdash D \preceq^* x \Longrightarrow P \vdash C \preceq^* x$ **by** *simp*
with *rec.prems*(*1*) **have** $\forall$ *x*. $P \vdash D \preceq^* x \longrightarrow$ *class P x* = *class* P' *x* **by** *simp*
with *rec* **show** *?case* **by**(*auto intro*: *has-fields-rec*)
qed
qed
with *fs cls* **show** *?thesis* **by** *simp*
qed

lemma *classes-above-has-fields-dne*:
assumes *classes-above P C* $\cap$ *classes-changed P* P' = {}
shows ($\forall$ *FDTs*. $\neg$ $P \vdash C$ *has-fields FDTs*) = ($\forall$ *FDTs*. $\neg$ $P' \vdash C$ *has-fields FDTs*)
proof(*rule iffI*)
assume *asm*: $\forall$ *FDTs*. $\neg$ $P \vdash C$ *has-fields FDTs*
from *assms classes-changed-sym*[**where** *P*=*P*] *classes-above-set*[*OF assms*]
have *int'*: *classes-above* P' *C* $\cap$ *classes-changed* P' *P* = {} **by** *simp*
from *asm classes-above-has-fields*[*OF int'*] **show** $\forall$ *FDTs*. $\neg$ $P' \vdash C$ *has-fields FDTs* **by** *auto*
next
assume $\forall$ *FDTs*. $\neg$ $P' \vdash C$ *has-fields FDTs*
with *assms* **show** $\forall$ *FDTs*. $\neg$ $P \vdash C$ *has-fields FDTs* **by**(*auto dest*: *classes-above-has-fields*)
qed

lemma *classes-above-has-field*:
$\llbracket$ *classes-above P C* $\cap$ *classes-changed P* P' = {};
$P \vdash C$ *has F*,*b*:*t in* C' $\rrbracket$
$\Longrightarrow$ $P' \vdash C$ *has F*,*b*:*t in* C'
by(*auto dest*: *classes-above-has-fields simp*: *has-field-def*)

lemma *classes-above-has-field2*:
$\llbracket$ *classes-above P C* $\cap$ *classes-changed P* P' = {};
$P' \vdash C$ *has F*,*b*:*t in* C' $\rrbracket$

$\Longrightarrow P \vdash C\ has\ F,b{:}t\ in\ C'$
by(*auto intro*: *classes-above-has-field dest*: *classes-above-classes-changed-sym*)

lemma *classes-above-sees-field*:
$[\![$ *classes-above* $P\ C \cap$ *classes-changed* $P\ P' = \{\}$;
$P \vdash C\ sees\ F,b{:}t\ in\ C'$ $]\!]$
$\Longrightarrow P' \vdash C\ sees\ F,b{:}t\ in\ C'$
by(*auto dest*: *classes-above-has-fields simp*: *sees-field-def*)

lemma *classes-above-sees-field2*:
$[\![$ *classes-above* $P\ C \cap$ *classes-changed* $P\ P' = \{\}$;
$P' \vdash C\ sees\ F,b{:}t\ in\ C'$ $]\!]$
$\Longrightarrow P \vdash C\ sees\ F,b{:}t\ in\ C'$
by (*auto intro*: *classes-above-sees-field dest*: *classes-above-classes-changed-sym*)

lemma *classes-above-field*:
assumes *classes-above* $P\ C \cap$ *classes-changed* $P\ P' = \{\}$
shows *field* $P\ C\ F =$ *field* $P'\ C\ F$
proof(*cases* $\exists\ T\ D\ b.\ P \vdash C\ sees\ F,b : T\ in\ D$)
case *True*
with *assms* **show** *?thesis* **by** (*auto dest*: *classes-above-sees-field*)
next
case *False*
with *assms* **have** $\neg(\exists\ T\ D\ b.\ P' \vdash C\ sees\ F,b : T\ in\ D)$
by (*auto dest*: *classes-above-sees-field2*)
with *False* **show** *?thesis* **by**(*simp add*: *field-def*)
qed

lemma *classes-above-fields*:
assumes *classes-above* $P\ C \cap$ *classes-changed* $P\ P' = \{\}$
shows *fields* $P\ C =$ *fields* $P'\ C$
proof(*cases* $\exists$ *FDTs*. $P \vdash C$ *has-fields FDTs*)
case *True*
with *assms* **show** *?thesis* **by**(*auto dest*: *classes-above-has-fields*)
next
case *False*
with *assms* **show** *?thesis* **by** (*auto dest*: *classes-above-has-fields-dne simp*: *fields-def*)
qed

lemma *classes-above-ifields*:
$[\![$ *classes-above* $P\ C \cap$ *classes-changed* $P\ P' = \{\}$ $]\!]$
$\Longrightarrow$
ifields $P\ C =$ *ifields* $P'\ C$
by (*simp add*: *ifields-def classes-above-fields*)

lemma *classes-above-blank*:
$[\![$ *classes-above* $P\ C \cap$ *classes-changed* $P\ P' = \{\}$ $]\!]$
$\Longrightarrow$

blank P C = *blank P′ C*
by (*simp add*: *blank-def classes-above-ifields*)

lemma *classes-above-isfields*:
⟦ *classes-above P C* ∩ *classes-changed P P′* = {} ⟧
⟹
isfields P C = *isfields P′ C*
by (*simp add*: *isfields-def classes-above-fields*)

lemma *classes-above-sblank*:
⟦ *classes-above P C* ∩ *classes-changed P P′* = {} ⟧
⟹
sblank P C = *sblank P′ C*
by (*simp add*: *sblank-def classes-above-isfields*)

9.3 Other

lemma *classes-above-start-heap*:
assumes *classes-above-xcpts P* ∩ *classes-changed P P′* = {}
shows *start-heap P* = *start-heap P′*
proof −
from *assms* **have** ∀ *C* ∈ *sys-xcpts*. *blank P C* = *blank P′ C* **by** (*auto intro*: *classes-above-blank*)
then show *?thesis* **by**(*simp add*: *start-heap-def*)
qed

end

10 Instantiating *CollectionSemantics* with Jinja JVM

theory *JVMCollectionSemantics*
imports ../*Common*/*CollectionSemantics JVMSemantics* ../*JinjaSuppl*/*ClassesAbove*

begin

abbreviation *JVMcombine* :: *cname set* ⇒ *cname set* ⇒ *cname set* **where**
JVMcombine C C′ ≡ *C* ∪ *C′*

abbreviation *JVMcollect-id* :: *cname set* **where**
JVMcollect-id ≡ {}

10.1 JVM-specific *classes-above* theory

fun *classes-above-frames* :: ′*m prog* ⇒ *frame list* ⇒ *cname set* **where**
classes-above-frames P ((*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*) = *classes-above P C* ∪ *classes-above-frames P frs* |
classes-above-frames P [] = {}

lemma *classes-above-start-state*:

assumes *above-xcpts*: *classes-above-xcpts P* ∩ *classes-changed P P′* = {}
shows *start-state P* = *start-state P′*
using *assms classes-above-start-heap* **by**(*simp add*: *start-state-def*)

lemma *classes-above-matches-ex-entry*:
classes-above P C ∩ *classes-changed P P′* = {}
⟹ *matches-ex-entry P C pc xcp* = *matches-ex-entry P′ C pc xcp*
using *classes-above-subcls classes-above-subcls2*
by(*auto simp*: *matches-ex-entry-def*)

lemma *classes-above-match-ex-table*:
assumes *classes-above P C* ∩ *classes-changed P P′* = {}
shows *match-ex-table P C pc es* = *match-ex-table P′ C pc es*
using *classes-above-matches-ex-entry*[*OF assms*] **proof**(*induct es*) **qed**(*auto*)

lemma *classes-above-find-handler*:
assumes *classes-above P* (*cname-of h a*) ∩ *classes-changed P P′* = {}
shows *classes-above-frames P frs* ∩ *classes-changed P P′* = {}
⟹ *find-handler P a h frs sh* = *find-handler P′ a h frs sh*
proof(*induct frs*)
case (*Cons fr′ frs′*)
obtain *stk loc C M pc ics* **where** *fr′*: *fr′* = (*stk*,*loc*,*C*,*M*,*pc*,*ics*) **by**(*cases fr′*)
with *Cons* **have**
intC: *classes-above P C* ∩ *classes-changed P P′* = {}
and *int*: *classes-above-frames P frs′* ∩ *classes-changed P P′* = {} **by** *auto*
show *?case* **using** *Cons fr′ int classes-above-method*[*OF intC*]
classes-above-match-ex-table[*OF assms*(*1*)] **by**(*auto split*: *bool.splits*)
qed(*simp*)

lemma *find-handler-classes-above-frames*:
find-handler P a h frs sh = (*xp′*,*h′*,*frs′*,*sh′*)
⟹ *classes-above-frames P frs′* ⊆ *classes-above-frames P frs*
proof(*induct frs*)
case (*Cons f1 frs1*)
then obtain *stk loc C M pc ics* **where** *f1*: *f1* = (*stk*,*loc*,*C*,*M*,*pc*,*ics*) **by**(*cases f1*)
show *?case*
proof(*cases match-ex-table P* (*cname-of h a*) *pc* (*ex-table-of P C M*))
case *None* **then show** *?thesis* **using** *f1 None Cons*
by(*auto split*: *bool.splits list.splits init-call-status.splits*)
next
case (*Some a*) **then show** *?thesis* **using** *f1 Some Cons* **by** *auto*
qed
qed(*simp*)

lemma *find-handler-pieces*:
find-handler P a h frs sh = (*xp′*,*h′*,*frs′*,*sh′*)
⟹ *h* = *h′* ∧ *sh* = *sh′* ∧ *classes-above-frames P frs′* ⊆ *classes-above-frames P frs*
using *find-handler-classes-above-frames* **by**(*auto dest*: *find-handler-heap find-handler-sheap*)

10.2 Naive RTS algorithm

fun *JVMinstr-ncollect* ::
 [*jvm-prog*, *instr*, *heap*, *val list*] ⇒ *cname set* **where**
JVMinstr-ncollect P (*New C*) *h stk* = *classes-above P C* |
JVMinstr-ncollect P (*Getfield F C*) *h stk* =
  (*if* (*hd stk*) = *Null then* {}
   else classes-above P (*cname-of h* (*the-Addr* (*hd stk*)))) |
JVMinstr-ncollect P (*Getstatic C F D*) *h stk* = *classes-above P C* |
JVMinstr-ncollect P (*Putfield F C*) *h stk* =
  (*if* (*hd* (*tl stk*)) = *Null then* {}
   else classes-above P (*cname-of h* (*the-Addr* (*hd* (*tl stk*))))) |
JVMinstr-ncollect P (*Putstatic C F D*) *h stk* = *classes-above P C* |
JVMinstr-ncollect P (*Checkcast C*) *h stk* =
  (*if* (*hd stk*) = *Null then* {}
   else classes-above P (*cname-of h* (*the-Addr* (*hd stk*)))) |
JVMinstr-ncollect P (*Invoke M n*) *h stk* =
  (*if* (*stk* ! *n*) = *Null then* {}
   else classes-above P (*cname-of h* (*the-Addr* (*stk* ! *n*)))) |
JVMinstr-ncollect P (*Invokestatic C M n*) *h stk* = *classes-above P C* |
JVMinstr-ncollect P Throw h stk =
  (*if* (*hd stk*) = *Null then* {}
   else classes-above P (*cname-of h* (*the-Addr* (*hd stk*)))) |
JVMinstr-ncollect P - h stk = {}

fun *JVMstep-ncollect* ::
 [*jvm-prog*, *heap*, *val list*, *cname*, *mname*, *pc*, *init-call-status*] ⇒ *cname set* **where**
JVMstep-ncollect P h stk C M pc (*Calling C′ Cs*) = *classes-above P C′* |
JVMstep-ncollect P h stk C M pc (*Called* (*C′#Cs*))
 = *classes-above P C′* ∪ *classes-above P* (*fst*(*method P C′ clinit*)) |
JVMstep-ncollect P h stk C M pc (*Throwing Cs a*) = *classes-above P* (*cname-of h a*) |
JVMstep-ncollect P h stk C M pc ics = *JVMinstr-ncollect P* (*instrs-of P C M* ! *pc*) *h stk*

— naive collection function
fun *JVMexec-ncollect* :: *jvm-prog* ⇒ *jvm-state* ⇒ *cname set* **where**
JVMexec-ncollect P (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*) =
  (*JVMstep-ncollect P h stk C M pc ics*
    ∪ *classes-above P C* ∪ *classes-above-frames P frs* ∪ *classes-above-xcpts P*
)
| *JVMexec-ncollect P* - = {}

fun *JVMNaiveCollect* :: *jvm-prog* ⇒ *jvm-state* ⇒ *jvm-state* ⇒ *cname set* **where**
JVMNaiveCollect P σ σ′ = *JVMexec-ncollect P σ*

interpretation *JVMNaiveCollectionSemantics*:
 CollectionSemantics JVMsmall JVMendset JVMNaiveCollect JVMcombine JVM-

collect-id
by *unfold-locales auto*

10.3 Smarter RTS algorithm

fun *JVMinstr-scollect* ::
[*jvm-prog*, *instr*] ⇒ *cname set* **where**
JVMinstr-scollect P (*Getstatic C F D*)
= (*if* ¬(∃ *t*. *P* ⊢ *C has F*,*Static*:*t in D*) *then classes-above P C*
else classes-between P C D − {*D*}) |
JVMinstr-scollect P (*Putstatic C F D*)
= (*if* ¬(∃ *t*. *P* ⊢ *C has F*,*Static*:*t in D*) *then classes-above P C*
else classes-between P C D − {*D*}) |
JVMinstr-scollect P (*Invokestatic C M n*)
= (*if* ¬(∃ *Ts T m D*. *P* ⊢ *C sees M*,*Static*:*Ts* → *T* = *m in D*) *then classes-above P C*
else classes-between P C (*fst*(*method P C M*)) − {*fst*(*method P C M*)}) |
JVMinstr-scollect P - = {}

fun *JVMstep-scollect* ::
[*jvm-prog*, *instr*, *init-call-status*] ⇒ *cname set* **where**
JVMstep-scollect P i (*Calling C′ Cs*) = {*C′*} |
JVMstep-scollect P i (*Called* (*C′*#*Cs*)) = {} |
JVMstep-scollect P i (*Throwing Cs a*) = {} |
JVMstep-scollect P i ics = *JVMinstr-scollect P i*

— smarter collection function
fun *JVMexec-scollect* :: *jvm-prog* ⇒ *jvm-state* ⇒ *cname set* **where**
JVMexec-scollect P (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*) =
JVMstep-scollect P (*instrs-of P C M* ! *pc*) *ics*
| *JVMexec-scollect P* - = {}

fun *JVMSmartCollect* :: *jvm-prog* ⇒ *jvm-state* ⇒ *jvm-state* ⇒ *cname set* **where**
JVMSmartCollect P σ σ′ = *JVMexec-scollect P σ*

interpretation *JVMSmartCollectionSemantics*:
CollectionSemantics JVMsmall JVMendset JVMSmartCollect JVMcombine JVM-collect-id
by *unfold-locales*

10.4 A few lemmas using the instantiations

lemma *JVMnaive-csmallD*:
(*σ′*, *cset*) ∈ *JVMNaiveCollectionSemantics.csmall P σ*
⟹ *JVMexec-ncollect P σ* = *cset* ∧ *σ′* ∈ *JVMsmall P σ*
by(*simp add*: *JVMNaiveCollectionSemantics.csmall-def*)

lemma *JVMsmart-csmallD*:

$(\sigma', cset) \in JVMSmartCollectionSemantics.csmall\ P\ \sigma$
$\Longrightarrow JVMexec\text{-}scollect\ P\ \sigma = cset \wedge \sigma' \in JVMsmall\ P\ \sigma$
by(*simp add*: *JVMSmartCollectionSemantics.csmall-def*)

lemma *jvm-naive-to-smart-csmall-nstep-last-eq*:
$[\![(\sigma', cset_n) \in JVMNaiveCollectionSemantics.cbig\ P\ \sigma;$
$(\sigma', cset_n) \in JVMNaiveCollectionSemantics.csmall\text{-}nstep\ P\ \sigma\ n;$
$(\sigma', cset_s) \in JVMSmartCollectionSemantics.csmall\text{-}nstep\ P\ \sigma\ n']\!]$
$\Longrightarrow n = n'$
proof(*induct n arbitrary*: $n'\ \sigma\ \sigma'\ cset_n\ cset_s$)
case *0*
have $\sigma' = \sigma$ **using** *0.prems*(*2*) *JVMNaiveCollectionSemantics.csmall-nstep-base* **by** *blast*
then have *endset*: $\sigma \in JVMendset$ **using** *0.prems*(*1*) *JVMNaiveCollectionSemantics.cbigD* **by** *blast*
show *?case*
proof(*cases n'*)
case *Suc* **then show** *?thesis* **using** *0.prems*(*3*) *JVMSmartCollectionSemantics.csmall-nstep-Suc-nend*
endset **by** *blast*
qed(*simp*)
next
case (*Suc n1*)
then have *endset*: $\sigma' \in JVMendset$ **using** *Suc.prems*(*1*) *JVMNaiveCollectionSemantics.cbigD* **by** *blast*
have *nend*: $\sigma \notin JVMendset$
using *JVMNaiveCollectionSemantics.csmall-nstep-Suc-nend*[*OF Suc.prems*(*2*)] **by** *simp*
then have *neq*: $\sigma' \neq \sigma$ **using** *endset* **by** *auto*
obtain *σ1 cset cset1* **where** *σ1*: $(\sigma 1, cset1) \in JVMNaiveCollectionSemantics.csmall\ P\ \sigma$
$cset_n = cset1 \cup cset\ (\sigma', cset) \in JVMNaiveCollectionSemantics.csmall\text{-}nstep\ P\ \sigma 1\ n1$
using *JVMNaiveCollectionSemantics.csmall-nstep-SucD*[*OF Suc.prems*(*2*)] **by** *clarsimp*
then have *cbig*: $(\sigma', cset) \in JVMNaiveCollectionSemantics.cbig\ P\ \sigma 1$
using *endset* **by**(*auto simp*: *JVMNaiveCollectionSemantics.cbig-def*)
show *?case*
proof(*cases n'*)
case *0* **then show** *?thesis*
using *neq Suc.prems*(*3*) *JVMSmartCollectionSemantics.csmall-nstep-base* **by** *blast*
next
case *Suc'*: (*Suc n1'*)
then obtain *σ1' cset2 cset1'* **where** *σ1'*: $(\sigma 1', cset1') \in JVMSmartCollectionSemantics.csmall\ P\ \sigma$
$cset_s = cset1' \cup cset2\ (\sigma', cset2) \in JVMSmartCollectionSemantics.csmall\text{-}nstep\ P\ \sigma 1'\ n1'$

using *JVMSmartCollectionSemantics.csmall-nstep-SucD*[**where** σ=σ **and** σ'=σ' **and** *coll'*=$cset_s$
and *n*=*n1'*] *Suc.prems*(*3*) **by** *blast*
then have $\sigma 1$=$\sigma 1'$ **using** $\sigma 1$ *JVMNaiveCollectionSemantics.csmall-def* *JVMSmartCollectionSemantics.csmall-def* **by** *auto*
then show *?thesis* **using** *Suc.hyps*(*1*)[*OF cbig* $\sigma 1$(*3*)] $\sigma 1'$(*3*) *Suc'* **by** *blast*
qed
qed

end

11 Inductive JVM execution

theory *JVMExecStepInductive*
imports *JinjaDCI.JVMExec*
begin

datatype *step-input* = *StepI instr* |
StepC cname cname list | *StepC2 cname cname list* |
StepT cname list addr

inductive *exec-step-ind* :: [*step-input*, *jvm-prog*, *heap*, *val list*, *val list*, *cname*, *mname*, *pc*, *init-call-status*, *frame list*, *sheap*,*jvm-state*] ⇒ *bool*
where
exec-step-ind-Load:
exec-step-ind (*StepI* (*Load n*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
(*None*, *h*, ((*loc* ! *n*) # *stk*, *loc*, C_0, M_0, *Suc pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Store*:
exec-step-ind (*StepI* (*Store n*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
(*None*, *h*, (*tl stk*, *loc*[*n*:=*hd stk*], C_0, M_0, *Suc pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Push*:
exec-step-ind (*StepI* (*Push v*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
(*None*, *h*, (*v* # *stk*, *loc*, C_0, M_0, *Suc pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-NewOOM-Called*:
new-Addr h = *None*
⟹ *exec-step-ind* (*StepI* (*New C*)) *P h stk loc* C_0 M_0 *pc* (*Called Cs*) *frs sh*
(⌊*addr-of-sys-xcpt OutOfMemory*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *No-ics*)#*frs*, *sh*)

| *exec-step-ind-NewOOM-Done*:
⟦ *sh C* = *Some*(*obj*, *Done*); *new-Addr h* = *None*; ∀ *Cs*. *ics* ≠ *Called Cs* ⟧
⟹ *exec-step-ind* (*StepI* (*New C*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
(⌊*addr-of-sys-xcpt OutOfMemory*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-New-Called*:

new-Addr h = *Some a*
  $\Longrightarrow$ *exec-step-ind* (*StepI* (*New C*)) *P h stk loc* C_0 M_0 *pc* (*Called Cs*) *frs sh*
  (*None*, *h*(*a*↦*blank P C*), (*Addr a*#*stk*, *loc*, C_0, M_0, *Suc pc*, *No-ics*)#*frs*, *sh*)

| *exec-step-ind-New-Done*:
⟦ *sh C* = *Some*(*obj*, *Done*); *new-Addr h* = *Some a*; ∀ *Cs*. *ics* ≠ *Called Cs* ⟧
  $\Longrightarrow$ *exec-step-ind* (*StepI* (*New C*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
  (*None*, *h*(*a*↦*blank P C*), (*Addr a*#*stk*, *loc*, C_0, M_0, *Suc pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-New-Init*:
⟦ ∀ *obj*. *sh C* ≠ *Some*(*obj*, *Done*); ∀ *Cs*. *ics* ≠ *Called Cs* ⟧
  $\Longrightarrow$ *exec-step-ind* (*StepI* (*New C*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
  (*None*, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *Calling C* [])#*frs*, *sh*)

| *exec-step-ind-Getfield-Null*:
hd stk = *Null*
  $\Longrightarrow$ *exec-step-ind* (*StepI* (*Getfield F C*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
  (⌊*addr-of-sys-xcpt NullPointer*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Getfield-NoField*:
⟦ *v* = *hd stk*; (*D*,*fs*) = *the*(*h*(*the-Addr v*)); *v* ≠ *Null*; ¬(∃ *t b*. *P* ⊢ *D has F*,*b*:*t in C*) ⟧
  $\Longrightarrow$ *exec-step-ind* (*StepI* (*Getfield F C*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
  (⌊*addr-of-sys-xcpt NoSuchFieldError*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Getfield-Static*:
⟦ *v* = *hd stk*; (*D*,*fs*) = *the*(*h*(*the-Addr v*)); *v* ≠ *Null*; *P* ⊢ *D has F*,*Static*:*t in C* ⟧
  $\Longrightarrow$ *exec-step-ind* (*StepI* (*Getfield F C*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
  (⌊*addr-of-sys-xcpt IncompatibleClassChangeError*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Getfield*:
⟦ *v* = *hd stk*; (*D*,*fs*) = *the*(*h*(*the-Addr v*)); (*D′*,*b*,*t*) = *field P C F*; *v* ≠ *Null*;
  P ⊢ *D has F*,*NonStatic*:*t in C* ⟧
  $\Longrightarrow$ *exec-step-ind* (*StepI* (*Getfield F C*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
  (*None*, *h*, (*the*(*fs*(*F*,*C*))#(*tl stk*), *loc*, C_0, M_0, *pc*+*1*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Getstatic-NoField*:
¬(∃ *t b*. *P* ⊢ *C has F*,*b*:*t in D*)
  $\Longrightarrow$ *exec-step-ind* (*StepI* (*Getstatic C F D*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
  (⌊*addr-of-sys-xcpt NoSuchFieldError*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Getstatic-NonStatic*:
P ⊢ *C has F*,*NonStatic*:*t in D*
  $\Longrightarrow$ *exec-step-ind* (*StepI* (*Getstatic C F D*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
  (⌊*addr-of-sys-xcpt IncompatibleClassChangeError*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Getstatic-Called*:

$[\![\ (D',b,t) = \textit{field}\ P\ D\ F;\ P \vdash C\ \textit{has}\ F,\textit{Static}{:}t\ \textit{in}\ D;$
$v = \textit{the}\ ((\textit{fst}(\textit{the}(\textit{sh}\ D')))\ F)\ ]\!]$
$\Longrightarrow \textit{exec-step-ind}\ (\textit{StepI}\ (\textit{Getstatic}\ C\ F\ D))\ P\ h\ \textit{stk}\ \textit{loc}\ C_0\ M_0\ \textit{pc}\ (\textit{Called}\ \textit{Cs})\ \textit{frs}\ \textit{sh}$
$(\textit{None},\ h,\ (v\#\textit{stk},\ \textit{loc},\ C_0,\ M_0,\ \textit{Suc}\ \textit{pc},\ \textit{No-ics})\#\textit{frs},\ \textit{sh})$

| *exec-step-ind-Getstatic-Done*:
$[\![\ (D',b,t) = \textit{field}\ P\ D\ F;\ P \vdash C\ \textit{has}\ F,\textit{Static}{:}t\ \textit{in}\ D;$
$\forall\,\textit{Cs}.\ \textit{ics} \neq \textit{Called}\ \textit{Cs};\ \textit{sh}\ D' = \textit{Some}(\textit{sfs},\textit{Done});$
$v = \textit{the}\ (\textit{sfs}\ F)\ ]\!]$
$\Longrightarrow \textit{exec-step-ind}\ (\textit{StepI}\ (\textit{Getstatic}\ C\ F\ D))\ P\ h\ \textit{stk}\ \textit{loc}\ C_0\ M_0\ \textit{pc}\ \textit{ics}\ \textit{frs}\ \textit{sh}$
$(\textit{None},\ h,\ (v\#\textit{stk},\ \textit{loc},\ C_0,\ M_0,\ \textit{Suc}\ \textit{pc},\ \textit{ics})\#\textit{frs},\ \textit{sh})$

| *exec-step-ind-Getstatic-Init*:
$[\![\ (D',b,t) = \textit{field}\ P\ D\ F;\ P \vdash C\ \textit{has}\ F,\textit{Static}{:}t\ \textit{in}\ D;$
$\forall\,\textit{sfs}.\ \textit{sh}\ D' \neq \textit{Some}(\textit{sfs},\textit{Done});\ \forall\,\textit{Cs}.\ \textit{ics} \neq \textit{Called}\ \textit{Cs}\ ]\!]$
$\Longrightarrow \textit{exec-step-ind}\ (\textit{StepI}\ (\textit{Getstatic}\ C\ F\ D))\ P\ h\ \textit{stk}\ \textit{loc}\ C_0\ M_0\ \textit{pc}\ \textit{ics}\ \textit{frs}\ \textit{sh}$
$(\textit{None},\ h,\ (\textit{stk},\ \textit{loc},\ C_0,\ M_0,\ \textit{pc},\ \textit{Calling}\ D'\ [])\#\textit{frs},\ \textit{sh})$

| *exec-step-ind-Putfield-Null*:
$\textit{hd}(\textit{tl}\ \textit{stk}) = \textit{Null}$
$\Longrightarrow \textit{exec-step-ind}\ (\textit{StepI}\ (\textit{Putfield}\ F\ C))\ P\ h\ \textit{stk}\ \textit{loc}\ C_0\ M_0\ \textit{pc}\ \textit{ics}\ \textit{frs}\ \textit{sh}$
$(\lfloor \textit{addr-of-sys-xcpt}\ \textit{NullPointer} \rfloor,\ h,\ (\textit{stk},\ \textit{loc},\ C_0,\ M_0,\ \textit{pc},\ \textit{ics})\#\textit{frs},\ \textit{sh})$

| *exec-step-ind-Putfield-NoField*:
$[\![\ r = \textit{hd}(\textit{tl}\ \textit{stk});\ a = \textit{the-Addr}\ r;\ (D,\textit{fs}) = \textit{the}\ (h\ a);\ r \neq \textit{Null};\ \neg(\exists\,t\ b.\ P \vdash D\ \textit{has}\ F,b{:}t\ \textit{in}\ C)\ ]\!]$
$\Longrightarrow \textit{exec-step-ind}\ (\textit{StepI}\ (\textit{Putfield}\ F\ C))\ P\ h\ \textit{stk}\ \textit{loc}\ C_0\ M_0\ \textit{pc}\ \textit{ics}\ \textit{frs}\ \textit{sh}$
$(\lfloor \textit{addr-of-sys-xcpt}\ \textit{NoSuchFieldError} \rfloor,\ h,\ (\textit{stk},\ \textit{loc},\ C_0,\ M_0,\ \textit{pc},\ \textit{ics})\#\textit{frs},\ \textit{sh})$

| *exec-step-ind-Putfield-Static*:
$[\![\ r = \textit{hd}(\textit{tl}\ \textit{stk});\ a = \textit{the-Addr}\ r;\ (D,\textit{fs}) = \textit{the}\ (h\ a);\ r \neq \textit{Null};\ P \vdash D\ \textit{has}\ F,\textit{Static}{:}t\ \textit{in}\ C\ ]\!]$
$\Longrightarrow \textit{exec-step-ind}\ (\textit{StepI}\ (\textit{Putfield}\ F\ C))\ P\ h\ \textit{stk}\ \textit{loc}\ C_0\ M_0\ \textit{pc}\ \textit{ics}\ \textit{frs}\ \textit{sh}$
$(\lfloor \textit{addr-of-sys-xcpt}\ \textit{IncompatibleClassChangeError} \rfloor,\ h,\ (\textit{stk},\ \textit{loc},\ C_0,\ M_0,\ \textit{pc},\ \textit{ics})\#\textit{frs},\ \textit{sh})$

| *exec-step-ind-Putfield*:
$[\![\ v = \textit{hd}\ \textit{stk};\ r = \textit{hd}(\textit{tl}\ \textit{stk});\ a = \textit{the-Addr}\ r;\ (D,\textit{fs}) = \textit{the}\ (h\ a);\ (D',b,t) = \textit{field}\ P\ C\ F;$
$r \neq \textit{Null};\ P \vdash D\ \textit{has}\ F,\textit{NonStatic}{:}t\ \textit{in}\ C\ ]\!]$
$\Longrightarrow \textit{exec-step-ind}\ (\textit{StepI}\ (\textit{Putfield}\ F\ C))\ P\ h\ \textit{stk}\ \textit{loc}\ C_0\ M_0\ \textit{pc}\ \textit{ics}\ \textit{frs}\ \textit{sh}$
$(\textit{None},\ h(a \mapsto (D,\ \textit{fs}((F,C) \mapsto v))),\ (\textit{tl}\ (\textit{tl}\ \textit{stk}),\ \textit{loc},\ C_0,\ M_0,\ \textit{pc}{+}1,\ \textit{ics})\#\textit{frs},\ \textit{sh})$

| *exec-step-ind-Putstatic-NoField*:
$\neg(\exists\,t\ b.\ P \vdash C\ \textit{has}\ F,b{:}t\ \textit{in}\ D)$
$\Longrightarrow \textit{exec-step-ind}\ (\textit{StepI}\ (\textit{Putstatic}\ C\ F\ D))\ P\ h\ \textit{stk}\ \textit{loc}\ C_0\ M_0\ \textit{pc}\ \textit{ics}\ \textit{frs}\ \textit{sh}$
$(\lfloor \textit{addr-of-sys-xcpt}\ \textit{NoSuchFieldError} \rfloor,\ h,\ (\textit{stk},\ \textit{loc},\ C_0,\ M_0,\ \textit{pc},\ \textit{ics})\#\textit{frs},\ \textit{sh})$

| *exec-step-ind-Putstatic-NonStatic*:
P ⊢ *C has F*,*NonStatic*:*t in D*
  ⟹ *exec-step-ind* (*StepI* (*Putstatic C F D*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
    (⌊*addr-of-sys-xcpt IncompatibleClassChangeError*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Putstatic-Called*:
⟦ (D',*b*,*t*) = *field P D F*; *P* ⊢ *C has F*,*Static*:*t in D*; *the*(*sh* D') = (*sfs*,*i*) ⟧
  ⟹ *exec-step-ind* (*StepI* (*Putstatic C F D*)) *P h stk loc* C_0 M_0 *pc* (*Called Cs*) *frs sh*
    (*None*, *h*, (*tl stk*, *loc*, C_0, M_0, *Suc pc*, *No-ics*)#*frs*, *sh*(D':=*Some* ((*sfs*(*F* ↦ *hd stk*)), *i*)))

| *exec-step-ind-Putstatic-Done*:
⟦ (D',*b*,*t*) = *field P D F*; *P* ⊢ *C has F*,*Static*:*t in D*;
  ∀ *Cs*. *ics* ≠ *Called Cs*; *sh* D' = *Some* (*sfs*, *Done*) ⟧
  ⟹ *exec-step-ind* (*StepI* (*Putstatic C F D*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
    (*None*, *h*, (*tl stk*, *loc*, C_0, M_0, *Suc pc*, *ics*)#*frs*, *sh*(D':=*Some* ((*sfs*(*F* ↦ *hd stk*)), *Done*)))

| *exec-step-ind-Putstatic-Init*:
⟦ (D',*b*,*t*) = *field P D F*; *P* ⊢ *C has F*,*Static*:*t in D*;
  ∀ *sfs*. *sh* D' ≠ *Some* (*sfs*, *Done*); ∀ *Cs*. *ics* ≠ *Called Cs* ⟧
  ⟹ *exec-step-ind* (*StepI* (*Putstatic C F D*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
   (*None*, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *Calling* D' [])#*frs*, *sh*)

| *exec-step-ind-Checkcast*:
cast-ok P C h (*hd stk*)
  ⟹ *exec-step-ind* (*StepI* (*Checkcast C*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
   (*None*, *h*, (*stk*, *loc*, C_0, M_0, *Suc pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Checkcast-Error*:
¬*cast-ok P C h* (*hd stk*)
  ⟹ *exec-step-ind* (*StepI* (*Checkcast C*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
   (⌊*addr-of-sys-xcpt ClassCast*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Invoke-Null*:
stk!*n* = *Null*
  ⟹ *exec-step-ind* (*StepI* (*Invoke M n*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
   (⌊*addr-of-sys-xcpt NullPointer*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Invoke-NoMethod*:
⟦ *r* = *stk*!*n*; *C* = *fst*(*the*(*h*(*the-Addr r*))); *r* ≠ *Null*;
  ¬(∃ *Ts T m D b*. *P* ⊢ *C sees M*,*b*:*Ts* → *T* = *m in D*) ⟧
  ⟹ *exec-step-ind* (*StepI* (*Invoke M n*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
   (⌊*addr-of-sys-xcpt NoSuchMethodError*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Invoke-Static*:
⟦ *r* = *stk*!*n*; *C* = *fst*(*the*(*h*(*the-Addr r*)));

$(D,b,Ts,T,mxs,mxl_0,ins,xt)=$ *method P C M*; $r \neq$ *Null*;
$P \vdash C$ *sees* M,*Static*:$Ts \rightarrow T = m$ *in D* ⟧
$\Longrightarrow$ *exec-step-ind* (*StepI* (*Invoke M n*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
(⌊*addr-of-sys-xcpt IncompatibleClassChangeError*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Invoke*:
⟦ *ps* = *take n stk*; *r* = *stk*!*n*; *C* = *fst*(*the*(*h*(*the-Addr r*)));
$(D,b,Ts,T,mxs,mxl_0,ins,xt)=$ *method P C M*; $r \neq$ *Null*;
$P \vdash C$ *sees* M,*NonStatic*:$Ts \rightarrow T = m$ *in D*;
f' = ([],[*r*]@(*rev ps*)@(*replicate* mxl_0 *undefined*),*D*,*M*,*0*,*No-ics*) ⟧
$\Longrightarrow$ *exec-step-ind* (*StepI* (*Invoke M n*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
(*None*, *h*, f'#(*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Invokestatic-NoMethod*:
⟦ $(D,b,Ts,T,mxs,mxl_0,ins,xt)=$ *method P C M*; $\neg(\exists$ *Ts T m D b*. $P \vdash C$ *sees* M,*b*:$Ts \rightarrow T = m$ *in D*) ⟧
$\Longrightarrow$ *exec-step-ind* (*StepI* (*Invokestatic C M n*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
(⌊*addr-of-sys-xcpt NoSuchMethodError*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Invokestatic-NonStatic*:
⟦ $(D,b,Ts,T,mxs,mxl_0,ins,xt)=$ *method P C M*; $P \vdash C$ *sees* M,*NonStatic*:$Ts \rightarrow T = m$ *in D* ⟧
$\Longrightarrow$ *exec-step-ind* (*StepI* (*Invokestatic C M n*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
(⌊*addr-of-sys-xcpt IncompatibleClassChangeError*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Invokestatic-Called*:
⟦ *ps* = *take n stk*; $(D,b,Ts,T,mxs,mxl_0,ins,xt)$ = *method P C M*;
$P \vdash C$ *sees* M,*Static*:$Ts \rightarrow T = m$ *in D*;
f' = ([],(*rev ps*)@(*replicate* mxl_0 *undefined*),*D*,*M*,*0*,*No-ics*) ⟧
$\Longrightarrow$ *exec-step-ind* (*StepI* (*Invokestatic C M n*)) *P h stk loc* C_0 M_0 *pc* (*Called Cs*) *frs sh*
(*None*, *h*, f'#(*stk*, *loc*, C_0, M_0, *pc*, *No-ics*)#*frs*, *sh*)

| *exec-step-ind-Invokestatic-Done*:
⟦ *ps* = *take n stk*; $(D,b,Ts,T,mxs,mxl_0,ins,xt)$ = *method P C M*;
$P \vdash C$ *sees* M,*Static*:$Ts \rightarrow T = m$ *in D*;
∀ *Cs*. *ics* ≠ *Called Cs*; *sh D* = *Some* (*sfs*, *Done*);
f' = ([],(*rev ps*)@(*replicate* mxl_0 *undefined*),*D*,*M*,*0*,*No-ics*) ⟧
$\Longrightarrow$ *exec-step-ind* (*StepI* (*Invokestatic C M n*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
(*None*, *h*, f'#(*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Invokestatic-Init*:
⟦ $(D,b,Ts,T,mxs,mxl_0,ins,xt)$ = *method P C M*;
$P \vdash C$ *sees* M,*Static*:$Ts \rightarrow T = m$ *in D*;
∀ *sfs*. *sh D* ≠ *Some* (*sfs*, *Done*); ∀ *Cs*. *ics* ≠ *Called Cs* ⟧
$\Longrightarrow$ *exec-step-ind* (*StepI* (*Invokestatic C M n*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
(*None*, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *Calling D* [])#*frs*, *sh*)

| $exec\text{-}step\text{-}ind\text{-}Return\text{-}Last\text{-}Init$:
$exec\text{-}step\text{-}ind\ (StepI\ Return)\ P\ h\ stk_0\ loc_0\ C_0\ clinit\ pc\ ics\ []\ sh$
$(None,\ h,\ [],\ sh(C_0{:=}Some(fst(the(sh\ C_0)),\ Done)))$

| $exec\text{-}step\text{-}ind\text{-}Return\text{-}Last$:
$M_0 \neq clinit$
$\Longrightarrow exec\text{-}step\text{-}ind\ (StepI\ Return)\ P\ h\ stk_0\ loc_0\ C_0\ M_0\ pc\ ics\ []\ sh\ (None,\ h,\ [],$
$sh)$

| $exec\text{-}step\text{-}ind\text{-}Return\text{-}Init$:
$[\![\ (D,b,Ts,T,m) = method\ P\ C_0\ clinit\ ]\!]$
$\Longrightarrow exec\text{-}step\text{-}ind\ (StepI\ Return)\ P\ h\ stk_0\ loc_0\ C_0\ clinit\ pc\ ics\ ((stk',loc',C',m',pc',ics')\#frs')$
sh
$(None,\ h,\ (stk',loc',C',m',pc',ics')\#frs',\ sh(C_0{:=}Some(fst(the(sh\ C_0)),\ Done)))$

| $exec\text{-}step\text{-}ind\text{-}Return\text{-}NonStatic$:
$[\![\ (D,NonStatic,Ts,T,m) = method\ P\ C_0\ M_0;\ M_0 \neq clinit\ ]\!]$
$\Longrightarrow exec\text{-}step\text{-}ind\ (StepI\ Return)\ P\ h\ stk_0\ loc_0\ C_0\ M_0\ pc\ ics\ ((stk',loc',C',m',pc',ics')\#frs')$
sh
$(None,\ h,\ ((hd\ stk_0)\#(drop\ (length\ Ts + 1)\ stk'),loc',C',m',Suc\ pc',ics')\#frs',$
$sh)$

| $exec\text{-}step\text{-}ind\text{-}Return\text{-}Static$:
$[\![\ (D,Static,Ts,T,m) = method\ P\ C_0\ M_0;\ M_0 \neq clinit\ ]\!]$
$\Longrightarrow exec\text{-}step\text{-}ind\ (StepI\ Return)\ P\ h\ stk_0\ loc_0\ C_0\ M_0\ pc\ ics\ ((stk',loc',C',m',pc',ics')\#frs')$
sh
$(None,\ h,\ ((hd\ stk_0)\#(drop\ (length\ Ts)\ stk'),loc',C',m',Suc\ pc',ics')\#frs',\ sh)$

| $exec\text{-}step\text{-}ind\text{-}Pop$:
$exec\text{-}step\text{-}ind\ (StepI\ Pop)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
$(None,\ h,\ (tl\ stk,\ loc,\ C_0,\ M_0,\ Suc\ pc,\ ics)\#frs,\ sh)$

| $exec\text{-}step\text{-}ind\text{-}IAdd$:
$exec\text{-}step\text{-}ind\ (StepI\ IAdd)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
$(None,\ h,\ (Intg\ (the\text{-}Intg\ (hd\ (tl\ stk)) + the\text{-}Intg\ (hd\ stk))\#(tl\ (tl\ stk)),\ loc,\ C_0,$
$M_0,\ Suc\ pc,\ ics)\#frs,\ sh)$

| $exec\text{-}step\text{-}ind\text{-}IfFalse\text{-}False$:
$hd\ stk = Bool\ False$
$\Longrightarrow exec\text{-}step\text{-}ind\ (StepI\ (IfFalse\ i))\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
$(None,\ h,\ (tl\ stk,\ loc,\ C_0,\ M_0,\ nat(int\ pc{+}i),\ ics)\#frs,\ sh)$

| $exec\text{-}step\text{-}ind\text{-}IfFalse\text{-}nFalse$:
$hd\ stk \neq Bool\ False$
$\Longrightarrow exec\text{-}step\text{-}ind\ (StepI\ (IfFalse\ i))\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
$(None,\ h,\ (tl\ stk,\ loc,\ C_0,\ M_0,\ Suc\ pc,\ ics)\#frs,\ sh)$

| $exec\text{-}step\text{-}ind\text{-}CmpEq$:

exec-step-ind (*StepI CmpEq*) *P h stk loc* C_0 M_0 *pc ics frs sh*
(*None*, *h*, (*Bool* (*hd* (*tl stk*) = *hd stk*) # *tl* (*tl stk*), *loc*, C_0, M_0, *Suc pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Goto*:
exec-step-ind (*StepI* (*Goto i*)) *P h stk loc* C_0 M_0 *pc ics frs sh*
(*None*, *h*, (*stk*, *loc*, C_0, M_0, *nat*(*int pc*+*i*), *ics*)#*frs*, *sh*)

| *exec-step-ind-Throw*:
hd stk ≠ *Null*
⟹ *exec-step-ind* (*StepI Throw*) *P h stk loc* C_0 M_0 *pc ics frs sh*
(⌊*the-Addr* (*hd stk*)⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Throw-Null*:
hd stk = *Null*
⟹ *exec-step-ind* (*StepI Throw*) *P h stk loc* C_0 M_0 *pc ics frs sh*
(⌊*addr-of-sys-xcpt NullPointer*⌋, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Init-None-Called*:
⟦ *sh C* = *None* ⟧
⟹ *exec-step-ind* (*StepC C Cs*) *P h stk loc* C_0 M_0 *pc ics frs sh*
(*None*, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *Calling C Cs*)#*frs*, *sh*(*C* := *Some* (*sblank P C*, *Prepared*)))

| *exec-step-ind-Init-Done*:
sh C = *Some* (*sfs*, *Done*)
⟹ *exec-step-ind* (*StepC C Cs*) *P h stk loc* C_0 M_0 *pc ics frs sh*
(*None*, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *Called Cs*)#*frs*, *sh*)

| *exec-step-ind-Init-Processing*:
sh C = *Some* (*sfs*, *Processing*)
⟹ *exec-step-ind* (*StepC C Cs*) *P h stk loc* C_0 M_0 *pc ics frs sh*
(*None*, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *Called Cs*)#*frs*, *sh*)

| *exec-step-ind-Init-Error*:
⟦ *sh C* = *Some* (*sfs*, *Error*) ⟧
⟹ *exec-step-ind* (*StepC C Cs*) *P h stk loc* C_0 M_0 *pc ics frs sh*
(*None*, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *Throwing Cs* (*addr-of-sys-xcpt NoClassDefFoundError*))#*frs*, *sh*)

| *exec-step-ind-Init-Prepared-Object*:
⟦ *sh C* = *Some* (*sfs*, *Prepared*);
sh′ = *sh*(*C*:=*Some*(*fst*(*the*(*sh C*)), *Processing*));
C = *Object* ⟧
⟹ *exec-step-ind* (*StepC C Cs*) *P h stk loc* C_0 M_0 *pc ics frs sh*
(*None*, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *Called* (*C*#*Cs*))#*frs*, *sh′*)

| *exec-step-ind-Init-Prepared-nObject*:
⟦ *sh C* = *Some* (*sfs*, *Prepared*);

$sh' = sh(C{:=}Some(fst(the(sh\ C)),\ Processing))$;
$C \neq Object$; $D = fst(the(class\ P\ C))$ ⟧
$\Longrightarrow$ *exec-step-ind* (*StepC C Cs*) *P h stk loc* C_0 M_0 *pc ics frs sh*
(*None*, *h*, (*stk*, *loc*, C_0, M_0, *pc*, *Calling D* ($C\#Cs$))#*frs*, *sh'*)

| *exec-step-ind-Init*:
exec-step-ind (*StepC2 C Cs*) *P h stk loc* C_0 M_0 *pc ics frs sh*
(*None*, *h*, *create-init-frame P C*#(*stk*, *loc*, C_0, M_0, *pc*, *Called Cs*)#*frs*, *sh*)

| *exec-step-ind-InitThrow*:
exec-step-ind (*StepT* ($C\#Cs$) *a*) *P h stk loc* C_0 M_0 *pc ics frs sh*
(*None*, *h*, (*stk*,*loc*,C_0,M_0,*pc*,*Throwing Cs a*)#*frs*, (*sh*($C \mapsto$ (*fst*(*the*(*sh C*)), *Error*))))

| *exec-step-ind-InitThrow-End*:
exec-step-ind (*StepT* [] *a*) *P h stk loc* C_0 M_0 *pc ics frs sh*
($\lfloor a \rfloor$, *h*, (*stk*,*loc*,C_0,M_0,*pc*,*No-ics*)#*frs*, *sh*)

inductive-cases *exec-step-ind-cases* [*cases set*]:
exec-step-ind (*StepI* (*Load n*)) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepI* (*Store n*)) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepI* (*Push v*)) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepI* (*New C*)) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepI* (*Getfield F C*)) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepI* (*Getstatic C F D*)) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepI* (*Putfield F C*)) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepI* (*Putstatic C F D*)) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepI* (*Checkcast C*)) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepI* (*Invoke M n*)) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepI* (*Invokestatic C M n*)) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepI Return*) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepI Pop*) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepI IAdd*) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepI* (*IfFalse i*)) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepI CmpEq*) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepI* (*Goto i*)) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepI Throw*) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepC C' Cs*) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepC2 C' Cs*) *P h stk loc C M pc ics frs sh* σ
exec-step-ind (*StepT Cs a*) *P h stk loc C M pc ics frs sh* σ

— Deriving *step-input* for *exec-step-ind* from *exec-step* arguments
fun *exec-step-input* :: [*jvm-prog*, *cname*, *mname*, *pc*, *init-call-status*] $\Rightarrow$ *step-input*
where
exec-step-input P C M pc (*Calling C' Cs*) = *StepC C' Cs* |
exec-step-input P C M pc (*Called* ($C'\#Cs$)) = *StepC2 C' Cs* |

exec-step-input P C M pc (*Throwing Cs a*) = *StepT Cs a* |
exec-step-input P C M pc ics = *StepI* (*instrs-of P C M* ! *pc*)

lemma *exec-step-input-StepTD*[*simp*]:
assumes *exec-step-input P C M pc ics* = *StepT Cs a* **shows** *ics* = *Throwing Cs a*
using *assms* **proof**(*cases ics*)
  case (*Called Cs*) **with** *assms* **show** *?thesis* **by**(*cases Cs*; *simp*)
qed(*auto*)

lemma *exec-step-input-StepCD*[*simp*]:
assumes *exec-step-input P C M pc ics* = *StepC C′ Cs* **shows** *ics* = *Calling C′ Cs*
using *assms* **proof**(*cases ics*)
  case (*Called Cs*) **with** *assms* **show** *?thesis* **by**(*cases Cs*; *simp*)
qed(*auto*)

lemma *exec-step-input-StepC2D*[*simp*]:
assumes *exec-step-input P C M pc ics* = *StepC2 C′ Cs* **shows** *ics* = *Called* (*C′#Cs*)
using *assms* **proof**(*cases ics*)
  case (*Called Cs*) **with** *assms* **show** *?thesis* **by**(*cases Cs*; *simp*)
qed(*auto*)

lemma *exec-step-input-StepID*:
assumes *exec-step-input P C M pc ics* = *StepI i*
shows (*ics* = *Called* [] ∨ *ics* = *No-ics*) ∧ *instrs-of P C M* ! *pc* = *i*
using *assms* **proof**(*cases ics*)
  case (*Called Cs*) **with** *assms* **show** *?thesis* **by**(*cases Cs*; *simp*)
qed(*auto*)

11.1 Equivalence of *exec-step* and *exec-step-input*

lemma *exec-step-imp-exec-step-ind*:
assumes *es*: *exec-step P h stk loc C M pc ics frs sh* = (*xp′*, *h′*, *frs′*, *sh′*)
shows *exec-step-ind* (*exec-step-input P C M pc ics*) *P h stk loc C M pc ics frs sh* (*xp′*, *h′*, *frs′*, *sh′*)
proof(*cases exec-step-input P C M pc ics*)
  case (*StepT Cs a*)
  then have *ics* = *Throwing Cs a* **by** *simp*
  then show *?thesis* **using** *exec-step-ind-InitThrow exec-step-ind-InitThrow-End StepT es*
   by(*cases Cs*, *auto*)
next
  case (*StepC C1 Cs*)
  then have *ics*: *ics* = *Calling C1 Cs* **by** *simp*
  obtain *D b Ts T m* **where** *lets*: *method P C1 clinit* = (*D*,*b*,*Ts*,*T*,*m*) **by**(*cases method P C1 clinit*)
  then obtain *mxs* mxl_0 *ins xt* **where** *m*: *m* = (*mxs*,mxl_0,*ins*,*xt*) **by**(*cases m*)
  show *?thesis*
  proof(*cases sh C1*)

```
    case None then show ?thesis
     using exec-step-ind-Init-None-Called ics assms by auto
  next
    case (Some a)
    then obtain sfs i where sfsi: a = (sfs,i) by(cases a)
    then show ?thesis using exec-step-ind-Init-Done exec-step-ind-Init-Processing
      exec-step-ind-Init-Error m lets Some ics assms
    proof(cases i)
      case Prepared
      show ?thesis
    using exec-step-ind-Init-Prepared-Object[where P=P] exec-step-ind-Init-Prepared-nObject
       sfsi m lets Prepared Some ics assms by(auto split: if-split-asm)
    qed(auto)
  qed
next
  case (StepC2 C1 Cs)
  then have ics: ics = Called (C1#Cs) by simp
  then show ?thesis using exec-step-ind-Init assms by auto
next
  case (StepI i)
  then have
    ics: ics = Called [] ∨ ics = No-ics and
    exec-instr: exec-instr i P h stk loc C M pc ics frs sh = (xp', h', frs', sh')
    using assms by(auto dest!: exec-step-input-StepID)
  show ?thesis
  proof(cases i)
     case (Load x1) then show ?thesis using exec-instr exec-step-ind-Load StepI
by auto
  next
     case (Store x2) then show ?thesis using exec-instr exec-step-ind-Store StepI
by auto
  next
     case (Push x3) then show ?thesis using exec-instr exec-step-ind-Push StepI
by auto
  next
    case (New C1)
    then obtain sfs i where sfsi: the(sh C1) = (sfs,i) by(cases the(sh C1))
   then show ?thesis using exec-step-ind-New-Called exec-step-ind-NewOOM-Called
       exec-step-ind-New-Done exec-step-ind-NewOOM-Done
     exec-step-ind-New-Init sfsi New StepI exec-instr ics by(auto split: init-state.splits)
  next
    case (Getfield F1 C1)
    then obtain D fs D' b t where lets: the(h(the-Addr (hd stk))) = (D,fs)
      field P C1 F1 = (D',b,t) by(cases the(h(the-Addr (hd stk))), cases field P C1
F1)
    then have ⋀b' t'. P ⊢ D has F1,b':t' in C1 ⟹ (D', b, t) = (C1, b', t')
      using field-def2 has-field-idemp has-field-sees by fastforce
   then show ?thesis using exec-step-ind-Getfield-Null exec-step-ind-Getfield-NoField
      exec-step-ind-Getfield-Static exec-step-ind-Getfield lets Getfield StepI exec-instr
```

by(*auto split*: *if-split-asm staticb.splits*) *metis*+
next
case (*Getstatic C1 F1 D1*)
then obtain *D′ b t* **where** *lets*: *field P D1 F1* = (*D′*,*b*,*t*) **by**(*cases field P D1 F1*)
then have *field*: $\bigwedge$*b′ t′*. *P* ⊢ *C1 has F1*,*b′*:*t′ in D1* ⟹ (*D′*, *b*, *t*) = (*D1*, *b′*, *t′*)
using *field-def2 has-field-idemp has-field-sees* **by** *fastforce*
show *?thesis*
proof(*cases b*)
case *NonStatic* **then show** *?thesis*
using *exec-step-ind-Getstatic-NoField exec-step-ind-Getstatic-NonStatic field lets Getstatic exec-instr StepI* **by**(*auto split*: *if-split-asm*) *fastforce*
next
case *Static* **show** *?thesis*
proof(*cases ics* = *Called* [])
case *True* **then show** *?thesis* **using** *exec-step-ind-Getstatic-NoField exec-step-ind-Getstatic-Called exec-step-ind-Getstatic-Init Static field lets Getstatic exec-instr StepI ics*
by(*auto simp*: *split-beta split*: *if-split-asm*) *metis*
next
case *False*
then have *nCalled*: ∀ *Cs*. *ics* ≠ *Called Cs* **using** *ics* **by** *simp*
show *?thesis*
proof(*cases sh D1*)
case *None*
then have *nDone*: ∀ *sfs*. *sh D1* ≠ *Some*(*sfs*, *Done*) **by** *simp*
then show *?thesis* **using** *exec-step-ind-Getstatic-NoField exec-step-ind-Getstatic-Init*[**where** *sh*=*sh*, *OF* - - *nDone nCalled*] *field lets None False Static Getstatic exec-instr StepI ics*
by(*auto split*: *if-split-asm*) *metis*
next
case (*Some a*)
then obtain *sfs i* **where** *sfsi*: *a*=(*sfs*,*i*) **by**(*cases a*)
show *?thesis* **using** *exec-step-ind-Getstatic-NoField exec-step-ind-Getstatic-Init sfsi False Static Some field lets Getstatic exec-instr*
proof(*cases i* = *Done*)
case *True* **then show** *?thesis* **using** *exec-step-ind-Getstatic-NoField exec-step-ind-Getstatic-Done*[*OF* - - *nCalled*] *exec-step-ind-Getstatic-Init sfsi False Static Some field lets Getstatic exec-instr StepI ics*
by(*auto split*: *if-split-asm*) *metis*
next
case *nD*: *False*
then have *nDone*: ∀ *sfs*. *sh D1* ≠ *Some*(*sfs*, *Done*) **using** *sfsi Some* **by** *simp*
show *?thesis* **using** *nD*
proof(*cases i*)
case *Processing* **then show** *?thesis* **using** *exec-step-ind-Getstatic-NoField exec-step-ind-Getstatic-Init*[**where** *sh*=*sh*, *OF* - - *nDone nCalled*]

```
          sfsi False Static Some field lets Getstatic exec-instr StepI ics
        by(auto split: if-split-asm) metis
      next
      case Prepared then show ?thesis using exec-step-ind-Getstatic-NoField
          exec-step-ind-Getstatic-Init[where sh=sh, OF - - nDone nCalled]
          sfsi False Static Some field lets Getstatic exec-instr StepI ics
        by(auto split: if-split-asm) metis
      next
        case Error then show ?thesis using exec-step-ind-Getstatic-NoField
          exec-step-ind-Getstatic-Init[where sh=sh, OF - - nDone nCalled]
          sfsi False Static Some field lets Getstatic exec-instr StepI ics
        by(auto split: if-split-asm) metis
      qed(simp)
     qed
    qed
   qed
  qed
 next
  case (Putfield F1 C1)
  then obtain D fs D′ b t where lets: the(h(the-Addr (hd(tl stk)))) = (D,fs)
    field P C1 F1 = (D′,b,t) by(cases the(h(the-Addr (hd(tl stk)))), cases field P
C1 F1)
  then have ⋀b′ t′. P ⊢ D has F1,b′:t′ in C1 ⟹ (D′, b, t) = (C1, b′, t′)
    using field-def2 has-field-idemp has-field-sees by fastforce
 then show ?thesis using exec-step-ind-Putfield-Null exec-step-ind-Putfield-NoField
    exec-step-ind-Putfield-Static exec-step-ind-Putfield lets Putfield exec-instr StepI
    by(auto split: if-split-asm staticb.splits) metis+
 next
  case (Putstatic C1 F1 D1)
  then obtain D′ b t where lets: field P D1 F1 = (D′,b,t) by(cases field P D1
F1)
  then have field: ⋀b′ t′. P ⊢ C1 has F1,b′:t′ in D1 ⟹ (D′, b, t) = (D1, b′, t′)
    using field-def2 has-field-idemp has-field-sees by fastforce
  show ?thesis
  proof(cases b)
    case NonStatic then show ?thesis
     using exec-step-ind-Putstatic-NoField exec-step-ind-Putstatic-NonStatic
      field lets Putstatic exec-instr StepI by(auto split: if-split-asm) fastforce
  next
    case Static show ?thesis
    proof(cases ics = Called [])
      case True then show ?thesis using exec-step-ind-Putstatic-NoField
        exec-step-ind-Putstatic-Called exec-step-ind-Putstatic-Init
        Static field lets Putstatic exec-instr StepI ics
       by(cases the(sh D1), auto split: if-split-asm) metis
    next
      case False
      then have nCalled: ∀ Cs. ics ≠ Called Cs using ics by simp
      show ?thesis
```

proof(*cases sh D1*)
case *None*
then have *nDone*: ∀ *sfs. sh D1* ≠ *Some*(*sfs*, *Done*) **by** *simp*
then show *?thesis* **using** *exec-step-ind-Putstatic-NoField*
exec-step-ind-Putstatic-Init[**where** *sh*=*sh*, *OF - - nDone nCalled*]
field lets None False Static Putstatic exec-instr StepI ics
by(*auto split*: *if-split-asm*) *metis*
next
case (*Some a*)
then obtain *sfs i* **where** *sfsi*: *a*=(*sfs*,*i*) **by**(*cases a*)
show *?thesis* **using** *exec-step-ind-Putstatic-NoField*
exec-step-ind-Putstatic-Init sfsi False Static Some field lets Putstatic exec-instr
proof(*cases i* = *Done*)
case *True* **then show** *?thesis* **using** *exec-step-ind-Putstatic-NoField*
exec-step-ind-Putstatic-Done[*OF - - nCalled*] *exec-step-ind-Putstatic-Init*
sfsi False Static Some field lets Putstatic exec-instr StepI ics
by(*auto split*: *if-split-asm*) *metis*
next
case *nD*: *False*
then have *nDone*: ∀ *sfs. sh D1* ≠ *Some*(*sfs*, *Done*) **using** *sfsi Some* **by** *simp*
show *?thesis* **using** *nD*
proof(*cases i*)
case *Processing* **then show** *?thesis* **using** *exec-step-ind-Putstatic-NoField*
exec-step-ind-Putstatic-Init[**where** *sh*=*sh*, *OF - - nDone nCalled*]
sfsi False Static Some field lets Putstatic exec-instr StepI ics
by(*auto split*: *if-split-asm*) *metis*
next
case *Prepared* **then show** *?thesis* **using** *exec-step-ind-Putstatic-NoField*
exec-step-ind-Putstatic-Init[**where** *sh*=*sh*, *OF - - nDone nCalled*]
sfsi False Static Some field lets Putstatic exec-instr StepI ics
by(*auto split*: *if-split-asm*) *metis*
next
case *Error* **then show** *?thesis* **using** *exec-step-ind-Putstatic-NoField*
exec-step-ind-Putstatic-Init[**where** *sh*=*sh*, *OF - - nDone nCalled*]
sfsi False Static Some field lets Putstatic exec-instr StepI ics
by(*auto split*: *if-split-asm*) *metis*
qed(*simp*)
qed
qed
qed
qed
next
case *Checkcast* **then show** *?thesis*
using *exec-step-ind-Checkcast exec-step-ind-Checkcast-Error exec-instr StepI*
by(*auto split*: *if-split-asm*)
next
case (*Invoke M1 n*) **show** *?thesis*

proof(*cases stk*!*n* = *Null*)
case *True* **then show** *?thesis* **using** *exec-step-ind-Invoke-Null Invoke exec-instr StepI*
by *clarsimp*
next
case *False*
let *?C* = *cname-of h* (*the-Addr* (*stk* ! *n*))
obtain *D b Ts T m* **where** *method*: *method P ?C M1* = (*D*,*b*,*Ts*,*T*,*m*) **by**(*cases method P ?C M1*)
then obtain *mxs* mxl_0 *ins xt* **where** *m* = (*mxs*,mxl_0,*ins*,*xt*) **by**(*cases m*)
then show *?thesis* **using** *exec-step-ind-Invoke-NoMethod exec-step-ind-Invoke-Static exec-step-ind-Invoke method False Invoke exec-instr StepI*
by(*auto split*: *if-split-asm staticb.splits*)
qed
next
case (*Invokestatic C1 M1 n*)
obtain *D b Ts T m* **where** *lets*: *method P C1 M1* = (*D*,*b*,*Ts*,*T*,*m*) **by**(*cases method P C1 M1*)
then obtain *mxs* mxl_0 *ins xt* **where** *m*: *m* = (*mxs*,mxl_0,*ins*,*xt*) **by**(*cases m*)
have *method*: $\bigwedge b'\ Ts'\ t'\ m'\ D'.\ P \vdash C1\ sees\ M1{,}b'{:}Ts' \to t' = m'\ in\ D'$
$\Longrightarrow (D,b,Ts,T,m) = (D',b',Ts',t',m')$ **using** *lets* **by** *auto*
show *?thesis*
proof(*cases b*)
case *NonStatic* **then show** *?thesis*
using *exec-step-ind-Invokestatic-NoMethod exec-step-ind-Invokestatic-NonStatic m method lets Invokestatic exec-instr StepI* **by**(*auto split*: *if-split-asm*)
next
case *Static* **show** *?thesis*
proof(*cases ics* = *Called* [])
case *True* **then show** *?thesis* **using** *exec-step-ind-Invokestatic-NoMethod exec-step-ind-Invokestatic-Called exec-step-ind-Invokestatic-Init Static m method lets Invokestatic exec-instr StepI ics*
by(*auto split*: *if-split-asm*)
next
case *False*
then have *nCalled*: ∀ *Cs*. *ics* ≠ *Called Cs* **using** *ics* **by** *simp*
show *?thesis*
proof(*cases sh D*)
case *None*
then have *nDone*: ∀ *sfs*. *sh D* ≠ *Some*(*sfs*, *Done*) **by** *simp*
show *?thesis* **using** *exec-step-ind-Invokestatic-NoMethod exec-step-ind-Invokestatic-Init*[**where** *sh*=*sh*, *OF - - nDone nCalled*] *method m lets None False Static Invokestatic exec-instr StepI ics*
by(*auto split*: *if-split-asm*)
next
case (*Some a*)
then obtain *sfs i* **where** *sfsi*: *a*=(*sfs*,*i*) **by**(*cases a*)
show *?thesis* **using** *exec-step-ind-Invokestatic-NoMethod*

exec-step-ind-Invokestatic-Init sfsi False Static Some method lets Invokestatic exec-instr
proof(*cases i = Done*)
case *True* **then show** *?thesis* **using** *exec-step-ind-Invokestatic-NoMethod exec-step-ind-Invokestatic-Done*[*OF - - - nCalled*] *exec-step-ind-Invokestatic-Init sfsi False Static Some m method lets Invokestatic exec-instr StepI ics*
by(*auto split*: *if-split-asm*)
next
case *nD*: *False*
then have *nDone*: $\forall$ *sfs. sh D* $\neq$ *Some*(*sfs*, *Done*) **using** *sfsi Some* **by** *simp*
show *?thesis* **using** *nD*
proof(*cases i*)
case *Processing* **then show** *?thesis* **using** *exec-step-ind-Invokestatic-NoMethod exec-step-ind-Invokestatic-Init*[**where** *sh=sh*, *OF - - nDone nCalled*] *sfsi False Static Some m method lets Invokestatic exec-instr StepI ics*
by(*auto split*: *if-split-asm*)
next
case *Prepared* **then show** *?thesis* **using** *exec-step-ind-Invokestatic-NoMethod exec-step-ind-Invokestatic-Init*[**where** *sh=sh*, *OF - - nDone nCalled*] *sfsi False Static Some m method lets Invokestatic exec-instr StepI ics*
by(*auto split*: *if-split-asm*)
next
case *Error* **then show** *?thesis* **using** *exec-step-ind-Invokestatic-NoMethod exec-step-ind-Invokestatic-Init*[**where** *sh=sh*, *OF - - nDone nCalled*] *sfsi False Static Some m method lets Invokestatic exec-instr StepI ics*
by(*auto split*: *if-split-asm*)
qed(*simp*)
qed
qed
qed
qed
next
case *Return*
obtain *D b Ts T m* **where** *method*: *method P C M* = (*D*,*b*,*Ts*,*T*,*m*) **by**(*cases method P C M*)
then obtain *mxs* mxl_0 *ins xt* **where** *m* = (*mxs*,mxl_0,*ins*,*xt*) **by**(*cases m*)
then show *?thesis* **using** *exec-step-ind-Return-Last-Init exec-step-ind-Return-Last exec-step-ind-Return-Init exec-step-ind-Return-NonStatic exec-step-ind-Return-Static method Return exec-instr StepI ics*
by(*auto split*: *if-split-asm staticb.splits bool.splits list.splits*)
next
case *Pop* **then show** *?thesis* **using** *exec-instr StepI exec-step-ind-Pop* **by** *auto*
next
case *IAdd* **then show** *?thesis* **using** *exec-instr StepI exec-step-ind-IAdd* **by** *auto*
next
case *Goto* **then show** *?thesis* **using** *exec-instr StepI exec-step-ind-Goto* **by** *auto*

next
case *CmpEq* **then show** *?thesis* **using** *exec-instr StepI exec-step-ind-CmpEq* **by** *auto*
next
case (*IfFalse x17*) **then show** *?thesis*
using *exec-instr StepI exec-step-ind-IfFalse-nFalse exec-step-ind-IfFalse-False exec-instr StepI* **by**(*auto split*: *val.splits staticb.splits*)
next
case *Throw* **then show** *?thesis*
using *exec-instr StepI exec-step-ind-Throw exec-step-ind-Throw-Null*
by(*auto split*: *val.splits*)
qed
qed

lemma *exec-step-ind-imp-exec-step*:
assumes *esi*: *exec-step-ind si P h stk loc C M pc ics frs sh* (xp', h', frs', sh')
and *si*: *exec-step-input P C M pc ics* = *si*
shows *exec-step P h stk loc C M pc ics frs sh* = (xp', h', frs', sh')
proof −
have *StepI*:
$\bigwedge P\ C\ M\ pc\ Cs\ i$. *exec-step-input P C M pc* (*Called Cs*) = *StepI i*
$\Longrightarrow$ *instrs-of P C M* ! *pc* = *i* $\wedge$ *Cs* = []
proof −
fix *P C M pc Cs i* **show** *exec-step-input P C M pc* (*Called Cs*) = *StepI i*
$\Longrightarrow$ *instrs-of P C M* ! *pc* = *i* $\wedge$ *Cs* = [] **by**(*cases Cs*; *simp*)
qed
have *StepC*:
$\bigwedge P\ C\ M\ pc\ ics\ C'\ Cs$. *exec-step-input P C M pc ics* = *StepC* C' *Cs* $\Longrightarrow$ *ics* = *Calling* C' *Cs*
by *simp*
have *StepT*:
$\bigwedge P\ C\ M\ pc\ ics\ Cs\ a$. *exec-step-input P C M pc ics* = *StepT Cs a* $\Longrightarrow$ *ics* = *Throwing Cs a*
by *simp*
show *?thesis* **using** *assms*
proof(*induct rule*: *exec-step-ind.induct*)
case (*exec-step-ind-NewOOM-Done sh C obj h ics P stk loc* C_0 M_0 *pc frs*)
then show *?case* **by**(*cases ics*, *auto*)
next
case (*exec-step-ind-New-Done sh C obj h a ics P stk loc* C_0 M_0 *pc frs*)
then show *?case* **by**(*cases ics*, *auto*)
next
case (*exec-step-ind-New-Init sh C ics P h stk loc* C_0 M_0 *pc frs*)
then show *?case* **by**(*cases ics*, *auto split*: *init-state.splits*)
next
case (*exec-step-ind-Getfield-NoField v stk D fs h P F C loc* C_0 M_0 *pc ics frs sh*)
then show *?case* **by**(*cases the* (*h* (*the-Addr* (*hd stk*))), *cases ics*, *auto dest*!: *StepI*)

next
case (*exec-step-ind-Getfield-Static v stk D fs h P F t C loc* C_0 M_0 *pc ics frs sh*)
then show *?case*
by(*cases the* (*h* (*the-Addr* (*hd stk*))), *cases fst*(*snd*(*field P C F*)),
cases ics, *auto simp*: *split-beta dest*: *has-field-sees*[*OF has-field-idemp*] *dest*!: *StepI*)
next
case (*exec-step-ind-Getfield v stk D fs h D′ b t P C F loc* C_0 M_0 *pc ics frs sh*)
then show *?case*
by(*cases the* (*h* (*the-Addr* (*hd stk*))),
cases ics; *fastforce simp*: *split-beta dest*: *has-field-sees*[*OF has-field-idemp*] *dest*!: *StepI*)
next
case (*exec-step-ind-Getstatic-NonStatic P C F t D h stk loc* C_0 M_0 *pc ics frs sh*)
then show *?case*
by(*cases ics*; *fastforce simp*: *split-beta split*: *staticb.splits*
dest: *has-field-sees*[*OF has-field-idemp*] *dest*!: *StepI*)
next
case *exec-step-ind-Getstatic-Called*
then show *?case* **by**(*fastforce simp*: *split-beta split*: *staticb.splits dest*!: *StepI*
dest: *has-field-sees*[*OF has-field-idemp*])
next
case (*exec-step-ind-Getstatic-Done D′ b t P D F C ics sh sfs v h stk loc* C_0 M_0 *pc frs*)
then show *?case* **by**(*cases ics*, *auto simp*: *split-beta split*: *staticb.splits*
dest: *has-field-sees*[*OF has-field-idemp*])
next
case (*exec-step-ind-Getstatic-Init D′ b t P D F C sh ics h stk loc* C_0 M_0 *pc frs*)
then show *?case*
by(*cases ics*, *auto simp*: *split-beta split*: *init-state.splits staticb.splits*
dest: *has-field-sees*[*OF has-field-idemp*])
next
case (*exec-step-ind-Putfield-NoField r stk a D fs h P F C loc* C_0 M_0 *pc ics frs sh*)
then show *?case* **by**(*cases the* (*h* (*the-Addr* (*hd*(*tl stk*)))), *cases ics*, *auto dest*!: *StepI*)
next
case (*exec-step-ind-Putfield-Static r stk a D fs h P F t C loc* C_0 M_0 *pc ics frs sh*)
then show *?case*
by(*cases the* (*h* (*the-Addr* (*hd*(*tl stk*)))), *cases fst*(*snd*(*field P C F*)),
cases ics, *auto simp*: *split-beta dest*: *has-field-sees*[*OF has-field-idemp*] *dest*!: *StepI*)
next
case (*exec-step-ind-Putfield v stk r a D fs h D′ b t P C F loc* C_0 M_0 *pc ics frs sh*)
then show *?case*
by(*cases the* (*h* (*the-Addr* (*hd*(*tl stk*)))),

cases ics; *fastforce simp*: *split-beta dest*: *has-field-sees*[*OF has-field-idemp*] *dest*!: *StepI*)
next
case (*exec-step-ind-Putstatic-NonStatic P C F t D h stk loc* C_0 M_0 *pc ics frs sh*)
then show *?case*
by(*cases ics*; *fastforce simp*: *split-beta split*: *staticb.splits*
dest: *has-field-sees*[*OF has-field-idemp*] *dest*!: *StepI*)
next
case *exec-step-ind-Putstatic-Called*
then show *?case* **by**(*fastforce simp*: *split-beta split*: *staticb.splits dest*!: *StepI*
dest: *has-field-sees*[*OF has-field-idemp*])
next
case (*exec-step-ind-Putstatic-Done* D' *b t P D F C ics sh sfs h stk loc* C_0 M_0 *pc frs*)
then show *?case* **by**(*cases ics*, *auto simp*: *split-beta split*: *staticb.splits*
dest: *has-field-sees*[*OF has-field-idemp*])
next
case (*exec-step-ind-Putstatic-Init* D' *b t P D F C sh ics h stk loc* C_0 M_0 *pc frs*)
then show *?case*
by(*cases ics*, *auto simp*: *split-beta split*: *staticb.splits init-state.splits*
dest: *has-field-sees*[*OF has-field-idemp*])
next
case (*exec-step-ind-Invoke ps n stk r C h D b Ts T mxs* mxl_0 *ins xt P M m* f' *loc* C_0 M_0 *pc ics frs sh*)
then show *?case* **by**(*cases ics*; *fastforce dest*!: *StepI*)
next
case (*exec-step-ind-Invokestatic-Called ps n stk D b Ts T mxs* mxl_0 *ins xt P C M m ics* ics' *sh*)
then show *?case* **by**(*cases ics*; *fastforce dest*!: *StepI*)
next
case (*exec-step-ind-Invokestatic-Done ps n stk D b Ts T mxs* mxl_0 *ins xt P C M m ics sh sfs* f')
then show *?case* **by**(*cases ics*; *fastforce*)
next
case (*exec-step-ind-Invokestatic-Init D b Ts T mxs* mxl_0 *ins xt P C M m sh ics n h stk loc* C_0 M_0 *pc frs*)
then show *?case* **by**(*cases ics*; *fastforce split*: *init-state.splits*)
next
case (*exec-step-ind-Return-NonStatic D Ts T m P* C_0 M_0 *h* stk_0 loc_0 *pc ics* stk' loc' C' m' pc' ics' frs' *sh*)
then show *?case* **by**(*cases method P* C_0 M_0, *cases ics*, *auto dest*!: *StepI*)
next
case (*exec-step-ind-Return-Static D Ts T m P* C_0 M_0 *h* stk_0 loc_0 *pc ics* stk' loc' C' m' pc' ics' frs' *sh*)
then show *?case* **by**(*cases method P* C_0 M_0, *cases ics*, *auto dest*!: *StepI*)
next
case (*exec-step-ind-IfFalse-nFalse stk i P h loc* C_0 M_0 *pc ics frs sh*)
then show *?case* **by**(*cases hd stk*; *cases ics*, *auto dest*!: *StepI*)

next
  case (*exec-step-ind-Throw-Null stk P h loc* C_0 M_0 *pc ics frs sh*)
  then show *?case* **by**(*cases hd stk*; *cases ics*, *auto dest*!: *StepI*)
next
  case (*exec-step-ind-Init C Cs P h stk loc* C_0 M_0 *pc ics frs sh*)
  then have *ics* = *Called* (*C*#*Cs*) **by** *simp*
  then show *?case* **by** *auto*

next
  case (*exec-step-ind-Load n P h stk loc* C_0 M_0 *pc ics frs sh*)
  then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
  case (*exec-step-ind-Store n P h stk loc* C_0 M_0 *pc ics frs sh*)
  then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
  case (*exec-step-ind-Push v P h stk loc* C_0 M_0 *pc ics frs sh*)
  then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
  case (*exec-step-ind-NewOOM-Called h C P stk loc* C_0 M_0 *pc frs sh* ics')
  then show *?case* **by**(*auto dest*!: *StepI*)
next
  case (*exec-step-ind-New-Called h a C P stk loc* C_0 M_0 *pc frs sh* ics')
  then show *?case* **by**(*auto dest*!: *StepI*)
next
  case (*exec-step-ind-Getfield-Null stk F C P h loc* C_0 M_0 *pc ics frs sh*)
  then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
  case (*exec-step-ind-Getstatic-NoField P C F D h stk loc* C_0 M_0 *pc ics frs sh*)
  then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
  case (*exec-step-ind-Putfield-Null stk F C P h loc* C_0 M_0 *pc ics frs sh*)
  then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
  case (*exec-step-ind-Putstatic-NoField P C F D h stk loc* C_0 M_0 *pc ics frs sh*)
  then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
  case (*exec-step-ind-Checkcast P C h stk loc* C_0 M_0 *pc ics frs sh*)
  then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
  case (*exec-step-ind-Checkcast-Error P C h stk loc* C_0 M_0 *pc ics frs sh*)
  then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
  case (*exec-step-ind-Invoke-Null stk n M P h loc* C_0 M_0 *pc ics frs sh*)
  then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
  case (*exec-step-ind-Invoke-NoMethod r stk n C h P M loc* C_0 M_0 *pc ics frs sh*)
  then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
  case (*exec-step-ind-Invoke-Static r stk n C h D b Ts T mxs* mxl_0 *ins xt P M m*

loc C_0 M_0 *pc ics*)
then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
case (*exec-step-ind-Invokestatic-NoMethod D b Ts T mxs* mxl_0 *ins xt P C M n h stk loc* C_0 M_0 *pc ics*)
then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
case (*exec-step-ind-Invokestatic-NonStatic D b Ts T mxs* mxl_0 *ins xt P C M m n h stk loc* C_0 M_0 *pc ics*)
then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
case (*exec-step-ind-Return-Last-Init P h* stk_0 loc_0 C_0 *pc ics sh*)
then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
case (*exec-step-ind-Return-Last* M_0 *P h* stk_0 loc_0 C_0 *pc ics sh*)
then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
case (*exec-step-ind-Return-Init D b Ts T m P* C_0 *h* stk_0 loc_0 *pc ics* stk' loc' C' m' pc' ics')
then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
case (*exec-step-ind-Pop P h stk loc* C_0 M_0 *pc ics frs sh*)
then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
case (*exec-step-ind-IAdd P h stk loc* C_0 M_0 *pc ics frs sh*)
then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
case (*exec-step-ind-IfFalse-False stk i P h loc* C_0 M_0 *pc ics frs sh*)
then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
case (*exec-step-ind-CmpEq P h stk loc* C_0 M_0 *pc ics frs sh*)
then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
case (*exec-step-ind-Goto i P h stk loc* C_0 M_0 *pc ics frs sh*)
then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
case (*exec-step-ind-Throw stk P h loc* C_0 M_0 *pc ics frs sh*)
then show *?case* **by**(*cases ics*, *auto dest*!: *StepI*)
next
case (*exec-step-ind-Init-None-Called sh C Cs P h stk loc* C_0 M_0 *pc ics frs*)
then show *?case* **by**(*auto dest*!: *StepC*)
next
case (*exec-step-ind-Init-Done sh C sfs Cs P h stk loc* C_0 M_0 *pc ics frs*)
then show *?case* **by**(*auto dest*!: *StepC*)
next
case (*exec-step-ind-Init-Processing sh C sfs Cs P h stk loc* C_0 M_0 *pc ics frs*)
then show *?case* **by**(*auto dest*!: *StepC*)
next
case (*exec-step-ind-Init-Error sh C sfs Cs P h stk loc* C_0 M_0 *pc ics frs*)

then show *?case* **by**(*auto dest*!: *StepC*)
next
case (*exec-step-ind-Init-Prepared-Object sh C sfs sh′ Cs P h stk loc* C_0 M_0 *pc ics frs*)
then show *?case* **by**(*auto dest*!: *StepC*)
next
case (*exec-step-ind-Init-Prepared-nObject sh C sfs sh′ D P Cs h stk loc* C_0 M_0 *pc ics frs*)
then show *?case* **by**(*auto dest*!: *StepC*)
next
case (*exec-step-ind-InitThrow C Cs a P h stk loc* C_0 M_0 *pc ics frs sh*)
then show *?case* **by**(*auto dest*!: *StepT*)
next
case (*exec-step-ind-InitThrow-End a P h stk loc* C_0 M_0 *pc ics frs sh*)
then show *?case* **by**(*auto dest*!: *StepT*)
qed
qed

— *exec-step* and *exec-step-ind* reach the same result given equivalent input
lemma *exec-step-ind-equiv*:
exec-step P h stk loc C M pc ics frs sh = (*xp′*, *h′*, *frs′*, *sh′*)
= *exec-step-ind* (*exec-step-input P C M pc ics*) *P h stk loc C M pc ics frs sh* (*xp′*, *h′*, *frs′*, *sh′*)
using *exec-step-imp-exec-step-ind exec-step-ind-imp-exec-step* **by** *auto*

end

12 Instantiating *CollectionBasedRTS* with Jinja JVM

theory *JVMCollectionBasedRTS*
imports ../*Common*/*CollectionBasedRTS JVMCollectionSemantics*
JinjaDCI.BVSpecTypeSafe ../*JinjaSuppl*/*JVMExecStepInductive*

begin

lemma *eq-equiv*[*simp*]: *equiv UNIV* {(*x*, *y*). *x* = *y*}
by(*simp add*: *equiv-def refl-on-def sym-def trans-def*)

12.1 Some *classes-above* lemmas

lemma *start-prog-classes-above-Start*:
classes-above (*start-prog P C M*) *Start* = {*Object*,*Start*}
using *start-prog-Start-super*[*of C M P*] *subcls1-confluent* **by** *auto*

lemma *class-add-classes-above*:
assumes *ns*: ¬ *is-class P C* **and** $\neg P \vdash D \preceq^* C$
shows *classes-above* (*class-add P* (*C*, *cdec*)) *D* = *classes-above P D*
using *assms* **by**(*auto intro*: *class-add-subcls class-add-subcls-rev*)

lemma *class-add-classes-above-xcpts*:
assumes *ns*: $\neg$ *is-class P C*
and *ncp*: $\bigwedge D.\ D \in$ *sys-xcpts* $\Longrightarrow \neg P \vdash D \preceq^* C$
shows *classes-above-xcpts* (*class-add P* (*C*, *cdec*)) = *classes-above-xcpts P*
using *assms class-add-classes-above* **by** *simp*

12.2 JVM next-step lemmas for initialization calling

lemma *JVM-New-next-step*:
assumes *step*: $\sigma' \in$ *JVMsmall P* σ
and *nend*: $\sigma \notin$ *JVMendset*
and *curr*: *curr-instr P* (*hd*(*frames-of* σ)) = *New C*
and *nDone*: $\neg(\exists$ *sfs i*. *sheap* σ *C* = *Some*(*sfs*,*i*) $\wedge$ *i* = *Done*)
and *ics*: *ics-of*(*hd*(*frames-of* σ)) = *No-ics*
shows *ics-of* (*hd*(*frames-of* σ')) = *Calling C* [] $\wedge$ *sheap* σ = *sheap* σ' $\wedge$ $\sigma' \notin$ *JVMendset*
proof –
obtain *xp h frs sh* **where** σ: σ=(*xp*,*h*,*frs*,*sh*) **by**(*cases* σ)
then obtain *f1 frs1* **where** *frs*: *frs*=*f1*#*frs1* **using** *nend* **by**(*cases frs*, *simp-all add*: *JVMendset-def*)
then obtain *stk loc C′ M′ pc ics* **where** *f1*:*f1*=(*stk*,*loc*,*C′*,*M′*,*pc*,*ics*) **by**(*cases f1*)
have *xp*: *xp* = *None* **using** σ *nend* **by**(*simp add*: *JVMendset-def*)
obtain *xp′ h′ frs′ sh′* **where** σ': σ'=(*xp′*,*h′*,*frs′*,*sh′*) **by**(*cases* σ')
have *ics-of* (*hd frs′*) = *Calling C* [] $\wedge$ *sh* = *sh′* $\wedge$ *frs′* $\neq$ [] $\wedge$ *xp′* = *None*
proof(*cases sh C*)
case *None* **then show** *?thesis* **using** σ' *xp f1 frs* σ *assms* **by** *auto*
next
case (*Some a*)
then obtain *sfs i* **where** *a*: *a*=(*sfs*,*i*) **by**(*cases a*)
then have *nDone′*: *i* $\neq$ *Done* **using** *nDone Some f1 frs* σ **by** *simp*
show *?thesis* **using** *a Some* σ' *xp f1 frs* σ *assms* **by**(*auto split*: *init-state.splits*)
qed
then show *?thesis* **using** *ics* σ σ' **by**(*cases frs′*, *auto simp*: *JVMendset-def*)
qed

lemma *JVM-Getstatic-next-step*:
assumes *step*: $\sigma' \in$ *JVMsmall P* σ
and *nend*: $\sigma \notin$ *JVMendset*
and *curr*: *curr-instr P* (*hd*(*frames-of* σ)) = *Getstatic C F D*
and *fC*: *P* $\vdash$ *C has F*,*Static*:*t in D*
and *nDone*: $\neg(\exists$ *sfs i*. *sheap* σ *D* = *Some*(*sfs*,*i*) $\wedge$ *i* = *Done*)
and *ics*: *ics-of*(*hd*(*frames-of* σ)) = *No-ics*
shows *ics-of* (*hd*(*frames-of* σ')) = *Calling D* [] $\wedge$ *sheap* σ = *sheap* σ' $\wedge$ $\sigma' \notin$ *JVMendset*
proof –
obtain *xp h frs sh* **where** σ: σ=(*xp*,*h*,*frs*,*sh*) **by**(*cases* σ)
then obtain *f1 frs1* **where** *frs*: *frs*=*f1*#*frs1* **using** *nend* **by**(*cases frs*, *simp-all add*: *JVMendset-def*)

then obtain *stk loc C′ M′ pc ics* **where** *f1*:*f1*=(*stk*,*loc*,*C′*,*M′*,*pc*,*ics*) **by**(*cases f1*)
have *xp*: *xp* = *None* **using** σ *nend* **by**(*simp add*: *JVMendset-def*)
obtain *xp′ h′ frs′ sh′* **where** σ′: σ′=(*xp′*,*h′*,*frs′*,*sh′*) **by**(*cases* σ′)
have *ex′*: ∃ *t b*. *P* ⊢ *C has F*,*b*:*t in D* **using** *fC* **by** *auto*
have *field*: ∃ *t*. *field P D F* = (*D*,*Static*,*t*)
using *fC field-def2 has-field-idemp has-field-sees* **by** *blast*
have *nCalled′*: ∀ *Cs*. *ics* ≠ *Called Cs* **using** *ics f1 frs* σ **by** *simp*
have *ics-of* (*hd frs′*) = *Calling D* [] ∧ *sh* = *sh′* ∧ *frs′* ≠ [] ∧ *xp′* = *None*
proof(*cases sh D*)
case *None* **then show** *?thesis* **using** *field ex′* σ′ *xp f1 frs* σ *assms* **by** *auto*
next
case (*Some a*)
then obtain *sfs i* **where** *a*: *a*=(*sfs*,*i*) **by**(*cases a*)
show *?thesis* **using** *field ex′ a Some* σ′ *xp f1 frs* σ *assms* **by**(*auto split*: *init-state.splits*)
qed
then show *?thesis* **using** *ics* σ σ′ **by**(*auto simp*: *JVMendset-def*)
qed

lemma *JVM-Putstatic-next-step*:
assumes *step*: σ′ ∈ *JVMsmall P* σ
and *nend*: σ ∉ *JVMendset*
and *curr*: *curr-instr P* (*hd*(*frames-of* σ)) = *Putstatic C F D*
and *fC*: *P* ⊢ *C has F*,*Static*:*t in D*
and *nDone*: ¬(∃ *sfs i*. *sheap* σ *D* = *Some*(*sfs*,*i*) ∧ *i* = *Done*)
and *ics*: *ics-of*(*hd*(*frames-of* σ)) = *No-ics*
shows *ics-of* (*hd*(*frames-of* σ′)) = *Calling D* [] ∧ *sheap* σ = *sheap* σ′ ∧ σ′ ∉ *JVMendset*
proof −
obtain *xp h frs sh* **where** σ: σ=(*xp*,*h*,*frs*,*sh*) **by**(*cases* σ)
then obtain *f1 frs1* **where** *frs*: *frs*=*f1*#*frs1* **using** *nend* **by**(*cases frs*, *simp-all add*: *JVMendset-def*)
then obtain *stk loc C′ M′ pc ics* **where** *f1*:*f1*=(*stk*,*loc*,*C′*,*M′*,*pc*,*ics*) **by**(*cases f1*)
have *xp*: *xp* = *None* **using** σ *nend* **by**(*simp add*: *JVMendset-def*)
obtain *xp′ h′ frs′ sh′* **where** σ′: σ′=(*xp′*,*h′*,*frs′*,*sh′*) **by**(*cases* σ′)
have *ex′*: ∃ *t b*. *P* ⊢ *C has F*,*b*:*t in D* **using** *fC* **by** *auto*
have *field*: *field P D F* = (*D*,*Static*,*t*)
using *fC field-def2 has-field-idemp has-field-sees* **by** *blast*
have *ics′*: *ics-of* (*hd frs′*) = *Calling D* [] ∧ *sh* = *sh′* ∧ *frs′* ≠ [] ∧ *xp′* = *None*
proof(*cases sh D*)
case *None* **then show** *?thesis* **using** *field ex′* σ′ *xp f1 frs* σ *assms* **by** *auto*
next
case (*Some a*)
then obtain *sfs i* **where** *a*: *a*=(*sfs*,*i*) **by**(*cases a*)
show *?thesis* **using** *field ex′ a Some* σ′ *xp f1 frs* σ *assms* **by**(*auto split*: *init-state.splits*)
qed

then show *?thesis* **using** *ics* σ σ' **by**(*auto simp*: *JVMendset-def*)
qed

lemma *JVM-Invokestatic-next-step*:
assumes *step*: $\sigma' \in$ *JVMsmall P* σ
and *nend*: $\sigma \notin$ *JVMendset*
and *curr*: *curr-instr P* (*hd*(*frames-of* σ)) = *Invokestatic C M n*
and *mC*: *P* $\vdash$ *C sees M*,*Static*:*Ts* $\rightarrow$ *T* = *m in D*
and *nDone*: $\neg(\exists$ *sfs i*. *sheap* σ *D* = *Some*(*sfs*,*i*) $\wedge$ *i* = *Done*)
and *ics*: *ics-of*(*hd*(*frames-of* σ)) = *No-ics*
shows *ics-of* (*hd*(*frames-of* σ')) = *Calling D* [] $\wedge$ *sheap* σ = *sheap* σ' $\wedge$ $\sigma' \notin$ *JVMendset*
proof –
obtain *xp h frs sh* **where** σ: σ=(*xp*,*h*,*frs*,*sh*) **by**(*cases* σ)
then obtain *f1 frs1* **where** *frs*: *frs*=*f1*#*frs1* **using** *nend* **by**(*cases frs*, *simp-all add*: *JVMendset-def*)
then obtain *stk loc C' M' pc ics* **where** *f1*:*f1*=(*stk*,*loc*,*C'*,*M'*,*pc*,*ics*) **by**(*cases f1*)
have *xp*: *xp* = *None* **using** σ *nend* **by**(*simp add*: *JVMendset-def*)
obtain *xp' h' frs' sh'* **where** σ': σ'=(*xp'*,*h'*,*frs'*,*sh'*) **by**(*cases* σ')
have *ex'*: $\exists$ *Ts T m D b*. *P* $\vdash$ *C sees M*,*b*:*Ts* $\rightarrow$ *T* = *m in D* **using** *mC* **by** *fastforce*
have *method*: $\exists$ *m*. *method P C M* = (*D*,*Static*,*m*) **using** *mC* **by**(*cases m*, *auto*)
have *ics'*: *ics-of* (*hd frs'*) = *Calling D* [] $\wedge$ *sh* = *sh'* $\wedge$ *frs'* $\neq$ [] $\wedge$ *xp'* = *None*
proof(*cases sh D*)
case *None* **then show** *?thesis* **using** *method ex'* σ' *xp f1 frs* σ *assms* **by** *auto*
next
case (*Some a*)
then obtain *sfs i* **where** *a*: *a*=(*sfs*,*i*) **by**(*cases a*)
then have *nDone'*: *i* $\neq$ *Done* **using** *nDone Some f1 frs* σ **by** *simp*
show *?thesis* **using** *method ex' a Some* σ' *xp f1 frs* σ *assms* **by**(*auto split*: *init-state.splits*)
qed
then show *?thesis* **using** *ics* σ σ' **by**(*auto simp*: *JVMendset-def*)
qed

12.3 Definitions

definition *main* :: *string* **where** *main* = ''*main*''
definition *Test* :: *string* **where** *Test* = ''*Test*''
definition *test-oracle* :: *string* **where** *test-oracle* = ''*oracle*''

type-synonym *jvm-class* = *jvm-method cdecl*
type-synonym *jvm-prog-out* = *jvm-state* $\times$ *cname set*

A deselection algorithm based on classes that have changed from *P1* to *P2*:

primrec *jvm-deselect* :: *jvm-prog* $\Rightarrow$ *jvm-prog-out* $\Rightarrow$ *jvm-prog* $\Rightarrow$ *bool* **where**
jvm-deselect P1 (σ, *cset*) *P2* = (*cset* $\cap$ (*classes-changed P1 P2*) = {})

definition *jvm-progs* :: *jvm-prog set* **where**
jvm-progs ≡ {*P*. *wf-jvm-prog P* ∧ ¬*is-class P Start* ∧ ¬*is-class P Test*
∧ (∀ *b′ Ts′ T′ m′ D′*. *P* ⊢ *Object sees start-m*, *b′* : *Ts′*→*T′* = *m′ in D′*
⟶ *b′* = *Static* ∧ *Ts′* = [] ∧ *T′* = *Void*) }

definition *jvm-tests* :: *jvm-class set* **where**
jvm-tests = {*t*. *fst t* = *Test*
∧ (∀ *P* ∈ *jvm-progs*. *wf-jvm-prog* (*t*#*P*) ∧ (∃ *m*. *t*#*P* ⊢ *Test sees main*,*Static*: []
→ *Void* = *m in Test*)) }

abbreviation *jvm-make-test-prog* :: *jvm-prog* ⇒ *jvm-class* ⇒ *jvm-prog* **where**
jvm-make-test-prog P t ≡ *start-prog* (*t*#*P*) (*fst t*) *main*

declare *jvm-progs-def* [*simp*]
declare *jvm-tests-def* [*simp*]

12.4 Definition lemmas

lemma *jvm-progs-tests-nStart*:
assumes *P*: *P* ∈ *jvm-progs* **and** *t*: *t* ∈ *jvm-tests*
shows ¬*is-class* (*t*#*P*) *Start*
using *assms* **by**(*simp add*: *is-class-def class-def Start-def Test-def*)

lemma *jvm-make-test-prog-classes-above-xcpts*:
assumes *P*: *P* ∈ *jvm-progs* **and** *t*: *t* ∈ *jvm-tests*
shows *classes-above-xcpts* (*jvm-make-test-prog P t*) = *classes-above-xcpts P*
proof −
have *nS*: ¬*is-class* (*t*#*P*) *Start* **by**(*rule jvm-progs-tests-nStart*[*OF P t*])
from *P* **have** *nT*: ¬*is-class P Test* **by** *simp*
from *P t* **have** *wf-syscls* (*t*#*P*) ∧ *wf-syscls P*
by(*simp add*: *wf-jvm-prog-def wf-jvm-prog-phi-def wf-prog-def*)

then have [*simp*]: ⋀*D*. *D* ∈ *sys-xcpts* ⟹ *is-class* (*t*#*P*) *D* ∧ *is-class P D*
by(*cases t*, *auto simp*: *wf-syscls-def is-class-def class-def dest*!: *weak-map-of-SomeI*)
from *wf-nclass-nsub*[*OF - - nS*] *P t* **have** *nspS*: ⋀*D*. *D* ∈ *sys-xcpts* ⟹ ¬(*t*#*P*)
⊢ *D* ⪯* *Start*
by(*auto simp*: *wf-jvm-prog-def wf-jvm-prog-phi-def*)
from *wf-nclass-nsub*[*OF - - nT*] *P* **have** *nspT*: ⋀*D*. *D* ∈ *sys-xcpts* ⟹ ¬*P* ⊢ *D*
⪯* *Test*
by(*auto simp*: *wf-jvm-prog-def wf-jvm-prog-phi-def*)

from *class-add-classes-above-xcpts*[**where** *P*=*t*#*P* **and** *C*=*Start*, *OF nS nspS*]
have *classes-above-xcpts* (*jvm-make-test-prog P t*) = *classes-above-xcpts* (*t*#*P*)
by *simp*
also from *class-add-classes-above-xcpts*[**where** *P*=*P* **and** *C*=*Test*, *OF nT nspT*]
t
have . . . = *classes-above-xcpts P* **by**(*cases t*, *simp*)
finally show *?thesis* **by** *simp*
qed

lemma *jvm-make-test-prog-sees-Test-main*:
assumes *P*: $P \in$ *jvm-progs* **and** *t*: $t \in$ *jvm-tests*
shows $\exists$ *m. jvm-make-test-prog P t* $\vdash$ *Test sees main*, *Static* : $[] \rightarrow$ *Void* = *m in Test*
proof –
  let *?P* = *jvm-make-test-prog P t*
  from *P t* **obtain** *m* **where**
    meth: *t#P* $\vdash$ *Test sees main*,*Static*:[] $\rightarrow$ *Void* = *m in Test* **and**
    nstart: $\neg$ *is-class* (*t # P*) *Start*
   by(*auto simp*: *is-class-def class-def Start-def Test-def*)
  from *class-add-sees-method*[*OF meth nstart*] **show** *?thesis* **by** *fastforce*
qed

12.5 Naive RTS algorithm

12.5.1 Definitions

fun *jvm-naive-out* :: *jvm-prog* $\Rightarrow$ *jvm-class* $\Rightarrow$ *jvm-prog-out set* **where**
jvm-naive-out P t = *JVMNaiveCollectionSemantics.cbig* (*jvm-make-test-prog P t*) (*start-state* (*t#P*))

abbreviation *jvm-naive-collect-start* :: *jvm-prog* $\Rightarrow$ *cname set* **where**
jvm-naive-collect-start P $\equiv$ {}

lemma *jvm-naive-out-xcpts-collected*:
assumes *o1* $\in$ *jvm-naive-out P t*
shows *classes-above-xcpts* (*start-prog* (*t # P*) (*fst t*) *main*) $\subseteq$ *snd o1*
using *assms*
proof –
  obtain σ' *coll'* **where** *o1* = (σ', *coll'*) **and**
    cbig: (σ', *coll'*) $\in$ *JVMNaiveCollectionSemantics.cbig* (*start-prog* (*t#P*) (*fst t*) *main*) (*start-state* (*t#P*))
   using *assms* **by**(*cases o1*, *simp*)
  with *JVMNaiveCollectionSemantics.cbig-stepD*[*OF cbig start-state-nend*]
 show *?thesis* **by**(*auto simp*: *JVMNaiveCollectionSemantics.csmall-def start-state-def*)
qed

12.5.2 Naive algorithm correctness

We start with correctness over *exec-instr*, then all the functions/pieces that are used by naive *csmall* (that is, pieces used by *exec* - such as which frames are used based on *ics* - and all functions used by the collection function). We then prove that *csmall* is existence safe, extend this result to *cbig*, and finally prove the *existence-safe* statement over the locale pieces.

lemma *ncollect-exec-instr*:
assumes *JVMinstr-ncollect P i h stk* $\cap$ *classes-changed P P'* = {}
  and *above-C*: *classes-above P C* $\cap$ *classes-changed P P'* = {}
  and *ics*: *ics* = *Called* [] $\vee$ *ics* = *No-ics*

and *i*: *i* = *instrs-of P C M* ! *pc*
shows *exec-instr i P h stk loc C M pc ics frs sh* = *exec-instr i P′ h stk loc C M pc ics frs sh*
using *assms* **proof**(*cases i*)
  case (*New C1*) **then show** *?thesis* **using** *assms classes-above-blank*[*of C1 P P′*]
    by(*auto split*: *init-state.splits option.splits*)
next
  case (*Getfield F1 C1*) **show** *?thesis*
  proof(*cases hd stk* = *Null*)
    case *True* **then show** *?thesis* **using** *Getfield assms* **by** *simp*
  next
    case *False*
    let *?D* = (*cname-of h* (*the-Addr* (*hd stk*)))
    have *D*: *classes-above P ?D* ∩ *classes-changed P P′* = {}
      using *False Getfield assms* **by** *simp*
    show *?thesis*
    proof(*cases* ∃ *b t*. *P* ⊢ *?D has F1*,*b*:*t in C1*)
      case *True*
      then obtain *b1 t1* **where** *P* ⊢ *?D has F1*,*b1*:*t1 in C1* **by** *auto*
      then have *has*: *P′* ⊢ *?D has F1*,*b1*:*t1 in C1*
       using *Getfield assms classes-above-has-field*[*OF D*] **by** *auto*
      have *P* ⊢ *?D* ⪯* *C1* **using** *has-field-decl-above True* **by** *auto*
    then have *classes-above P C1* ⊆ *classes-above P ?D* **by**(*rule classes-above-subcls-subset*)
      then have *C1*: *classes-above P C1* ∩ *classes-changed P P′* = {} **using** *D* **by** *auto*
      then show *?thesis* **using** *has True Getfield assms classes-above-field*[*of C1 P P′ F1*]
        by(*cases field P′ C1 F1*, *cases the* (*h* (*the-Addr* (*hd stk*))), *auto*)
    next
      case *nex*: *False*
      then have ∄ *b t*. *P′* ⊢ *?D has F1*,*b*:*t in C1*
       using *False Getfield assms*
          classes-above-has-field2[**where** *C*=*?D* **and** *P*=*P* **and** *P′*=*P′* **and** *F*=*F1* **and** *C′*=*C1*]
        by *auto*
       then show *?thesis* **using** *nex Getfield assms classes-above-field*[*of C1 P P′ F1*]
        by(*cases field P′ C1 F1*, *cases the* (*h* (*the-Addr* (*hd stk*))), *auto*)
    qed
  qed
next
  case (*Getstatic C1 F1 D1*)
  then have *C1*: *classes-above P C1* ∩ *classes-changed P P′* = {} **using** *assms* **by** *auto*
  show *?thesis*
  proof(*cases* ∃ *b t*. *P* ⊢ *C1 has F1*,*b*:*t in D1*)
    case *True*
    then obtain *b t* **where** *meth*: *P* ⊢ *C1 has F1*,*b*:*t in D1* **by** *auto*
    then have *P* ⊢ *C1* ⪯* *D1* **by**(*rule has-field-decl-above*)

then have *D1*: *classes-above P D1* ∩ *classes-changed P P′* = {}
using *C1 rtrancl-trans* **by** *fastforce*
have *P′* ⊢ *C1 has F1*,*b*:*t in D1*
using *meth Getstatic assms classes-above-has-field*[*OF C1*] **by** *auto*
then show *?thesis* **using** *True D1 Getstatic assms classes-above-field*[*of D1 P P′ F1*]
by(*cases field P′ D1 F1*, *auto*)
next
case *False*
then have ∄ *b t*. *P′* ⊢ *C1 has F1*,*b*:*t in D1*
using *Getstatic assms*
classes-above-has-field2[**where** *C*=*C1* **and** *P*=*P* **and** *P′*=*P′* **and** *F*=*F1* **and** *C′*=*D1*]
by *auto*
then show *?thesis* **using** *False Getstatic assms*
by(*cases field P′ D1 F1*, *auto*)
qed
next
case (*Putfield F1 C1*) **show** *?thesis*
proof(*cases hd*(*tl stk*) = *Null*)
case *True* **then show** *?thesis* **using** *Putfield assms* **by** *simp*
next
case *False*
let *?D* = (*cname-of h* (*the-Addr* (*hd* (*tl stk*))))
have *D*: *classes-above P ?D* ∩ *classes-changed P P′* = {} **using** *False Putfield assms* **by** *simp*
show *?thesis*
proof(*cases* ∃ *b t*. *P* ⊢ *?D has F1*,*b*:*t in C1*)
case *True*
then obtain *b1 t1* **where** *P* ⊢ *?D has F1*,*b1*:*t1 in C1* **by** *auto*
then have *has*: *P′* ⊢ *?D has F1*,*b1*:*t1 in C1*
using *Putfield assms classes-above-has-field*[*OF D*] **by** *auto*
have *P* ⊢ *?D* ⪯* *C1* **using** *has-field-decl-above True* **by** *auto*
then have *classes-above P C1* ⊆ *classes-above P ?D* **by**(*rule classes-above-subcls-subset*)
then have *C1*: *classes-above P C1* ∩ *classes-changed P P′* = {} **using** *D* **by** *auto*
then show *?thesis* **using** *has True Putfield assms classes-above-field*[*of C1 P P′ F1*]
by(*cases field P′ C1 F1*, *cases the* (*h* (*the-Addr* (*hd* (*tl stk*)))), *auto*)
next
case *nex*: *False*
then have ∄ *b t*. *P′* ⊢ *?D has F1*,*b*:*t in C1*
using *False Putfield assms*
classes-above-has-field2[**where** *C*=*?D* **and** *P*=*P* **and** *P′*=*P′* **and** *F*=*F1* **and** *C′*=*C1*]
by *auto*
then show *?thesis* **using** *nex Putfield assms classes-above-field*[*of C1 P P′ F1*]
by(*cases field P′ C1 F1*, *cases the* (*h* (*the-Addr* (*hd* (*tl stk*)))), *auto*)

qed
qed
next
case (*Putstatic C1 F1 D1*)
then have *C1*: *classes-above P C1* ∩ *classes-changed P P′* = {} **using** *Putstatic assms* **by** *auto*
show *?thesis*
proof(*cases* ∃ *b t*. *P* ⊢ *C1 has F1*,*b*:*t in D1*)
case *True*
then obtain *b t* **where** *meth*: *P* ⊢ *C1 has F1*,*b*:*t in D1* **by** *auto*
then have *P* ⊢ *C1* $\preceq^*$ *D1* **by**(*rule has-field-decl-above*)
then have *D1*: *classes-above P D1* ∩ *classes-changed P P′* = {}
using *C1 rtrancl-trans* **by** *fastforce*
then have *P′* ⊢ *C1 has F1*,*b*:*t in D1*
using *meth Putstatic assms classes-above-has-field*[*OF C1*] **by** *auto*
then show *?thesis* **using** *True D1 Putstatic assms classes-above-field*[*of D1 P P′ F1*]
by(*cases field P′ D1 F1*, *auto*)
next
case *False*
then have ∄ *b t*. *P′* ⊢ *C1 has F1*,*b*:*t in D1*
using *Putstatic assms classes-above-has-field2*[**where** *C*=*C1* **and** *P*=*P* **and** *P′*=*P′* **and** *F*=*F1* **and** *C′*=*D1*]
by *auto*
then show *?thesis* **using** *False Putstatic assms*
by(*cases field P′ D1 F1*, *auto*)
qed
next
case (*Checkcast C1*)
then show *?thesis* **using** *assms*
proof(*cases hd stk* = *Null*)
case *False* **then show** *?thesis*
using *Checkcast assms classes-above-subcls classes-above-subcls2*
by(*simp add*: *cast-ok-def*) *blast*
qed(*simp add*: *cast-ok-def*)
next
case (*Invoke M n*)
let *?C* = *cname-of h* (*the-Addr* (*stk* ! *n*))
show *?thesis*
proof(*cases stk* ! *n* = *Null*)
case *True* **then show** *?thesis* **using** *Invoke assms* **by** *simp*
next
case *False*
then have *above*: *classes-above P ?C* ∩ *classes-changed P P′* = {}
using *Invoke assms* **by** *simp*
obtain *D b Ts T mxs mxl ins xt* **where** *meth*: *method P′ ?C M* = (*D*,*b*,*Ts*,*T*,*mxs*,*mxl*,*ins*,*xt*)
by(*cases method P′ ?C M*, *clarsimp*)
have *meq*: *method P ?C M* = *method P′ ?C M*
using *classes-above-method*[*OF above*] **by** *simp*

then show *?thesis*
proof(*cases* ∃ *Ts T m D b. P* ⊢ *?C sees M*,*b*:*Ts* → *T* = *m in D*)
case *nex*: *False*
then have ¬(∃ *Ts T m D b. P′* ⊢ *?C sees M*,*b*:*Ts* → *T* = *m in D*)
using *classes-above-sees-method2*[*OF above*, *of M*] **by** *clarsimp*
then show *?thesis* **using** *nex False Invoke* **by** *simp*
next
case *True*
then have ∃ *Ts T m D b. P′* ⊢ *?C sees M*,*b*:*Ts* → *T* = *m in D*
by(*fastforce dest*!: *classes-above-sees-method*[*OF above*, *of M*])
then show *?thesis* **using** *meq meth True Invoke* **by** *simp*
qed
qed
next
case (*Invokestatic C1 M n*)
then have *above*: *classes-above P C1* ∩ *classes-changed P P′* = {}
using *assms* **by** *simp*
obtain *D b Ts T mxs mxl ins xt* **where** *meth*: *method P′ C1 M* = (*D*,*b*,*Ts*,*T*,*mxs*,*mxl*,*ins*,*xt*)
by(*cases method P′ C1 M*, *clarsimp*)
have *meq*: *method P C1 M* = *method P′ C1 M*
using *classes-above-method*[*OF above*] **by** *simp*
show *?thesis*
proof(*cases* ∃ *Ts T m D b. P* ⊢ *C1 sees M*,*b*:*Ts* → *T* = *m in D*)
case *False*
then have ¬(∃ *Ts T m D b. P′* ⊢ *C1 sees M*,*b*:*Ts* → *T* = *m in D*)
using *classes-above-sees-method2*[*OF above*, *of M*] **by** *clarsimp*
then show *?thesis* **using** *False Invokestatic* **by** *simp*
next
case *True*
then have ∃ *Ts T m D b. P′* ⊢ *C1 sees M*,*b*:*Ts* → *T* = *m in D*
by(*fastforce dest*!: *classes-above-sees-method*[*OF above*, *of M*])
then show *?thesis* **using** *meq meth True Invokestatic* **by** *simp*
qed
next
case *Return* **then show** *?thesis* **using** *assms classes-above-method*[*OF above-C*]
by(*cases frs*, *auto*)
next
case (*Load x1*) **then show** *?thesis* **using** *assms* **by** *auto*
next
case (*Store x2*) **then show** *?thesis* **using** *assms* **by** *auto*
next
case (*Push x3*) **then show** *?thesis* **using** *assms* **by** *auto*
next
case (*Goto x15*) **then show** *?thesis* **using** *assms* **by** *auto*
next
case (*IfFalse x17*) **then show** *?thesis* **using** *assms* **by** *auto*
qed(*auto*)

— if collected classes unchanged, instruction collection unchanged
lemma *ncollect-JVMinstr-ncollect*:
assumes *JVMinstr-ncollect P i h stk* ∩ *classes-changed P P′* = {}
shows *JVMinstr-ncollect P i h stk* = *JVMinstr-ncollect P′ i h stk*
proof(*cases i*)
  case (*New C1*)
  then show *?thesis* **using** *assms classes-above-set*[*of C1 P P′*] **by** *auto*
next
  case (*Getfield F C1*) **show** *?thesis*
  proof(*cases hd stk* = *Null*)
    case *True* **then show** *?thesis* **using** *Getfield assms* **by** *simp*
  next
    case *False*
    let *?D* = *cname-of h* (*the-Addr* (*hd stk*))
     have *classes-above P ?D* ∩ *classes-changed P P′* = {} **using** *False Getfield assms* **by** *auto*
    then have *classes-above P ?D* = *classes-above P′ ?D*
      using *classes-above-set* **by** *blast*
    then show *?thesis* **using** *assms Getfield* **by** *auto*
  qed
next
  case (*Getstatic C1 P1 D1*)
  then have *classes-above P C1* ∩ *classes-changed P P′* = {} **using** *assms* **by** *auto*
  then have *classes-above P C1* = *classes-above P′ C1*
    using *classes-above-set assms Getstatic* **by** *blast*
  then show *?thesis* **using** *assms Getstatic* **by** *auto*
next
  case (*Putfield F C1*) **show** *?thesis*
  proof(*cases hd*(*tl stk*) = *Null*)
    case *True* **then show** *?thesis* **using** *Putfield assms* **by** *simp*
  next
    case *False*
    let *?D* = *cname-of h* (*the-Addr* (*hd* (*tl stk*)))
     have *classes-above P ?D* ∩ *classes-changed P P′* = {} **using** *False Putfield assms* **by** *auto*
    then have *classes-above P ?D* = *classes-above P′ ?D*
      using *classes-above-set* **by** *blast*
    then show *?thesis* **using** *assms Putfield* **by** *auto*
  qed
next
  case (*Putstatic C1 F D1*)
  then have *classes-above P C1* ∩ *classes-changed P P′* = {} **using** *assms* **by** *auto*
  then have *classes-above P C1* = *classes-above P′ C1*
    using *classes-above-set assms Putstatic* **by** *blast*
  then show *?thesis* **using** *assms Putstatic* **by** *auto*
next
  case (*Checkcast C1*)
  then show *?thesis* **using** *assms*
   classes-above-set[*of cname-of h* (*the-Addr* (*hd stk*)) *P P′*] **by** *auto*

next
  case (*Invoke M n*)
  then show *?thesis* **using** *assms*
   classes-above-set[*of cname-of h* (*the-Addr* (*stk* ! *n*)) *P P′*] **by** *auto*
next
  case (*Invokestatic C1 M n*)
  then show *?thesis* **using** *assms classes-above-set*[*of C1 P P′*] **by** *auto*
next
  case *Return*
  then show *?thesis* **using** *assms classes-above-set*[*of - P P′*] **by** *auto*
next
  case *Throw*
  then show *?thesis* **using** *assms*
   classes-above-set[*of cname-of h* (*the-Addr* (*hd stk*)) *P P′*] **by** *auto*
qed(*auto*)

— if collected classes unchanged, *exec-step* unchanged
lemma *ncollect-exec-step*:
assumes *JVMstep-ncollect P h stk C M pc ics* ∩ *classes-changed P P′* = {}
  and *above-C*: *classes-above P C* ∩ *classes-changed P P′* = {}
shows *exec-step P h stk loc C M pc ics frs sh* = *exec-step P′ h stk loc C M pc ics frs sh*
proof(*cases ics*)
  case *No-ics* **then show** *?thesis*
  using *ncollect-exec-instr assms classes-above-method*[*OF above-C*, *THEN sym*]
    by *simp*
next
  case (*Calling C1 Cs*)
  then have *above-C1*: *classes-above P C1* ∩ *classes-changed P P′* = {}
    using *assms*(*1*) **by** *auto*
  show *?thesis*
  proof(*cases sh C1*)
    case *None*
    then show *?thesis* **using** *Calling assms classes-above-sblank*[*OF above-C1*] **by** *simp*
  next
    case (*Some a*)
    then obtain *sfs i* **where** *sfsi*: *a* = (*sfs*, *i*) **by**(*cases a*)
    then show *?thesis* **using** *Calling Some assms*
    proof(*cases i*)
      case *Prepared* **then show** *?thesis*
       using *above-C1 sfsi Calling Some assms classes-above-method*[*OF above-C1*]
           by(*simp add*: *split-beta classes-above-class classes-changed-class*[**where** *cn*=*C1*])
    next
      case *Error* **then show** *?thesis*
       using *above-C1 sfsi Calling Some assms classes-above-method*[*of C1 P P′*]
           by(*simp add*: *split-beta classes-above-class classes-changed-class*[**where** *cn*=*C1*])

qed(*auto*)
qed
next
case (*Called Cs*) **show** *?thesis*
proof(*cases Cs*)
case *Nil* **then show** *?thesis*
using *ncollect-exec-instr assms classes-above-method*[*OF above-C*, *THEN sym*] *Called*
by *simp*
next
case (*Cons C1 Cs1*)
then have *above-C'*: *classes-above P C1* ∩ *classes-changed P P'* = {} **using** *assms Called* **by** *auto*
show *?thesis* **using** *assms classes-above-method*[*OF above-C'*] *Cons Called* **by** *simp*
qed
next
case (*Throwing Cs a*) **then show** *?thesis* **using** *assms* **by**(*cases Cs*; *simp*)
qed

— if collected classes unchanged, *exec-step* collection unchanged
lemma *ncollect-JVMstep-ncollect*:
assumes *JVMstep-ncollect P h stk C M pc ics* ∩ *classes-changed P P'* = {}
and *above-C*: *classes-above P C* ∩ *classes-changed P P'* = {}
shows *JVMstep-ncollect P h stk C M pc ics* = *JVMstep-ncollect P' h stk C M pc ics*
proof(*cases ics*)
case *No-ics* **then show** *?thesis*
using *assms ncollect-JVMinstr-ncollect classes-above-method*[*OF above-C*]
by *simp*
next
case (*Calling C1 Cs*)
then have *above-C*: *classes-above P C1* ∩ *classes-changed P P'* = {}
using *assms*(*1*) **by** *auto*
let *?C* = *fst*(*method P C1 clinit*)
show *?thesis* **using** *Calling assms classes-above-method*[*OF above-C*]
classes-above-set[*OF above-C*] **by** *auto*
next
case (*Called Cs*) **show** *?thesis*
proof(*cases Cs*)
case *Nil* **then show** *?thesis*
using *assms ncollect-JVMinstr-ncollect classes-above-method*[*OF above-C*] *Called*
by *simp*
next
case (*Cons C1 Cs1*)
then have *above-C1*: *classes-above P C1* ∩ *classes-changed P P'* = {}
and *above-C'*: *classes-above P* (*fst* (*method P C1 clinit*)) ∩ *classes-changed P P'* = {}
using *assms Called* **by** *auto*

show *?thesis* **using** *assms Cons Called classes-above-set*[*OF above-C1*]
classes-above-set[*OF above-C'*] *classes-above-method*[*OF above-C1*]
by *simp*
qed
next
case (*Throwing Cs a*) **then show** *?thesis*
using *assms classes-above-set*[*of cname-of h a P P'*] **by** *simp*
qed

— if collected classes unchanged, *classes-above-frames* unchanged
lemma *ncollect-classes-above-frames*:
JVMexec-ncollect P (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*) $\cap$ *classes-changed P P'* = {}
$\Longrightarrow$ *classes-above-frames P frs* = *classes-above-frames P' frs*
proof(*induct frs*)
case (*Cons f frs'*)
then obtain *stk loc C M pc ics* **where** *f*: *f* = (*stk*,*loc*,*C*,*M*,*pc*,*ics*) **by**(*cases f*)
then have *above-C*: *classes-above P C* $\cap$ *classes-changed P P'* = {} **using** *Cons* **by** *auto*
show *?case* **using** *f Cons classes-above-subcls*[*OF above-C*]
classes-above-subcls2[*OF above-C*] **by** *auto*
qed(*auto*)

— if collected classes unchanged, *classes-above-xcpts* unchanged
lemma *ncollect-classes-above-xcpts*:
assumes *JVMexec-ncollect P* (*None*, *h*, (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*, *sh*) $\cap$ *classes-changed P P'* = {}
shows *classes-above-xcpts P* = *classes-above-xcpts P'*
proof –
have *left*: $\bigwedge x\ x'.\ x' \in$ *sys-xcpts* $\Longrightarrow P \vdash x' \preceq^* x \Longrightarrow \exists\, xa{\in}$*sys-xcpts*. $P' \vdash xa \preceq^* x$
proof –
fix $x\ x'$
assume x': $x' \in$ *sys-xcpts* **and** *above*: $P \vdash x' \preceq^* x$
then show $\exists\, xa{\in}$*sys-xcpts*. $P' \vdash xa \preceq^* x$ **using** *assms classes-above-subcls*[*OF - above*]
by(*rule-tac x=x'* **in** *bexI*) *auto*
qed
have *right*: $\bigwedge x\ x'.\ x' \in$ *sys-xcpts* $\Longrightarrow P' \vdash x' \preceq^* x \Longrightarrow \exists\, xa{\in}$*sys-xcpts*. $P \vdash xa \preceq^* x$
proof –
fix $x\ x'$
assume x': $x' \in$ *sys-xcpts* **and** *above*: $P' \vdash x' \preceq^* x$
then show $\exists\, xa{\in}$*sys-xcpts*. $P \vdash xa \preceq^* x$ **using** *assms classes-above-subcls2*[*OF - above*]
by(*rule-tac x=x'* **in** *bexI*) *auto*
qed
show *?thesis* **using** *left right* **by** *auto*
qed

— if collected classes unchanged, *exec* collection unchanged
lemma *ncollect-JVMexec-ncollect*:
assumes *JVMexec-ncollect P σ ∩ classes-changed P P′ = {}*
shows *JVMexec-ncollect P σ = JVMexec-ncollect P′ σ*
proof –
  obtain *xp h frs sh* **where** *σ*: *σ = (xp,h,frs,sh)* **by**(*cases σ*)
  then show *?thesis* **using** *assms*
  proof(*cases ∃x. xp = Some x ∨ frs = []*)
    case *False*
    then obtain *stk loc C M pc ics frs′* **where** *frs*: *frs = (stk,loc,C,M,pc,ics)#frs′*
      by(*cases frs*, *auto*)
    have *step*: *JVMstep-ncollect P h stk C M pc ics ∩ classes-changed P P′ = {}*
      using *False σ frs assms* **by**(*cases ics*, *auto simp*: *JVMNaiveCollectionSemantics.csmall-def*)
    have *above-C*: *classes-above P C ∩ classes-changed P P′ = {}*
    using *False σ frs assms* **by**(*auto simp*: *JVMNaiveCollectionSemantics.csmall-def*)
    have *frames*: *classes-above-frames P frs′ = classes-above-frames P′ frs′*
     using *ncollect-classes-above-frames frs σ False assms* **by** *simp*
    have *xcpts*: *classes-above-xcpts P = classes-above-xcpts P′*
     using *ncollect-classes-above-xcpts frs σ False assms* **by** *simp*
    show *?thesis* **using** *False xcpts frames frs σ ncollect-JVMstep-ncollect*[*OF step above-C*]
        classes-above-subcls[*OF above-C*] *classes-above-subcls2*[*OF above-C*]
      by *auto*
  qed(*auto*)
qed

— if collected classes unchanged, classes above an exception returned by *exec-instr* unchanged
lemma *ncollect-exec-instr-xcpts*:
assumes *collect*: *JVMinstr-ncollect P i h stk ∩ classes-changed P P′ = {}*
  and *xpcollect*: *classes-above-xcpts P ∩ classes-changed P P′ = {}*
  and *prealloc*: *preallocated h*
  and *σ′*: *σ′ = exec-instr i P h stk loc C M pc ics′ frs sh*
  and *xp*: *fst σ′ = Some a*
  and *i*: *i = instrs-of P C M ! pc*
shows *classes-above P (cname-of h a) ∩ classes-changed P P′ = {}*
using *assms exec-instr-xcpts*[*OF σ′ xp*]
proof(*cases i*)
  case *Throw* **then show** *?thesis* **using** *assms* **by**(*cases hd stk*, *fastforce+*)
qed(*fastforce+*)

— if collected classes unchanged, classes above an exception returned by *exec-step* unchanged
lemma *ncollect-exec-step-xcpts*:
assumes *collect*: *JVMstep-ncollect P h stk C M pc ics ∩ classes-changed P P′ = {}*
  and *xpcollect*: *classes-above-xcpts P ∩ classes-changed P P′ = {}*

and *prealloc*: *preallocated h*
and σ': σ' = *exec-step P h stk loc C M pc ics frs sh*
and *xp*: *fst* σ' = *Some a*
shows *classes-above P* (*cname-of h a*) ∩ *classes-changed P P′* = {}
proof(*cases ics*)
case *No-ics* **then show** *?thesis* **using** *assms ncollect-exec-instr-xcpts* **by** *simp*
next
case (*Calling x21 x22*)
then show *?thesis* **using** *assms* **by**(*clarsimp split*: *option.splits init-state.splits if-split-asm*)
next
case (*Called Cs*) **then show** *?thesis* **using** *assms ncollect-exec-instr-xcpts* **by**(*cases Cs*; *simp*)
next
case (*Throwing Cs a*) **then show** *?thesis* **using** *assms* **by**(*cases Cs*; *simp*)
qed

— if collected classes unchanged, if *csmall* returned a result under *P*, *P′* returns the same
lemma *ncollect-JVMsmall*:
assumes *collect*: (σ', *cset*) ∈ *JVMNaiveCollectionSemantics.csmall P* σ
and *intersect*: *cset* ∩ *classes-changed P P′* = {}
and *prealloc*: *preallocated* (*fst*(*snd* σ))
shows (σ', *cset*) ∈ *JVMNaiveCollectionSemantics.csmall P′* σ
proof −
obtain *xp h frs sh* **where** σ: σ = (*xp*,*h*,*frs*,*sh*) **by**(*cases* σ)
then have *prealloc′*: *preallocated h* **using** *prealloc* **by** *simp*
show *?thesis* **using** σ *assms*
proof(*cases* ∃ *x*. *xp* = *Some x* ∨ *frs* = [])
case *False*
then obtain *stk loc C M pc ics frs′* **where** *frs*: *frs* = (*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs′*
by(*cases frs*, *auto*)
have *step*: *JVMstep-ncollect P h stk C M pc ics* ∩ *classes-changed P P′* = {}
using *False* σ *frs assms* **by**(*cases ics*, *auto simp*: *JVMNaiveCollectionSemantics.csmall-def*)
have *above-C*: *classes-above P C* ∩ *classes-changed P P′* = {}
using *False* σ *frs assms* **by**(*auto simp*: *JVMNaiveCollectionSemantics.csmall-def*)
obtain *xp1 h1 frs1 sh1* **where** *exec*: *exec-step P′ h stk loc C M pc ics frs′ sh* = (*xp1*,*h1*,*frs1*,*sh1*)
by(*cases exec-step P′ h stk loc C M pc ics frs′ sh*)
have *collect*: *JVMexec-ncollect P* σ = *JVMexec-ncollect P′* σ
using *assms ncollect-JVMexec-ncollect* **by**(*simp add*: *JVMNaiveCollectionSemantics.csmall-def*)
have *exec-instr*: *exec-step P h stk loc C M pc ics frs′ sh*
= *exec-step P′ h stk loc C M pc ics frs′ sh*
using *ncollect-exec-step*[*OF step above-C*] σ *frs False* **by** *simp*
show *?thesis*
proof(*cases xp1*)
case *None* **then show** *?thesis*

using *None* σ *frs step False assms ncollect-exec-step*[*OF step above-C*] *collect exec*
by(*auto simp*: *JVMNaiveCollectionSemantics.csmall-def*)
next
case (*Some a*)
then show *?thesis* **using** σ *assms*
proof(*cases xp*)
case *None*
have *frames*: *classes-above-frames P* (*frames-of* σ) $\cap$ *classes-changed P P*$'$ = {}
using *None Some frs* σ *assms* **by**(*auto simp*: *JVMNaiveCollectionSemantics.csmall-def*)
have *frsi*: *classes-above-frames P frs* $\cap$ *classes-changed P P*$'$ = {} **using** σ *frames* **by** *simp*
have *xpcollect*: *classes-above-xcpts P* $\cap$ *classes-changed P P*$'$ = {}
using *None Some frs* σ *assms* **by**(*auto simp*: *JVMNaiveCollectionSemantics.csmall-def*)
have *xp*: *classes-above P* (*cname-of h a*) $\cap$ *classes-changed P P*$'$ = {}
using *ncollect-exec-step-xcpts*[*OF step xpcollect prealloc*$'$,
where σ' = (*xp1*,*h1*,*frs1*,*sh1*) **and** *frs*=*frs*$'$ **and** *loc*=*loc* **and** *a*=*a* **and** *sh*=*sh*]
exec exec-instr Some assms **by** *auto*
show *?thesis* **using** *Some exec* σ *frs False assms exec-instr collect*
classes-above-find-handler[**where** *h*=*h* **and** *sh*=*sh*, *OF xp frsi*]
by(*auto simp*: *JVMNaiveCollectionSemantics.csmall-def*)
qed(*auto simp*: *JVMNaiveCollectionSemantics.csmall-def*)
qed
qed(*auto simp*: *JVMNaiveCollectionSemantics.csmall-def*)
qed

— if collected classes unchanged, if *cbig* returned a result under *P*, *P*$'$ returns the same
lemma *ncollect-JVMbig*:
assumes *collect*: (σ', *cset*) $\in$ *JVMNaiveCollectionSemantics.cbig P* σ
and *intersect*: *cset* $\cap$ *classes-changed P P*$'$ = {}
and *prealloc*: *preallocated* (*fst*(*snd* σ))
shows (σ', *cset*) $\in$ *JVMNaiveCollectionSemantics.cbig P*$'$ σ
using *JVMNaiveCollectionSemantics.csmall-to-cbig-prop2*[**where** *R*=λ*P P*$'$ *cset*. *cset* $\cap$ *classes-changed P P*$'$ = {}
and *Q*=$\lambda\sigma$. *preallocated* (*fst*(*snd* σ)) **and** *P*=*P* **and** *P*$'$=*P*$'$ **and** σ=σ **and** σ'=σ' **and** *coll*=*cset*]
ncollect-JVMsmall JVMsmall-prealloc-pres assms **by** *auto*

— and finally, RTS algorithm based on *ncollect* is existence safe
theorem *jvm-naive-existence-safe*:
assumes *p*: *P* $\in$ *jvm-progs* **and** *P*$'$ $\in$ *jvm-progs* **and** *t*: *t* $\in$ *jvm-tests*
and *out*: *o1* $\in$ *jvm-naive-out P t* **and** *jvm-deselect P o1 P*$'$
shows $\exists$ *o2* $\in$ *jvm-naive-out P*$'$ *t*. *o1* = *o2*
using *assms*

proof –
let *?P* = *start-prog* (*t*#*P*) (*fst t*) *main*
let *?P′* = *start-prog* (*t*#*P′*) (*fst t*) *main*
obtain *wf-md* **where** *wf′*: *wf-prog wf-md* (*t*#*P*) **using** *p t*
by(*auto simp*: *wf-jvm-prog-def wf-jvm-prog-phi-def*)
have *ns*: ¬*is-class* (*t*#*P*) *Start* **using** *p t*
by(*clarsimp simp*: *is-class-def class-def Start-def Test-def*)
obtain *σ1 coll1* **where** *o1*: *o1* = (*σ1*, *coll1*) **by**(*cases o1*)
then have *cbig*: (*σ1*, *coll1*) ∈ *JVMNaiveCollectionSemantics.cbig ?P* (*start-state* (*t* # *P*))
using *assms* **by** *simp*
have *coll1* ∩ *classes-changed P P′* = {} **using** *cbig o1 assms* **by** *auto*
then have *changed*: *coll1* ∩ *classes-changed* (*t*#*P*) (*t*#*P′*) = {} **by**(*rule classes-changed-int-Cons*)
then have *changed′*: *coll1* ∩ *classes-changed ?P ?P′* = {} **by**(*rule classes-changed-int-Cons*)
have *classes-above-xcpts ?P* = *classes-above-xcpts* (*t*#*P*)
using *class-add-classes-above*[*OF ns wf-sys-xcpt-nsub-Start*[*OF wf′ ns*]] **by** *simp*
then have *classes-above-xcpts* (*t*#*P*) ∩ *classes-changed* (*t*#*P*) (*t*#*P′*) = {}
using *jvm-naive-out-xcpts-collected*[*OF out*] *o1 changed* **by** *auto*
then have *ss-eq*: *start-state* (*t*#*P*) = *start-state* (*t*#*P′*)
using *classes-above-start-state* **by** *simp*
show *?thesis* **using** *ncollect-JVMbig*[*OF cbig changed′*]
preallocated-start-state changed′ ss-eq o1 assms **by** *auto*
qed

— ...thus *JVMNaiveCollection* is an instance of *CollectionBasedRTS*
interpretation *JVMNaiveCollectionRTS* :
CollectionBasedRTS (=) *jvm-deselect jvm-progs jvm-tests*
JVMendset JVMcombine JVMcollect-id JVMsmall JVMNaiveCollect jvm-naive-out
jvm-make-test-prog jvm-naive-collect-start
by *unfold-locales* (*rule jvm-naive-existence-safe*, *auto simp*: *start-state-def*)

12.6 Smarter RTS algorithm

12.6.1 Definitions and helper lemmas

fun *jvm-smart-out* :: *jvm-prog* ⇒ *jvm-class* ⇒ *jvm-prog-out set* **where**
jvm-smart-out P t
= {(*σ′*,*coll′*). ∃ *coll*. (*σ′*,*coll*) ∈ *JVMSmartCollectionSemantics.cbig*
(*jvm-make-test-prog P t*) (*start-state* (*t*#*P*))
∧ *coll′* = *coll* ∪ *classes-above-xcpts P* ∪ {*Object*,*Start*}}

abbreviation *jvm-smart-collect-start* :: *jvm-prog* ⇒ *cname set* **where**
jvm-smart-collect-start P ≡ *classes-above-xcpts P* ∪ {*Object*,*Start*}

lemma *jvm-naive-iff-smart*:
(∃ $cset_n$. (*σ′*,$cset_n$) ∈ *jvm-naive-out P t*) ⟷ (∃ $cset_s$. (*σ′*,$cset_s$) ∈ *jvm-smart-out P t*)
by(*auto simp*: *JVMNaiveCollectionSemantics.cbig-big-equiv*
JVMSmartCollectionSemantics.cbig-big-equiv)

lemma *jvm-smart-out-classes-above-xcpts*:
assumes *s*: $(\sigma', cset_s) \in$ *jvm-smart-out P t* **and** *P*: $P \in$ *jvm-progs* **and** *t*: $t \in$ *jvm-tests*
shows *classes-above-xcpts* (*jvm-make-test-prog P t*) $\subseteq cset_s$
using *jvm-make-test-prog-classes-above-xcpts*[*OF P t*] *s* **by** *clarsimp*

lemma *jvm-smart-collect-start-make-test-prog*:
$[\![$ $P \in$ *jvm-progs*; $t \in$ *jvm-tests* $]\!]$
$\Longrightarrow$ *jvm-smart-collect-start* (*jvm-make-test-prog P t*) = *jvm-smart-collect-start P*
using *jvm-make-test-prog-classes-above-xcpts* **by** *simp*

lemma *jvm-smart-out-classes-above-start-heap*:
assumes *s*: $(\sigma', cset_s) \in$ *jvm-smart-out P t* **and** *h*: *start-heap* (*t#P*) *a* = *Some*(*C*,*fs*)
and *P*: $P \in$ *jvm-progs* **and** *t*: $t \in$ *jvm-tests*
shows *classes-above* (*jvm-make-test-prog P t*) $C \subseteq cset_s$
using *start-heap-classes*[*OF h*] *jvm-smart-out-classes-above-xcpts*[*OF s P t*] **by** *auto*

lemma *jvm-smart-out-classes-above-start-sheap*:
assumes $(\sigma', cset_s) \in$ *jvm-smart-out P t* **and** *start-sheap C* = *Some*(*sfs*,*i*)
shows *classes-above* (*jvm-make-test-prog P t*) $C \subseteq cset_s$
using *assms start-prog-classes-above-Start* **by**(*clarsimp split*: *if-split-asm*)

lemma *jvm-smart-out-classes-above-frames*:
$(\sigma', cset_s) \in$ *jvm-smart-out P t*
$\Longrightarrow$ *classes-above-frames* (*jvm-make-test-prog P t*) (*frames-of* (*start-state* (*t#P*)))
$\subseteq cset_s$
using *start-prog-classes-above-Start* **by**(*clarsimp split*: *if-split-asm simp*: *start-state-def*)

12.6.2 Additional well-formedness conditions

fun *coll-init-class* :: *'m prog* $\Rightarrow$ *instr* $\Rightarrow$ *cname option* **where**
coll-init-class P (*New C*) = *Some C* |
coll-init-class P (*Getstatic C F D*) = (*if* $\exists$ *t*. $P \vdash$ *C has F*,*Static*:*t in D*
then Some D else None) |
coll-init-class P (*Putstatic C F D*) = (*if* $\exists$ *t*. $P \vdash$ *C has F*,*Static*:*t in D*
then Some D else None) |
coll-init-class P (*Invokestatic C M n*) = *seeing-class P C M* |
coll-init-class - - = *None*

— checks whether the given value is a pointer; if it's an address, checks whether it points to an object in the given heap
fun *is-ptr* :: *heap* $\Rightarrow$ *val* $\Rightarrow$ *bool* **where**
is-ptr h Null = *True* |
is-ptr h (*Addr a*) = ($\exists$ *Cfs*. *h a* = *Some Cfs*) |
is-ptr h - = *False*

lemma *is-ptrD*: *is-ptr h v* $\Longrightarrow$ *v* = *Null* $\vee$ ($\exists$ *a*. *v* = *Addr a* $\wedge$ ($\exists$ *Cfs*. *h a* = *Some Cfs*))
by(*cases v*, *auto*)

— shorthand for: given stack has entries required by given instr, including pointer where necessary
fun *stack-safe* :: *instr* $\Rightarrow$ *heap* $\Rightarrow$ *val list* $\Rightarrow$ *bool* **where**
stack-safe (*Getfield F C*) *h stk* = (*length stk* > *0* $\wedge$ *is-ptr h* (*hd stk*)) |
stack-safe (*Putfield F C*) *h stk* = (*length stk* > *1* $\wedge$ *is-ptr h* (*hd* (*tl stk*))) |
stack-safe (*Checkcast C*) *h stk* = (*length stk* > *0* $\wedge$ *is-ptr h* (*hd stk*)) |
stack-safe (*Invoke M n*) *h stk* = (*length stk* > *n* $\wedge$ *is-ptr h* (*stk* ! *n*)) |
stack-safe JVMInstructions.Throw h stk = (*length stk* > *0* $\wedge$ *is-ptr h* (*hd stk*)) |
stack-safe i h stk = *True*

lemma *well-formed-stack-safe*:
assumes *wtp*: *wf-jvm-prog*$_\Phi$ *P* **and** *correct*: *P*,Φ $\vdash$ (*xp*,*h*,(*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*,*sh*)$\surd$
shows *stack-safe* (*instrs-of P C M* ! *pc*) *h stk*
proof −
  from *correct* **obtain** *b Ts T mxs mxl*$_0$ *ins xt* **where**
    mC: *P* $\vdash$ *C sees M*,*b*:*Ts*→*T*=(*mxs*,*mxl*$_0$,*ins*,*xt*) *in C* **and**
    pc: *pc* < *length ins* **by** *clarsimp*
  from *sees-wf-mdecl*[*OF* - *mC*] *wtp* **have** *wt-method P C b Ts T mxs mxl*$_0$ *ins xt* (Φ *C M*)
    by(*auto simp*: *wf-jvm-prog-phi-def wf-mdecl-def*)
  with *pc* **have** *wt*: *P*,*T*,*mxs*,*length ins*,*xt* $\vdash$ *ins* ! *pc*,*pc* :: Φ *C M* **by**(*simp add*: *wt-method-def*)
  from *mC correct* **obtain** *ST LT* **where**
    Φ: Φ *C M* ! *pc* = *Some* (*ST*,*LT*) **and**
    stk: *P*,*h* $\vdash$ *stk* [:≤] *ST* **by** *fastforce*
  show *?thesis*
  proof(*cases instrs-of P C M* ! *pc*)
    case (*Getfield F D*)
    with *mC* Φ *wt stk* **obtain** *oT ST'* **where**
      oT: *P* $\vdash$ *oT* ≤ *Class D* **and**
      ST: *ST* = *oT* # *ST'* **by** *fastforce*
    with *stk* **obtain** *ref stk'* **where**
      stk': *stk* = *ref*#*stk'* **and**
      ref: *P*,*h* $\vdash$ *ref* :≤ *oT* **by** *auto*
    with *ref oT* **have** *ref* = *Null* $\vee$ (*ref* ≠ *Null* $\wedge$ *P*,*h* $\vdash$ *ref* :≤ *Class D*) **by** *auto*
    with *Getfield mC* **have** *is-ptr h ref* **by**(*fastforce dest*: *non-npD*)
    with *stk' Getfield* **show** *?thesis* **by** *auto*
  next
    case (*Putfield F D*)
    with *mC* Φ *wt stk* **obtain** *vT oT ST'* **where**
      oT: *P* $\vdash$ *oT* ≤ *Class D* **and**
      ST: *ST* = *vT* # *oT* # *ST'* **by** *fastforce*
    with *stk* **obtain** *v ref stk'* **where**
      stk': *stk* = *v*#*ref*#*stk'* **and**
      ref: *P*,*h* $\vdash$ *ref* :≤ *oT* **by** *auto*

with *ref oT* **have** $ref = Null \lor (ref \neq Null \land P,h \vdash ref :\leq Class\ D)$ **by** *auto*
with *Putfield mC* **have** *is-ptr h ref* **by**(*fastforce dest*: *non-npD*)
with *stk′ Putfield* **show** *?thesis* **by** *auto*
next
case (*Checkcast D*)
with *mC Φ wt stk* **obtain** *oT ST′* **where**
oT: *is-refT oT* **and**
ST: $ST = oT \# ST'$ **by** *fastforce*
with *stk* **obtain** *ref stk′* **where**
stk′: $stk = ref\#stk'$ **and**
ref: $P,h \vdash ref :\leq oT$ **by** *auto*
with *ref oT* **have** $ref = Null \lor (ref \neq Null \land (\exists D'.\ P,h \vdash ref :\leq Class\ D'))$
by(*auto simp*: *is-refT-def*)
with *Checkcast mC* **have** *is-ptr h ref* **by**(*fastforce dest*: *non-npD*)
with *stk′ Checkcast* **show** *?thesis* **by** *auto*
next
case (*Invoke M1 n*)
with *mC Φ wt stk* **have**
ST: $n < size\ ST$ **and**
oT: $ST!n = NT \lor (\exists D.\ ST!n = Class\ D)$ **by** *auto*
with *stk* **have** *stk′*: $n < size\ stk$ **by** (*auto simp*: *list-all2-lengthD*)
with *stk ST oT list-all2-nthD2*
have $stk!n = Null \lor (stk!n \neq Null \land (\exists D.\ P,h \vdash stk!n :\leq Class\ D))$ **by** *fastforce*
with *Invoke mC* **have** *is-ptr h* (*stk*!*n*) **by**(*fastforce dest*: *non-npD*)
with *stk′ Invoke* **show** *?thesis* **by** *auto*
next
case *Throw*
with *mC Φ wt stk* **obtain** *oT ST′* **where**
oT: *is-refT oT* **and**
ST: $ST = oT \# ST'$ **by** *fastforce*
with *stk* **obtain** *ref stk′* **where**
stk′: $stk = ref\#stk'$ **and**
ref: $P,h \vdash ref :\leq oT$ **by** *auto*
with *ref oT* **have** $ref = Null \lor (ref \neq Null \land (\exists D'.\ P,h \vdash ref :\leq Class\ D'))$
by(*auto simp*: *is-refT-def*)
with *Throw mC* **have** *is-ptr h ref* **by**(*fastforce dest*: *non-npD*)
with *stk′ Throw* **show** *?thesis* **by** *auto*
qed(*simp-all*)
qed

12.6.3 Proving naive ⊆ smart

We prove that, given well-formedness of the program and state, and ”promises” about what has or will be collected in previous or future steps, *jvm-smart* collects everything *jvm-naive* does. We prove that promises about previously-collected classes (”backward promises”) are maintained by execution, and promises about to-be-collected classes (”forward promises”) are met by the end of execution. We then show that the required initial conditions (well-formedness and backward promises) are met by the defined start states, and thus that a run test will collect at least those classes collected by the naive algorithm.

If backward promises have been kept, a single step preserves this property; i.e., any classes that have been added to this set (new heap objects, newly prepared sheap classes, new frames) are collected by the smart collection algorithm in that step or by forward promises:

lemma *backward-coll-promises-kept*:
assumes
— well-formedness
 wtp: $\textit{wf-jvm-prog}_{\Phi}\ P$
 and *correct*: $P,\Phi \vdash (xp,h,frs,sh)\surd$
— defs
 and *f'*: $\textit{hd frs} = (stk,loc,C',M',pc,ics)$
— backward promises - will be collected prior
 and *heap*: $\bigwedge C\ fs.\ \exists a.\ h\ a = Some(C,fs) \Longrightarrow \textit{classes-above}\ P\ C \subseteq cset$
 and *sheap*: $\bigwedge C\ sfs\ i.\ sh\ C = Some(sfs,i) \Longrightarrow \textit{classes-above}\ P\ C \subseteq cset$
 and *xcpts*: $\textit{classes-above-xcpts}\ P \subseteq cset$
 and *frames*: $\textit{classes-above-frames}\ P\ frs \subseteq cset$
— forward promises - will be collected after if not already
 and *init-class-prom*: $\bigwedge C.\ ics = Called\ [] \lor ics = \textit{No-ics}$
 $\Longrightarrow \textit{coll-init-class}\ P\ (\textit{instrs-of}\ P\ C'\ M'\ !\ pc) = Some\ C \Longrightarrow \textit{classes-above}\ P\ C \subseteq cset$
 and *Calling-prom*: $\bigwedge C'\ Cs'.\ ics = Calling\ C'\ Cs' \Longrightarrow \textit{classes-above}\ P\ C' \subseteq cset$
— collection and step
 and *smart*: $\textit{JVMexec-scollect}\ P\ (xp,h,frs,sh) \subseteq cset$
 and *small*: $(xp',h',frs',sh') \in \textit{JVMsmall}\ P\ (xp,h,frs,sh)$
shows $(h'\ a = Some(C,fs) \longrightarrow \textit{classes-above}\ P\ C \subseteq cset)$
 $\land (sh'\ C = Some(sfs',i') \longrightarrow \textit{classes-above}\ P\ C \subseteq cset)$
 $\land (\textit{classes-above-frames}\ P\ frs' \subseteq cset)$
using *assms*
proof(*cases frs*)
 case (*Cons f1 frs1*)

 then have *cr'*: $P,\Phi \vdash (xp,h,(stk,loc,C',M',pc,ics)\#frs1,sh)\surd$ **using** *correct f'* **by** *simp*
 let $?i = \textit{instrs-of}\ P\ C'\ M'\ !\ pc$
 from *well-formed-stack-safe*[*OF wtp cr'*] *correct-state-Throwing-ex*[*OF cr'*] **obtain**
 stack-safe: *stack-safe ?i h stk* **and**
 Throwing-ex: $\bigwedge Cs\ a.\ ics = Throwing\ Cs\ a \Longrightarrow \exists obj.\ h\ a = Some\ obj$ **by** *simp*
 have *confc*: *conf-clinit P sh frs* **using** *correct Cons* **by** *simp*
 have *Called-prom*: $\bigwedge C'\ Cs'.\ ics = Called\ (C'\#Cs')$
 $\Longrightarrow \textit{classes-above}\ P\ C' \subseteq cset \land \textit{classes-above}\ P\ (fst(method\ P\ C'\ clinit)) \subseteq cset$
 proof −
 fix $C'\ Cs'$ **assume** [*simp*]: $ics = Called\ (C'\#Cs')$
 then have $C' \in set(\textit{clinit-classes}\ frs)$ **using** *f' Cons* **by** *simp*

then obtain *sfs* **where** *shC′*: *sh C′ = Some*(*sfs*, *Processing*) **and** *is-class P C′*
using *confc* **by**(*auto simp*: *conf-clinit-def*)
then have *C′eq*: *C′ = fst*(*method P C′ clinit*) **using** *wf-sees-clinit wtp*
by(*fastforce simp*: *is-class-def wf-jvm-prog-phi-def*)
then show *classes-above P C′* ⊆ *cset* ∧ *classes-above P* (*fst*(*method P C′ clinit*)) ⊆ *cset*
using *sheap shC′* **by** *auto*
qed
have *Called-prom2*: ⋀*Cs. ics = Called Cs* ⟹ ∃ *C1 sobj. Called-context P C1 ?i* ∧ *sh C1 = Some sobj*
using *cr′* **by**(*auto simp*: *conf-f-def2*)
have *Throwing-prom*: ⋀*C′ Cs a. ics = Throwing* (*C′#Cs*) *a* ⟹ ∃ *sfs. sh C′ = Some*(*sfs*, *Processing*)
proof −
fix *C′ Cs a* **assume** [*simp*]: *ics = Throwing* (*C′#Cs*) *a*
then have *C′* ∈ *set*(*clinit-classes frs*) **using** *f′ Cons* **by** *simp*
then show ∃ *sfs. sh C′ = Some*(*sfs*, *Processing*) **using** *confc* **by**(*clarsimp simp*: *conf-clinit-def*)
qed

show *?thesis* **using** *Cons assms*
proof(*cases xp*)
case *None*
then have *exec*: *exec* (*P*, *None*, *h*, (*stk*,*loc*,*C′*,*M′*,*pc*,*ics*)#*frs1*, *sh*) = *Some* (*xp′*,*h′*,*frs′*,*sh′*)
using *small f′ Cons* **by** *auto*
obtain *si* **where** *si*: *exec-step-input P C′ M′ pc ics = si* **by** *simp*
obtain xp_0 h_0 frs_0 sh_0 **where**
exec-step: *exec-step P h stk loc C′ M′ pc ics frs1 sh* = (xp_0, h_0, frs_0, sh_0)
by(*cases exec-step P h stk loc C′ M′ pc ics frs1 sh*)
then have *ind*: *exec-step-ind si P h stk loc C′ M′ pc ics frs1 sh*
(xp_0, h_0, frs_0, sh_0) **using** *exec-step-ind-equiv si* **by** *auto*
then show *?thesis* **using** *heap sheap frames exec exec-step f′ Cons*
si init-class-prom stack-safe Calling-prom Called-prom Called-prom2 Throwing-prom
proof(*induct rule*: *exec-step-ind.induct*)
case *exec-step-ind-Load* **show** *?case* **using** *exec-step-ind-Load.prems*(*1*−*7*) **by** *auto*
next
case *exec-step-ind-Store* **show** *?case* **using** *exec-step-ind-Store.prems*(*1*−*7*) **by** *auto*
next
case *exec-step-ind-Push* **show** *?case* **using** *exec-step-ind-Push.prems*(*1*−*7*) **by** *auto*
next
case *exec-step-ind-NewOOM-Called* **show** *?case* **using** *exec-step-ind-NewOOM-Called.prems*(*1*−*7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast*+
next
case *exec-step-ind-NewOOM-Done* **show** *?case* **using** *exec-step-ind-NewOOM-Done.prems*(*1*−*7*)

by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast+*
next
case *exec-step-ind-New-Called* **show** *?case*
using *exec-step-ind-New-Called.hyps exec-step-ind-New-Called.prems*(*1−9*)
by(*auto split*: *if-split-asm simp*: *blank-def dest*!: *exec-step-input-StepID*) *blast+*
next
case *exec-step-ind-New-Done* **show** *?case*
using *exec-step-ind-New-Done.hyps exec-step-ind-New-Done.prems*(*1−9*)
by(*auto split*: *if-split-asm simp*: *blank-def dest*!: *exec-step-input-StepID*) *blast+*
next
case *exec-step-ind-New-Init* **show** *?case*
using *exec-step-ind-New-Init.prems*(*1−7*) **by** *auto*
next
case *exec-step-ind-Getfield-Null* **show** *?case* **using** *exec-step-ind-Getfield-Null.prems*(*1−7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast+*
next
case *exec-step-ind-Getfield-NoField* **show** *?case*
using *exec-step-ind-Getfield-NoField.prems*(*1−7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast+*
next
case *exec-step-ind-Getfield-Static* **show** *?case*
using *exec-step-ind-Getfield-Static.prems*(*1−7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast+*
next
case *exec-step-ind-Getfield* **show** *?case*
using *exec-step-ind-Getfield.prems*(*1−7*) **by** *auto*
next
case *exec-step-ind-Getstatic-NoField* **show** *?case*
using *exec-step-ind-Getstatic-NoField.prems*(*1−7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast+*
next
case *exec-step-ind-Getstatic-NonStatic* **show** *?case*
using *exec-step-ind-Getstatic-NonStatic.prems*(*1−7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast+*
next
case *exec-step-ind-Getstatic-Called* **show** *?case*
using *exec-step-ind-Getstatic-Called.prems*(*1−7*) **by** *auto*
next
case *exec-step-ind-Getstatic-Done* **show** *?case*
using *exec-step-ind-Getstatic-Done.prems*(*1−7*) **by** *auto*
next
case *exec-step-ind-Getstatic-Init* **show** *?case*
using *exec-step-ind-Getstatic-Init.prems*(*1−7*) **by** *auto*
next
case *exec-step-ind-Putfield-Null* **show** *?case*
using *exec-step-ind-Putfield-Null.prems*(*1−7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast+*

next
case *exec-step-ind-Putfield-NoField* **show** *?case*
using *exec-step-ind-Putfield-NoField.prems*(*1*− *7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast*+
next
case *exec-step-ind-Putfield-Static* **show** *?case*
using *exec-step-ind-Putfield-Static.prems*(*1*− *7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast*+
next
case (*exec-step-ind-Putfield v stk r a D fs h D′ b t P C F loc* C_0 M_0 *pc ics frs sh*)
obtain *a C1 fs* **where** *addr*: *hd* (*tl stk*) = *Null* ∨ (*hd* (*tl stk*) = *Addr a* ∧ *h a* = *Some*(*C1*,*fs*))
using *exec-step-ind-Putfield.prems*(*8*,*10*) **by**(*auto dest*!: *exec-step-input-StepID is-ptrD*)
then have ⋀*a*. *hd*(*tl stk*) = *Addr a* ⟹ *classes-above P C1* ⊆ *cset*
using *exec-step-ind-Putfield.prems*(*1*) *addr* **by** *auto*
then show *?case* **using** *exec-step-ind-Putfield.hyps exec-step-ind-Putfield.prems*(*1*− *7*) *addr*
by(*auto split*: *if-split-asm*) *blast*+
next
case *exec-step-ind-Putstatic-NoField* **show** *?case*
using *exec-step-ind-Putstatic-NoField.prems*(*1*− *7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast*+
next
case *exec-step-ind-Putstatic-NonStatic* **show** *?case*
using *exec-step-ind-Putstatic-NonStatic.prems*(*1*− *7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast*+
next
case (*exec-step-ind-Putstatic-Called D′ b t P D F C sh sfs i h stk loc* C_0 M_0 *pc Cs frs*)
then have *P* ⊢ *D sees F*,*Static*:*t in D* **by**(*simp add*: *has-field-sees*[*OF has-field-idemp*])
then have *D′eq*: *D′* = *D* **using** *exec-step-ind-Putstatic-Called.hyps*(*1*) **by** *simp*
obtain *sobj* **where** *sh D* = *Some sobj*
using *exec-step-ind-Putstatic-Called.hyps*(*2*) *exec-step-ind-Putstatic-Called.prems*(*8*,*13*)
by(*fastforce dest*!: *exec-step-input-StepID*)
then show *?case* **using** *exec-step-ind-Putstatic-Called.hyps*
exec-step-ind-Putstatic-Called.prems(*1*− *7*) *D′eq*
by(*auto split*: *if-split-asm*) *blast*+
next
case *exec-step-ind-Putstatic-Done* **show** *?case*
using *exec-step-ind-Putstatic-Done.hyps exec-step-ind-Putstatic-Done.prems*(*1*− *7*)
by(*auto split*: *if-split-asm*) *blast*+
next
case *exec-step-ind-Putstatic-Init* **show** *?case*
using *exec-step-ind-Putstatic-Init.hyps exec-step-ind-Putstatic-Init.prems*(*1*− *7*)
by(*auto split*: *if-split-asm*) *blast*+

next
case *exec-step-ind-Checkcast* **show** *?case*
using *exec-step-ind-Checkcast.prems*(*1*−*7*) **by** *auto*
next
case *exec-step-ind-Checkcast-Error* **show** *?case* **using** *exec-step-ind-Checkcast-Error.prems*(*1*−*7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast*+
next
case *exec-step-ind-Invoke-Null* **show** *?case* **using** *exec-step-ind-Invoke-Null.prems*(*1*−*7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast*+
next
case *exec-step-ind-Invoke-NoMethod* **show** *?case* **using** *exec-step-ind-Invoke-NoMethod.prems*(*1*−*7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast*+
next
case *exec-step-ind-Invoke-Static* **show** *?case* **using** *exec-step-ind-Invoke-Static.prems*(*1*−*7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast*+
next
case (*exec-step-ind-Invoke ps n stk r C h D b Ts T mxs* mxl_0 *ins xt P*)
have *classes-above P D* ⊆ *cset*
using *exec-step-ind-Invoke.hyps*(*2*,*3*,*5*) *exec-step-ind-Invoke.prems*(*1*,*8*,*10*,*13*)
rtrancl-trans[*OF sees-method-decl-above*[*OF exec-step-ind-Invoke.hyps*(*6*)]]
by(*auto dest*!: *exec-step-input-StepID is-ptrD*) *blast*+
then show *?case*
using *exec-step-ind-Invoke.hyps*(*7*) *exec-step-ind-Invoke.prems*(*1*−*7*) **by** *auto*
next
case *exec-step-ind-Invokestatic-NoMethod*
show *?case* **using** *exec-step-ind-Invokestatic-NoMethod.prems*(*1*−*7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast*+
next
case *exec-step-ind-Invokestatic-NonStatic*
show *?case* **using** *exec-step-ind-Invokestatic-NonStatic.prems*(*1*−*7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast*+
next
case (*exec-step-ind-Invokestatic-Called ps n stk D b Ts T mxs* mxl_0 *ins xt P*
C M)
have *seeing-class P C M* = *Some D* **using** *exec-step-ind-Invokestatic-Called.hyps*(*2*,*3*)
by(*fastforce simp*: *seeing-class-def*)
then have *classes-above P D* ⊆ *cset* **using** *exec-step-ind-Invokestatic-Called.prems*(*8*−*9*)
by(*fastforce dest*!: *exec-step-input-StepID*)
then show *?case*
using *exec-step-ind-Invokestatic-Called.hyps exec-step-ind-Invokestatic-Called.prems*(*1*−*7*)
by(*auto simp*: *seeing-class-def*)
next
case (*exec-step-ind-Invokestatic-Done ps n stk D b Ts T mxs* mxl_0 *ins xt P C*
M)
have *seeing-class P C M* = *Some D* **using** *exec-step-ind-Invokestatic-Done.hyps*(*2*,*3*)
by(*fastforce simp*: *seeing-class-def*)
then have *classes-above P D* ⊆ *cset* **using** *exec-step-ind-Invokestatic-Done.prems*(*8*−*9*)
by(*fastforce dest*!: *exec-step-input-StepID*)
then show *?case*

using *exec-step-ind-Invokestatic-Done.hyps exec-step-ind-Invokestatic-Done.prems*(*1*− *7*)
by *auto blast*+
next
case *exec-step-ind-Invokestatic-Init* **show** *?case*
using *exec-step-ind-Invokestatic-Init.hyps exec-step-ind-Invokestatic-Init.prems*(*1*− *7*)
by *auto blast*+
next
case *exec-step-ind-Return-Last-Init* **show** *?case*
using *exec-step-ind-Return-Last-Init.prems*(*1*− *7*) **by**(*auto split*: *if-split-asm*)
blast+
next
case *exec-step-ind-Return-Last* **show** *?case*
using *exec-step-ind-Return-Last.prems*(*1*− *7*) **by** *auto*
next
case *exec-step-ind-Return-Init* **show** *?case*
using *exec-step-ind-Return-Init.prems*(*1*− *7*) **by**(*auto split*: *if-split-asm*) *blast*+
next
case *exec-step-ind-Return-NonStatic* **show** *?case*
using *exec-step-ind-Return-NonStatic.prems*(*1*− *7*) **by** *auto*
next
case *exec-step-ind-Return-Static* **show** *?case*
using *exec-step-ind-Return-Static.prems*(*1*− *7*) **by** *auto*
next
case *exec-step-ind-Pop* **show** *?case* **using** *exec-step-ind-Pop.prems*(*1*− *7*) **by**
auto
next
case *exec-step-ind-IAdd* **show** *?case* **using** *exec-step-ind-IAdd.prems*(*1*− *7*)**by**
auto
next
case *exec-step-ind-IfFalse-False* **show** *?case*
using *exec-step-ind-IfFalse-False.prems*(*1*− *7*) **by** *auto*
next
case *exec-step-ind-IfFalse-nFalse* **show** *?case*
using *exec-step-ind-IfFalse-nFalse.prems*(*1*− *7*) **by** *auto*
next
case *exec-step-ind-CmpEq* **show** *?case* **using** *exec-step-ind-CmpEq.prems*(*1*− *7*)
by *auto*
next
case *exec-step-ind-Goto* **show** *?case* **using** *exec-step-ind-Goto.prems*(*1*− *7*)
by *auto*
next
case *exec-step-ind-Throw* **show** *?case* **using** *exec-step-ind-Throw.prems*(*1*− *7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast*+
next
case *exec-step-ind-Throw-Null* **show** *?case* **using** *exec-step-ind-Throw-Null.prems*(*1*− *7*)
by(*auto simp del*: *find-handler.simps dest*!: *find-handler-pieces*) *blast*+
next
case (*exec-step-ind-Init-None-Called sh C Cs P*)
have *classes-above P C* ⊆ *cset* **using** *exec-step-ind-Init-None-Called.prems*(*8*,*11*)

```
        by(auto dest!: exec-step-input-StepCD)
      then show ?case using exec-step-ind-Init-None-Called.prems(1−7)
        by(auto split: if-split-asm) blast+
    next
      case exec-step-ind-Init-Done show ?case
       using exec-step-ind-Init-Done.prems(1−7) by auto
    next
      case exec-step-ind-Init-Processing show ?case
       using exec-step-ind-Init-Processing.prems(1−7) by auto
    next
      case exec-step-ind-Init-Error show ?case
       using exec-step-ind-Init-Error.prems(1−7) by auto
    next
      case exec-step-ind-Init-Prepared-Object show ?case
       using exec-step-ind-Init-Prepared-Object.hyps
             exec-step-ind-Init-Prepared-Object.prems(1−7,10)
        by(auto split: if-split-asm dest!: exec-step-input-StepCD) blast+
    next
      case exec-step-ind-Init-Prepared-nObject show ?case
     using exec-step-ind-Init-Prepared-nObject.hyps exec-step-ind-Init-Prepared-nObject.prems(1−7)
        by(auto split: if-split-asm) blast+
    next
      case exec-step-ind-Init show ?case
       using exec-step-ind-Init.prems(1−7,8,12)
        by(auto simp: split-beta dest!: exec-step-input-StepC2D)
    next
      case (exec-step-ind-InitThrow C Cs a P h stk loc C0 M0 pc ics frs sh)
      obtain sfs where sh C = Some(sfs,Processing)
     using exec-step-ind-InitThrow.prems(8,14) by(fastforce dest!: exec-step-input-StepTD)
      then show ?case using exec-step-ind-InitThrow.prems(1−7)
        by(auto split: if-split-asm) blast+
    next
    case exec-step-ind-InitThrow-End show ?case using exec-step-ind-InitThrow-End.prems(1−7)
        by(auto simp del: find-handler.simps dest!: find-handler-pieces) blast+
    qed
  qed(simp)
qed(simp)
```

— Forward promises (classes that will be collected by the end of execution)
— - Classes that the current instruction will check initialization for will be collected
— - Class whose initialization is actively being called by the current frame will be collected

We prove that an *ics* of *Calling C Cs* (meaning *C*'s initialization procedure is actively being called) means that classes above *C* will be collected by *cbig* (i.e., by the end of execution) using proof by induction, proving the base and IH separately.

lemma *Calling-collects-base*:
assumes *big*: (σ', *cset'*) ∈ *JVMSmartCollectionSemantics.cbig P* σ

and *nend*: $\sigma \notin$ *JVMendset*
and *ics*: *ics-of* (*hd*(*frames-of* σ)) = *Calling Object Cs*
shows *classes-above P Object* $\subseteq$ *cset* $\cup$ *cset'*
proof(*cases frames-of* σ)
case *Nil* **then show** *?thesis* **using** *nend* **by**(*clarsimp simp*: *JVMendset-def*)
next
case (*Cons f1 frs1*)
then obtain *stk loc C M pc ics* **where** *f1* = (*stk*,*loc*,*C*,*M*,*pc*,*ics*) **by**(*cases f1*)
then show *?thesis*
using *JVMSmartCollectionSemantics.cbig-stepD*[*OF big nend*] *nend ics Cons*
by(*clarsimp simp*: *JVMSmartCollectionSemantics.csmall-def JVMendset-def*)
qed

— IH case where C has not been prepared yet
lemma *Calling-None-next-state*:
assumes *ics*: *ics-of* (*hd*(*frames-of* σ)) = *Calling C Cs*
and *none*: *sheap* σ *C* = *None*
and *set*: $\forall C'. P \vdash C \preceq^* C' \longrightarrow (\exists sfs\ i.\ sheap\ \sigma\ C' = Some(sfs,i))$
$\longrightarrow$ *classes-above P C'* $\subseteq$ *cset*
and σ': $(\sigma', cset') \in$ *JVMSmartCollectionSemantics.csmall P* σ
shows $\sigma' \notin$ *JVMendset* $\wedge$ *ics-of* (*hd*(*frames-of* σ')) = *Calling C Cs*
$\wedge$ ($\exists$ *sfs*. *sheap* σ' *C* = *Some*(*sfs*,*Prepared*))
$\wedge$ ($\forall C'. P \vdash C \preceq^* C' \longrightarrow C \neq C'$
$\longrightarrow (\exists sfs\ i.\ sheap\ \sigma'\ C' = Some(sfs,i)) \longrightarrow$ *classes-above P C'* $\subseteq$ *cset*)
proof(*cases frames-of* σ = [] $\vee$ ($\exists x$. *fst* σ = *Some x*))
case *True* **then show** *?thesis* **using** *assms*
by(*cases* σ, *auto simp*: *JVMSmartCollectionSemantics.csmall-def*)
next
case *False*
then obtain *f1 frs1* **where** *frs*: *frames-of* σ = *f1#frs1* **by**(*cases frames-of* σ, *auto*)
obtain *stk loc C' M pc ics* **where** *f1*: *f1* = (*stk*,*loc*,*C'*,*M*,*pc*,*ics*) **by**(*cases f1*)
show *?thesis* **using** *f1 frs False assms*
by(*cases* σ, *cases method P C clinit*)
(*clarsimp simp*: *split-beta JVMSmartCollectionSemantics.csmall-def JVMendset-def*)
qed

— IH case where C has been prepared (and has a direct superclass - i.e., is not *Object*)
lemma *Calling-Prepared-next-state*:
assumes *sub*: $P \vdash C \prec^1 D$
and *obj*: $P \vdash D \preceq^*$ *Object*
and *ics*: *ics-of* (*hd*(*frames-of* σ)) = *Calling C Cs*
and *prep*: *sheap* σ *C* = *Some*(*sfs*,*Prepared*)
and *set*: $\forall C'. P \vdash C \preceq^* C' \longrightarrow C \neq C' \longrightarrow (\exists sfs\ i.\ sheap\ \sigma\ C' = Some(sfs,i))$
$\longrightarrow$ *classes-above P C'* $\subseteq$ *cset*
and σ': $(\sigma', cset') \in$ *JVMSmartCollectionSemantics.csmall P* σ
shows $\sigma' \notin$ *JVMendset* $\wedge$ *ics-of* (*hd* (*frames-of* σ')) = *Calling D* (*C#Cs*)

$\wedge$ ($\forall$ C'. $P \vdash D \preceq^* C' \longrightarrow$ ($\exists$ *sfs i. sheap* σ' C' = *Some*(*sfs*,*i*))
$\longrightarrow$ *classes-above P* $C' \subseteq$ *cset*)
using *sub*
proof(*cases C=Object*)
case *nobj*:*False* **show** *?thesis*
proof(*cases frames-of* σ = [] $\vee$ ($\exists$ *x. fst* σ = *Some x*))
case *True* **then show** *?thesis* **using** *assms*
by(*cases* σ, *auto simp*: *JVMSmartCollectionSemantics.csmall-def*)
next
case *False*
then obtain *f1 frs1* **where** *frs*: *frames-of* σ = *f1#frs1* **by**(*cases frames-of* σ, *auto*)
obtain *stk loc* C' *M pc ics* **where** *f1*: *f1* = (*stk*,*loc*,C',*M*,*pc*,*ics*) **by**(*cases f1*)
have $C \neq D$ **using** *sub obj subcls-self-superclass* **by** *auto*
then have *dimp*: $\forall$ C'. $P \vdash D \preceq^* C' \longrightarrow P \vdash C \preceq^* C' \wedge C \neq C'$
using *sub subcls-of-Obj-acyclic*[*OF obj*] **by** *fastforce*
have $\forall$ C'. $P \vdash C \preceq^* C' \longrightarrow C \neq C' \longrightarrow$ ($\exists$ *sfs i. sheap* σ' C' = *Some*(*sfs*,*i*))
$\longrightarrow$ *classes-above P* $C' \subseteq$ *cset*
using *f1 frs False nobj assms*
by(*cases* σ, *cases method P C clinit*)
(*auto simp*: *JVMSmartCollectionSemantics.csmall-def JVMendset-def*)
then have $\forall$ C'. $P \vdash D \preceq^* C' \longrightarrow$ ($\exists$ *sfs i. sheap* σ' C' = *Some*(*sfs*,*i*))
$\longrightarrow$ *classes-above P* $C' \subseteq$ *cset* **using** *sub dimp* **by** *auto*
then show *?thesis* **using** *f1 frs False nobj assms*
by(*cases* σ, *cases method P C clinit*)
(*auto dest*:*subcls1D simp*: *JVMSmartCollectionSemantics.csmall-def JVMendset-def*)
qed
qed(*simp*)

— completed IH case: non-*Object* (pulls together above IH cases)
lemma *Calling-collects-IH*:
assumes *sub*: $P \vdash C \prec^1 D$
and *obj*: $P \vdash D \preceq^* Object$
and *step*: $\bigwedge\sigma$ *cset'* *Cs*. (σ', *cset'*) $\in$ *JVMSmartCollectionSemantics.cbig P* $\sigma \Longrightarrow$ $\sigma \notin$ *JVMendset*
$\Longrightarrow$ *ics-of* (*hd*(*frames-of* σ)) = *Calling D Cs*
$\Longrightarrow \forall$ C'. $P \vdash D \preceq^* C' \longrightarrow$ ($\exists$ *sfs i. sheap* σ C' = *Some*(*sfs*,*i*))
$\longrightarrow$ *classes-above P* $C' \subseteq$ *cset*
$\Longrightarrow$ *classes-above P D* $\subseteq$ *cset* $\cup$ *cset'*
and *big*: (σ', *cset'*) $\in$ *JVMSmartCollectionSemantics.cbig P* σ
and *nend*: $\sigma \notin$ *JVMendset*
and *curr*: *ics-of* (*hd*(*frames-of* σ)) = *Calling C Cs*
and *set*: $\forall$ C'. $P \vdash C \preceq^* C' \longrightarrow$ ($\exists$ *sfs i. sheap* σ C' = *Some*(*sfs*,*i*))
$\longrightarrow$ *classes-above P* $C' \subseteq$ *cset*
shows *classes-above P C* $\subseteq$ *cset* $\cup$ *cset'*
proof(*cases frames-of* σ)
case *Nil* **then show** *?thesis* **using** *nend* **by**(*clarsimp simp*: *JVMendset-def*)
next

case (*Cons f1 frs1*)
show *?thesis* **using** *sub*
proof(*cases* $\exists$ *sfs i. sheap* σ *C* = *Some*(*sfs*,*i*))
case *True* **then show** *?thesis* **using** *set* **by** *auto*
next
case *False*
obtain *stk loc C' M pc ics* **where** *f1*: *f1* = (*stk*,*loc*,*C'*,*M*,*pc*,*ics*) **by**(*cases f1*)
then obtain σ*1 coll1 coll* **where** σ*1*: (σ*1*, *coll1*) $\in$ *JVMSmartCollectionSemantics.csmall P* σ
cset' = *coll1* $\cup$ *coll* (σ', *coll*) $\in$ *JVMSmartCollectionSemantics.cbig P* σ*1*
using *JVMSmartCollectionSemantics.cbig-stepD*[*OF big nend*] **by** *clarsimp*
show *?thesis*
proof(*cases* $\exists$ *sfs. sheap* σ *C* = *Some*(*sfs*,*Prepared*))
case *True*
then obtain *sfs* **where** *sfs*: *sheap* σ *C* = *Some*(*sfs*,*Prepared*) **by** *clarsimp*
have *set'*: $\forall$ *C'*. *P* $\vdash$ *C* $\preceq^*$ *C'* $\longrightarrow$ *C*$\neq$*C'* $\longrightarrow$ ($\exists$ *sfs i. sheap* σ *C'* = *Some*(*sfs*,*i*)) $\longrightarrow$ *classes-above P C'* $\subseteq$ *cset* **using** *set* **by** *auto*
then have σ*1* $\notin$ *JVMendset* $\wedge$ *ics-of* (*hd* (*frames-of* σ*1*)) = *Calling D* (*C*#*Cs*)
$\forall$ *C'*. *P* $\vdash$ *D* $\preceq^*$ *C'* $\longrightarrow$ ($\exists$ *sfs i. sheap* σ*1 C'* = *Some*(*sfs*,*i*)) $\longrightarrow$ *classes-above P C'* $\subseteq$ *cset*
using *Calling-Prepared-next-state*[*OF sub obj curr sfs set'* σ*1*(*1*)]
by(*auto simp*: *JVMSmartCollectionSemantics.csmall-def*)
then show *?thesis* **using** *step*[*of coll* σ*1*] *classes-above-def2*[*OF sub*] σ*1 f1 Cons nend curr*
by(*clarsimp simp*: *JVMSmartCollectionSemantics.csmall-def JVMendset-def*)
next
case *none*: *False* — *Calling C Cs* is the next *ics*, but after that is *Calling D* (*C*#*Cs*)
then have *sNone*: *sheap* σ *C* = *None* **using** *False* **by**(*cases sheap* σ *C*, *auto*)
then have *nend1*: σ*1* $\notin$ *JVMendset* **and** *curr1*: *ics-of* (*hd* (*frames-of* σ*1*)) = *Calling C Cs*
and *prep*: $\exists$ *sfs. sheap* σ*1 C* = $\lfloor$(*sfs*, *Prepared*)$\rfloor$
and *set1*: $\forall$ *C'*. *P* $\vdash$ *C* $\preceq^*$ *C'* $\longrightarrow$ *C* $\neq$ *C'* $\longrightarrow$ ($\exists$ *sfs i. sheap* σ*1 C'* = $\lfloor$(*sfs*, *i*)$\rfloor$) $\longrightarrow$ *classes-above P C'* $\subseteq$ *cset*
using *Calling-None-next-state*[*OF curr sNone set* σ*1*(*1*)] **by** *simp+*
then obtain *f2 frs2* **where** *frs2*: *frames-of* σ*1* = *f2*#*frs2*
by(*cases* σ*1*, *cases frames-of* σ*1*, *clarsimp simp*: *JVMendset-def*)
obtain *sfs1* **where** *sfs1*: *sheap* σ*1 C* = *Some*(*sfs1*,*Prepared*) **using** *prep* **by** *clarsimp*
obtain *stk2 loc2 C2 M2 pc2 ics2* **where** *f2*: *f2* = (*stk2*,*loc2*,*C2*,*M2*,*pc2*,*ics2*) **by**(*cases f2*)
then obtain σ*2 coll2 coll'* **where** σ*2*: (σ*2*, *coll2*) $\in$ *JVMSmartCollectionSemantics.csmall P* σ*1*
coll = *coll2* $\cup$ *coll'* (σ', *coll'*) $\in$ *JVMSmartCollectionSemantics.cbig P* σ*2*
using *JVMSmartCollectionSemantics.cbig-stepD*[*OF* σ*1*(*3*) *nend1*] **by** *clarsimp*
then have σ*2* $\notin$ *JVMendset* $\wedge$ *ics-of* (*hd* (*frames-of* σ*2*)) = *Calling D* (*C*#*Cs*)

$\forall C'.\ P \vdash D \preceq^* C' \longrightarrow (\exists sfs\ i.\ sheap\ \sigma 2\ C' = Some(sfs,i))$
$\longrightarrow classes\text{-}above\ P\ C' \subseteq cset$
using *Calling-Prepared-next-state*[*OF sub obj curr1 sfs1 set1* $\sigma 2(1)$]
by(*auto simp*: *JVMSmartCollectionSemantics.csmall-def*)
then show *?thesis* **using** *step*[*of coll'* $\sigma 2$] *classes-above-def2*[*OF sub*] $\sigma 2$ $\sigma 1$ *f2 frs2 f1 Cons*
nend1 nend curr1 curr
by(*clarsimp simp*: *JVMSmartCollectionSemantics.csmall-def JVMendset-def*)
qed
qed
qed

— pulls together above base and IH cases
lemma *Calling-collects*:
assumes *sub*: $P \vdash C \preceq^* Object$
and $(\sigma', cset') \in JVMSmartCollectionSemantics.cbig\ P\ \sigma$
and $\sigma \notin JVMendset$
and $ics\text{-}of\ (hd(frames\text{-}of\ \sigma)) = Calling\ C\ Cs$
and $\forall C'.\ P \vdash C \preceq^* C' \longrightarrow (\exists sfs\ i.\ sheap\ \sigma\ C' = Some(sfs,i))$
$\longrightarrow classes\text{-}above\ P\ C' \subseteq cset$
and $cset' \subseteq cset$
shows $classes\text{-}above\ P\ C \subseteq cset$
proof –
have *base*: $\forall \sigma\ cset'\ Cs.$
$(\sigma', cset') \in JVMSmartCollectionSemantics.cbig\ P\ \sigma \longrightarrow \sigma \notin JVMendset$
$\longrightarrow ics\text{-}of\ (hd\ (frames\text{-}of\ \sigma)) = Calling\ Object\ Cs$
$\longrightarrow (\forall C'.\ P \vdash Object \preceq^* C' \longrightarrow (\exists sfs\ i.\ sheap\ \sigma\ C' = \lfloor(sfs,\ i)\rfloor)$
$\longrightarrow classes\text{-}above\ P\ C' \subseteq cset)$
$\longrightarrow classes\text{-}above\ P\ Object \subseteq JVMcombine\ cset\ cset'$ **using** *Calling-collects-base* **by** *simp*
have *IH*: $\bigwedge y\ z.\ P \vdash y \prec^1 z \Longrightarrow$
$P \vdash z \preceq^* Object \Longrightarrow$
$\forall \sigma\ cset'\ Cs.\ (\sigma', cset') \in JVMSmartCollectionSemantics.cbig\ P\ \sigma \longrightarrow \sigma \notin JVMendset$
$\longrightarrow ics\text{-}of\ (hd(frames\text{-}of\ \sigma)) = Calling\ z\ Cs$
$\longrightarrow (\forall C'.\ P \vdash z \preceq^* C' \longrightarrow (\exists sfs\ i.\ sheap\ \sigma\ C' = Some(sfs,i))$
$\longrightarrow classes\text{-}above\ P\ C' \subseteq cset)$
$\longrightarrow classes\text{-}above\ P\ z \subseteq cset \cup cset' \Longrightarrow$
$\forall \sigma\ cset'\ Cs.\ (\sigma', cset') \in JVMSmartCollectionSemantics.cbig\ P\ \sigma \longrightarrow \sigma \notin JVMendset$
$\longrightarrow ics\text{-}of\ (hd(frames\text{-}of\ \sigma)) = Calling\ y\ Cs$
$\longrightarrow (\forall C'.\ P \vdash y \preceq^* C' \longrightarrow (\exists sfs\ i.\ sheap\ \sigma\ C' = Some(sfs,i))$
$\longrightarrow classes\text{-}above\ P\ C' \subseteq cset)$
$\longrightarrow classes\text{-}above\ P\ y \subseteq cset \cup cset'$
using *Calling-collects-IH* **by** *blast*
have *result*: $\forall \sigma\ cset'\ Cs.$
$(\sigma', cset') \in JVMSmartCollectionSemantics.cbig\ P\ \sigma \longrightarrow \sigma \notin JVMendset$
$\longrightarrow ics\text{-}of\ (hd(frames\text{-}of\ \sigma)) = Calling\ C\ Cs$
$\longrightarrow (\forall C'.\ P \vdash C \preceq^* C' \longrightarrow (\exists sfs\ i.\ sheap\ \sigma\ C' = Some(sfs,i))$

$\longrightarrow$ *classes-above P C′* $\subseteq$ *cset*)
$\longrightarrow$ *classes-above P C* $\subseteq$ *cset* $\cup$ *cset′*
using *converse-rtrancl-induct*[*OF sub*,
where *P*=λ*C*. $\forall$ σ *cset′ Cs*. (σ',*cset′*) $\in$ *JVMSmartCollectionSemantics.cbig P* σ $\longrightarrow$ $\sigma \notin$ *JVMendset*
$\longrightarrow$ *ics-of* (*hd*(*frames-of* σ)) = *Calling C Cs*
$\longrightarrow$ ($\forall$ *C′*. *P* $\vdash$ *C* $\preceq^*$ *C′* $\longrightarrow$ ($\exists$ *sfs i*. *sheap* σ *C′* = *Some*(*sfs*,*i*))
$\longrightarrow$ *classes-above P C′* $\subseteq$ *cset*)
$\longrightarrow$ *classes-above P C* $\subseteq$ *cset* $\cup$ *cset′*]
using *base IH* **by** *blast*
then show *?thesis* **using** *assms* **by** *blast*
qed

Instructions that call the initialization procedure will collect classes above the class initialized by the end of execution (using the above *Calling-collects*).

lemma *New-collects*:
assumes *sub*: *P* $\vdash$ *C* $\preceq^*$ *Object*
and *cbig*: (σ', *cset′*) $\in$ *JVMSmartCollectionSemantics.cbig P* σ
and *nend*: $\sigma \notin$ *JVMendset*
and *curr*: *curr-instr P* (*hd*(*frames-of* σ)) = *New C*
and *ics*: *ics-of* (*hd*(*frames-of* σ)) = *No-ics*
and *sheap*: $\forall$ *C′*. *P* $\vdash$ *C* $\preceq^*$ *C′* $\longrightarrow$ ($\exists$ *sfs i*. *sheap* σ *C′* = *Some*(*sfs*,*i*))
$\longrightarrow$ *classes-above P C′* $\subseteq$ *cset*
and *smart*: *cset′* $\subseteq$ *cset*
shows *classes-above P C* $\subseteq$ *cset*
proof(*cases* ($\exists$ *sfs i*. *sheap* σ *C* = *Some*(*sfs*,*i*) $\wedge$ *i* = *Done*))
case *True* **then show** *?thesis* **using** *sheap* **by** *auto*
next
case *False*
obtain *n* **where** *nstep*: (σ', *cset′*) $\in$ *JVMSmartCollectionSemantics.csmall-nstep P* σ *n*
and *n* $\neq$ *0* **using** *nend cbig JVMSmartCollectionSemantics.cbig-def2 JVMSmartCollectionSemantics.csmall-nstep-base* **by** (*metis empty-iff insert-iff*)
then show *?thesis*
proof(*cases n*)
case (*Suc n1*)
then obtain σ*1 cset0 cset1* **where** σ*1*: (σ*1*,*cset1*) $\in$ *JVMSmartCollectionSemantics.csmall P* σ
cset′ = *cset1* $\cup$ *cset0* (σ',*cset0*) $\in$ *JVMSmartCollectionSemantics.csmall-nstep P* σ*1 n1*
using *JVMSmartCollectionSemantics.csmall-nstep-SucD nstep* **by** *blast*
obtain *xp h frs sh* **where** σ=(*xp*,*h*,*frs*,*sh*) **by**(*cases* σ)
then have *ics1*: *ics-of* (*hd*(*frames-of* σ*1*)) = *Calling C* []
and *sheap′*: *sheap* σ = *sheap* σ*1* **and** *nend1*: σ*1* $\notin$ *JVMendset*
using *JVM-New-next-step*[*OF - nend curr*] σ*1*(*1*) *False ics*
by(*simp add*: *JVMSmartCollectionSemantics.csmall-def*)+
have $\sigma' \in$ *JVMendset* **using** *cbig JVMSmartCollectionSemantics.cbig-def2* **by** *blast*
then have *cbig1*: (σ', *cset0*) $\in$ *JVMSmartCollectionSemantics.cbig P* σ*1*

using *JVMSmartCollectionSemantics.cbig-def2 σ1(3)* **by** *blast*
have *sheap1*: ∀ *C′*. *P* ⊢ *C* ⪯* *C′* ⟶ (∃ *sfs i*. *sheap σ1 C′* = ⌊(*sfs*, *i*)⌋)
⟶ *classes-above P C′* ⊆ *cset* **using** *sheap′ sheap* **by** *simp*
have *cset0* ⊆ *cset* **using** *σ1(2) smart* **by** *blast*
then have *classes-above P C* ⊆ *cset*
using *Calling-collects*[*OF sub cbig1 nend1 ics1 sheap1*] **by** *simp*
then show *?thesis* **using** *σ1(2) smart* **by** *auto*
qed(*simp*)
qed

lemma *Getstatic-collects*:
assumes *sub*: *P* ⊢ *D* ⪯* *Object*
and *cbig*: (*σ′*, *cset′*) ∈ *JVMSmartCollectionSemantics.cbig P σ*
and *nend*: *σ* ∉ *JVMendset*
and *curr*: *curr-instr P* (*hd*(*frames-of σ*)) = *Getstatic C F D*
and *ics*: *ics-of* (*hd*(*frames-of σ*)) = *No-ics*
and *fC*: *P* ⊢ *C has F*,*Static*:*t in D*
and *sheap*: ∀ *C′*. *P* ⊢ *D* ⪯* *C′* ⟶ (∃ *sfs i*. *sheap σ C′* = *Some*(*sfs*,*i*))
⟶ *classes-above P C′* ⊆ *cset*
and *smart*: *cset′* ⊆ *cset*
shows *classes-above P D* ⊆ *cset*
proof(*cases* (∃ *sfs i*. *sheap σ D* = *Some*(*sfs*,*i*) ∧ *i* = *Done*)
∨ (*ics-of*(*hd*(*frames-of σ*)) = *Called* []))
case *True* **then show** *?thesis*
proof(*cases* ∃ *sfs i*. *sheap σ D* = *Some*(*sfs*,*i*) ∧ *i* = *Done*)
case *True* **then show** *?thesis* **using** *sheap* **by** *auto*
next
case *False*
then have *ics-of*(*hd*(*frames-of σ*)) = *Called* [] **using** *True* **by** *clarsimp*
then show *?thesis* **using** *ics* **by** *auto*
qed
next
case *False*
obtain *n* **where** *nstep*: (*σ′*, *cset′*) ∈ *JVMSmartCollectionSemantics.csmall-nstep P σ n*
and *n* ≠ *0* **using** *nend cbig JVMSmartCollectionSemantics.cbig-def2 JVMSmartCollectionSemantics.csmall-nstep-base* **by** (*metis empty-iff insert-iff*)
then show *?thesis*
proof(*cases n*)
case (*Suc n1*)
then obtain *σ1 cset0 cset1* **where** *σ1*: (*σ1*,*cset1*) ∈ *JVMSmartCollectionSemantics.csmall P σ*
cset′ = *cset1* ∪ *cset0* (*σ′*,*cset0*) ∈ *JVMSmartCollectionSemantics.csmall-nstep P σ1 n1*
using *JVMSmartCollectionSemantics.csmall-nstep-SucD nstep* **by** *blast*
obtain *xp h frs sh* **where** *σ*=(*xp*,*h*,*frs*,*sh*) **by**(*cases σ*)
then have *curr1*: *ics-of* (*hd*(*frames-of σ1*)) = *Calling D* []
and *sheap′*: *sheap σ* = *sheap σ1* **and** *nend1*: *σ1* ∉ *JVMendset*
using *JVM-Getstatic-next-step*[*OF - nend curr fC*] *σ1(1) False ics*

by(*simp add*: *JVMSmartCollectionSemantics.csmall-def*)+
have σ' ∈ *JVMendset* **using** *cbig JVMSmartCollectionSemantics.cbig-def2* **by** *blast*
then have *cbig1*: (σ', *cset0*) ∈ *JVMSmartCollectionSemantics.cbig P* $\sigma 1$
using *JVMSmartCollectionSemantics.cbig-def2* $\sigma 1(3)$ **by** *blast*
have *sheap1*: ∀ *C'*. *P* ⊢ *D* $\preceq^*$ *C'* ⟶ (∃ *sfs i*. *sheap* $\sigma 1$ *C'* = ⌊(*sfs*, *i*)⌋)
⟶ *classes-above P C'* ⊆ *cset* **using** *sheap' sheap* **by** *simp*
have *cset0* ⊆ *cset* **using** $\sigma 1(2)$ *smart* **by** *blast*
then have *classes-above P D* ⊆ *cset*
using *Calling-collects*[*OF sub cbig1 nend1 curr1 sheap1*] **by** *simp*
then show *?thesis* **using** $\sigma 1(2)$ *smart* **by** *auto*
qed(*simp*)
qed

lemma *Putstatic-collects*:
assumes *sub*: *P* ⊢ *D* $\preceq^*$ *Object*
and *cbig*: (σ', *cset'*) ∈ *JVMSmartCollectionSemantics.cbig P* σ
and *nend*: σ ∉ *JVMendset*
and *curr*: *curr-instr P* (*hd*(*frames-of* σ)) = *Putstatic C F D*
and *ics*: *ics-of* (*hd*(*frames-of* σ)) = *No-ics*
and *fC*: *P* ⊢ *C has F*,*Static*:*t in D*
and *sheap*: ∀ *C'*. *P* ⊢ *D* $\preceq^*$ *C'* ⟶ (∃ *sfs i*. *sheap* σ *C'* = *Some*(*sfs*,*i*))
⟶ *classes-above P C'* ⊆ *cset*
and *smart*: *cset'* ⊆ *cset*
shows *classes-above P D* ⊆ *cset*
proof(*cases* (∃ *sfs i*. *sheap* σ *D* = *Some*(*sfs*,*i*) ∧ *i* = *Done*)
∨ (*ics-of*(*hd*(*frames-of* σ)) = *Called* []))
case *True* **then show** *?thesis*
proof(*cases* ∃ *sfs i*. *sheap* σ *D* = *Some*(*sfs*,*i*) ∧ *i* = *Done*)
case *True* **then show** *?thesis* **using** *sheap* **by** *auto*
next
case *False*
then have *ics-of*(*hd*(*frames-of* σ)) = *Called* [] **using** *True* **by** *clarsimp*
then show *?thesis* **using** *ics* **by** *auto*
qed
next
case *False*
obtain *n* **where** *nstep*: (σ', *cset'*) ∈ *JVMSmartCollectionSemantics.csmall-nstep P* σ *n*
and *n* ≠ *0* **using** *nend cbig JVMSmartCollectionSemantics.cbig-def2*
JVMSmartCollectionSemantics.csmall-nstep-base **by** (*metis empty-iff insert-iff*)
then show *?thesis*
proof(*cases n*)
case (*Suc n1*)
then obtain $\sigma 1$ *cset0 cset1* **where** $\sigma 1$: ($\sigma 1$,*cset1*) ∈ *JVMSmartCollectionSemantics.csmall P* σ
cset' = *cset1* ∪ *cset0* (σ',*cset0*) ∈ *JVMSmartCollectionSemantics.csmall-nstep P* $\sigma 1$ *n1*
using *JVMSmartCollectionSemantics.csmall-nstep-SucD nstep* **by** *blast*

obtain *xp h frs sh* **where** σ=(*xp*,*h*,*frs*,*sh*) **by**(*cases* σ)
then have *curr1*: *ics-of* (*hd*(*frames-of* $\sigma 1$)) = *Calling D* []
and *sheap'*: *sheap* σ = *sheap* $\sigma 1$ **and** *nend1*: $\sigma 1 \notin$ *JVMendset*
using *JVM-Putstatic-next-step*[*OF - nend curr fC*] $\sigma 1(1)$ *False ics*
by(*simp add*: *JVMSmartCollectionSemantics.csmall-def*)+
have $\sigma' \in$ *JVMendset* **using** *cbig JVMSmartCollectionSemantics.cbig-def2* **by** *blast*
then have *cbig1*: (σ', *cset0*) $\in$ *JVMSmartCollectionSemantics.cbig P* $\sigma 1$
using *JVMSmartCollectionSemantics.cbig-def2* $\sigma 1(3)$ **by** *blast*
have *sheap1*: $\forall C'$. $P \vdash D \preceq^* C' \longrightarrow (\exists$ *sfs i*. *sheap* $\sigma 1$ $C' = \lfloor(sfs, i)\rfloor$)
$\longrightarrow$ *classes-above P C'* $\subseteq$ *cset* **using** *sheap' sheap* **by** *simp*
have *cset0* $\subseteq$ *cset* **using** $\sigma 1(2)$ *smart* **by** *blast*
then have *classes-above P D* $\subseteq$ *cset*
using *Calling-collects*[*OF sub cbig1 nend1 curr1 sheap1*] **by** *simp*
then show *?thesis* **using** $\sigma 1(2)$ *smart* **by** *auto*
qed(*simp*)
qed

lemma *Invokestatic-collects*:
assumes *sub*: $P \vdash D \preceq^*$ *Object*
and *cbig*: (σ', *cset'*) $\in$ *JVMSmartCollectionSemantics.cbig P* σ
and *smart*: *cset'* $\subseteq$ *cset*
and *nend*: $\sigma \notin$ *JVMendset*
and *curr*: *curr-instr P* (*hd*(*frames-of* σ)) = *Invokestatic C M n*
and *ics*: *ics-of* (*hd*(*frames-of* σ)) = *No-ics*
and *mC*: $P \vdash$ *C sees M*,*Static*:*Ts* $\rightarrow$ *T* = *m in D*
and *sheap*: $\forall C'$. $P \vdash D \preceq^* C' \longrightarrow (\exists$ *sfs i*. *sheap* σ C' = *Some*(*sfs*,*i*))
$\longrightarrow$ *classes-above P C'* $\subseteq$ *cset*
shows *classes-above P D* $\subseteq$ *cset*
proof(*cases* ($\exists$ *sfs i*. *sheap* σ *D* = *Some*(*sfs*,*i*) $\wedge$ *i* = *Done*)
$\vee$ (*ics-of*(*hd*(*frames-of* σ)) = *Called* []))
case *True* **then show** *?thesis*
proof(*cases* $\exists$ *sfs i*. *sheap* σ *D* = *Some*(*sfs*,*i*) $\wedge$ *i* = *Done*)
case *True* **then show** *?thesis* **using** *sheap* **by** *auto*
next
case *False*
then have *ics-of*(*hd*(*frames-of* σ)) = *Called* [] **using** *True* **by** *clarsimp*
then show *?thesis* **using** *ics* **by** *auto*
qed
next
case *False*
obtain *n* **where** *nstep*: (σ', *cset'*) $\in$ *JVMSmartCollectionSemantics.csmall-nstep P* σ *n*
and $n \neq 0$ **using** *nend cbig JVMSmartCollectionSemantics.cbig-def2 JVMSmartCollectionSemantics.csmall-nstep-base* **by** (*metis empty-iff insert-iff*)
then show *?thesis*
proof(*cases n*)
case (*Suc n1*)
then obtain $\sigma 1$ *cset0 cset1* **where** $\sigma 1$: ($\sigma 1$,*cset1*) $\in$ *JVMSmartCollectionSe-*

mantics.csmall P σ
cset′ = cset1 ∪ cset0 (σ′,cset0) ∈ JVMSmartCollectionSemantics.csmall-nstep P σ1 n1
using *JVMSmartCollectionSemantics.csmall-nstep-SucD nstep* **by** *blast*
obtain *xp h frs sh* **where** *σ=(xp,h,frs,sh)* **by**(*cases σ*)
then have *curr1*: *ics-of (hd(frames-of σ1)) = Calling D []*
and *sheap′*: *sheap σ = sheap σ1* **and** *nend1*: *σ1 ∉ JVMendset*
using *JVM-Invokestatic-next-step[OF - nend curr mC] σ1(1) False ics*
by(*simp add*: *JVMSmartCollectionSemantics.csmall-def*)+
have *σ′ ∈ JVMendset* **using** *cbig JVMSmartCollectionSemantics.cbig-def2* **by** *blast*
then have *cbig1*: *(σ′, cset0) ∈ JVMSmartCollectionSemantics.cbig P σ1*
using *JVMSmartCollectionSemantics.cbig-def2 σ1(3)* **by** *blast*
have *sheap1*: $\forall C'.\ P \vdash D \preceq^* C' \longrightarrow (\exists sfs\ i.\ sheap\ \sigma 1\ C' = \lfloor(sfs, i)\rfloor)$
$\longrightarrow$ *classes-above P C′ ⊆ cset* **using** *sheap′ sheap* **by** *simp*
have *cset0 ⊆ cset* **using** *σ1(2) smart* **by** *blast*
then have *classes-above P D ⊆ cset*
using *Calling-collects[OF sub cbig1 nend1 curr1 sheap1]* **by** *simp*
then show *?thesis* **using** *σ1(2) smart* **by** *auto*
qed(*simp*)
qed

The *smart-out* execution function keeps the promise to collect above the initial class (*Test*):

lemma *jvm-smart-out-classes-above-Test*:
assumes *s*: $(\sigma',cset_s) \in$ *jvm-smart-out P t* **and** *P*: *P ∈ jvm-progs* **and** *t*: *t ∈ jvm-tests*
shows *classes-above (jvm-make-test-prog P t) Test* $\subseteq cset_s$
(**is** *classes-above ?P ?D ⊆ ?cset*)
proof −
let *?σ = start-state (t#P)* **and** *?M = main*
let *?ics = ics-of (hd(frames-of ?σ))*
have *called*: *?ics = Called [] ⟹ classes-above ?P ?D ⊆ ?cset*
by(*simp add*: *start-state-def*)
then show *?thesis*
proof(*cases ?ics = Called []*)
case *True* **then show** *?thesis* **using** *called* **by** *simp*
next
case *False*
from *P t* **obtain** *wf-md* **where** *wf*: *wf-prog wf-md (t#P)*
by(*auto simp*: *wf-jvm-prog-phi-def wf-jvm-prog-def*)
from *jvm-make-test-prog-sees-Test-main[OF P t]* **obtain** *m* **where**
mC: *?P ⊢ ?D sees ?M,Static:[] → Void = m in ?D* **by** *fast*

then have $?P \vdash ?D \preceq^*$ *Object* **by**(*rule sees-method-sub-Obj*)
moreover from *s* **obtain** *cset′* **where**
cbig: *(σ′, cset′) ∈ JVMSmartCollectionSemantics.cbig ?P ?σ* **and** *cset′ ⊆ ?cset* **by** *clarsimp*
moreover have *nend*: *?σ ∉ JVMendset* **by**(*rule start-state-nend*)

moreover from *start-prog-start-m-instrs*[*OF wf*] *t*
have *curr*: *curr-instr ?P* (*hd*(*frames-of ?σ*)) = *Invokestatic ?D ?M 0*
by(*simp add*: *start-state-def*)
moreover have *ics*: *?ics* = *No-ics*
by(*simp add*: *start-state-def*)
moreover note *mC*
moreover from *jvm-smart-out-classes-above-start-sheap*[*OF s*]
have *sheap*: $\forall C'.\ ?P \vdash ?D \preceq^* C' \longrightarrow (\exists sfs\ i.\ sheap\ ?\sigma\ C' = Some(sfs,i))$
$\longrightarrow$ *classes-above ?P C'* ⊆ *?cset* **by**(*simp add*: *start-state-def*)
ultimately show *?thesis* **by**(*rule Invokestatic-collects*)
qed
qed

Using lemmas proving preservation of backward promises and keeping of forward promises, we prove that the smart algorithm collects at least the classes as the naive algorithm does.

lemma *jvm-naive-to-smart-exec-collect*:
assumes
— well-formedness
wtp: $wf\text{-}jvm\text{-}prog_{\Phi}\ P$
and *correct*: $P,\Phi \vdash (xp,h,frs,sh)\surd$
— defs
and *f'*: *hd frs* = (*stk*,*loc*,*C'*,*M'*,*pc*,*ics*)
— backward promises - will be collected prior
and *heap*: $\bigwedge C\ fs.\ \exists a.\ h\ a = Some(C,fs) \Longrightarrow$ *classes-above P C* ⊆ *cset*
and *sheap*: $\bigwedge C\ sfs\ i.\ sh\ C = Some(sfs,i) \Longrightarrow$ *classes-above P C* ⊆ *cset*
and *xcpts*: *classes-above-xcpts P* ⊆ *cset*
and *frames*: *classes-above-frames P frs* ⊆ *cset*
— forward promises - will be collected after if not already
and *init-class-prom*: $\bigwedge C.$ *ics* = *Called* [] ∨ *ics* = *No-ics*
$\Longrightarrow$ *coll-init-class P* (*instrs-of P C' M'* ! *pc*) = *Some C* $\Longrightarrow$ *classes-above P C* ⊆ *cset*
and *Calling-prom*: $\bigwedge C'\ Cs'.$ *ics* = *Calling C' Cs'* $\Longrightarrow$ *classes-above P C'* ⊆ *cset*
— collection
and *smart*: *JVMexec-scollect P* (*xp*,*h*,*frs*,*sh*) ⊆ *cset*
shows *JVMexec-ncollect P* (*xp*,*h*,*frs*,*sh*) ⊆ *cset*
using *assms*
proof(*cases frs*)
case (*Cons f' frs'*)
then have [*simp*]: *classes-above P C'* ⊆ *cset* **using** *frames f'* **by** *simp*
let *?i* = *instrs-of P C' M'* ! *pc*
have *cr'*: $P,\Phi \vdash (xp,h,(stk,loc,C',M',pc,ics)\#frs',sh)\surd$ **using** *correct f' Cons* **by** *simp*
from *well-formed-stack-safe*[*OF wtp cr'*] *correct-state-Throwing-ex*[*OF cr'*] **obtain**
stack-safe: *stack-safe ?i h stk* **and**
Throwing-ex: $\bigwedge Cs\ a.$ *ics* = *Throwing Cs a* $\Longrightarrow \exists obj.\ h\ a = Some\ obj$ **by** *simp*
have *confc*: *conf-clinit P sh frs* **using** *correct Cons* **by** *simp*
have *Called-prom*: $\bigwedge C'\ Cs'.$ *ics* = *Called* (*C'*#*Cs'*)

$\Longrightarrow$ *classes-above P C' ⊆ cset* ∧ *classes-above P* (*fst*(*method P C' clinit*)) ⊆ *cset*
proof −
fix *C' Cs'* **assume** [*simp*]: *ics* = *Called* (*C'#Cs'*)
then have *C'* ∈ *set*(*clinit-classes frs*) **using** *f' Cons* **by** *simp*
then obtain *sfs* **where** *shC'*: *sh C'* = *Some*(*sfs*, *Processing*) **and** *is-class P C'*
using *confc* **by**(*auto simp*: *conf-clinit-def*)
then have *C'eq*: *C'* = *fst*(*method P C' clinit*) **using** *wf-sees-clinit wtp*
by(*fastforce simp*: *is-class-def wf-jvm-prog-phi-def*)
then show *classes-above P C'* ⊆ *cset* ∧ *classes-above P* (*fst*(*method P C' clinit*)) ⊆ *cset*
using *sheap shC'* **by** *auto*
qed
show *?thesis* **using** *Cons assms*
proof(*cases xp*)
case *None*
{ assume *ics*: *ics* = *Called* [] ∨ *ics* = *No-ics*
then have [*simp*]: *JVMexec-ncollect P* (*xp*,*h*,*frs*,*sh*)
= *JVMinstr-ncollect P ?i h stk* ∪ *classes-above P C'*
∪ *classes-above-frames P frs* ∪ *classes-above-xcpts P*
and [*simp*]: *JVMexec-scollect P* (*xp*,*h*,*frs*,*sh*) = *JVMinstr-scollect P ?i*
using *f' None Cons* **by** *auto*
have *?thesis* **using** *assms*
proof(*cases ?i*)
case (*New C*)
then have *classes-above P C* ⊆ *cset* **using** *ics New assms* **by** *simp*
then show *?thesis* **using** *New xcpts frames smart f'* **by** *auto*
next
case (*Getfield F C*) **show** *?thesis*
proof(*cases hd stk* = *Null*)
case *True* **then show** *?thesis* **using** *Getfield assms* **by** *simp*
next
case *False*
let *?C* = *cname-of h* (*the-Addr* (*hd stk*))
have *above-stk*: *classes-above P ?C* ⊆ *cset*
using *stack-safe heap f' False Cons Getfield* **by**(*auto dest!*: *is-ptrD*) *blast*
then show *?thesis* **using** *Getfield assms* **by** *auto*
qed
next
case (*Getstatic C F D*)
show *?thesis*
proof(*cases* ∃ *t*. *P* ⊢ *C has F*,*Static*:*t in D*)
case *True*
then have *above-D*: *classes-above P D* ⊆ *cset* **using** *ics init-class-prom Getstatic* **by** *simp*
have *sub*: *P* ⊢ *C* ⪯* *D* **using** *has-field-decl-above True* **by** *blast*
then have *above-C*: *classes-between P C D* − {*D*} ⊆ *cset*
using *True Getstatic above-D smart f'* **by** *simp*
then have *classes-above P C* ⊆ *cset*

using *classes-above-sub-classes-between-eq*[*OF sub*] *above-D above-C* **by** *auto*
then show *?thesis* **using** *Getstatic assms* **by** *auto*
next
case *False* **then show** *?thesis* **using** *Getstatic assms* **by** *auto*
qed
next
case (*Putfield F C*) **show** *?thesis*
proof(*cases hd*(*tl stk*) = *Null*)
case *True* **then show** *?thesis* **using** *Putfield assms* **by** *simp*
next
case *False*
let *?C* = *cname-of h* (*the-Addr* (*hd* (*tl stk*)))
have *above-stk*: *classes-above P ?C* ⊆ *cset*
using *stack-safe heap f′ False Cons Putfield* **by**(*auto dest*!: *is-ptrD*) *blast*
then show *?thesis* **using** *Putfield assms* **by** *auto*
qed
next
case (*Putstatic C F D*)
show *?thesis*
proof(*cases* ∃ *t*. *P* ⊢ *C has F*,*Static*:*t in D*)
case *True*
then have *above-D*: *classes-above P D* ⊆ *cset* **using** *ics init-class-prom Putstatic* **by** *simp*
have *sub*: *P* ⊢ *C* ⪯* *D* **using** *has-field-decl-above True* **by** *blast*
then have *above-C*: *classes-between P C D* − {*D*} ⊆ *cset*
using *True Putstatic above-D smart f′* **by** *simp*
then have *classes-above P C* ⊆ *cset*
using *classes-above-sub-classes-between-eq*[*OF sub*] *above-D above-C* **by** *auto*
then show *?thesis* **using** *Putstatic assms* **by** *auto*
next
case *False* **then show** *?thesis* **using** *Putstatic assms* **by** *auto*
qed
next
case (*Checkcast C*) **show** *?thesis*
proof(*cases hd stk* = *Null*)
case *True* **then show** *?thesis* **using** *Checkcast assms* **by** *simp*
next
case *False*
let *?C* = *cname-of h* (*the-Addr* (*hd stk*))
have *above-stk*: *classes-above P ?C* ⊆ *cset*
using *stack-safe heap False Cons f′ Checkcast* **by**(*auto dest*!: *is-ptrD*) *blast*
then show *?thesis* **using** *above-stk Checkcast assms* **by**(*cases hd stk* = *Null*, *auto*)
qed
next
case (*Invoke M n*) **show** *?thesis*

proof(*cases stk* ! *n* = *Null*)
case *True* **then show** *?thesis* **using** *Invoke assms* **by** *simp*
next
case *False*
let *?C* = *cname-of h* (*the-Addr* (*stk* ! *n*))
have *above-stk*: *classes-above P ?C* ⊆ *cset* **using** *stack-safe heap False Cons f′ Invoke*
by(*auto dest*!: *is-ptrD*) *blast*
then show *?thesis* **using** *Invoke assms* **by** *auto*
qed
next
case (*Invokestatic C M n*)
let *?D* = *fst* (*method P C M*)
show *?thesis*
proof(*cases* ∃ *Ts T m D*. *P* ⊢ *C sees M*,*Static*:*Ts* → *T* = *m in D*)
case *True*
then have *above-D*: *classes-above P ?D* ⊆ *cset* **using** *ics init-class-prom Invokestatic*
by(*simp add*: *seeing-class-def*)
have *sub*: *P* ⊢ *C* $\preceq^*$ *?D* **using** *method-def2 sees-method-decl-above True* **by** *auto*
then show *?thesis*
proof(*cases C* = *?D*)
case *True* **then show** *?thesis*
using *Invokestatic above-D xcpts frames smart f′* **by** *auto*
next
case *False*
then have *above-C*: *classes-between P C ?D* − {*?D*} ⊆ *cset*
using *True Invokestatic above-D smart f′* **by** *simp*
then have *classes-above P C* ⊆ *cset*
using *classes-above-sub-classes-between-eq*[*OF sub*] *above-D above-C* **by** *auto*
then show *?thesis* **using** *Invokestatic assms* **by** *auto*
qed
next
case *False* **then show** *?thesis* **using** *Invokestatic assms* **by** *auto*
qed
next
case *Throw* **show** *?thesis*
proof(*cases hd stk* = *Null*)
case *True* **then show** *?thesis* **using** *Throw assms* **by** *simp*
next
case *False*
let *?C* = *cname-of h* (*the-Addr* (*hd stk*))
have *above-stk*: *classes-above P ?C* ⊆ *cset*
using *stack-safe heap False Cons f′ Throw* **by**(*auto dest*!: *is-ptrD*) *blast*
then show *?thesis* **using** *above-stk Throw assms* **by** *auto*
qed
next

case *Load* **then show** *?thesis* **using** *assms* **by** *auto*
next
case *Store* **then show** *?thesis* **using** *assms* **by** *auto*
next
case *Push* **then show** *?thesis* **using** *assms* **by** *auto*
next
case *Goto* **then show** *?thesis* **using** *assms* **by** *auto*
next
case *IfFalse* **then show** *?thesis* **using** *assms* **by** *auto*
qed(*auto*)
}
moreover
{ fix *C1 Cs1* **assume** *ics*: *ics* = *Called* (*C1#Cs1*)
then have *?thesis* **using** *None Cons Called-prom*[*OF ics*] *xcpts frames f′* **by** *simp*
}
moreover
{ fix *Cs1 a* **assume** *ics*: *ics* = *Throwing Cs1 a*
then obtain *C fs* **where** *h a* = *Some*(*C*,*fs*) **using** *Throwing-ex* **by** *fastforce*
then have *above-stk*: *classes-above P* (*cname-of h a*) ⊆ *cset* **using** *heap* **by** *auto*
then have *?thesis* **using** *ics None Cons xcpts frames f′* **by** *simp*
}
moreover
{ fix *C1 Cs1* **assume** *ics*: *ics* = *Calling C1 Cs1*
then have *?thesis* **using** *None Cons Calling-prom*[*OF ics*] *xcpts frames f′* **by** *simp*
}
ultimately show *?thesis* **by** (*metis ics-classes.cases list.exhaust*)
qed(*simp*)
qed(*simp*)

— ... which is the same as *csmall*
lemma *jvm-naive-to-smart-csmall*:
assumes
— well-formedness
wtp: $wf\text{-}jvm\text{-}prog_{\Phi}$ *P*
and *correct*: *P*,Φ ⊢ (*xp*,*h*,*frs*,*sh*)√
— defs
and *f′*: *hd frs* = (*stk*,*loc*,*C′*,*M′*,*pc*,*ics*)
— backward promises - will be collected prior
and *heap*: ⋀*C fs*. ∃ *a*. *h a* = *Some*(*C*,*fs*) ⟹ *classes-above P C* ⊆ *cset*
and *sheap*: ⋀*C sfs i*. *sh C* = *Some*(*sfs*,*i*) ⟹ *classes-above P C* ⊆ *cset*
and *xcpts*: *classes-above-xcpts P* ⊆ *cset*
and *frames*: *classes-above-frames P frs* ⊆ *cset*
— forward promises - will be collected after if not already
and *init-class-prom*: ⋀*C*. *ics* = *Called* [] ∨ *ics* = *No-ics*
⟹ *coll-init-class P* (*instrs-of P C′ M′* ! *pc*) = *Some C* ⟹ *classes-above P C* ⊆ *cset*

and *Calling-prom*: $\bigwedge C'\ Cs'.\ ics = Calling\ C'\ Cs' \Longrightarrow classes\text{-}above\ P\ C' \subseteq cset$
— collections
and *smart-coll*: $(\sigma', cset_s) \in JVMSmartCollectionSemantics.csmall\ P\ (xp,h,frs,sh)$
and *naive-coll*: $(\sigma', cset_n) \in JVMNaiveCollectionSemantics.csmall\ P\ (xp,h,frs,sh)$
and *smart*: $cset_s \subseteq cset$
shows $cset_n \subseteq cset$
using *jvm-naive-to-smart-exec-collect*[**where** *h=h* **and** *sh=sh*, *OF assms*(*1−9*)]
smart smart-coll naive-coll
by(*fastforce simp*: *JVMNaiveCollectionSemantics.csmall-def*
JVMSmartCollectionSemantics.csmall-def)

— ...which means over *csmall-nstep*, stepping from the end state (the point by which future promises will have been fulfilled) (uses backward and forward promise lemmas)
lemma *jvm-naive-to-smart-csmall-nstep*:
⟦ $wf\text{-}jvm\text{-}prog_\Phi\ P$;
$P,\Phi \vdash (xp,h,frs,sh)\surd$;
$hd\ frs = (stk,loc,C',M',pc,ics)$;
$\bigwedge C\ fs.\ \exists a.\ h\ a = Some(C,fs) \Longrightarrow classes\text{-}above\ P\ C \subseteq cset$;
$\bigwedge C\ sfs\ i.\ sh\ C = Some(sfs,i) \Longrightarrow classes\text{-}above\ P\ C \subseteq cset$;
$classes\text{-}above\text{-}xcpts\ P \subseteq cset$;
$classes\text{-}above\text{-}frames\ P\ frs \subseteq cset$;
$\bigwedge C.\ ics = Called\ [] \vee ics = No\text{-}ics$
$\Longrightarrow coll\text{-}init\text{-}class\ P\ (instrs\text{-}of\ P\ C'\ M'\ !\ pc) = Some\ C \Longrightarrow classes\text{-}above\ P\ C \subseteq cset$;
$\bigwedge C'\ Cs'.\ ics = Calling\ C'\ Cs' \Longrightarrow classes\text{-}above\ P\ C' \subseteq cset$;
$(\sigma', cset_n) \in JVMNaiveCollectionSemantics.csmall\text{-}nstep\ P\ (xp,h,frs,sh)\ n$;
$(\sigma', cset_s) \in JVMSmartCollectionSemantics.csmall\text{-}nstep\ P\ (xp,h,frs,sh)\ n$;
$cset_s \subseteq cset$;
$\sigma' \in JVMendset$ ⟧
$\Longrightarrow cset_n \subseteq cset$
proof(*induct n arbitrary*: *xp h frs sh stk loc* $C'\ M'$ *pc ics* $\sigma'\ cset_n\ cset_s$ *cset*)
case *0* **then show** *?case*
using *JVMNaiveCollectionSemantics.csmall-nstep-base subsetI old.prod.inject singletonD*
by (*metis* (*no-types*, *lifting*) *equals0D*)
next
case (*Suc n1*)
let $?\sigma = (xp,h,frs,sh)$
obtain $\sigma 1$ *cset1* $cset'$ **where** $\sigma 1$: $(\sigma 1, cset1) \in JVMNaiveCollectionSemantics.csmall\ P\ ?\sigma$
$cset_n = cset1 \cup cset'$ $(\sigma', cset') \in JVMNaiveCollectionSemantics.csmall\text{-}nstep\ P\ \sigma 1\ n1$
using *JVMNaiveCollectionSemantics.csmall-nstep-SucD*[*OF Suc.prems*(*10*)] **by** *clarsimp+*
obtain $\sigma 1'$ $cset1'$ $cset''$ **where** $\sigma 1'$: $(\sigma 1', cset1') \in JVMSmartCollectionSemantics.csmall\ P\ ?\sigma$
$cset_s = cset1' \cup cset''$ $(\sigma', cset'') \in JVMSmartCollectionSemantics.csmall\text{-}nstep\ P\ \sigma 1'\ n1$

using *JVMSmartCollectionSemantics.csmall-nstep-SucD*[*OF Suc.prems*(*11*)] **by** *clarsimp+*
have *σ-eq*: *σ1* = *σ1′* **using** *σ1*(*1*) *σ1′*(*1*) **by**(*simp add*: *JVMNaiveCollectionSemantics.csmall-def*
JVMSmartCollectionSemantics.csmall-def)
have *sub1′*: *cset1′* ⊆ *cset* **and** *sub′′*: *cset′′* ⊆ *cset* **using** *Suc.prems*(*12*) *σ1′*(*2*) **by** *auto*
then have *sub1*: *cset1* ⊆ *cset*
using *jvm-naive-to-smart-csmall*[**where** *h*=*h* **and** *sh*=*sh* **and** *σ′*=*σ1*, *OF Suc.prems*(*1*−*9*) - - *sub1′*]
Suc.prems(*11*,*12*) *σ1*(*1*) *σ1′*(*1*) *σ-eq* **by** *fastforce*
show *?case*
proof(*cases n1*)
case *0* **then show** *?thesis* **using** *σ1*(*2*,*3*) *sub1* **by** *auto*
next
case *Suc2*: (*Suc n2*)
then have *nend1*: *σ1* ∉ *JVMendset*
using *JVMNaiveCollectionSemantics.csmall-nstep-Suc-nend σ1*(*3*) **by** *blast*
obtain *xp1 h1 frs1 sh1* **where** *σ1-case* [*simp*]: *σ1* = (*xp1*,*h1*,*frs1*,*sh1*) **by**(*cases σ1*)
obtain *stk1 loc1 C1′ M1′ pc1 ics1* **where** *f1′*: *hd frs1* = (*stk1*,*loc1*,*C1′*,*M1′*,*pc1*,*ics1*)
by(*cases hd frs1*)
then obtain *frs1′* **where** [*simp*]: *frs1* = (*stk1*,*loc1*,*C1′*,*M1′*,*pc1*,*ics1*)#*frs1′*
using *JVMendset-def nend1* **by**(*cases frs1*, *auto*)
have *cbig1*: (*σ′*, *cset′*) ∈ *JVMNaiveCollectionSemantics.cbig P σ1*
(*σ′*, *cset′′*) ∈ *JVMSmartCollectionSemantics.cbig P σ1* **using** *σ1*(*3*) *σ1′*(*3*) *Suc.prems*(*13*) *σ-eq*
using *JVMNaiveCollectionSemantics.cbig-def2*
JVMSmartCollectionSemantics.cbig-def2 **by** *blast+*
obtain *σ2′ cset2′ cset2′′* **where** *σ2′*: (*σ2′*, *cset2′*) ∈ *JVMSmartCollectionSemantics.csmall P σ1*
cset′′ = *cset2′* ∪ *cset2′′* (*σ′*, *cset2′′*) ∈ *JVMSmartCollectionSemantics.csmall-nstep P σ2′ n2*
using *JVMSmartCollectionSemantics.csmall-nstep-SucD σ1′*(*3*) *Suc2 σ-eq* **by** *blast*

have *wtp*: *wf-jvm-prog*$_\Phi$ *P* **by** *fact*
let *?i1* = *instrs-of P C1′ M1′* ! *pc1*
let *?ics1* = *ics-of* (*hd* (*frames-of σ1*))
have *step*: *P* ⊢ (*xp*,*h*,*frs*,*sh*) −*jvm*→ (*xp1*,*h1*,*frs1*,*sh1*)
proof −
have *exec* (*P*, *?σ*) = ⌊*σ1′*⌋ **using** *JVMsmart-csmallD*[*OF σ1′*(*1*)] **by** *simp*
then have *P* ⊢ *?σ* −*jvm*→ *σ1′* **using** *jvm-one-step1*[*OF exec-1.exec-1I*] **by** *simp*
then show *?thesis* **using** *Suc.prems*(*12*) *σ-eq* **by** *fastforce*
qed
have *correct1*: *P*,Φ ⊢ (*xp1*,*h1*,*frs1*,*sh1*)√ **by**(*rule BV-correct*[*OF wtp step Suc.prems*(*2*)])

have *vics1*: *P*,*h1*,*sh1* $\vdash_i$ (*C1′*, *M1′*, *pc1*, *ics1*)
using *correct1 Suc.prems*(*7*) **by**(*auto simp*: *conf-f-def2*)
from *correct1* **obtain** *b Ts T mxs* mxl_0 *ins xt ST LT* **where**
meth1: *P* ⊢ *C1′ sees M1′*,*b*:*Ts*→*T*=(*mxs*,mxl_0,*ins*,*xt*) *in C1′* **and**
pc1: *pc1* < *length ins* **and**
Φ-*pc1*: Φ *C1′ M1′*!*pc1* = *Some* (*ST*,*LT*) **by**(*auto dest*: *sees-method-fun*)
then have *wt1*: *P*,*T*,*mxs*,*size ins*,*xt* ⊢ *ins*!*pc1*,*pc1* :: Φ *C1′ M1′*
using *wt-jvm-prog-impl-wt-instr*[*OF wtp meth1*] **by** *fast*

have ⋀*a C fs sfs′ i′*. (*h1 a* = ⌊(*C*, *fs*)⌋ ⟶ *classes-above P C* ⊆ *cset*) ∧
(*sh1 C* = ⌊(*sfs′*, *i′*)⌋ ⟶ *classes-above P C* ⊆ *cset*) ∧
classes-above-frames P frs1 ⊆ *cset*
proof –
fix *a C fs sfs′ i′*
show (*h1 a* = ⌊(*C*, *fs*)⌋ ⟶ *classes-above P C* ⊆ *cset*) ∧
(*sh1 C* = ⌊(*sfs′*, *i′*)⌋ ⟶ *classes-above P C* ⊆ *cset*) ∧
(*classes-above-frames P frs1* ⊆ *cset*)
using *Suc.prems*(*11*–*12*) *σ1′ σ-eq*[*THEN sym*] *JVMsmart-csmallD*[*OF σ1′*(*1*)]
backward-coll-promises-kept[**where** *h*=*h* **and** *xp*=*xp* **and** *sh*=*sh* **and** *frs*=*frs*
and *frs′*=*frs1*
and *xp′*=*xp1* **and** *h′*=*h1* **and** *sh′*=*sh1*, *OF Suc.prems*(*1*–*9*)] **by** *auto*
qed
then have *heap1*: ⋀*C fs*. ∃ *a*. *h1 a* = *Some*(*C*,*fs*) ⟹ *classes-above P C* ⊆ *cset*
and *sheap1*: ⋀*C sfs i*. *sh1 C* = *Some*(*sfs*,*i*) ⟹ *classes-above P C* ⊆ *cset*
and *frames1*: *classes-above-frames P frs1* ⊆ *cset* **by** *blast*+
have *xcpts1*: *classes-above-xcpts P* ⊆ *cset* **using** *Suc.prems*(*6*) **by** *auto*
— *init-class* promise
have *sheap2*: ⋀*C*. *coll-init-class P ?i1* = *Some C*
⟹ ∀ *C′*. *P* ⊢ *C* ⪯* *C′* ⟶ (∃ *sfs i*. *sheap σ1 C′* = ⌊(*sfs*, *i*)⌋)
⟶ *classes-above P C′* ⊆ *cset* **using** *sheap1* **by** *auto*
have *called*: ⋀*C*. *coll-init-class P ?i1* = *Some C*
⟹ *ics-of* (*hd* (*frames-of σ1*)) = *Called* [] ⟹ *classes-above P C* ⊆ *cset*
proof –
fix *C* **assume** *cic*: *coll-init-class P ?i1* = *Some C* **and**
ics: *ics-of* (*hd* (*frames-of σ1*)) = *Called* []
then obtain *sobj* **where** *sh1 C* = *Some sobj* **using** *vics1 f1′*
by(*cases ?i1*, *auto simp*: *seeing-class-def split*: *if-split-asm*)
then show *classes-above P C* ⊆ *cset* **using** *sheap1* **by**(*cases sobj*, *simp*)
qed
have *init-class-prom1*: ⋀*C*. *ics1* = *Called* [] ∨ *ics1* = *No-ics*
⟹ *coll-init-class P ?i1* = *Some C* ⟹ *classes-above P C* ⊆ *cset*
proof –
fix *C* **assume** *ics1* = *Called* [] ∨ *ics1* = *No-ics* **and** *cic*: *coll-init-class P ?i1*
= *Some C*
then have *ics*: *?ics1* = *Called* [] ∨ *?ics1* = *No-ics* **using** *f1′* **by** *simp*
then show *classes-above P C* ⊆ *cset* **using** *cic*
proof(*cases ?ics1* = *Called* [])
case *True* **then show** *?thesis* **using** *cic called* **by** *simp*
next

case *False*
then have *ics'*: *?ics1* = *No-ics* **using** *ics* **by** *simp*
then show *?thesis* **using** *cic*
proof(*cases ?i1*)
case (*New C1*)
then have *is-class P C1* **using** Φ*-pc1 wt1 meth1* **by** *auto*
then have $P \vdash C1 \preceq^* Object$ **using** *wtp is-class-is-subcls*
by(*auto simp*: *wf-jvm-prog-phi-def*)
then show *?thesis* **using** *New-collects*[*OF - cbig1*(*2*) *nend1 - ics' sheap2 sub''*]
f1' ics cic New **by** *auto*
next
case (*Getstatic C1 F1 D1*)
then obtain *t* **where** *mC1*: $P \vdash C1$ *has F1*,*Static*:*t in D1* **and** *eq*: *C* = *D1*
using *cic* **by** (*metis coll-init-class.simps*(*2*) *option.inject option.simps*(*3*))
then have *is-class P C* **using** *has-field-is-class'*[*OF mC1*] **by** *simp*
then have $P \vdash C \preceq^* Object$ **using** *wtp is-class-is-subcls*
by(*auto simp*: *wf-jvm-prog-phi-def*)
then show *?thesis* **using** *Getstatic-collects*[*OF - cbig1*(*2*) *nend1 - ics' - sheap2 sub''*]
eq f1' Getstatic ics cic **by** *fastforce*
next
case (*Putstatic C1 F1 D1*)
then obtain *t* **where** *mC1*: $P \vdash C1$ *has F1*,*Static*:*t in D1* **and** *eq*: *C* = *D1*
using *cic* **by** (*metis coll-init-class.simps*(*3*) *option.inject option.simps*(*3*))
then have *is-class P C* **using** *has-field-is-class'*[*OF mC1*] **by** *simp*
then have $P \vdash C \preceq^* Object$ **using** *wtp is-class-is-subcls*
by(*auto simp*: *wf-jvm-prog-phi-def*)
then show *?thesis* **using** *Putstatic-collects*[*OF - cbig1*(*2*) *nend1 - ics' - sheap2 sub''*]
eq f1' Putstatic ics cic **by** *fastforce*
next
case (*Invokestatic C1 M1 n'*)
then obtain *Ts T m* **where** *mC*: $P \vdash C1$ *sees M1*, *Static* : $Ts \rightarrow T$ = *m in C*
using *cic* **by**(*fastforce simp*: *seeing-class-def split*: *if-split-asm*)
then have *is-class P C* **by**(*rule sees-method-is-class'*)
then have *Obj*: $P \vdash C \preceq^* Object$ **using** *wtp is-class-is-subcls*
by(*auto simp*: *wf-jvm-prog-phi-def*)
show *?thesis* **using** *Invokestatic-collects*[*OF - cbig1*(*2*) *sub'' nend1 - ics' mC sheap2*]
Obj mC f1' Invokestatic ics cic **by** *auto*
qed(*simp+*)
qed
qed
— Calling promise
have *Calling-prom1*: $\bigwedge C'\ Cs'.$ *ics1* = *Calling C' Cs'* $\Longrightarrow$ *classes-above P C'* $\subseteq$

cset
  proof –
   fix *C′ Cs′* **assume** *ics*: *ics1* = *Calling C′ Cs′*
   then have *is-class P C′* **using** *vics1* **by** *simp*
   then have *obj*: *P* ⊢ *C′* ⪯* *Object* **using** *wtp is-class-is-subcls*
    by(*auto simp*: *wf-jvm-prog-phi-def*)
   have *sheap3*: ∀ *C1*. *P* ⊢ *C′* ⪯* *C1* ⟶ (∃ *sfs i*. *sheap σ1 C1* = ⌊(*sfs*, *i*)⌋)
    ⟶ *classes-above P C1* ⊆ *cset* **using** *sheap1* **by** *auto*
   show *classes-above P C′* ⊆ *cset*
    using *Calling-collects*[*OF obj cbig1*(*2*) *nend1* - *sheap3 sub″*] *ics f1′* **by** *simp*
  qed
 have *in-naive*: (*σ′*, *cset′*) ∈ *JVMNaiveCollectionSemantics.csmall-nstep P* (*xp1*, *h1*, *frs1*, *sh1*) *n1*
   and *in-smart*: (*σ′*, *cset″*) ∈ *JVMSmartCollectionSemantics.csmall-nstep P* (*xp1*, *h1*, *frs1*, *sh1*) *n1*
   using *σ1*(*3*) *σ1′*(*3*) *σ-eq*[*THEN sym*] **by** *simp*+
  have *sub2*: *cset′* ⊆ *cset*
 by(*rule Suc.hyps*[*OF wtp correct1 f1′ heap1 sheap1 xcpts1 frames1 init-class-prom1*
      Calling-prom1 in-naive in-smart sub″ Suc.prems(*13*)]) *simp-all*
  then show *?thesis* **using** *σ1*(*2*) *σ1′*(*2*) *sub1 sub2* **by** *fastforce*
 qed
qed

— ...which means over *cbig*
lemma *jvm-naive-to-smart-cbig*:
assumes
— well-formedness
  wtp: $wf\text{-}jvm\text{-}prog_{\Phi}$ *P*
 and *correct*: *P*,Φ ⊢ (*xp*,*h*,*frs*,*sh*)√
— defs
 and *f′*: *hd frs* = (*stk*,*loc*,*C′*,*M′*,*pc*,*ics*)
— backward promises - will be collected/maintained prior
 and *heap*: ⋀*C fs*. ∃ *a*. *h a* = *Some*(*C*,*fs*) ⟹ *classes-above P C* ⊆ *cset*
 and *sheap*: ⋀*C sfs i*. *sh C* = *Some*(*sfs*,*i*) ⟹ *classes-above P C* ⊆ *cset*
 and *xcpts*: *classes-above-xcpts P* ⊆ *cset*
 and *frames*: *classes-above-frames P frs* ⊆ *cset*
— forward promises - will be collected after if not already
 and *init-class-prom*: ⋀*C*. *ics* = *Called* [] ∨ *ics* = *No-ics*
  ⟹ *coll-init-class P* (*instrs-of P C′ M′* ! *pc*) = *Some C* ⟹ *classes-above P C* ⊆ *cset*
 and *Calling-prom*: ⋀*C′ Cs′*. *ics* = *Calling C′ Cs′* ⟹ *classes-above P C′* ⊆ *cset*
— collections
 and *n*: (*σ′*, $cset_n$) ∈ *JVMNaiveCollectionSemantics.cbig P* (*xp*,*h*,*frs*,*sh*)
 and *s*: (*σ′*, $cset_s$) ∈ *JVMSmartCollectionSemantics.cbig P* (*xp*,*h*,*frs*,*sh*)
 and *smart*: $cset_s$ ⊆ *cset*
shows $cset_n$ ⊆ *cset*
proof –
 let *?σ* = (*xp*,*h*,*frs*,*sh*)
 have *nend*: *σ′* ∈ *JVMendset* **using** *n* **by**(*simp add*: *JVMNaiveCollectionSeman-*

tics.cbig-def)
obtain *n* **where** *n'*: (σ', $cset_n$) ∈ *JVMNaiveCollectionSemantics.csmall-nstep P ?σ n σ'* ∈ *JVMendset*
using *JVMNaiveCollectionSemantics.cbig-def2 n* **by** *auto*
obtain *s* **where** *s'*: (σ', $cset_s$) ∈ *JVMSmartCollectionSemantics.csmall-nstep P ?σ s σ'* ∈ *JVMendset*
using *JVMSmartCollectionSemantics.cbig-def2 s* **by** *auto*
have *n=s* **using** *jvm-naive-to-smart-csmall-nstep-last-eq*[*OF n n'*(*1*) *s'*(*1*)] **by** *simp*
then have *sn*: (σ', $cset_s$) ∈ *JVMSmartCollectionSemantics.csmall-nstep P ?σ n* **using** *s'*(*1*) **by** *simp*
then show *?thesis*
using *jvm-naive-to-smart-csmall-nstep*[*OF assms*(*1−9*) *n'*(*1*) *sn assms*(*12*) *nend*] **by** *fast*
qed

— ...thus naive ⊆ smart over the out function, since all conditions will be met - and promises kept - by the defined starting point
lemma *jvm-naive-to-smart-collection*:
assumes *naive*: (σ',$cset_n$) ∈ *jvm-naive-out P t* **and** *smart*: (σ',$cset_s$) ∈ *jvm-smart-out P t*
and *P*: *P* ∈ *jvm-progs* **and** *t*: *t* ∈ *jvm-tests*
shows $cset_n \subseteq cset_s$
proof −
let *?P* = *jvm-make-test-prog P t*
let *?σ* = *start-state* (*t#P*)
let *?i* = *instrs-of ?P Start start-m* ! *0* **and** *?ics* = *No-ics*
obtain *xp h frs sh* **where**
[*simp*]: *?σ* = (*xp*,*h*,*frs*,*sh*) **and**
[*simp*]: *h* = *start-heap* (*t#P*) **and**
[*simp*]: *frs* = [([], [], *Start*, *start-m*, *0*, *No-ics*)] **and**
[*simp*]: *sh* = *start-sheap*
by(*clarsimp simp*: *start-state-def*)

from *P t* **have** *nS*: ¬ *is-class* (*t # P*) *Start*
by(*simp add*: *is-class-def class-def Start-def Test-def*)
from *P* **have** *nT*: ¬ *is-class P Test* **by** *simp*
from *P t* **obtain** *m* **where** *tPm*: *t # P* ⊢ (*fst t*) *sees main*, *Static* : []→*Void* = *m in* (*fst t*)
by *auto*
have *nclinit*: *main* ≠ *clinit* **by**(*simp add*: *main-def clinit-def*)
have *Objp*: ⋀*b' Ts' T' m' D'*.
t#P ⊢ *Object sees start-m*, *b'* : *Ts'*→*T'* = *m' in D'* ⟹ *b'* = *Static* ∧ *Ts'* = [] ∧ *T'* = *Void*
proof −
fix *b' Ts' T' m' D'*
assume *mObj*: *t#P* ⊢ *Object sees start-m*, *b'* : *Ts'*→*T'* = *m' in D'*
from *P* **have** *ot-nsub*: ¬ *P* ⊢ *Object* ⪯* *Test*
by(*clarsimp simp*: *wf-jvm-prog-def wf-jvm-prog-phi-def*)

from *class-add-sees-method-rev*[*OF - ot-nsub*] *mObj t*
have $P \vdash$ *Object sees start-m*, b' : $Ts' \rightarrow T'$ = m' *in* D' **by**(*cases t*, *auto*)
with *P jvm-progs-def* **show** b' = *Static* $\wedge$ Ts' = [] $\wedge$ T' = *Void* **by** *blast*
qed
from *P t* **obtain** Φ **where** *wtp0*: *wf-jvm-prog*$_\Phi$ (*t#P*) **by**(*auto simp*: *wf-jvm-prog-def*)
let *?*Φ' = $\lambda C f$. *if* C = *Start* $\wedge$ (f = *start-m* $\vee$ f = *clinit*) *then start-*φ_m *else* Φ $C f$
from *wtp0* **have** *wtp*: *wf-jvm-prog*$_{?\Phi'}$ *?P*
proof –
note *wtp0 nS tPm nclinit*
moreover obtain $\bigwedge C$. $C \neq$ *Start* $\Longrightarrow$ *?*Φ' C = Φ C *?*Φ' *Start start-m* = *start-*φ_m
?Φ' *Start clinit* = *start-*φ_m **by** *simp*
moreover note *Objp*
ultimately show *?thesis* **by**(*rule start-prog-wf-jvm-prog-phi*)
qed
have *cic*: *coll-init-class ?P ?i* = *Some Test*
proof –
from *wtp0* **obtain** *wf-md* **where** *wf*: *wf-prog wf-md* (*t#P*)
by(*clarsimp dest*!: *wt-jvm-progD*)
with *start-prog-start-m-instrs t* **have** *i*: *?i* = *Invokestatic Test main 0* **by** *simp*
from *jvm-make-test-prog-sees-Test-main*[*OF P t*] **obtain** *m* **where**
?P $\vdash$ *Test sees main*, *Static* : $[] \rightarrow$ *Void* = *m in Test* **by** *fast*
with *t* **have** *seeing-class* (*jvm-make-test-prog P t*) *Test main* = $\lfloor$*Test*$\rfloor$
by(*cases m*, *fastforce simp*: *seeing-class-def*)
with *i* **show** *?thesis* **by** *simp*
qed
— well-formedness
note *wtp*
moreover have *correct*: *?P*,*?*Φ' $\vdash$ (*xp*,*h*,*frs*,*sh*)$\surd$
proof –
note *wtp0 nS tPm nclinit*
moreover have *?*Φ' *Start start-m* = *start-*φ_m **by** *simp*
ultimately have *?P*,*?*Φ' $\vdash$ *?*$\sigma\surd$ **by**(*rule BV-correct-initial*)
then show *?thesis* **by** *simp*
qed
— defs
moreover have *hd frs* = ([], [], *Start*, *start-m*, *0*, *No-ics*) **by** *simp*
— backward promises
moreover from *jvm-smart-out-classes-above-start-heap*[*OF smart - P t*]
have *heap*: $\bigwedge C$ *fs*. $\exists$ *a*. *h a* = *Some*(*C*,*fs*) $\Longrightarrow$ *classes-above ?P C* $\subseteq$ $cset_s$ **by** *auto*
moreover from *jvm-smart-out-classes-above-start-sheap*[*OF smart*]
have *sheap*: $\bigwedge C$ *sfs i*. *sh C* = *Some*(*sfs*,*i*) $\Longrightarrow$ *classes-above ?P C* $\subseteq$ $cset_s$ **by** *simp*
moreover from *jvm-smart-out-classes-above-xcpts*[*OF smart P t*]
have *xcpts*: *classes-above-xcpts ?P* $\subseteq$ $cset_s$ **by** *simp*
moreover from *jvm-smart-out-classes-above-frames*[*OF smart*]
have *frames*: *classes-above-frames ?P frs* $\subseteq$ $cset_s$ **by** *simp*

— forward promises - will be collected after if not already
  moreover from *jvm-smart-out-classes-above-Test*[*OF smart P t*] *cic*
  have *init-class-prom*: $\bigwedge C$. *?ics = Called* [] ∨ *?ics = No-ics*
    ⟹ *coll-init-class ?P ?i = Some C* ⟹ *classes-above ?P C* ⊆ $cset_s$ **by** *simp*
  moreover have $\bigwedge C'$ *Cs'*. *?ics = Calling C' Cs'* ⟹ *classes-above ?P C'* ⊆ $cset_s$
by *simp*
— collections
  moreover from *naive*
   have *n*: (σ', $cset_n$) ∈ *JVMNaiveCollectionSemantics.cbig ?P* (*xp*,*h*,*frs*,*sh*) **by**
simp
  moreover from *smart* **obtain** $cset_s'$ **where**
    s: (σ', $cset_s'$) ∈ *JVMSmartCollectionSemantics.cbig ?P* (*xp*,*h*,*frs*,*sh*) **and**
      $cset_s' \subseteq cset_s$
   by *clarsimp*
  ultimately show $cset_n \subseteq cset_s$ **by**(*rule jvm-naive-to-smart-cbig*; *simp*)
qed

12.6.4 Proving smart ⊆ naive

We prove that *jvm-naive* collects everything *jvm-smart* does. Combined with the other direction, this shows that the naive and smart algorithms collect the same set of classes.

lemma *jvm-smart-to-naive-exec-collect*:
  JVMexec-scollect P σ ⊆ *JVMexec-ncollect P* σ
proof –
  obtain *xp h frs sh* **where** σ: σ=(*xp*,*h*,*frs*,*sh*) **by**(*cases* σ)
  then show *?thesis*
  proof(*cases* ∃ *x*. *xp = Some x* ∨ *frs* = [])
    case *False*
    then obtain *stk loc C M pc ics frs'*
     where *none*: *xp = None* **and** *frs*: *frs*=(*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs'*
      by(*cases xp*, *auto*, *cases frs*, *auto*)
    have *instr-case*: *ics = Called* [] ∨ *ics = No-ics* ⟹ *?thesis*
    proof –
      assume *ics*: *ics = Called* [] ∨ *ics = No-ics*
      then show *?thesis* **using** σ *none frs*
      proof(*cases curr-instr P* (*stk*,*loc*,*C*,*M*,*pc*,*ics*)) **qed**(*auto split*: *if-split-asm*)
    qed
    then show *?thesis* **using** σ *none frs*
    proof(*cases ics*)
     case(*Called Cs*) **then show** *?thesis* **using** *instr-case* σ *none frs* **by**(*cases Cs*,
auto)
    qed(*auto*)
  qed(*auto*)
qed

lemma *jvm-smart-to-naive-csmall*:
assumes (σ', $cset_n$) ∈ *JVMNaiveCollectionSemantics.csmall P* σ
    and (σ', $cset_s$) ∈ *JVMSmartCollectionSemantics.csmall P* σ

shows *cset*$_s$ ⊆ *cset*$_n$
using *jvm-smart-to-naive-exec-collect assms*
by(*auto simp*: *JVMNaiveCollectionSemantics.csmall-def*
JVMSmartCollectionSemantics.csmall-def)

lemma *jvm-smart-to-naive-csmall-nstep*:
⟦ (σ′, *cset*$_n$) ∈ *JVMNaiveCollectionSemantics.csmall-nstep P* σ *n*;
(σ′, *cset*$_s$) ∈ *JVMSmartCollectionSemantics.csmall-nstep P* σ *n* ⟧
⟹ *cset*$_s$ ⊆ *cset*$_n$
proof(*induct n arbitrary*: σ σ′ *cset*$_n$ *cset*$_s$)
case (*Suc n*′)
obtain σ*1 cset1 cset*′ **where** σ*1*: (σ*1*, *cset1*) ∈ *JVMNaiveCollectionSemantics.csmall P* σ
cset$_n$ = *cset1* ∪ *cset*′ (σ′, *cset*′) ∈ *JVMNaiveCollectionSemantics.csmall-nstep P* σ*1 n*′
using *JVMNaiveCollectionSemantics.csmall-nstep-SucD* [*OF Suc.prems*(*1*)] **by** *clarsimp+*
obtain σ*1*′ *cset1*′ *cset*″ **where** σ*1*′: (σ*1*′, *cset1*′) ∈ *JVMSmartCollectionSemantics.csmall P* σ
cset$_s$ = *cset1*′ ∪ *cset*″ (σ′, *cset*″) ∈ *JVMSmartCollectionSemantics.csmall-nstep P* σ*1*′ *n*′
using *JVMSmartCollectionSemantics.csmall-nstep-SucD* [*OF Suc.prems*(*2*)] **by** *clarsimp+*
have σ*-eq*: σ*1* = σ*1*′ **using** σ*1*(*1*) σ*1*′(*1*) **by**(*simp add*: *JVMNaiveCollectionSemantics.csmall-def*
JVMSmartCollectionSemantics.csmall-def)
then have *sub1*: *cset1*′ ⊆ *cset1* **using** σ*1*(*1*) σ*1*′(*1*) *jvm-smart-to-naive-csmall* **by** *blast*
have *sub2*: *cset*″ ⊆ *cset*′ **using** σ*1*(*3*) σ*1*′(*3*) σ*-eq Suc.hyps* **by** *blast*
then show *?case* **using** σ*1*(*2*) σ*1*′(*2*) *sub1 sub2* **by** *blast*
qed(*simp*)

lemma *jvm-smart-to-naive-cbig*:
assumes *n*: (σ′, *cset*$_n$) ∈ *JVMNaiveCollectionSemantics.cbig P* σ
and *s*: (σ′, *cset*$_s$) ∈ *JVMSmartCollectionSemantics.cbig P* σ
shows *cset*$_s$ ⊆ *cset*$_n$
proof −
obtain *n* **where** *n*′: (σ′, *cset*$_n$) ∈ *JVMNaiveCollectionSemantics.csmall-nstep P* σ *n* σ′ ∈ *JVMendset*
using *JVMNaiveCollectionSemantics.cbig-def2 n* **by** *auto*
obtain *s* **where** *s*′: (σ′, *cset*$_s$) ∈ *JVMSmartCollectionSemantics.csmall-nstep P* σ *s* σ′ ∈ *JVMendset*
using *JVMSmartCollectionSemantics.cbig-def2 s* **by** *auto*
have *n=s* **using** *jvm-naive-to-smart-csmall-nstep-last-eq*[*OF n n*′(*1*) *s*′(*1*)] **by** *simp*
then show *?thesis* **using** *jvm-smart-to-naive-csmall-nstep n*′(*1*) *s*′(*1*) **by** *blast*
qed

lemma *jvm-smart-to-naive-collection*:

assumes *naive*: $(\sigma', cset_n) \in$ *jvm-naive-out P t* **and** *smart*: $(\sigma', cset_s) \in$ *jvm-smart-out P t*
and $P \in$ *jvm-progs* **and** $t \in$ *jvm-tests*
shows $cset_s \subseteq cset_n$
proof –
have *nend*: *start-state* $(t\#P) \notin$ *JVMendset* **by**(*simp add*: *JVMendset-def start-state-def*)
from *naive* **obtain** *n* **where**
nstep: $(\sigma', cset_n) \in$ *JVMNaiveCollectionSemantics.csmall-nstep*
(*jvm-make-test-prog P t*) (*start-state* $(t\#P)$) *n*
by(*auto dest*!: *JVMNaiveCollectionSemantics.cbigD*)
with *nend naive* **obtain** n' **where** n': $n = Suc\ n'$
by(*cases n*; *simp add*: *JVMNaiveCollectionSemantics.cbig-def*)
from *start-prog-classes-above-Start*
have *classes-above-frames* (*jvm-make-test-prog P t*) (*frames-of* (*start-state* $(t\#P)$)) $= \{Object, Start\}$
by(*simp add*: *start-state-def*)
with *nstep* n'
have *jvm-smart-collect-start* (*jvm-make-test-prog P t*) $\subseteq cset_n$
by(*auto simp*: *start-state-def JVMNaiveCollectionSemantics.csmall-def*
dest!: *JVMNaiveCollectionSemantics.csmall-nstep-SucD*
simp del: *JVMNaiveCollectionSemantics.csmall-nstep-Rec*)
with *jvm-smart-to-naive-cbig*[**where** *P=jvm-make-test-prog P t* **and** σ=*start-state* $(t\#P)$ **and** $\sigma'=\sigma'$]
jvm-smart-collect-start-make-test-prog assms **show** *?thesis* **by** *auto*
qed

12.6.5 Safety of the smart algorithm

Having proved containment in both directions, we get naive = smart:

lemma *jvm-naive-eq-smart-collection*:
assumes *naive*: $(\sigma', cset_n) \in$ *jvm-naive-out P t* **and** *smart*: $(\sigma', cset_s) \in$ *jvm-smart-out P t*
and $P \in$ *jvm-progs* **and** $t \in$ *jvm-tests*
shows $cset_n = cset_s$
using *jvm-naive-to-smart-collection*[*OF assms*] *jvm-smart-to-naive-collection*[*OF assms*] **by** *simp*

Thus, since the RTS algorithm based on *ncollect* is existence safe, the algorithm based on *scollect* is as well.

theorem *jvm-smart-existence-safe*:
assumes *P*: $P \in$ *jvm-progs* **and** P': $P' \in$ *jvm-progs* **and** *t*: $t \in$ *jvm-tests*
and *out*: $o1 \in$ *jvm-smart-out P t* **and** *dss*: *jvm-deselect P o1* P'
shows $\exists\, o2 \in$ *jvm-smart-out* P' *t*. $o1 = o2$
proof –
obtain σ' $cset_s$ **where** *o1*[*simp*]: $o1 = (\sigma', cset_s)$ **by**(*cases o1*)
with *jvm-naive-iff-smart out* **obtain** $cset_n$ **where** *n*: $(\sigma', cset_n) \in$ *jvm-naive-out P t* **by** *blast*

from *jvm-naive-eq-smart-collection*[*OF n - P t*] *out* **have** *eq*: $cset_n = cset_s$ **by** *simp*
with *jvm-naive-existence-safe*[*OF P P′ t n*] *dss* **have** *n′*: $(\sigma', cset_n) \in$ *jvm-naive-out P′ t* **by** *simp*
with *jvm-naive-iff-smart* **obtain** $cset_s{}'$ **where** *s′*: $(\sigma', cset_s{}') \in$ *jvm-smart-out P′ t* **by** *blast*

from *jvm-naive-eq-smart-collection*[*OF n′ s′ P′ t*] *eq* **have** $cset_s = cset_s{}'$ **by** *simp*
then show *?thesis* **using** *s′* **by** *simp*
qed

...thus *JVMSmartCollection* is an instance of *CollectionBasedRTS*:

interpretation *JVMSmartCollectionRTS* :
CollectionBasedRTS (=) *jvm-deselect jvm-progs jvm-tests*
JVMendset JVMcombine JVMcollect-id JVMsmall JVMSmartCollect jvm-smart-out
jvm-make-test-prog jvm-smart-collect-start
by *unfold-locales* (*rule jvm-smart-existence-safe*, *auto simp*: *start-state-def*)

end
theory *RTS*
imports
JVM-RTS/JVMCollectionBasedRTS
begin

end

References

[1] S. Mansky and E. L. Gunter. Safety of a smart classes-used regression test selection algorithm. *Electronic Notes in Theoretical Computer Science*, 351:51–73, 2020. Proceedings of LSFA 2020, the 15th International Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2020).

[2] S. E. Mansky. *Verified collection-based regression test selection via an extended Jinja semantics*. PhD thesis, University of Illinois at Urbana-Champaign, 2020.