

Regression Test Selection over JVM

Susannah Mansky

October 13, 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 <i>small</i> to multiple steps	6
3.2	Extending <i>small</i> to a big-step semantics	7
4	Collection Semantics	8
4.1	Small-Step Collection Semantics	9
4.2	Extending <i>csmall</i> to multiple steps	9
4.3	Extending <i>csmall</i> to a big-step semantics	12
5	Collection-based RTS	14
6	Instantiating <i>Semantics</i> with Jinja JVM	15
7	<i>classes-changed</i> theory	15
8	<i>subcls</i> theory	17

9	<i>classes-above</i> theory	18
9.1	Methods	20
9.2	Fields	21
9.3	Other	24
10	Instantiating <i>CollectionSemantics</i> with Jinja JVM	24
10.1	JVM-specific <i>classes-above</i> 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 <i>exec-step</i> and <i>exec-step-input</i>	37
12	Instantiating <i>CollectionBasedRTS</i> with Jinja JVM	49
12.1	Some <i>classes-above</i> 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.

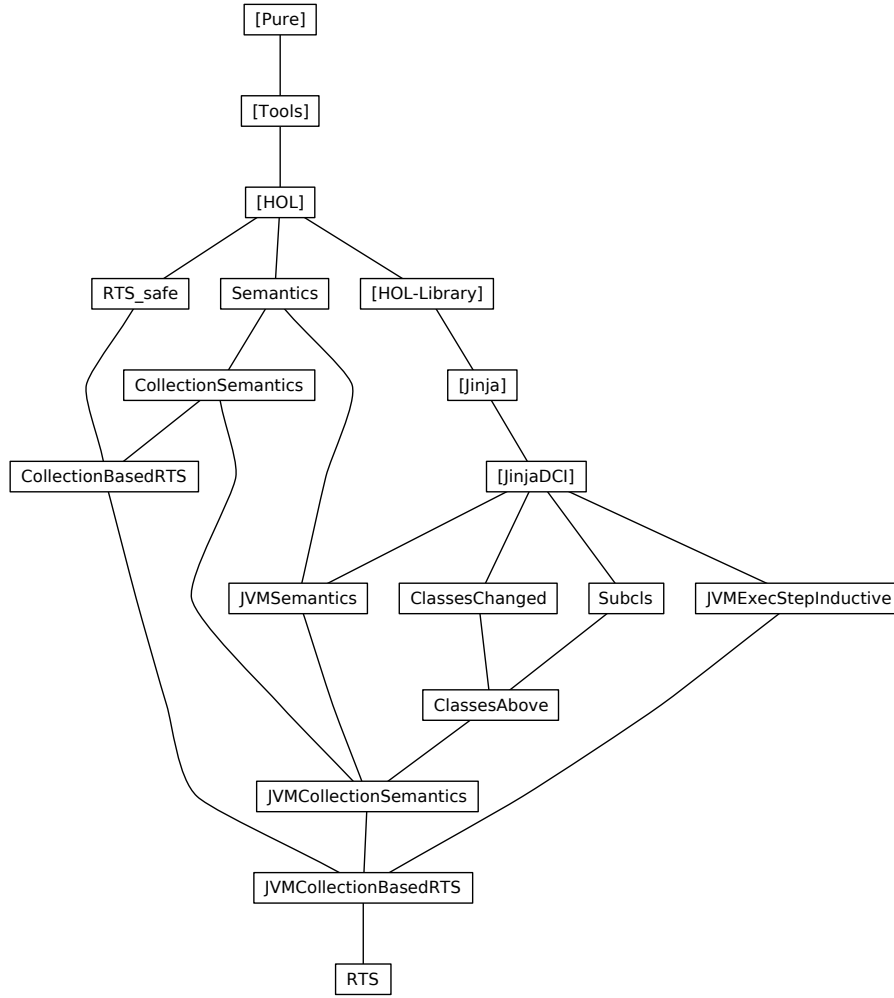


Figure 1: Theory Dependency Graph

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-out set and
    equiv-out :: 'prog-out  $\Rightarrow$  'prog-out  $\Rightarrow$  bool and
    deselect :: 'prog  $\Rightarrow$  'prog-out  $\Rightarrow$  'prog  $\Rightarrow$  bool and
    progs :: 'prog set and
    tests :: 'test set
  assumes
    existence-safe:  $\llbracket P \in \text{progs}; P' \in \text{progs}; t \in \text{tests}; o1 \in \text{out } P \ t; \text{deselect } P \ o1 \ P' \rrbracket$ 
       $\implies (\exists o2 \in \text{out } P' \ t. \text{equiv-out } o1 \ o2)$  and
    equiv-out-equiv: equiv UNIV  $\{(x,y). \text{equiv-out } x \ y\}$  and
    equiv-out-deselect:  $\llbracket \text{equiv-out } o1 \ o2; \text{deselect } P \ o1 \ P' \rrbracket \implies \text{deselect } P \ o2 \ P'$ 

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:  $\llbracket \text{equiv-out } a \ b; \text{equiv-out } b \ c \rrbracket \implies \text{equiv-out } a \ c$ 
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:  $P_s \neq []$  set  $P_s \subseteq \text{progs}$   $t \in \text{tests}$  and
  o0:  $o_0 \in \text{out } (P_s!0) \ t$  and
  des:  $\forall n < (\text{length } P_s) - 1. \text{deselect } (P_s!n) \ o_0 \ (P_s!(\text{Suc } n))$ 
shows  $\exists o_n \in \text{out } (\text{last } P_s) \ t. \text{equiv-out } o_0 \ o_n$ 
using assms proof(induct length  $P_s$  arbitrary:  $P_s$ )
  case 0 with Pst-in show ?case by simp
next
  case (Suc x) then show ?case
  proof(induct x)
    case z: 0
    from z.prem1(2,3) have  $P_s ! (\text{length } P_s - 2) = \text{last } P_s$ 
    by (simp add: last-conv-nth numeral-2-eq-2)
    with equiv-out-refl z.prem1(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  $nmt': take\ (Suc\ x')\ Ps \neq []$  using  $len'$  by auto
moreover have  $sub': set\ (take\ (Suc\ x')\ Ps) \subseteq progs$  using  $Suc'.prems(2)$ 
by (meson order-trans set-take-subset)
moreover have  $t \in tests$  using  $Pst-in(3)$  by simp
moreover have  $o_0 \in out\ (take\ (Suc\ x')\ Ps\ !\ 0)\ t$  using  $Suc'.prems(4)$  by simp
moreover have  $\forall n < length\ (take\ (Suc\ x')\ Ps) - 1.$ 
   $deselect\ (take\ (Suc\ x')\ Ps\ !\ n)\ o_0\ (take\ (Suc\ x')\ Ps\ !\ (Suc\ n))$ 
  using  $Suc'.prems(5)\ len'$  by simp
ultimately have  $\exists o' \in out\ (last\ ?Ps)\ t.$  equiv-out  $o_0\ o'$  by (rule  $Suc'.prems(1)[of\ ?Ps]$ )
  then obtain  $o'$  where  $o': o' \in out\ (last\ ?Ps)\ t$  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  $Ps-in: last\ (take\ (Suc\ x')\ Ps) \in progs\ last\ Ps \in progs$  by blast +
  have  $\exists o_n \in out\ (last\ Ps)\ t.$  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  $oN: o_n \in out\ (last\ Ps)\ t$  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  $\Rightarrow$  'state  $\Rightarrow$  'state set and
    endset :: 'state set

```

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

context *Semantics* **begin**

3.1 Extending *small* to multiple steps

primrec *small-nstep* :: '*prog* $\Rightarrow$ '*state* $\Rightarrow$ *nat* $\Rightarrow$ '*state set* **where**

small-nstep-base:

small-nstep *P* σ 0 = $\{\sigma\}$ |

small-nstep-Rec:

small-nstep *P* σ (*Suc* *n*) =
 $\{ \sigma 2. \exists \sigma 1. \sigma 1 \in \text{small-nstep } P \sigma n \wedge \sigma 2 \in \text{small } P \sigma 1 \}$

lemma *small-nstep-Rec2*:

small-nstep *P* σ (*Suc* *n*) =
 $\{ \sigma 2. \exists \sigma 1. \sigma 1 \in \text{small } P \sigma \wedge \sigma 2 \in \text{small-nstep } P \sigma 1 n \}$

proof(*induct n arbitrary: σ*)

case (*Suc* *n*)

have *right*: $\bigwedge \sigma'. \sigma' \in \text{small-nstep } P \sigma (\text{Suc}(\text{Suc } n))$

$\implies \exists \sigma 1. \sigma 1 \in \text{small } P \sigma \wedge \sigma' \in \text{small-nstep } P \sigma 1 (\text{Suc } n)$

proof –

fix σ'

assume $\sigma' \in \text{small-nstep } P \sigma (\text{Suc}(\text{Suc } n))$

then obtain $\sigma 1$ **where** *Sucnstep*: $\sigma 1 \in \text{small-nstep } P \sigma (\text{Suc } n)$ $\sigma' \in \text{small } P$
 $\sigma 1$ **by** *fastforce*

obtain $\sigma 12$ **where** *nstep*: $\sigma 12 \in \text{small } P \sigma \wedge \sigma 1 \in \text{small-nstep } P \sigma 12 n$

using *Suc Sucnstep(1)* **by** *fastforce*

then show $\exists \sigma 1. \sigma 1 \in \text{small } P \sigma \wedge \sigma' \in \text{small-nstep } P \sigma 1 (\text{Suc } n)$

using *Sucnstep* **by** *fastforce*

qed

have *left*: $\bigwedge \sigma'. \exists \sigma 1. \sigma 1 \in \text{small } P \sigma \wedge \sigma' \in \text{small-nstep } P \sigma 1 (\text{Suc } n)$

$\implies \sigma' \in \text{small-nstep } P \sigma (\text{Suc}(\text{Suc } n))$

proof –

fix σ'

assume $\exists \sigma 1. \sigma 1 \in \text{small } P \sigma \wedge \sigma' \in \text{small-nstep } P \sigma 1 (\text{Suc } n)$

then obtain $\sigma 1$ **where** *Sucnstep*: $\sigma 1 \in \text{small } P \sigma$ $\sigma' \in \text{small-nstep } P \sigma 1 (\text{Suc } n)$
 n)

by *fastforce*

obtain $\sigma 12$ **where** *nstep*: $\sigma 12 \in \text{small-nstep } P \sigma 1 n \wedge \sigma' \in \text{small } P \sigma 12$

using *Sucnstep(2)* **by** *auto*

then show $\sigma' \in \text{small-nstep } P \sigma (\text{Suc}(\text{Suc } n))$ **using** *Suc Sucnstep* **by** *fastforce*

qed

show ?*case* **using** *right left* **by** *fast*

qed(*simp*)

lemma *small-nstep-SucD*:

assumes $\sigma' \in \text{small-nstep } P \sigma (\text{Suc } n)$

shows $\exists \sigma 1. \sigma 1 \in \text{small } P \sigma \wedge \sigma' \in \text{small-nstep } P \sigma 1 n$

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

lemma *small-nstep-Suc-nend*: $\sigma' \in \text{small-nstep } P \sigma (\text{Suc } n1) \implies \sigma \notin \text{endset}$
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* $\sigma \equiv \{ \sigma'. \exists n. \sigma' \in \text{small-nstep } P \sigma n \wedge \sigma' \in \text{endset} \}$

lemma *bigI*:
 $\llbracket \sigma' \in \text{small-nstep } P \sigma n; \sigma' \in \text{endset} \rrbracket \implies \sigma' \in \text{big } P \sigma$
by (*fastforce simp add: big-def*)

lemma *bigD*:
 $\llbracket \sigma' \in \text{big } P \sigma \rrbracket \implies \exists n. \sigma' \in \text{small-nstep } P \sigma n \wedge \sigma' \in \text{endset}$
by (*simp add: big-def*)

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

lemma *big-stepD*:
assumes *big*: $\sigma' \in \text{big } P \sigma$ **and** *nend*: $\sigma \notin \text{endset}$
shows $\exists \sigma 1. \sigma 1 \in \text{small } P \sigma \wedge \sigma' \in \text{big } P \sigma 1$
proof –
obtain *n* **where** $n: \sigma' \in \text{small-nstep } P \sigma n \wedge \sigma' \in \text{endset}$
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*: $\forall \sigma. \text{small } P \sigma = \{\} \vee (\exists \sigma'. \text{small } P \sigma = \{\sigma'\})$
shows $\llbracket \sigma' \in \text{big } P \sigma; \sigma' \in \text{small-nstep } P \sigma n; \sigma' \in \text{small-nstep } P \sigma n' \rrbracket \implies n = n'$
proof(*induct n arbitrary: n' σ σ'*)
case 0
have $\sigma' = \sigma$ **using** 0.premis(2) *small-nstep-base* **by** *blast*
then have *endset*: $\sigma \in \text{endset}$ **using** 0.premis(1) *bigD* **by** *blast*
show *?case*

```

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

```

```

fixes

```

```

  collect :: 'prog  $\Rightarrow$  'state  $\Rightarrow$  'state  $\Rightarrow$  'coll and
  combine :: 'coll  $\Rightarrow$  'coll  $\Rightarrow$  '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* $\Rightarrow$ '*state* $\Rightarrow$ ('*state* $\times$ '*coll*) set **where**
csmall *P* $\sigma \equiv \{ (\sigma', coll). \sigma' \in small\ P\ \sigma \wedge collect\ P\ \sigma\ \sigma' = coll \}$

lemma *small-det-csmall-det*:

assumes $\forall \sigma. small\ P\ \sigma = \{\}$ $\vee (\exists \sigma'. small\ P\ \sigma = \{\sigma'\})$

shows $\forall \sigma. csmall\ P\ \sigma = \{\}$ $\vee (\exists o'. csmall\ P\ \sigma = \{o'\})$

using *assms* **by**(*fastforce simp: csmall-def*)

4.2 Extending *csmall* to multiple steps

primrec *csmall-nstep* :: '*prog* $\Rightarrow$ '*state* $\Rightarrow$ nat $\Rightarrow$ ('*state* $\times$ '*coll*) set **where**

csmall-nstep-base:

csmall-nstep *P* $\sigma\ 0 = \{(\sigma, collect-id)\}$ |

csmall-nstep-Rec:

csmall-nstep *P* $\sigma\ (Suc\ n) =$

$\{ (\sigma 2, coll). \exists \sigma 1\ coll1\ coll2. (\sigma 1, coll1) \in csmall-nstep\ P\ \sigma\ n$
 $\wedge (\sigma 2, coll2) \in csmall\ P\ \sigma 1 \wedge combine\ coll1\ coll2 = coll \}$

lemma *small-nstep-csmall-nstep-equiv*:

small-nstep *P* $\sigma\ n$

$= \{ \sigma'. \exists coll. (\sigma', coll) \in csmall-nstep\ P\ \sigma\ n \}$

proof (*induct n*) **qed**(*simp-all add: csmall-def*)

lemma *csmall-nstep-exists*:

$\sigma' \in big\ P\ \sigma \implies \exists n\ coll. (\sigma', coll) \in csmall-nstep\ P\ \sigma\ n \wedge \sigma' \in endset$

proof(*drule bigD*) **qed**(*clarsimp simp: small-nstep-csmall-nstep-equiv*)

lemma *csmall-det-csmall-nstep-det*:

assumes $\forall \sigma. csmall\ P\ \sigma = \{\}$ $\vee (\exists o'. csmall\ P\ \sigma = \{o'\})$

shows $\forall \sigma. csmall-nstep\ P\ \sigma\ n = \{\}$ $\vee (\exists o'. csmall-nstep\ P\ \sigma\ n = \{o'\})$

using *assms*

proof(*induct n*)

case (*Suc n*) **then show** ?*case* **by** *fastforce*

qed(*simp*)

lemma *csmall-nstep-Rec2*:

csmall-nstep *P* $\sigma\ (Suc\ n) =$

$\{ (\sigma 2, coll). \exists \sigma 1\ coll1\ coll2. (\sigma 1, coll1) \in csmall\ P\ \sigma$
 $\wedge (\sigma 2, coll2) \in csmall-nstep\ P\ \sigma 1\ n \wedge combine\ coll1\ coll2 = coll$

$\}$

proof(*induct n arbitrary: σ*)

case (*Suc n*)

have *right*: $\bigwedge \sigma' coll'. (\sigma', coll') \in csmall-nstep\ P\ \sigma\ (Suc(Suc\ n))$

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

lemma csmall-nstep-SucD :

assumes $(\sigma', \text{coll}') \in \text{csmall-nstep } P \sigma (\text{Suc } n)$

shows $\exists \sigma 1 \text{ coll1. } (\sigma 1, \text{coll1}) \in \text{csmall } P \sigma$

$\quad \wedge (\exists \text{coll. coll}' = \text{combine coll1 coll} \wedge (\sigma', \text{coll}') \in \text{csmall-nstep } P \sigma 1 n)$

using $\text{csmall-nstep-Rec2 CollectionSemantics-axioms case-prodD assms}$ **by fastforce**

lemma $\text{csmall-nstep-Suc-nend}$: $\sigma' \in \text{csmall-nstep } P \sigma (\text{Suc } n1) \implies \sigma \notin \text{endset}$

using $\text{endset-final csmall-nstep-SucD CollectionSemantics.csmall-def CollectionSemantics-axioms}$

by *fastforce*

lemma *small-to-csmall-nstep-pres*:

assumes *Qpres*: $\bigwedge P \sigma \sigma'. Q P \sigma \implies \sigma' \in \text{small } P \sigma \implies Q P \sigma'$

shows $Q P \sigma \implies (\sigma', \text{coll}) \in \text{csmall-nstep } P \sigma n \implies Q P \sigma'$

proof(*induct n arbitrary: $\sigma \sigma' \text{coll}$*)

case (*Suc n*)

then obtain $\sigma 1 \text{ coll1 coll2}$ where *nstep*: $(\sigma 1, \text{coll1}) \in \text{csmall-nstep } P \sigma n$

$\wedge (\sigma', \text{coll2}) \in \text{csmall } P \sigma 1 \wedge \text{combine coll1 coll2} = \text{coll}$ 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*: $\bigwedge P \sigma \sigma' \text{coll}. (\sigma', \text{coll}) \in \text{csmall } P \sigma \implies R P \text{coll} \implies Q P \sigma \implies R' P \sigma \sigma' \text{coll}$

and *Rcomb*: $\bigwedge P \text{coll1 coll2}. R P (\text{combine coll1 coll2}) = (R P \text{coll1} \wedge R P \text{coll2})$

and *Qpres*: $\bigwedge P \sigma \sigma'. Q P \sigma \implies \sigma' \in \text{small } P \sigma \implies Q P \sigma'$

and *Rtrans'*: $\bigwedge P \sigma \sigma 1 \sigma' \text{coll1 coll2}.$

$R' P \sigma \sigma 1 \text{coll1} \wedge R' P \sigma 1 \sigma' \text{coll2} \implies R' P \sigma \sigma' (\text{combine}$

$\text{coll1 coll2})$

and *base*: $\bigwedge \sigma. R' P \sigma \sigma \text{collect-id}$

shows $(\sigma', \text{coll}) \in \text{csmall-nstep } P \sigma n \implies R P \text{coll} \implies Q P \sigma \implies R' P \sigma \sigma' \text{coll}$

proof(*induct n arbitrary: $\sigma \sigma' \text{coll}$*)

case (*Suc n*)

then obtain $\sigma 1 \text{ coll1 coll2}$ where *nstep*: $(\sigma 1, \text{coll1}) \in \text{csmall-nstep } P \sigma n$

$\wedge (\sigma', \text{coll2}) \in \text{csmall } P \sigma 1 \wedge \text{combine coll1 coll2} = \text{coll}$ 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*: $\bigwedge P P' \sigma \sigma' \text{coll}. (\sigma', \text{coll}) \in \text{csmall } P \sigma$

$\implies R P P' \text{coll} \implies Q \sigma \implies (\sigma', \text{coll}) \in \text{csmall } P' \sigma$

and *Rcomb*: $\bigwedge P P' \text{coll1 coll2}. R P P' (\text{combine coll1 coll2}) = (R P P' \text{coll1} \wedge R P P' \text{coll2})$

and *Qpres*: $\bigwedge P \sigma \sigma'. Q \sigma \implies \sigma' \in \text{small } P \sigma \implies Q \sigma'$

shows $(\sigma', \text{coll}) \in \text{csmall-nstep } P \sigma n \implies R P P' \text{coll} \implies Q \sigma \implies (\sigma', \text{coll}) \in \text{csmall-nstep } P' \sigma n$

proof(*induct n arbitrary: $\sigma \sigma' \text{coll}$*)

case (*Suc n*)

then obtain $\sigma 1 \text{ coll1 coll2}$ where *nstep*: $(\sigma 1, \text{coll1}) \in \text{csmall-nstep } P \sigma n$

$\wedge (\sigma', \text{coll2}) \in \text{csmall } P \sigma 1 \wedge \text{combine coll1 coll2} = \text{coll}$ 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*:

$\llbracket (\sigma', coll') \in cbig\ P\ \sigma \rrbracket \implies \exists n. (\sigma', coll') \in csmall\text{-}nstep\ P\ \sigma\ n \wedge \sigma' \in endset$
by(*simp add: cbig-def*)

lemma *cbigD'*:

$\llbracket o' \in cbig\ P\ \sigma \rrbracket \implies \exists n. o' \in csmall\text{-}nstep\ P\ \sigma\ n \wedge fst\ o' \in endset$
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*:

$(\sigma', coll) \in cbig\ P\ \sigma \implies \sigma' \in big\ P\ \sigma$

using *cbig-big-equiv* **by** *blast*

lemma *csmall-to-cbig-prop*:

assumes $\bigwedge P\ \sigma\ \sigma'\ coll. (\sigma', coll) \in csmall\ P\ \sigma \implies R\ P\ coll \implies Q\ P\ \sigma \implies R'$
 $P\ \sigma\ \sigma'\ coll$

and $\bigwedge P\ coll1\ coll2. R\ P\ (combine\ coll1\ coll2) = (R\ P\ coll1 \wedge R\ P\ coll2)$

and $\bigwedge P\ \sigma\ \sigma'. Q\ P\ \sigma \implies \sigma' \in small\ P\ \sigma \implies Q\ P\ \sigma'$

and $\bigwedge P \sigma \sigma 1 \sigma' coll1 coll2.$
 $R' P \sigma \sigma 1 coll1 \wedge R' P \sigma 1 \sigma' coll2 \implies R' P \sigma \sigma' \text{ (combine$
 $coll1 coll2)}$
and $\bigwedge \sigma. R' P \sigma \sigma \text{ collect-id}$
shows $(\sigma', coll) \in cbig P \sigma \implies R P coll \implies Q P \sigma \implies R' P \sigma \sigma' coll$
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 $\bigwedge P P' \sigma \sigma' coll. (\sigma', coll) \in csmall P \sigma \implies R P P' coll \implies Q \sigma \implies$
 $(\sigma', coll) \in csmall P' \sigma$
and $\bigwedge P P' coll1 coll2. R P P' \text{ (combine coll1 coll2)} = (R P P' coll1 \wedge R P P'$
 $coll2)$
and $Qpres: \bigwedge P \sigma \sigma'. Q \sigma \implies \sigma' \in small P \sigma \implies Q \sigma'$
shows $(\sigma', coll) \in cbig P \sigma \implies R P P' coll \implies Q \sigma \implies (\sigma', coll) \in cbig P' \sigma$
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 $\exists \sigma 1 coll1. (\sigma 1, coll1) \in csmall P \sigma$
 $\wedge (\exists coll. coll' = \text{combine coll1 coll} \wedge (\sigma', coll') \in cbig P \sigma 1)$
proof –
obtain n **where** $n: (\sigma', coll') \in csmall-nstep P \sigma n \sigma' \in endset$
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 $\llbracket (\sigma', coll') \in cbig P \sigma; (\sigma', coll') \in csmall-nstep P \sigma n; (\sigma', coll'') \in cs-$
 $mall-nstep P \sigma n' \rrbracket$
 $\implies n = n'$
proof (*induct n arbitrary: n' $\sigma \sigma' coll' coll''$*)
case 0
have $\sigma' = \sigma$ **using** 0.prem(2) *csmall-nstep-base by blast*
then have *endset*: $\sigma \in endset$ **using** 0.prem(1) *cbigD by blast*
show *?case*
proof (*cases n'*)
case *Suc* **then show** *?thesis using 0.prem(3) csmall-nstep-Suc-nend endset*
by *blast*
qed (*simp*)
next
case (*Suc n1*)
then have *endset*: $\sigma' \in endset$ **using** *Suc.prem(1) cbigD by blast*

```

have nend:  $\sigma \notin \text{endset}$  using csmall-nstep-Suc-nend[OF Suc.premis(2)] by simp
then have neq:  $\sigma' \neq \sigma$  using endset by auto
obtain  $\sigma 1$  coll coll1 where  $\sigma 1$ :  $(\sigma 1, \text{coll} 1) \in \text{csmall } P \ \sigma$   $\text{coll}' = \text{combine coll} 1$ 
coll
   $(\sigma', \text{coll}) \in \text{csmall-nstep } P \ \sigma 1 \ n1$ 
  using csmall-nstep-SucD[OF Suc.premis(2)] by clarsimp
then have cbig:  $(\sigma', \text{coll}) \in \text{cbig } P \ \sigma 1$  using endset by (auto simp: cbig-def)
show ?case
proof(cases n')
  case 0 then show ?thesis using neq Suc.premis(3) using csmall-nstep-base by
simp
  next
    case Suc': (Suc n1')
    then obtain  $\sigma 1'$  coll2 coll1' where  $\sigma 1'$ :  $(\sigma 1', \text{coll} 1') \in \text{csmall } P \ \sigma$   $\text{coll}'' =$ 
combine coll1' coll2
       $(\sigma', \text{coll} 2) \in \text{csmall-nstep } P \ \sigma 1' \ n1'$ 
      using csmall-nstep-SucD[where  $\sigma = \sigma$  and  $\sigma' = \sigma'$  and  $\text{coll}' = \text{coll}''$  and  $n = n1'$ ]
Suc.premis(3) by blast
    then have  $\sigma 1 = \sigma 1'$  using  $\sigma 1$  det 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 — 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  $\Rightarrow$  'state  $\Rightarrow$  'state set and
  collect = collect :: 'prog  $\Rightarrow$  'state  $\Rightarrow$  'state  $\Rightarrow$  'coll and
  out = out :: 'prog  $\Rightarrow$  'test  $\Rightarrow$  ('state  $\times$  'coll) set
for small collect out

```

```

+

```

```

fixes

```

```

  make-test-prog :: 'prog  $\Rightarrow$  'test  $\Rightarrow$  'prog and
  collect-start :: 'prog  $\Rightarrow$  'coll

```

```

assumes

```

```

  out-cbig:

```

```

   $\exists i. \text{out } P \ t = \{(\sigma', \text{coll}'). \exists \text{coll}. (\sigma', \text{coll}) \in \text{cbig } (\text{make-test-prog } P \ t) \ i$ 
 $\wedge \text{coll}' = \text{combine coll } (\text{collect-start } P) \}$ 

```

context *CollectionBasedRTS* **begin**

end — *CollectionBasedRTS*

end

6 Instantiating *Semantics* with Jinja JVM

theory *JVMSemantics*

imports *../Common/Semantics JinjaDCI.JVMExec*

begin

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

lemma *JVMsmall-prealloc-pres*:

assumes *pre*: *preallocated* (*fst*(*snd* σ))

and $\sigma' \in \text{JVMsmall } P \ \sigma$

shows *preallocated* (*fst*(*snd* σ'))

using *exec-prealloc-pres*[*OF pre*] *assms* **by**(*cases* σ , *cases* σ' , *auto*)

lemma *JVMsmall-det*: *JVMsmall* *P* σ = { } $\vee$ ($\exists \sigma'. \text{JVMsmall } P \ \sigma = \{\sigma'\}$)

by *auto*

definition *JVMendset* :: *jvm-state set* **where**

JVMendset $\equiv$ { (*xp*, *h*, *frs*, *sh*). *frs* = [] $\vee$ ($\exists x. \text{xp} = \text{Some } x$) }

lemma *JVMendset-final*: $\sigma \in \text{JVMendset} \implies \forall P. \text{JVMsmall } P \ \sigma = \{ \}$

by(*auto simp: JVMendset-def*)

lemma *start-state-nend*:

start-state *P* $\notin \text{JVMendset}$

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 $\Rightarrow$ 'm prog $\Rightarrow$ cname set **where**
classes-changed P1 P2 = {cn. class P1 cn $\neq$ class P2 cn}

definition *class-changed* :: 'm prog $\Rightarrow$ 'm prog $\Rightarrow$ cname $\Rightarrow$ bool **where**
class-changed P1 P2 cn = (class P1 cn $\neq$ class P2 cn)

lemma *classes-changed-class-changed[simp]*: cn $\in$ *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*: $\llbracket \text{cn} \notin \text{classes-changed } P \ P' \rrbracket \implies \text{class } P \ \text{cn} = \text{class } P' \ \text{cn}$
by (clarsimp simp: *class-changed-def*)

lemma *classes-changed-class-set*: $\llbracket S \cap \text{classes-changed } P \ P' = \{\} \rrbracket$
 $\implies \forall C \in S. \text{class } P \ C = \text{class } P' \ C$
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})
 $\cup$ (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 $\notin$ *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 $\in$ *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 $\cap$ *classes-changed* P P' = {}

```

shows  $coll \cap \text{classes-changed } (t \# P) (t \# P') = \{\}$ 
proof(cases fst t ∈ classes-changed P P')
  case True
    then have  $\text{classes-changed } P P' = \text{classes-changed } (t \# P) (t \# P') \cup \{\text{fst } t\}$ 
    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  $\text{classes-changed } P P' = \text{classes-changed } (t \# P) (t \# P')$ 
    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:  $\llbracket P \vdash C \preceq^* C'; C \neq C' \rrbracket$ 
 $\implies \exists D' fs ms. \text{class } P C = \lfloor (D', fs, ms) \rfloor$ 
proof(induct rule: converse-rtrancl-induct)
  case (step y z) then show ?case by(auto dest: subcls1D)
qed(simp)

```

```

lemma class-subcls1:
 $\llbracket \text{class } P y = \text{class } P' y; P \vdash y \prec^1 z \rrbracket \implies P' \vdash y \prec^1 z$ 
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:
 $\llbracket P \vdash C \preceq^* C'; P \vdash C \preceq^* C'' \rrbracket \implies P \vdash C' \preceq^* C'' \vee P \vdash C'' \preceq^* C'$ 
by (simp add: single-valued-confluent subcls1-single-valued)

```

```

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

```

```

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

```

```

lemma subcls-of-Obj-acyclic:

```

```

[[ P ⊢ C ≼* Object; C ≠ D ]] ⇒ ¬(P ⊢ C ≼* D ∧ P ⊢ D ≼* 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 neg: False
    with step.hyps(3) have ¬(P ⊢ z ≼* D ∧ P ⊢ D ≼* z) by simp
    moreover from step.hyps(1)
    have P ⊢ D ≼* y ⇒ P ⊢ D ≼* z by(simp add: rtrancl-into-rtrancl)
    moreover from step.hyps(1) step.prem(1)
    have P ⊢ y ≼* D ⇒ P ⊢ z ≼* D by(simp add: subcls1-confluent)
    ultimately show ?thesis by clarsimp
  qed
qed
qed

lemma subcls-of-Obj: [[ P ⊢ C ≼* Object; P ⊢ C ≼* D ]] ⇒ P ⊢ D ≼* 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 ⊢ c ≼* cn }

```

```

abbreviation classes-between :: 'm prog ⇒ cname ⇒ cname ⇒ cname set where
classes-between P c d ≡ { cn. (P ⊢ c ≼* cn ∧ P ⊢ cn ≼* d) }

```

```

abbreviation classes-above-xcpts :: 'm prog ⇒ cname set where
classes-above-xcpts P ≡ ⋃x∈sys-xcpts. classes-above P x

```

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

lemma *classes-above-class*:
 $\llbracket \text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\}; P \vdash C \preceq^* C' \rrbracket$
 $\implies \text{class } P \ C' = \text{class } P' \ C'$
by (*drule classes-changed-class-set, simp*)

lemma *classes-above-subset*:
assumes $\text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\}$
shows $\text{classes-above } P \ C \subseteq \text{classes-above } P' \ C$
proof –
have *ind*: $\bigwedge x. P \vdash C \preceq^* x \implies P' \vdash C \preceq^* x$
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*:
 $\llbracket \text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\}; P \vdash C \preceq^* C' \rrbracket$
 $\implies P' \vdash C \preceq^* C'$
by (*fastforce dest: classes-above-subset*)

lemma *classes-above-subset2*:
assumes $\text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\}$
shows $\text{classes-above } P' \ C \subseteq \text{classes-above } P \ C$
proof –
have *ind*: $\bigwedge x. P' \vdash C \preceq^* x \implies P \vdash C \preceq^* x$
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*:
 $\llbracket \text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\}; P' \vdash C \preceq^* C' \rrbracket$
 $\implies P \vdash C \preceq^* C'$
 by (*fastforce dest: classes-above-subset2*)

lemma *classes-above-set*:
 $\llbracket \text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\} \rrbracket$
 $\implies \text{classes-above } P \ C = \text{classes-above } P' \ C$
 by(*fastforce dest: classes-above-subset classes-above-subset2*)

lemma *classes-above-classes-changed-sym*:
assumes *classes-above* $P \ C \cap \text{classes-changed } P \ P' = \{\}$
shows *classes-above* $P' \ C \cap \text{classes-changed } P' \ P = \{\}$
proof –
 have *classes-above* $P \ C = \text{classes-above } P' \ C$ 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*:
 $P \vdash C \preceq^* D \implies \text{classes-above } P \ C = (\text{classes-between } P \ C \ D - \{D\}) \cup \text{classes-above } P \ D$
 using *subcls-confluent* by *auto*

lemma *classes-above-subcls-subset*:
 $\llbracket P \vdash C \preceq^* C' \rrbracket \implies \text{classes-above } P \ C' \subseteq \text{classes-above } P \ C$
 by *auto*

9.1 Methods

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

 have $\bigwedge C \ Mm. P \vdash C$ *sees-methods* *Mm* $\implies$
 $\forall C' \in \text{classes-above } P \ C. \text{class } P \ C' = \text{class } P' \ C' \implies P' \vdash C$
sees-methods *Mm*
proof –
 fix *C* *Mm* **assume** $P \vdash C$ *sees-methods* *Mm* **and** $\forall C' \in \text{classes-above } P \ C. \text{class } P \ C' = \text{class } P' \ C'$
 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 =  $\lfloor (D, fs, ms) \rfloor$  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-rtranc1[OF subcls1I])
  with converse-rtranc1-into-rtranc1 have  $\bigwedge x. P \vdash D \preceq^* x \implies P \vdash C \preceq^* x$ 
by simp
  with rec.prem1 have  $\forall C' \in \text{classes-above } P \ D. \text{class } P \ C' = \text{class } P' \ C'$  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*:

```

 $\llbracket \text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\};$ 
 $P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } C' \rrbracket$ 
 $\implies P' \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } C'$ 
by (auto dest: classes-above-sees-methods simp: Method-def)

```

lemma *classes-above-sees-method2*:

```

 $\llbracket \text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\};$ 
 $P' \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } C' \rrbracket$ 
 $\implies P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } C'$ 
by (auto dest: classes-above-classes-changed-sym intro: classes-above-sees-method)

```

lemma *classes-above-method*:

assumes *classes-above* $P \ C \cap \text{classes-changed } P \ P' = \{\}$

shows *method* $P \ C \ M = \text{method } P' \ C \ M$

proof(*cases* $\exists Ts \ T \ m \ D \ b. P \vdash C \text{ sees } M, b : Ts \rightarrow T = m \text{ in } D$)

```

  case True
  with assms show ?thesis by (auto dest: classes-above-sees-method)
next
  case False
  with assms have  $\neg(\exists Ts \ T \ m \ D \ b. P' \vdash C \text{ sees } M, b : Ts \rightarrow T = m \text{ in } D)$ 
  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* $P \ C \cap \text{classes-changed } P \ P' = \{\}$ **and** *fs*: $P \vdash C$

has-fields *FDTs*

shows $P' \vdash C$ *has-fields* *FDTs*

proof –

```

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

have  $\bigwedge C \text{ Mm}. P \vdash C \text{ has-fields FDTs} \implies$ 
 $\forall C' \in \text{classes-above } P \ C. \text{ class } P \ C' = \text{ class } P' \ C' \implies P' \vdash C \text{ has-fields}$ 
FDTs
proof –
  fix C Mm assume  $P \vdash C \text{ has-fields FDTs}$  and  $\forall C' \in \text{classes-above } P \ C. \text{ class }$ 
P C' = class P' C'
  then show  $P' \vdash C \text{ has-fields FDTs}$ 
  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-rtranc1[OF subcls1I])
    with converse-rtranc1-into-rtranc1 have  $\bigwedge x. P \vdash D \preceq^* x \implies P \vdash C \preceq^* x$ 
by simp
    with rec.prem1(1) have  $\forall x. P \vdash D \preceq^* x \longrightarrow \text{class } P \ x = \text{class } P' \ x$  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  $(\forall \text{FDTs}. \neg P \vdash C \text{ has-fields FDTs}) = (\forall \text{FDTs}. \neg P' \vdash C \text{ has-fields FDTs})$ 
proof(rule iffI)
  assume asm:  $\forall \text{FDTs}. \neg P \vdash C \text{ has-fields FDTs}$ 
  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  $\forall \text{FDTs}. \neg P' \vdash C \text{ has-fields}$ 
FDTs by auto
next
  assume  $\forall \text{FDTs}. \neg P' \vdash C \text{ has-fields FDTs}$ 
  with assms show  $\forall \text{FDTs}. \neg P \vdash C \text{ has-fields FDTs}$  by(auto dest: classes-above-has-fields)
qed

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

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

```

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

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

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

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

lemma *classes-above-fields*:
assumes *classes-above* $P \ C \cap \text{classes-changed } P \ P' = \{\}$
shows *fields* $P \ C = \text{fields } P' \ C$
proof(*cases* $\exists \text{FDTs}. P \vdash C \text{ has-fields } \text{FDTs}$)
 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*:
 $\llbracket \text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\} \rrbracket$
 $\Rightarrow$
 $\text{ifields } P \ C = \text{ifields } P' \ C$
by (*simp add: ifields-def classes-above-fields*)

lemma *classes-above-blank*:
 $\llbracket \text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\} \rrbracket$
 $\Rightarrow$

blank $P \ C = \text{blank } P' \ C$
by (*simp add: blank-def classes-above-ifields*)

lemma *classes-above-isfields*:
 $\llbracket \text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\} \rrbracket$
 $\implies$
isfields $P \ C = \text{isfields } P' \ C$
by (*simp add: isfields-def classes-above-fields*)

lemma *classes-above-sblank*:
 $\llbracket \text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\} \rrbracket$
 $\implies$
sblank $P \ C = \text{sblank } P' \ C$
by (*simp add: sblank-def classes-above-isfields*)

9.3 Other

lemma *classes-above-start-heap*:
assumes *classes-above-xcpts* $P \cap \text{classes-changed } P \ P' = \{\}$
shows *start-heap* $P = \text{start-heap } P'$
proof –
from *assms* **have** $\forall C \in \text{sys-xcpts}. \text{blank } P \ C = \text{blank } P' \ C$ **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* $:: \text{cname set} \Rightarrow \text{cname set} \Rightarrow \text{cname set}$ **where**
JVMcombine $C \ C' \equiv C \cup C'$

abbreviation *JVMcollect-id* $:: \text{cname set}$ **where**
JVMcollect-id $\equiv \{\}$

10.1 JVM-specific *classes-above* theory

fun *classes-above-frames* $:: 'm \text{ prog} \Rightarrow \text{frame list} \Rightarrow \text{cname set}$ **where**
classes-above-frames $P \ ((\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics}) \# \text{frs}) = \text{classes-above } P \ C \cup \text{classes-above-frames } P \ \text{frs} \mid$
classes-above-frames $P \ [] = \{\}$

lemma *classes-above-start-state*:

assumes *above-xcpts*: *classes-above-xcpts* $P \cap \text{classes-changed } P P' = \{\}$
shows *start-state* $P = \text{start-state } P'$
using *assms* *classes-above-start-heap* **by**(*simp add: start-state-def*)

lemma *classes-above-matches-ex-entry*:
classes-above $P C \cap \text{classes-changed } P P' = \{\}$
 $\implies \text{matches-ex-entry } P C pc xcp = \text{matches-ex-entry } P' C pc xcp$
using *classes-above-subcls* *classes-above-subcls2*
by(*auto simp: matches-ex-entry-def*)

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

lemma *classes-above-find-handler*:
assumes *classes-above* $P (\text{cname-of } h a) \cap \text{classes-changed } P P' = \{\}$
shows *classes-above-frames* $P frs \cap \text{classes-changed } P P' = \{\}$
 $\implies \text{find-handler } P a h frs sh = \text{find-handler } P' a h frs sh$
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**
 $\text{intC: } \text{classes-above } P C \cap \text{classes-changed } P P' = \{\}$
and *int*: *classes-above-frames* $P frs' \cap \text{classes-changed } P P' = \{\}$ **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 $P a h frs sh = (\text{xp}', h', frs', sh')$
 $\implies \text{classes-above-frames } P frs' \subseteq \text{classes-above-frames } P frs$
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 $P a h frs sh = (\text{xp}', h', frs', sh')$
 $\implies h = h' \wedge sh = sh' \wedge \text{classes-above-frames } P frs' \subseteq \text{classes-above-frames } P frs$
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]  $\Rightarrow$  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]  $\Rightarrow$  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'  $\cup$  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  $\Rightarrow$  jvm-state  $\Rightarrow$  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
      $\cup$  classes-above P C  $\cup$  classes-above-frames P frs  $\cup$  classes-above-xcpts P
    )
  | JVMexec-ncollect P - = {}

```

```

fun JVMNaiveCollect :: jvm-prog  $\Rightarrow$  jvm-state  $\Rightarrow$  jvm-state  $\Rightarrow$  cname set where
  JVMNaiveCollect P  $\sigma$   $\sigma'$  = JVMexec-ncollect P  $\sigma$ 

```

interpretation *JVMNaiveCollectionSemantics*:

CollectionSemantics *JVMsmall* *JVMendset* *JVMNaiveCollect* *JVMcombine* *JVM-*

collect-id
by *unfold-locales auto*

10.3 Smarter RTS algorithm

fun *JVMinstr-scollect* ::
 $[jvm\text{-}prog, instr] \Rightarrow cname\ set$ **where**
JVMinstr-scollect *P* (*Getstatic* *C F D*)
= (if $\neg(\exists t. P \vdash C \text{ has } F, Static:t \text{ in } D)$ then *classes-above* *P C*
else *classes-between* *P C D* - {*D*}) |
JVMinstr-scollect *P* (*Putstatic* *C F D*)
= (if $\neg(\exists t. P \vdash C \text{ has } F, Static:t \text{ in } D)$ then *classes-above* *P C*
else *classes-between* *P C D* - {*D*}) |
JVMinstr-scollect *P* (*Invokestatic* *C M n*)
= (if $\neg(\exists Ts\ T\ m\ D. P \vdash C \text{ sees } M, Static:Ts \rightarrow T = m \text{ in } D)$ 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* ::
 $[jvm\text{-}prog, instr, init\text{-}call\text{-}status] \Rightarrow cname\ set$ **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* $\Rightarrow$ *jvm-state* $\Rightarrow$ *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* $\Rightarrow$ *jvm-state* $\Rightarrow$ *jvm-state* $\Rightarrow$ *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*:
 $(\sigma', cset) \in JVMNaiveCollectionSemantics.csmall\ P\ \sigma$
 $\Rightarrow JVMexec\text{-}ncollect\ P\ \sigma = cset \wedge \sigma' \in JVMsmall\ P\ \sigma$
by(*simp add: JVMNaiveCollectionSemantics.csmall-def*)

lemma *JVMsmart-csmallD*:

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

lemma *jvm-naive-to-smart-csmall-nstep-last-eq*:

$\llbracket (\sigma', cset_n) \in \text{JVMNaiveCollectionSemantics.cbig } P \sigma;$
 $(\sigma', cset_n) \in \text{JVMNaiveCollectionSemantics.csmall-nstep } P \sigma n;$
 $(\sigma', cset_s) \in \text{JVMSmartCollectionSemantics.csmall-nstep } P \sigma n' \rrbracket$
 $\implies n = n'$
proof(induct n arbitrary: $n' \sigma \sigma' cset_n cset_s$)
case 0
have $\sigma' = \sigma$ **using** 0.prem(2) JVMNaiveCollectionSemantics.csmall-nstep-base
by blast
then have $\text{endset}: \sigma \in \text{JVMendset}$ **using** 0.prem(1) JVMNaiveCollectionSemantics.cbigD **by** blast
show ?case
proof(cases n')
case Suc **then show** ?thesis **using** 0.prem(3) JVMSmartCollectionSemantics.csmall-nstep-Suc-nend
 endset **by** blast
qed(simp)
next
case (Suc $n1$)
then have $\text{endset}: \sigma' \in \text{JVMendset}$ **using** Suc.prem(1) JVMNaiveCollectionSemantics.cbigD **by** blast
have $\text{nend}: \sigma \notin \text{JVMendset}$
using JVMNaiveCollectionSemantics.csmall-nstep-Suc-nend[OF Suc.prem(2)]
by simp
then have $\text{neg}: \sigma' \neq \sigma$ **using** endset **by** auto
obtain $\sigma1$ $cset$ $cset1$ **where** $\sigma1: (\sigma1, cset1) \in \text{JVMNaiveCollectionSemantics.csmall } P \sigma$
 $cset_n = cset1 \cup cset$ $(\sigma', cset) \in \text{JVMNaiveCollectionSemantics.csmall-nstep } P \sigma1 n1$
using JVMNaiveCollectionSemantics.csmall-nstep-SucD[OF Suc.prem(2)] **by** clarsimp
then have $\text{cbig}: (\sigma', cset) \in \text{JVMNaiveCollectionSemantics.cbig } P \sigma1$
using endset **by**(auto simp: JVMNaiveCollectionSemantics.cbig-def)
show ?case
proof(cases n')
case 0 **then show** ?thesis
using neg Suc.prem(3) JVMSmartCollectionSemantics.csmall-nstep-base **by** blast
next
case Suc': (Suc $n1'$)
then obtain $\sigma1'$ $cset2$ $cset1'$ **where** $\sigma1': (\sigma1', cset1') \in \text{JVMSmartCollectionSemantics.csmall } P \sigma$
 $cset_s = cset1' \cup cset2$ $(\sigma', cset2) \in \text{JVMSmartCollectionSemantics.csmall-nstep } P \sigma1' n1'$

```

    using JVMSmartCollectionSemantics.csmall-nstep-SucD[where  $\sigma=\sigma$  and
 $\sigma'=\sigma'$  and  $coll'=cset_s$ 
    and  $n=n1'$ ] Suc.premis( $\mathcal{J}$ ) by blast
  then have  $\sigma1=\sigma1'$  using  $\sigma1$  JVMNaiveCollectionSemantics.csmall-def
    JVMSmartCollectionSemantics.csmall-def by auto
  then show ?thesis using Suc.hyps(1)[OF cbig  $\sigma1(\mathcal{J})$ ]  $\sigma1'(\mathcal{J})$  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]  $\Rightarrow$ 

```

```

bool

```

```

where

```

```

  exec-step-ind-Load:

```

```

exec-step-ind (StepI (Load n)) P h stk loc C0 M0 pc ics frs sh
  (None, h, ((loc ! n) # stk, loc, C0, M0, Suc pc, ics)#frs, sh)

```

```

| exec-step-ind-Store:

```

```

exec-step-ind (StepI (Store n)) P h stk loc C0 M0 pc ics frs sh
  (None, h, (tl stk, loc[n:=hd stk], C0, M0, Suc pc, ics)#frs, sh)

```

```

| exec-step-ind-Push:

```

```

exec-step-ind (StepI (Push v)) P h stk loc C0 M0 pc ics frs sh
  (None, h, (v # stk, loc, C0, M0, Suc pc, ics)#frs, sh)

```

```

| exec-step-ind-NewOOM-Called:

```

```

new-Addr h = None
 $\Rightarrow$  exec-step-ind (StepI (New C)) P h stk loc C0 M0 pc (Called Cs) frs sh
  ([addr-of-sys-xcpt OutOfMemory], h, (stk, loc, C0, M0, pc, No-ics)#frs, sh)

```

```

| exec-step-ind-NewOOM-Done:

```

```

[[ sh C = Some(obj, Done); new-Addr h = None;  $\forall$  Cs. ics  $\neq$  Called Cs ]]
 $\Rightarrow$  exec-step-ind (StepI (New C)) P h stk loc C0 M0 pc ics frs sh
  ([addr-of-sys-xcpt OutOfMemory], h, (stk, loc, C0, M0, pc, ics)#frs, sh)

```

```

| exec-step-ind-New-Called:

```

$new_Addr\ h = Some\ a$
 $\implies exec_step_ind\ (StepI\ (New\ C))\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ (Called\ Cs)\ frs\ sh$
 $(None, h(a \rightarrow blank\ P\ C), (Addr\ a \# stk, loc, C_0, M_0, Suc\ pc, No_ics) \# frs, sh)$

$| exec_step_ind_New_Done:$
 $\llbracket sh\ C = Some(obj, Done); new_Addr\ h = Some\ a; \forall Cs. ics \neq Called\ Cs \rrbracket$
 $\implies exec_step_ind\ (StepI\ (New\ C))\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
 $(None, h(a \rightarrow blank\ P\ C), (Addr\ a \# stk, loc, C_0, M_0, Suc\ pc, ics) \# frs, sh)$

$| exec_step_ind_New_Init:$
 $\llbracket \forall obj. sh\ C \neq Some(obj, Done); \forall Cs. ics \neq Called\ Cs \rrbracket$
 $\implies exec_step_ind\ (StepI\ (New\ C))\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
 $(None, h, (stk, loc, C_0, M_0, pc, Calling\ C\ []) \# frs, sh)$

$| exec_step_ind_Getfield_Null:$
 $hd\ stk = Null$
 $\implies exec_step_ind\ (StepI\ (Getfield\ F\ C))\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
 $(\lfloor addr_of_sys_xcpt\ NullPointer \rfloor, h, (stk, loc, C_0, M_0, pc, ics) \# frs, sh)$

$| exec_step_ind_Getfield_NoField:$
 $\llbracket v = hd\ stk; (D, fs) = the(h(the_Addr\ v)); v \neq Null; \neg(\exists t\ b. P \vdash D\ has\ F, b:t\ in\ C) \rrbracket$
 $\implies exec_step_ind\ (StepI\ (Getfield\ F\ C))\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
 $(\lfloor addr_of_sys_xcpt\ NoSuchFieldError \rfloor, h, (stk, loc, C_0, M_0, pc, ics) \# frs, sh)$

$| exec_step_ind_Getfield_Static:$
 $\llbracket v = hd\ stk; (D, fs) = the(h(the_Addr\ v)); v \neq Null; P \vdash D\ has\ F, Static:t\ in\ C \rrbracket$
 $\implies exec_step_ind\ (StepI\ (Getfield\ F\ C))\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
 $(\lfloor addr_of_sys_xcpt\ IncompatibleClassChangeError \rfloor, h, (stk, loc, C_0, M_0, pc, ics) \# frs, sh)$

$| exec_step_ind_Getfield:$
 $\llbracket v = hd\ stk; (D, fs) = the(h(the_Addr\ v)); (D', b, t) = field\ P\ C\ F; v \neq Null; P \vdash D\ has\ F, NonStatic:t\ in\ C \rrbracket$
 $\implies exec_step_ind\ (StepI\ (Getfield\ F\ C))\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
 $(None, h, (the(fs(F, C)) \# (tl\ stk), loc, C_0, M_0, pc+1, ics) \# frs, sh)$

$| exec_step_ind_Getstatic_NoField:$
 $\neg(\exists t\ b. P \vdash C\ has\ F, b:t\ in\ D)$
 $\implies exec_step_ind\ (StepI\ (Getstatic\ C\ F\ D))\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
 $(\lfloor addr_of_sys_xcpt\ NoSuchFieldError \rfloor, h, (stk, loc, C_0, M_0, pc, ics) \# frs, sh)$

$| exec_step_ind_Getstatic_NonStatic:$
 $P \vdash C\ has\ F, NonStatic:t\ in\ D$
 $\implies exec_step_ind\ (StepI\ (Getstatic\ C\ F\ D))\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
 $(\lfloor addr_of_sys_xcpt\ IncompatibleClassChangeError \rfloor, h, (stk, loc, C_0, M_0, pc, ics) \# frs, sh)$

$| exec_step_ind_Getstatic_Called:$

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

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

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

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

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

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

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

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

| *exec-step-ind-Putstatic-NonStatic*:
 $P \vdash C \text{ has } F, \text{NonStatic}:t \text{ in } D$
 $\implies \text{exec-step-ind } (\text{StepI } (\text{Putstatic } C \ F \ D)) \ P \ h \ \text{stk} \ \text{loc} \ C_0 \ M_0 \ \text{pc} \ \text{ics} \ \text{frs} \ sh$
 $(\lfloor \text{addr-of-sys-xcpt } \text{IncompatibleClassChangeError} \rfloor, h, (\text{stk}, \text{loc}, C_0, M_0, \text{pc}, \text{ics}) \# \text{frs}, sh)$

| *exec-step-ind-Putstatic-Called*:
 $\llbracket (D', b, t) = \text{field } P \ D \ F; P \vdash C \text{ has } F, \text{Static}:t \text{ in } D; \text{the}(sh \ D') = (sfs, i) \rrbracket$
 $\implies \text{exec-step-ind } (\text{StepI } (\text{Putstatic } C \ F \ D)) \ P \ h \ \text{stk} \ \text{loc} \ C_0 \ M_0 \ \text{pc} \ (\text{Called } Cs)$
 $\text{frs} \ sh$
 $(\text{None}, h, (tl \ \text{stk}, \text{loc}, C_0, M_0, \text{Suc } \text{pc}, \text{No-ics}) \# \text{frs}, sh(D' := \text{Some } ((sfs(F \mapsto hd \ \text{stk})), i)))$

| *exec-step-ind-Putstatic-Done*:
 $\llbracket (D', b, t) = \text{field } P \ D \ F; P \vdash C \text{ has } F, \text{Static}:t \text{ in } D;$
 $\forall Cs. \text{ics} \neq \text{Called } Cs; sh \ D' = \text{Some } (sfs, \text{Done}) \rrbracket$
 $\implies \text{exec-step-ind } (\text{StepI } (\text{Putstatic } C \ F \ D)) \ P \ h \ \text{stk} \ \text{loc} \ C_0 \ M_0 \ \text{pc} \ \text{ics} \ \text{frs} \ sh$
 $(\text{None}, h, (tl \ \text{stk}, \text{loc}, C_0, M_0, \text{Suc } \text{pc}, \text{ics}) \# \text{frs}, sh(D' := \text{Some } ((sfs(F \mapsto hd \ \text{stk})), \text{Done}))))$

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

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

| *exec-step-ind-Checkcast-Error*:
 $\neg \text{cast-ok } P \ C \ h \ (hd \ \text{stk})$
 $\implies \text{exec-step-ind } (\text{StepI } (\text{Checkcast } C)) \ P \ h \ \text{stk} \ \text{loc} \ C_0 \ M_0 \ \text{pc} \ \text{ics} \ \text{frs} \ sh$
 $(\lfloor \text{addr-of-sys-xcpt } \text{ClassCast} \rfloor, h, (\text{stk}, \text{loc}, C_0, M_0, \text{pc}, \text{ics}) \# \text{frs}, sh)$

| *exec-step-ind-Invoke-Null*:
 $\text{stk}!n = \text{Null}$
 $\implies \text{exec-step-ind } (\text{StepI } (\text{Invoke } M \ n)) \ P \ h \ \text{stk} \ \text{loc} \ C_0 \ M_0 \ \text{pc} \ \text{ics} \ \text{frs} \ sh$
 $(\lfloor \text{addr-of-sys-xcpt } \text{NullPointerException} \rfloor, h, (\text{stk}, \text{loc}, C_0, M_0, \text{pc}, \text{ics}) \# \text{frs}, sh)$

| *exec-step-ind-Invoke-NoMethod*:
 $\llbracket r = \text{stk}!n; C = \text{fst}(\text{the}(h(\text{the-Addr } r))); r \neq \text{Null};$
 $\neg(\exists Ts \ T \ m \ D \ b. P \vdash C \text{ sees } M, b:Ts \rightarrow T = m \text{ in } D) \rrbracket$
 $\implies \text{exec-step-ind } (\text{StepI } (\text{Invoke } M \ n)) \ P \ h \ \text{stk} \ \text{loc} \ C_0 \ M_0 \ \text{pc} \ \text{ics} \ \text{frs} \ sh$
 $(\lfloor \text{addr-of-sys-xcpt } \text{NoSuchMethodError} \rfloor, h, (\text{stk}, \text{loc}, C_0, M_0, \text{pc}, \text{ics}) \# \text{frs}, sh)$

| *exec-step-ind-Invoke-Static*:
 $\llbracket r = \text{stk}!n; C = \text{fst}(\text{the}(h(\text{the-Addr } r)));$

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

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

$\mid \text{exec-step-ind-Invokestatic-NoMethod:}$
 $\parallel (D, b, Ts, T, mxs, mxl_0, ins, xt) = \text{method } P \ C \ M; \neg(\exists Ts \ T \ m \ D \ b. P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D) \parallel$
 $\Rightarrow \text{exec-step-ind } (\text{StepI } (\text{Invokestatic } C \ M \ n)) \ P \ h \ stk \ loc \ C_0 \ M_0 \ pc \ ics \ frs \ sh$
 $(\lfloor \text{addr-of-sys-xcpt } \text{NoSuchMethodError} \rfloor, h, (stk, loc, C_0, M_0, pc, ics) \# frs, sh)$

$\mid \text{exec-step-ind-Invokestatic-NonStatic:}$
 $\parallel (D, b, Ts, T, mxs, mxl_0, ins, xt) = \text{method } P \ C \ M; P \vdash C \text{ sees } M, \text{NonStatic}: Ts \rightarrow T = m \text{ in } D \parallel$
 $\Rightarrow \text{exec-step-ind } (\text{StepI } (\text{Invokestatic } C \ M \ n)) \ P \ h \ stk \ loc \ C_0 \ M_0 \ pc \ ics \ frs \ sh$
 $(\lfloor \text{addr-of-sys-xcpt } \text{IncompatibleClassChangeError} \rfloor, h, (stk, loc, C_0, M_0, pc, ics) \# frs, sh)$

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

$\mid \text{exec-step-ind-Invokestatic-Done:}$
 $\parallel ps = \text{take } n \ stk; (D, b, Ts, T, mxs, mxl_0, ins, xt) = \text{method } P \ C \ M;$
 $P \vdash C \text{ sees } M, \text{Static}: Ts \rightarrow T = m \text{ in } D;$
 $\forall Cs. ics \neq \text{Called } Cs; sh \ D = \text{Some } (sfs, \text{Done});$
 $f' = (\parallel, (\text{rev } ps) @ (\text{replicate } mxl_0 \ \text{undefined}), D, M, 0, \text{No-ics}) \parallel$
 $\Rightarrow \text{exec-step-ind } (\text{StepI } (\text{Invokestatic } C \ M \ n)) \ P \ h \ stk \ loc \ C_0 \ M_0 \ pc \ ics \ frs \ sh$
 $(\text{None}, h, f' \# (stk, loc, C_0, M_0, pc, ics) \# frs, sh)$

$\mid \text{exec-step-ind-Invokestatic-Init:}$
 $\parallel (D, b, Ts, T, mxs, mxl_0, ins, xt) = \text{method } P \ C \ M;$
 $P \vdash C \text{ sees } M, \text{Static}: Ts \rightarrow T = m \text{ in } D;$
 $\forall sfs. sh \ D \neq \text{Some } (sfs, \text{Done}); \forall Cs. ics \neq \text{Called } Cs \parallel$
 $\Rightarrow \text{exec-step-ind } (\text{StepI } (\text{Invokestatic } C \ M \ n)) \ P \ h \ stk \ loc \ C_0 \ M_0 \ pc \ ics \ frs \ sh$
 $(\text{None}, h, (stk, loc, C_0, M_0, pc, \text{Calling } D \ \parallel) \# frs, sh)$

| *exec-step-ind-Return-Last-Init*:
 $\text{exec-step-ind } (\text{StepI Return}) P h \text{ stk}_0 \text{ loc}_0 C_0 \text{ clinit } pc \text{ ics } [] sh$
 $(None, h, [], sh(C_0 := \text{Some}(\text{fst}(\text{the}(sh C_0))), Done)))$

| *exec-step-ind-Return-Last*:
 $M_0 \neq \text{clinit}$
 $\implies \text{exec-step-ind } (\text{StepI Return}) P h \text{ stk}_0 \text{ loc}_0 C_0 M_0 pc \text{ ics } [] sh (None, h, [], sh)$

| *exec-step-ind-Return-Init*:
 $\llbracket (D, b, Ts, T, m) = \text{method } P C_0 \text{ clinit} \rrbracket$
 $\implies \text{exec-step-ind } (\text{StepI Return}) P h \text{ stk}_0 \text{ loc}_0 C_0 \text{ clinit } pc \text{ ics } ((\text{stk}', \text{loc}', C', m', pc', \text{ics}') \# \text{frs}')$
 sh
 $(None, h, (\text{stk}', \text{loc}', C', m', pc', \text{ics}') \# \text{frs}', sh(C_0 := \text{Some}(\text{fst}(\text{the}(sh C_0))), Done)))$

| *exec-step-ind-Return-NonStatic*:
 $\llbracket (D, \text{NonStatic}, Ts, T, m) = \text{method } P C_0 M_0; M_0 \neq \text{clinit} \rrbracket$
 $\implies \text{exec-step-ind } (\text{StepI Return}) P h \text{ stk}_0 \text{ loc}_0 C_0 M_0 pc \text{ ics } ((\text{stk}', \text{loc}', C', m', pc', \text{ics}') \# \text{frs}')$
 sh
 $(None, h, ((hd \text{ stk}_0) \# (\text{drop } (\text{length } Ts + 1) \text{ stk}'), \text{loc}', C', m', \text{Suc } pc', \text{ics}') \# \text{frs}', sh)$

| *exec-step-ind-Return-Static*:
 $\llbracket (D, \text{Static}, Ts, T, m) = \text{method } P C_0 M_0; M_0 \neq \text{clinit} \rrbracket$
 $\implies \text{exec-step-ind } (\text{StepI Return}) P h \text{ stk}_0 \text{ loc}_0 C_0 M_0 pc \text{ ics } ((\text{stk}', \text{loc}', C', m', pc', \text{ics}') \# \text{frs}')$
 sh
 $(None, h, ((hd \text{ stk}_0) \# (\text{drop } (\text{length } Ts) \text{ stk}'), \text{loc}', C', m', \text{Suc } pc', \text{ics}') \# \text{frs}', sh)$

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

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

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

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

| *exec-step-ind-CmpEq*:

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

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

$| exec\text{-}step\text{-}ind\text{-}Throw:$
 $hd\ stk \neq Null$
 $\implies exec\text{-}step\text{-}ind\ (StepI\ Throw)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
 $([the\text{-}Addr\ (hd\ stk)], h, (stk, loc, C_0, M_0, pc, ics)\#frs, sh)$

$| exec\text{-}step\text{-}ind\text{-}Throw\text{-}Null:$
 $hd\ stk = Null$
 $\implies exec\text{-}step\text{-}ind\ (StepI\ Throw)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
 $([addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer], h, (stk, loc, C_0, M_0, pc, ics)\#frs, sh)$

$| exec\text{-}step\text{-}ind\text{-}Init\text{-}None\text{-}Called:$
 $\llbracket sh\ C = None \rrbracket$
 $\implies exec\text{-}step\text{-}ind\ (StepC\ C\ Cs)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
 $(None, h, (stk, loc, C_0, M_0, pc, Calling\ C\ Cs)\#frs, sh(C := Some\ (sblank\ P$
 $C, Prepared)))$

$| exec\text{-}step\text{-}ind\text{-}Init\text{-}Done:$
 $sh\ C = Some\ (sfs, Done)$
 $\implies exec\text{-}step\text{-}ind\ (StepC\ C\ Cs)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
 $(None, h, (stk, loc, C_0, M_0, pc, Called\ Cs)\#frs, sh)$

$| exec\text{-}step\text{-}ind\text{-}Init\text{-}Processing:$
 $sh\ C = Some\ (sfs, Processing)$
 $\implies exec\text{-}step\text{-}ind\ (StepC\ C\ Cs)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
 $(None, h, (stk, loc, C_0, M_0, pc, Called\ Cs)\#frs, sh)$

$| exec\text{-}step\text{-}ind\text{-}Init\text{-}Error:$
 $\llbracket sh\ C = Some\ (sfs, Error) \rrbracket$
 $\implies exec\text{-}step\text{-}ind\ (StepC\ C\ Cs)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
 $(None, h, (stk, loc, C_0, M_0, pc, Throwing\ Cs\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NoClassDef$
 $FoundError))\#frs, sh)$

$| exec\text{-}step\text{-}ind\text{-}Init\text{-}Prepared\text{-}Object:$
 $\llbracket sh\ C = Some\ (sfs, Prepared);$
 $sh' = sh(C := Some(fst(the(sh\ C)), Processing));$
 $C = Object \rrbracket$
 $\implies exec\text{-}step\text{-}ind\ (StepC\ C\ Cs)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
 $(None, h, (stk, loc, C_0, M_0, pc, Called\ (C\#Cs))\#frs, sh')$

$| exec\text{-}step\text{-}ind\text{-}Init\text{-}Prepared\text{-}nObject:$
 $\llbracket sh\ C = Some\ (sfs, Prepared);$

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

$\mid \text{exec-step-ind-Init:}$
 $\text{exec-step-ind } (\text{StepC2 } C\ Cs) P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
 $(\text{None}, h, \text{create-init-frame } P\ C \# (stk, loc, C_0, M_0, pc, \text{Called } Cs) \# frs, sh)$

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

$\mid \text{exec-step-ind-InitThrow-End:}$
 $\text{exec-step-ind } (\text{StepT } []\ a) P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$
 $([a], h, (stk, loc, C_0, M_0, pc, \text{No-ics}) \# frs, sh)$

inductive-cases *exec-step-ind-cases* [cases set]:

$\text{exec-step-ind } (\text{StepI } (\text{Load } n)) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepI } (\text{Store } n)) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepI } (\text{Push } v)) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepI } (\text{New } C)) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepI } (\text{Getfield } F\ C)) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepI } (\text{Getstatic } C\ F\ D)) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepI } (\text{Putfield } F\ C)) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepI } (\text{Putstatic } C\ F\ D)) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepI } (\text{Checkcast } C)) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepI } (\text{Invoke } M\ n)) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepI } (\text{Invokestatic } C\ M\ n)) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepI } \text{Return}) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepI } \text{Pop}) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepI } \text{IAdd}) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepI } (\text{IfFalse } i)) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepI } \text{CmpEq}) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepI } (\text{Goto } i)) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepI } \text{Throw}) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepC } C'\ Cs) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepC2 } C'\ Cs) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$
 $\text{exec-step-ind } (\text{StepT } Cs\ a) P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh\ \sigma$

— 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

$\text{exec-step-input } P\ C\ M\ pc\ (\text{Calling } C'\ Cs) = \text{StepC } C'\ Cs \mid$
 $\text{exec-step-input } P\ C\ M\ pc\ (\text{Called } (C' \# Cs)) = \text{StepC2 } C'\ Cs \mid$

$exec\text{-}step\text{-}input\ P\ C\ M\ pc\ (Throwing\ Cs\ a) = StepT\ Cs\ a \mid$
 $exec\text{-}step\text{-}input\ P\ C\ M\ pc\ ics = StepI\ (instrs\text{-}of\ P\ C\ M\ !\ pc)$

lemma *exec-step-input-StepTD[simp]*:
assumes $exec\text{-}step\text{-}input\ P\ C\ M\ pc\ ics = StepT\ Cs\ a$ **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 $exec\text{-}step\text{-}input\ P\ C\ M\ pc\ ics = StepC\ C'\ Cs$ **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 $exec\text{-}step\text{-}input\ P\ C\ M\ pc\ ics = StepC2\ C'\ Cs$ **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 $exec\text{-}step\text{-}input\ P\ C\ M\ pc\ ics = StepI\ i$
shows $(ics = Called\ [] \vee ics = No\text{-}ics) \wedge instrs\text{-}of\ P\ C\ M\ !\ pc = i$
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 $es: exec\text{-}step\ P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh = (xp', h', frs', sh')$
shows $exec\text{-}step\text{-}ind\ (exec\text{-}step\text{-}input\ P\ C\ M\ pc\ ics)\ P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh$
 (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 $ics = Calling\ C1\ Cs$ **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 $m\ x\ m\ x_0\ ins\ xt$ **where** $m: m = (m\ x\ s, m\ x_0, ins, xt)$ **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  $\bigwedge b' t'. P \vdash D \text{ has } F1, b':t' \text{ in } C1 \implies (D', b, t) = (C1, b', t')$ 
      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:  $\bigwedge b' t'. P \vdash C1 \text{ has } F1, b':t' \text{ in } D1 \implies (D', b, t) = (D1, b', t')$ 
    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:  $\forall Cs. ics \neq \text{Called } Cs$  using ics by simp
          show ?thesis
          proof(cases sh D1)
            case None
            then have nDone:  $\forall sfs. sh \ D1 \neq \text{Some}(sfs, \text{Done})$  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:  $\forall sfs. sh \ D1 \neq \text{Some}(sfs, \text{Done})$  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
  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  $\bigwedge b' t'. P \vdash D \text{ has } F1, b':t' \text{ in } C1 \implies (D', b, t) = (C1, b', t')$ 
    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:  $\bigwedge b' t'. P \vdash C1 \text{ has } F1, b':t' \text{ in } D1 \implies (D', b, t) = (D1, b', t')$ 
    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:  $\forall Cs. ics \neq Called \ Cs$  using ics by simp
  show ?thesis

```

```

proof(cases sh D1)
  case None
  then have nDone:  $\forall$  sfs. sh D1  $\neq$  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:  $\forall$  sfs. sh D1  $\neq$  Some(sfs, Done) using sfsi Some by
simp
      show ?thesis using nD
      proof(cases i)
      case Processing then show ?thesis using exec-step-ind-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  $stk!n = \text{Null}$ )
  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 mxl0 ins xt where m = (mxs,mxl0,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 mxl0 ins xt where m: m = (mxs,mxl0,ins,xt) by(cases m)
        have method:  $\bigwedge b' Ts' t' m' D'. P \vdash C1 \text{ sees } M1, b': Ts' \rightarrow t' = m' \text{ in } D'$ 
           $\Rightarrow (D,b,Ts,T,m) = (D',b',Ts',t',m')$  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:  $\forall Cs. ics \neq \text{Called } Cs$  using ics by simp
            show ?thesis
            proof(cases sh D)
              case None
              then have nDone:  $\forall sfs. sh D \neq \text{Some}(sfs, \text{Done})$  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:  $\forall sfs. sh D \neq \text{Some}(sfs, Done)$  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 mxl0 ins xt where m = (mxs,mxl0,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:
     $\bigwedge P C M pc Cs i . \text{exec-step-input } P C M pc (\text{Called } Cs) = \text{StepI } i$ 
     $\implies \text{instrs-of } P C M ! pc = i \wedge Cs = []$ 
  proof -
    fix P C M pc Cs i show exec-step-input P C M pc (Called Cs) = StepI i
     $\implies \text{instrs-of } P C M ! pc = i \wedge Cs = []$  by(cases Cs; simp)
  qed
  have StepC:
     $\bigwedge P C M pc ics C' Cs . \text{exec-step-input } P C M pc ics = \text{StepC } C' Cs \implies ics =$ 
    Calling C' Cs
    by simp
  have StepT:
     $\bigwedge P C M pc ics Cs a . \text{exec-step-input } P C M pc ics = \text{StepT } Cs a \implies ics =$ 
    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 C0 M0 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 C0 M0 pc frs)
    then show ?case by(cases ics, auto)
  next
    case (exec-step-ind-New-Init sh C ics P h stk loc C0 M0 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 C0 M0 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 C0 M0 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 C0 M0 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 C0 M0 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 C0 M0
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 C0 M0 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 C0 M0 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 C0 M0 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 C0 M0 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 C0 M0 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 C0 M0
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 C0 M0 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 mxl0 ins xt P M m f'
loc C0 M0 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 mxl0 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 mxl0 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 mxl0 ins xt P C M m sh ics
n h stk loc C0 M0 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 C0 M0 h stk0 loc0 pc ics stk'
loc' C' m' pc' ics' frs' sh)
  then show ?case by(cases method P C0 M0, cases ics, auto dest!: StepI)
next
  case (exec-step-ind-Return-Static D Ts T m P C0 M0 h stk0 loc0 pc ics stk'
loc' C' m' pc' ics' frs' sh)
  then show ?case by(cases method P C0 M0, cases ics, auto dest!: StepI)
next
  case (exec-step-ind-IfFalse-nFalse stk i P h loc C0 M0 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 C0 M0 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 C0 M0 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 C0 M0 pc ics frs sh)
  then show ?case by(cases ics, auto dest!: StepI)
next
  case (exec-step-ind-Store n P h stk loc C0 M0 pc ics frs sh)
  then show ?case by(cases ics, auto dest!: StepI)
next
  case (exec-step-ind-Push v P h stk loc C0 M0 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 C0 M0 pc frs sh ics')
  then show ?case by(auto dest!: StepI)
next
  case (exec-step-ind-New-Called h a C P stk loc C0 M0 pc frs sh ics')
  then show ?case by(auto dest!: StepI)
next
  case (exec-step-ind-Getfield-Null stk F C P h loc C0 M0 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 C0 M0 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 C0 M0 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 C0 M0 pc ics frs sh)
  then show ?case by(cases ics, auto dest!: StepI)
next
  case (exec-step-ind-Checkcast P C h stk loc C0 M0 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 C0 M0 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 C0 M0 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 C0 M0 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 mxtl0 ins xt P M m

```

```

loc C0 M0 pc ics)
  then show ?case by(cases ics, auto dest!: StepI)
next
  case (exec-step-ind-Invokestatic-NoMethod D b Ts T mxs mxl0 ins xt P C M n
h stk loc C0 M0 pc ics)
  then show ?case by(cases ics, auto dest!: StepI)
next
  case (exec-step-ind-Invokestatic-NonStatic D b Ts T mxs mxl0 ins xt P C M m
n h stk loc C0 M0 pc ics)
  then show ?case by(cases ics, auto dest!: StepI)
next
  case (exec-step-ind-Return-Last-Init P h stk0 loc0 C0 pc ics sh)
  then show ?case by(cases ics, auto dest!: StepI)
next
  case (exec-step-ind-Return-Last M0 P h stk0 loc0 C0 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 C0 h stk0 loc0 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 C0 M0 pc ics frs sh)
  then show ?case by(cases ics, auto dest!: StepI)
next
  case (exec-step-ind-IAdd P h stk loc C0 M0 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 C0 M0 pc ics frs sh)
  then show ?case by(cases ics, auto dest!: StepI)
next
  case (exec-step-ind-CmpEq P h stk loc C0 M0 pc ics frs sh)
  then show ?case by(cases ics, auto dest!: StepI)
next
  case (exec-step-ind-Goto i P h stk loc C0 M0 pc ics frs sh)
  then show ?case by(cases ics, auto dest!: StepI)
next
  case (exec-step-ind-Throw stk P h loc C0 M0 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 C0 M0 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 C0 M0 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 C0 M0 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 C0 M0 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 C0 M0 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 C0 M0
pc ics frs)
    then show ?case by(auto dest!: StepC)
  next
    case (exec-step-ind-InitThrow C Cs a P h stk loc C0 M0 pc ics frs sh)
    then show ?case by(auto dest!: StepT)
  next
    case (exec-step-ind-InitThrow-End a P h stk loc C0 M0 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** *¬P ⊢ D ≼* 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*: $\neg \text{is-class } P \ C$
and *nep*: $\bigwedge D. D \in \text{sys-xcpts} \implies \neg P \vdash D \preceq^* C$
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*: $\sigma' \in \text{JVMsmall } P \ \sigma$
and *nend*: $\sigma \notin \text{JVMendset}$
and *curr*: *curr-instr* *P* (*hd*(*frames-of* σ)) = *New* *C*
and *nDone*: $\neg(\exists \text{sfs } i. \text{sheap } \sigma \ C = \text{Some}(\text{sfs}, i) \wedge i = \text{Done})$
and *ics*: *ics-of*(*hd*(*frames-of* σ)) = *No-ics*
shows *ics-of* (*hd*(*frames-of* σ')) = *Calling* *C* [] $\wedge \text{sheap } \sigma = \text{sheap } \sigma' \wedge \sigma' \notin \text{JVMendset}$
proof –
obtain *xp h frs sh* **where** $\sigma: \sigma=(xp, h, frs, sh)$ **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** $\sigma': \sigma'=(xp', h', frs', sh')$ **by**(*cases* σ')
have *ics-of* (*hd* *frs'*) = *Calling* *C* [] $\wedge sh = sh' \wedge frs' \neq [] \wedge xp' = \text{None}$
proof(*cases* *sh* *C*)
case *None* **then show** *?thesis* **using** $\sigma' xp f1 frs \sigma$ *assms* **by** *auto*
next
case (*Some* *a*)
then obtain *sfs i* **where** *a*: *a*=(*sfs*, *i*) **by**(*cases* *a*)
then have *nDone'*: *i* $\neq \text{Done}$ **using** *nDone* *Some* *f1 frs* σ **by** *simp*
show *?thesis* **using** *a* *Some* $\sigma' xp f1 frs \sigma$ *assms* **by**(*auto* *split*: *init-state.splits*)
qed
then show *?thesis* **using** *ics* $\sigma \sigma'$ **by**(*cases* *frs'*, *auto* *simp*: *JVMendset-def*)
qed

lemma *JVM-Getstatic-next-step*:
assumes *step*: $\sigma' \in \text{JVMsmall } P \ \sigma$
and *nend*: $\sigma \notin \text{JVMendset}$
and *curr*: *curr-instr* *P* (*hd*(*frames-of* σ)) = *Getstatic* *C F D*
and *fC*: $P \vdash C \text{ has } F, \text{Static}:t \text{ in } D$
and *nDone*: $\neg(\exists \text{sfs } i. \text{sheap } \sigma \ D = \text{Some}(\text{sfs}, i) \wedge i = \text{Done})$
and *ics*: *ics-of*(*hd*(*frames-of* σ)) = *No-ics*
shows *ics-of* (*hd*(*frames-of* σ')) = *Calling* *D* [] $\wedge \text{sheap } \sigma = \text{sheap } \sigma' \wedge \sigma' \notin \text{JVMendset}$
proof –
obtain *xp h frs sh* **where** $\sigma: \sigma=(xp, h, frs, sh)$ **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** $\sigma\ nend$ **by**(simp add: $JVMendset-def$)
obtain $xp'\ h'\ frs'\ sh'$ **where** $\sigma': \sigma'=(xp',h',frs',sh')$ **by**(cases σ')
have $ex': \exists t\ b.\ P \vdash C\ has\ F,b:t\ in\ D$ **using** fC **by** *auto*
have $field: \exists t.\ field\ P\ D\ F = (D,Static,t)$
using $fC\ field-def2\ has-field-idemp\ has-field-sees$ **by** *blast*
have $nCalled': \forall Cs.\ ics \neq Called\ Cs$ **using** $ics\ f1\ frs\ \sigma$ **by** *simp*
have $ics-of\ (hd\ frs') = Calling\ D\ [] \wedge sh = sh' \wedge frs' \neq [] \wedge xp' = None$
proof(cases $sh\ D$)
case *None* **then show** *?thesis* **using** $field\ ex'\ \sigma'\ xp\ f1\ frs\ \sigma\ assms$ **by** *auto*
next
case (*Some a*)
then obtain $sfs\ i$ **where** $a=(sfs,i)$ **by**(cases a)
show *?thesis* **using** $field\ ex'\ a\ Some\ \sigma'\ xp\ f1\ frs\ \sigma\ assms$ **by**(*auto split: init-state.splits*)
qed
then show *?thesis* **using** $ics\ \sigma\ \sigma'$ **by**(*auto simp: JVMendset-def*)
qed

lemma *JVM-Putstatic-next-step:*

assumes $step: \sigma' \in JVMsmall\ P\ \sigma$

and $nend: \sigma \notin JVMendset$

and $curr: curr-instr\ P\ (hd(frames-of\ \sigma)) = Putstatic\ C\ F\ D$

and $fC: P \vdash C\ has\ F,Static:t\ in\ D$

and $nDone: \neg(\exists sfs\ i.\ sheap\ \sigma\ D = Some(sfs,i) \wedge i = Done)$

and $ics: ics-of(hd(frames-of\ \sigma)) = No-ics$

shows $ics-of\ (hd(frames-of\ \sigma')) = Calling\ D\ [] \wedge sheap\ \sigma = sheap\ \sigma' \wedge \sigma' \notin JVMendset$

proof –

obtain $xp\ h\ frs\ sh$ **where** $\sigma: \sigma=(xp,h,frs,sh)$ **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** $\sigma\ nend$ **by**(simp add: $JVMendset-def$)

obtain $xp'\ h'\ frs'\ sh'$ **where** $\sigma': \sigma'=(xp',h',frs',sh')$ **by**(cases σ')

have $ex': \exists t\ b.\ P \vdash C\ has\ F,b:t\ in\ D$ **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 $ics': ics-of\ (hd\ frs') = Calling\ D\ [] \wedge sh = sh' \wedge frs' \neq [] \wedge xp' = None$

proof(cases $sh\ D$)

case *None* **then show** *?thesis* **using** $field\ ex'\ \sigma'\ xp\ f1\ frs\ \sigma\ assms$ **by** *auto*

next

case (*Some a*)

then obtain $sfs\ i$ **where** $a=(sfs,i)$ **by**(cases a)

show *?thesis* **using** $field\ ex'\ a\ Some\ \sigma'\ xp\ f1\ frs\ \sigma\ assms$ **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*: $\sigma' \in \text{JVMsmall } P \ \sigma$
 and *nend*: $\sigma \notin \text{JVMendset}$
 and *curr*: $\text{curr-instr } P \ (\text{hd}(\text{frames-of } \sigma)) = \text{Invokestatic } C \ M \ n$
 and *mC*: $P \vdash C \text{ sees } M, \text{Static}: Ts \rightarrow T = m \text{ in } D$
 and *nDone*: $\neg(\exists \text{sfs } i. \text{sheap } \sigma \ D = \text{Some}(\text{sfs}, i) \wedge i = \text{Done})$
 and *ics*: $\text{ics-of}(\text{hd}(\text{frames-of } \sigma)) = \text{No-ics}$
 shows $\text{ics-of}(\text{hd}(\text{frames-of } \sigma')) = \text{Calling } D \ [] \wedge \text{sheap } \sigma = \text{sheap } \sigma' \wedge \sigma' \notin \text{JVMendset}$
 proof –
 obtain *xp h frs sh* where $\sigma = (xp, h, frs, sh)$ by (cases σ)
 then obtain *f1 frs1* where $\text{frs} = f1 \# frs1$ using *nend* by (cases *frs*, simp-all add: *JVMendset-def*)
 then obtain *stk loc C' M' pc ics* where $f1: f1 = (\text{stk}, \text{loc}, C', M', \text{pc}, \text{ics})$ by (cases *f1*)
 have *xp*: $xp = \text{None}$ using σ *nend* by (simp add: *JVMendset-def*)
 obtain *xp' h' frs' sh'* where $\sigma' = (xp', h', frs', sh')$ by (cases σ')
 have *ex'*: $\exists Ts \ T \ m \ D \ b. P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D$ using *mC* by *fastforce*
 have *method*: $\exists m. \text{method } P \ C \ M = (D, \text{Static}, m)$ using *mC* by (cases *m*, auto)
 have *ics'*: $\text{ics-of}(\text{hd } frs') = \text{Calling } D \ [] \wedge sh = sh' \wedge frs' \neq [] \wedge xp' = \text{None}$
 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 $a = (\text{sfs}, i)$ by (cases *a*)
 then have *nDone'*: $i \neq \text{Done}$ 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 $\equiv \{P. \text{wf-jvm-prog } P \wedge \neg \text{is-class } P \text{ Start} \wedge \neg \text{is-class } P \text{ Test}$
 $\wedge (\forall b' Ts' T' m' D'. P \vdash \text{Object sees start-}m, b' : Ts' \rightarrow T' = m' \text{ in } D'$
 $\rightarrow b' = \text{Static} \wedge Ts' = [] \wedge T' = \text{Void}) \}$

definition *jvm-tests* :: *jvm-class* set **where**
jvm-tests $= \{t. \text{fst } t = \text{Test}$
 $\wedge (\forall P \in \text{jvm-progs}. \text{wf-jvm-prog } (t\#P) \wedge (\exists m. t\#P \vdash \text{Test sees main, Static: } []$
 $\rightarrow \text{Void} = m \text{ in } \text{Test})) \}$

abbreviation *jvm-make-test-prog* :: *jvm-prog* $\Rightarrow$ *jvm-class* $\Rightarrow$ *jvm-prog* **where**
jvm-make-test-prog $P \ t \equiv \text{start-prog } (t\#P) \ (\text{fst } t) \ \text{main}$

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

12.4 Definition lemmas

lemma *jvm-progs-tests-nStart*:
assumes $P: P \in \text{jvm-progs}$ **and** $t: t \in \text{jvm-tests}$
shows $\neg \text{is-class } (t\#P) \text{ Start}$
using *assms* **by** (*simp add: is-class-def class-def Start-def Test-def*)

lemma *jvm-make-test-prog-classes-above-xcpts*:
assumes $P: P \in \text{jvm-progs}$ **and** $t: t \in \text{jvm-tests}$
shows *classes-above-xcpts* (*jvm-make-test-prog* $P \ t$) = *classes-above-xcpts* P
proof –
have $nS: \neg \text{is-class } (t\#P) \text{ Start}$ **by** (*rule jvm-progs-tests-nStart[OF P t]*)
from P **have** $nT: \neg \text{is-class } P \text{ Test}$ **by** *simp*
from $P \ t$ **have** $\text{wf-syscls } (t\#P) \wedge \text{wf-syscls } P$
by (*simp add: wf-jvm-prog-def wf-jvm-prog-phi-def wf-prog-def*)

then have [*simp*]: $\bigwedge D. D \in \text{sys-xcpts} \implies \text{is-class } (t\#P) \ D \wedge \text{is-class } P \ D$
by (*cases t, auto simp: wf-syscls-def is-class-def class-def dest!: weak-map-of-SomeI*)
from $\text{wf-nclass-nsub}[OF \ - \ nS] \ P \ t$ **have** $nspS: \bigwedge D. D \in \text{sys-xcpts} \implies \neg(t\#P)$
 $\vdash D \preceq^* \text{Start}$
by (*auto simp: wf-jvm-prog-def wf-jvm-prog-phi-def*)
from $\text{wf-nclass-nsub}[OF \ - \ nT] \ P$ **have** $nspT: \bigwedge D. D \in \text{sys-xcpts} \implies \neg P \vdash D$
 $\preceq^* \text{Test}$
by (*auto simp: wf-jvm-prog-def wf-jvm-prog-phi-def*)

from *class-add-classes-above-xcpts* [**where** $P=t\#P$ **and** $C=\text{Start}$, *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** $C=\text{Test}$, *OF* $nT \ nspT$]
 t
have $\dots = \text{classes-above-xcpts } P$ **by** (*cases t, simp*)
finally show *?thesis* **by** *simp*
qed

lemma *jvm-make-test-prog-sees-Test-main*:
assumes $P: P \in \text{jvm-progs}$ **and** $t: t \in \text{jvm-tests}$
shows $\exists m. \text{jvm-make-test-prog } P \ t \vdash \text{Test sees main, Static : } [] \rightarrow \text{Void} = m \text{ in Test}$
proof –
let $?P = \text{jvm-make-test-prog } P \ t$
from $P \ t$ **obtain** m **where**
 $\text{meth: } t \# P \vdash \text{Test sees main, Static: } [] \rightarrow \text{Void} = m \text{ in Test}$ **and**
 $\text{nstart: } \neg \text{is-class } (t \# P) \text{ Start}$
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 $P \ t = \text{JVMNaiveCollectionSemantics.cbig } (\text{jvm-make-test-prog } P \ t)$
(*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 $o1 \in \text{jvm-naive-out } P \ t$
shows $\text{classes-above-xcpts } (\text{start-prog } (t \# P) \ (\text{fst } t) \ \text{main}) \subseteq \text{snd } o1$
using *assms*
proof –
obtain $\sigma' \ \text{coll'}$ **where** $o1 = (\sigma', \text{coll'})$ **and**
 $\text{cbig: } (\sigma', \text{coll'}) \in \text{JVMNaiveCollectionSemantics.cbig } (\text{start-prog } (t \# P) \ (\text{fst } t) \ \text{main}) \ (\text{start-state } (t \# P))$
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* $P \ i \ h \ \text{stk} \cap \text{classes-changed } P \ P' = \{\}$
and *above-C*: $\text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\}$
and *ics*: $\text{ics} = \text{Called } [] \vee \text{ics} = \text{No-ics}$

```

    and  $i: i = \text{instrs-of } P \ C \ M \ ! \ pc$ 
  shows  $\text{exec-instr } i \ P \ h \ stk \ loc \ C \ M \ pc \ ics \ frs \ sh = \text{exec-instr } i \ P' \ h \ stk \ loc \ C \ M \ pc \ ics \ frs \ sh$ 
  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  $hd \ stk = \text{Null}$ )
    case True then show  $?thesis$  using Getfield assms by simp
  next
    case False
    let  $?D = (\text{cname-of } h \ (\text{the-Addr } (hd \ stk)))$ 
    have  $D: \text{classes-above } P \ ?D \cap \text{classes-changed } P \ P' = \{\}$ 
      using False Getfield assms by simp
    show  $?thesis$ 
  proof(cases  $\exists b \ t. \ P \vdash \ ?D \text{ has } F1, b:t \text{ in } C1$ )
    case True
    then obtain  $b1 \ t1$  where  $P \vdash \ ?D \text{ has } F1, b1:t1 \text{ in } C1$  by auto
    then have  $\text{has}: P' \vdash \ ?D \text{ has } F1, b1:t1 \text{ in } C1$ 
      using Getfield assms classes-above-has-field[OF  $D$ ] by auto
    have  $P \vdash \ ?D \preceq^* C1$  using has-field-decl-above True by auto
    then have  $\text{classes-above } P \ C1 \subseteq \text{classes-above } P \ ?D$  by(rule classes-above-subcls-subset)
    then have  $C1: \text{classes-above } P \ C1 \cap \text{classes-changed } P \ P' = \{\}$  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 ( $h \ (\text{the-Addr } (hd \ stk)))$ ), auto)
  next
    case nex: False
    then have  $\nexists b \ t. \ P' \vdash \ ?D \text{ has } F1, b:t \text{ in } C1$ 
      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 ( $h \ (\text{the-Addr } (hd \ stk)))$ ), auto)
    qed
  qed
next
  case (Getstatic  $C1 \ F1 \ D1$ )
  then have  $C1: \text{classes-above } P \ C1 \cap \text{classes-changed } P \ P' = \{\}$  using assms by
auto
  show  $?thesis$ 
  proof(cases  $\exists b \ t. \ P \vdash \ C1 \text{ has } F1, b:t \text{ in } D1$ )
    case True
    then obtain  $b \ t$  where  $\text{meth}: P \vdash \ C1 \text{ has } F1, b:t \text{ in } D1$  by auto
    then have  $P \vdash \ C1 \preceq^* D1$  by(rule has-field-decl-above)

```

```

then have D1: classes-above P D1  $\cap$  classes-changed P P' = {}
  using C1 rtranc-trans by fastforce
have P'  $\vdash$  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  $\nexists$  b t. P'  $\vdash$  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  $\cap$  classes-changed P P' = {} using False Putfield
assms by simp
show ?thesis
proof(cases  $\exists$  b t. P  $\vdash$  ?D has F1,b:t in C1)
  case True
  then obtain b1 t1 where P  $\vdash$  ?D has F1,b1:t1 in C1 by auto
  then have has: P'  $\vdash$  ?D has F1,b1:t1 in C1
    using Putfield assms classes-above-has-field[OF D] by auto
  have P  $\vdash$  ?D  $\preceq^*$  C1 using has-field-decl-above True by auto
  then have classes-above P C1  $\subseteq$  classes-above P ?D by(rule classes-above-subcls-subset)
  then have C1: classes-above P C1  $\cap$  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  $\nexists$  b t. P'  $\vdash$  ?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  $\cap$  classes-changed P P' = {} using Putstatic
  assms by auto
  show ?thesis
  proof(cases  $\exists b\ t. P \vdash C1 \text{ has } F1, b:t \text{ in } D1$ )
    case True
    then obtain b t where meth: P  $\vdash$  C1 has F1, b:t in D1 by auto
    then have P  $\vdash$  C1  $\preceq^*$  D1 by(rule has-field-decl-above)
    then have D1: classes-above P D1  $\cap$  classes-changed P P' = {}
      using C1 rtranc1-trans by fastforce
    then have P'  $\vdash$  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  $\nexists b\ t. P' \vdash C1 \text{ has } F1, b:t \text{ in } D1$ 
      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
  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  $\cap$  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  $\exists Ts\ T\ m\ D\ b. P \vdash ?C\ sees\ M, b:Ts \rightarrow T = m\ in\ D$ )
  case nex: False
    then have  $\neg(\exists Ts\ T\ m\ D\ b. P' \vdash ?C\ sees\ M, b:Ts \rightarrow T = m\ in\ D)$ 
      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  $\exists Ts\ T\ m\ D\ b. P' \vdash ?C\ sees\ M, b:Ts \rightarrow T = m\ in\ D$ 
        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  $\cap$  classes-changed P P' = {}
      using assms by simp
    obtain D b Ts T m xs 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  $\exists Ts\ T\ m\ D\ b. P \vdash C1\ sees\ M, b:Ts \rightarrow T = m\ in\ D$ )
      case False
        then have  $\neg(\exists Ts\ T\ m\ D\ b. P' \vdash C1\ sees\ M, b:Ts \rightarrow T = m\ in\ D)$ 
          using classes-above-sees-method2[OF above, of M] by clarsimp
        then show ?thesis using False Invokestatic by simp
      next
        case True
          then have  $\exists Ts\ T\ m\ D\ b. P' \vdash C1\ sees\ M, b:Ts \rightarrow T = m\ in\ D$ 
            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 $stk \cap \text{classes-changed } P \ P' = \{\}$

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 $?D = \text{cname-of } h \ (\text{the-Addr } (\text{hd } stk))$

have *classes-above* $P \ ?D \cap \text{classes-changed } P \ P' = \{\}$ **using** *False Getfield*

assms **by** *auto*

then have *classes-above* $P \ ?D = \text{classes-above } P' \ ?D$

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* $P \ C1 \cap \text{classes-changed } P \ P' = \{\}$ **using** *assms* **by** *auto*

then have *classes-above* $P \ C1 = \text{classes-above } P' \ C1$

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 $?D = \text{cname-of } h \ (\text{the-Addr } (\text{hd } (\text{tl } stk)))$

have *classes-above* $P \ ?D \cap \text{classes-changed } P \ P' = \{\}$ **using** *False Putfield*

assms **by** *auto*

then have *classes-above* $P \ ?D = \text{classes-above } P' \ ?D$

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* $P \ C1 \cap \text{classes-changed } P \ P' = \{\}$ **using** *assms* **by** *auto*

then have *classes-above* $P \ C1 = \text{classes-above } P' \ C1$

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  $\cap$  classes-changed P P' = {}
  and above-C: classes-above P C  $\cap$  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  $\cap$  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  $\cap$  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 $\cap$ classes-changed P P' = {}

and above-C: classes-above P C $\cap$ 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 $\cap$ 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 $\cap$ classes-changed P P' = {}

and above-C': classes-above P (fst (method P C1 clinit)) $\cap$ 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*:
 $JVMexec\text{-}ncollect\ P\ (None, h, (stk, loc, C, M, pc, ics)\#frs, sh) \cap classes\text{-}changed\ P$
 $P' = \{\}$
 $\implies classes\text{-}above\text{-}frames\ P\ frs = classes\text{-}above\text{-}frames\ P'\ frs$
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*: $classes\text{-}above\ P\ C \cap classes\text{-}changed\ P\ P' = \{\}$ **using** *Cons*
by *auto*
show *?case* **using** $f\ Cons\ classes\text{-}above\text{-}subcls[OF\ above\text{-}C]$
 $classes\text{-}above\text{-}subcls2[OF\ above\text{-}C]$ **by** *auto*
qed(*auto*)

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

shows *JVMexec-ncollect* $P \sigma = \text{JVMexec-ncollect } P' \sigma$

proof –

obtain $xp \ h \ frs \ sh$ **where** $\sigma: \sigma = (xp, h, frs, sh)$ **by** $(\text{cases } \sigma)$

then show *?thesis* **using** *assms*

proof $(\text{cases } \exists x. xp = \text{Some } x \vee frs = [])$

case *False*

then obtain $stk \ loc \ C \ M \ pc \ ics \ frs'$ **where** $frs: frs = (stk, loc, C, M, pc, ics) \# frs'$

by $(\text{cases } frs, \text{auto})$

have *step*: *JVMstep-ncollect* $P \ h \ stk \ C \ M \ pc \ ics \cap \text{classes-changed } P P' = \{\}$

using *False* $\sigma \ frs \ assms$ **by** $(\text{cases } ics, \text{auto } \text{simp: JVMNaiveCollectionSemantics.csmall-def})$

have *above-C*: *classes-above* $P \ C \cap \text{classes-changed } P P' = \{\}$

using *False* $\sigma \ frs \ assms$ **by** $(\text{auto } \text{simp: JVMNaiveCollectionSemantics.csmall-def})$

have *frames*: *classes-above-frames* $P \ frs' = \text{classes-above-frames } P' \ frs'$

using *ncollect-classes-above-frames* $frs \ \sigma \ \text{False} \ assms$ **by** *simp*

have *xcpts*: *classes-above-xcpts* $P = \text{classes-above-xcpts } P'$

using *ncollect-classes-above-xcpts* $frs \ \sigma \ \text{False} \ assms$ **by** *simp*

show *?thesis* **using** *False* *xcpts* *frames* $frs \ \sigma \ \text{ncollect-JVMstep-ncollect}[OF \ \text{step above-C}]$

$\text{classes-above-subcls}[OF \ \text{above-C}] \ \text{classes-above-subcls2}[OF \ \text{above-C}]$

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* $P \ i \ h \ stk \cap \text{classes-changed } P P' = \{\}$

and *xpcollect*: *classes-above-xcpts* $P \cap \text{classes-changed } P P' = \{\}$

and *prealloc*: *preallocated* h

and $\sigma': \sigma' = \text{exec-instr } i \ P \ h \ stk \ loc \ C \ M \ pc \ ics' \ frs \ sh$

and $xp: \text{fst } \sigma' = \text{Some } a$

and $i: i = \text{instrs-of } P \ C \ M \ ! \ pc$

shows *classes-above* $P \ (\text{cname-of } h \ a) \cap \text{classes-changed } P P' = \{\}$

using *assms* *exec-instr-xcpts* $[OF \ \sigma' \ xp]$

proof $(\text{cases } i)$

case *Throw* **then show** *?thesis* **using** *assms* **by** $(\text{cases } hd \ stk, \text{fastforce+})$

qed (fastforce+)

— if collected classes unchanged, classes above an exception returned by *exec-step* unchanged

lemma *ncollect-exec-step-xcpts*:

assumes *collect*: *JVMstep-ncollect* $P \ h \ stk \ C \ M \ pc \ ics \cap \text{classes-changed } P P' = \{\}$

and *xpcollect*: *classes-above-xcpts* $P \cap \text{classes-changed } P P' = \{\}$

```

    and prealloc: preallocated h
    and  $\sigma'$ :  $\sigma' = \text{exec-step } P \ h \ \text{stk} \ \text{loc} \ C \ M \ \text{pc} \ \text{ics} \ \text{frs} \ sh$ 
    and  $xp$ :  $\text{fst } \sigma' = \text{Some } a$ 
shows classes-above  $P \ (\text{cname-of } h \ a) \cap \text{classes-changed } P \ P' = \{\}$ 
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:  $(\sigma', cset) \in \text{JVMNaiveCollectionSemantics.csmall } P \ \sigma$ 
    and intersect:  $cset \cap \text{classes-changed } P \ P' = \{\}$ 
    and prealloc: preallocated (fst(snd  $\sigma$ ))
shows  $(\sigma', cset) \in \text{JVMNaiveCollectionSemantics.csmall } P' \ \sigma$ 
proof -
  obtain  $xp \ h \ \text{frs} \ sh$  where  $\sigma$ :  $\sigma = (xp, h, \text{frs}, sh)$  by(cases  $\sigma$ )
  then have prealloc': preallocated h using prealloc by simp
  show ?thesis using  $\sigma$  assms
proof(cases  $\exists x. xp = \text{Some } x \vee \text{frs} = []$ )
  case False
  then obtain  $\text{stk} \ \text{loc} \ C \ M \ \text{pc} \ \text{ics} \ \text{frs}'$  where  $\text{frs}$ :  $\text{frs} = (\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics}) \# \text{frs}'$ 
    by(cases  $\text{frs}$ , auto)
  have step:  $\text{JVMstep-ncollect } P \ h \ \text{stk} \ C \ M \ \text{pc} \ \text{ics} \cap \text{classes-changed } P \ P' = \{\}$ 
    using False  $\sigma \ \text{frs}$  assms by(cases ics, auto simp: JVMNaiveCollectionSemantics.csmall-def)
  have above-C:  $\text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\}$ 
    using False  $\sigma \ \text{frs}$  assms by(auto simp: JVMNaiveCollectionSemantics.csmall-def)
  obtain  $xp1 \ h1 \ \text{frs1} \ sh1$  where  $\text{exec}$ :  $\text{exec-step } P' \ h \ \text{stk} \ \text{loc} \ C \ M \ \text{pc} \ \text{ics} \ \text{frs}' \ sh =$ 
    ( $xp1, h1, \text{frs1}, sh1$ )
    by(cases  $\text{exec-step } P' \ h \ \text{stk} \ \text{loc} \ C \ M \ \text{pc} \ \text{ics} \ \text{frs}' \ sh$ )
  have collect:  $\text{JVMexec-ncollect } P \ \sigma = \text{JVMexec-ncollect } P' \ \sigma$ 
    using assms ncollect-JVMexec-ncollect by(simp add: JVMNaiveCollectionSemantics.csmall-def)
  have  $\text{exec-instr}$ :  $\text{exec-step } P \ h \ \text{stk} \ \text{loc} \ C \ M \ \text{pc} \ \text{ics} \ \text{frs}' \ sh$ 
    =  $\text{exec-step } P' \ h \ \text{stk} \ \text{loc} \ C \ M \ \text{pc} \ \text{ics} \ \text{frs}' \ sh$ 
    using ncollect-exec-step[OF step above-C]  $\sigma \ \text{frs}$  False by simp
  show ?thesis
proof(cases  $xp1$ )
  case None then show ?thesis

```

```

using None  $\sigma$  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  $\sigma$  assms
  proof(cases xp)
    case None
    have frames: classes-above-frames P (frames-of  $\sigma$ )  $\cap$  classes-changed P P'
= {}
    using None Some frs  $\sigma$  assms by(auto simp: JVMNaiveCollectionSemantics.csmall-def)
    have frsi: classes-above-frames P frs  $\cap$  classes-changed P P' = {} using  $\sigma$ 
frames by simp
    have xpcollect: classes-above-xcpts P  $\cap$  classes-changed P P' = {}
    using None Some frs  $\sigma$  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  $\sigma' = (xp1, h1, frs1, sh1)$  and frs=frs' and loc=loc and a=a and
sh=sh]
    exec exec-instr Some assms by auto
    show ?thesis using Some exec  $\sigma$  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
qed

```

— if collected classes unchanged, if *cbig* returned a result under *P*, *P'* returns the same

lemma ncollect-JVMbig:

```

assumes collect: ( $\sigma'$ , cset)  $\in$  JVMNaiveCollectionSemantics.cbig P  $\sigma$ 
and intersect: cset  $\cap$  classes-changed P P' = {}
and prealloc: preallocated (fst(snd  $\sigma$ ))
shows ( $\sigma'$ , cset)  $\in$  JVMNaiveCollectionSemantics.cbig P'  $\sigma$ 
using JVMNaiveCollectionSemantics.csmall-to-cbig-prop2[where R= $\lambda P$  P' cset.
cset  $\cap$  classes-changed P P' = {}]
and Q= $\lambda \sigma$ . preallocated (fst(snd  $\sigma$ )) and P=P and P'=P' and  $\sigma$ = $\sigma$  and  $\sigma'$ = $\sigma'$ 
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*:

```

(∃ csetn. (σ', csetn) ∈ jvm-naive-out P t) ⟷ (∃ csets. (σ', csets) ∈ jvm-smart-out
P t)

```

```

by(auto simp: JVMNaiveCollectionSemantics.cbig-big-equiv
  JVMSmartCollectionSemantics.cbig-big-equiv)

```

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

lemma *jvm-smart-collect-start-make-test-prog*:
 $\llbracket P \in \text{jvm-progs}; t \in \text{jvm-tests} \rrbracket$
 $\implies \text{jvm-smart-collect-start } (\text{jvm-make-test-prog } P \ t) = \text{jvm-smart-collect-start } P$
using *jvm-make-test-prog-classes-above-xcpts* **by** *simp*

lemma *jvm-smart-out-classes-above-start-heap*:
assumes $s: (\sigma', cset_s) \in \text{jvm-smart-out } P \ t$ **and** $h: \text{start-heap } (t\#P) \ a = \text{Some}(C, fs)$
and $P: P \in \text{jvm-progs}$ **and** $t: t \in \text{jvm-tests}$
shows $\text{classes-above } (\text{jvm-make-test-prog } P \ t) \ C \subseteq cset_s$
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 $(\sigma', cset_s) \in \text{jvm-smart-out } P \ t$ **and** $\text{start-sheap } C = \text{Some}(sfs, i)$
shows $\text{classes-above } (\text{jvm-make-test-prog } P \ t) \ C \subseteq cset_s$
using *assms start-prog-classes-above-Start* **by**(*clarsimp split: if-split-asm*)

lemma *jvm-smart-out-classes-above-frames*:
 $(\sigma', cset_s) \in \text{jvm-smart-out } P \ t$
 $\implies \text{classes-above-frames } (\text{jvm-make-test-prog } P \ t) \ (\text{frames-of } (\text{start-state } (t\#P)))$
 $\subseteq cset_s$
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 $\exists t. P \vdash C \text{ has } F, \text{Static}:t \text{ in } D$
then *Some D* else *None*) |
coll-init-class P (*Putstatic* $C \ F \ D$) = (if $\exists t. P \vdash C \text{ has } F, \text{Static}:t \text{ in } D$
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) = ($\exists Cfs. h \ a = \text{Some } Cfs$) |
is-ptr h - = *False*

lemma *is-ptrD*: $is_ptr\ h\ v \implies v = Null \vee (\exists a. v = Addr\ a \wedge (\exists Cfs. h\ a = Some\ Cfs))$

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* _{Φ} *P* **and** *correct*: $P, \Phi \vdash (xp, h, (stk, loc, C, M, pc, ics) \# frs, sh) \checkmark$

shows *stack-safe* (*instrs-of P C M* ! *pc*) *h stk*

proof —

from *correct* **obtain** *b Ts T mxs mxl₀ ins xt* **where**
mC: $P \vdash C\ sees\ M, b: Ts \rightarrow T = (mxs, mxl_0, ins, xt)$ **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 mxl₀ ins xt*
($\Phi\ C\ M$)
by(*auto simp: wf-jvm-prog-phi-def wf-mdecl-def*)
with *pc* **have** *wt*: $P, T, mxs, length\ ins, xt \vdash ins\ !\ pc, pc :: \Phi\ C\ M$ **by**(*simp add: wt-method-def*)
from *mC correct* **obtain** *ST LT* **where**
 $\Phi: \Phi\ C\ M\ !\ pc = Some\ (ST, LT)$ **and**
 $stk: P, h \vdash stk\ [: \leq]\ ST$ **by** *fastforce*
show *?thesis*
proof(*cases instrs-of P C M* ! *pc*)
case (*Getfield F D*)
with *mC* $\Phi\ wt\ stk$ **obtain** *oT ST'* **where**
 $oT: P \vdash oT \leq Class\ D$ **and**
 $ST: ST = oT \# ST'$ **by** *fastforce*
with *stk* **obtain** *ref stk'* **where**
 $stk': stk = ref \# stk'$ **and**
 $ref: P, h \vdash ref\ [: \leq]\ oT$ **by** *auto*
with *ref oT* **have** $ref = Null \vee (ref \neq Null \wedge P, h \vdash ref\ [: \leq]\ Class\ D)$ **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* $\Phi\ wt\ stk$ **obtain** *vT oT ST'* **where**
 $oT: P \vdash oT \leq Class\ D$ **and**
 $ST: ST = vT \# oT \# ST'$ **by** *fastforce*
with *stk* **obtain** *v ref stk'* **where**
 $stk': stk = v \# ref \# stk'$ **and**
 $ref: P, h \vdash ref\ [: \leq]\ oT$ **by** *auto*

```

with  $ref\ oT$  have  $ref = Null \vee (ref \neq Null \wedge P, h \vdash ref : \leq Class\ D)$  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  $mC\ \Phi\ wt\ stk$  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  $ref = Null \vee (ref \neq Null \wedge (\exists D'. P, h \vdash ref : \leq Class\ D'))$ 
    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  $mC\ \Phi\ wt\ stk$  have
     $ST$ :  $n < size\ ST$  and
     $oT$ :  $ST!n = NT \vee (\exists D. ST!n = Class\ D)$  by auto
  with  $stk$  have  $stk'$ :  $n < size\ stk$  by (auto simp: list-all2-lengthD)
  with  $stk\ ST\ oT$  list-all2-nthD2
  have  $stk!n = Null \vee (stk!n \neq Null \wedge (\exists D. P, h \vdash stk!n : \leq Class\ D))$  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  $mC\ \Phi\ wt\ stk$  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  $ref = Null \vee (ref \neq Null \wedge (\exists D'. P, h \vdash ref : \leq Class\ D'))$ 
    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 $\subseteq$ 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

$wtp: wf-jvm-prog_{\Phi} P$

and *correct*: $P, \Phi \vdash (xp, h, frs, sh) \checkmark$

— defs

and f' : $hd\ frs = (stk, loc, C', M', pc, ics)$

— backward promises - will be collected prior

and *heap*: $\bigwedge C\ fs. \exists a. h\ a = Some(C, fs) \implies classes\ above\ P\ C \subseteq cset$

and *sheap*: $\bigwedge C\ sfs\ i. sh\ C = Some(sfs, i) \implies classes\ above\ P\ C \subseteq cset$

and *xcpts*: $classes\ above\ xcpts\ P \subseteq cset$

and *frames*: $classes\ above\ frames\ P\ frs \subseteq cset$

— forward promises - will be collected after if not already

and *init-class-prom*: $\bigwedge C. ics = Called\ [] \vee ics = No\ ics$

$\implies coll\ init\ class\ P\ (instrs\ of\ P\ C'\ M'\ !\ pc) = Some\ C \implies classes\ above\ P\ C \subseteq cset$

and *Calling-prom*: $\bigwedge C'\ Cs'. ics = Calling\ C'\ Cs' \implies classes\ above\ P\ C' \subseteq cset$

— collection and step

and *smart*: $JVMexec\ scollect\ P\ (xp, h, frs, sh) \subseteq cset$

and *small*: $(xp', h', frs', sh') \in JVMsmall\ P\ (xp, h, frs, sh)$

shows $(h'\ a = Some(C, fs) \longrightarrow classes\ above\ P\ C \subseteq cset)$

$\wedge (sh'\ C = Some(sfs', i') \longrightarrow classes\ above\ P\ C \subseteq cset)$

$\wedge (classes\ above\ frames\ P\ frs' \subseteq cset)$

using *assms*

proof(*cases* *frs*)

case (*Cons* *f1* *frs1*)

then have cr' : $P, \Phi \vdash (xp, h, (stk, loc, C', M', pc, ics) \# frs1, sh) \checkmark$ **using** *correct* f' **by** *simp*

let $?i = instrs\ of\ P\ C'\ M'\ !\ pc$

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: $\bigwedge Cs\ a. ics = Throwing\ Cs\ a \implies \exists obj. h\ a = Some\ obj$ **by** *simp*

have *confc*: *conf-clinit* $P\ sh\ frs$ **using** *correct* *Cons* **by** *simp*

have *Called-prom*: $\bigwedge C'\ Cs'. ics = Called\ (C' \# Cs')$

$\implies classes\ above\ P\ C' \subseteq cset \wedge classes\ above\ P\ (fst(method\ P\ C'\ clinit))$

$\subseteq cset$

proof —

fix $C'\ Cs'$ **assume** [*simp*]: $ics = Called\ (C' \# Cs')$

then have $C' \in set(clinit\ classes\ frs)$ **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:  $\bigwedge Cs. ics = Called\ Cs \implies \exists C1\ subj. Called\ context\ P\ C1\ ?i$ 
 $\wedge sh\ C1 = Some\ subj$ 
using cr' by(auto simp: conf-f-def2)
have Throwing-prom:  $\bigwedge C'\ Cs\ a. ics = Throwing\ (C'\#Cs)\ a \implies \exists sfs. sh\ C' =$ 
 $Some(sfs, Processing)$ 
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  $\exists sfs. sh\ C' = Some(sfs, Processing)$  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 xp0 h0 frs0 sh0 where
    exec-step: exec-step P h stk loc C' M' pc ics frs1 sh = (xp0, h0, frs0, sh0)
    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 shear frames exec exec-step f' Cons
    si init-class-prom stack-safe Calling-prom Called-prom Called-prom2 Throw-
ing-prom
  proof(induct rule: exec-step-ind.induct)
    case exec-step-ind-Load show ?case using exec-step-ind-Load.prem(1–7)
  by auto
  next
    case exec-step-ind-Store show ?case using exec-step-ind-Store.prem(1–7)
  by auto
  next
    case exec-step-ind-Push show ?case using exec-step-ind-Push.prem(1–7)
  by auto
  next
    case exec-step-ind-NewOOM-Called show ?case using exec-step-ind-NewOOM-Called.prem(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.prem(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.prem1(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.prem1(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.prem1(1-7) by auto
next
case exec-step-ind-Getfield-Null show ?case using exec-step-ind-Getfield-Null.prem1(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.prem1(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.prem1(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.prem1(1-7) by auto
next
  case exec-step-ind-Getstatic-NoField show ?case
    using exec-step-ind-Getstatic-NoField.prem1(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.prem1(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.prem1(1-7) by auto
next
  case exec-step-ind-Getstatic-Done show ?case
    using exec-step-ind-Getstatic-Done.prem1(1-7) by auto
next
  case exec-step-ind-Getstatic-Init show ?case
    using exec-step-ind-Getstatic-Init.prem1(1-7) by auto
next
  case exec-step-ind-Putfield-Null show ?case
    using exec-step-ind-Putfield-Null.prem1(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.prem(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.prem(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 C0 M0 pc ics frs
sh)
  obtain a C1 fs where addr: hd (tl stk) = Null  $\vee$  (hd (tl stk) = Addr a  $\wedge$  h
a = Some(C1,fs))
  using exec-step-ind-Putfield.prem(8,10) by(auto dest!: exec-step-input-StepID
is-ptrD)
  then have  $\bigwedge a. \text{hd}(tl\ stk) = \text{Addr } a \implies \text{classes-above } P\ C1 \subseteq \text{cset}$ 
  using exec-step-ind-Putfield.prem(1) addr by auto
  then show ?case using exec-step-ind-Putfield.hyps exec-step-ind-Putfield.prem(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.prem(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.prem(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 C0 M0
pc Cs frs)
  then have P  $\vdash$  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.prem(8,13)
  by(fastforce dest!: exec-step-input-StepID)
  then show ?case using exec-step-ind-Putstatic-Called.hyps
    exec-step-ind-Putstatic-Called.prem(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.prem(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.prem(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.sims 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.sims 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.sims 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.sims dest!: find-handler-pieces) blast+
next
  case (exec-step-ind-Invoke ps n stk r C h D b Ts T mxs mxl0 ins xt P)
  have classes-above P D  $\subseteq$  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.sims 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.sims dest!: find-handler-pieces) blast+
next
  case (exec-step-ind-Invokestatic-Called ps n stk D b Ts T mxs mxl0 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  $\subseteq$  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 mxl0 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  $\subseteq$  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.prem(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.prem(1-7)
  by auto blast+
next
  case exec-step-ind-Return-Last-Init show ?case
  using exec-step-ind-Return-Last-Init.prem(1-7) by (auto split: if-split-asm)
blast+
next
  case exec-step-ind-Return-Last show ?case
  using exec-step-ind-Return-Last.prem(1-7) by auto
next
  case exec-step-ind-Return-Init show ?case
  using exec-step-ind-Return-Init.prem(1-7) by (auto split: if-split-asm) blast+
next
  case exec-step-ind-Return-NonStatic show ?case
  using exec-step-ind-Return-NonStatic.prem(1-7) by auto
next
  case exec-step-ind-Return-Static show ?case
  using exec-step-ind-Return-Static.prem(1-7) by auto
next
  case exec-step-ind-Pop show ?case using exec-step-ind-Pop.prem(1-7) by
auto
next
  case exec-step-ind-IAdd show ?case using exec-step-ind-IAdd.prem(1-7) by
auto
next
  case exec-step-ind-IfFalse-False show ?case
  using exec-step-ind-IfFalse-False.prem(1-7) by auto
next
  case exec-step-ind-IfFalse-nFalse show ?case
  using exec-step-ind-IfFalse-nFalse.prem(1-7) by auto
next
  case exec-step-ind-CmpEq show ?case using exec-step-ind-CmpEq.prem(1-7)
by auto
next
  case exec-step-ind-Goto show ?case using exec-step-ind-Goto.prem(1-7)
by auto
next
  case exec-step-ind-Throw show ?case using exec-step-ind-Throw.prem(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.prem(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  $\subseteq$  cset using exec-step-ind-Init-None-Called.prem(8,11)

```

```

    by(auto dest!: exec-step-input-StepCD)
  then show ?case using exec-step-ind-Init-None-Called.prem(1-7)
    by(auto split: if-split-asm) blast+
next
  case exec-step-ind-Init-Done show ?case
    using exec-step-ind-Init-Done.prem(1-7) by auto
next
  case exec-step-ind-Init-Processing show ?case
    using exec-step-ind-Init-Processing.prem(1-7) by auto
next
  case exec-step-ind-Init-Error show ?case
    using exec-step-ind-Init-Error.prem(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.prem(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.prem(1-7)
    by(auto split: if-split-asm) blast+
next
  case exec-step-ind-Init show ?case
    using exec-step-ind-Init.prem(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.prem(8,14) by(fastforce dest!: exec-step-input-StepTD)
    then show ?case using exec-step-ind-InitThrow.prem(1-7)
    by(auto split: if-split-asm) blast+
next
  case exec-step-ind-InitThrow-End show ?case using exec-step-ind-InitThrow-End.prem(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*: $(\sigma', cset') \in JVMSmartCollectionSemantics.cbig\ P\ \sigma$

and $nend: \sigma \notin JVMendset$
and $ics: ics\text{-of } (hd(frames\text{-of } \sigma)) = Calling\ Object\ Cs$
shows $classes\text{-above } P\ Object \subseteq cset \cup cset'$
proof($cases\ frames\text{-of } \sigma$)
 case Nil **then show** $?thesis$ **using** $nend$ **by**($clarsimp\ simp: JVMendset\text{-def}$)
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 $JVMSmartCollectionSemantics.cb\big\text{-}stepD[OF\ big\ nend]$ $nend\ ics\ Cons$
 by($clarsimp\ simp: JVMSmartCollectionSemantics.csmall\text{-}def\ JVMendset\text{-def}$)
qed

— IH case where C has not been prepared yet

lemma *Calling-None-next-state:*

assumes $ics: ics\text{-of } (hd(frames\text{-of } \sigma)) = Calling\ C\ Cs$
and $none: sheap\ \sigma\ C = None$
and $set: \forall C'. P \vdash C \preceq^* C' \longrightarrow (\exists sfs\ i. sheap\ \sigma\ C' = Some(sfs, i))$
 $\longrightarrow classes\text{-above } P\ C' \subseteq cset$
and $\sigma': (\sigma', cset') \in JVMSmartCollectionSemantics.csmall\ P\ \sigma$
shows $\sigma' \notin JVMendset \wedge ics\text{-of } (hd(frames\text{-of } \sigma')) = Calling\ C\ Cs$
 $\wedge (\exists sfs. sheap\ \sigma'\ C = Some(sfs, Prepared))$
 $\wedge (\forall C'. P \vdash C \preceq^* C' \longrightarrow C \neq C')$
 $\longrightarrow (\exists sfs\ i. sheap\ \sigma'\ C' = Some(sfs, i)) \longrightarrow classes\text{-above } P\ C' \subseteq cset$
proof($cases\ frames\text{-of } \sigma = [] \vee (\exists x. fst\ \sigma = Some\ x)$)
 case $True$ **then show** $?thesis$ **using** $assms$
 by($cases\ \sigma, auto\ simp: JVMSmartCollectionSemantics.csmall\text{-}def$)
next
 case $False$
 then obtain $f1\ frs1$ **where** $frs: frames\text{-of } \sigma = f1 \# frs1$ **by**($cases\ frames\text{-of } \sigma, auto$)
 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($cases\ \sigma, cases\ method\ P\ C\ clinit$)
 ($clarsimp\ simp: split\text{-}beta\ JVMSmartCollectionSemantics.csmall\text{-}def\ JVMendset\text{-}def$)
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 $ics: ics\text{-of } (hd(frames\text{-of } \sigma)) = Calling\ C\ Cs$
and $prep: sheap\ \sigma\ C = Some(sfs, Prepared)$
and $set: \forall C'. P \vdash C \preceq^* C' \longrightarrow C \neq C' \longrightarrow (\exists sfs\ i. sheap\ \sigma\ C' = Some(sfs, i))$
 $\longrightarrow classes\text{-above } P\ C' \subseteq cset$
and $\sigma': (\sigma', cset') \in JVMSmartCollectionSemantics.csmall\ P\ \sigma$
shows $\sigma' \notin JVMendset \wedge ics\text{-of } (hd(frames\text{-of } \sigma')) = Calling\ D\ (C \# Cs)$

$\wedge (\forall C'. P \vdash D \preceq^* C' \longrightarrow (\exists sfs\ i. \text{sheap } \sigma' C' = \text{Some}(sfs, i))$
 $\longrightarrow \text{classes-above } P\ C' \subseteq \text{cset})$

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

and *step*: $\bigwedge \sigma\ cset'\ Cs. (\sigma', cset') \in \text{JVMSmartCollectionSemantics.cbig } P\ \sigma \implies \sigma \notin \text{JVMendset}$

$\implies \text{ics-of } (\text{hd}(\text{frames-of } \sigma)) = \text{Calling } D\ Cs$
 $\implies \forall C'. P \vdash D \preceq^* C' \longrightarrow (\exists sfs\ i. \text{sheap } \sigma\ C' = \text{Some}(sfs, i))$
 $\longrightarrow \text{classes-above } P\ C' \subseteq \text{cset}$
 $\implies \text{classes-above } P\ D \subseteq \text{cset} \cup \text{cset}'$

and *big*: $(\sigma', cset') \in \text{JVMSmartCollectionSemantics.cbig } P\ \sigma$

and *nend*: $\sigma \notin \text{JVMendset}$

and *curr*: $\text{ics-of } (\text{hd}(\text{frames-of } \sigma)) = \text{Calling } C\ Cs$

and *set*: $\forall C'. P \vdash C \preceq^* C' \longrightarrow (\exists sfs\ i. \text{sheap } \sigma\ C' = \text{Some}(sfs, i))$
 $\longrightarrow \text{classes-above } P\ C' \subseteq \text{cset}$

shows $\text{classes-above } P\ C \subseteq \text{cset} \cup \text{cset}'$

proof(*cases* $\text{frames-of } \sigma$)

case *Nil* **then show** *?thesis* **using** *nend* **by**(*clarsimp simp: JVMendset-def*)

next

```

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

```

```

       $\forall C'. P \vdash D \preceq^* C' \longrightarrow (\exists sfs\ i.\ sheap\ \sigma\ C' = Some(sfs,i))$ 
       $\longrightarrow classes\text{-}above\ P\ C' \subseteq cset$ 
    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*: $\bigwedge y\ z. P \vdash y \prec^1 z \implies$

$P \vdash z \preceq^* Object \implies$

$\forall \sigma\ cset'\ Cs. (\sigma', cset') \in JVMSmartCollectionSemantics.cbig\ P\ \sigma \longrightarrow \sigma \notin JVMendset$

$\longrightarrow ics\text{-}of\ (hd(frames\text{-}of\ \sigma)) = Calling\ z\ Cs$

$\longrightarrow (\forall C'. P \vdash z \preceq^* C' \longrightarrow (\exists sfs\ i.\ sheap\ \sigma\ C' = Some(sfs,i)))$

$\longrightarrow classes\text{-}above\ P\ C' \subseteq cset$

$\longrightarrow classes\text{-}above\ P\ z \subseteq cset \cup cset' \implies$

$\forall \sigma\ cset'\ Cs. (\sigma', cset') \in JVMSmartCollectionSemantics.cbig\ P\ \sigma \longrightarrow \sigma \notin JVMendset$

$\longrightarrow ics\text{-}of\ (hd(frames\text{-}of\ \sigma)) = Calling\ y\ Cs$

$\longrightarrow (\forall C'. P \vdash y \preceq^* C' \longrightarrow (\exists sfs\ i.\ sheap\ \sigma\ C' = Some(sfs,i)))$

$\longrightarrow classes\text{-}above\ P\ C' \subseteq cset$

$\longrightarrow classes\text{-}above\ P\ y \subseteq cset \cup cset'$

using *Calling-collects-IH* **by** *blast*

have *result*: $\forall \sigma\ cset'\ Cs.$

$(\sigma', cset') \in JVMSmartCollectionSemantics.cbig\ P\ \sigma \longrightarrow \sigma \notin JVMendset$

$\longrightarrow ics\text{-}of\ (hd(frames\text{-}of\ \sigma)) = Calling\ C\ Cs$

$\longrightarrow (\forall C'. P \vdash C \preceq^* C' \longrightarrow (\exists sfs\ i.\ sheap\ \sigma\ C' = Some(sfs,i)))$

$\longrightarrow \text{classes-above } P \ C' \subseteq \text{cset}$
 $\longrightarrow \text{classes-above } P \ C \subseteq \text{cset} \cup \text{cset}'$
using *converse-rtrancl-induct*[*OF sub*,
where $P = \lambda C. \forall \sigma \text{ cset}' \text{ Cs. } (\sigma', \text{cset}') \in \text{JVMSmartCollectionSemantics.cbigr}$
 $P \ \sigma \longrightarrow \sigma \notin \text{JVMendset}$
 $\longrightarrow \text{ics-of } (\text{hd}(\text{frames-of } \sigma)) = \text{Calling } C \ \text{Cs}$
 $\longrightarrow (\forall C'. P \vdash C \preceq^* C' \longrightarrow (\exists \text{sfs } i. \text{sheap } \sigma \ C' = \text{Some}(\text{sfs}, i)))$
 $\longrightarrow \text{classes-above } P \ C' \subseteq \text{cset}$
 $\longrightarrow \text{classes-above } P \ C \subseteq \text{cset} \cup \text{cset}'$
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*: $P \vdash C \preceq^* \text{Object}$
and *cbigr*: $(\sigma', \text{cset}') \in \text{JVMSmartCollectionSemantics.cbigr}$ $P \ \sigma$
and *nend*: $\sigma \notin \text{JVMendset}$
and *curr*: $\text{curr-instr } P \ (\text{hd}(\text{frames-of } \sigma)) = \text{New } C$
and *ics*: $\text{ics-of } (\text{hd}(\text{frames-of } \sigma)) = \text{No-ics}$
and *sheap*: $\forall C'. P \vdash C \preceq^* C' \longrightarrow (\exists \text{sfs } i. \text{sheap } \sigma \ C' = \text{Some}(\text{sfs}, i))$
 $\longrightarrow \text{classes-above } P \ C' \subseteq \text{cset}$
and *smart*: $\text{cset}' \subseteq \text{cset}$
shows $\text{classes-above } P \ C \subseteq \text{cset}$
proof(*cases* $(\exists \text{sfs } i. \text{sheap } \sigma \ C = \text{Some}(\text{sfs}, i) \wedge i = \text{Done}))$
case *True* **then show** *?thesis using sheap by auto*
next
case *False*
obtain *n* **where** *nstep*: $(\sigma', \text{cset}') \in \text{JVMSmartCollectionSemantics.csmall-nstep}$
 $P \ \sigma \ n$
and $n \neq 0$ **using** *nend cbigr JVMSmartCollectionSemantics.cbigr-def2*
 $\text{JVMSmartCollectionSemantics.csmall-nstep-base}$ **by** (*metis empty-iff insert-iff*)
then show *?thesis*
proof(*cases n*)
case (*Suc n1*)
then obtain $\sigma 1 \ \text{cset0} \ \text{cset1}$ **where** $\sigma 1: (\sigma 1, \text{cset1}) \in \text{JVMSmartCollectionSemantics.csmall}$ $P \ \sigma$
 $\text{cset}' = \text{cset1} \cup \text{cset0}$ $(\sigma', \text{cset0}) \in \text{JVMSmartCollectionSemantics.csmall-nstep}$
 $P \ \sigma 1 \ n1$
using $\text{JVMSmartCollectionSemantics.csmall-nstep-SucD } nstep$ **by** *blast*
obtain $xp \ h \ \text{frs} \ sh$ **where** $\sigma = (xp, h, \text{frs}, sh)$ **by** (*cases* σ)
then have *ics1*: $\text{ics-of } (\text{hd}(\text{frames-of } \sigma 1)) = \text{Calling } C \ []$
and *sheap'*: $\text{sheap } \sigma = \text{sheap } \sigma 1$ **and** *nend1*: $\sigma 1 \notin \text{JVMendset}$
using *JVM-New-next-step*[*OF - nend curr*] $\sigma 1(1)$ *False ics*
by (*simp add: JVMSmartCollectionSemantics.csmall-def*) +
have $\sigma' \in \text{JVMendset}$ **using** *cbigr JVMSmartCollectionSemantics.cbigr-def2* **by**
blast
then have *cbigr1*: $(\sigma', \text{cset0}) \in \text{JVMSmartCollectionSemantics.cbigr}$ $P \ \sigma 1$

```

    using JVMSmartCollectionSemantics.cbig-def2  $\sigma 1(3)$  by blast
  have sheap1:  $\forall C'. P \vdash C \preceq^* C' \longrightarrow (\exists sfs\ i. sheap\ \sigma 1\ C' = \lfloor (sfs, i) \rfloor)$ 
     $\longrightarrow$  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  $P\ C \subseteq cset$ 
    using Calling-collects[OF sub cbig1 nend1 ics1 sheap1] by simp
  then show ?thesis using  $\sigma 1(2)$  smart by auto
qed(simp)
qed

```

lemma Getstatic-collects:

```

assumes sub:  $P \vdash D \preceq^* Object$ 
and cbig:  $(\sigma', cset') \in JVMSmartCollectionSemantics.cbig\ P\ \sigma$ 
and nend:  $\sigma \notin JVMendset$ 
and curr:  $curr\text{-}instr\ P\ (hd(frames\text{-}of\ \sigma)) = Getstatic\ C\ F\ D$ 
and ics:  $ics\text{-}of\ (hd(frames\text{-}of\ \sigma)) = No\text{-}ics$ 
and fC:  $P \vdash C\ has\ F, Static:t\ in\ D$ 
and sheap:  $\forall C'. P \vdash D \preceq^* C' \longrightarrow (\exists sfs\ i. sheap\ \sigma\ C' = Some(sfs, i))$ 
     $\longrightarrow$  classes-above  $P\ C' \subseteq cset$ 
and smart:  $cset' \subseteq cset$ 
shows classes-above  $P\ D \subseteq cset$ 
proof(cases  $(\exists sfs\ i. sheap\ \sigma\ D = Some(sfs, i) \wedge i = Done)$ 
   $\vee (ics\text{-}of(hd(frames\text{-}of\ \sigma)) = Called\ [])$ )
  case True then show ?thesis
  proof(cases  $\exists sfs\ i. sheap\ \sigma\ D = Some(sfs, i) \wedge i = Done$ )
    case True then show ?thesis using sheap by auto
  next
    case False
    then have  $ics\text{-}of(hd(frames\text{-}of\ \sigma)) = Called\ []$  using True by clarsimp
    then show ?thesis using ics by auto
  qed
next
  case False
  obtain n where nstep:  $(\sigma', cset') \in JVMSmartCollectionSemantics.csmall\text{-}nstep$ 
   $P\ \sigma\ n$ 
  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  $\sigma 1$ :  $(\sigma 1, cset1) \in JVMSmartCollectionSe-$ 
     $mantics.csmall\ P\ \sigma$ 
     $cset' = cset1 \cup cset0$   $(\sigma', cset0) \in JVMSmartCollectionSemantics.csmall\text{-}nstep$ 
     $P\ \sigma 1\ n1$ 
    using JVMSmartCollectionSemantics.csmall-nstep-SucD nstep by blast
    obtain  $xp\ h\ frs\ sh$  where  $\sigma = (xp, h, frs, sh)$  by (cases  $\sigma$ )
    then have curr1:  $ics\text{-}of\ (hd(frames\text{-}of\ \sigma 1)) = Calling\ D\ []$ 
    and sheap':  $sheap\ \sigma = sheap\ \sigma 1$  and nend1:  $\sigma 1 \notin JVMendset$ 
    using JVM-Getstatic-next-step[OF - nend curr fC]  $\sigma 1(1)$  False ics

```

```

    by(simp add: JVMSmartCollectionSemantics.csmall-def)+
    have  $\sigma' \in \text{JVMendset}$  using cbig JVMSmartCollectionSemantics.cbig-def2 by
blast
    then have cbig1:  $(\sigma', \text{cset0}) \in \text{JVMSmartCollectionSemantics.cbig } P \sigma 1$ 
    using JVMSmartCollectionSemantics.cbig-def2  $\sigma 1(3)$  by blast
    have sheap1:  $\forall C'. P \vdash D \preceq^* C' \longrightarrow (\exists \text{sfs } i. \text{sheap } \sigma 1 C' = \lfloor (\text{sfs}, i) \rfloor)$ 
     $\longrightarrow \text{classes-above } P C' \subseteq \text{cset}$  using sheap' sheap by simp
    have  $\text{cset0} \subseteq \text{cset}$  using  $\sigma 1(2)$  smart by blast
    then have  $\text{classes-above } P D \subseteq \text{cset}$ 
    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:  $P \vdash D \preceq^* \text{Object}$ 
and cbig:  $(\sigma', \text{cset}') \in \text{JVMSmartCollectionSemantics.cbig } P \sigma$ 
and nend:  $\sigma \notin \text{JVMendset}$ 
and curr:  $\text{curr-instr } P (\text{hd}(\text{frames-of } \sigma)) = \text{Putstatic } C F D$ 
and ics:  $\text{ics-of } (\text{hd}(\text{frames-of } \sigma)) = \text{No-ics}$ 
and fC:  $P \vdash C \text{ has } F, \text{Static:} t \text{ in } D$ 
and sheap:  $\forall C'. P \vdash D \preceq^* C' \longrightarrow (\exists \text{sfs } i. \text{sheap } \sigma C' = \text{Some}(\text{sfs}, i))$ 
 $\longrightarrow \text{classes-above } P C' \subseteq \text{cset}$ 
and smart:  $\text{cset}' \subseteq \text{cset}$ 
shows  $\text{classes-above } P D \subseteq \text{cset}$ 
proof(cases  $(\exists \text{sfs } i. \text{sheap } \sigma D = \text{Some}(\text{sfs}, i) \wedge i = \text{Done})$ 
 $\vee (\text{ics-of } (\text{hd}(\text{frames-of } \sigma)) = \text{Called } [])$ )
  case True then show ?thesis
  proof(cases  $\exists \text{sfs } i. \text{sheap } \sigma D = \text{Some}(\text{sfs}, i) \wedge i = \text{Done}$ )
    case True then show ?thesis using sheap by auto
  next
    case False
    then have  $\text{ics-of } (\text{hd}(\text{frames-of } \sigma)) = \text{Called } []$  using True by clarsimp
    then show ?thesis using ics by auto
  qed
next
  case False
  obtain n where nstep:  $(\sigma', \text{cset}') \in \text{JVMSmartCollectionSemantics.csmall-nstep}$ 
 $P \sigma n$ 
  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  $\sigma 1 \text{ cset0 cset1}$  where  $\sigma 1$ :  $(\sigma 1, \text{cset1}) \in \text{JVMSmartCollectionSe-}$ 
 $\text{mantics.csmall } P \sigma$ 
     $\text{cset}' = \text{cset1} \cup \text{cset0}$   $(\sigma', \text{cset0}) \in \text{JVMSmartCollectionSemantics.csmall-nstep}$ 
 $P \sigma 1 n1$ 
    using JVMSmartCollectionSemantics.csmall-nstep-SucD nstep by blast
  qed

```

```

obtain  $xp\ h\ frs\ sh$  where  $\sigma = (xp, h, frs, sh)$  by (cases  $\sigma$ )
then have  $curr1: ics\text{-}of\ (hd(frames\text{-}of\ \sigma 1)) = Calling\ D\ []$ 
and  $sheap'$ :  $sheap\ \sigma = sheap\ \sigma 1$  and  $nend1: \sigma 1 \notin JVMendset$ 
using  $JVM\text{-}Putstatic\text{-}next\text{-}step[OF\ \text{-}\ nend\ curr\ fC]\ \sigma 1(1)\ False\ ics$ 
by (simp add:  $JVM\text{SmartCollectionSemantics.csmall}\text{-}def$ ) +
have  $\sigma' \in JVMendset$  using  $cbig\ JVM\text{SmartCollectionSemantics.cbig}\text{-}def2$  by
blast
then have  $cbig1: (\sigma', cset0) \in JVM\text{SmartCollectionSemantics.cbig}\ P\ \sigma 1$ 
using  $JVM\text{SmartCollectionSemantics.cbig}\text{-}def2\ \sigma 1(3)$  by blast
have  $sheap1: \forall C'. P \vdash D \preceq^* C' \longrightarrow (\exists sfs\ i. sheap\ \sigma 1\ C' = [(sfs, i)])$ 
 $\longrightarrow classes\text{-}above\ P\ C' \subseteq cset$  using  $sheap'\ sheap$  by simp
have  $cset0 \subseteq cset$  using  $\sigma 1(2)\ smart$  by blast
then have  $classes\text{-}above\ P\ D \subseteq cset$ 
using  $Calling\text{-}collects[OF\ sub\ cbig1\ nend1\ curr1\ sheap1]$  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  $cbig: (\sigma', cset') \in JVM\text{SmartCollectionSemantics.cbig}\ P\ \sigma$ 
and  $smart: cset' \subseteq cset$ 
and  $nend: \sigma \notin JVMendset$ 
and  $curr: curr\text{-}instr\ P\ (hd(frames\text{-}of\ \sigma)) = Invokestatic\ C\ M\ n$ 
and  $ics: ics\text{-}of\ (hd(frames\text{-}of\ \sigma)) = No\text{-}ics$ 
and  $mC: P \vdash C\ sees\ M, Static: Ts \rightarrow T = m\ in\ D$ 
and  $sheap: \forall C'. P \vdash D \preceq^* C' \longrightarrow (\exists sfs\ i. sheap\ \sigma\ C' = Some(sfs, i))$ 
 $\longrightarrow classes\text{-}above\ P\ C' \subseteq cset$ 
shows  $classes\text{-}above\ P\ D \subseteq cset$ 
proof (cases  $(\exists sfs\ i. sheap\ \sigma\ D = Some(sfs, i) \wedge i = Done)$ 
 $\vee (ics\text{-}of(hd(frames\text{-}of\ \sigma)) = Called\ [])$ )
case True then show  $?thesis$ 
proof (cases  $\exists sfs\ i. sheap\ \sigma\ D = Some(sfs, i) \wedge i = Done$ )
case True then show  $?thesis$  using  $sheap$  by auto
next
case False
then have  $ics\text{-}of(hd(frames\text{-}of\ \sigma)) = Called\ []$  using True by clarsimp
then show  $?thesis$  using  $ics$  by auto
qed
next
case False
obtain  $n$  where  $nstep: (\sigma', cset') \in JVM\text{SmartCollectionSemantics.csmall}\text{-}nstep$ 
 $P\ \sigma\ n$ 
and  $n \neq 0$  using  $nend\ cbig\ JVM\text{SmartCollectionSemantics.cbig}\text{-}def2$ 
 $JVM\text{SmartCollectionSemantics.csmall}\text{-}nstep\text{-}base$  by (metis empty-iff insert-iff)
then show  $?thesis$ 
proof (cases  $n$ )
case (Suc  $n1$ )
then obtain  $\sigma 1\ cset0\ cset1$  where  $\sigma 1: (\sigma 1, cset1) \in JVM\text{SmartCollectionSe-}$ 

```

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

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

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

then have $?P \vdash ?D \preceq^* \text{Object}$ **by** $(\text{rule sees-method-sub-Obj})$
moreover from s **obtain** cset' **where**
 cbig : $(\sigma', \text{cset}') \in \text{JVMSmartCollectionSemantics.cbig } ?P \ ? \sigma$ **and** $\text{cset}' \subseteq ?\text{cset}$ **by** clarsimp
moreover have nend : $? \sigma \notin \text{JVMendset}$ **by** $(\text{rule start-state-nend})$

```

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'  $\subseteq$  ?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

$wtp: wf-jvm-prog_{\Phi}\ P$

and correct: $P, \Phi \vdash (xp, h, frs, sh) \checkmark$

— defs

and $f': hd\ frs = (stk, loc, C', M', pc, ics)$

— backward promises - will be collected prior

and heap: $\bigwedge C\ fs. \exists a. h\ a = Some(C, fs) \implies classes-above\ P\ C \subseteq cset$

and sheap: $\bigwedge C\ sfs\ i. sh\ C = Some(sfs, i) \implies classes-above\ P\ C \subseteq cset$

and xcpts: $classes-above-xcpts\ P \subseteq cset$

and frames: $classes-above-frames\ P\ frs \subseteq cset$

— forward promises - will be collected after if not already

and init-class-prom: $\bigwedge C. ics = Called\ [] \vee ics = No-ics$

$\implies coll-init-class\ P\ (instrs-of\ P\ C'\ M'\ !\ pc) = Some\ C \implies classes-above\ P\ C \subseteq cset$

and Calling-prom: $\bigwedge C'\ Cs'. ics = Calling\ C'\ Cs' \implies classes-above\ P\ C' \subseteq cset$

— collection

and smart: $JVMexec-scollect\ P\ (xp, h, frs, sh) \subseteq cset$

shows $JVMexec-ncollect\ P\ (xp, h, frs, sh) \subseteq cset$

using *assms*

proof(cases *frs*)

case (Cons $f'\ frs'$)

then have [simp]: $classes-above\ P\ C' \subseteq cset$ **using** frames f' **by** simp

let $?i = instrs-of\ P\ C'\ M'\ !\ pc$

have $cr': P, \Phi \vdash (xp, h, (stk, loc, C', M', pc, ics) \# frs', sh) \checkmark$ **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**

$Throwing-ex: \bigwedge Cs\ a. ics = Throwing\ Cs\ a \implies \exists obj. h\ a = Some\ obj$ **by** simp

have confc: $conf-clinit\ P\ sh\ frs$ **using** correct Cons **by** simp

have Called-prom: $\bigwedge C'\ Cs'. ics = Called\ (C' \# Cs')$

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

```

      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  $\subseteq$  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  $\exists t. P \vdash C \text{ has } F, \text{Static}:t \text{ in } D$ )
  case True
    then have above-D: classes-above P D  $\subseteq$  cset using ics init-class-prom
Putstatic by simp
    have sub:  $P \vdash C \preceq^* D$  using has-field-decl-above True by blast
    then have above-C: classes-between P C D - {D}  $\subseteq$  cset
      using True Putstatic above-D smart f' by simp
    then have classes-above P C  $\subseteq$  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  $\subseteq$  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  $\subseteq$  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  $\exists Ts \ T \ m \ D. \ P \vdash \ C \text{ sees } M, \text{Static}: Ts \rightarrow T = m \text{ in } D$ )
    case True
    then have above-D: classes-above P ?D  $\subseteq$  cset using ics init-class-prom
  Invokestatic
    by(simp add: seeing-class-def)
    have sub:  $P \vdash C \preceq^* ?D$  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}  $\subseteq$  cset
      using True Invokestatic above-D smart f' by simp
      then have classes-above P C  $\subseteq$  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  $\subseteq$  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-jvm-prog_Φ 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*: $\bigwedge C fs. \exists a. h a = \text{Some}(C, fs) \implies \text{classes-above } P C \subseteq cset$

and *sheap*: $\bigwedge C sfs i. sh C = \text{Some}(sfs, i) \implies \text{classes-above } P C \subseteq cset$

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*: $\bigwedge C. ics = \text{Called } [] \vee ics = \text{No-ics}$

$\implies \text{coll-init-class } P (\text{instrs-of } P C' M' ! pc) = \text{Some } C \implies \text{classes-above } P C$

$\subseteq cset$

and *Calling-prom*: $\bigwedge C' Cs'. ics = \text{Calling } C' Cs' \implies \text{classes-above } P C' \subseteq cset$
 — collections
and *smart-coll*: $(\sigma', cset_s) \in \text{JVMSmartCollectionSemantics.csmall } P (xp, h, frs, sh)$
and *naive-coll*: $(\sigma', cset_n) \in \text{JVMNaiveCollectionSemantics.csmall } P (xp, h, frs, sh)$
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*:

$\llbracket \text{wf-jvm-prog}_{\Phi} P;$
 $P, \Phi \vdash (xp, h, frs, sh) \checkmark;$
 $hd \text{ frs} = (stk, loc, C', M', pc, ics);$
 $\bigwedge C fs. \exists a. h a = \text{Some}(C, fs) \implies \text{classes-above } P C \subseteq cset;$
 $\bigwedge C sfs i. sh C = \text{Some}(sfs, i) \implies \text{classes-above } P C \subseteq cset;$
 $\text{classes-above-xcpts } P \subseteq cset;$
 $\text{classes-above-frames } P frs \subseteq cset;$
 $\bigwedge C. ics = \text{Called } [] \vee ics = \text{No-ics}$
 $\implies \text{coll-init-class } P (\text{instrs-of } P C' M' ! pc) = \text{Some } C \implies \text{classes-above } P$
 $C \subseteq cset;$
 $\bigwedge C' Cs'. ics = \text{Calling } C' Cs' \implies \text{classes-above } P C' \subseteq cset;$
 $(\sigma', cset_n) \in \text{JVMNaiveCollectionSemantics.csmall-nstep } P (xp, h, frs, sh) n;$
 $(\sigma', cset_s) \in \text{JVMSmartCollectionSemantics.csmall-nstep } P (xp, h, frs, sh) n;$
 $cset_s \subseteq cset;$
 $\sigma' \in \text{JVMendset} \rrbracket$
 $\implies cset_n \subseteq cset$
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** $\sigma 1: (\sigma 1, cset1) \in \text{JVMNaiveCollectionSemantics.csmall } P ?\sigma$
 $cset_n = cset1 \cup cset' (\sigma', cset') \in \text{JVMNaiveCollectionSemantics.csmall-nstep } P \sigma 1 n1$
 using *JVMNaiveCollectionSemantics.csmall-nstep-SucD*[*OF Suc.prem*(10)] **by** *clarsimp+*
 obtain $\sigma 1' \ cset1' \ cset''$ **where** $\sigma 1': (\sigma 1', cset1') \in \text{JVMSmartCollectionSemantics.csmall } P ?\sigma$
 $cset_s = cset1' \cup cset'' (\sigma', cset'') \in \text{JVMSmartCollectionSemantics.csmall-nstep } P \sigma 1' n1$

```

using JVMSmartCollectionSemantics.csmall-nstep-SucD[OF Suc.prem(11)] by
  clarsimp+
have  $\sigma$ -eq:  $\sigma 1 = \sigma 1'$  using  $\sigma 1(1)$   $\sigma 1'(1)$  by(simp add: JVMNaiveCollectionSe-
  mantics.csmall-def
                                JVMSmartCollectionSemantics.csmall-def)
have sub1': cset1'  $\subseteq$  cset and sub'': cset''  $\subseteq$  cset using Suc.prem(12)  $\sigma 1'(2)$ 
by auto
then have sub1: cset1  $\subseteq$  cset
using jvm-naive-to-smart-csmall[where  $h=h$  and  $sh=sh$  and  $\sigma'=\sigma 1$ , OF
  Suc.prem(1-9) - - sub1']
  Suc.prem(11,12)  $\sigma 1(1)$   $\sigma 1'(1)$   $\sigma$ -eq by fastforce
show ?case
proof(cases n1)
  case 0 then show ?thesis using  $\sigma 1(2,3)$  sub1 by auto
next
  case Suc2: (Suc n2)
  then have nend1:  $\sigma 1 \notin \text{JVMendset}$ 
    using JVMNaiveCollectionSemantics.csmall-nstep-Suc-nend  $\sigma 1(3)$  by blast
  obtain xp1 h1 frs1 sh1 where  $\sigma 1$ -case [simp]:  $\sigma 1 = (xp1, h1, frs1, sh1)$  by(cases
 $\sigma 1$ )
  obtain stk1 loc1 C1' M1' pc1 ics1 where f1':  $\text{hd } frs1 = (stk1, loc1, C1', M1', pc1, ics1)$ 
    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:  $(\sigma', cset') \in \text{JVMNaiveCollectionSemantics.cbig } P \sigma 1$ 
     $(\sigma', cset'') \in \text{JVMSmartCollectionSemantics.cbig } P \sigma 1$  using  $\sigma 1(3)$   $\sigma 1'(3)$ 
  Suc.prem(13)  $\sigma$ -eq
    using JVMNaiveCollectionSemantics.cbig-def2
      JVMSmartCollectionSemantics.cbig-def2 by blast+
  obtain  $\sigma 2'$  cset2' cset2'' where  $\sigma 2'$ :  $(\sigma 2', cset2') \in \text{JVMSmartCollectionSe-}$ 
  mantics.csmall  $P \sigma 1$ 
    cset'' = cset2'  $\cup$  cset2''  $(\sigma', cset2'') \in \text{JVMSmartCollectionSemantics.csmall-nstep}$ 
 $P \sigma 2' n2$ 
    using JVMSmartCollectionSemantics.csmall-nstep-SucD  $\sigma 1'(3)$  Suc2  $\sigma$ -eq by
  blast

  have wtp:  $wf\text{-jvm-prog}_{\Phi} P$  by fact
  let ?i1 =  $\text{instrs-of } P \ C1' \ M1' \ ! \ pc1$ 
  let ?ics1 =  $\text{ics-of } (\text{hd } (\text{frames-of } \sigma 1))$ 
  have step:  $P \vdash (xp, h, frs, sh) \text{-jvm} \rightarrow (xp1, h1, frs1, sh1)$ 
  proof -
    have exec  $(P, ?\sigma) = \lfloor \sigma 1' \rfloor$  using JVMsmart-csmallD[OF  $\sigma 1'(1)$ ] by simp
    then have  $P \vdash ?\sigma \text{-jvm} \rightarrow \sigma 1'$  using jvm-one-step1[OF exec-1.exec-1I] by
  simp
  then show ?thesis using Suc.prem(12)  $\sigma$ -eq by fastforce
  qed
  have correct1:  $P, \Phi \vdash (xp1, h1, frs1, sh1) \checkmark$  by(rule BV-correct[OF wtp step
  Suc.prem(2)])

```

```

have vics1:  $P, h1, sh1 \vdash_i (C1', M1', pc1, ics1)$ 
  using correct1 Suc.premis(7) by(auto simp: conf-f-def2)
from correct1 obtain b Ts T mxs mxl0 ins xt ST LT where
  meth1:  $P \vdash C1' \text{ sees } M1', b: Ts \rightarrow T = (mxs, mxl_0, ins, xt) \text{ in } C1'$  and
  pc1:  $pc1 < \text{length ins}$  and
   $\Phi\text{-}pc1: \Phi C1' M1' ! pc1 = \text{Some } (ST, LT)$  by(auto dest: sees-method-fun)
then have wt1:  $P, T, mxs, \text{size ins}, xt \vdash \text{ins} ! pc1, pc1 :: \Phi C1' M1'$ 
  using wt-jvm-prog-impl-wt-instr[OF wtp meth1] by fast

have  $\bigwedge a C fs sfs' i'. (h1 a = \lfloor (C, fs) \rfloor \longrightarrow \text{classes-above } P C \subseteq cset) \wedge$ 
   $(sh1 C = \lfloor (sfs', i') \rfloor \longrightarrow \text{classes-above } P C \subseteq cset) \wedge$ 
   $\text{classes-above-frames } P frs1 \subseteq cset$ 
proof -
  fix a C fs sfs' i'
  show  $(h1 a = \lfloor (C, fs) \rfloor \longrightarrow \text{classes-above } P C \subseteq cset) \wedge$ 
     $(sh1 C = \lfloor (sfs', i') \rfloor \longrightarrow \text{classes-above } P C \subseteq cset) \wedge$ 
     $(\text{classes-above-frames } P frs1 \subseteq cset)$ 
  using Suc.premis(11–12)  $\sigma 1' \sigma\text{-eq}[THEN sym] \text{JVMsmart-csmallD}[OF \sigma 1'(1)]$ 
    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.premis(1–9)] by auto
qed
then have heap1:  $\bigwedge C fs. \exists a. h1 a = \text{Some}(C, fs) \implies \text{classes-above } P C \subseteq cset$ 
  and sheap1:  $\bigwedge C sfs i. sh1 C = \text{Some}(sfs, i) \implies \text{classes-above } P C \subseteq cset$ 
  and frames1:  $\text{classes-above-frames } P frs1 \subseteq cset$  by blast+
have xcpts1:  $\text{classes-above-xcpts } P \subseteq cset$  using Suc.premis(6) by auto
— init-class promise
have sheap2:  $\bigwedge C. \text{coll-init-class } P ?i1 = \text{Some } C$ 
   $\implies \forall C'. P \vdash C \preceq^* C' \longrightarrow (\exists sfs i. \text{sheap } \sigma 1 C' = \lfloor (sfs, i) \rfloor)$ 
   $\longrightarrow \text{classes-above } P C' \subseteq cset$  using sheap1 by auto
have called:  $\bigwedge C. \text{coll-init-class } P ?i1 = \text{Some } C$ 
   $\implies \text{ics-of } (hd (\text{frames-of } \sigma 1)) = \text{Called } [] \implies \text{classes-above } P C \subseteq cset$ 
proof -
  fix C assume cic:  $\text{coll-init-class } P ?i1 = \text{Some } C$  and
    ics:  $\text{ics-of } (hd (\text{frames-of } \sigma 1)) = \text{Called } []$ 
  then obtain sobj where  $sh1 C = \text{Some sobj}$  using vics1 f1'
  by(cases ?i1, auto simp: seeing-class-def split: if-split-asm)
  then show  $\text{classes-above } P C \subseteq cset$  using sheap1 by(cases sobj, simp)
qed
have init-class-prom1:  $\bigwedge C. ics1 = \text{Called } [] \vee ics1 = \text{No-ics}$ 
   $\implies \text{coll-init-class } P ?i1 = \text{Some } C \implies \text{classes-above } P C \subseteq cset$ 
proof -
  fix C assume ics1 =  $\text{Called } [] \vee ics1 = \text{No-ics}$  and cic:  $\text{coll-init-class } P ?i1$ 
  =  $\text{Some } C$ 
  then have ics:  $?ics1 = \text{Called } [] \vee ?ics1 = \text{No-ics}$  using f1' by simp
  then show  $\text{classes-above } P C \subseteq cset$  using cic
  proof(cases ?ics1 =  $\text{Called } []$ )
    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  $\Phi$ -pc1 wt1 meth1 by auto
    then have  $P \vdash C1 \preceq^* \text{Object}$  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 ⊢ 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  $P \vdash C \preceq^* \text{Object}$  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 ⊢ 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  $P \vdash C \preceq^* \text{Object}$  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 ⊢ C1 sees M1, Static : Ts → 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 ⊢ C ≼* 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:  $\bigwedge C' Cs'. ics1 = \text{Calling } C' Cs' \implies \text{classes-above } P C' \subseteq$ 

```

cset

proof –

fix $C' Cs'$ **assume** $ics: ics1 = \text{Calling } C' Cs'$

then have $is\text{-}class\ P\ C'$ **using** $vics1$ **by** $simp$

then have $obj: P \vdash C' \preceq^* \text{Object}$ **using** $wtp\ is\text{-}class\text{-}is\text{-}subcls$

by $(auto\ simp: wf\text{-}jvm\text{-}prog\text{-}\phi\text{-}def)$

have $sheap3: \forall C1. P \vdash C' \preceq^* C1 \longrightarrow (\exists sfs\ i. sheap\ \sigma1\ C1 = [(sfs, i)])$

$\longrightarrow classes\text{-}above\ P\ C1 \subseteq cset$ **using** $sheap1$ **by** $auto$

show $classes\text{-}above\ P\ C' \subseteq cset$

using $\text{Calling}\text{-}collects[OF\ obj\ cbig1(2)\ nend1 - sheap3\ sub'']\ ics\ f1'$ **by** $simp$

qed

have $in\text{-}naive: (\sigma', cset') \in JVMNaiveCollectionSemantics.csmall\text{-}nstep\ P\ (xp1, h1, frs1, sh1)\ n1$

and $in\text{-}smart: (\sigma', cset'') \in JVMSmartCollectionSemantics.csmall\text{-}nstep\ P\ (xp1, h1, frs1, sh1)\ n1$

using $\sigma1(3)\ \sigma1'(3)\ \sigma\text{-}eq[THEN\ sym]$ **by** $simp+$

have $sub2: cset' \subseteq cset$

by $(rule\ Suc.hyps[OF\ wtp\ correct1\ f1'\ heap1\ sheap1\ xcpts1\ frames1\ init\text{-}class\text{-}prom1\ Calling\text{-}prom1\ in\text{-}naive\ in\text{-}smart\ sub''\ Suc.prem(13)])\ simp\text{-}all$

then show $?thesis$ **using** $\sigma1(2)\ \sigma1'(2)\ sub1\ sub2$ **by** $fastforce$

qed

qed

— ...which means over $cbig$

lemma $jvm\text{-}naive\text{-}to\text{-}smart\text{-}cbig$:

assumes

— well-formedness

$wtp: wf\text{-}jvm\text{-}prog_{\Phi}\ P$

and $correct: P, \Phi \vdash (xp, h, frs, sh) \checkmark$

— defs

and $f': hd\ frs = (stk, loc, C', M', pc, ics)$

— backward promises - will be collected/maintained prior

and $heap: \bigwedge C\ fs. \exists a. h\ a = Some(C, fs) \implies classes\text{-}above\ P\ C \subseteq cset$

and $sheap: \bigwedge C\ sfs\ i. sh\ C = Some(sfs, i) \implies classes\text{-}above\ P\ C \subseteq cset$

and $xcpts: classes\text{-}above\text{-}xcpts\ P \subseteq cset$

and $frames: classes\text{-}above\text{-}frames\ P\ frs \subseteq cset$

— forward promises - will be collected after if not already

and $init\text{-}class\text{-}prom: \bigwedge C. ics = Called\ [] \vee ics = No\text{-}ics$

$\implies coll\text{-}init\text{-}class\ P\ (instrs\text{-}of\ P\ C'\ M'\ !\ pc) = Some\ C \implies classes\text{-}above\ P\ C \subseteq cset$

and $\text{Calling}\text{-}prom: \bigwedge C'\ Cs'. ics = \text{Calling } C'\ Cs' \implies classes\text{-}above\ P\ C' \subseteq cset$

— collections

and $n: (\sigma', cset_n) \in JVMNaiveCollectionSemantics.cbig\ P\ (xp, h, frs, sh)$

and $s: (\sigma', cset_s) \in JVMSmartCollectionSemantics.cbig\ P\ (xp, h, frs, sh)$

and $smart: cset_s \subseteq cset$

shows $cset_n \subseteq cset$

proof –

let $?s = (xp, h, frs, sh)$

have $nend: \sigma' \in JVMendset$ **using** n **by** $(simp\ add: JVMNaiveCollectionSeman-$

tics.cbig-def)
obtain n **where** $n': (\sigma', cset_n) \in JVMNaiveCollectionSemantics.csmall-nstep\ P$
 $? \sigma\ n\ \sigma' \in JVMendset$
using *JVMNaiveCollectionSemantics.cbig-def2* n **by** *auto*
obtain s **where** $s': (\sigma', cset_s) \in JVMSmartCollectionSemantics.csmall-nstep\ P$
 $? \sigma\ s\ \sigma' \in JVMendset$
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 $sn: (\sigma', cset_s) \in JVMSmartCollectionSemantics.csmall-nstep\ P\ ? \sigma\ n$
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 $\subseteq$ 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*: $(\sigma', cset_n) \in jvm-naive-out\ P\ t$ **and** *smart*: $(\sigma', cset_s) \in jvm-smart-out\ P\ t$

and $P: P \in jvm-progs$ **and** $t: t \in jvm-tests$

shows $cset_n \subseteq cset_s$

proof –

let $?P = jvm-make-test-prog\ P\ t$

let $? \sigma = start-state\ (t \# P)$

let $?i = instrs-of\ ?P\ Start\ start-m\ !\ 0$ **and** $?ics = No-ics$

obtain $xp\ h\ frs\ sh$ **where**

[*simp*]: $? \sigma = (xp, h, frs, sh)$ **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** $nS: \neg is-class\ (t \# P)\ Start$

by(*simp add: is-class-def class-def Start-def Test-def*)

from P **have** $nT: \neg is-class\ P\ Test$ **by** *simp*

from $P\ t$ **obtain** m **where** $tPm: t \# P \vdash (fst\ t)\ sees\ main, Static : [] \rightarrow Void = m\ in\ (fst\ t)$

by *auto*

have $nclinit: main \neq clinit$ **by**(*simp add: main-def clinit-def*)

have $Objp: \bigwedge b' Ts' T' m' D'.$

$t \# P \vdash Object\ sees\ start-m, b' : Ts' \rightarrow T' = m' in D' \implies b' = Static \wedge Ts' =$

$[] \wedge T' = Void$

proof –

fix $b' Ts' T' m' D'$

assume $mObj: t \# P \vdash Object\ sees\ start-m, b' : Ts' \rightarrow T' = m' in D'$

from P **have** $ot-nsub: \neg P \vdash Object \preceq^* Test$

by(*clarsimp simp: wf-jvm-prog-def wf-jvm-prog-phi-def*)

from *class-add-sees-method-rev*[*OF* - *ot-ns*] *mObj* *t*
have $P \vdash \text{Object sees start-}m, b' : Ts' \rightarrow T' = m' \text{ in } D'$ **by** (*cases* *t*, *auto*)
with $P \text{ jvm-progs-def}$ **show** $b' = \text{Static} \wedge Ts' = [] \wedge T' = \text{Void}$ **by** *blast*
qed
from $P \text{ t obtain } \Phi$ **where** $wtP0: wf\text{-jvm-prog}_\Phi (t\#P)$ **by** (*auto simp: wf-jvm-prog-def*)
let $? \Phi' = \lambda C f. \text{ if } C = \text{Start} \wedge (f = \text{start-}m \vee f = \text{clinit}) \text{ then start-}\varphi_m \text{ else } \Phi$
C f
from $wtP0$ **have** $wtP: wf\text{-jvm-prog}_{? \Phi'} ?P$
proof –
note $wtP0 \ nS \ tPm \ nclinit$
moreover obtain $\bigwedge C. C \neq \text{Start} \implies ? \Phi' \ C = \Phi \ C \ ? \Phi' \ \text{Start} \ \text{start-}m =$
start-}\varphi_m
 $? \Phi' \ \text{Start} \ \text{clinit} = \text{start-}\varphi_m$ **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** $wf: wf\text{-prog wf-md} (t\#P)$
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**
 $?P \vdash \text{Test sees main, Static} : [] \rightarrow \text{Void} = m \text{ in Test}$ **by** *fast*
with *t* **have** *seeing-class (jvm-make-test-prog P t) Test main = [Test]*
by (*cases m, fastforce simp: seeing-class-def*)
with *i* **show** *?thesis* **by** *simp*
qed
— *well-formedness*
note *wtP*
moreover have *correct: ?P, ?\Phi' \vdash (xp, h, frs, sh) \checkmark*
proof –
note $wtP0 \ nS \ tPm \ nclinit$
moreover have $? \Phi' \ \text{Start} \ \text{start-}m = \text{start-}\varphi_m$ **by** *simp*
ultimately have $?P, ? \Phi' \vdash ?\sigma \checkmark$ **by** (*rule BV-correct-initial*)
then show *?thesis* **by** *simp*
qed
— *defs*
moreover have $hd \ frs = ([], [], \text{Start}, \text{start-}m, 0, \text{No-ics})$ **by** *simp*
— *backward promises*
moreover from *jvm-smart-out-classes-above-start-heap*[*OF smart - P t*]
have $\text{heap: } \bigwedge C \ fs. \exists a. h \ a = \text{Some}(C, fs) \implies \text{classes-above } ?P \ C \subseteq cset_s$ **by**
auto
moreover from *jvm-smart-out-classes-above-start-sheap*[*OF smart*]
have $\text{sheap: } \bigwedge C \ sfs \ i. sh \ C = \text{Some}(sfs, i) \implies \text{classes-above } ?P \ C \subseteq cset_s$ **by**
simp
moreover from *jvm-smart-out-classes-above-xcpts*[*OF smart P t*]
have $xcpts: \text{classes-above-xcpts } ?P \subseteq cset_s$ **by** *simp*
moreover from *jvm-smart-out-classes-above-frames*[*OF smart*]
have $\text{frames: classes-above-frames } ?P \ frs \subseteq cset_s$ **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*: $\bigwedge C. ?ics = \text{Called } [] \vee ?ics = \text{No-ics}$
 $\implies \text{coll-init-class } ?P ?i = \text{Some } C \implies \text{classes-above } ?P C \subseteq cset_s$ **by** *simp*
moreover have $\bigwedge C' Cs'. ?ics = \text{Calling } C' Cs' \implies \text{classes-above } ?P C' \subseteq cset_s$
by *simp*
 — collections
moreover from *naive*
have $n: (\sigma', cset_n) \in \text{JVMNaiveCollectionSemantics.cbig } ?P (xp, h, frs, sh)$ **by** *simp*
moreover from *smart* **obtain** $cset_s'$ **where**
 $s: (\sigma', cset_s') \in \text{JVMSmartCollectionSemantics.cbig } ?P (xp, h, frs, sh)$ **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 $\text{smart} \subseteq \text{naive}$

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** $\sigma: \sigma = (xp, h, frs, sh)$ **by**(*cases σ*)
then show *?thesis*
proof(*cases* $\exists x. xp = \text{Some } x \vee frs = []$)
case *False*
then obtain $stk\ loc\ C\ M\ pc\ ics\ frs'$
where $none: xp = \text{None}$ **and** $frs: frs = (stk, loc, C, M, pc, ics) \# frs'$
by(*cases xp, auto, cases frs, auto*)
have *instr-case*: $ics = \text{Called } [] \vee ics = \text{No-ics} \implies ?thesis$
proof —
assume $ics: ics = \text{Called } [] \vee ics = \text{No-ics}$
then show *?thesis* **using** $\sigma\ none\ frs$
proof(*cases curr-instr P (stk, loc, C, M, pc, ics)*) **qed**(*auto split: if-split-asm*)
qed
then show *?thesis* **using** $\sigma\ none\ frs$
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 $(\sigma', cset_n) \in \text{JVMNaiveCollectionSemantics.csmall } P\ \sigma$

and $(\sigma', cset_s) \in \text{JVMSmartCollectionSemantics.csmall } P\ \sigma$

shows $cset_s \subseteq cset_n$
using *jvm-smart-to-naive-exec-collect* *assms*
by (*auto simp: JVMNaiveCollectionSemantics.csmall-def*
JVMSmartCollectionSemantics.csmall-def)

lemma *jvm-smart-to-naive-csmall-nstep*:
 $\llbracket (\sigma', cset_n) \in JVMNaiveCollectionSemantics.csmall-nstep\ P\ \sigma\ n;$
 $(\sigma', cset_s) \in JVMSmartCollectionSemantics.csmall-nstep\ P\ \sigma\ n \rrbracket$
 $\implies cset_s \subseteq cset_n$
proof (*induct n arbitrary: $\sigma\ \sigma'\ cset_n\ cset_s$*)
case (*Suc n'*)
obtain $\sigma 1\ cset1\ cset'$ **where** $\sigma 1$: $(\sigma 1, cset1) \in JVMNaiveCollectionSemantics.csmall\ P\ \sigma$
 $cset_n = cset1 \cup cset'$ $(\sigma', cset') \in JVMNaiveCollectionSemantics.csmall-nstep\ P\ \sigma 1\ n'$
using *JVMNaiveCollectionSemantics.csmall-nstep-SucD [OF Suc.prem1]* **by** *clarsimp+*
obtain $\sigma 1'\ cset1'\ cset''$ **where** $\sigma 1'$: $(\sigma 1', cset1') \in JVMSmartCollectionSemantics.csmall\ P\ \sigma$
 $cset_s = cset1' \cup cset''$ $(\sigma', cset'') \in JVMSmartCollectionSemantics.csmall-nstep\ P\ \sigma 1'\ n'$
using *JVMSmartCollectionSemantics.csmall-nstep-SucD [OF Suc.prem2]* **by** *clarsimp+*
have $\sigma\text{-eq}$: $\sigma 1 = \sigma 1'$ **using** $\sigma 1(1)\ \sigma 1'(1)$ **by** (*simp add: JVMNaiveCollectionSemantics.csmall-def*
JVMSmartCollectionSemantics.csmall-def)
then have $sub1$: $cset1' \subseteq cset1$ **using** $\sigma 1(1)\ \sigma 1'(1)$ *jvm-smart-to-naive-csmall*
by *blast*
have $sub2$: $cset'' \subseteq cset'$ **using** $\sigma 1(3)\ \sigma 1'(3)\ \sigma\text{-eq}\ Suc.hyps$ **by** *blast*
then show $?case$ **using** $\sigma 1(2)\ \sigma 1'(2)\ sub1\ sub2$ **by** *blast*
qed (*simp*)

lemma *jvm-smart-to-naive-cbig*:
assumes n : $(\sigma', cset_n) \in JVMNaiveCollectionSemantics.cbig\ P\ \sigma$
and s : $(\sigma', cset_s) \in JVMSmartCollectionSemantics.cbig\ P\ \sigma$
shows $cset_s \subseteq cset_n$
proof –
obtain n **where** n' : $(\sigma', cset_n) \in JVMNaiveCollectionSemantics.csmall-nstep\ P\ \sigma\ n\ \sigma' \in JVMendset$
using *JVMNaiveCollectionSemantics.cbig-def2 n* **by** *auto*
obtain s **where** s' : $(\sigma', cset_s) \in JVMSmartCollectionSemantics.csmall-nstep\ P\ \sigma\ s\ \sigma' \in JVMendset$
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*: $(\sigma', cset_n) \in \text{jvm-naive-out } P \ t$ **and** *smart*: $(\sigma', cset_s) \in \text{jvm-smart-out } P \ t$
and $P \in \text{jvm-progs}$ **and** $t \in \text{jvm-tests}$
shows $cset_s \subseteq cset_n$
proof –
have *nend*: $\text{start-state } (t\#P) \notin \text{JVMendset}$ **by** (*simp add*: *JVMendset-def start-state-def*)
from *naive* **obtain** *n* **where**
 $nstep: (\sigma', cset_n) \in \text{JVMNaiveCollectionSemantics.csmall-nstep}$
 $(\text{jvm-make-test-prog } P \ t) \ (\text{start-state } (t\#P)) \ n$
by (*auto dest!*: *JVMNaiveCollectionSemantics.cbigD*)
with *nend naive* **obtain** *n'* **where** $n': n = \text{Suc } n'$
by (*cases n*; *simp add*: *JVMNaiveCollectionSemantics.cbig-def*)
from *start-prog-classes-above-Start*
have *classes-above-frames* $(\text{jvm-make-test-prog } P \ t) \ (\text{frames-of } (\text{start-state } (t\#P)))$
 $= \{\text{Object}, \text{Start}\}$
by (*simp add*: *start-state-def*)
with *nstep n'*
have *jvm-smart-collect-start* $(\text{jvm-make-test-prog } P \ t) \subseteq cset_n$
by (*auto simp*: *start-state-def JVMNaiveCollectionSemantics.csmall-def*
 $\text{dest!}: \text{JVMNaiveCollectionSemantics.csmall-nstep-SucD}$
 $\text{simp del}: \text{JVMNaiveCollectionSemantics.csmall-nstep-Rec}$)
with *jvm-smart-to-naive-cbig* [**where** $P = \text{jvm-make-test-prog } P \ t$ **and** $\sigma = \text{start-state } (t\#P)$ **and** $\sigma' = \sigma$]
 $\text{jvm-smart-collect-start-make-test-prog assms}$ **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*: $(\sigma', cset_n) \in \text{jvm-naive-out } P \ t$ **and** *smart*: $(\sigma', cset_s) \in \text{jvm-smart-out } P \ t$

and $P \in \text{jvm-progs}$ **and** $t \in \text{jvm-tests}$

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*: $P \in \text{jvm-progs}$ **and** *P'*: $P' \in \text{jvm-progs}$ **and** *t*: $t \in \text{jvm-tests}$

and *out*: $o1 \in \text{jvm-smart-out } P \ t$ **and** *dss*: $\text{jvm-deselect } P \ o1 \ P'$

shows $\exists o2 \in \text{jvm-smart-out } P' \ t. \ o1 = o2$

proof –

obtain $\sigma' \ cset_s$ **where** $o1[\text{simp}]: o1 = (\sigma', cset_s)$ **by** (*cases o1*)

with *jvm-naive-iff-smart out* **obtain** $cset_n$ **where** $n: (\sigma', cset_n) \in \text{jvm-naive-out } P \ t$ **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'*: $(\sigma', cset_n) \in \text{jvm-naive-out } P' \text{ } t$ **by** *simp*
with *jvm-naive-iff-smart* **obtain** $cset_s'$ **where** *s'*: $(\sigma', cset_s') \in \text{jvm-smart-out } P' \text{ } t$ **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.