

Regression Test Selection over JVM

Susannah Mansky

February 6, 2026

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	4
3.1	Extending <i>small</i> to multiple steps	5
3.2	Extending <i>small</i> to a big-step semantics	5
4	Collection Semantics	6
4.1	Small-Step Collection Semantics	6
4.2	Extending <i>csmall</i> to multiple steps	7
4.3	Extending <i>csmall</i> to a big-step semantics	8
5	Collection-based RTS	9
6	Instantiating <i>Semantics</i> with Jinja JVM	10
7	<i>classes-changed</i> theory	11
8	<i>subcls</i> theory	12

9	<i>classes-above</i> theory	13
9.1	Methods	14
9.2	Fields	14
9.3	Other	16
10	Instantiating <i>CollectionSemantics</i> with Jinja JVM	16
10.1	JVM-specific <i>classes-above</i> theory	16
10.2	Naive RTS algorithm	17
10.3	Smarter RTS algorithm	18
10.4	A few lemmas using the instantiations	19
11	Inductive JVM execution	20
11.1	Equivalence of <i>exec-step</i> and <i>exec-step-input</i>	28
12	Instantiating <i>CollectionBasedRTS</i> with Jinja JVM	28
12.1	Some <i>classes-above</i> lemmas	28
12.2	JVM next-step lemmas for initialization calling	29
12.3	Definitions	30
12.4	Definition lemmas	30
12.5	Naive RTS algorithm	31
	12.5.1 Definitions	31
	12.5.2 Naive algorithm correctness	31
12.6	Smarter RTS algorithm	33
	12.6.1 Definitions and helper lemmas	33
	12.6.2 Additional well-formedness conditions	34
	12.6.3 Proving naive \subseteq smart	34
	12.6.4 Proving smart \subseteq naive	40
	12.6.5 Safety of the smart algorithm	40

1 Theory Dependencies

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

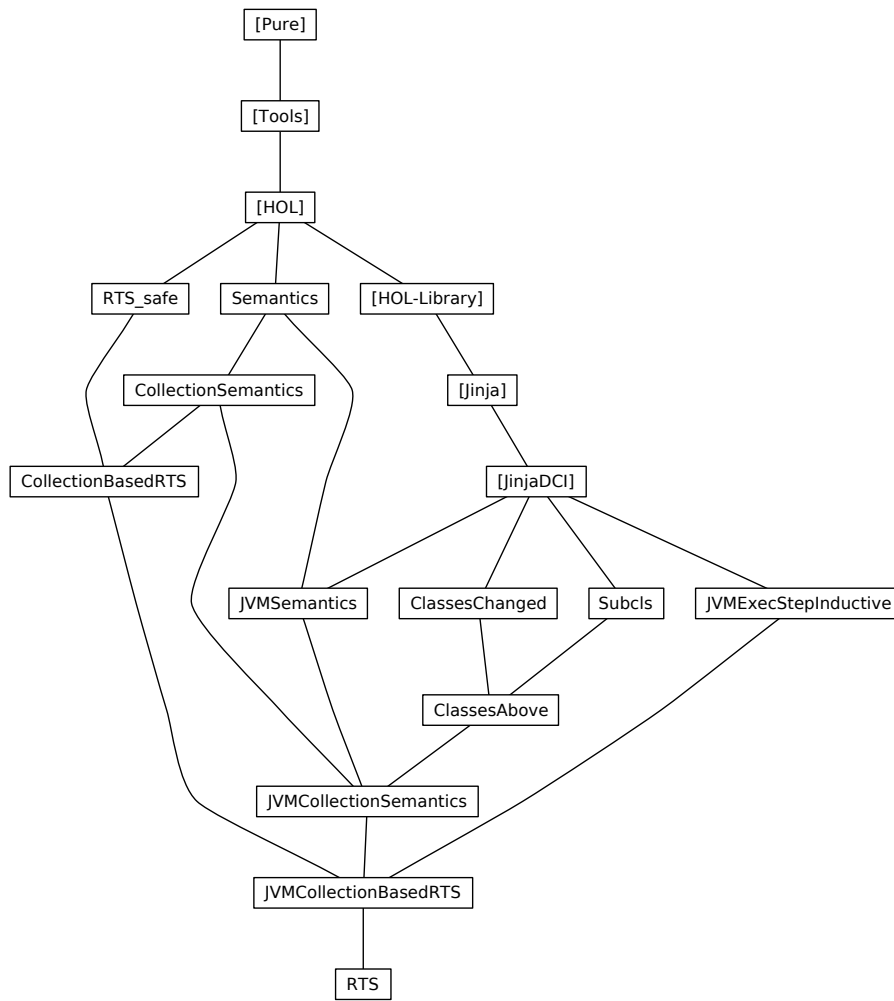


Figure 1: Theory Dependency Graph

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
<proof>
```

```
lemma equiv-out-trans:  $\llbracket \text{equiv-out } a \ b; \text{equiv-out } b \ c \rrbracket \implies \text{equiv-out } a \ c$ 
<proof>
```

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  $\exists o_n \in \text{out } (\text{last } P_s) \ t. \text{equiv-out } o_0 \ o_n$ 
```

```
<proof>
```

```
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>

lemma *small-nstep-SucD*:
assumes $\sigma' \in \text{small-nstep } P \sigma$ (Suc *n*)
shows $\exists \sigma 1. \sigma 1 \in \text{small } P \sigma \wedge \sigma' \in \text{small-nstep } P \sigma 1 n$
 <proof>

lemma *small-nstep-Suc-nend*: $\sigma' \in \text{small-nstep } P \sigma$ (Suc *n*1) $\implies \sigma \notin \text{endset}$
 <proof>

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*:
 $\llbracket \sigma' \in \text{small-nstep } P \sigma n; \sigma' \in \text{endset} \rrbracket \implies \sigma' \in \text{big } P \sigma$
 <proof>

lemma *bigD*:
 $\llbracket \sigma' \in \text{big } P \sigma \rrbracket \implies \exists n. \sigma' \in \text{small-nstep } P \sigma n \wedge \sigma' \in \text{endset}$
 <proof>

lemma *big-def2*:

$\sigma' \in \text{big } P \sigma \iff (\exists n. \sigma' \in \text{small-nstep } P \sigma n \wedge \sigma' \in \text{endset})$
 <proof>

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>

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>

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 :: $'prog \Rightarrow 'state \Rightarrow 'state \text{ set}$ **and**
endset :: $'state \text{ set}$

fixes

collect :: $'prog \Rightarrow 'state \Rightarrow 'state \Rightarrow 'coll$ **and**
combine :: $'coll \Rightarrow 'coll \Rightarrow 'coll$ **and**
collect-id :: $'coll$

assumes

combine-assoc: $\text{combine } (\text{combine } c1 \ c2) \ c3 = \text{combine } c1 \ (\text{combine } c2 \ c3)$ **and**
collect-idl[simp]: $\text{combine } \text{collect-id } c = c$ **and**
collect-idr[simp]: $\text{combine } c \ \text{collect-id} = c$

context *CollectionSemantics* **begin**

4.1 Small-Step Collection Semantics

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

lemma *small-det-csmall-det*:

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

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

<proof>

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* σ 0 = $\{(\sigma, \text{collect-id})\}$ |

csmall-nstep-Rec:

csmall-nstep *P* σ (*Suc* *n*) =

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

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

small-nstep *P* σ *n*

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

<proof>

lemma *csmall-nstep-exists*:

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

<proof>

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

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

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

<proof>

lemma *csmall-nstep-Rec2*:

csmall-nstep *P* σ (*Suc* *n*) =

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

$\}$

<proof>

lemma *csmall-nstep-SucD*:

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

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

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

<proof>

lemma *csmall-nstep-Suc-nend*: $o' \in \text{csmall-nstep } P \sigma (\text{Suc } n 1) \implies \sigma \notin \text{endset}$

<proof>

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>

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'*: $\bigwedge P \sigma \sigma1 \sigma' \text{ coll1 coll2}.$

$R' P \sigma \sigma1 \text{ coll1} \wedge R' P \sigma1 \sigma' \text{ coll2} \implies R' P \sigma \sigma' (\text{combine}$

$\text{coll1 coll2})$

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

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

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

$\langle \text{proof} \rangle$

4.3 Extending *csmall* to a big-step semantics

definition *cbig* :: $'\text{prog} \Rightarrow '\text{state} \Rightarrow ('state \times 'coll) \text{ set}$ **where**

$\text{cbig } P \sigma \equiv$

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

lemma *cbigD*:

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

$\langle \text{proof} \rangle$

lemma *cbigD'*:

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

$\langle \text{proof} \rangle$

lemma *cbig-def2*:

$(\sigma', \text{coll}) \in \text{cbig } P \sigma \iff (\exists n. (\sigma', \text{coll}) \in \text{csmall-nstep } P \sigma n \wedge \sigma' \in \text{endset})$

$\langle \text{proof} \rangle$

lemma *cbig-big-equiv*:

$(\exists \text{coll}. (\sigma', \text{coll}) \in \text{cbig } P \sigma) \iff \sigma' \in \text{big } P \sigma$

$\langle \text{proof} \rangle$

lemma *cbig-big-implies*:

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

$\langle \text{proof} \rangle$

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 $\bigwedge P \sigma \sigma1 \sigma' coll1 coll2. R' P \sigma \sigma1 coll1 \wedge R' P \sigma1 \sigma' coll2 \implies R' P \sigma \sigma' (combine coll1 coll2)$
and $\bigwedge \sigma. R' P \sigma \sigma collect-id$
shows $(\sigma', coll) \in cbig P \sigma \implies R P coll \implies Q P \sigma \implies R' P \sigma \sigma' coll$
 $\langle proof \rangle$

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 $\bigwedge P P' coll1 coll2. R P P' (combine coll1 coll2) = (R P P' coll1 \wedge R P P' coll2)$
and $Qpres: \bigwedge P \sigma \sigma'. Q \sigma \implies \sigma' \in small P \sigma \implies Q \sigma'$
shows $(\sigma', coll) \in cbig P \sigma \implies R P P' coll \implies Q \sigma \implies (\sigma', coll) \in cbig P' \sigma$
 $\langle proof \rangle$

lemma *cbig-stepD*:
assumes *cbig*: $(\sigma', coll') \in cbig P \sigma$ **and** *nend*: $\sigma \notin endset$
shows $\exists \sigma1 coll1. (\sigma1, coll1) \in csmall P \sigma$
 $\wedge (\exists coll. coll' = combine coll1 coll \wedge (\sigma', coll) \in cbig P \sigma1)$
 $\langle proof \rangle$

lemma *csmall-nstep-det-last-eq*:
assumes *det*: $\forall \sigma. small P \sigma = \{\} \vee (\exists \sigma'. small P \sigma = \{\sigma'\})$
shows $\llbracket (\sigma', coll') \in cbig P \sigma; (\sigma', coll') \in csmall-nstep P \sigma n; (\sigma', coll'') \in csmall-nstep P \sigma n' \rrbracket$
 $\implies n = n'$
 $\langle proof \rangle$

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  $\sigma$  = {  $\sigma'$ . exec (P,  $\sigma$ ) = Some  $\sigma'$  }

lemma JVMsmall-prealloc-pres:
assumes pre: preallocated (fst(snd  $\sigma$ ))
  and  $\sigma' \in \text{JVMsmall } P \ \sigma$ 
shows preallocated (fst(snd  $\sigma'$ ))
   $\langle \text{proof} \rangle$ 

lemma JVMsmall-det: JVMsmall P  $\sigma$  = {}  $\vee$  ( $\exists \sigma'. \text{JVMsmall } P \ \sigma = \{\sigma'\}$ )
   $\langle \text{proof} \rangle$ 

definition JVMendset :: jvm-state set where
  JVMendset  $\equiv$  { (xp, h, frs, sh). frs = []  $\vee$  ( $\exists x. \text{xp} = \text{Some } x$ ) }

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

lemma start-state-nend:
  start-state P  $\notin \text{JVMendset}$ 
   $\langle \text{proof} \rangle$ 

interpretation JVMSemantics: Semantics JVMsmall JVMendset

```

⟨proof⟩

end

7 *classes-changed* theory

theory *ClassesChanged*
imports *JinjaDCI.Decl*
begin

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

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

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

lemma *classes-changed-class-changed*[simp]: cn ∈ *classes-changed* P1 P2 = *class-changed* P1 P2 cn
⟨proof⟩

lemma *classes-changed-self*[simp]: *classes-changed* P P = {}
⟨proof⟩

lemma *classes-changed-sym*: *classes-changed* P P' = *classes-changed* P' P
⟨proof⟩

lemma *classes-changed-class*: [cn ∉ *classes-changed* P P'] ⇒ class P cn = class P' cn
⟨proof⟩

lemma *classes-changed-class-set*: [S ∩ *classes-changed* P P' = {}]
⇒ ∀ C ∈ S. class P C = class P' C
⟨proof⟩

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

lemma *classes-changed-cons-eq*:
classes-changed (t # P) P' = (*classes-changed* P P' - {fst t})
∪ (if *class-changed* [t] P' (fst t) then {fst t} else {})
⟨proof⟩

lemma *class-changed-cons*:
fst t ∉ *classes-changed* (t#P) (t#P')
⟨proof⟩

lemma *classes-changed-cons*:

classes-changed (t # P) (t # P') = *classes-changed* P P' - {fst t}
 <proof>

lemma *classes-changed-int-Cons*:

assumes coll \cap *classes-changed* P P' = {}

shows coll \cap *classes-changed* (t # P) (t # P') = {}

<proof>

end

8 subcls theory

theory *Subcls*

imports *JinjaDCI.TypeRel*

begin

lemma *subcls-class-ex*: $\llbracket P \vdash C \preceq^* C'; C \neq C' \rrbracket$

$\implies \exists D' fs ms. \text{class } P C = \llbracket (D', fs, ms) \rrbracket$

<proof>

lemma *class-subcls1*:

$\llbracket \text{class } P y = \text{class } P' y; P \vdash y \prec^1 z \rrbracket \implies P' \vdash y \prec^1 z$

<proof>

lemma *subcls1-single-valued*: *single-valued* (*subcls1* P)

<proof>

lemma *subcls-confluent*:

$\llbracket P \vdash C \preceq^* C'; P \vdash C \preceq^* C'' \rrbracket \implies P \vdash C' \preceq^* C'' \vee P \vdash C'' \preceq^* C'$

<proof>

lemma *subcls1-confluent*: $\llbracket P \vdash a \prec^1 b; P \vdash a \preceq^* c; a \neq c \rrbracket \implies P \vdash b \preceq^* c$

<proof>

lemma *subcls-self-superclass*: $\llbracket P \vdash C \prec^1 C; P \vdash C \preceq^* D \rrbracket \implies D = C$

<proof>

lemma *subcls-of-Obj-acyclic*:

$\llbracket P \vdash C \preceq^* \text{Object}; C \neq D \rrbracket \implies \neg(P \vdash C \preceq^* D \wedge P \vdash D \preceq^* C)$

<proof>

lemma *subcls-of-Obj*: $\llbracket P \vdash C \preceq^* \text{Object}; P \vdash C \preceq^* D \rrbracket \implies P \vdash D \preceq^* \text{Object}$

<proof>

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 \Rightarrow cname \Rightarrow cname set **where**
classes-above P c \equiv { cn. P \vdash c \preceq^* cn }

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

abbreviation *classes-above-xcpts* :: 'm prog \Rightarrow cname set **where**
classes-above-xcpts P \equiv $\bigcup_{x \in \text{sys-xcpts.}} \text{classes-above } P \ x$

lemma *classes-above-def2*:

P \vdash C \prec^1 D \implies *classes-above* P C = {C} \cup *classes-above* P D
 \langle proof \rangle

lemma *classes-above-class*:

\llbracket *classes-above* P C \cap *classes-changed* P P' = {}; P \vdash C \preceq^* C' \rrbracket
 \implies *class* P C' = *class* P' C'
 \langle proof \rangle

lemma *classes-above-subset*:

assumes *classes-above* P C \cap *classes-changed* P P' = {}
shows *classes-above* P C \subseteq *classes-above* P' C
 \langle proof \rangle

lemma *classes-above-subcls*:

\llbracket *classes-above* P C \cap *classes-changed* P P' = {}; P \vdash C \preceq^* C' \rrbracket
 \implies P' \vdash C \preceq^* C'
 \langle proof \rangle

lemma *classes-above-subset2*:

assumes *classes-above* P C \cap *classes-changed* P P' = {}
shows *classes-above* P' C \subseteq *classes-above* P C
 \langle proof \rangle

lemma *classes-above-subcls2*:

\llbracket *classes-above* P C \cap *classes-changed* P P' = {}; P' \vdash C \preceq^* C' \rrbracket
 \implies P \vdash C \preceq^* C'

<proof>

lemma *classes-above-set:*

$\llbracket \text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\} \rrbracket$
 $\implies \text{classes-above } P \ C = \text{classes-above } P' \ C$

<proof>

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>

lemma *classes-above-sub-classes-between-eq:*

$P \vdash C \preceq^* D \implies \text{classes-above } P \ C = (\text{classes-between } P \ C \ D - \{D\}) \cup$
classes-above $P \ D$

<proof>

lemma *classes-above-subcls-subset:*

$\llbracket P \vdash C \preceq^* C' \rrbracket \implies \text{classes-above } P \ C' \subseteq \text{classes-above } P \ C$

<proof>

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>

lemma *classes-above-sees-method:*

$\llbracket \text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\};$

$P \vdash C$ *sees* $M, b: Ts \rightarrow T = m$ *in* C'

$\implies P' \vdash C$ *sees* $M, b: Ts \rightarrow T = m$ *in* C'

<proof>

lemma *classes-above-sees-method2:*

$\llbracket \text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\};$

$P' \vdash C$ *sees* $M, b: Ts \rightarrow T = m$ *in* C'

$\implies P \vdash C$ *sees* $M, b: Ts \rightarrow T = m$ *in* C'

<proof>

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>

9.2 Fields

lemma *classes-above-has-fields:*

assumes *int*: $classes\text{-}above\ P\ C \cap classes\text{-}changed\ P\ P' = \{\}$ **and** *fs*: $P \vdash C$
has-fields FDTs
shows $P' \vdash C$ *has-fields FDTs*
 $\langle proof \rangle$

lemma *classes-above-has-fields-dne*:
assumes $classes\text{-}above\ P\ C \cap classes\text{-}changed\ P\ P' = \{\}$
shows $(\forall FDTs. \neg P \vdash C \text{ has-fields FDTs}) = (\forall FDTs. \neg P' \vdash C \text{ has-fields FDTs})$
 $\langle proof \rangle$

lemma *classes-above-has-field*:
 $\llbracket classes\text{-}above\ P\ C \cap classes\text{-}changed\ P\ P' = \{\};$
 $P \vdash C \text{ has } F, b:t \text{ in } C' \rrbracket$
 $\implies P' \vdash C \text{ has } F, b:t \text{ in } C'$
 $\langle proof \rangle$

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

lemma *classes-above-sees-field*:
 $\llbracket classes\text{-}above\ P\ C \cap classes\text{-}changed\ P\ P' = \{\};$
 $P \vdash C \text{ sees } F, b:t \text{ in } C' \rrbracket$
 $\implies P' \vdash C \text{ sees } F, b:t \text{ in } C'$
 $\langle proof \rangle$

lemma *classes-above-sees-field2*:
 $\llbracket classes\text{-}above\ P\ C \cap classes\text{-}changed\ P\ P' = \{\};$
 $P' \vdash C \text{ sees } F, b:t \text{ in } C' \rrbracket$
 $\implies P \vdash C \text{ sees } F, b:t \text{ in } C'$
 $\langle proof \rangle$

lemma *classes-above-field*:
assumes $classes\text{-}above\ P\ C \cap classes\text{-}changed\ P\ P' = \{\}$
shows $field\ P\ C\ F = field\ P'\ C\ F$
 $\langle proof \rangle$

lemma *classes-above-fields*:
assumes $classes\text{-}above\ P\ C \cap classes\text{-}changed\ P\ P' = \{\}$
shows $fields\ P\ C = fields\ P'\ C$
 $\langle proof \rangle$

lemma *classes-above-ifields*:
 $\llbracket classes\text{-}above\ P\ C \cap classes\text{-}changed\ P\ P' = \{\} \rrbracket$
 \implies
 $ifields\ P\ C = ifields\ P'\ C$
 $\langle proof \rangle$

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

lemma *classes-above-isfields*:
 $\llbracket \text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\} \rrbracket$
 \implies
 $\text{isfields } P \ C = \text{isfields } P' \ C$
 $\langle \text{proof} \rangle$

lemma *classes-above-sblank*:
 $\llbracket \text{classes-above } P \ C \cap \text{classes-changed } P \ P' = \{\} \rrbracket$
 \implies
 $\text{sblank } P \ C = \text{sblank } P' \ C$
 $\langle \text{proof} \rangle$

9.3 Other

lemma *classes-above-start-heap*:
assumes *classes-above-xcpts* $P \cap \text{classes-changed } P \ P' = \{\}$
shows $\text{start-heap } P = \text{start-heap } P'$
 $\langle \text{proof} \rangle$

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**
 $\text{JVMcombine } C \ C' \equiv C \cup C'$

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

10.1 JVM-specific *classes-above* theory

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

lemma *classes-above-start-state*:

assumes *above-xcpts*: *classes-above-xcpts* $P \cap$ *classes-changed* $P P' = \{\}$

shows *start-state* $P =$ *start-state* P'

<proof>

lemma *classes-above-matches-ex-entry*:

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

\implies *matches-ex-entry* $P C pc xcp =$ *matches-ex-entry* $P' C pc xcp$

<proof>

lemma *classes-above-match-ex-table*:

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

shows *match-ex-table* $P C pc es =$ *match-ex-table* $P' C pc es$

<proof>

lemma *classes-above-find-handler*:

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

shows *classes-above-frames* $P frs \cap$ *classes-changed* $P P' = \{\}$

\implies *find-handler* $P a h frs sh =$ *find-handler* $P' a h frs sh$

<proof>

lemma *find-handler-classes-above-frames*:

find-handler $P a h frs sh = (xp',h',frs',sh')$

\implies *classes-above-frames* $P frs' \subseteq$ *classes-above-frames* $P frs$

<proof>

lemma *find-handler-pieces*:

find-handler $P a h frs sh = (xp',h',frs',sh')$

$\implies h = h' \wedge sh = sh' \wedge$ *classes-above-frames* $P frs' \subseteq$ *classes-above-frames* $P frs$

<proof>

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* $P C \mid$

JVMinstr-ncollect $P (Getfield F C) h stk =$

(*if* (*hd* stk) = *Null* *then* $\{\}$)

else *classes-above* $P (cname-of h (the-Addr (hd stk))) \mid$

JVMinstr-ncollect $P (Getstatic C F D) h stk =$ *classes-above* $P C \mid$

JVMinstr-ncollect $P (Putfield F C) h stk =$

(*if* (*hd* (*tl* stk)) = *Null* *then* $\{\}$)

else *classes-above* $P (cname-of h (the-Addr (hd (tl stk)))) \mid$

JVMinstr-ncollect $P (Putstatic C F D) h stk =$ *classes-above* $P C \mid$

JVMinstr-ncollect $P (Checkcast C) h stk =$

(*if* (*hd* stk) = *Null* *then* $\{\}$)

else *classes-above* $P (cname-of h (the-Addr (hd stk))) \mid$

JVMinstr-ncollect $P (Invoke M n) h stk =$

(*if* ($stk ! n$) = *Null* *then* $\{\}$)

```

    else classes-above P (cname-of h (the-Addr (stk ! n))) |
  JVMinstr-ncollect P (Invokestatic C M n) h stk = classes-above P C |
  JVMinstr-ncollect P Throw h stk =
    (if (hd stk) = Null then {}
     else classes-above P (cname-of h (the-Addr (hd stk)))) |
  JVMinstr-ncollect P - h stk = {}

```

```

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

```

— naive collection function

```

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

```

```

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

```

interpretation JVMNaiveCollectionSemantics:

```

  CollectionSemantics JVMsmall JVMendset JVMNaiveCollect JVMcombine JVM-
  collect-id
  ⟨proof⟩

```

10.3 Smarter RTS algorithm

```

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

```

$JVM_{instr}\text{-scollect } P - = \{\}$

fun $JVM_{step}\text{-scollect} ::$

$[jvm\text{-prog}, instr, init\text{-call}\text{-status}] \Rightarrow cname\ set$ **where**

$JVM_{step}\text{-scollect } P i (Calling\ C' Cs) = \{C'\} \mid$

$JVM_{step}\text{-scollect } P i (Called\ (C'\#Cs)) = \{\} \mid$

$JVM_{step}\text{-scollect } P i (Throwing\ Cs a) = \{\} \mid$

$JVM_{step}\text{-scollect } P i ics = JVM_{instr}\text{-scollect } P i$

— smarter collection function

fun $JVM_{exec}\text{-scollect} :: jvm\text{-prog} \Rightarrow jvm\text{-state} \Rightarrow cname\ set$ **where**

$JVM_{exec}\text{-scollect } P (None, h, (stk, loc, C, M, pc, ics)\#frs, sh) =$

$JVM_{step}\text{-scollect } P (instrs\text{-of } P\ C\ M\ !\ pc)\ ics$

$\mid JVM_{exec}\text{-scollect } P - = \{\}$

fun $JVM_{smart}\text{Collect} :: jvm\text{-prog} \Rightarrow jvm\text{-state} \Rightarrow jvm\text{-state} \Rightarrow cname\ set$ **where**

$JVM_{smart}\text{Collect } P\ \sigma\ \sigma' = JVM_{exec}\text{-scollect } P\ \sigma$

interpretation $JVM_{smart}\text{CollectionSemantics}$:

$CollectionSemantics\ JVM_{small}\ JVM_{endset}\ JVM_{smart}\text{Collect}\ JVM_{combine}\ JVM_{collect}\text{-id}$

$\langle proof \rangle$

10.4 A few lemmas using the instantiations

lemma $JVM_{naive}\text{-csmallD}$:

$(\sigma', cset) \in JVM_{Naive}\text{CollectionSemantics.csmall } P\ \sigma$

$\implies JVM_{exec}\text{-ncollect } P\ \sigma = cset \wedge \sigma' \in JVM_{small}\ P\ \sigma$

$\langle proof \rangle$

lemma $JVM_{smart}\text{-csmallD}$:

$(\sigma', cset) \in JVM_{Smart}\text{CollectionSemantics.csmall } P\ \sigma$

$\implies JVM_{exec}\text{-scollect } P\ \sigma = cset \wedge \sigma' \in JVM_{small}\ P\ \sigma$

$\langle proof \rangle$

lemma $jvm\text{-naive}\text{-to}\text{-smart}\text{-csmall}\text{-nstep}\text{-last}\text{-eq}$:

$\llbracket (\sigma', cset_n) \in JVM_{Naive}\text{CollectionSemantics.cbig } P\ \sigma;$

$(\sigma', cset_n) \in JVM_{Naive}\text{CollectionSemantics.csmall}\text{-nstep } P\ \sigma\ n;$

$(\sigma', cset_s) \in JVM_{Smart}\text{CollectionSemantics.csmall}\text{-nstep } P\ \sigma\ n' \rrbracket$

$\implies n = n'$

$\langle proof \rangle$

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

```

```

 $\Rightarrow$  exec-step-ind (StepI (New C)) P h stk loc C0 M0 pc (Called Cs) frs sh
  (None, h(a $\rightarrow$ blank P C), (Addr a#stk, loc, C0, M0, Suc pc, No-ics)#frs, sh)

```

```

| exec-step-ind-New-Done:

```

```

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

```

```

| exec-step-ind-New-Init:

```

```

[[  $\forall obj. sh C \neq Some(obj, Done)$ ;  $\forall Cs. ics \neq Called Cs$  ]]

```

$\implies \text{exec-step-ind (StepI (New C)) } P \text{ h stk loc } C_0 \text{ } M_0 \text{ pc ics frs sh}$
 $(\text{None, h, (stk, loc, } C_0, M_0, \text{ pc, Calling C [])\#frs, sh})$

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

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

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

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

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

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

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

| *exec-step-ind-Getstatic-Done*:
 $\llbracket (D',b,t) = \text{field } P \text{ } D \text{ } F; P \vdash C \text{ has } F, \text{Static}:t \text{ in } D;$
 $\forall \text{Cs. ics} \neq \text{Called Cs}; sh \text{ } D' = \text{Some}(sfs, \text{Done});$
 $v = \text{the} (sfs \text{ } F) \rrbracket$
 $\implies \text{exec-step-ind (StepI (Getstatic C F D)) } P \text{ h stk loc } C_0 \text{ } M_0 \text{ pc ics frs sh}$

(None, h, (v#stk, loc, C₀, M₀, Suc pc, ics)#frs, sh)

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

[[(D',b,t) = field P D F; P ⊢ C has F,Static:t in D;
 ∀ sfs. sh D' ≠ Some(sfs,Done); ∀ Cs. ics ≠ Called Cs]]
 ⇒ *exec-step-ind* (StepI (Getstatic C F D)) P h stk loc C₀ M₀ pc ics frs sh
 (None, h, (stk, loc, C₀, M₀, pc, Calling D' [])#frs, sh)

| *exec-step-ind-Putfield-Null*:

hd(tl stk) = Null
 ⇒ *exec-step-ind* (StepI (Putfield F C)) P h stk loc C₀ M₀ pc ics frs sh
 ([addr-of-sys-xcpt NullPointer], h, (stk, loc, C₀, M₀, pc, ics)#frs, sh)

| *exec-step-ind-Putfield-NoField*:

[[r = hd(tl stk); a = the-Addr r; (D,fs) = the (h a); r ≠ Null; ¬(∃ t b. P ⊢ D has
 F,b:t in C)]]
 ⇒ *exec-step-ind* (StepI (Putfield F C)) P h stk loc C₀ M₀ pc ics frs sh
 ([addr-of-sys-xcpt NoSuchFieldError], h, (stk, loc, C₀, M₀, pc, ics)#frs, sh)

| *exec-step-ind-Putfield-Static*:

[[r = hd(tl stk); a = the-Addr r; (D,fs) = the (h a); r ≠ Null; P ⊢ D has F,Static:t
 in C]]
 ⇒ *exec-step-ind* (StepI (Putfield F C)) P h stk loc C₀ M₀ pc ics frs sh
 ([addr-of-sys-xcpt IncompatibleClassChangeError], h, (stk, loc, C₀, M₀, pc, ics)#frs,
 sh)

| *exec-step-ind-Putfield*:

[[v = hd stk; r = hd(tl stk); a = the-Addr r; (D,fs) = the (h a); (D',b,t) = field P
 C F;
 r ≠ Null; P ⊢ D has F,NonStatic:t in C]]
 ⇒ *exec-step-ind* (StepI (Putfield F C)) P h stk loc C₀ M₀ pc ics frs sh
 (None, h(a ↦ (D, fs((F,C) ↦ v))), (tl (tl stk), loc, C₀, M₀, pc+1, ics)#frs, sh)

| *exec-step-ind-Putstatic-NoField*:

¬(∃ t b. P ⊢ C has F,b:t in D)
 ⇒ *exec-step-ind* (StepI (Putstatic C F D)) P h stk loc C₀ M₀ pc ics frs sh
 ([addr-of-sys-xcpt NoSuchFieldError], h, (stk, loc, C₀, M₀, pc, ics)#frs, sh)

| *exec-step-ind-Putstatic-NonStatic*:

P ⊢ C has F,NonStatic:t in D
 ⇒ *exec-step-ind* (StepI (Putstatic C F D)) P h stk loc C₀ M₀ pc ics frs sh
 ([addr-of-sys-xcpt IncompatibleClassChangeError], h, (stk, loc, C₀, M₀, pc,
 ics)#frs, sh)

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

[[(D',b,t) = field P D F; P ⊢ C has F,Static:t in D; the(sh D') = (sfs,i)]]
 ⇒ *exec-step-ind* (StepI (Putstatic C F D)) P h stk loc C₀ M₀ pc (Called Cs)
 frs sh
 (None, h, (tl stk, loc, C₀, M₀, Suc pc, No-ics)#frs, sh(D' := Some ((sfs(F ↦ hd

$stk)), i))$

| *exec-step-ind-Putstatic-Done*:

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

| *exec-step-ind-Putstatic-Init*:

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

| *exec-step-ind-Checkcast*:

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

| *exec-step-ind-Checkcast-Error*:

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

| *exec-step-ind-Invoke-Null*:

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

| *exec-step-ind-Invoke-NoMethod*:

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

| *exec-step-ind-Invoke-Static*:

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

| *exec-step-ind-Invoke*:

$\llbracket ps = \text{take } n \ stk; r = stk!n; C = \text{fst}(\text{the}(\text{h}(\text{the-Addr } r)));$
 $(D, b, Ts, T, mxs, mxl_0, ins, xt) = \text{method } P \ C \ M; r \neq \text{Null};$
 $P \vdash C \text{ sees } M, \text{NonStatic}:Ts \rightarrow T = m \text{ in } D;$
 $f' = ([], [r]@\text{rev } ps)@\text{replicate } mxl_0 \ \text{undefined}, D, M, 0, \text{No-ics} \rrbracket$

$\implies \text{exec-step-ind (StepI (Invoke M n)) } P \ h \ \text{stk} \ \text{loc} \ C_0 \ M_0 \ \text{pc} \ \text{ics} \ \text{frs} \ \text{sh}$
 $(\text{None}, h, f' \# (\text{stk}, \text{loc}, C_0, M_0, \text{pc}, \text{ics}) \# \text{frs}, \text{sh})$

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

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

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

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

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

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

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

| *exec-step-ind-Return-Init:*

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

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

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

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

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

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

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

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

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

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

$([the-Addr (hd\ stk)], h, (stk, loc, C_0, M_0, pc, ics)\#frs, sh)$

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

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

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

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

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

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

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

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

| *exec-step-ind-InitThrow*:
 $exec-step-ind (StepT\ (C\#Cs)\ a)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ ics\ frs\ sh$

(None, h, (stk,loc,C₀,M₀,pc,Throwing Cs a)#frs, (sh(C ↦ (fst(the(sh C))), Error))))

| *exec-step-ind-InitThrow-End*:

exec-step-ind (StepT [] a) P h stk loc C₀ M₀ pc ics frs sh
 ([a], h, (stk,loc,C₀,M₀,pc,No-ics)#frs, sh)

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

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

— Deriving *step-input* for *exec-step-ind* from *exec-step* arguments

fun *exec-step-input* :: [jvm-prog, cname, mname, pc, init-call-status] ⇒ *step-input*
where

exec-step-input P C M pc (Calling C' Cs) = StepC C' Cs |
exec-step-input P C M pc (Called (C'#Cs)) = StepC2 C' Cs |
exec-step-input P C M pc (Throwing Cs a) = StepT Cs a |
exec-step-input P C M pc ics = StepI (instrs-of P C M ! pc)

lemma *exec-step-input-StepTD[simp]*:

assumes *exec-step-input* P C M pc ics = StepT Cs a **shows** ics = Throwing Cs a
 ⟨proof⟩

lemma *exec-step-input-StepCD[simp]*:

assumes *exec-step-input* P C M pc ics = StepC C' Cs **shows** ics = Calling C' Cs
 ⟨proof⟩

lemma *exec-step-input-StepC2D[simp]*:
assumes *exec-step-input* $P C M pc ics = StepC2 C' Cs$ **shows** $ics = Called$
 $(C' \# Cs)$
 $\langle proof \rangle$

lemma *exec-step-input-StepID*:
assumes *exec-step-input* $P C M pc ics = StepI i$
shows $(ics = Called [] \vee ics = No-ics) \wedge instrs-of P C M ! pc = i$
 $\langle proof \rangle$

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

lemma *exec-step-imp-exec-step-ind*:
assumes *es*: *exec-step* $P h stk loc C M pc ics frs sh = (xp', h', frs', sh')$
shows *exec-step-ind* $(exec-step-input P C M pc ics) P h stk loc C M pc ics frs sh$
 (xp', h', frs', sh')
 $\langle proof \rangle$

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* $P h stk loc C M pc ics frs sh = (xp', h', frs', sh')$
 $\langle proof \rangle$

lemma *exec-step-ind-equiv*:
 $exec-step P h stk loc C M pc ics frs sh = (xp', h', frs', sh')$
 $= exec-step-ind (exec-step-input P C M pc ics) P h stk loc C M pc ics frs sh (xp',$
 $h', frs', sh')$
 $\langle proof \rangle$

end

12 Instantiating *CollectionBasedRTS* with Jinja JVM

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

begin

lemma *eq-equiv[simp]*: *equiv UNIV* $\{(x, y). x = y\}$
 $\langle proof \rangle$

12.1 Some *classes-above* lemmas

lemma *start-prog-classes-above-Start*:
 $classes-above (start-prog P C M) Start = \{Object, Start\}$
 $\langle proof \rangle$

lemma *class-add-classes-above*:

assumes $ns: \neg \text{is-class } P \ C$ **and** $\neg P \vdash D \preceq^* C$
shows $\text{classes-above } (\text{class-add } P \ (C, \text{cdec})) \ D = \text{classes-above } P \ D$
 $\langle \text{proof} \rangle$

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 $\text{classes-above-xcpts } (\text{class-add } P \ (C, \text{cdec})) = \text{classes-above-xcpts } P$
 $\langle \text{proof} \rangle$

12.2 JVM next-step lemmas for initialization calling

lemma *JVM-New-next-step*:

assumes $\text{step}: \sigma' \in \text{JVMsmall } P \ \sigma$
and $nend: \sigma \notin \text{JVMendset}$
and $\text{curr}: \text{curr-instr } P \ (\text{hd}(\text{frames-of } \sigma)) = \text{New } C$
and $nDone: \neg(\exists \text{sfs } i. \text{sheap } \sigma \ C = \text{Some}(\text{sfs}, i) \wedge i = \text{Done})$
and $\text{ics}: \text{ics-of}(\text{hd}(\text{frames-of } \sigma)) = \text{No-ics}$
shows $\text{ics-of}(\text{hd}(\text{frames-of } \sigma')) = \text{Calling } C \ \square \wedge \text{sheap } \sigma = \text{sheap } \sigma' \wedge \sigma' \notin \text{JVMendset}$
 $\langle \text{proof} \rangle$

lemma *JVM-Getstatic-next-step*:

assumes $\text{step}: \sigma' \in \text{JVMsmall } P \ \sigma$
and $nend: \sigma \notin \text{JVMendset}$
and $\text{curr}: \text{curr-instr } P \ (\text{hd}(\text{frames-of } \sigma)) = \text{Getstatic } C \ F \ D$
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 $\text{ics}: \text{ics-of}(\text{hd}(\text{frames-of } \sigma)) = \text{No-ics}$
shows $\text{ics-of}(\text{hd}(\text{frames-of } \sigma')) = \text{Calling } D \ \square \wedge \text{sheap } \sigma = \text{sheap } \sigma' \wedge \sigma' \notin \text{JVMendset}$
 $\langle \text{proof} \rangle$

lemma *JVM-Putstatic-next-step*:

assumes $\text{step}: \sigma' \in \text{JVMsmall } P \ \sigma$
and $nend: \sigma \notin \text{JVMendset}$
and $\text{curr}: \text{curr-instr } P \ (\text{hd}(\text{frames-of } \sigma)) = \text{Putstatic } C \ F \ D$
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 $\text{ics}: \text{ics-of}(\text{hd}(\text{frames-of } \sigma)) = \text{No-ics}$
shows $\text{ics-of}(\text{hd}(\text{frames-of } \sigma')) = \text{Calling } D \ \square \wedge \text{sheap } \sigma = \text{sheap } \sigma' \wedge \sigma' \notin \text{JVMendset}$
 $\langle \text{proof} \rangle$

lemma *JVM-Invokestatic-next-step*:

assumes $\text{step}: \sigma' \in \text{JVMsmall } P \ \sigma$
and $nend: \sigma \notin \text{JVMendset}$
and $\text{curr}: \text{curr-instr } P \ (\text{hd}(\text{frames-of } \sigma)) = \text{Invokestatic } C \ M \ n$
and $mC: P \vdash C \text{ sees } M, \text{Static}:Ts \rightarrow T = m \text{ in } D$

and $nDone$: $\neg(\exists sfs\ i.\ sheap\ \sigma\ D = Some(sfs,i) \wedge i = Done)$
and ics : $ics-of(hd(frames-of\ \sigma)) = No-ics$
shows $ics-of(hd(frames-of\ \sigma')) = Calling\ D\ [] \wedge sheap\ \sigma = sheap\ \sigma' \wedge \sigma' \notin JVMendset$
 $\langle proof \rangle$

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**
 $jvm-deselect\ P1\ (\sigma,\ cset)\ P2 = (cset \cap (classes-changed\ P1\ P2)) = \{\}$

definition $jvm-progs$:: $jvm-prog\ set$ **where**
 $jvm-progs \equiv \{P.\ wf-jvm-prog\ P \wedge \neg is-class\ P\ Start \wedge \neg is-class\ P\ Test$
 $\wedge (\forall b'\ Ts'\ T'\ m'\ D'. P \vdash Object\ sees\ start-m,\ b' : Ts' \rightarrow T' = m' \text{ in } D'$
 $\rightarrow b' = Static \wedge Ts' = [] \wedge T' = Void)\}$

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

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

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

12.4 Definition lemmas

lemma $jvm-progs-tests-nStart$:
assumes P : $P \in jvm-progs$ **and** t : $t \in jvm-tests$
shows $\neg is-class\ (t\#P)\ Start$
 $\langle proof \rangle$

lemma $jvm-make-test-prog-classes-above-xcpts$:
assumes P : $P \in jvm-progs$ **and** t : $t \in jvm-tests$
shows $classes-above-xcpts\ (jvm-make-test-prog\ P\ t) = classes-above-xcpts\ P$
 $\langle proof \rangle$

lemma $jvm-make-test-prog-sees-Test-main$:
assumes P : $P \in jvm-progs$ **and** t : $t \in jvm-tests$

shows $\exists m. \text{jvm-make-test-prog } P \ t \vdash \text{Test sees main, Static : } [] \rightarrow \text{Void} = m \text{ in } \text{Test}$
 $\langle \text{proof} \rangle$

12.5 Naive RTS algorithm

12.5.1 Definitions

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

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

lemma $\text{jvm-naive-out-xcpts-collected}$:

assumes $o1 \in \text{jvm-naive-out } P \ t$

shows $\text{classes-above-xcpts } (\text{start-prog } (t \# P) \ (\text{fst } t) \ \text{main}) \subseteq \text{snd } o1$

$\langle \text{proof} \rangle$

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 $\text{ncollect-exec-instr}$:

assumes $\text{JVMinstr-ncollect } P \ i \ h \ \text{stk} \cap \text{classes-changed } P \ P' = \{\}$

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

and $\text{ics: ics} = \text{Called } [] \vee \text{ics} = \text{No-ics}$

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

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

$\langle \text{proof} \rangle$

lemma $\text{ncollect-JVMinstr-ncollect}$:

assumes $\text{JVMinstr-ncollect } P \ i \ h \ \text{stk} \cap \text{classes-changed } P \ P' = \{\}$

shows $\text{JVMinstr-ncollect } P \ i \ h \ \text{stk} = \text{JVMinstr-ncollect } P' \ i \ h \ \text{stk}$

$\langle \text{proof} \rangle$

lemma $\text{ncollect-exec-step}$:

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

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

shows $\text{exec-step } P \ h \ \text{stk} \ \text{loc } C \ M \ \text{pc} \ \text{ics} \ \text{frs} \ \text{sh} = \text{exec-step } P' \ h \ \text{stk} \ \text{loc } C \ M \ \text{pc} \ \text{ics}$
 $\text{frs} \ \text{sh}$

$\langle \text{proof} \rangle$

lemma $\text{ncollect-JVMstep-ncollect}$:

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

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

shows $JVMstep\text{-}ncollect\ P\ h\ stk\ C\ M\ pc\ ics = JVMstep\text{-}ncollect\ P'\ h\ stk\ C\ M\ pc\ ics$
 <proof>
lemma *ncollect-classes-above-frames*:
 $JVMexec\text{-}ncollect\ P\ (None,\ h,\ (stk,\ loc,\ C,\ M,\ pc,\ ics)\#\text{frs},\ sh) \cap \text{classes-changed}\ P\ P' = \{\}$
 $\implies \text{classes-above-frames}\ P\ \text{frs} = \text{classes-above-frames}\ P'\ \text{frs}$
 <proof>
lemma *ncollect-classes-above-xcpts*:
assumes $JVMexec\text{-}ncollect\ P\ (None,\ h,\ (stk,\ loc,\ C,\ M,\ pc,\ ics)\#\text{frs},\ sh) \cap \text{classes-changed}\ P\ P' = \{\}$
shows $\text{classes-above-xcpts}\ P = \text{classes-above-xcpts}\ P'$
 <proof>
lemma *ncollect-JVMexec-ncollect*:
assumes $JVMexec\text{-}ncollect\ P\ \sigma \cap \text{classes-changed}\ P\ P' = \{\}$
shows $JVMexec\text{-}ncollect\ P\ \sigma = JVMexec\text{-}ncollect\ P'\ \sigma$
 <proof>
lemma *ncollect-exec-instr-xcpts*:
assumes $\text{collect}: JVMinstr\text{-}ncollect\ P\ i\ h\ stk \cap \text{classes-changed}\ P\ P' = \{\}$
and $\text{xpcollect}: \text{classes-above-xcpts}\ P \cap \text{classes-changed}\ P\ P' = \{\}$
and $\text{prealloc}: \text{preallocated}\ h$
and $\sigma': \sigma' = \text{exec-instr}\ i\ P\ h\ stk\ loc\ C\ M\ pc\ ics'\ \text{frs}\ sh$
and $\text{xp}: \text{fst}\ \sigma' = \text{Some}\ a$
and $i: i = \text{instrs-of}\ P\ C\ M\ !\ pc$
shows $\text{classes-above}\ P\ (\text{cname-of}\ h\ a) \cap \text{classes-changed}\ P\ P' = \{\}$
 <proof>
lemma *ncollect-exec-step-xcpts*:
assumes $\text{collect}: JVMstep\text{-}ncollect\ P\ h\ stk\ C\ M\ pc\ ics \cap \text{classes-changed}\ P\ P' = \{\}$
and $\text{xpcollect}: \text{classes-above-xcpts}\ P \cap \text{classes-changed}\ P\ P' = \{\}$
and $\text{prealloc}: \text{preallocated}\ h$
and $\sigma': \sigma' = \text{exec-step}\ P\ h\ stk\ loc\ C\ M\ pc\ ics'\ \text{frs}\ sh$
and $\text{xp}: \text{fst}\ \sigma' = \text{Some}\ a$
shows $\text{classes-above}\ P\ (\text{cname-of}\ h\ a) \cap \text{classes-changed}\ P\ P' = \{\}$
 <proof>
lemma *ncollect-JVMsmall*:
assumes $\text{collect}: (\sigma',\ cset) \in JVMNaiveCollectionSemantics.\text{csmall}\ P\ \sigma$
and $\text{intersect}: cset \cap \text{classes-changed}\ P\ P' = \{\}$
and $\text{prealloc}: \text{preallocated}\ (\text{fst}(\text{snd}\ \sigma))$
shows $(\sigma',\ cset) \in JVMNaiveCollectionSemantics.\text{csmall}\ P'\ \sigma$
 <proof>
lemma *ncollect-JVMbig*:
assumes $\text{collect}: (\sigma',\ cset) \in JVMNaiveCollectionSemantics.\text{cbig}\ P\ \sigma$
and $\text{intersect}: cset \cap \text{classes-changed}\ P\ P' = \{\}$
and $\text{prealloc}: \text{preallocated}\ (\text{fst}(\text{snd}\ \sigma))$
shows $(\sigma',\ cset) \in JVMNaiveCollectionSemantics.\text{cbig}\ P'\ \sigma$
 <proof>
theorem *jvm-naive-existence-safe*:
assumes $p: P \in \text{jvm-progs}$ **and** $P' \in \text{jvm-progs}$ **and** $t: t \in \text{jvm-tests}$

and $out: o1 \in jvm-naive-out\ P\ t$ **and** $jvm-deselect\ P\ o1\ P'$
shows $\exists o2 \in jvm-naive-out\ P'\ t. o1 = o2$
 $\langle proof \rangle$
interpretation $JVMNaiveCollectionRTS$:
 $CollectionBasedRTS (=) jvm-deselect\ jvm-progs\ jvm-tests$
 $JVMendset\ JVMcombine\ JVMcollect-id\ JVMsmall\ JVMNaiveCollect\ jvm-naive-out$
 $jvm-make-test-prog\ jvm-naive-collect-start$
 $\langle proof \rangle$

12.6 Smarter RTS algorithm

12.6.1 Definitions and helper lemmas

fun $jvm-smart-out :: jvm-prog \Rightarrow jvm-class \Rightarrow jvm-prog-out\ set$ **where**
 $jvm-smart-out\ P\ t$
 $= \{(\sigma', coll'). \exists coll. (\sigma', coll) \in JVMSmartCollectionSemantics.cbigs$
 $(jvm-make-test-prog\ P\ t)\ (start-state\ (t\#P))$
 $\wedge coll' = coll \cup classes-above-xcpts\ P \cup \{Object, Start\}\}$

abbreviation $jvm-smart-collect-start :: jvm-prog \Rightarrow cname\ set$ **where**
 $jvm-smart-collect-start\ P \equiv classes-above-xcpts\ P \cup \{Object, Start\}$

lemma $jvm-naive-iff-smart$:
 $(\exists cset_n. (\sigma', cset_n) \in jvm-naive-out\ P\ t) \longleftrightarrow (\exists cset_s. (\sigma', cset_s) \in jvm-smart-out\ P\ t)$
 $\langle proof \rangle$

lemma $jvm-smart-out-classes-above-xcpts$:
assumes $s: (\sigma', cset_s) \in jvm-smart-out\ P\ t$ **and** $P: P \in jvm-progs$ **and** $t: t \in jvm-tests$
shows $classes-above-xcpts\ (jvm-make-test-prog\ P\ t) \subseteq cset_s$
 $\langle proof \rangle$

lemma $jvm-smart-collect-start-make-test-prog$:
 $\llbracket P \in jvm-progs; t \in jvm-tests \rrbracket$
 $\implies jvm-smart-collect-start\ (jvm-make-test-prog\ P\ t) = jvm-smart-collect-start\ P$
 $\langle proof \rangle$

lemma $jvm-smart-out-classes-above-start-heap$:
assumes $s: (\sigma', cset_s) \in jvm-smart-out\ P\ t$ **and** $h: start-heap\ (t\#P)\ a = Some(C, fs)$
and $P: P \in jvm-progs$ **and** $t: t \in jvm-tests$
shows $classes-above\ (jvm-make-test-prog\ P\ t)\ C \subseteq cset_s$
 $\langle proof \rangle$

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

<proof>

lemma *jvm-smart-out-classes-above-frames*:

$(\sigma', cset_s) \in \text{jvm-smart-out } P \ t$
 $\implies \text{classes-above-frames } (\text{jvm-make-test-prog } P \ t) \ (\text{frames-of } (\text{start-state } (t\#P)))$
 $\subseteq cset_s$
<proof>

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* $\implies v = \text{Null} \vee (\exists a. v = \text{Addr } a \wedge (\exists Cfs. h \ a = \text{Some } Cfs))$

<proof>

fun *stack-safe* :: *instr* \Rightarrow *heap* \Rightarrow *val list* \Rightarrow *bool* **where**

stack-safe (*Getfield* *F C*) *h* *stk* = ($\text{length } stk > 0 \wedge \text{is_ptr } h \ (\text{hd } stk)$) |

stack-safe (*Putfield* *F C*) *h* *stk* = ($\text{length } stk > 1 \wedge \text{is_ptr } h \ (\text{hd } (\text{tl } stk))$) |

stack-safe (*Checkcast* *C*) *h* *stk* = ($\text{length } stk > 0 \wedge \text{is_ptr } h \ (\text{hd } stk)$) |

stack-safe (*Invoke* *M n*) *h* *stk* = ($\text{length } stk > n \wedge \text{is_ptr } h \ (stk \ ! \ n)$) |

stack-safe *JVMInstructions.Throw* *h* *stk* = ($\text{length } stk > 0 \wedge \text{is_ptr } h \ (\text{hd } stk)$) |

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>

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

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

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

— defs

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

— backward promises - will be collected prior

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

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

— collection and step

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

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

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

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

$\wedge (classes\text{-}above\text{-}frames\ P\ frs' \subseteq cset)$

<proof>

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>

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

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

and σ' : $(\sigma', \text{cset}') \in \text{JVMSmartCollectionSemantics.csmall } P \ \sigma$
shows $\sigma' \notin \text{JVMendset} \wedge \text{ics-of } (\text{hd}(\text{frames-of } \sigma')) = \text{Calling } C \ Cs$
 $\wedge (\exists \text{sfs. sheap } \sigma' \ C = \text{Some}(\text{sfs}, \text{Prepared}))$
 $\wedge (\forall C'. P \vdash C \preceq^* C' \longrightarrow C \neq C')$
 $\longrightarrow (\exists \text{sfs } i. \text{sheap } \sigma' \ C' = \text{Some}(\text{sfs}, i)) \longrightarrow \text{classes-above } P \ C' \subseteq \text{cset}$
 $\langle \text{proof} \rangle$

lemma *Calling-Prepared-next-state*:
assumes $\text{sub: } P \vdash C \prec^1 D$
and $\text{obj: } P \vdash D \preceq^* \text{Object}$
and $\text{ics: } \text{ics-of } (\text{hd}(\text{frames-of } \sigma)) = \text{Calling } C \ Cs$
and $\text{prep: } \text{sheap } \sigma \ C = \text{Some}(\text{sfs}, \text{Prepared})$
and $\text{set: } \forall C'. P \vdash C \preceq^* C' \longrightarrow C \neq C' \longrightarrow (\exists \text{sfs } i. \text{sheap } \sigma \ C' = \text{Some}(\text{sfs}, i))$
 $\longrightarrow \text{classes-above } P \ C' \subseteq \text{cset}$
and σ' : $(\sigma', \text{cset}') \in \text{JVMSmartCollectionSemantics.csmall } P \ \sigma$
shows $\sigma' \notin \text{JVMendset} \wedge \text{ics-of } (\text{hd}(\text{frames-of } \sigma')) = \text{Calling } D \ (C\#Cs)$
 $\wedge (\forall C'. P \vdash D \preceq^* C' \longrightarrow (\exists \text{sfs } i. \text{sheap } \sigma' \ C' = \text{Some}(\text{sfs}, i))$
 $\longrightarrow \text{classes-above } P \ C' \subseteq \text{cset})$
 $\langle \text{proof} \rangle$

lemma *Calling-collects-IH*:
assumes $\text{sub: } P \vdash C \prec^1 D$
and $\text{obj: } P \vdash D \preceq^* \text{Object}$
and $\text{step: } \bigwedge \sigma \ \text{cset}' \ Cs. (\sigma', \text{cset}') \in \text{JVMSmartCollectionSemantics.cbig } P \ \sigma \implies$
 $\sigma \notin \text{JVMendset}$
 $\implies \text{ics-of } (\text{hd}(\text{frames-of } \sigma)) = \text{Calling } D \ Cs$
 $\implies \forall C'. P \vdash D \preceq^* C' \longrightarrow (\exists \text{sfs } i. \text{sheap } \sigma \ C' = \text{Some}(\text{sfs}, i))$
 $\longrightarrow \text{classes-above } P \ C' \subseteq \text{cset}$
 $\implies \text{classes-above } P \ D \subseteq \text{cset} \cup \text{cset}'$
and $\text{big: } (\sigma', \text{cset}') \in \text{JVMSmartCollectionSemantics.cbig } P \ \sigma$
and $\text{nend: } \sigma \notin \text{JVMendset}$
and $\text{curr: } \text{ics-of } (\text{hd}(\text{frames-of } \sigma)) = \text{Calling } C \ Cs$
and $\text{set: } \forall C'. P \vdash C \preceq^* C' \longrightarrow (\exists \text{sfs } i. \text{sheap } \sigma \ C' = \text{Some}(\text{sfs}, i))$
 $\longrightarrow \text{classes-above } P \ C' \subseteq \text{cset}$

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

lemma *Calling-collects*:
assumes $\text{sub: } P \vdash C \preceq^* \text{Object}$
and $(\sigma', \text{cset}') \in \text{JVMSmartCollectionSemantics.cbig } P \ \sigma$
and $\sigma \notin \text{JVMendset}$
and $\text{ics-of } (\text{hd}(\text{frames-of } \sigma)) = \text{Calling } C \ Cs$
and $\forall C'. P \vdash C \preceq^* C' \longrightarrow (\exists \text{sfs } i. \text{sheap } \sigma \ C' = \text{Some}(\text{sfs}, i))$
 $\longrightarrow \text{classes-above } P \ C' \subseteq \text{cset}$
and $\text{cset}' \subseteq \text{cset}$

shows $\text{classes-above } P \ C \subseteq \text{cset}$
 $\langle \text{proof} \rangle$

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 *cbig*: $(\sigma', cset') \in \text{JVMSmartCollectionSemantics.cbig } P \sigma$
and *nend*: $\sigma \notin \text{JVMendset}$
and *curr*: $\text{curr-instr } P (\text{hd}(\text{frames-of } \sigma)) = \text{New } C$
and *ics*: $\text{ics-of } (\text{hd}(\text{frames-of } \sigma)) = \text{No-ics}$
and *sheap*: $\forall C'. P \vdash C \preceq^* C' \longrightarrow (\exists \text{sfs } i. \text{sheap } \sigma C' = \text{Some}(\text{sfs}, i))$
 $\longrightarrow \text{classes-above } P C' \subseteq cset$
and *smart*: $cset' \subseteq cset$
shows $\text{classes-above } P C \subseteq cset$
<proof>

lemma *Getstatic-collects*:

assumes *sub*: $P \vdash D \preceq^* \text{Object}$
and *cbig*: $(\sigma', cset') \in \text{JVMSmartCollectionSemantics.cbig } P \sigma$
and *nend*: $\sigma \notin \text{JVMendset}$
and *curr*: $\text{curr-instr } P (\text{hd}(\text{frames-of } \sigma)) = \text{Getstatic } C F D$
and *ics*: $\text{ics-of } (\text{hd}(\text{frames-of } \sigma)) = \text{No-ics}$
and *fC*: $P \vdash C \text{ has } F, \text{Static:t in } D$
and *sheap*: $\forall C'. P \vdash D \preceq^* C' \longrightarrow (\exists \text{sfs } i. \text{sheap } \sigma C' = \text{Some}(\text{sfs}, i))$
 $\longrightarrow \text{classes-above } P C' \subseteq cset$
and *smart*: $cset' \subseteq cset$
shows $\text{classes-above } P D \subseteq cset$
<proof>

lemma *Putstatic-collects*:

assumes *sub*: $P \vdash D \preceq^* \text{Object}$
and *cbig*: $(\sigma', cset') \in \text{JVMSmartCollectionSemantics.cbig } P \sigma$
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*: $P \vdash C \text{ has } F, \text{Static:t in } D$
and *sheap*: $\forall C'. P \vdash D \preceq^* C' \longrightarrow (\exists \text{sfs } i. \text{sheap } \sigma C' = \text{Some}(\text{sfs}, i))$
 $\longrightarrow \text{classes-above } P C' \subseteq cset$
and *smart*: $cset' \subseteq cset$
shows $\text{classes-above } P D \subseteq cset$
<proof>

lemma *Invokestatic-collects*:

assumes *sub*: $P \vdash D \preceq^* \text{Object}$
and *cbig*: $(\sigma', cset') \in \text{JVMSmartCollectionSemantics.cbig } P \sigma$
and *smart*: $cset' \subseteq cset$
and *nend*: $\sigma \notin \text{JVMendset}$
and *curr*: $\text{curr-instr } P (\text{hd}(\text{frames-of } \sigma)) = \text{Invokestatic } C M n$
and *ics*: $\text{ics-of } (\text{hd}(\text{frames-of } \sigma)) = \text{No-ics}$
and *mC*: $P \vdash C \text{ sees } M, \text{Static:Ts} \rightarrow T = m \text{ in } D$
and *sheap*: $\forall C'. P \vdash D \preceq^* C' \longrightarrow (\exists \text{sfs } i. \text{sheap } \sigma C' = \text{Some}(\text{sfs}, i))$
 $\longrightarrow \text{classes-above } P C' \subseteq cset$
shows $\text{classes-above } P D \subseteq cset$
<proof>

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

lemma *jvm-smart-out-classes-above-Test*:

assumes $s: (\sigma', cset_s) \in \text{jvm-smart-out } P \ t$ **and** $P: P \in \text{jvm-progs}$ **and** $t: t \in \text{jvm-tests}$

shows $\text{classes-above } (\text{jvm-make-test-prog } P \ t) \ \text{Test} \subseteq cset_s$

(**is** $\text{classes-above } ?P \ ?D \subseteq ?cset$)

<proof>

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

lemma *jvm-naive-to-smart-exec-collect*:

assumes

— well-formedness

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

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

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

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

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

— forward promises - will be collected after if not already

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

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

$C \subseteq cset$

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

— collection

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

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

<proof>

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

assumes

— well-formedness

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

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

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

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

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

— forward promises - will be collected after if not already

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

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

$\subseteq cset$

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 $cset_n \subseteq cset$
 $\langle \text{proof} \rangle$
lemma *jvm-naive-to-smart-csmall-nstep*:
 $\llbracket \text{wf-jvm-prog}_{\Phi} P;$
 $P, \Phi \vdash (xp, h, frs, sh) \surd;$
 $hd \text{ frs} = (stk, loc, C', M', pc, ics);$
 $\bigwedge C fs. \exists a. h a = \text{Some}(C, fs) \implies \text{classes-above } P C \subseteq cset;$
 $\bigwedge C sfs i. sh C = \text{Some}(sfs, i) \implies \text{classes-above } P C \subseteq cset;$
 $\text{classes-above-xcpts } P \subseteq cset;$
 $\text{classes-above-frames } P \text{ frs} \subseteq cset;$
 $\bigwedge C. ics = \text{Called } \square \vee ics = \text{No-ics}$
 $\implies \text{coll-init-class } P (\text{instrs-of } P C' M' ! pc) = \text{Some } C \implies \text{classes-above } P$
 $C \subseteq cset;$
 $\bigwedge C' Cs'. ics = \text{Calling } C' Cs' \implies \text{classes-above } P C' \subseteq cset;$
 $(\sigma', cset_n) \in \text{JVMNaiveCollectionSemantics.csmall-nstep } P (xp, h, frs, sh) n;$
 $(\sigma', cset_s) \in \text{JVMSmartCollectionSemantics.csmall-nstep } P (xp, h, frs, sh) n;$
 $cset_s \subseteq cset;$
 $\sigma' \in \text{JVMendset} \rrbracket$
 $\implies cset_n \subseteq cset$
 $\langle \text{proof} \rangle$
lemma *jvm-naive-to-smart-cbig*:
assumes
— well-formedness
 $wtp: \text{wf-jvm-prog}_{\Phi} P$
and *correct*: $P, \Phi \vdash (xp, h, frs, sh) \surd$
— defs
and *f'*: $hd \text{ frs} = (stk, loc, C', M', pc, ics)$
— backward promises - will be collected/maintained 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*: $\text{classes-above-xcpts } P \subseteq cset$
and *frames*: $\text{classes-above-frames } P \text{ frs} \subseteq cset$
— forward promises - will be collected after if not already
and *init-class-prom*: $\bigwedge C. ics = \text{Called } \square \vee ics = \text{No-ics}$
 $\implies \text{coll-init-class } P (\text{instrs-of } P C' M' ! pc) = \text{Some } C \implies \text{classes-above } P C$
 $\subseteq cset$
and *Calling-prom*: $\bigwedge C' Cs'. ics = \text{Calling } C' Cs' \implies \text{classes-above } P C' \subseteq cset$
— collections
and *n*: $(\sigma', cset_n) \in \text{JVMNaiveCollectionSemantics.cbig } P (xp, h, frs, sh)$
and *s*: $(\sigma', cset_s) \in \text{JVMSmartCollectionSemantics.cbig } P (xp, h, frs, sh)$
and *smart*: $cset_s \subseteq cset$
shows $cset_n \subseteq cset$
 $\langle \text{proof} \rangle$
lemma *jvm-naive-to-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 $cset_n \subseteq cset_s$
 $\langle \text{proof} \rangle$

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*:
 $\text{JVMe}x\text{ec-scollect } P \sigma \subseteq \text{JVMe}x\text{ec-ncollect } P \sigma$
 $\langle \text{proof} \rangle$

lemma *jvm-smart-to-naive-csmall*:
assumes $(\sigma', cset_n) \in \text{JVMe}x\text{ec-csmall } P \sigma$
and $(\sigma', cset_s) \in \text{JVMe}x\text{ec-smart-csmall } P \sigma$
shows $cset_s \subseteq cset_n$
 $\langle \text{proof} \rangle$

lemma *jvm-smart-to-naive-csmall-nstep*:
 $\llbracket (\sigma', cset_n) \in \text{JVMe}x\text{ec-csmall-nstep } P \sigma n;$
 $(\sigma', cset_s) \in \text{JVMe}x\text{ec-smart-csmall-nstep } P \sigma n \rrbracket$
 $\implies cset_s \subseteq cset_n$
 $\langle \text{proof} \rangle$

lemma *jvm-smart-to-naive-cbig*:
assumes $n: (\sigma', cset_n) \in \text{JVMe}x\text{ec-cbig } P \sigma$
and $s: (\sigma', cset_s) \in \text{JVMe}x\text{ec-smart-cbig } P \sigma$
shows $cset_s \subseteq cset_n$
 $\langle \text{proof} \rangle$

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 $cset_s \subseteq cset_n$
 $\langle \text{proof} \rangle$

12.6.5 Safety of the smart algorithm

Having proved containment in both directions, we get $\text{naive} = \text{smart}$:

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 $cset_n = cset_s$

<proof>

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*: *jvm-deselect* *P* *o1* *P'*

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

<proof>

...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*

<proof>

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.