# Regression Test Selection over JVM

Susannah Mansky

September 13, 2023

**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

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

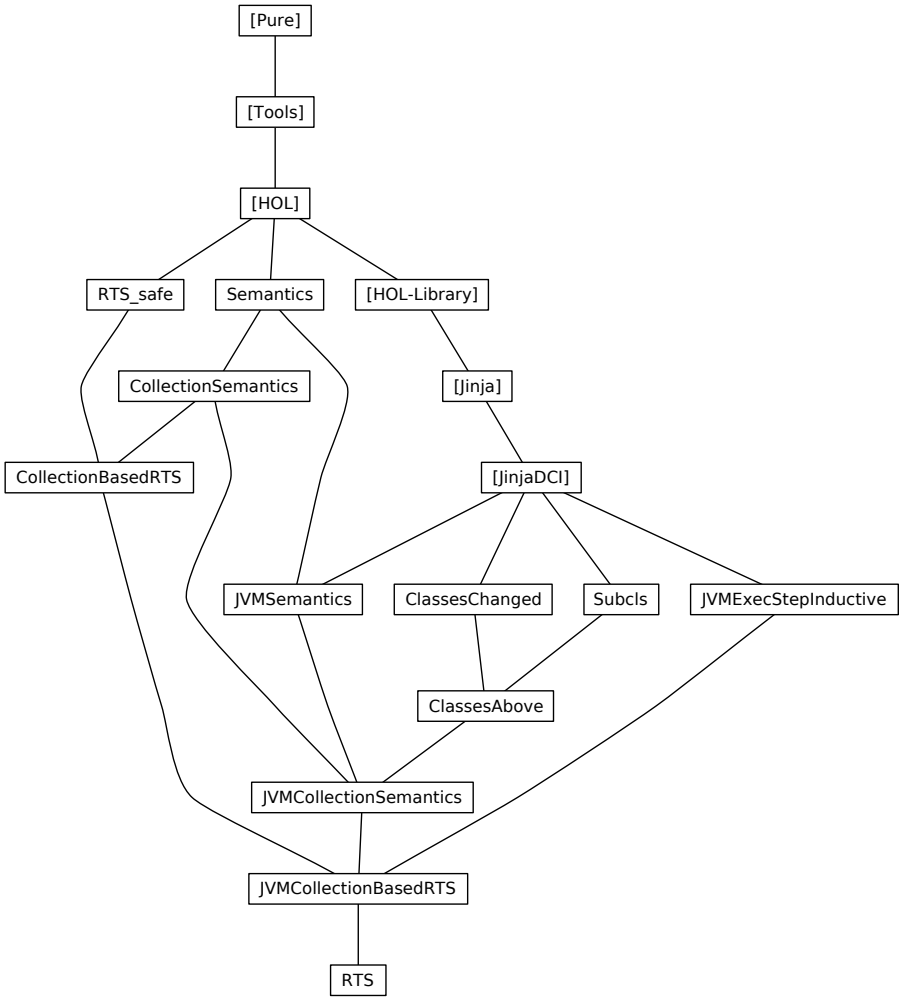


Figure 1: Theory Dependency Graph

# 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* ⇒ *'test* ⇒ *'prog-out set* **and**
*equiv-out* :: *'prog-out* ⇒ *'prog-out* ⇒ *bool* **and**
*deselect* :: *'prog* ⇒ *'prog-out* ⇒ *'prog* ⇒ *bool* **and**
*progs* :: *'prog set* **and**
*tests* :: *'test set*
**assumes**
*existence-safe*: ⟦ *P* ∈ *progs*; *P'* ∈ *progs*; *t* ∈ *tests*; *o1* ∈ *out P t*; *deselect P o1 P'* ⟧
⟹ (∃ *o2* ∈ *out P' t*. *equiv-out o1 o2*) **and**
*equiv-out-equiv*: *equiv UNIV* {(*x*,*y*). *equiv-out x y*} **and**
*equiv-out-deselect*: ⟦ *equiv-out o1 o2*; *deselect P o1 P'* ⟧ ⟹ *deselect P o2 P'*

**context** *RTS-safe* **begin**

**lemma** *equiv-out-refl*: *equiv-out a a*
⟨*proof*⟩

**lemma** *equiv-out-trans*: ⟦ *equiv-out a b*; *equiv-out b c* ⟧ ⟹ *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*: *Ps* ≠ [] *set Ps* ⊆ *progs t* ∈ *tests* **and**
*o0*: $o_0$ ∈ *out* (*Ps*!*0*) *t* **and**
*des*: ∀ *n* < (*length Ps*) − *1*. *deselect* (*Ps*!*n*) $o_0$ (*Ps*!(*Suc n*))
**shows** ∃ $o_n$ ∈ *out* (*last Ps*) *t*. *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* ⇒ *'state* ⇒ *'state set* **and**
*endset* :: *'state set*
**assumes**
*endset-final*: $\sigma \in endset \Longrightarrow \forall P.\ small\ P\ \sigma = \{\}$

**context** *Semantics* **begin**

## 3.1 Extending *small* to multiple steps

**primrec** *small-nstep* :: *'prog* ⇒ *'state* ⇒ *nat* ⇒ *'state set* **where**
*small-nstep-base*:
$small\text{-}nstep\ P\ \sigma\ 0 = \{\sigma\}$ |
*small-nstep-Rec*:
$small\text{-}nstep\ P\ \sigma\ (Suc\ n) =$
$\{\ \sigma 2.\ \exists \sigma 1.\ \sigma 1 \in small\text{-}nstep\ P\ \sigma\ n \wedge \sigma 2 \in small\ P\ \sigma 1\ \}$

**lemma** *small-nstep-Rec2*:
$small\text{-}nstep\ P\ \sigma\ (Suc\ n) =$
$\{\ \sigma 2.\ \exists \sigma 1.\ \sigma 1 \in small\ P\ \sigma \wedge \sigma 2 \in small\text{-}nstep\ P\ \sigma 1\ n\ \}$
⟨*proof*⟩

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

**lemma** *small-nstep-Suc-nend*: $\sigma' \in small\text{-}nstep\ P\ \sigma\ (Suc\ n1) \Longrightarrow \sigma \notin endset$
⟨*proof*⟩

## 3.2 Extending *small* to a big-step semantics

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

**lemma** *bigI*:
$[\![\ \sigma' \in small\text{-}nstep\ P\ \sigma\ n; \sigma' \in endset\ ]\!] \Longrightarrow \sigma' \in big\ P\ \sigma$
⟨*proof*⟩

**lemma** *bigD*:
$[\![\ \sigma' \in big\ P\ \sigma\ ]\!] \Longrightarrow \exists n.\ \sigma' \in small\text{-}nstep\ P\ \sigma\ n \wedge \sigma' \in endset$
⟨*proof*⟩

**lemma** *big-def2*:

$\sigma' \in big\ P\ \sigma \longleftrightarrow (\exists\, n.\ \sigma' \in small\text{-}nstep\ P\ \sigma\ n \wedge \sigma' \in endset)$
⟨*proof*⟩

**lemma** *big-stepD*:
**assumes** *big*: $\sigma' \in big\ P\ \sigma$ **and** *nend*: $\sigma \notin endset$
**shows** $\exists\, \sigma 1.\ \sigma 1 \in small\ P\ \sigma \wedge \sigma' \in big\ P\ \sigma 1$
⟨*proof*⟩

**lemma** *small-nstep-det-last-eq*:
**assumes** *det*: $\forall \sigma.\ small\ P\ \sigma = \{\} \vee (\exists \sigma'.\ small\ P\ \sigma = \{\sigma'\})$
**shows** $[\![\ \sigma' \in big\ P\ \sigma;\ \sigma' \in small\text{-}nstep\ P\ \sigma\ n;\ \sigma' \in small\text{-}nstep\ P\ \sigma\ n'\ ]\!] \Longrightarrow n = n'$
⟨*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* ⇒ *'state* ⇒ *'state set* **and**
*endset* :: *'state set*
**fixes**
*collect* :: *'prog* ⇒ *'state* ⇒ *'state* ⇒ *'coll* **and**
*combine* :: *'coll* ⇒ *'coll* ⇒ *'coll* **and**
*collect-id* :: *'coll*
**assumes**
*combine-assoc*: *combine* (*combine c1 c2*) *c3* = *combine c1* (*combine c2 c3*) **and**
*collect-idl*[*simp*]: *combine collect-id c* = *c* **and**
*collect-idr*[*simp*]: *combine c collect-id* = *c*

**context** *CollectionSemantics* **begin**

## 4.1 Small-Step Collection Semantics

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

**lemma** *small-det-csmall-det*:
**assumes** $\forall \sigma.\ small\ P\ \sigma = \{\} \vee (\exists \sigma'.\ small\ P\ \sigma = \{\sigma'\})$
**shows** $\forall \sigma.\ csmall\ P\ \sigma = \{\} \vee (\exists o'.\ csmall\ P\ \sigma = \{o'\})$
⟨*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*:
$csmall\text{-}nstep\ P\ \sigma\ 0 = \{(\sigma,\ collect\text{-}id)\}$ |
*csmall-nstep-Rec*:
$csmall\text{-}nstep\ P\ \sigma\ (Suc\ n) =$
$\{\ (\sigma 2,\ coll).\ \exists \sigma 1\ coll1\ coll2.\ (\sigma 1,\ coll1) \in csmall\text{-}nstep\ P\ \sigma\ n$
$\wedge\ (\sigma 2,\ coll2) \in csmall\ P\ \sigma 1 \wedge combine\ coll1\ coll2 = coll\ \}$

**lemma** *small-nstep-csmall-nstep-equiv*:
$small\text{-}nstep\ P\ \sigma\ n$
$= \{\ \sigma'.\ \exists coll.\ (\sigma',\ coll) \in csmall\text{-}nstep\ P\ \sigma\ n\ \}$
⟨*proof*⟩

**lemma** *csmall-nstep-exists*:
$\sigma' \in big\ P\ \sigma \Longrightarrow \exists n\ coll.\ (\sigma',\ coll) \in csmall\text{-}nstep\ P\ \sigma\ n \wedge \sigma' \in endset$
⟨*proof*⟩

**lemma** *csmall-det-csmall-nstep-det*:
**assumes** $\forall \sigma.\ csmall\ P\ \sigma = \{\} \vee (\exists o'.\ csmall\ P\ \sigma = \{o'\})$
**shows** $\forall \sigma.\ csmall\text{-}nstep\ P\ \sigma\ n = \{\} \vee (\exists o'.\ csmall\text{-}nstep\ P\ \sigma\ n = \{o'\})$
⟨*proof*⟩

**lemma** *csmall-nstep-Rec2*:
$csmall\text{-}nstep\ P\ \sigma\ (Suc\ n) =$
$\{\ (\sigma 2,\ coll).\ \exists \sigma 1\ coll1\ coll2.\ (\sigma 1,\ coll1) \in csmall\ P\ \sigma$
$\wedge\ (\sigma 2,\ coll2) \in csmall\text{-}nstep\ P\ \sigma 1\ n \wedge combine\ coll1\ coll2 = coll$
$\}$
⟨*proof*⟩

**lemma** *csmall-nstep-SucD*:
**assumes** $(\sigma',coll') \in csmall\text{-}nstep\ P\ \sigma\ (Suc\ n)$
**shows** $\exists \sigma 1\ coll1.\ (\sigma 1,\ coll1) \in csmall\ P\ \sigma$
$\wedge\ (\exists coll.\ coll' = combine\ coll1\ coll \wedge (\sigma',coll) \in csmall\text{-}nstep\ P\ \sigma 1\ n)$
⟨*proof*⟩

**lemma** *csmall-nstep-Suc-nend*: $o' \in csmall\text{-}nstep\ P\ \sigma\ (Suc\ n1) \Longrightarrow \sigma \notin endset$
⟨*proof*⟩

**lemma** *small-to-csmall-nstep-pres*:
**assumes** *Qpres*: $\bigwedge P\ \sigma\ \sigma'.\ Q\ P\ \sigma \Longrightarrow \sigma' \in small\ P\ \sigma \Longrightarrow Q\ P\ \sigma'$
**shows** $Q\ P\ \sigma \Longrightarrow (\sigma',\ coll) \in csmall\text{-}nstep\ P\ \sigma\ n \Longrightarrow Q\ P\ \sigma'$
⟨*proof*⟩

**lemma** *csmall-to-csmall-nstep-prop*:
**assumes** *cond*: $\bigwedge P\ \sigma\ \sigma'\ coll.\ (\sigma',\ coll) \in csmall\ P\ \sigma \Longrightarrow R\ P\ coll \Longrightarrow Q\ P\ \sigma \Longrightarrow R'\ P\ \sigma\ \sigma'\ coll$
**and** *Rcomb*: $\bigwedge P\ coll1\ coll2.\ R\ P\ (combine\ coll1\ coll2) = (R\ P\ coll1 \wedge R\ P\ coll2)$
**and** *Qpres*: $\bigwedge P\ \sigma\ \sigma'.\ Q\ P\ \sigma \Longrightarrow \sigma' \in small\ P\ \sigma \Longrightarrow Q\ P\ \sigma'$
**and** *Rtrans'*: $\bigwedge P\ \sigma\ \sigma 1\ \sigma'\ coll1\ coll2.$
$R'\ P\ \sigma\ \sigma 1\ coll1 \wedge R'\ P\ \sigma 1\ \sigma'\ coll2 \Longrightarrow R'\ P\ \sigma\ \sigma'\ (combine\ coll1\ coll2)$
**and** *base*: $\bigwedge \sigma.\ R'\ P\ \sigma\ \sigma\ collect\text{-}id$
**shows** $(\sigma',\ coll) \in csmall\text{-}nstep\ P\ \sigma\ n \Longrightarrow R\ P\ coll \Longrightarrow Q\ P\ \sigma \Longrightarrow R'\ P\ \sigma\ \sigma'\ coll$
⟨*proof*⟩

**lemma** *csmall-to-csmall-nstep-prop2*:
**assumes** *cond*: $\bigwedge P\ P'\ \sigma\ \sigma'\ coll.\ (\sigma',\ coll) \in csmall\ P\ \sigma$
$\Longrightarrow R\ P\ P'\ coll \Longrightarrow Q\ \sigma \Longrightarrow (\sigma',\ coll) \in csmall\ P'\ \sigma$
**and** *Rcomb*: $\bigwedge P\ P'\ coll1\ coll2.\ R\ P\ P'\ (combine\ coll1\ coll2) = (R\ P\ P'\ coll1 \wedge R\ P\ P'\ coll2)$
**and** *Qpres*: $\bigwedge P\ \sigma\ \sigma'.\ Q\ \sigma \Longrightarrow \sigma' \in small\ P\ \sigma \Longrightarrow Q\ \sigma'$
**shows** $(\sigma',\ coll) \in csmall\text{-}nstep\ P\ \sigma\ n \Longrightarrow R\ P\ P'\ coll \Longrightarrow Q\ \sigma \Longrightarrow (\sigma',\ coll) \in csmall\text{-}nstep\ P'\ \sigma\ n$
⟨*proof*⟩

## 4.3 Extending *csmall* to a big-step semantics

**definition** *cbig* :: *'prog* ⇒ *'state* ⇒ (*'state* × *'coll*) *set* **where**
*cbig P σ* ≡
{ (*σ'*, *coll*). ∃ *n*. (*σ'*, *coll*) ∈ *csmall-nstep P σ n* ∧ *σ'* ∈ *endset* }

**lemma** *cbigD*:
⟦ (*σ'*,*coll'*) ∈ *cbig P σ* ⟧ ⟹ ∃ *n*. (*σ'*,*coll'*) ∈ *csmall-nstep P σ n* ∧ *σ'* ∈ *endset*
⟨*proof*⟩

**lemma** *cbigD'*:
⟦ *o'* ∈ *cbig P σ* ⟧ ⟹ ∃ *n*. *o'* ∈ *csmall-nstep P σ n* ∧ *fst o'* ∈ *endset*
⟨*proof*⟩

**lemma** *cbig-def2*:
(*σ'*, *coll*) ∈ *cbig P σ* ⟷ (∃ *n*. (*σ'*, *coll*) ∈ *csmall-nstep P σ n* ∧ *σ'* ∈ *endset*)
⟨*proof*⟩

**lemma** *cbig-big-equiv*:
(∃ *coll*. (*σ'*, *coll*) ∈ *cbig P σ*) ⟷ *σ'* ∈ *big P σ*
⟨*proof*⟩

**lemma** *cbig-big-implies*:
(*σ'*, *coll*) ∈ *cbig P σ* ⟹ *σ'* ∈ *big P σ*
⟨*proof*⟩

**lemma** *csmall-to-cbig-prop*:
**assumes** $\bigwedge P\ \sigma\ \sigma'\ coll.\ (\sigma',\ coll) \in csmall\ P\ \sigma \Longrightarrow R\ P\ coll \Longrightarrow Q\ P\ \sigma \Longrightarrow R'\ P\ \sigma\ \sigma'\ coll$
**and** $\bigwedge P\ coll1\ coll2.\ R\ P\ (combine\ coll1\ coll2) = (R\ P\ coll1 \wedge R\ P\ coll2)$
**and** $\bigwedge P\ \sigma\ \sigma'.\ Q\ P\ \sigma \Longrightarrow \sigma' \in small\ P\ \sigma \Longrightarrow Q\ P\ \sigma'$
**and** $\bigwedge P\ \sigma\ \sigma 1\ \sigma'\ coll1\ coll2.$
$R'\ P\ \sigma\ \sigma 1\ coll1 \wedge R'\ P\ \sigma 1\ \sigma'\ coll2 \Longrightarrow R'\ P\ \sigma\ \sigma'\ (combine\ coll1\ coll2)$
**and** $\bigwedge \sigma.\ R'\ P\ \sigma\ \sigma\ collect\text{-}id$
**shows** $(\sigma',\ coll) \in cbig\ P\ \sigma \Longrightarrow R\ P\ coll \Longrightarrow Q\ P\ \sigma \Longrightarrow R'\ P\ \sigma\ \sigma'\ coll$
⟨*proof*⟩

**lemma** *csmall-to-cbig-prop2*:
**assumes** $\bigwedge P\ P'\ \sigma\ \sigma'\ coll.\ (\sigma',\ coll) \in csmall\ P\ \sigma \Longrightarrow R\ P\ P'\ coll \Longrightarrow Q\ \sigma \Longrightarrow (\sigma',\ coll) \in csmall\ P'\ \sigma$
**and** $\bigwedge P\ P'\ coll1\ coll2.\ R\ P\ P'\ (combine\ coll1\ coll2) = (R\ P\ P'\ coll1 \wedge R\ P\ P'\ coll2)$
**and** *Qpres*: $\bigwedge P\ \sigma\ \sigma'.\ Q\ \sigma \Longrightarrow \sigma' \in small\ P\ \sigma \Longrightarrow Q\ \sigma'$
**shows** $(\sigma',\ coll) \in cbig\ P\ \sigma \Longrightarrow R\ P\ P'\ coll \Longrightarrow Q\ \sigma \Longrightarrow (\sigma',\ coll) \in cbig\ P'\ \sigma$
⟨*proof*⟩

**lemma** *cbig-stepD*:
**assumes** *cbig*: $(\sigma',coll') \in cbig\ P\ \sigma$ **and** *nend*: $\sigma \notin endset$
**shows** $\exists \sigma 1\ coll1.\ (\sigma 1,\ coll1) \in csmall\ P\ \sigma$
$\wedge\ (\exists\, coll.\ coll' = combine\ coll1\ coll \wedge (\sigma',coll) \in cbig\ P\ \sigma 1)$
⟨*proof*⟩

**lemma** *csmall-nstep-det-last-eq*:
**assumes** *det*: $\forall \sigma.\ small\ P\ \sigma = \{\} \vee (\exists \sigma'.\ small\ P\ \sigma = \{\sigma'\})$
**shows** $[\![\ (\sigma',coll') \in cbig\ P\ \sigma;\ (\sigma',coll') \in csmall\text{-}nstep\ P\ \sigma\ n;\ (\sigma',coll'') \in csmall\text{-}nstep\ P\ \sigma\ n'\ ]\!]$
$\Longrightarrow n = n'$
⟨*proof*⟩

**end** — CollectionSemantics

**end**

# 5 Collection-based RTS

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

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

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

**locale** *CollectionBasedRTS = CollectionBasedRTS-base* **where**
*small = small* :: *'prog ⇒ 'state ⇒ 'state set* **and**
*collect = collect* :: *'prog ⇒ 'state ⇒ 'state ⇒ 'coll* **and**
*out = out* :: *'prog ⇒ 'test ⇒ ('state × 'coll) set*
**for** *small collect out*
+
**fixes**
*make-test-prog* :: *'prog ⇒ 'test ⇒ 'prog* **and**
*collect-start* :: *'prog ⇒ 'coll*
**assumes**
*out-cbig*:
*∃ i. out P t = {(σ',coll'). ∃ coll. (σ',coll) ∈ cbig (make-test-prog P t) i*
*∧ coll' = combine coll (collect-start P) }*

**context** *CollectionBasedRTS* **begin**

**end** — CollectionBasedRTS

**end**

# 6 Instantiating *Semantics* with Jinja JVM

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

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

**lemma** *JVMsmall-prealloc-pres*:
**assumes** *pre*: *preallocated (fst(snd σ))*
**and** *σ' ∈ JVMsmall P σ*
**shows** *preallocated (fst(snd σ'))*
⟨*proof*⟩

**lemma** *JVMsmall-det*: *JVMsmall P σ = {} ∨ (∃ σ'. JVMsmall P σ = {σ'})*
⟨*proof*⟩

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

**lemma** *JVMendset-final*: *σ ∈ JVMendset ⟹ ∀ P. JVMsmall P σ = {}*
⟨*proof*⟩

**lemma** *start-state-nend*:
*start-state P ∉ JVMendset*
⟨*proof*⟩

**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*: $[\![ P \vdash C \preceq^* C';\ C \neq C' ]\!]$
$\Longrightarrow \exists D'\ fs\ ms.\ class\ P\ C = \lfloor (D', fs, ms) \rfloor$
⟨*proof*⟩

**lemma** *class-subcls1*:
$[\![ class\ P\ y = class\ P'\ y;\ P \vdash y \prec^1 z ]\!] \Longrightarrow P' \vdash y \prec^1 z$
⟨*proof*⟩

**lemma** *subcls1-single-valued*: *single-valued* (*subcls1* $P$)
⟨*proof*⟩

**lemma** *subcls-confluent*:
$[\![ P \vdash C \preceq^* C';\ P \vdash C \preceq^* C'' ]\!] \Longrightarrow P \vdash C' \preceq^* C'' \vee P \vdash C'' \preceq^* C'$
⟨*proof*⟩

**lemma** *subcls1-confluent*: $[\![ P \vdash a \prec^1 b;\ P \vdash a \preceq^* c;\ a \neq c ]\!] \Longrightarrow P \vdash b \preceq^* c$
⟨*proof*⟩

**lemma** *subcls-self-superclass*: $[\![ P \vdash C \prec^1 C;\ P \vdash C \preceq^* D ]\!] \Longrightarrow D = C$
⟨*proof*⟩

**lemma** *subcls-of-Obj-acyclic*:
$[\![ P \vdash C \preceq^* Object;\ C \neq D ]\!] \Longrightarrow \neg(P \vdash C \preceq^* D \wedge P \vdash D \preceq^* C)$
⟨*proof*⟩

**lemma** *subcls-of-Obj*: $[\![ P \vdash C \preceq^* Object;\ P \vdash C \preceq^* D ]\!] \Longrightarrow P \vdash D \preceq^* Object$
⟨*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* ⇒ *cname* ⇒ *cname set* **where**
*classes-above P c* ≡ { *cn*. *P* ⊢ *c* ≼* *cn* }

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

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

**lemma** *classes-above-def2*:
*P* ⊢ *C* ≺¹ *D* ⟹ *classes-above P C* = {*C*} ∪ *classes-above P D*
⟨*proof*⟩

**lemma** *classes-above-class*:
⟦ *classes-above P C* ∩ *classes-changed P P'* = {}; *P* ⊢ *C* ≼* *C'* ⟧
⟹ *class P C'* = *class P' C'*
⟨*proof*⟩

**lemma** *classes-above-subset*:
**assumes** *classes-above P C* ∩ *classes-changed P P'* = {}
**shows** *classes-above P C* ⊆ *classes-above P' C*
⟨*proof*⟩

**lemma** *classes-above-subcls*:
⟦ *classes-above P C* ∩ *classes-changed P P'* = {}; *P* ⊢ *C* ≼* *C'* ⟧
⟹ *P'* ⊢ *C* ≼* *C'*
⟨*proof*⟩

**lemma** *classes-above-subset2*:
**assumes** *classes-above P C* ∩ *classes-changed P P'* = {}
**shows** *classes-above P' C* ⊆ *classes-above P C*
⟨*proof*⟩

**lemma** *classes-above-subcls2*:
⟦ *classes-above P C* ∩ *classes-changed P P'* = {}; *P'* ⊢ *C* ≼* *C'* ⟧
⟹ *P* ⊢ *C* ≼* *C'*

⟨*proof*⟩

**lemma** *classes-above-set*:
⟦ *classes-above P C* ∩ *classes-changed P P'* = {} ⟧
⟹ *classes-above P C* = *classes-above P' C*
⟨*proof*⟩

**lemma** *classes-above-classes-changed-sym*:
**assumes** *classes-above P C* ∩ *classes-changed P P'* = {}
**shows** *classes-above P' C* ∩ *classes-changed P' P* = {}
⟨*proof*⟩

**lemma** *classes-above-sub-classes-between-eq*:
*P* ⊢ *C* ≼* *D* ⟹ *classes-above P C* = (*classes-between P C D* − {*D*}) ∪ *classes-above P D*
⟨*proof*⟩

**lemma** *classes-above-subcls-subset*:
⟦ *P* ⊢ *C* ≼* *C'* ⟧ ⟹ *classes-above P C'* ⊆ *classes-above P C*
⟨*proof*⟩

## 9.1 Methods

**lemma** *classes-above-sees-methods*:
**assumes** *int*: *classes-above P C* ∩ *classes-changed P P'* = {} **and** *ms*: *P* ⊢ *C sees-methods Mm*
**shows** *P'* ⊢ *C sees-methods Mm*
⟨*proof*⟩

**lemma** *classes-above-sees-method*:
⟦ *classes-above P C* ∩ *classes-changed P P'* = {};
*P* ⊢ *C sees M*,*b*: *Ts*→*T* = *m in C'* ⟧
⟹ *P'* ⊢ *C sees M*,*b*: *Ts*→*T* = *m in C'*
⟨*proof*⟩

**lemma** *classes-above-sees-method2*:
⟦ *classes-above P C* ∩ *classes-changed P P'* = {};
*P'* ⊢ *C sees M*,*b*: *Ts*→*T* = *m in C'* ⟧
⟹ *P* ⊢ *C sees M*,*b*: *Ts*→*T* = *m in C'*
⟨*proof*⟩

**lemma** *classes-above-method*:
**assumes** *classes-above P C* ∩ *classes-changed P P'* = {}
**shows** *method P C M* = *method P' C M*
⟨*proof*⟩

## 9.2 Fields

**lemma** *classes-above-has-fields*:

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

**lemma** *classes-above-has-fields-dne*:
**assumes** *classes-above P C* ∩ *classes-changed P P′* = {}
**shows** (∀ *FDTs*. ¬ *P* ⊢ *C has-fields FDTs*) = (∀ *FDTs*. ¬ *P′* ⊢ *C has-fields FDTs*)
⟨*proof*⟩

**lemma** *classes-above-has-field*:
⟦ *classes-above P C* ∩ *classes-changed P P′* = {};
*P* ⊢ *C has F*,*b*:*t in C′* ⟧
⟹ *P′* ⊢ *C has F*,*b*:*t in C′*
⟨*proof*⟩

**lemma** *classes-above-has-field2*:
⟦ *classes-above P C* ∩ *classes-changed P P′* = {};
*P′* ⊢ *C has F*,*b*:*t in C′* ⟧
⟹ *P* ⊢ *C has F*,*b*:*t in C′*
⟨*proof*⟩

**lemma** *classes-above-sees-field*:
⟦ *classes-above P C* ∩ *classes-changed P P′* = {};
*P* ⊢ *C sees F*,*b*:*t in C′* ⟧
⟹ *P′* ⊢ *C sees F*,*b*:*t in C′*
⟨*proof*⟩

**lemma** *classes-above-sees-field2*:
⟦ *classes-above P C* ∩ *classes-changed P P′* = {};
*P′* ⊢ *C sees F*,*b*:*t in C′* ⟧
⟹ *P* ⊢ *C sees F*,*b*:*t in C′*
⟨*proof*⟩

**lemma** *classes-above-field*:
**assumes** *classes-above P C* ∩ *classes-changed P P′* = {}
**shows** *field P C F* = *field P′ C F*
⟨*proof*⟩

**lemma** *classes-above-fields*:
**assumes** *classes-above P C* ∩ *classes-changed P P′* = {}
**shows** *fields P C* = *fields P′ C*
⟨*proof*⟩

**lemma** *classes-above-ifields*:
⟦ *classes-above P C* ∩ *classes-changed P P′* = {} ⟧
⟹
*ifields P C* = *ifields P′ C*
⟨*proof*⟩

**lemma** *classes-above-blank*:
⟦ *classes-above P C* ∩ *classes-changed P P′* = {} ⟧
⟹
*blank P C* = *blank P′ C*
⟨*proof*⟩

**lemma** *classes-above-isfields*:
⟦ *classes-above P C* ∩ *classes-changed P P′* = {} ⟧
⟹
*isfields P C* = *isfields P′ C*
⟨*proof*⟩

**lemma** *classes-above-sblank*:
⟦ *classes-above P C* ∩ *classes-changed P P′* = {} ⟧
⟹
*sblank P C* = *sblank P′ C*
⟨*proof*⟩

### 9.3 Other

**lemma** *classes-above-start-heap*:
**assumes** *classes-above-xcpts P* ∩ *classes-changed P P′* = {}
**shows** *start-heap P* = *start-heap P′*
⟨*proof*⟩

**end**

## 10 Instantiating *CollectionSemantics* with Jinja JVM

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

**begin**

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

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

### 10.1 JVM-specific *classes-above* theory

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

**lemma** *classes-above-start-state*:
**assumes** *above-xcpts*: *classes-above-xcpts P* ∩ *classes-changed P P'* = {}
**shows** *start-state P* = *start-state P'*
⟨*proof*⟩

**lemma** *classes-above-matches-ex-entry*:
*classes-above P C* ∩ *classes-changed P P'* = {}
⟹ *matches-ex-entry P C pc xcp* = *matches-ex-entry P' C pc xcp*
⟨*proof*⟩

**lemma** *classes-above-match-ex-table*:
**assumes** *classes-above P C* ∩ *classes-changed P P'* = {}
**shows** *match-ex-table P C pc es* = *match-ex-table P' C pc es*
⟨*proof*⟩

**lemma** *classes-above-find-handler*:
**assumes** *classes-above P* (*cname-of h a*) ∩ *classes-changed P P'* = {}
**shows** *classes-above-frames P frs* ∩ *classes-changed P P'* = {}
⟹ *find-handler P a h frs sh* = *find-handler P' a h frs sh*
⟨*proof*⟩

**lemma** *find-handler-classes-above-frames*:
*find-handler P a h frs sh* = (*xp'*,*h'*,*frs'*,*sh'*)
⟹ *classes-above-frames P frs'* ⊆ *classes-above-frames P frs*
⟨*proof*⟩

**lemma** *find-handler-pieces*:
*find-handler P a h frs sh* = (*xp'*,*h'*,*frs'*,*sh'*)
⟹ *h* = *h'* ∧ *sh* = *sh'* ∧ *classes-above-frames P frs'* ⊆ *classes-above-frames P frs*
⟨*proof*⟩

## 10.2 Naive RTS algorithm

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

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

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

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

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

**interpretation** *JVMNaiveCollectionSemantics*:
*CollectionSemantics JVMsmall JVMendset JVMNaiveCollect JVMcombine JVM-collect-id*
⟨*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)}) |*

*JVMinstr-scollect P - = {}*

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

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

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

**interpretation** *JVMSmartCollectionSemantics*:
*CollectionSemantics JVMsmall JVMendset JVMSmartCollect JVMcombine JVM-collect-id*
⟨*proof*⟩

## 10.4 A few lemmas using the instantiations

**lemma** *JVMnaive-csmallD*:
*(σ', cset) ∈ JVMNaiveCollectionSemantics.csmall P σ*
*⟹ JVMexec-ncollect P σ = cset ∧ σ' ∈ JVMsmall P σ*
⟨*proof*⟩

**lemma** *JVMsmart-csmallD*:
*(σ', cset) ∈ JVMSmartCollectionSemantics.csmall P σ*
*⟹ JVMexec-scollect P σ = cset ∧ σ' ∈ JVMsmall P σ*
⟨*proof*⟩

**lemma** *jvm-naive-to-smart-csmall-nstep-last-eq*:
*⟦ (σ',cset$_n$) ∈ JVMNaiveCollectionSemantics.cbig P σ;*
*(σ',cset$_n$) ∈ JVMNaiveCollectionSemantics.csmall-nstep P σ n;*
*(σ',cset$_s$) ∈ JVMSmartCollectionSemantics.csmall-nstep P σ n' ⟧*
*⟹ n = n'*
⟨*proof*⟩

**end**

# 11 Inductive JVM execution

**theory** *JVMExecStepInductive*
**imports** *JinjaDCI.JVMExec*
**begin**

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

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

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

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

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

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

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

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

| *exec-step-ind-New-Init*:
⟦ ∀ *obj*. *sh C* ≠ *Some*(*obj*, *Done*); ∀ *Cs*. *ics* ≠ *Called Cs* ⟧

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

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

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

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

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

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

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

| *exec-step-ind-Getstatic-Called*:
⟦ (*D*′,*b*,*t*) = *field P D F*; *P* ⊢ *C has F*,*Static*:*t in D*;
*v* = *the* ((*fst*(*the*(*sh D*′))) *F*) ⟧
$\Longrightarrow$ *exec-step-ind* (*StepI* (*Getstatic C F D*)) *P h stk loc* $C_0$ $M_0$ *pc* (*Called Cs*) *frs sh*
(*None*, *h*, (*v*#*stk*, *loc*, $C_0$, $M_0$, *Suc pc*, *No-ics*)#*frs*, *sh*)

| *exec-step-ind-Getstatic-Done*:
⟦ (*D*′,*b*,*t*) = *field P D F*; *P* ⊢ *C has F*,*Static*:*t in D*;
∀ *Cs*. *ics* ≠ *Called Cs*; *sh D*′ = *Some*(*sfs*,*Done*);
*v* = *the* (*sfs F*) ⟧
$\Longrightarrow$ *exec-step-ind* (*StepI* (*Getstatic C F D*)) *P h stk loc* $C_0$ $M_0$ *pc ics frs sh*

(*None*, *h*, (*v*#*stk*, *loc*, $C_0$, $M_0$, *Suc pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-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* $C_0$ $M_0$ *pc ics frs sh*
(*None*, *h*, (*stk*, *loc*, $C_0$, $M_0$, *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* $C_0$ $M_0$ *pc ics frs sh*
(⌊*addr-of-sys-xcpt NullPointer*⌋, *h*, (*stk*, *loc*, $C_0$, $M_0$, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-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* $C_0$ $M_0$ *pc ics frs sh*
(⌊*addr-of-sys-xcpt NoSuchFieldError*⌋, *h*, (*stk*, *loc*, $C_0$, $M_0$, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-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* $C_0$ $M_0$ *pc ics frs sh*
(⌊*addr-of-sys-xcpt IncompatibleClassChangeError*⌋, *h*, (*stk*, *loc*, $C_0$, $M_0$, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-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* $C_0$ $M_0$ *pc ics frs sh*
(*None*, *h*(*a* ↦ (*D*, *fs*((*F*,*C*) ↦ *v*))), (*tl* (*tl stk*), *loc*, $C_0$, $M_0$, *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* $C_0$ $M_0$ *pc ics frs sh*
(⌊*addr-of-sys-xcpt NoSuchFieldError*⌋, *h*, (*stk*, *loc*, $C_0$, $M_0$, *pc*, *ics*)#*frs*, *sh*)

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

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

*stk*)), *i*)))

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

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

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

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

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

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

| *exec-step-ind-Invoke-Static*:
⟦ *r* = *stk*!*n*; *C* = *fst*(*the*(*h*(*the-Addr r*)));
(*D*,*b*,*Ts*,*T*,*mxs*,$mxl_0$,*ins*,*xt*)= *method P C M*; *r* ≠ *Null*;
*P* ⊢ *C sees M*,*Static*:*Ts* → *T* = *m in D* ⟧
⟹ *exec-step-ind* (*StepI* (*Invoke M n*)) *P h stk loc* $C_0$ $M_0$ *pc ics frs sh*
(⌊*addr-of-sys-xcpt IncompatibleClassChangeError*⌋, *h*, (*stk*, *loc*, $C_0$, $M_0$, *pc*, *ics*)#*frs*, *sh*)

| *exec-step-ind-Invoke*:
⟦ *ps* = *take n stk*; *r* = *stk*!*n*; *C* = *fst*(*the*(*h*(*the-Addr r*)));
(*D*,*b*,*Ts*,*T*,*mxs*,$mxl_0$,*ins*,*xt*)= *method P C M*; *r* ≠ *Null*;
*P* ⊢ *C sees M*,*NonStatic*:*Ts* → *T* = *m in D*;
*f'* = ([],[*r*]@(*rev ps*)@(*replicate* $mxl_0$ *undefined*),*D*,*M*,0,*No-ics*) ⟧

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$(\lfloor$*the-Addr* (*hd stk*)$\rfloor$, *h*, (*stk*, *loc*, $C_0$, $M_0$, *pc*, *ics*)#*frs*, *sh*)

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

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

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

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

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

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

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

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

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

(*None*, *h*, (*stk*,*loc*,$C_0$,$M_0$,*pc*,*Throwing Cs a*)#*frs*, (*sh*(*C* ↦ (*fst*(*the*(*sh C*)), *Error*))))

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

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

— Deriving *step-input* for *exec-step-ind* from *exec-step* arguments
**fun** *exec-step-input* :: [*jvm-prog*, *cname*, *mname*, *pc*, *init-call-status*] ⇒ *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*)
⟨*proof*⟩

**lemma** *exec-step-input-StepID*:
**assumes** *exec-step-input P C M pc ics = StepI i*
**shows** (*ics = Called* [] ∨ *ics = No-ics*) ∧ *instrs-of P C M* ! *pc = i*
⟨*proof*⟩

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

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

**lemma** *exec-step-ind-imp-exec-step*:
**assumes** *esi*: *exec-step-ind si P h stk loc C M pc ics frs sh* (*xp'*, *h'*, *frs'*, *sh'*)
**and** *si*: *exec-step-input P C M pc ics = si*
**shows** *exec-step P h stk loc C M pc ics frs sh* = (*xp'*, *h'*, *frs'*, *sh'*)
⟨*proof*⟩
**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'*)
⟨*proof*⟩

**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*}
⟨*proof*⟩

### 12.1 Some *classes-above* lemmas

**lemma** *start-prog-classes-above-Start*:
*classes-above* (*start-prog P C M*) *Start* = {*Object*,*Start*}
⟨*proof*⟩

**lemma** *class-add-classes-above*:

**assumes** *ns*: $\neg$ *is-class P C* **and** $\neg P \vdash D \preceq^* C$
**shows** *classes-above* (*class-add P* (*C*, *cdec*)) *D* = *classes-above P D*
⟨*proof*⟩

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

## 12.2 JVM next-step lemmas for initialization calling

**lemma** *JVM-New-next-step*:
**assumes** *step*: $\sigma' \in$ *JVMsmall P* $\sigma$
**and** *nend*: $\sigma \notin$ *JVMendset*
**and** *curr*: *curr-instr P* (*hd*(*frames-of* $\sigma$)) = *New C*
**and** *nDone*: $\neg(\exists$ *sfs i. sheap* $\sigma$ *C* = *Some*(*sfs*,*i*) $\wedge$ *i* = *Done*)
**and** *ics*: *ics-of*(*hd*(*frames-of* $\sigma$)) = *No-ics*
**shows** *ics-of* (*hd*(*frames-of* $\sigma'$)) = *Calling C* [] $\wedge$ *sheap* $\sigma$ = *sheap* $\sigma' \wedge \sigma' \notin$ *JVMendset*
⟨*proof*⟩

**lemma** *JVM-Getstatic-next-step*:
**assumes** *step*: $\sigma' \in$ *JVMsmall P* $\sigma$
**and** *nend*: $\sigma \notin$ *JVMendset*
**and** *curr*: *curr-instr P* (*hd*(*frames-of* $\sigma$)) = *Getstatic C F D*
**and** *fC*: *P* $\vdash$ *C has F*,*Static*:*t in D*
**and** *nDone*: $\neg(\exists$ *sfs i. sheap* $\sigma$ *D* = *Some*(*sfs*,*i*) $\wedge$ *i* = *Done*)
**and** *ics*: *ics-of*(*hd*(*frames-of* $\sigma$)) = *No-ics*
**shows** *ics-of* (*hd*(*frames-of* $\sigma'$)) = *Calling D* [] $\wedge$ *sheap* $\sigma$ = *sheap* $\sigma' \wedge \sigma' \notin$ *JVMendset*
⟨*proof*⟩

**lemma** *JVM-Putstatic-next-step*:
**assumes** *step*: $\sigma' \in$ *JVMsmall P* $\sigma$
**and** *nend*: $\sigma \notin$ *JVMendset*
**and** *curr*: *curr-instr P* (*hd*(*frames-of* $\sigma$)) = *Putstatic C F D*
**and** *fC*: *P* $\vdash$ *C has F*,*Static*:*t in D*
**and** *nDone*: $\neg(\exists$ *sfs i. sheap* $\sigma$ *D* = *Some*(*sfs*,*i*) $\wedge$ *i* = *Done*)
**and** *ics*: *ics-of*(*hd*(*frames-of* $\sigma$)) = *No-ics*
**shows** *ics-of* (*hd*(*frames-of* $\sigma'$)) = *Calling D* [] $\wedge$ *sheap* $\sigma$ = *sheap* $\sigma' \wedge \sigma' \notin$ *JVMendset*
⟨*proof*⟩

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

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

## 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* × *cname set*

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

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

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

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

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

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

## 12.4 Definition lemmas

**lemma** *jvm-progs-tests-nStart*:
**assumes** *P*: $P \in jvm\text{-}progs$ **and** *t*: $t \in jvm\text{-}tests$
**shows** $\neg is\text{-}class\ (t\#P)\ Start$
⟨*proof*⟩

**lemma** *jvm-make-test-prog-classes-above-xcpts*:
**assumes** *P*: $P \in jvm\text{-}progs$ **and** *t*: $t \in jvm\text{-}tests$
**shows** $classes\text{-}above\text{-}xcpts\ (jvm\text{-}make\text{-}test\text{-}prog\ P\ t) = classes\text{-}above\text{-}xcpts\ P$
⟨*proof*⟩

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

**shows** $\exists m.$ *jvm-make-test-prog P t* $\vdash$ *Test sees main, Static* : $[]\rightarrow$*Void* = *m in Test*
⟨*proof*⟩

## 12.5 Naive RTS algorithm

### 12.5.1 Definitions

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

**abbreviation** *jvm-naive-collect-start* :: *jvm-prog* ⇒ *cname set* **where**
*jvm-naive-collect-start P* ≡ {}

**lemma** *jvm-naive-out-xcpts-collected*:
**assumes** *o1* ∈ *jvm-naive-out P t*
**shows** *classes-above-xcpts* (*start-prog* (*t* # *P*) (*fst t*) *main*) ⊆ *snd o1*
⟨*proof*⟩

### 12.5.2 Naive algorithm correctness

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

**lemma** *ncollect-exec-instr*:
**assumes** *JVMinstr-ncollect P i h stk* ∩ *classes-changed P P′* = {}
**and** *above-C*: *classes-above P C* ∩ *classes-changed P P′* = {}
**and** *ics*: *ics* = *Called* [] ∨ *ics* = *No-ics*
**and** *i*: *i* = *instrs-of P C M* ! *pc*
**shows** *exec-instr i P h stk loc C M pc ics frs sh* = *exec-instr i P′ h stk loc C M pc ics frs sh*
⟨*proof*⟩
**lemma** *ncollect-JVMinstr-ncollect*:
**assumes** *JVMinstr-ncollect P i h stk* ∩ *classes-changed P P′* = {}
**shows** *JVMinstr-ncollect P i h stk* = *JVMinstr-ncollect P′ i h stk*
⟨*proof*⟩
**lemma** *ncollect-exec-step*:
**assumes** *JVMstep-ncollect P h stk C M pc ics* ∩ *classes-changed P P′* = {}
**and** *above-C*: *classes-above P C* ∩ *classes-changed P P′* = {}
**shows** *exec-step P h stk loc C M pc ics frs sh* = *exec-step P′ h stk loc C M pc ics frs sh*
⟨*proof*⟩
**lemma** *ncollect-JVMstep-ncollect*:
**assumes** *JVMstep-ncollect P h stk C M pc ics* ∩ *classes-changed P P′* = {}
**and** *above-C*: *classes-above P C* ∩ *classes-changed P P′* = {}

**shows** *JVMstep-ncollect P h stk C M pc ics = JVMstep-ncollect P′ h stk C M pc ics*
⟨*proof*⟩
**lemma** *ncollect-classes-above-frames*:
*JVMexec-ncollect P (None, h, (stk,loc,C,M,pc,ics)#frs, sh) ∩ classes-changed P P′ = {}*
*⟹ classes-above-frames P frs = classes-above-frames P′ frs*
⟨*proof*⟩
**lemma** *ncollect-classes-above-xcpts*:
**assumes** *JVMexec-ncollect P (None, h, (stk,loc,C,M,pc,ics)#frs, sh) ∩ classes-changed P P′ = {}*
**shows** *classes-above-xcpts P = classes-above-xcpts P′*
⟨*proof*⟩
**lemma** *ncollect-JVMexec-ncollect*:
**assumes** *JVMexec-ncollect P σ ∩ classes-changed P P′ = {}*
**shows** *JVMexec-ncollect P σ = JVMexec-ncollect P′ σ*
⟨*proof*⟩
**lemma** *ncollect-exec-instr-xcpts*:
**assumes** *collect*: *JVMinstr-ncollect P i h stk ∩ classes-changed P P′ = {}*
**and** *xpcollect*: *classes-above-xcpts P ∩ classes-changed P P′ = {}*
**and** *prealloc*: *preallocated h*
**and** *σ′*: *σ′ = exec-instr i P h stk loc C M pc ics′ frs sh*
**and** *xp*: *fst σ′ = Some a*
**and** *i*: *i = instrs-of P C M ! pc*
**shows** *classes-above P (cname-of h a) ∩ classes-changed P P′ = {}*
⟨*proof*⟩
**lemma** *ncollect-exec-step-xcpts*:
**assumes** *collect*: *JVMstep-ncollect P h stk C M pc ics ∩ classes-changed P P′ = {}*
**and** *xpcollect*: *classes-above-xcpts P ∩ classes-changed P P′ = {}*
**and** *prealloc*: *preallocated h*
**and** *σ′*: *σ′ = exec-step P h stk loc C M pc ics frs sh*
**and** *xp*: *fst σ′ = Some a*
**shows** *classes-above P (cname-of h a) ∩ classes-changed P P′ = {}*
⟨*proof*⟩
**lemma** *ncollect-JVMsmall*:
**assumes** *collect*: *(σ′, cset) ∈ JVMNaiveCollectionSemantics.csmall P σ*
**and** *intersect*: *cset ∩ classes-changed P P′ = {}*
**and** *prealloc*: *preallocated (fst(snd σ))*
**shows** *(σ′, cset) ∈ JVMNaiveCollectionSemantics.csmall P′ σ*
⟨*proof*⟩
**lemma** *ncollect-JVMbig*:
**assumes** *collect*: *(σ′, cset) ∈ JVMNaiveCollectionSemantics.cbig P σ*
**and** *intersect*: *cset ∩ classes-changed P P′ = {}*
**and** *prealloc*: *preallocated (fst(snd σ))*
**shows** *(σ′, cset) ∈ JVMNaiveCollectionSemantics.cbig P′ σ*
⟨*proof*⟩
**theorem** *jvm-naive-existence-safe*:
**assumes** *p*: *P ∈ jvm-progs* **and** *P′ ∈ jvm-progs* **and** *t*: *t ∈ jvm-tests*

**and** *out*: *o1* ∈ *jvm-naive-out P t* **and** *jvm-deselect P o1 P′*
**shows** ∃ *o2* ∈ *jvm-naive-out P′ t*. *o1* = *o2*
⟨*proof*⟩
**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*
⟨*proof*⟩

## 12.6 Smarter RTS algorithm

### 12.6.1 Definitions and helper lemmas

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

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

**lemma** *jvm-naive-iff-smart*:
(∃ $cset_n$. (σ′,$cset_n$) ∈ *jvm-naive-out P t*) ⟷ (∃ $cset_s$. (σ′,$cset_s$) ∈ *jvm-smart-out P t*)
⟨*proof*⟩

**lemma** *jvm-smart-out-classes-above-xcpts*:
**assumes** *s*: (σ′,$cset_s$) ∈ *jvm-smart-out P t* **and** *P*: *P* ∈ *jvm-progs* **and** *t*: *t* ∈ *jvm-tests*
**shows** *classes-above-xcpts* (*jvm-make-test-prog P t*) ⊆ $cset_s$
⟨*proof*⟩

**lemma** *jvm-smart-collect-start-make-test-prog*:
⟦ *P* ∈ *jvm-progs*; *t* ∈ *jvm-tests* ⟧
⟹ *jvm-smart-collect-start* (*jvm-make-test-prog P t*) = *jvm-smart-collect-start P*
⟨*proof*⟩

**lemma** *jvm-smart-out-classes-above-start-heap*:
**assumes** *s*: (σ′,$cset_s$) ∈ *jvm-smart-out P t* **and** *h*: *start-heap* (*t*#*P*) *a* = *Some*(*C*,*fs*)
**and** *P*: *P* ∈ *jvm-progs* **and** *t*: *t* ∈ *jvm-tests*
**shows** *classes-above* (*jvm-make-test-prog P t*) *C* ⊆ $cset_s$
⟨*proof*⟩

**lemma** *jvm-smart-out-classes-above-start-sheap*:
**assumes** (σ′,$cset_s$) ∈ *jvm-smart-out P t* **and** *start-sheap C* = *Some*(*sfs*,*i*)
**shows** *classes-above* (*jvm-make-test-prog P t*) *C* ⊆ $cset_s$

⟨*proof*⟩

**lemma** *jvm-smart-out-classes-above-frames*:
(σ',$cset_s$) ∈ *jvm-smart-out P t*
⟹ *classes-above-frames* (*jvm-make-test-prog P t*) (*frames-of* (*start-state* (*t*#*P*)))
⊆ $cset_s$
⟨*proof*⟩

### 12.6.2 Additional well-formedness conditions

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

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

**lemma** *is-ptrD*: *is-ptr h v* ⟹ *v* = *Null* ∨ (∃ *a*. *v* = *Addr a* ∧ (∃ *Cfs*. *h a* = *Some Cfs*))
⟨*proof*⟩
**fun** *stack-safe* :: *instr* ⇒ *heap* ⇒ *val list* ⇒ *bool* **where**
*stack-safe* (*Getfield F C*) *h stk* = (*length stk* > *0* ∧ *is-ptr h* (*hd stk*)) |
*stack-safe* (*Putfield F C*) *h stk* = (*length stk* > *1* ∧ *is-ptr h* (*hd* (*tl stk*))) |
*stack-safe* (*Checkcast C*) *h stk* = (*length stk* > *0* ∧ *is-ptr h* (*hd stk*)) |
*stack-safe* (*Invoke M n*) *h stk* = (*length stk* > *n* ∧ *is-ptr h* (*stk* ! *n*)) |
*stack-safe JVMInstructions.Throw h stk* = (*length stk* > *0* ∧ *is-ptr h* (*hd stk*)) |
*stack-safe i h stk* = *True*

**lemma** *well-formed-stack-safe*:
**assumes** *wtp*: *wf-jvm-prog*$_\Phi$ *P* **and** *correct*: *P*,Φ ⊢ (*xp*,*h*,(*stk*,*loc*,*C*,*M*,*pc*,*ics*)#*frs*,*sh*)√
**shows** *stack-safe* (*instrs-of P C M* ! *pc*) *h stk*
⟨*proof*⟩

### 12.6.3 Proving naive ⊆ smart

We prove that, given well-formedness of the program and state, and "promises" about what has or will be collected in previous or future steps, *jvm-smart* collects everything *jvm-naive* does. We prove that promises about previously-collected classes ("backward promises") are maintained by execution, and promises about to-be-collected classes ("forward promises") are met by the

end of execution. We then show that the required initial conditions (well-formedness and backward promises) are met by the defined start states, and thus that a run test will collect at least those classes collected by the naive algorithm.

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

**lemma** *backward-coll-promises-kept*:
**assumes**
— well-formedness
*wtp*: *wf-jvm-prog*$_\Phi$ *P*
**and** *correct*: *P*,Φ ⊢ (*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*. ∃ *a*. *h a* = *Some*(*C*,*fs*) ⟹ *classes-above P C* ⊆ *cset*
**and** *sheap*: $\bigwedge$*C sfs i*. *sh C* = *Some*(*sfs*,*i*) ⟹ *classes-above P C* ⊆ *cset*
**and** *xcpts*: *classes-above-xcpts P* ⊆ *cset*
**and** *frames*: *classes-above-frames P frs* ⊆ *cset*
— forward promises - will be collected after if not already
**and** *init-class-prom*: $\bigwedge$*C*. *ics* = *Called* [] ∨ *ics* = *No-ics*
⟹ *coll-init-class P* (*instrs-of P C' M'* ! *pc*) = *Some C* ⟹ *classes-above P C* ⊆ *cset*
**and** *Calling-prom*: $\bigwedge$*C' Cs'*. *ics* = *Calling C' Cs'* ⟹ *classes-above P C'* ⊆ *cset*
— collection and step
**and** *smart*: *JVMexec-scollect P* (*xp*,*h*,*frs*,*sh*) ⊆ *cset*
**and** *small*: (*xp'*,*h'*,*frs'*,*sh'*) ∈ *JVMsmall P* (*xp*,*h*,*frs*,*sh*)
**shows** (*h' a* = *Some*(*C*,*fs*) ⟶ *classes-above P C* ⊆ *cset*)
∧ (*sh' C* = *Some*(*sfs'*,*i'*) ⟶ *classes-above P C* ⊆ *cset*)
∧ (*classes-above-frames P frs'* ⊆ *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*: (σ', *cset'*) ∈ *JVMSmartCollectionSemantics.cbig P* σ
**and** *nend*: σ ∉ *JVMendset*
**and** *ics*: *ics-of* (*hd*(*frames-of* σ)) = *Calling Object Cs*
**shows** *classes-above P Object* ⊆ *cset* ∪ *cset'*
⟨*proof*⟩
**lemma** *Calling-None-next-state*:
**assumes** *ics*: *ics-of* (*hd*(*frames-of* σ)) = *Calling C Cs*
**and** *none*: *sheap* σ *C* = *None*
**and** *set*: ∀ *C'*. *P* ⊢ *C* ≼* *C'* ⟶ (∃ *sfs i*. *sheap* σ *C'* = *Some*(*sfs*,*i*))

$\longrightarrow$ *classes-above* $P\ C' \subseteq$ *cset*
**and** $\sigma'$: $(\sigma',$ *cset*$') \in$ *JVMSmartCollectionSemantics.csmall* $P\ \sigma$
**shows** $\sigma' \notin$ *JVMendset* $\wedge$ *ics-of* (*hd*(*frames-of* $\sigma'$)) = *Calling* $C$ *Cs*
$\wedge\ (\exists$ *sfs. sheap* $\sigma'\ C =$ *Some*(*sfs*,*Prepared*))
$\wedge\ (\forall\ C'.\ P \vdash C \preceq^* C' \longrightarrow C \neq C'$
$\longrightarrow\ (\exists$ *sfs i. sheap* $\sigma'\ C' =$ *Some*(*sfs*,*i*)) $\longrightarrow$ *classes-above* $P\ C' \subseteq$ *cset*)
⟨*proof*⟩
**lemma** *Calling-Prepared-next-state*:
**assumes** *sub*: $P \vdash C \prec^1 D$
**and** *obj*: $P \vdash D \preceq^*$ *Object*
**and** *ics*: *ics-of* (*hd*(*frames-of* $\sigma$)) = *Calling* $C$ *Cs*
**and** *prep*: *sheap* $\sigma\ C =$ *Some*(*sfs*,*Prepared*)
**and** *set*: $\forall\ C'.\ P \vdash C \preceq^* C' \longrightarrow C \neq C' \longrightarrow (\exists$ *sfs i. sheap* $\sigma\ C' =$ *Some*(*sfs*,*i*))
$\longrightarrow$ *classes-above* $P\ C' \subseteq$ *cset*
**and** $\sigma'$: $(\sigma',$ *cset*$') \in$ *JVMSmartCollectionSemantics.csmall* $P\ \sigma$
**shows** $\sigma' \notin$ *JVMendset* $\wedge$ *ics-of* (*hd* (*frames-of* $\sigma'$)) = *Calling* $D$ ($C$#*Cs*)
$\wedge\ (\forall\ C'.\ P \vdash D \preceq^* C' \longrightarrow (\exists$ *sfs i. sheap* $\sigma'\ C' =$ *Some*(*sfs*,*i*))
$\longrightarrow$ *classes-above* $P\ C' \subseteq$ *cset*)
⟨*proof*⟩
**lemma** *Calling-collects-IH*:
**assumes** *sub*: $P \vdash C \prec^1 D$
**and** *obj*: $P \vdash D \preceq^*$ *Object*
**and** *step*: $\bigwedge\sigma$ *cset*$'$ *Cs*. $(\sigma',$ *cset*$') \in$ *JVMSmartCollectionSemantics.cbig* $P\ \sigma \Longrightarrow$
$\sigma \notin$ *JVMendset*
$\Longrightarrow$ *ics-of* (*hd*(*frames-of* $\sigma$)) = *Calling* $D$ *Cs*
$\Longrightarrow \forall\ C'.\ P \vdash D \preceq^* C' \longrightarrow (\exists$ *sfs i. sheap* $\sigma\ C' =$ *Some*(*sfs*,*i*))
$\longrightarrow$ *classes-above* $P\ C' \subseteq$ *cset*
$\Longrightarrow$ *classes-above* $P\ D \subseteq$ *cset* $\cup$ *cset*$'$
**and** *big*: $(\sigma',$ *cset*$') \in$ *JVMSmartCollectionSemantics.cbig* $P\ \sigma$
**and** *nend*: $\sigma \notin$ *JVMendset*
**and** *curr*: *ics-of* (*hd*(*frames-of* $\sigma$)) = *Calling* $C$ *Cs*
**and** *set*: $\forall\ C'.\ P \vdash C \preceq^* C' \longrightarrow (\exists$ *sfs i. sheap* $\sigma\ C' =$ *Some*(*sfs*,*i*))
$\longrightarrow$ *classes-above* $P\ C' \subseteq$ *cset*
**shows** *classes-above* $P\ C \subseteq$ *cset* $\cup$ *cset*$'$
⟨*proof*⟩
**lemma** *Calling-collects*:
**assumes** *sub*: $P \vdash C \preceq^*$ *Object*
**and** $(\sigma',$ *cset*$') \in$ *JVMSmartCollectionSemantics.cbig* $P\ \sigma$
**and** $\sigma \notin$ *JVMendset*
**and** *ics-of* (*hd*(*frames-of* $\sigma$)) = *Calling* $C$ *Cs*
**and** $\forall\ C'.\ P \vdash C \preceq^* C' \longrightarrow (\exists$ *sfs i. sheap* $\sigma\ C' =$ *Some*(*sfs*,*i*))
$\longrightarrow$ *classes-above* $P\ C' \subseteq$ *cset*
**and** *cset*$' \subseteq$ *cset*
**shows** *classes-above* $P\ C \subseteq$ *cset*
⟨*proof*⟩

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

**lemma** *New-collects*:

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

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

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

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

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

**lemma** *jvm-smart-out-classes-above-Test*:
**assumes** *s*: $(\sigma',cset_s) \in$ *jvm-smart-out P t* **and** *P*: *P* ∈ *jvm-progs* **and** *t*: *t* ∈ *jvm-tests*
**shows** *classes-above* (*jvm-make-test-prog P t*) *Test* ⊆ $cset_s$
(**is** *classes-above ?P ?D* ⊆ *?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-jvm-prog*$_\Phi$ *P*
**and** *correct*: *P*,Φ ⊢ (*xp*,*h*,*frs*,*sh*)$\surd$
— defs
**and** *f'*: *hd frs* = (*stk*,*loc*,*C'*,*M'*,*pc*,*ics*)
— backward promises - will be collected prior
**and** *heap*: ⋀*C fs*. ∃ *a*. *h a* = *Some*(*C*,*fs*) ⟹ *classes-above P C* ⊆ *cset*
**and** *sheap*: ⋀*C sfs i*. *sh C* = *Some*(*sfs*,*i*) ⟹ *classes-above P C* ⊆ *cset*
**and** *xcpts*: *classes-above-xcpts P* ⊆ *cset*
**and** *frames*: *classes-above-frames P frs* ⊆ *cset*
— forward promises - will be collected after if not already
**and** *init-class-prom*: ⋀*C*. *ics* = *Called* [] ∨ *ics* = *No-ics*
⟹ *coll-init-class P* (*instrs-of P C' M'* ! *pc*) = *Some C* ⟹ *classes-above P C* ⊆ *cset*
**and** *Calling-prom*: ⋀*C' Cs'*. *ics* = *Calling C' Cs'* ⟹ *classes-above P C'* ⊆ *cset*
— collection
**and** *smart*: *JVMexec-scollect P* (*xp*,*h*,*frs*,*sh*) ⊆ *cset*
**shows** *JVMexec-ncollect P* (*xp*,*h*,*frs*,*sh*) ⊆ *cset*
⟨*proof*⟩
**lemma** *jvm-naive-to-smart-csmall*:
**assumes**
— well-formedness
*wtp*: *wf-jvm-prog*$_\Phi$ *P*
**and** *correct*: *P*,Φ ⊢ (*xp*,*h*,*frs*,*sh*)$\surd$
— defs
**and** *f'*: *hd frs* = (*stk*,*loc*,*C'*,*M'*,*pc*,*ics*)
— backward promises - will be collected prior
**and** *heap*: ⋀*C fs*. ∃ *a*. *h a* = *Some*(*C*,*fs*) ⟹ *classes-above P C* ⊆ *cset*
**and** *sheap*: ⋀*C sfs i*. *sh C* = *Some*(*sfs*,*i*) ⟹ *classes-above P C* ⊆ *cset*
**and** *xcpts*: *classes-above-xcpts P* ⊆ *cset*
**and** *frames*: *classes-above-frames P frs* ⊆ *cset*
— forward promises - will be collected after if not already
**and** *init-class-prom*: ⋀*C*. *ics* = *Called* [] ∨ *ics* = *No-ics*
⟹ *coll-init-class P* (*instrs-of P C' M'* ! *pc*) = *Some C* ⟹ *classes-above P C* ⊆ *cset*

**and** *Calling-prom*: $\bigwedge C'\ Cs'.\ ics = Calling\ C'\ Cs' \Longrightarrow classes\text{-}above\ P\ C' \subseteq cset$
— collections
**and** *smart-coll*: $(\sigma', cset_s) \in JVMSmartCollectionSemantics.csmall\ P\ (xp,h,frs,sh)$
**and** *naive-coll*: $(\sigma', cset_n) \in JVMNaiveCollectionSemantics.csmall\ P\ (xp,h,frs,sh)$
**and** *smart*: $cset_s \subseteq cset$
**shows** $cset_n \subseteq cset$
⟨*proof*⟩
**lemma** *jvm-naive-to-smart-csmall-nstep*:
$[\![$ *wf-jvm-prog*$_\Phi$ $P$;
$P,\Phi \vdash (xp,h,frs,sh)\surd$;
$hd\ frs = (stk,loc,C',M',pc,ics)$;
$\bigwedge C\ fs.\ \exists a.\ h\ a = Some(C,fs) \Longrightarrow classes\text{-}above\ P\ C \subseteq cset$;
$\bigwedge C\ sfs\ i.\ sh\ C = Some(sfs,i) \Longrightarrow classes\text{-}above\ P\ C \subseteq cset$;
$classes\text{-}above\text{-}xcpts\ P \subseteq cset$;
$classes\text{-}above\text{-}frames\ P\ frs \subseteq cset$;
$\bigwedge C.\ ics = Called\ [] \vee ics = No\text{-}ics$
$\Longrightarrow coll\text{-}init\text{-}class\ P\ (instrs\text{-}of\ P\ C'\ M'\ !\ pc) = Some\ C \Longrightarrow classes\text{-}above\ P\ C \subseteq cset$;
$\bigwedge C'\ Cs'.\ ics = Calling\ C'\ Cs' \Longrightarrow classes\text{-}above\ P\ C' \subseteq cset$;
$(\sigma', cset_n) \in JVMNaiveCollectionSemantics.csmall\text{-}nstep\ P\ (xp,h,frs,sh)\ n$;
$(\sigma', cset_s) \in JVMSmartCollectionSemantics.csmall\text{-}nstep\ P\ (xp,h,frs,sh)\ n$;
$cset_s \subseteq cset$;
$\sigma' \in JVMendset\ ]\!]$
$\Longrightarrow cset_n \subseteq cset$
⟨*proof*⟩
**lemma** *jvm-naive-to-smart-cbig*:
**assumes**
— well-formedness
*wtp*: *wf-jvm-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/maintained prior
**and** *heap*: $\bigwedge C\ fs.\ \exists a.\ h\ a = Some(C,fs) \Longrightarrow classes\text{-}above\ P\ C \subseteq cset$
**and** *sheap*: $\bigwedge C\ sfs\ i.\ sh\ C = Some(sfs,i) \Longrightarrow classes\text{-}above\ P\ C \subseteq cset$
**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*: $\bigwedge C.\ ics = Called\ [] \vee ics = No\text{-}ics$
$\Longrightarrow coll\text{-}init\text{-}class\ P\ (instrs\text{-}of\ P\ C'\ M'\ !\ pc) = Some\ C \Longrightarrow classes\text{-}above\ P\ C \subseteq cset$
**and** *Calling-prom*: $\bigwedge C'\ Cs'.\ ics = Calling\ C'\ Cs' \Longrightarrow classes\text{-}above\ P\ C' \subseteq cset$
— collections
**and** *n*: $(\sigma', cset_n) \in JVMNaiveCollectionSemantics.cbig\ P\ (xp,h,frs,sh)$
**and** *s*: $(\sigma', cset_s) \in JVMSmartCollectionSemantics.cbig\ P\ (xp,h,frs,sh)$
**and** *smart*: $cset_s \subseteq cset$
**shows** $cset_n \subseteq cset$
⟨*proof*⟩
**lemma** *jvm-naive-to-smart-collection*:

**assumes** *naive*: $(\sigma', cset_n) \in$ *jvm-naive-out* $P\ t$ **and** *smart*: $(\sigma', cset_s) \in$ *jvm-smart-out* $P\ t$
**and** $P$: $P \in$ *jvm-progs* **and** $t$: $t \in$ *jvm-tests*
**shows** $cset_n \subseteq cset_s$
⟨*proof*⟩

### 12.6.4 Proving smart ⊆ naive

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

**lemma** *jvm-smart-to-naive-exec-collect*:
*JVMexec-scollect* $P\ \sigma \subseteq$ *JVMexec-ncollect* $P\ \sigma$
⟨*proof*⟩

**lemma** *jvm-smart-to-naive-csmall*:
**assumes** $(\sigma',\ cset_n) \in$ *JVMNaiveCollectionSemantics.csmall* $P\ \sigma$
**and** $(\sigma',\ cset_s) \in$ *JVMSmartCollectionSemantics.csmall* $P\ \sigma$
**shows** $cset_s \subseteq cset_n$
⟨*proof*⟩

**lemma** *jvm-smart-to-naive-csmall-nstep*:
$[\![\ (\sigma',\ cset_n) \in$ *JVMNaiveCollectionSemantics.csmall-nstep* $P\ \sigma\ n$;
$(\sigma',\ cset_s) \in$ *JVMSmartCollectionSemantics.csmall-nstep* $P\ \sigma\ n\ ]\!]$
$\Longrightarrow cset_s \subseteq cset_n$
⟨*proof*⟩

**lemma** *jvm-smart-to-naive-cbig*:
**assumes** $n$: $(\sigma',\ cset_n) \in$ *JVMNaiveCollectionSemantics.cbig* $P\ \sigma$
**and** $s$: $(\sigma',\ cset_s) \in$ *JVMSmartCollectionSemantics.cbig* $P\ \sigma$
**shows** $cset_s \subseteq cset_n$
⟨*proof*⟩

**lemma** *jvm-smart-to-naive-collection*:
**assumes** *naive*: $(\sigma', cset_n) \in$ *jvm-naive-out* $P\ t$ **and** *smart*: $(\sigma', cset_s) \in$ *jvm-smart-out* $P\ t$
**and** $P \in$ *jvm-progs* **and** $t \in$ *jvm-tests*
**shows** $cset_s \subseteq cset_n$
⟨*proof*⟩

### 12.6.5 Safety of the smart algorithm

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

**lemma** *jvm-naive-eq-smart-collection*:
**assumes** *naive*: $(\sigma', cset_n) \in$ *jvm-naive-out* $P\ t$ **and** *smart*: $(\sigma', cset_s) \in$ *jvm-smart-out* $P\ t$
**and** $P \in$ *jvm-progs* **and** $t \in$ *jvm-tests*
**shows** $cset_n = cset_s$

⟨*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*: $P \in$ *jvm-progs* **and** $P'$: $P' \in$ *jvm-progs* **and** *t*: $t \in$ *jvm-tests*
**and** *out*: *o1* $\in$ *jvm-smart-out P t* **and** *dss*: *jvm-deselect P o1 P'*
**shows** $\exists$ *o2* $\in$ *jvm-smart-out P' t*. *o1* = *o2*
⟨*proof*⟩

...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.