

JinjaDCI: a Java semantics with dynamic class initialization

Susannah Mansky

February 6, 2026

Abstract. This work is an extension of the Jinja semantics for Java and the JVM by Klein and Nipkow to include static fields and methods and dynamic class initialization. In Java, class initialization methods are run dynamically, called when classes are first used. Such calls are handled by the running of an initialization procedure, which interrupts execution and determines which initialization methods must be run before execution continues. This interrupting is modeled here in a couple of ways. In the Java semantics, evaluation is performed via expressions that are manipulated through evaluation until a final value is reached. In JinjaDCI, we have added two types of initialization expressions whose evaluations produce the steps of the initialization procedure. These expressions can occur during evaluation and store the calling expression away to continue being evaluated once the procedure is complete. In the JVM semantics, since programs are static sequences of instructions, the initialization procedure is run instead by the execution function. This function performs steps of the procedure rather than calling instructions when the initialization procedure has been called.

This extension includes the necessary updates to all major proofs from the original Jinja, including type safety and correctness of compilation from the Java semantics to the JVM semantics.

This work is partially described in [1].

Contents

1	Jinja Source Language	5
1.1	Auxiliary Definitions	5
1.2	Jinja types	7
1.3	Class Declarations and Programs	8
1.4	Relations between Jinja Types	9
1.5	Jinja Values	16
1.6	Objects and the Heap	16
1.7	Exceptions	19
1.8	Expressions	22
1.9	Well-typedness of Jinja expressions	31
1.10	Runtime Well-typedness	34
1.11	Program State	37
1.12	System Classes	37
1.13	Generic Well-formedness of programs	38
1.14	Weak well-formedness of Jinja programs	43
1.15	Big Step Semantics	43
1.16	Definite assignment	52
1.17	Conformance Relations for Type Soundness Proofs	54
1.18	Small Step Semantics	56
1.19	Expression conformance properties	65
1.20	Progress of Small Step Semantics	71
1.21	Well-formedness Constraints	73
1.22	Type Safety Proof	74
1.23	Equivalence of Big Step and Small Step Semantics	78
1.24	Program annotation	110
2	Jinja Virtual Machine	113
2.1	State of the JVM	113
2.2	Instructions of the JVM	114
2.3	Exception handling in the JVM	115
2.4	Program Execution in the JVM	117
2.5	Program Execution in the JVM in full small step style	124
2.6	A Defensive JVM	128
2.7	The Jinja Type System as a Semilattice	131
2.8	The JVM Type System as Semilattice	133
2.9	Effect of Instructions on the State Type	135
2.10	Monotonicity of eff and app	142

2.11	The Bytecode Verifier	142
2.12	The Typing Framework for the JVM	144
2.13	Kildall for the JVM	146
2.14	LBV for the JVM	148
2.15	BV Type Safety Invariant	149
2.16	Property preservation under <i>class-add</i>	156
2.17	Properties and types of the starting program	169
2.18	BV Type Safety Proof	173
2.19	Welltyped Programs produce no Type Errors	181
3	Compilation	185
3.1	An Intermediate Language	185
3.2	Well-Formedness of Intermediate Language	194
3.3	Program Compilation	200
3.4	Compilation Stage 1	203
3.5	Correctness of Stage 1	204
3.6	Compilation Stage 2	206
3.7	Correctness of Stage 2	209
3.8	Combining Stages 1 and 2	229
3.9	Preservation of Well-Typedness	229

Chapter 1

Jinja Source Language

1.1 Auxiliary Definitions

theory *Auxiliary* imports *Main* begin

lemma *nat-add-max-le[simp]*:
 $((n::nat) + \max i j \leq m) = (n + i \leq m \wedge n + j \leq m)$

lemma *Suc-add-max-le[simp]*:
 $(\text{Suc}(n + \max i j) \leq m) = (\text{Suc}(n + i) \leq m \wedge \text{Suc}(n + j) \leq m)$

notation *Some* ($\langle\langle[-]\rangle\rangle$)

1.1.1 *distinct-fst*

definition *distinct-fst* :: $('a \times 'b) \text{ list} \Rightarrow \text{bool}$

where

$\text{distinct-fst} \equiv \text{distinct} \circ \text{map fst}$

lemma *distinct-fst-Nil [simp]*:
 $\text{distinct-fst } []$

lemma *distinct-fst-Cons [simp]*:
 $\text{distinct-fst } ((k,x)\#kxs) = (\text{distinct-fst } kxs \wedge (\forall y. (k,y) \notin \text{set } kxs))$

lemma *distinct-fst-appendD*:
 $\text{distinct-fst}(kxs @ kxs') \Longrightarrow \text{distinct-fst } kxs \wedge \text{distinct-fst } kxs'$

lemma *map-of-SomeI*:
 $\llbracket \text{distinct-fst } kxs; (k,x) \in \text{set } kxs \rrbracket \Longrightarrow \text{map-of } kxs k = \text{Some } x$

1.1.2 Using *list-all2* for relations

definition *fun-of* :: $('a \times 'b) \text{ set} \Rightarrow 'a \Rightarrow 'b \Rightarrow \text{bool}$

where

$\text{fun-of } S \equiv \lambda x y. (x,y) \in S$

Convenience lemmas

lemma *rel-list-all2-Cons [iff]*:
 $\text{list-all2 } (\text{fun-of } S) (x\#xs) (y\#ys) =$
 $((x,y) \in S \wedge \text{list-all2 } (\text{fun-of } S) xs ys)$

lemma *rel-list-all2-Cons1*:

$$\begin{aligned} & \text{list-all2 (fun-of } S) (x\#xs) ys = \\ & (\exists z zs. ys = z\#zs \wedge (x,z) \in S \wedge \text{list-all2 (fun-of } S) xs zs) \end{aligned}$$

lemma *rel-list-all2-Cons2*:

$$\begin{aligned} & \text{list-all2 (fun-of } S) xs (y\#ys) = \\ & (\exists z zs. xs = z\#zs \wedge (z,y) \in S \wedge \text{list-all2 (fun-of } S) zs ys) \end{aligned}$$

lemma *rel-list-all2-refl*:

$$(\bigwedge x. (x,x) \in S) \implies \text{list-all2 (fun-of } S) xs xs$$

lemma *rel-list-all2-antisym*:

$$\begin{aligned} & \llbracket (\bigwedge x y. \llbracket (x,y) \in S; (y,x) \in T \rrbracket \implies x = y); \\ & \text{list-all2 (fun-of } S) xs ys; \text{list-all2 (fun-of } T) ys xs \rrbracket \implies xs = ys \end{aligned}$$

lemma *rel-list-all2-trans*:

$$\begin{aligned} & \llbracket \bigwedge a b c. \llbracket (a,b) \in R; (b,c) \in S \rrbracket \implies (a,c) \in T; \\ & \text{list-all2 (fun-of } R) as bs; \text{list-all2 (fun-of } S) bs cs \rrbracket \\ & \implies \text{list-all2 (fun-of } T) as cs \end{aligned}$$

lemma *rel-list-all2-update-cong*:

$$\begin{aligned} & \llbracket i < \text{size } xs; \text{list-all2 (fun-of } S) xs ys; (x,y) \in S \rrbracket \\ & \implies \text{list-all2 (fun-of } S) (xs[i:=x]) (ys[i:=y]) \end{aligned}$$

lemma *rel-list-all2-nthD*:

$$\llbracket \text{list-all2 (fun-of } S) xs ys; p < \text{size } xs \rrbracket \implies (xs!p, ys!p) \in S$$

lemma *rel-list-all2I*:

$$\llbracket \text{length } a = \text{length } b; \bigwedge n. n < \text{length } a \implies (a!n, b!n) \in S \rrbracket \implies \text{list-all2 (fun-of } S) a b$$

1.1.3 Auxiliary properties of *map-of* function

lemma *map-of-set-pcs-notin*: $C \notin (\lambda t. \text{snd } (fst t))$ ‘set FDTs \implies *map-of* FDTs $(F, C) = \text{None}$

lemma *map-of-insertmap-SomeD'*:

$$\text{map-of } fs F = \text{Some } y \implies \text{map-of (map } (\lambda(F, y). (F, D, y)) fs) F = \text{Some}(D,y)$$

lemma *map-of-reinsert-neq-None*:

$$Ca \neq D \implies \text{map-of (map } (\lambda(F, y). ((F, Ca), y)) fs) (F, D) = \text{None}$$

lemma *map-of-remap-insertmap*:

$$\begin{aligned} & \text{map-of (map } ((\lambda((F, D), b, T). (F, D, b, T)) \circ (\lambda(F, y). ((F, D), y))) fs) \\ & = \text{map-of (map } (\lambda(F, y). (F, D, y)) fs) \end{aligned}$$

lemma *map-of-reinsert-SomeD*:

$$\text{map-of (map } (\lambda(F, y). ((F, D), y)) fs) (F, D) = \text{Some } T \implies \text{map-of } fs F = \text{Some } T$$

lemma *map-of-filtered-SomeD*:

$$\begin{aligned} & \text{map-of } fs (F,D) = \text{Some } (a, T) \implies Q ((F,D),a,T) \implies \\ & \text{map-of (map } (\lambda((F,D), b, T). ((F,D), P T)) (filter Q fs)) \\ & (F,D) = \text{Some } (P T) \end{aligned}$$

lemma *map-of-remove-filtered-SomeD*:

$$\begin{aligned} & \text{map-of } fs (F,C) = \text{Some } (a, T) \implies Q ((F,C),a,T) \implies \\ & \text{map-of (map } (\lambda((F,D), b, T). (F, P T)) [((F, D), b, T) \leftarrow fs . Q ((F, D), b, T) \wedge D = C]) \\ & F = \text{Some } (P T) \end{aligned}$$

lemma *map-of-Some-None-split*:

assumes $t = \text{map } (\lambda(F, y). ((F, C), y)) \text{ fs} @ t'$ *map-of* $t' (F, C) = \text{None}$ *map-of* $t (F, C) = \text{Some } y$
shows *map-of* $(\text{map } (\lambda((F, D), b, T). (F, D, b, T)) t) F = \text{Some } (C, y)$
end

1.2 Jinja types

theory *Type* **imports** *Auxiliary* **begin**

type-synonym *cname* = *string* — class names

type-synonym *mname* = *string* — method name

type-synonym *vname* = *string* — names for local/field variables

definition *Object* :: *cname*

where

Object \equiv "*Object*"

definition *this* :: *vname*

where

this \equiv "*this*"

definition *clinit* :: *string* **where** *clinit* = "<*clinit*>"

definition *init* :: *string* **where** *init* = "<*init*>"

definition *start-m* :: *string* **where** *start-m* = "<*start*>"

definition *Start* :: *string* **where** *Start* = "<*Start*>"

lemma *start-m-neq-clinit* [*simp*]: *start-m* \neq *clinit* **by**(*simp* *add*: *start-m-def* *clinit-def*)

lemma *Object-neq-Start* [*simp*]: *Object* \neq *Start* **by**(*simp* *add*: *Object-def* *Start-def*)

lemma *Start-neq-Object* [*simp*]: *Start* \neq *Object* **by**(*simp* *add*: *Object-def* *Start-def*)

— field/method static flag

datatype *staticb* = *Static* | *NonStatic*

— types

datatype *ty*

= *Void* — type of statements

| *Boolean*

| *Integer*

| *NT* — null type

| *Class* *cname* — class type

definition *is-refT* :: *ty* \Rightarrow *bool*

where

is-refT *T* \equiv $T = \text{NT} \vee (\exists C. T = \text{Class } C)$

lemma [*iff*]: *is-refT* *NT*

lemma [*iff*]: *is-refT* (*Class* *C*)

lemma *refTE*:

$\llbracket \text{is-refT } T; T = \text{NT} \implies P; \bigwedge C. T = \text{Class } C \implies P \rrbracket \implies P$

lemma *not-refTE*:

```

[[ ¬is-refT T; T = Void ∨ T = Boolean ∨ T = Integer ⇒ P ]] ⇒ P
end

```

1.3 Class Declarations and Programs

```

theory Decl imports Type begin

```

type-synonym

```

fdecl = vname × staticb × ty      — field declaration

```

type-synonym

```

'm mdecl = mname × staticb × ty list × ty × 'm      — method = name, static flag, arg. types,
return type, body

```

type-synonym

```

'm class = cname × fdecl list × 'm mdecl list      — class = superclass, fields, methods

```

type-synonym

```

'm cdecl = cname × 'm class      — class declaration

```

type-synonym

```

'm prog = 'm cdecl list      — program

```

definition *class* :: 'm prog ⇒ cname → 'm class

where

```

class ≡ map-of

```

lemma *class-cons*: [[C ≠ fst x]] ⇒ class (x # P) C = class P C

by (*simp add: class-def*)

definition *is-class* :: 'm prog ⇒ cname ⇒ bool

where

```

is-class P C ≡ class P C ≠ None

```

lemma *finite-is-class*: finite {C. is-class P C}

definition *is-type* :: 'm prog ⇒ ty ⇒ bool

where

```

is-type P T ≡
(case T of Void ⇒ True | Boolean ⇒ True | Integer ⇒ True | NT ⇒ True
 | Class C ⇒ is-class P C)

```

lemma *is-type-simps* [*simp*]:

```

is-type P Void ∧ is-type P Boolean ∧ is-type P Integer ∧
is-type P NT ∧ is-type P (Class C) = is-class P C

```

abbreviation

```

types P == Collect (is-type P)

```

lemma *class-exists-equiv*:

```

(∃ x. fst x = cn ∧ x ∈ set P) = (class P cn ≠ None)

```

proof(*rule iffI*)

assume ∃ x. fst x = cn ∧ x ∈ set P **then show** class P cn ≠ None

by (*metis class-def image-eqI map-of-eq-None-iff*)

next

assume class P cn ≠ None **then show** ∃ x. fst x = cn ∧ x ∈ set P

by (metis class-def fst-conv map-of-SomeD option.exhaust)
qed

lemma class-exists-equiv2:

$(\exists x. \text{fst } x = \text{cn} \wedge x \in \text{set } (P1 @ P2)) = (\text{class } P1 \text{ cn} \neq \text{None} \vee \text{class } P2 \text{ cn} \neq \text{None})$
by (simp only: class-exists-equiv [where P = P1@P2], simp add: class-def)

end

1.4 Relations between Jinja Types

theory TypeRel imports

HOL-Library.Transitive-Closure-Table

Decl

begin

1.4.1 The subclass relations

inductive-set

$\text{subcls1} :: 'm \text{ prog} \Rightarrow (\text{cname} \times \text{cname}) \text{ set}$

and $\text{subcls1}' :: 'm \text{ prog} \Rightarrow [\text{cname}, \text{cname}] \Rightarrow \text{bool}$ ($\langle \cdot \vdash \cdot \prec^1 \cdot \rangle [71, 71, 71] 70$)

for $P :: 'm \text{ prog}$

where

$P \vdash C \prec^1 D \equiv (C, D) \in \text{subcls1 } P$

| $\text{subcls1I}: [\text{class } P \ C = \text{Some } (D, \text{rest}); C \neq \text{Object}] \Longrightarrow P \vdash C \prec^1 D$

abbreviation

$\text{subcls} :: 'm \text{ prog} \Rightarrow [\text{cname}, \text{cname}] \Rightarrow \text{bool}$ ($\langle \cdot \vdash \cdot \preceq^* \cdot \rangle [71, 71, 71] 70$)

where $P \vdash C \preceq^* D \equiv (C, D) \in (\text{subcls1 } P)^*$

lemma subcls1D: $P \vdash C \prec^1 D \Longrightarrow C \neq \text{Object} \wedge (\exists \text{fs } \text{ms}. \text{class } P \ C = \text{Some } (D, \text{fs}, \text{ms}))$

lemma [iff]: $\neg P \vdash \text{Object} \prec^1 C$

lemma [iff]: $(P \vdash \text{Object} \preceq^* C) = (C = \text{Object})$

lemma subcls1-def2:

$\text{subcls1 } P =$

$(\text{SIGMA } C: \{C. \text{is-class } P \ C\}. \{D. C \neq \text{Object} \wedge \text{fst } (\text{the } (\text{class } P \ C)) = D\})$

lemma finite-subcls1: $\text{finite } (\text{subcls1 } P)$

primrec supercls-1st $:: 'm \text{ prog} \Rightarrow \text{cname list} \Rightarrow \text{bool}$ **where**

$\text{supercls-1st } P \ (C \# Cs) = ((\forall C' \in \text{set } Cs. P \vdash C' \preceq^* C) \wedge \text{supercls-1st } P \ Cs) \mid$

$\text{supercls-1st } P \ [] = \text{True}$

lemma supercls-1st-app:

$[\text{supercls-1st } P \ (C \# Cs); P \vdash C \preceq^* C'] \Longrightarrow \text{supercls-1st } P \ (C' \# C \# Cs)$

by auto

1.4.2 The subtype relations

inductive

$\text{widen} :: 'm \text{ prog} \Rightarrow \text{ty} \Rightarrow \text{ty} \Rightarrow \text{bool}$ ($\langle \cdot \vdash \cdot \leq \cdot \rangle [71, 71, 71] 70$)

for $P :: 'm \text{ prog}$

where

$\text{widen-refl}[iff]: P \vdash T \leq T$

| $\text{widen-subcls}: P \vdash C \preceq^* D \Longrightarrow P \vdash \text{Class } C \leq \text{Class } D$

| $\text{widen-null}[iff]: P \vdash \text{NT} \leq \text{Class } C$

abbreviation

widens :: 'm prog ⇒ ty list ⇒ ty list ⇒ bool
 (λ- ⊢ - [≤] → [71,71,71] 70) **where**
widens P Ts Ts' ≡ list-all2 (widen P) Ts Ts'

lemma [iff]: (P ⊢ T ≤ Void) = (T = Void)
lemma [iff]: (P ⊢ T ≤ Boolean) = (T = Boolean)
lemma [iff]: (P ⊢ T ≤ Integer) = (T = Integer)
lemma [iff]: (P ⊢ Void ≤ T) = (T = Void)
lemma [iff]: (P ⊢ Boolean ≤ T) = (T = Boolean)
lemma [iff]: (P ⊢ Integer ≤ T) = (T = Integer)

lemma *Class-widen*: P ⊢ Class C ≤ T ⇒ ∃ D. T = Class D
lemma [iff]: (P ⊢ T ≤ NT) = (T = NT)
lemma *Class-widen-Class* [iff]: (P ⊢ Class C ≤ Class D) = (P ⊢ C ≤* D)
lemma *widen-Class*: (P ⊢ T ≤ Class C) = (T = NT ∨ (∃ D. T = Class D ∧ P ⊢ D ≤* C))

lemma *widen-trans*[trans]: [P ⊢ S ≤ U; P ⊢ U ≤ T] ⇒ P ⊢ S ≤ T
lemma *widens-trans* [trans]: [P ⊢ Ss [≤] Ts; P ⊢ Ts [≤] Us] ⇒ P ⊢ Ss [≤] Us

1.4.3 Method lookup**inductive**

Methods :: ['m prog, cname, mname → (staticb × ty list × ty × 'm) × cname] ⇒ bool
 (λ- ⊢ - sees'-methods → [51,51,51] 50)

for P :: 'm prog

where

sees-methods-Object:

[class P Object = Some(D,fs,ms); Mm = map-option (λm. (m, Object)) ∘ map-of ms]
 ⇒ P ⊢ Object sees-methods Mm

| *sees-methods-rec*:

[class P C = Some(D,fs,ms); C ≠ Object; P ⊢ D sees-methods Mm;
 Mm' = Mm ++ (map-option (λm. (m, C)) ∘ map-of ms)]
 ⇒ P ⊢ C sees-methods Mm'

lemma *sees-methods-fun*:

assumes 1: P ⊢ C sees-methods Mm

shows ∧ Mm'. P ⊢ C sees-methods Mm' ⇒ Mm' = Mm

lemma *visible-methods-exist*:

P ⊢ C sees-methods Mm ⇒ Mm M = Some(m, D) ⇒
 (∃ D' fs ms. class P D = Some(D', fs, ms) ∧ map-of ms M = Some m)

lemma *sees-methods-decl-above*:

assumes Csees: P ⊢ C sees-methods Mm

shows Mm M = Some(m, D) ⇒ P ⊢ C ≤* D

lemma *sees-methods-idemp*:

assumes Cmethods: P ⊢ C sees-methods Mm

shows ∧ m D. Mm M = Some(m, D) ⇒

∃ Mm'. (P ⊢ D sees-methods Mm') ∧ Mm' M = Some(m, D)

lemma sees-methods-decl-mono:

assumes $sub: P \vdash C' \preceq^* C$

shows $P \vdash C \text{ sees-methods } Mm \implies$

$$\exists Mm' Mm_2. P \vdash C' \text{ sees-methods } Mm' \wedge Mm' = Mm ++ Mm_2 \wedge$$

$$(\forall M m D. Mm_2 M = \text{Some}(m,D) \longrightarrow P \vdash D \preceq^* C)$$

lemma sees-methods-is-class-Object:

$P \vdash D \text{ sees-methods } Mm \implies \text{is-class } P \text{ Object}$

by(*induct rule: Methods.induct; simp add: is-class-def*)

lemma sees-methods-sub-Obj: $P \vdash C \text{ sees-methods } Mm \implies P \vdash C \preceq^* \text{Object}$

proof(*induct rule: Methods.induct*)

case (*sees-methods-rec C D fs ms Mm Mm'*) **show** ?*case*

using *subcls1I[OF sees-methods-rec.hyps(1,2)] sees-methods-rec.hyps(4)*

by(*rule converse-rtrancl-into-rtrancl*)

qed(*simp*)

definition Method :: $'m \text{ prog} \Rightarrow \text{cname} \Rightarrow \text{mname} \Rightarrow \text{staticb} \Rightarrow \text{ty list} \Rightarrow \text{ty} \Rightarrow 'm \Rightarrow \text{cname} \Rightarrow \text{bool}$
 $(\langle \cdot \vdash \cdot \text{ sees } \cdot, \cdot : \cdot \rightarrow \cdot = \cdot \text{ in } \cdot \rangle [51,51,51,51,51,51,51,51] 50)$

where

$P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D \equiv$

$\exists Mm. P \vdash C \text{ sees-methods } Mm \wedge Mm M = \text{Some}((b, Ts, T, m), D)$

definition has-method :: $'m \text{ prog} \Rightarrow \text{cname} \Rightarrow \text{mname} \Rightarrow \text{staticb} \Rightarrow \text{bool}$

$(\langle \cdot \vdash \cdot \text{ has } \cdot, \cdot \rangle [51,0,0,51] 50)$

where

$P \vdash C \text{ has } M, b \equiv \exists Ts T m D. P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D$

lemma sees-method-fun:

$\llbracket P \vdash C \text{ sees } M, b: TS \rightarrow T = m \text{ in } D; P \vdash C \text{ sees } M, b': TS' \rightarrow T' = m' \text{ in } D' \rrbracket$
 $\implies b = b' \wedge TS' = TS \wedge T' = T \wedge m' = m \wedge D' = D$

lemma sees-method-decl-above:

$P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D \implies P \vdash C \preceq^* D$

lemma visible-method-exists:

$P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D \implies$

$\exists D' fs ms. \text{class } P D = \text{Some}(D', fs, ms) \wedge \text{map-of } ms M = \text{Some}(b, Ts, T, m)$

lemma sees-method-idemp:

$P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D \implies P \vdash D \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D$

lemma sees-method-decl-mono:

assumes $sub: P \vdash C' \preceq^* C$ **and**

$C\text{-sees}: P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D$ **and**

$C'\text{-sees}: P \vdash C' \text{ sees } M, b': Ts' \rightarrow T' = m' \text{ in } D'$

shows $P \vdash D' \preceq^* D$

lemma sees-methods-is-class: $P \vdash C \text{ sees-methods } Mm \implies \text{is-class } P C$

lemma sees-method-is-class:

$\llbracket P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D \rrbracket \implies \text{is-class } P C$

lemma sees-method-is-class':

$\llbracket P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D \rrbracket \implies \text{is-class } P D$

lemma sees-method-sub-Obj: $P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D \implies P \vdash C \preceq^* \text{Object}$
by (*auto simp: Method-def sees-methods-sub-Obj*)

1.4.4 Field lookup

inductive

$\text{Fields} :: ['m \text{ prog}, \text{cname}, ((\text{vname} \times \text{cname}) \times \text{staticb} \times \text{ty}) \text{ list}] \Rightarrow \text{bool}$
 $(\langle \vdash - \text{has}'\text{-fields} \rightarrow [51, 51, 51] \ 50)$

for $P :: 'm \text{ prog}$

where

has-fields-rec:

$\llbracket \text{class } P \ C = \text{Some}(D, fs, ms); C \neq \text{Object}; P \vdash D \text{ has-fields } FDTs;$
 $FDTs' = \text{map } (\lambda(F, b, T). ((F, C), b, T)) \ fs \ @ \ FDTs \rrbracket$
 $\implies P \vdash C \text{ has-fields } FDTs'$

| *has-fields-Object:*

$\llbracket \text{class } P \ \text{Object} = \text{Some}(D, fs, ms); FDTs = \text{map } (\lambda(F, b, T). ((F, \text{Object}), b, T)) \ fs \rrbracket$
 $\implies P \vdash \text{Object} \text{ has-fields } FDTs$

lemma has-fields-is-class:

$P \vdash C \text{ has-fields } FDTs \implies \text{is-class } P \ C$

lemma has-fields-fun:

assumes $1: P \vdash C \text{ has-fields } FDTs$

shows $\bigwedge FDTs'. P \vdash C \text{ has-fields } FDTs' \implies FDTs' = FDTs$

lemma all-fields-in-has-fields:

assumes *sub:* $P \vdash C \text{ has-fields } FDTs$

shows $\llbracket P \vdash C \preceq^* D; \text{class } P \ D = \text{Some}(D', fs, ms); (F, b, T) \in \text{set } fs \rrbracket$
 $\implies ((F, D), b, T) \in \text{set } FDTs$

lemma has-fields-decl-above:

assumes *fields:* $P \vdash C \text{ has-fields } FDTs$

shows $((F, D), b, T) \in \text{set } FDTs \implies P \vdash C \preceq^* D$

lemma subcls-notin-has-fields:

assumes *fields:* $P \vdash C \text{ has-fields } FDTs$

shows $((F, D), b, T) \in \text{set } FDTs \implies (D, C) \notin (\text{subcls1 } P)^+$

lemma subcls-notin-has-fields2:

assumes *fields:* $P \vdash C \text{ has-fields } FDTs$

shows $\llbracket C \neq \text{Object}; P \vdash C \prec^1 D \rrbracket \implies (D, C) \notin (\text{subcls1 } P)^*$

using *fields* **proof** (*induct arbitrary: D*)

case *has-fields-rec*

have $\forall C \ C' \ P. (C, C') \notin \text{subcls1 } P \vee C \neq \text{Object} \wedge (\exists fs \ ms. \text{class } P \ C = \llbracket (C', fs, ms) \rrbracket)$

using *subcls1D* **by** *blast*

then have $(D, D) \notin (\text{subcls1 } P)^+$

by (*metis (no-types) Pair-inject has-fields-rec.hyps(1) has-fields-rec.hyps(4)*)

has-fields-rec.prem(2) option.inject tranclD)

then show *?case*

by (*meson has-fields-rec.prem(2) rtrancl-into-trancl1*)

qed (*fastforce dest: tranclD*)

lemma has-fields-mono-lem:

assumes *sub:* $P \vdash D \preceq^* C$

shows $P \vdash C \text{ has-fields } FDTs$

$\implies \exists \text{pre}. P \vdash D \text{ has-fields } \text{pre} @ FDTs \wedge \text{dom}(\text{map-of pre}) \cap \text{dom}(\text{map-of } FDTs) = \{\}$

lemma has-fields-declaring-classes:

shows $P \vdash C$ has-fields FDTs
 $\implies \exists \text{pre FDTs}'. \text{FDTs} = \text{pre} @ \text{FDTs}'$
 $\wedge (C \neq \text{Object} \longrightarrow (\exists D \text{ fs ms. class } P \ C = [(D, \text{fs}, \text{ms})] \wedge P \vdash D \text{ has-fields FDTs}'))$
 $\wedge \text{set}(\text{map } (\lambda t. \text{snd}(\text{fst } t)) \text{pre}) \subseteq \{C\}$
 $\wedge \text{set}(\text{map } (\lambda t. \text{snd}(\text{fst } t)) \text{FDTs}') \subseteq \{C'. C' \neq C \wedge P \vdash C \preceq^* C'\}$
proof(*induct rule:Fields.induct*)
case (*has-fields-rec* $C \ D \ \text{fs} \ \text{ms} \ \text{FDTs} \ \text{FDTs}'$)
have $\text{sup1}: P \vdash C \prec^1 D$ **using** *has-fields-rec.hyps(1,2)* **by** (*simp add: subcls1.subcls1I*)
have $P \vdash C$ has-fields FDTs'
using *Fields.has-fields-rec[OF has-fields-rec.hyps(1-3)] has-fields-rec* **by** *auto*
then have $\text{nsup}: (D, C) \notin (\text{subcls1 } P)^*$ **using** *subcls-notin-has-fields2 sup1* **by** *auto*
show *?case* **using** *has-fields-rec sup1 nsup*
by(*rule-tac* $x = \text{map } (\lambda(F, y). ((F, C), y)) \ \text{fs}$ **in** *exI, clarsimp*) *auto*
next
case *has-fields-Object* **then show** *?case* **by** *fastforce*
qed

lemma *has-fields-mono-lem2*:
assumes $\text{hf}: P \vdash C$ has-fields FDTs
and $\text{cls}: \text{class } P \ C = \text{Some}(D, \text{fs}, \text{ms})$ **and** $\text{map-of}: \text{map-of FDTs } (F, C) = [(b, T)]$
shows $\exists \text{FDTs}'. \text{FDTs} = (\text{map } (\lambda(F, b, T). ((F, C), b, T)) \ \text{fs}) @ \text{FDTs}' \wedge \text{map-of FDTs}' (F, C) = \text{None}$
using *assms*
proof(*cases* $C = \text{Object}$)
case *False*
let $\text{?pre} = \text{map } (\lambda(F, b, T). ((F, C), b, T)) \ \text{fs}$
have $\text{sub}: P \vdash C \preceq^* D$ **using** cls *False* **by** (*simp add: r-into-rtrancl subcls1.subcls1I*)
obtain FDTs' **where** $\text{fdts}': P \vdash D$ has-fields FDTs' $\text{FDTs} = \text{?pre} @ \text{FDTs}'$
using *False assms(1,2) Fields.simps[of P C FDTs]* **by** *clarsimp*
then have $\text{int}: \text{dom } (\text{map-of } \text{?pre}) \cap \text{dom } (\text{map-of } \text{FDTs}') = \{\}$
using *has-fields-mono-lem[OF sub, of FDTs'] has-fields-fun[OF hf]* **by** *fastforce*
have $C \notin (\lambda t. \text{snd}(\text{fst } t)) \ \text{'set FDTs}'$
using *has-fields-declaring-classes[OF hf] cls False*
 $\text{has-fields-fun}[OF \ \text{fdts}'(1)] \ \text{fdts}'(2)$
by *clarify auto*
then have $\text{map-of FDTs}' (F, C) = \text{None}$ **by**(*rule map-of-set-pcs-notin*)
then show *?thesis* **using** $\text{fdts}' \ \text{int}$ **by** *simp*
qed(*auto dest: has-fields-Object has-fields-fun*)

lemma *has-fields-is-class-Object*:
 $P \vdash D$ has-fields FDTs $\implies \text{is-class } P \ \text{Object}$
by(*induct rule: Fields.induct; simp add: is-class-def*)

lemma *Object-fields*:
 $\llbracket P \vdash \text{Object} \text{ has-fields FDTs}; C \neq \text{Object} \rrbracket \implies \text{map-of FDTs } (F, C) = \text{None}$
by(*drule Fields.cases, auto simp: map-of-reinsert-neq-None*)

definition *has-field* :: ' $m \ \text{prog} \Rightarrow \text{cname} \Rightarrow \text{vname} \Rightarrow \text{staticb} \Rightarrow \text{ty} \Rightarrow \text{cname} \Rightarrow \text{bool}$ '
 $(\prec \vdash - \text{has } -, :- \text{ in } \rightarrow [51, 51, 51, 51, 51, 51] \ 50)$

where

$P \vdash C$ has $F, b: T$ in $D \equiv$
 $\exists \text{FDTs}. P \vdash C$ has-fields FDTs $\wedge \text{map-of FDTs } (F, D) = \text{Some } (b, T)$

lemma *has-field-mono*:

assumes *has*: $P \vdash C \text{ has } F, b:T \text{ in } D$ **and** *sub*: $P \vdash C' \preceq^* C$

shows $P \vdash C' \text{ has } F, b:T \text{ in } D$

lemma *has-field-fun*:

$\llbracket P \vdash C \text{ has } F, b:T \text{ in } D; P \vdash C \text{ has } F, b':T' \text{ in } D \rrbracket \implies b = b' \wedge T' = T$

lemma *has-field-idemp*:

assumes *has*: $P \vdash C \text{ has } F, b:T \text{ in } D$

shows $P \vdash D \text{ has } F, b:T \text{ in } D$

lemma *visible-fields-exist*:

assumes *fields*: $P \vdash C \text{ has-fields FDTs}$ **and**

FDTs: $\text{map-of FDTs } (F, D) = \text{Some } (b, T)$

shows $\exists D' \text{ fs ms. class } P D = \text{Some}(D', \text{fs}, \text{ms}) \wedge \text{map-of fs } F = \text{Some}(b, T)$

proof –

have $\text{map-of FDTs } (F, D) = \text{Some } (b, T) \longrightarrow$

$(\exists D' \text{ fs ms. class } P D = \text{Some}(D', \text{fs}, \text{ms}) \wedge \text{map-of fs } F = \text{Some}(b, T))$

using *fields* **proof** *induct*

case $(\text{has-fields-rec } C' D' \text{ fs ms FDTs}')$

with *assms* $\text{map-of-reinsert-SomeD map-of-reinsert-neq-None}$ **[where** $D=D$ **and** $F=F$ **and** $\text{fs}=\text{fs}$]

show $?case$ **proof** $(\text{cases } C' = D)$ **qed** *auto*

next

case $(\text{has-fields-Object } D' \text{ fs ms FDTs})$

with *assms* $\text{map-of-reinsert-SomeD map-of-reinsert-neq-None}$ **[where** $D=D$ **and** $F=F$ **and** $\text{fs}=\text{fs}$]

show $?case$ **proof** $(\text{cases } \text{Object} = D)$ **qed** *auto*

qed

then show $?thesis$ **using** *FDTs* **by** *simp*

qed

lemma *map-of-remap-SomeD*:

$\text{map-of } (\text{map } (\lambda((k, k'), x). (k, (k', x))) t) k = \text{Some } (k', x) \implies \text{map-of } t (k, k') = \text{Some } x$

lemma *map-of-remap-SomeD2*:

$\text{map-of } (\text{map } (\lambda((k, k'), x, x'). (k, (k', x, x'))) t) k = \text{Some } (k', x, x') \implies \text{map-of } t (k, k') = \text{Some } (x, x')$

lemma *has-field-decl-above*:

$P \vdash C \text{ has } F, b:T \text{ in } D \implies P \vdash C \preceq^* D$

definition *sees-field* :: $'m \text{ prog} \Rightarrow \text{cname} \Rightarrow \text{vname} \Rightarrow \text{staticb} \Rightarrow \text{ty} \Rightarrow \text{cname} \Rightarrow \text{bool}$

$(\langle \vdash - \text{ sees } -, :- \text{ in } \rightarrow [51, 51, 51, 51, 51, 51] 50)$

where

$P \vdash C \text{ sees } F, b:T \text{ in } D \equiv$

$\exists \text{FDTs. } P \vdash C \text{ has-fields FDTs} \wedge$

$\text{map-of } (\text{map } (\lambda((F, D), b, T). (F, (D, b, T))) \text{FDTs}) F = \text{Some}(D, b, T)$

lemma *has-visible-field*:

$P \vdash C \text{ sees } F, b:T \text{ in } D \implies P \vdash C \text{ has } F, b:T \text{ in } D$

lemma *sees-field-fun*:

$\llbracket P \vdash C \text{ sees } F, b:T \text{ in } D; P \vdash C \text{ sees } F, b':T' \text{ in } D' \rrbracket \implies b = b' \wedge T' = T \wedge D' = D$

lemma *sees-field-decl-above*:

$P \vdash C \text{ sees } F, b:T \text{ in } D \implies P \vdash C \preceq^* D$

lemma *sees-field-idemp*:

assumes *sees*: $P \vdash C \text{ sees } F, b:T \text{ in } D$

shows $P \vdash D \text{ sees } F, b:T \text{ in } D$

lemma *has-field-sees-ax*:

assumes *hf*: $P \vdash C \text{ has-fields FDTs}$ **and** *map*: $\text{map-of FDTs } (F, C) = \llbracket (b, T) \rrbracket$

shows *map-of* (*map* ($\lambda((F, D), b, T). (F, D, b, T)$) *FDTs*) *F* = [(*C*, *b*, *T*)]

proof –

obtain *D fs ms* **where** *fs*: *class P C = Some(D,fs,ms)*
using *visible-fields-exist[OF assms]* **by** *clarsimp*
then obtain *FDTs'* **where**
FDTs = map ($\lambda(F, b, T). ((F, C), b, T)$) *fs* @ *FDTs'* \wedge *map-of FDTs'* (*F*, *C*) = *None*
using *has-fields-mono-lem2[OF hf fs map]* **by** *clarsimp*
then show *?thesis* **using** *map-of-Some-None-split[OF - - map]* **by** *auto*
qed

lemma *has-field-sees*: $P \vdash C \text{ has } F, b: T \text{ in } C \implies P \vdash C \text{ sees } F, b: T \text{ in } C$
by (*auto simp: has-field-def sees-field-def has-field-sees-aux*)

lemma *has-field-is-class*:

$P \vdash C \text{ has } F, b: T \text{ in } D \implies \text{is-class } P C$

lemma *has-field-is-class'*:

$P \vdash C \text{ has } F, b: T \text{ in } D \implies \text{is-class } P D$

1.4.5 Functional lookup

definition *method* :: $'m \text{ prog} \Rightarrow \text{cname} \Rightarrow \text{mname} \Rightarrow \text{cname} \times \text{staticb} \times \text{ty list} \times \text{ty} \times 'm$
where

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

definition *field* :: $'m \text{ prog} \Rightarrow \text{cname} \Rightarrow \text{vname} \Rightarrow \text{cname} \times \text{staticb} \times \text{ty}$

where

field P C F \equiv *THE* (*D, b, T*). $P \vdash C \text{ sees } F, b: T \text{ in } D$

definition *fields* :: $'m \text{ prog} \Rightarrow \text{cname} \Rightarrow ((\text{vname} \times \text{cname}) \times \text{staticb} \times \text{ty}) \text{ list}$

where

fields P C \equiv *THE FDTs*. $P \vdash C \text{ has-fields } FDTs$

lemma *fields-def2* [*simp*]: $P \vdash C \text{ has-fields } FDTs \implies \text{fields } P C = FDTs$

lemma *field-def2* [*simp*]: $P \vdash C \text{ sees } F, b: T \text{ in } D \implies \text{field } P C F = (D, b, T)$

lemma *method-def2* [*simp*]: $P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D \implies \text{method } P C M = (D, b, Ts, T, m)$

The following are the fields for initializing an object (non-static fields) and a class (just that class's static fields), respectively.

definition *ifields* :: $'m \text{ prog} \Rightarrow \text{cname} \Rightarrow ((\text{vname} \times \text{cname}) \times \text{staticb} \times \text{ty}) \text{ list}$

where

ifields P C \equiv *filter* ($\lambda((F, D), b, T). b = \text{NonStatic}$) (*fields P C*)

definition *isfields* :: $'m \text{ prog} \Rightarrow \text{cname} \Rightarrow ((\text{vname} \times \text{cname}) \times \text{staticb} \times \text{ty}) \text{ list}$

where

isfields P C \equiv *filter* ($\lambda((F, D), b, T). b = \text{Static} \wedge D = C$) (*fields P C*)

lemma *ifields-def2* [*simp*]: $\llbracket P \vdash C \text{ has-fields } FDTs \rrbracket \implies \text{ifields } P C = \text{filter} (\lambda((F, D), b, T). b = \text{NonStatic}) FDTs$

by (*simp add: ifields-def*)

lemma *isfields-def2* [*simp*]: $\llbracket P \vdash C \text{ has-fields } FDTs \rrbracket \implies \text{isfields } P C = \text{filter} (\lambda((F, D), b, T). b = \text{Static} \wedge D = C) FDTs$

by (*simp add: isfields-def*)

lemma *ifields-def3*: $\llbracket P \vdash C \text{ sees } F, b: T \text{ in } D; b = \text{NonStatic} \rrbracket \implies (((F, D), b, T) \in \text{set } (\text{ifields } P \ C))$

lemma *isfields-def3*: $\llbracket P \vdash C \text{ sees } F, b: T \text{ in } D; b = \text{Static}; D = C \rrbracket \implies (((F, D), b, T) \in \text{set } (\text{isfields } P \ C))$

definition *seeing-class* :: $'m \text{ prog} \Rightarrow \text{cname} \Rightarrow \text{mname} \Rightarrow \text{cname option}$ **where**

seeing-class $P \ C \ M =$

(if $\exists Ts \ T \ m \ D. P \vdash C \text{ sees } M, \text{Static}: Ts \rightarrow T = m \text{ in } D$
then $\text{Some } (\text{fst}(\text{method } P \ C \ M))$
else None)

lemma *seeing-class-def2*[*simp*]:

$P \vdash C \text{ sees } M, \text{Static}: Ts \rightarrow T = m \text{ in } D \implies \text{seeing-class } P \ C \ M = \text{Some } D$

by(*fastforce simp: seeing-class-def*)

1.5 Jinja Values

theory *Value* **imports** *TypeRel* **begin**

type-synonym *addr* = *nat*

datatype *val*

= *Unit* — dummy result value of void expressions
| *Null* — null reference
| *Bool bool* — Boolean value
| *Intg int* — integer value
| *Addr addr* — addresses of objects in the heap

primrec *the-Intg* :: $\text{val} \Rightarrow \text{int}$ **where**

the-Intg (*Intg i*) = i

primrec *the-Addr* :: $\text{val} \Rightarrow \text{addr}$ **where**

the-Addr (*Addr a*) = a

primrec *default-val* :: $\text{ty} \Rightarrow \text{val}$ — default value for all types **where**

default-val Void = *Unit*
| *default-val Boolean* = *Bool False*
| *default-val Integer* = *Intg 0*
| *default-val NT* = *Null*
| *default-val (Class C)* = *Null*

end

1.6 Objects and the Heap

theory *Objects* **imports** *TypeRel Value* **begin**

1.6.1 Objects

type-synonym

fields = $\text{vname} \times \text{cname} \rightarrow \text{val}$ — field name, defining class, value

type-synonym

obj = $\text{cname} \times \text{fields}$ — class instance with class name and fields

type-synonym

$sfields = vname \rightarrow val$ — field name to value

definition $obj\text{-}ty :: obj \Rightarrow ty$

where

$obj\text{-}ty\ obj \equiv Class\ (fst\ obj)$

— initializes a given list of fields

definition $init\text{-}fields :: ((vname \times cname) \times staticb \times ty)\ list \Rightarrow fields$

where

$init\text{-}fields\ FDTs \equiv (map\text{-}of \circ map\ (\lambda((F,D),b,T). ((F,D),default\text{-}val\ T)))\ FDTs$

definition $init\text{-}sfields :: ((vname \times cname) \times staticb \times ty)\ list \Rightarrow sfields$

where

$init\text{-}sfields\ FDTs \equiv (map\text{-}of \circ map\ (\lambda((F,D),b,T). (F,default\text{-}val\ T)))\ FDTs$

— a new, blank object with default values for instance fields:

definition $blank :: 'm\ prog \Rightarrow cname \Rightarrow obj$

where

$blank\ P\ C \equiv (C,init\text{-}fields\ (ifields\ P\ C))$

— a new, blank object with default values for static fields:

definition $sblank :: 'm\ prog \Rightarrow cname \Rightarrow sfields$

where

$sblank\ P\ C \equiv init\text{-}sfields\ (isfields\ P\ C)$

lemma $[simp]: obj\text{-}ty\ (C,fs) = Class\ C$

translations

$(type)\ fields \leq (type)\ char\ list \times char\ list \Rightarrow val\ option$

$(type)\ obj \leq (type)\ char\ list \times fields$

$(type)\ sfields \leq (type)\ char\ list \Rightarrow val\ option$

1.6.2 Heap

type-synonym $heap = addr \rightarrow obj$

translations

$(type)\ heap \leq (type)\ nat \Rightarrow obj\ option$

abbreviation

$cname\text{-}of :: heap \Rightarrow addr \Rightarrow cname$ **where**

$cname\text{-}of\ hp\ a == fst\ (the\ (hp\ a))$

definition $new\text{-}Addr :: heap \Rightarrow addr\ option$

where

$new\text{-}Addr\ h \equiv if\ \exists a. h\ a = None\ then\ Some(LEAST\ a.\ h\ a = None)\ else\ None$

definition $cast\text{-}ok :: 'm\ prog \Rightarrow cname \Rightarrow heap \Rightarrow val \Rightarrow bool$

where

$cast\text{-}ok\ P\ C\ h\ v \equiv v = Null \vee P \vdash cname\text{-}of\ h\ (the\text{-}Addr\ v) \preceq^* C$

definition $hext :: heap \Rightarrow heap \Rightarrow bool\ (\prec \trianglelefteq \rightarrow [51,51]\ 50)$

where

$$h \trianglelefteq h' \equiv \forall a C fs. h a = \text{Some}(C, fs) \longrightarrow (\exists fs'. h' a = \text{Some}(C, fs'))$$

primrec *typeof-h* :: *heap* \Rightarrow *val* \Rightarrow *ty option* (*<typeof->*)

where

$$\begin{aligned} & \text{typeof}_h \text{ Unit} &= \text{Some Void} \\ | \text{typeof}_h \text{ Null} &= \text{Some NT} \\ | \text{typeof}_h (\text{Bool } b) &= \text{Some Boolean} \\ | \text{typeof}_h (\text{Intg } i) &= \text{Some Integer} \\ | \text{typeof}_h (\text{Addr } a) &= (\text{case } h a \text{ of None} \Rightarrow \text{None} \mid \text{Some}(C, fs) \Rightarrow \text{Some}(\text{Class } C)) \end{aligned}$$

lemma *new-Addr-SomeD*:

$$\text{new-Addr } h = \text{Some } a \Longrightarrow h a = \text{None}$$

lemma [*simp*]: (*typeof_h v = Some Boolean*) = ($\exists b. v = \text{Bool } b$)

lemma [*simp*]: (*typeof_h v = Some Integer*) = ($\exists i. v = \text{Intg } i$)

lemma [*simp*]: (*typeof_h v = Some NT*) = ($v = \text{Null}$)

lemma [*simp*]: (*typeof_h v = Some(Class C)*) = ($\exists a fs. v = \text{Addr } a \wedge h a = \text{Some}(C, fs)$)

lemma [*simp*]: $h a = \text{Some}(C, fs) \Longrightarrow \text{typeof}_{(h(a \mapsto (C, fs')))} v = \text{typeof}_h v$

For literal values the first parameter of *typeof* can be set to $\{\}$ because they do not contain addresses:

abbreviation

$$\begin{aligned} & \text{typeof} :: \text{val} \Rightarrow \text{ty option} \textbf{ where} \\ & \text{typeof } v == \text{typeof-h Map.empty } v \end{aligned}$$

lemma *typeof-lit-typeof*:

$$\text{typeof } v = \text{Some } T \Longrightarrow \text{typeof}_h v = \text{Some } T$$

lemma *typeof-lit-is-type*:

$$\text{typeof } v = \text{Some } T \Longrightarrow \text{is-type } P T$$

1.6.3 Heap extension \trianglelefteq

lemma *hextI*: $\forall a C fs. h a = \text{Some}(C, fs) \longrightarrow (\exists fs'. h' a = \text{Some}(C, fs')) \Longrightarrow h \trianglelefteq h'$

lemma *hext-objD*: $\llbracket h \trianglelefteq h'; h a = \text{Some}(C, fs) \rrbracket \Longrightarrow \exists fs'. h' a = \text{Some}(C, fs')$

lemma *hext-refl* [*iff*]: $h \trianglelefteq h$

lemma *hext-new* [*simp*]: $h a = \text{None} \Longrightarrow h \trianglelefteq h(a \mapsto x)$

lemma *hext-trans*: $\llbracket h \trianglelefteq h'; h' \trianglelefteq h'' \rrbracket \Longrightarrow h \trianglelefteq h''$

lemma *hext-upd-obj*: $h a = \text{Some}(C, fs) \Longrightarrow h \trianglelefteq h(a \mapsto (C, fs'))$

lemma *hext-typeof-mono*: $\llbracket h \trianglelefteq h'; \text{typeof}_h v = \text{Some } T \rrbracket \Longrightarrow \text{typeof}_{h'} v = \text{Some } T$

1.6.4 Static field information function

datatype *init-state* = *Done* | *Processing* | *Prepared* | *Error*

- *Done* = initialized
- *Processing* = currently being initialized
- *Prepared* = uninitialized and not currently being initialized
- *Error* = previous initialization attempt resulted in erroneous state

inductive *ipro* :: *init-state* \Rightarrow *init-state* \Rightarrow *bool* ($\langle - \leq_i - \rangle$ [51,51] 50)

where

[*simp*]: *Prepared* \leq_i *i*
| [*simp*]: *Processing* \leq_i *Done*
| [*simp*]: *Processing* \leq_i *Error*
| [*simp*]: *i* \leq_i *i*

lemma *ipro-Done*[*simp*]: (*Done* \leq_i *i*) = (*i* = *Done*)
by(*simp only: iprog.simps, simp*)

lemma *ipro-Error*[*simp*]: (*Error* \leq_i *i*) = (*i* = *Error*)
by(*simp only: iprog.simps, simp*)

lemma *ipro-Processing*[*simp*]: (*Processing* \leq_i *i*) = (*i* = *Done* \vee *i* = *Error* \vee *i* = *Processing*)
by(*simp only: iprog.simps, simp*)

lemma *ipro-trans*: $\llbracket i \leq_i i'; i' \leq_i i'' \rrbracket \Longrightarrow i \leq_i i''$

1.6.5 Static Heap

The static heap (sheap) is used for storing information about static field values and initialization status for classes.

type-synonym

sheap = *cname* \rightarrow *sfields* \times *init-state*

translations

(*type*) *sheap* \leq (*type*) *char list* \Rightarrow (*sfields* \times *init-state*) *option*

definition *shext* :: *sheap* \Rightarrow *sheap* \Rightarrow *bool* ($\langle - \leq_s - \rangle$ [51,51] 50)

where

$sh \leq_s sh' \equiv \forall C \text{ sfs } i. sh \ C = \text{Some}(sfs, i) \longrightarrow (\exists sfs' i'. sh' \ C = \text{Some}(sfs', i') \wedge i \leq_i i')$

lemma *shextI*: $\forall C \text{ sfs } i. sh \ C = \text{Some}(sfs, i) \longrightarrow (\exists sfs' i'. sh' \ C = \text{Some}(sfs', i') \wedge i \leq_i i') \Longrightarrow sh \leq_s sh'$

lemma *shext-objD*: $\llbracket sh \leq_s sh'; sh \ C = \text{Some}(sfs, i) \rrbracket \Longrightarrow \exists sfs' i'. sh' \ C = \text{Some}(sfs', i') \wedge i \leq_i i'$

lemma *shext-refl* [*iff*]: $sh \leq_s sh$

lemma *shext-new* [*simp*]: $sh \ C = \text{None} \Longrightarrow sh \leq_s sh(C \mapsto x)$

lemma *shext-trans*: $\llbracket sh \leq_s sh'; sh' \leq_s sh'' \rrbracket \Longrightarrow sh \leq_s sh''$

lemma *shext-upd-obj*: $\llbracket sh \ C = \text{Some}(sfs, i); i \leq_i i' \rrbracket \Longrightarrow sh \leq_s sh(C \mapsto (sfs', i'))$

end

1.7 Exceptions

theory *Exceptions* **imports** *Objects* **begin**

definition *ErrorCl* :: *string* **where** *ErrorCl* = "Error"

definition *ThrowCl* :: *string* **where** *ThrowCl* = "Throwable"

definition *NullPointer* :: *cname*

where

NullPointer \equiv "NullPointer"

definition *ClassCast* :: *cname*

where

ClassCast \equiv "ClassCast"

definition *OutOfMemory* :: *cname*

where

OutOfMemory \equiv "OutOfMemory"

definition *NoClassDefFoundError* :: *cname*

where

NoClassDefFoundError \equiv "NoClassDefFoundError"

definition *IncompatibleClassChangeError* :: *cname*

where

IncompatibleClassChangeError \equiv "IncompatibleClassChangeError"

definition *NoSuchFieldError* :: *cname*

where

NoSuchFieldError \equiv "NoSuchFieldError"

definition *NoSuchMethodError* :: *cname*

where

NoSuchMethodError \equiv "NoSuchMethodError"

definition *sys-xcpts* :: *cname set*

where

sys-xcpts \equiv {*NullPointer*, *ClassCast*, *OutOfMemory*, *NoClassDefFoundError*,
IncompatibleClassChangeError,
NoSuchFieldError, *NoSuchMethodError*}

definition *addr-of-sys-xcpt* :: *cname* \Rightarrow *addr*

where

addr-of-sys-xcpt *s* \equiv if *s* = *NullPointer* then 0 else
if *s* = *ClassCast* then 1 else
if *s* = *OutOfMemory* then 2 else
if *s* = *NoClassDefFoundError* then 3 else
if *s* = *IncompatibleClassChangeError* then 4 else
if *s* = *NoSuchFieldError* then 5 else
if *s* = *NoSuchMethodError* then 6 else undefined

lemmas *sys-xcpts-defs* = *NullPointer-def* *ClassCast-def* *OutOfMemory-def* *NoClassDefFoundError-def*
IncompatibleClassChangeError-def *NoSuchFieldError-def* *NoSuchMethodError-def*

lemma *Start-nsys-xcpts*: *Start* \notin *sys-xcpts*

by(*simp* *add*: *Start-def* *sys-xcpts-def* *sys-xcpts-defs*)

lemma *Start-nsys-xcpts1* [*simp*]: *Start* \neq *NullPointer* *Start* \neq *ClassCast*

Start \neq *OutOfMemory* *Start* \neq *NoClassDefFoundError*

Start \neq *IncompatibleClassChangeError* *Start* \neq *NoSuchFieldError*

Start \neq *NoSuchMethodError*

using *Start-nsys-xcpts* **by**(*auto simp*: *sys-xcpts-def*)

lemma *Start-nsys-xcpts2* [simp]: *NullPointer* \neq *Start* *ClassCast* \neq *Start*
OutOfMemory \neq *Start* *NoClassDefFoundError* \neq *Start*
IncompatibleClassChangeError \neq *Start* *NoSuchFieldError* \neq *Start*
NoSuchMethodError \neq *Start*
using *Start-nsys-xcpts* **by**(*auto simp: sys-xcpts-def dest: sym*)

definition *start-heap* :: 'c prog \Rightarrow heap

where

start-heap *G* \equiv *Map.empty* (*addr-of-sys-xcpt NullPointer* \mapsto *blank* *G* *NullPointer*,
addr-of-sys-xcpt ClassCast \mapsto *blank* *G* *ClassCast*,
addr-of-sys-xcpt OutOfMemory \mapsto *blank* *G* *OutOfMemory*,
addr-of-sys-xcpt NoClassDefFoundError \mapsto *blank* *G* *NoClassDefFoundError*,
addr-of-sys-xcpt IncompatibleClassChangeError \mapsto *blank* *G* *IncompatibleClass-*
ChangeError,
addr-of-sys-xcpt NoSuchFieldError \mapsto *blank* *G* *NoSuchFieldError*,
addr-of-sys-xcpt NoSuchMethodError \mapsto *blank* *G* *NoSuchMethodError*)

definition *preallocated* :: heap \Rightarrow bool

where

preallocated *h* \equiv $\forall C \in \text{sys-xcpts}. \exists fs. h(\text{addr-of-sys-xcpt } C) = \text{Some } (C, fs)$

1.7.1 System exceptions

lemma *sys-xcpts-incl* [simp]: *NullPointer* \in *sys-xcpts* \wedge *OutOfMemory* \in *sys-xcpts*
 \wedge *ClassCast* \in *sys-xcpts* \wedge *NoClassDefFoundError* \in *sys-xcpts*
 \wedge *IncompatibleClassChangeError* \in *sys-xcpts* \wedge *NoSuchFieldError* \in *sys-xcpts*
 \wedge *NoSuchMethodError* \in *sys-xcpts*

lemma *sys-xcpts-cases* [consumes 1, cases set]:

$\llbracket C \in \text{sys-xcpts}; P \text{ NullPointer}; P \text{ OutOfMemory}; P \text{ ClassCast}; P \text{ NoClassDefFoundError};$
 $P \text{ IncompatibleClassChangeError}; P \text{ NoSuchFieldError};$
 $P \text{ NoSuchMethodError} \rrbracket \Longrightarrow P C$

1.7.2 Starting heap

lemma *start-heap-sys-xcpts*:

assumes $C \in \text{sys-xcpts}$

shows *start-heap* *P* (*addr-of-sys-xcpt* *C*) = *Some*(*blank* *P* *C*)

by(*rule sys-xcpts-cases[OF assms]*)

(*auto simp add: start-heap-def sys-xcpts-def addr-of-sys-xcpt-def sys-xcpts-defs*)

lemma *start-heap-classes*:

start-heap *P* *a* = *Some*(*C, fs*) $\Longrightarrow C \in \text{sys-xcpts}$

by(*simp add: start-heap-def blank-def split: if-split-asm*)

lemma *start-heap-nStart*: *start-heap* *P* *a* = *Some* *obj* $\Longrightarrow \text{fst}(\text{obj}) \neq \text{Start}$

by(*cases obj, auto dest!: start-heap-classes simp: Start-nsys-xcpts*)

1.7.3 preallocated

lemma *preallocated-dom* [simp]:

$\llbracket \text{preallocated } h; C \in \text{sys-xcpts} \rrbracket \Longrightarrow \text{addr-of-sys-xcpt } C \in \text{dom } h$

lemma *preallocatedD*:

$\llbracket \text{preallocated } h; C \in \text{sys-xcpts} \rrbracket \Longrightarrow \exists fs. h(\text{addr-of-sys-xcpt } C) = \text{Some } (C, fs)$

lemma *preallocatedE* [*elim?*]:

$\llbracket \text{preallocated } h; C \in \text{sys-xcpts}; \bigwedge fs. h(\text{addr-of-sys-xcpt } C) = \text{Some}(C,fs) \implies P \ h \ C \rrbracket$
 $\implies P \ h \ C$

lemma *cname-of-xcp* [*simp*]:

$\llbracket \text{preallocated } h; C \in \text{sys-xcpts} \rrbracket \implies \text{cname-of } h \ (\text{addr-of-sys-xcpt } C) = C$

lemma *typeof-ClassCast* [*simp*]:

$\text{preallocated } h \implies \text{typeof}_h \ (\text{Addr}(\text{addr-of-sys-xcpt } \text{ClassCast})) = \text{Some}(\text{Class } \text{ClassCast})$

lemma *typeof-OutOfMemory* [*simp*]:

$\text{preallocated } h \implies \text{typeof}_h \ (\text{Addr}(\text{addr-of-sys-xcpt } \text{OutOfMemory})) = \text{Some}(\text{Class } \text{OutOfMemory})$

lemma *typeof-NullPointer* [*simp*]:

$\text{preallocated } h \implies \text{typeof}_h \ (\text{Addr}(\text{addr-of-sys-xcpt } \text{NullPointer})) = \text{Some}(\text{Class } \text{NullPointer})$

lemma *typeof-NoClassDefFoundError* [*simp*]:

$\text{preallocated } h \implies \text{typeof}_h \ (\text{Addr}(\text{addr-of-sys-xcpt } \text{NoClassDefFoundError})) = \text{Some}(\text{Class } \text{NoClassDefFoundError})$

lemma *typeof-IncompatibleClassChangeError* [*simp*]:

$\text{preallocated } h \implies \text{typeof}_h \ (\text{Addr}(\text{addr-of-sys-xcpt } \text{IncompatibleClassChangeError})) = \text{Some}(\text{Class } \text{IncompatibleClassChangeError})$

lemma *typeof-NoSuchFieldError* [*simp*]:

$\text{preallocated } h \implies \text{typeof}_h \ (\text{Addr}(\text{addr-of-sys-xcpt } \text{NoSuchFieldError})) = \text{Some}(\text{Class } \text{NoSuchFieldError})$

lemma *typeof-NoSuchMethodError* [*simp*]:

$\text{preallocated } h \implies \text{typeof}_h \ (\text{Addr}(\text{addr-of-sys-xcpt } \text{NoSuchMethodError})) = \text{Some}(\text{Class } \text{NoSuchMethodError})$

lemma *preallocated-hext*:

$\llbracket \text{preallocated } h; h \triangleleft h' \rrbracket \implies \text{preallocated } h'$

lemma *preallocated-start*:

$\text{preallocated } (\text{start-heap } P)$

by(*auto simp: start-heap-sys-xcpts blank-def preallocated-def*)

end

1.8 Expressions

theory *Expr*

imports *../Common/Exceptions*

begin

datatype *bop* = *Eq* | *Add* — names of binary operations

datatype *'a exp*

= *new cname* — class instance creation

| *Cast cname ('a exp)* — type cast

| *Val val* — value

| *BinOp ('a exp) bop ('a exp)* ($\langle \leftarrow \leftarrow \rightarrow \rangle [80,0,81] 80$) — binary operation

| *Var 'a* — local variable (incl. parameter)

| *LAss 'a ('a exp)* ($\langle \leftarrow := \rightarrow \rangle [90,90] 90$) — local assignment

| *FAcc ('a exp) vname cname* ($\langle \leftarrow \leftarrow \{-\} \rangle [10,90,99] 90$) — field access

| *SFAcc cname vname cname* ($\langle \leftarrow \leftarrow _s \{-\} \rangle [10,90,99] 90$) — static field access

| *FAss ('a exp) vname cname ('a exp)* ($\langle \leftarrow \leftarrow \{-\} := \rightarrow \rangle [10,90,99,90] 90$) — field assignment

| *SFAss* *cname vname cname ('a exp)* ($\langle \cdot \cdot_s \{-\} := \rightarrow [10,90,99,90] 90$) — static field assignment
 | *Call* (*'a exp*) *mname ('a exp list)* ($\langle \cdot \cdot \cdot \{-\} \rangle [90,99,0] 90$) — method call
 | *SCall* *cname mname ('a exp list)* ($\langle \cdot \cdot_s \{-\} \rangle [90,99,0] 90$) — static method call
 | *Block* *'a ty ('a exp)* ($\langle \{ \cdot \cdot \cdot \} \rangle$)
 | *Seq* (*'a exp*) (*'a exp*) ($\langle \cdot \cdot \cdot \{-\} \rightarrow [61,60] 60$)
 | *Cond* (*'a exp*) (*'a exp*) (*'a exp*) ($\langle \text{if } \{-\} \text{ -/ else } \rightarrow [80,79,79] 70$)
 | *While* (*'a exp*) (*'a exp*) ($\langle \text{while } \{-\} \rightarrow [80,79] 70$)
 | *throw* (*'a exp*)
 | *TryCatch* (*'a exp*) *cname 'a ('a exp)* ($\langle \text{try -/ catch } \{-\} \rightarrow [0,99,80,79] 70$)
 | *INIT* *cname cname list bool ('a exp)* ($\langle \text{INIT } \{-\} \leftarrow \rightarrow [60,60,60,60] 60$) — internal initialization
 command: class, list of superclasses to initialize, preparation flag; command on hold
 | *RI* *cname ('a exp) cname list ('a exp)* ($\langle \text{RI } \{-\} \cdot \leftarrow \rightarrow [60,60,60,60] 60$) — running of the
 initialization procedure for class with expression, classes still to initialize command on hold

type-synonym

expr = *vname exp* — Jinja expression

type-synonym

J-mb = *vname list* \times *expr* — Jinja method body: parameter names and expression

type-synonym

J-prog = *J-mb prog* — Jinja program

type-synonym

init-stack = *expr list* \times *bool* — Stack of expressions waiting on initialization in small step; indicator
 boolean True if current expression has been init checked

The semantics of binary operators:

fun *binop* :: *bop* \times *val* \times *val* \Rightarrow *val option* **where**
binop(*Eq*, *v*₁, *v*₂) = *Some*(*Bool* (*v*₁ = *v*₂))
| *binop*(*Add*, *Intg* *i*₁, *Intg* *i*₂) = *Some*(*Intg*(*i*₁+*i*₂))
| *binop*(*bop*, *v*₁, *v*₂) = *None*

lemma [*simp*]:

(*binop*(*Add*, *v*₁, *v*₂) = *Some* *v*) = (\exists *i*₁ *i*₂. *v*₁ = *Intg* *i*₁ \wedge *v*₂ = *Intg* *i*₂ \wedge *v* = *Intg*(*i*₁+*i*₂))

lemma *map-Val-throw-eq*:

map Val vs @ *throw ex* # *es* = *map Val vs'* @ *throw ex'* # *es'* \Longrightarrow *ex* = *ex'*

lemma *map-Val-nthrow-neq*:

map Val vs = *map Val vs'* @ *throw ex'* # *es'* \Longrightarrow *False*

lemma *map-Val-eq*:

map Val vs = *map Val vs'* \Longrightarrow *vs* = *vs'*

lemma *init-rhs-neq* [*simp*]: *e* \neq *INIT C* (*Cs*, *b*) \leftarrow *e***proof** –

have *size e* \neq *size (INIT C* (*Cs*, *b*) \leftarrow *e*) **by** *auto*

then show *?thesis* **by** *fastforce*

qed

lemma *init-rhs-neq'* [*simp*]: *INIT C* (*Cs*, *b*) \leftarrow *e* \neq *e***proof** –

have *size e* \neq *size (INIT C* (*Cs*, *b*) \leftarrow *e*) **by** *auto*

then show *?thesis* **by** *fastforce*

qed

lemma *ri-rhs-neq* [*simp*]: *e* \neq *RI*(*C*, *e'*); *Cs* \leftarrow *e*

proof –

have $\text{size } e \neq \text{size } (RI(C, e'); Cs \leftarrow e)$ **by** *auto*
then show *?thesis* **by** *fastforce*

qed

lemma *ri-rhs-neq'* [*simp*]: $RI(C, e'); Cs \leftarrow e \neq e$

proof –

have $\text{size } e \neq \text{size } (RI(C, e'); Cs \leftarrow e)$ **by** *auto*
then show *?thesis* **by** *fastforce*

qed

1.8.1 Syntactic sugar

abbreviation (*input*)

InitBlock:: $'a \Rightarrow ty \Rightarrow 'a \text{ exp} \Rightarrow 'a \text{ exp} \Rightarrow 'a \text{ exp} \quad (\langle (1' \{-:- := -;/ -\}) \rangle)$ **where**
InitBlock $V T e1 e2 == \{V:T; V := e1;; e2\}$

abbreviation *unit* **where** *unit* == *Val Unit*

abbreviation *null* **where** *null* == *Val Null*

abbreviation *addr* $a == \text{Val}(\text{Addr } a)$

abbreviation *true* == *Val(Bool True)*

abbreviation *false* == *Val(Bool False)*

abbreviation

Throw :: *addr* $\Rightarrow 'a \text{ exp}$ **where**

Throw $a == \text{throw}(\text{Val}(\text{Addr } a))$

abbreviation

THROW :: *cname* $\Rightarrow 'a \text{ exp}$ **where**

THROW $xc == \text{Throw}(\text{addr-of-sys-xcpt } xc)$

1.8.2 Free Variables

primrec *fv* :: *expr* $\Rightarrow \text{vname set}$ **and** *fvs* :: *expr list* $\Rightarrow \text{vname set}$ **where**

$\text{fv}(\text{new } C) = \{\}$
 $\text{fv}(\text{Cast } C \ e) = \text{fv } e$
 $\text{fv}(\text{Val } v) = \{\}$
 $\text{fv}(e_1 \ll \text{bop} \gg e_2) = \text{fv } e_1 \cup \text{fv } e_2$
 $\text{fv}(\text{Var } V) = \{V\}$
 $\text{fv}(\text{LAss } V \ e) = \{V\} \cup \text{fv } e$
 $\text{fv}(e \cdot F\{D\}) = \text{fv } e$
 $\text{fv}(C \cdot_s F\{D\}) = \{\}$
 $\text{fv}(e_1 \cdot F\{D\} := e_2) = \text{fv } e_1 \cup \text{fv } e_2$
 $\text{fv}(C \cdot_s F\{D\} := e_2) = \text{fv } e_2$
 $\text{fv}(e \cdot M(es)) = \text{fv } e \cup \text{fvs } es$
 $\text{fv}(C \cdot_s M(es)) = \text{fvs } es$
 $\text{fv}(\{V:T; e\}) = \text{fv } e - \{V\}$
 $\text{fv}(e_1;; e_2) = \text{fv } e_1 \cup \text{fv } e_2$
 $\text{fv}(\text{if } (b) \ e_1 \ \text{else } e_2) = \text{fv } b \cup \text{fv } e_1 \cup \text{fv } e_2$
 $\text{fv}(\text{while } (b) \ e) = \text{fv } b \cup \text{fv } e$
 $\text{fv}(\text{throw } e) = \text{fv } e$
 $\text{fv}(\text{try } e_1 \ \text{catch } (C \ V) \ e_2) = \text{fv } e_1 \cup (\text{fv } e_2 - \{V\})$
 $\text{fv}(\text{INIT } C \ (Cs, b) \leftarrow e) = \text{fv } e$
 $\text{fv}(RI \ (C, e); Cs \leftarrow e') = \text{fv } e \cup \text{fv } e'$

| $fvs(\[]) = \{\}$
| $fvs(e\#es) = fv\ e \cup fvs\ es$

lemma [simp]: $fvs(es_1 @ es_2) = fvs\ es_1 \cup fvs\ es_2$

lemma [simp]: $fvs(map\ Val\ vs) = \{\}$

1.8.3 Accessing expression constructor arguments

fun *val-of* :: 'a exp \Rightarrow val option **where**
val-of (Val v) = Some v |
val-of - = None

lemma *val-of-spec*: $val-of\ e = Some\ v \Longrightarrow e = Val\ v$

proof(cases e) **qed**(auto)

fun *lass-val-of* :: 'a exp \Rightarrow ('a \times val) option **where**
lass-val-of (V:=Val v) = Some (V, v) |
lass-val-of - = None

lemma *lass-val-of-spec*:

assumes *lass-val-of* e = [a]

shows e = (fst a:=Val (snd a))

using *assms* **proof**(cases e)

case (LAss V e') **then show** ?thesis **using** *assms* **proof**(cases e')**qed**(auto)

qed(auto)

fun *map-vals-of* :: 'a exp list \Rightarrow val list option **where**

map-vals-of (e#es) = (case *val-of* e of Some v \Rightarrow (case *map-vals-of* es of Some vs \Rightarrow Some (v#vs)
| - \Rightarrow None) |
| - \Rightarrow None) |

map-vals-of [] = Some []

lemma *map-vals-of-spec*: $map-vals-of\ es = Some\ vs \Longrightarrow es = map\ Val\ vs$

proof(induct es arbitrary: vs) **qed**(auto simp: *val-of-spec*)

lemma *map-vals-of-Vals*[simp]: $map-vals-of\ (map\ Val\ vs) = [vs]$ **by**(induct vs, auto)

lemma *map-vals-of-throw*[simp]:

map-vals-of (map Val vs @ throw e # es') = None

by(induct vs, auto)

fun *bool-of* :: 'a exp \Rightarrow bool option **where**

bool-of true = Some True |

bool-of false = Some False |

bool-of - = None

lemma *bool-of-specT*:

assumes *bool-of* e = Some True **shows** e = true

proof -

have *bool-of* e = Some True **by** fact

then show ?thesis

proof(cases e)

case (Val x3) **with** *assms* **show** ?thesis

```

proof(cases x $\beta$ )
  case (Bool x) with assms Val show ?thesis
  proof(cases x) qed(auto)
  qed(simp-all)
qed(auto)
qed

```

```

lemma bool-of-specF:
assumes bool-of e = Some False shows e = false
proof -
  have bool-of e = Some False by fact
  then show ?thesis
  proof(cases e)
    case (Val x $\beta$ ) with assms show ?thesis
    proof(cases x $\beta$ )
      case (Bool x) with assms Val show ?thesis
      proof(cases x) qed(auto)
      qed(simp-all)
    qed(auto)
  qed

```

```

fun throw-of :: 'a exp  $\Rightarrow$  'a exp option where
throw-of (throw e') = Some e' |
throw-of - = None

```

```

lemma throw-of-spec: throw-of e = Some e'  $\Longrightarrow$  e = throw e'
proof(cases e) qed(auto)

```

```

fun init-exp-of :: 'a exp  $\Rightarrow$  'a exp option where
init-exp-of (INIT C (Cs,b)  $\leftarrow$  e) = Some e |
init-exp-of (RI(C,e');Cs  $\leftarrow$  e) = Some e |
init-exp-of - = None

```

```

lemma init-exp-of-neq [simp]: init-exp-of e = [e']  $\Longrightarrow$  e'  $\neq$  e by(cases e, auto)

```

```

lemma init-exp-of-neq' [simp]: init-exp-of e = [e']  $\Longrightarrow$  e  $\neq$  e' by(cases e, auto)

```

1.8.4 Class initialization

This section defines a few functions that return information about an expression's current initialization status.

```

primrec sub-RI :: 'a exp  $\Rightarrow$  bool and sub-RIs :: 'a exp list  $\Rightarrow$  bool where
  sub-RI(new C) = False
| sub-RI(Cast C e) = sub-RI e
| sub-RI(Val v) = False
| sub-RI(e1 «bop» e2) = (sub-RI e1  $\vee$  sub-RI e2)
| sub-RI(Var V) = False
| sub-RI(LAss V e) = sub-RI e
| sub-RI(e·F{D}) = sub-RI e
| sub-RI(C·sF{D}) = False
| sub-RI(e1·F{D};=e2) = (sub-RI e1  $\vee$  sub-RI e2)
| sub-RI(C·sF{D};=e2) = sub-RI e2
| sub-RI(e·M(es)) = (sub-RI e  $\vee$  sub-RIs es)
| sub-RI(C·sM(es)) = (M = clinit  $\vee$  sub-RIs es)

```

```

| sub-RI({V:T; e}) = sub-RI e
| sub-RI(e1;;e2) = (sub-RI e1 ∨ sub-RI e2)
| sub-RI(if (b) e1 else e2) = (sub-RI b ∨ sub-RI e1 ∨ sub-RI e2)
| sub-RI(while (b) e) = (sub-RI b ∨ sub-RI e)
| sub-RI(throw e) = sub-RI e
| sub-RI(try e1 catch(C V) e2) = (sub-RI e1 ∨ sub-RI e2)
| sub-RI(INIT C (Cs,b) ← e) = True
| sub-RI(RI (C,e);Cs ← e') = True
| sub-RIs(()) = False
| sub-RIs(e#es) = (sub-RI e ∨ sub-RIs es)

```

lemmas *sub-RI-sub-RIs-induct* = *sub-RI.induct sub-RIs.induct*

lemma *nsub-RIs-def[simp]*:

$\neg \text{sub-RIs } es \implies \forall e \in \text{set } es. \neg \text{sub-RI } e$

by(*induct es, auto*)

lemma *sub-RI-base*:

$e = \text{INIT } C (Cs, b) \leftarrow e' \vee e = \text{RI}(C, e_0); Cs \leftarrow e' \implies \text{sub-RI } e$

by(*cases e, auto*)

lemma *nsub-RI-Vals[simp]*: $\neg \text{sub-RIs } (\text{map Val } vs)$

by(*induct vs, auto*)

lemma *lass-val-of-nsub-RI*: $\text{lass-val-of } e = [a] \implies \neg \text{sub-RI } e$

by(*drule lass-val-of-spec, simp*)

— is not currently initializing class C' (point past checking flag)

primrec *not-init* :: $\text{cname} \Rightarrow 'a \text{ exp} \Rightarrow \text{bool}$ **and** *not-inits* :: $\text{cname} \Rightarrow 'a \text{ exp list} \Rightarrow \text{bool}$ **where**

```

  not-init C' (new C) = True
| not-init C' (Cast C e) = not-init C' e
| not-init C' (Val v) = True
| not-init C' (e1 «bop» e2) = (not-init C' e1 ∧ not-init C' e2)
| not-init C' (Var V) = True
| not-init C' (LAss V e) = not-init C' e
| not-init C' (e·F{D}) = not-init C' e
| not-init C' (C·sF{D}) = True
| not-init C' (e1·F{D};=e2) = (not-init C' e1 ∧ not-init C' e2)
| not-init C' (C·sF{D};=e2) = not-init C' e2
| not-init C' (e·M(es)) = (not-init C' e ∧ not-inits C' es)
| not-init C' (C·sM(es)) = not-inits C' es
| not-init C' ({V:T; e}) = not-init C' e
| not-init C' (e1;;e2) = (not-init C' e1 ∧ not-init C' e2)
| not-init C' (if (b) e1 else e2) = (not-init C' b ∧ not-init C' e1 ∧ not-init C' e2)
| not-init C' (while (b) e) = (not-init C' b ∧ not-init C' e)
| not-init C' (throw e) = not-init C' e
| not-init C' (try e1 catch(C V) e2) = (not-init C' e1 ∧ not-init C' e2)
| not-init C' (INIT C (Cs,b) ← e) = ((b → Cs = Nil ∨ C' ≠ hd Cs) ∧ C' ∉ set(tl Cs) ∧ not-init C' e)
| not-init C' (RI (C,e);Cs ← e') = (C' ∉ set(C#Cs) ∧ not-init C' e ∧ not-init C' e')
| not-inits C' (()) = True
| not-inits C' (e#es) = (not-init C' e ∧ not-inits C' es)

```

lemma *not-inits-def'[simp]*:
not-inits C es $\implies \forall e \in \text{set } es. \text{not-init } C e$
by(*induct es, auto*)

lemma *nsub-RIs-not-inits-aux*: $\forall e \in \text{set } es. \neg \text{sub-RI } e \implies \text{not-init } C e$
 $\implies \neg \text{sub-RIs } es \implies \text{not-inits } C es$
by(*induct es, auto*)

lemma *nsub-RI-not-init*: $\neg \text{sub-RI } e \implies \text{not-init } C e$
proof(*induct e*) **qed**(*auto intro: nsub-RIs-not-inits-aux*)

lemma *nsub-RIs-not-inits*: $\neg \text{sub-RIs } es \implies \text{not-inits } C es$
by(*rule nsub-RIs-not-inits-aux*) (*simp-all add: nsub-RI-not-init*)

1.8.5 Subexpressions

primrec *subexp* :: 'a exp \Rightarrow 'a exp set **and** *subexps* :: 'a exp list \Rightarrow 'a exp set **where**
subexp(*new C*) = {}
subexp(*Cast C e*) = {*e*} \cup *subexp e*
subexp(*Val v*) = {}
subexp(*e₁ «bop» e₂*) = {*e₁*, *e₂*} \cup *subexp e₁* \cup *subexp e₂*
subexp(*Var V*) = {}
subexp(*LAss V e*) = {*e*} \cup *subexp e*
subexp(*e.F{D}*) = {*e*} \cup *subexp e*
subexp(*C_sF{D}*) = {}
subexp(*e₁.F{D}:=e₂*) = {*e₁*, *e₂*} \cup *subexp e₁* \cup *subexp e₂*
subexp(*C_sF{D}:=e₂*) = {*e₂*} \cup *subexp e₂*
subexp(*e.M(es)*) = {*e*} \cup *set es* \cup *subexp e* \cup *subexps es*
subexp(*C_sM(es)*) = *set es* \cup *subexps es*
subexp(*{V:T; e}*) = {*e*} \cup *subexp e*
subexp(*e₁::e₂*) = {*e₁*, *e₂*} \cup *subexp e₁* \cup *subexp e₂*
subexp(*if (b) e₁ else e₂*) = {*b*, *e₁*, *e₂*} \cup *subexp b* \cup *subexp e₁* \cup *subexp e₂*
subexp(*while (b) e*) = {*b*, *e*} \cup *subexp b* \cup *subexp e*
subexp(*throw e*) = {*e*} \cup *subexp e*
subexp(*try e₁ catch(C V) e₂*) = {*e₁*, *e₂*} \cup *subexp e₁* \cup *subexp e₂*
subexp(*INIT C (Cs,b) ← e*) = {*e*} \cup *subexp e*
subexp(*RI (C,e);Cs ← e'*) = {*e*, *e'*} \cup *subexp e* \cup *subexp e'*
subexps([]) = {}
subexps(*e#es*) = {*e*} \cup *subexp e* \cup *subexps es*

lemmas *subexp-subexps-induct* = *subexp.induct subexps.induct*

abbreviation *subexp-of* :: 'a exp \Rightarrow 'a exp \Rightarrow bool **where**
subexp-of e e' $\equiv e \in \text{subexp } e'$

lemma *subexp-size-le*:

(*e' ∈ subexp e* $\implies \text{size } e' < \text{size } e$) \wedge (*e' ∈ subexps es* $\implies \text{size } e' < \text{size-list } \text{size } es$)

proof(*induct rule: subexp-subexps.induct*)

case *Call:11* **then show** ?*case* **using** *not-less-eq size-list-estimation* **by** *fastforce*
next

case *SCall:12* **then show** ?*case* **using** *not-less-eq size-list-estimation* **by** *fastforce*
qed(*auto*)

lemma *subexprs-def2*: $\text{subexprs } es = \text{set } es \cup (\bigcup e \in \text{set } es. \text{subexp } e)$ **by**(*induct es, auto*)

— strong induction

lemma shows *subexp-induct*[*consumes 1*]:

$(\bigwedge e. \text{subexp } e = \{\} \implies R e) \implies (\bigwedge e. (\bigwedge e'. e' \in \text{subexp } e \implies R e') \implies R e)$
 $\implies (\bigwedge es. (\bigwedge e'. e' \in \text{subexprs } es \implies R e') \implies R es) \implies (\forall e'. e' \in \text{subexp } e \longrightarrow R e') \wedge R e$

and *subexprs-induct*[*consumes 1*]:

$(\bigwedge es. \text{subexprs } es = \{\} \implies R es) \implies (\bigwedge e. (\bigwedge e'. e' \in \text{subexp } e \implies R e') \implies R e)$
 $\implies (\bigwedge es. (\bigwedge e'. e' \in \text{subexprs } es \implies R e') \implies R es) \implies (\forall e'. e' \in \text{subexprs } es \longrightarrow R e') \wedge R es$

proof(*induct rule: subexp-subexprs-induct*)

case (*Cast* $x1\ x2$)

then have $(\forall e'. \text{subexp-of } e' x2 \longrightarrow R e') \wedge R x2$ **by** *fast*

then have $(\forall e'. \text{subexp-of } e' (\text{Cast } x1\ x2) \longrightarrow R e')$ **by** *auto*

then show *?case* **using** *Cast.prem*s(2) **by** *fast*

next

case (*BinOp* $x1\ x2\ x3$)

then have $(\forall e'. \text{subexp-of } e' x1 \longrightarrow R e') \wedge R x1$ **and** $(\forall e'. \text{subexp-of } e' x3 \longrightarrow R e') \wedge R x3$
by *fast+*

then have $(\forall e'. \text{subexp-of } e' (x1 \llbracket x2 \rrbracket x3) \longrightarrow R e')$ **by** *auto*

then show *?case* **using** *BinOp.prem*s(2) **by** *fast*

next

case (*LAss* $x1\ x2$)

then have $(\forall e'. \text{subexp-of } e' x2 \longrightarrow R e') \wedge R x2$ **by** *fast*

then have $(\forall e'. \text{subexp-of } e' (\text{LAss } x1\ x2) \longrightarrow R e')$ **by** *auto*

then show *?case* **using** *LAss.prem*s(2) **by** *fast*

next

case (*FAcc* $x1\ x2\ x3$)

then have $(\forall e'. \text{subexp-of } e' x1 \longrightarrow R e') \wedge R x1$ **by** *fast*

then have $(\forall e'. \text{subexp-of } e' (x1 \cdot x2 \{x3\}) \longrightarrow R e')$ **by** *auto*

then show *?case* **using** *FAcc.prem*s(2) **by** *fast*

next

case (*FAss* $x1\ x2\ x3\ x4$)

then have $(\forall e'. \text{subexp-of } e' x1 \longrightarrow R e') \wedge R x1$ **and** $(\forall e'. \text{subexp-of } e' x4 \longrightarrow R e') \wedge R x4$
by *fast+*

then have $(\forall e'. \text{subexp-of } e' (x1 \cdot x2 \{x3\} := x4) \longrightarrow R e')$ **by** *auto*

then show *?case* **using** *FAss.prem*s(2) **by** *fast*

next

case (*SFAss* $x1\ x2\ x3\ x4$)

then have $(\forall e'. \text{subexp-of } e' x4 \longrightarrow R e') \wedge R x4$ **by** *fast*

then have $(\forall e'. \text{subexp-of } e' (x1 \cdot_s x2 \{x3\} := x4) \longrightarrow R e')$ **by** *auto*

then show *?case* **using** *SFAss.prem*s(2) **by** *fast*

next

case (*Call* $x1\ x2\ x3$)

then have $(\forall e'. \text{subexp-of } e' x1 \longrightarrow R e') \wedge R x1$ **and** $(\forall e'. e' \in \text{subexprs } x3 \longrightarrow R e') \wedge R s\ x3$
by *fast+*

then have $(\forall e'. \text{subexp-of } e' (x1 \cdot x2(x3)) \longrightarrow R e')$ **using** *subexprs-def2* **by** *auto*

then show *?case* **using** *Call.prem*s(2) **by** *fast*

next

case (*SCall* $x1\ x2\ x3$)

then have $(\forall e'. e' \in \text{subexprs } x3 \longrightarrow R e') \wedge R s\ x3$ **by** *fast*

then have $(\forall e'. \text{subexp-of } e' (x1 \cdot_s x2(x3)) \longrightarrow R e')$ **using** *subexprs-def2* **by** *auto*

then show *?case* **using** *SCall.prem*s(2) **by** *fast*

next

```

case (Block  $x1\ x2\ x3$ )
then have  $(\forall e'. \text{subexp-of } e'\ x3 \longrightarrow R\ e') \wedge R\ x3$  by fast
then have  $(\forall e'. \text{subexp-of } e'\ \{x1;x2;\ x3\} \longrightarrow R\ e')$  by auto
then show ?case using Block.prems(2) by fast
next
case (Seq  $x1\ x2$ )
then have  $(\forall e'. \text{subexp-of } e'\ x1 \longrightarrow R\ e') \wedge R\ x1$  and  $(\forall e'. \text{subexp-of } e'\ x2 \longrightarrow R\ e') \wedge R\ x2$ 
by fast+
then have  $(\forall e'. \text{subexp-of } e'\ (x1;;\ x2) \longrightarrow R\ e')$  by auto
then show ?case using Seq.prems(2) by fast
next
case (Cond  $x1\ x2\ x3$ )
then have  $(\forall e'. \text{subexp-of } e'\ x1 \longrightarrow R\ e') \wedge R\ x1$  and  $(\forall e'. \text{subexp-of } e'\ x2 \longrightarrow R\ e') \wedge R\ x2$ 
and  $(\forall e'. \text{subexp-of } e'\ x3 \longrightarrow R\ e') \wedge R\ x3$  by fast+
then have  $(\forall e'. \text{subexp-of } e'\ (\text{if } (x1)\ x2\ \text{else } x3) \longrightarrow R\ e')$  by auto
then show ?case using Cond.prems(2) by fast
next
case (While  $x1\ x2$ )
then have  $(\forall e'. \text{subexp-of } e'\ x1 \longrightarrow R\ e') \wedge R\ x1$  and  $(\forall e'. \text{subexp-of } e'\ x2 \longrightarrow R\ e') \wedge R\ x2$ 
by fast+
then have  $(\forall e'. \text{subexp-of } e'\ (\text{while } (x1)\ x2) \longrightarrow R\ e')$  by auto
then show ?case using While.prems(2) by fast
next
case (throw  $x$ )
then have  $(\forall e'. \text{subexp-of } e'\ x \longrightarrow R\ e') \wedge R\ x$  by fast
then have  $(\forall e'. \text{subexp-of } e'\ (\text{throw } x) \longrightarrow R\ e')$  by auto
then show ?case using throw.prems(2) by fast
next
case (TryCatch  $x1\ x2\ x3\ x4$ )
then have  $(\forall e'. \text{subexp-of } e'\ x1 \longrightarrow R\ e') \wedge R\ x1$  and  $(\forall e'. \text{subexp-of } e'\ x4 \longrightarrow R\ e') \wedge R\ x4$ 
by fast+
then have  $(\forall e'. \text{subexp-of } e'\ (\text{try } x1\ \text{catch}(x2\ x3)\ x4) \longrightarrow R\ e')$  by auto
then show ?case using TryCatch.prems(2) by fast
next
case (INIT  $x1\ x2\ x3\ x4$ )
then have  $(\forall e'. \text{subexp-of } e'\ x4 \longrightarrow R\ e') \wedge R\ x4$  by fast
then have  $(\forall e'. \text{subexp-of } e'\ (\text{INIT } x1\ (x2,x3) \leftarrow x4) \longrightarrow R\ e')$  by auto
then show ?case using INIT.prems(2) by fast
next
case (RI  $x1\ x2\ x3\ x4$ )
then have  $(\forall e'. \text{subexp-of } e'\ x2 \longrightarrow R\ e') \wedge R\ x2$  and  $(\forall e'. \text{subexp-of } e'\ x4 \longrightarrow R\ e') \wedge R\ x4$ 
by fast+
then have  $(\forall e'. \text{subexp-of } e'\ (\text{RI } (x1,x2) ;\ x3 \leftarrow x4) \longrightarrow R\ e')$  by auto
then show ?case using RI.prems(2) by fast
next
case (Cons-exp  $x1\ x2$ )
then have  $(\forall e'. \text{subexp-of } e'\ x1 \longrightarrow R\ e') \wedge R\ x1$  and  $(\forall e'. e' \in \text{subexps } x2 \longrightarrow R\ e') \wedge R\ x2$ 
by fast+
then have  $(\forall e'. e' \in \text{subexps } (x1 \# x2) \longrightarrow R\ e')$  using subexps-def2 by auto
then show ?case using Cons-exp.prems(3) by fast
qed(auto)

```

1.8.6 Final expressions

definition *final* :: 'a exp \Rightarrow bool

where

final e \equiv ($\exists v. e = \text{Val } v$) \vee ($\exists a. e = \text{Throw } a$)

definition *finals*:: 'a exp list \Rightarrow bool

where

finals es \equiv ($\exists vs. es = \text{map Val } vs$) \vee ($\exists vs a es'. es = \text{map Val } vs @ \text{Throw } a \# es'$)

lemma [*simp*]: *final*(Val v)

lemma [*simp*]: *final*(throw e) = ($\exists a. e = \text{addr } a$)

lemma *finalE*: $\llbracket \text{final } e; \bigwedge v. e = \text{Val } v \Longrightarrow R; \bigwedge a. e = \text{Throw } a \Longrightarrow R \rrbracket \Longrightarrow R$

lemma *final-fv*[*iff*]: *final* e \Longrightarrow fv e = {}

by (auto *simp*: *final-def*)

lemma *finalsE*:

$\llbracket \text{finals } es; \bigwedge vs. es = \text{map Val } vs \Longrightarrow R; \bigwedge vs a es'. es = \text{map Val } vs @ \text{Throw } a \# es' \Longrightarrow R \rrbracket \Longrightarrow R$

lemma [*iff*]: *finals* []

lemma [*iff*]: *finals* (Val v # es) = *finals* es

lemma *finals-app-map*[*iff*]: *finals* (map Val vs @ es) = *finals* es

lemma [*iff*]: *finals* (map Val vs)

lemma [*iff*]: *finals* (throw e # es) = ($\exists a. e = \text{addr } a$)

lemma *not-finals-ConsI*: $\neg \text{final } e \Longrightarrow \neg \text{finals}(e \# es)$

lemma *not-finals-ConsI2*: $e = \text{Val } v \Longrightarrow \neg \text{finals } es \Longrightarrow \neg \text{finals}(e \# es)$

end

1.9 Well-typedness of Jinja expressions

theory WellType

imports ../Common/Objects Expr

begin

type-synonym

env = *vname* \rightarrow *ty*

inductive

WT :: [*J-prog*, *env*, *expr* , *ty*] \Rightarrow bool

($\langle -, \vdash - \rangle$:: \rightarrow [51,51,51]50)

and *WTs* :: [*J-prog*, *env*, *expr list*, *ty list*] \Rightarrow bool

($\langle -, \vdash - [::] \rangle$:: \rightarrow [51,51,51]50)

for *P* :: *J-prog*

where

WTNew:

is-class P C \Longrightarrow

P, E \vdash new C :: Class C

| *WTCast*:

$\llbracket P, E \vdash e :: \text{Class } D; \text{is-class } P C; P \vdash C \preceq^* D \vee P \vdash D \preceq^* C \rrbracket$

$\Longrightarrow P, E \vdash \text{Cast } C e :: \text{Class } C$

- | *WTVal*:
 $\text{typeof } v = \text{Some } T \implies$
 $P, E \vdash \text{Val } v :: T$
- | *WTVar*:
 $E V = \text{Some } T \implies$
 $P, E \vdash \text{Var } V :: T$
- | *WTBinOpEq*:
 $\llbracket P, E \vdash e_1 :: T_1; P, E \vdash e_2 :: T_2; P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1 \rrbracket$
 $\implies P, E \vdash e_1 \langle \text{Eq} \rangle e_2 :: \text{Boolean}$
- | *WTBinOpAdd*:
 $\llbracket P, E \vdash e_1 :: \text{Integer}; P, E \vdash e_2 :: \text{Integer} \rrbracket$
 $\implies P, E \vdash e_1 \langle \text{Add} \rangle e_2 :: \text{Integer}$
- | *WTLAss*:
 $\llbracket E V = \text{Some } T; P, E \vdash e :: T'; P \vdash T' \leq T; V \neq \text{this} \rrbracket$
 $\implies P, E \vdash V := e :: \text{Void}$
- | *WTFAcc*:
 $\llbracket P, E \vdash e :: \text{Class } C; P \vdash C \text{ sees } F, \text{NonStatic}:T \text{ in } D \rrbracket$
 $\implies P, E \vdash e \cdot F\{D\} :: T$
- | *WTSFAcc*:
 $\llbracket P \vdash C \text{ sees } F, \text{Static}:T \text{ in } D \rrbracket$
 $\implies P, E \vdash C \cdot_s F\{D\} :: T$
- | *WTFAss*:
 $\llbracket P, E \vdash e_1 :: \text{Class } C; P \vdash C \text{ sees } F, \text{NonStatic}:T \text{ in } D; P, E \vdash e_2 :: T'; P \vdash T' \leq T \rrbracket$
 $\implies P, E \vdash e_1 \cdot F\{D\} := e_2 :: \text{Void}$
- | *WTSFAss*:
 $\llbracket P \vdash C \text{ sees } F, \text{Static}:T \text{ in } D; P, E \vdash e_2 :: T'; P \vdash T' \leq T \rrbracket$
 $\implies P, E \vdash C \cdot_s F\{D\} := e_2 :: \text{Void}$
- | *WTCall*:
 $\llbracket P, E \vdash e :: \text{Class } C; P \vdash C \text{ sees } M, \text{NonStatic}:Ts \rightarrow T = (\text{pns}, \text{body}) \text{ in } D;$
 $P, E \vdash es [\text{:}] Ts'; P \vdash Ts' [\leq] Ts \rrbracket$
 $\implies P, E \vdash e \cdot M(es) :: T$
- | *WTSCall*:
 $\llbracket P \vdash C \text{ sees } M, \text{Static}:Ts \rightarrow T = (\text{pns}, \text{body}) \text{ in } D;$
 $P, E \vdash es [\text{:}] Ts'; P \vdash Ts' [\leq] Ts; M \neq \text{clinit} \rrbracket$
 $\implies P, E \vdash C \cdot_s M(es) :: T$
- | *WTBlock*:
 $\llbracket \text{is-type } P T; P, E(V \mapsto T) \vdash e :: T' \rrbracket$
 $\implies P, E \vdash \{V:T; e\} :: T'$
- | *WTSeq*:
 $\llbracket P, E \vdash e_1 :: T_1; P, E \vdash e_2 :: T_2 \rrbracket$
 $\implies P, E \vdash e_1 ; e_2 :: T_2$

| *WTCond*:

$$\begin{aligned} & \llbracket P, E \vdash e :: \text{Boolean}; P, E \vdash e_1 :: T_1; P, E \vdash e_2 :: T_2; \\ & \quad P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1; P \vdash T_1 \leq T_2 \longrightarrow T = T_2; P \vdash T_2 \leq T_1 \longrightarrow T = T_1 \rrbracket \\ & \implies P, E \vdash \text{if } (e) e_1 \text{ else } e_2 :: T \end{aligned}$$

| *WTWhile*:

$$\begin{aligned} & \llbracket P, E \vdash e :: \text{Boolean}; P, E \vdash c :: T \rrbracket \\ & \implies P, E \vdash \text{while } (e) c :: \text{Void} \end{aligned}$$

| *WTThrow*:

$$\begin{aligned} & P, E \vdash e :: \text{Class } C \implies \\ & P, E \vdash \text{throw } e :: \text{Void} \end{aligned}$$

| *WTTry*:

$$\begin{aligned} & \llbracket P, E \vdash e_1 :: T; P, E(V \mapsto \text{Class } C) \vdash e_2 :: T; \text{is-class } P C \rrbracket \\ & \implies P, E \vdash \text{try } e_1 \text{ catch}(C V) e_2 :: T \end{aligned}$$

— well-typed expression lists

| *WTNil*:

$$P, E \vdash [] [::] []$$

| *WTCons*:

$$\begin{aligned} & \llbracket P, E \vdash e :: T; P, E \vdash es [::] Ts \rrbracket \\ & \implies P, E \vdash e \# es [::] T \# Ts \end{aligned}$$

lemma *init-nwt* [*simp*]: $\neg P, E \vdash \text{INIT } C (Cs, b) \leftarrow e :: T$

by(*auto elim: WT.cases*)

lemma *ri-nwt* [*simp*]: $\neg P, E \vdash \text{RI}(C, e); Cs \leftarrow e' :: T$

by(*auto elim: WT.cases*)

lemma [*iff*]: $(P, E \vdash [] [::] Ts) = (Ts = [])$

lemma [*iff*]: $(P, E \vdash e \# es [::] T \# Ts) = (P, E \vdash e :: T \wedge P, E \vdash es [::] Ts)$

lemma [*iff*]: $(P, E \vdash (e \# es) [::] Ts) =$

$$(\exists U Us. Ts = U \# Us \wedge P, E \vdash e :: U \wedge P, E \vdash es [::] Us)$$

lemma [*iff*]: $\bigwedge Ts. (P, E \vdash es_1 @ es_2 [::] Ts) =$

$$(\exists Ts_1 Ts_2. Ts = Ts_1 @ Ts_2 \wedge P, E \vdash es_1 [::] Ts_1 \wedge P, E \vdash es_2 [::] Ts_2)$$

lemma [*iff*]: $P, E \vdash \text{Val } v :: T = (\text{typeof } v = \text{Some } T)$

lemma [*iff*]: $P, E \vdash \text{Var } V :: T = (E V = \text{Some } T)$

lemma [*iff*]: $P, E \vdash e_1; e_2 :: T_2 = (\exists T_1. P, E \vdash e_1 :: T_1 \wedge P, E \vdash e_2 :: T_2)$

lemma [*iff*]: $(P, E \vdash \{V: T; e\} :: T') = (\text{is-type } P T \wedge P, E(V \mapsto T) \vdash e :: T')$

lemma *wt-env-mono*:

$$P, E \vdash e :: T \implies (\bigwedge E'. E \subseteq_m E' \implies P, E' \vdash e :: T) \text{ and}$$

$$P, E \vdash es [::] Ts \implies (\bigwedge E'. E \subseteq_m E' \implies P, E' \vdash es [::] Ts)$$

lemma *WT-fv*: $P, E \vdash e :: T \implies \text{fv } e \subseteq \text{dom } E$

and $P, E \vdash es [::] Ts \implies \text{fvs } es \subseteq \text{dom } E$

lemma *WT-nsub-RI*: $P, E \vdash e :: T \implies \neg \text{sub-RI } e$

and *WTs-nsub-RI*s: $P, E \vdash es [::] Ts \implies \neg \text{sub-RI} s \text{ es}$

1.10 Runtime Well-typedness

theory *WellTypeRT*

imports *WellType*

begin

inductive

$WTrt :: J\text{-prog} \Rightarrow \text{heap} \Rightarrow \text{sheap} \Rightarrow \text{env} \Rightarrow \text{expr} \Rightarrow \text{ty} \Rightarrow \text{bool}$

and $WTrts :: J\text{-prog} \Rightarrow \text{heap} \Rightarrow \text{sheap} \Rightarrow \text{env} \Rightarrow \text{expr list} \Rightarrow \text{ty list} \Rightarrow \text{bool}$

and $WTrt2 :: [J\text{-prog}, \text{env}, \text{heap}, \text{sheap}, \text{expr}, \text{ty}] \Rightarrow \text{bool}$

$(\langle -, -, -, - \vdash - : - \rangle \rightarrow [51, 51, 51, 51] 50)$

and $WTrts2 :: [J\text{-prog}, \text{env}, \text{heap}, \text{sheap}, \text{expr list}, \text{ty list}] \Rightarrow \text{bool}$

$(\langle -, -, -, - \vdash - [:] \rightarrow [51, 51, 51, 51] 50)$

for $P :: J\text{-prog}$ **and** $h :: \text{heap}$ **and** $sh :: \text{sheap}$

where

$P, E, h, sh \vdash e : T \equiv WTrt P h sh E e T$

| $P, E, h, sh \vdash es[:] Ts \equiv WTrts P h sh E es Ts$

| *WTrtNew*:

$\text{is-class } P C \Longrightarrow$

$P, E, h, sh \vdash \text{new } C : \text{Class } C$

| *WTrtCast*:

$\llbracket P, E, h, sh \vdash e : T; \text{is-refT } T; \text{is-class } P C \rrbracket$

$\Longrightarrow P, E, h, sh \vdash \text{Cast } C e : \text{Class } C$

| *WTrtVal*:

$\text{typeof}_h v = \text{Some } T \Longrightarrow$

$P, E, h, sh \vdash \text{Val } v : T$

| *WTrtVar*:

$E V = \text{Some } T \Longrightarrow$

$P, E, h, sh \vdash \text{Var } V : T$

| *WTrtBinOpEq*:

$\llbracket P, E, h, sh \vdash e_1 : T_1; P, E, h, sh \vdash e_2 : T_2 \rrbracket$

$\Longrightarrow P, E, h, sh \vdash e_1 \ll\text{Eq}\gg e_2 : \text{Boolean}$

| *WTrtBinOpAdd*:

$\llbracket P, E, h, sh \vdash e_1 : \text{Integer}; P, E, h, sh \vdash e_2 : \text{Integer} \rrbracket$

$\Longrightarrow P, E, h, sh \vdash e_1 \ll\text{Add}\gg e_2 : \text{Integer}$

| *WTrtLAss*:

$\llbracket E V = \text{Some } T; P, E, h, sh \vdash e : T'; P \vdash T' \leq T \rrbracket$

$\Longrightarrow P, E, h, sh \vdash V := e : \text{Void}$

| *WTrtFAcc*:

$\llbracket P, E, h, sh \vdash e : \text{Class } C; P \vdash C \text{ has } F, \text{NonStatic}: T \text{ in } D \rrbracket \Longrightarrow$

$P, E, h, sh \vdash e \cdot F\{D\} : T$

| *WTrtFAccNT*:

$P, E, h, sh \vdash e : NT \Longrightarrow$

$P, E, h, sh \vdash e \cdot F\{D\} : T$

- | *WTrtSFAcc*:

$$\llbracket P \vdash C \text{ has } F, \text{Static}:T \text{ in } D \rrbracket \implies P, E, h, sh \vdash C \cdot_s F\{D\} : T$$
- | *WTrtFAss*:

$$\llbracket P, E, h, sh \vdash e_1 : \text{Class } C; P \vdash C \text{ has } F, \text{NonStatic}:T \text{ in } D; P, E, h, sh \vdash e_2 : T_2; P \vdash T_2 \leq T \rrbracket \implies P, E, h, sh \vdash e_1 \cdot F\{D\} := e_2 : \text{Void}$$
- | *WTrtFAssNT*:

$$\llbracket P, E, h, sh \vdash e_1 : NT; P, E, h, sh \vdash e_2 : T_2 \rrbracket \implies P, E, h, sh \vdash e_1 \cdot F\{D\} := e_2 : \text{Void}$$
- | *WTrtSFAss*:

$$\llbracket P, E, h, sh \vdash e_2 : T_2; P \vdash C \text{ has } F, \text{Static}:T \text{ in } D; P \vdash T_2 \leq T \rrbracket \implies P, E, h, sh \vdash C \cdot_s F\{D\} := e_2 : \text{Void}$$
- | *WTrtCall*:

$$\llbracket P, E, h, sh \vdash e : \text{Class } C; P \vdash C \text{ sees } M, \text{NonStatic}:Ts \rightarrow T = (pns, body) \text{ in } D; P, E, h, sh \vdash es \text{ [;} Ts'; P \vdash Ts' [\leq] Ts \rrbracket \implies P, E, h, sh \vdash e \cdot M(es) : T$$
- | *WTrtCallNT*:

$$\llbracket P, E, h, sh \vdash e : NT; P, E, h, sh \vdash es \text{ [;} Ts \rrbracket \implies P, E, h, sh \vdash e \cdot M(es) : T$$
- | *WTrtSCall*:

$$\llbracket P \vdash C \text{ sees } M, \text{Static}:Ts \rightarrow T = (pns, body) \text{ in } D; P, E, h, sh \vdash es \text{ [;} Ts'; P \vdash Ts' [\leq] Ts; M = \text{clinit} \rightarrow sh D = [(sfs, Processing)] \wedge es = \text{map Val vs} \rrbracket \implies P, E, h, sh \vdash C \cdot_s M(es) : T$$
- | *WTrtBlock*:

$$P, E(V \mapsto T), h, sh \vdash e : T' \implies P, E, h, sh \vdash \{V:T; e\} : T'$$
- | *WTrtSeq*:

$$\llbracket P, E, h, sh \vdash e_1 : T_1; P, E, h, sh \vdash e_2 : T_2 \rrbracket \implies P, E, h, sh \vdash e_1 ; e_2 : T_2$$
- | *WTrtCond*:

$$\llbracket P, E, h, sh \vdash e : \text{Boolean}; P, E, h, sh \vdash e_1 : T_1; P, E, h, sh \vdash e_2 : T_2; P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1; P \vdash T_1 \leq T_2 \rightarrow T = T_2; P \vdash T_2 \leq T_1 \rightarrow T = T_1 \rrbracket \implies P, E, h, sh \vdash \text{if } (e) e_1 \text{ else } e_2 : T$$
- | *WTrtWhile*:

$$\llbracket P, E, h, sh \vdash e : \text{Boolean}; P, E, h, sh \vdash c : T \rrbracket \implies P, E, h, sh \vdash \text{while}(e) c : \text{Void}$$
- | *WTrtThrow*:

$$\llbracket P, E, h, sh \vdash e : T_r; \text{is-refT } T_r \rrbracket \implies P, E, h, sh \vdash \text{throw } e : T$$
- | *WTrtTry*:

$$\begin{aligned} & \llbracket P, E, h, sh \vdash e_1 : T_1; P, E(V \mapsto \text{Class } C), h, sh \vdash e_2 : T_2; P \vdash T_1 \leq T_2 \rrbracket \\ & \implies P, E, h, sh \vdash \text{try } e_1 \text{ catch}(C \ V) \ e_2 : T_2 \end{aligned}$$

| *WTrtInit*:

$$\begin{aligned} & \llbracket P, E, h, sh \vdash e : T; \forall C' \in \text{set } (C \# Cs). \text{is-class } P \ C'; \neg \text{sub-RI } e; \\ & \quad \forall C' \in \text{set } (tl \ Cs). \exists \text{sfs. sh } C' = \llbracket (\text{sfs}, \text{Processing}) \rrbracket; \\ & \quad b \longrightarrow (\forall C' \in \text{set } Cs. \exists \text{sfs. sh } C' = \llbracket (\text{sfs}, \text{Processing}) \rrbracket); \\ & \quad \text{distinct } Cs; \text{supercls-lst } P \ Cs \rrbracket \\ & \implies P, E, h, sh \vdash \text{INIT } C \ (Cs, b) \leftarrow e : T \end{aligned}$$

| *WTrtRI*:

$$\begin{aligned} & \llbracket P, E, h, sh \vdash e : T; P, E, h, sh \vdash e' : T'; \forall C' \in \text{set } (C \# Cs). \text{is-class } P \ C'; \neg \text{sub-RI } e'; \\ & \quad \forall C' \in \text{set } (C \# Cs). \text{not-init } C' \ e'; \\ & \quad \forall C' \in \text{set } Cs. \exists \text{sfs. sh } C' = \llbracket (\text{sfs}, \text{Processing}) \rrbracket; \\ & \quad \exists \text{sfs. sh } C = \llbracket (\text{sfs}, \text{Processing}) \rrbracket \vee (\text{sh } C = \llbracket (\text{sfs}, \text{Error}) \rrbracket \wedge e = \text{THROW NoClassDefFoundError}); \\ & \quad \text{distinct } (C \# Cs); \text{supercls-lst } P \ (C \# Cs) \rrbracket \\ & \implies P, E, h, sh \vdash \text{RI}(C, e); Cs \leftarrow e' : T' \end{aligned}$$

— well-typed expression lists

| *WTrtNil*:

$$P, E, h, sh \vdash [] \ [:] \ []$$

| *WTrtCons*:

$$\begin{aligned} & \llbracket P, E, h, sh \vdash e : T; P, E, h, sh \vdash es \ [:] \ Ts \rrbracket \\ & \implies P, E, h, sh \vdash e \# es \ [:] \ T \# Ts \end{aligned}$$

1.10.1 Easy consequences

lemma [*iff*]: $(P, E, h, sh \vdash [] \ [:] \ Ts) = (Ts = [])$

lemma [*iff*]: $(P, E, h, sh \vdash e \# es \ [:] \ T \# Ts) = (P, E, h, sh \vdash e : T \wedge P, E, h, sh \vdash es \ [:] \ Ts)$

lemma [*iff*]: $(P, E, h, sh \vdash (e \# es) \ [:] \ Ts) =$
 $(\exists U \ Us. Ts = U \# Us \wedge P, E, h, sh \vdash e : U \wedge P, E, h, sh \vdash es \ [:] \ Us)$

lemma [*simp*]: $\forall Ts. (P, E, h, sh \vdash es_1 \ @ \ es_2 \ [:] \ Ts) =$
 $(\exists Ts_1 \ Ts_2. Ts = Ts_1 \ @ \ Ts_2 \wedge P, E, h, sh \vdash es_1 \ [:] \ Ts_1 \ \& \ P, E, h, sh \vdash es_2 \ [:] \ Ts_2)$

lemma [*iff*]: $P, E, h, sh \vdash \text{Val } v : T = (\text{typeof}_h \ v = \text{Some } T)$

lemma [*iff*]: $P, E, h, sh \vdash \text{Var } v : T = (E \ v = \text{Some } T)$

lemma [*iff*]: $P, E, h, sh \vdash e_1; e_2 : T_2 = (\exists T_1. P, E, h, sh \vdash e_1 : T_1 \wedge P, E, h, sh \vdash e_2 : T_2)$

lemma [*iff*]: $P, E, h, sh \vdash \{V : T; e\} : T' = (P, E(V \mapsto T), h, sh \vdash e : T')$

1.10.2 Some interesting lemmas

lemma *WTrts-Val[simp]*:

$$\bigwedge Ts. (P, E, h, sh \vdash \text{map Val } vs \ [:] \ Ts) = (\text{map } (\text{typeof}_h) \ vs = \text{map Some } Ts)$$

lemma *WTrts-same-length*: $\bigwedge Ts. P, E, h, sh \vdash es \ [:] \ Ts \implies \text{length } es = \text{length } Ts$

lemma *WTrt-env-mono*:

$$\begin{aligned} & P, E, h, sh \vdash e : T \implies (\bigwedge E'. E \subseteq_m E' \implies P, E', h, sh \vdash e : T) \ \mathbf{and} \\ & P, E, h, sh \vdash es \ [:] \ Ts \implies (\bigwedge E'. E \subseteq_m E' \implies P, E', h, sh \vdash es \ [:] \ Ts) \end{aligned}$$

lemma *WTrt-hext-mono*: $P, E, h, sh \vdash e : T \implies h \leq h' \implies P, E, h', sh \vdash e : T$

and *WTrts-hext-mono*: $P, E, h, sh \vdash es \ [:] \ Ts \implies h \leq h' \implies P, E, h', sh \vdash es \ [:] \ Ts$

lemma *WTrt-shext-mono*: $P, E, h, sh \vdash e : T \implies sh \sqsubseteq_s sh' \implies \neg \text{sub-RI } e \implies P, E, h, sh' \vdash e : T$
and *WTrts-shext-mono*: $P, E, h, sh \vdash es [:] Ts \implies sh \sqsubseteq_s sh' \implies \neg \text{sub-RIs } es \implies P, E, h, sh' \vdash es [:] Ts$

lemma *WTrt-heat-shext-mono*: $P, E, h, sh \vdash e : T$
 $\implies h \sqsubseteq h' \implies sh \sqsubseteq_s sh' \implies \neg \text{sub-RI } e \implies P, E, h', sh' \vdash e : T$
by(*auto intro: WTrt-heat-mono WTrt-shext-mono*)

lemma *WTrts-heat-shext-mono*: $P, E, h, sh \vdash es [:] Ts$
 $\implies h \sqsubseteq h' \implies sh \sqsubseteq_s sh' \implies \neg \text{sub-RIs } es \implies P, E, h', sh' \vdash es [:] Ts$
by(*auto intro: WTrts-heat-mono WTrts-shext-mono*)

lemma *WT-implies-WTrt*: $P, E \vdash e :: T \implies P, E, h, sh \vdash e : T$
and *WTs-implies-WTrts*: $P, E \vdash es [::] Ts \implies P, E, h, sh \vdash es [:] Ts$
end

1.11 Program State

theory *State* **imports** *../Common/Exceptions* **begin**

type-synonym

locals = *vname* \rightarrow *val* — local vars, incl. params and “this”

type-synonym

state = *heap* \times *locals* \times *sheap*

definition *hp* :: *state* \Rightarrow *heap*

where

hp \equiv *fst*

definition *lcl* :: *state* \Rightarrow *locals*

where

lcl \equiv *fst* \circ *snd*

definition *shp* :: *state* \Rightarrow *sheap*

where

shp \equiv *snd* \circ *snd*

end

1.12 System Classes

theory *SystemClasses*

imports *Decl Exceptions*

begin

This theory provides definitions for the *Object* class, and the system exceptions.

definition *ObjectC* :: 'm *cdecl*

where

ObjectC \equiv (*Object*, (*undefined*, [], []))

definition *NullPointerC* :: 'm *cdecl*

where

NullPointerC \equiv (*NullPointer*, (*Object*, [], []))

definition *ClassCastC* :: 'm *cdecl*

where

$ClassCastC \equiv (ClassCast, (Object, [], []))$

definition $OutOfMemoryC :: 'm cdecl$

where

$OutOfMemoryC \equiv (OutOfMemory, (Object, [], []))$

definition $NoClassDefFoundC :: 'm cdecl$

where

$NoClassDefFoundC \equiv (NoClassDefFoundError, (Object, [], []))$

definition $IncompatibleClassChangeC :: 'm cdecl$

where

$IncompatibleClassChangeC \equiv (IncompatibleClassChangeError, (Object, [], []))$

definition $NoSuchFieldC :: 'm cdecl$

where

$NoSuchFieldC \equiv (NoSuchFieldError, (Object, [], []))$

definition $NoSuchMethodC :: 'm cdecl$

where

$NoSuchMethodC \equiv (NoSuchMethodError, (Object, [], []))$

definition $SystemClasses :: 'm cdecl list$

where

$SystemClasses \equiv [ObjectC, NullPointerExceptionC, ClassCastC, OutOfMemoryC, NoClassDefFoundC, IncompatibleClassChangeC, NoSuchFieldC, NoSuchMethodC]$

end

1.13 Generic Well-formedness of programs

theory $WellForm$ **imports** $TypeRel SystemClasses$ **begin**

This theory defines global well-formedness conditions for programs but does not look inside method bodies. Hence it works for both Jinja and JVM programs. Well-typing of expressions is defined elsewhere (in theory $WellType$).

Because Jinja does not have method overloading, its policy for method overriding is the classical one: *covariant in the result type but contravariant in the argument types*. This means the result type of the overriding method becomes more specific, the argument types become more general.

type-synonym $'m wf-mdecl-test = 'm prog \Rightarrow cname \Rightarrow 'm mdecl \Rightarrow bool$

definition $wf-fdecl :: 'm prog \Rightarrow fdecl \Rightarrow bool$

where

$wf-fdecl P \equiv \lambda(F, b, T). is-type P T$

definition $wf-mdecl :: 'm wf-mdecl-test \Rightarrow 'm wf-mdecl-test$

where

$wf-mdecl wf-md P C \equiv \lambda(M, b, Ts, T, m).$

$(\forall T \in set Ts. is-type P T) \wedge is-type P T \wedge wf-md P C (M, b, Ts, T, m)$

definition $wf-clinit :: 'm mdecl list \Rightarrow bool$ **where**

$wf-clinit ms = (\exists m. (clinit, Static, [], Void, m) \in set ms)$

definition $wf\text{-}cdecl :: 'm\ wf\text{-}mdecl\text{-}test \Rightarrow 'm\ prog \Rightarrow 'm\ cdecl \Rightarrow bool$

where

$$\begin{aligned} wf\text{-}cdecl\ wf\text{-}md\ P &\equiv \lambda(C,(D,fs,ms)). \\ &(\forall f \in set\ fs.\ wf\text{-}fdecl\ P\ f) \wedge distinct\text{-}fst\ fs \wedge \\ &(\forall m \in set\ ms.\ wf\text{-}mdecl\ wf\text{-}md\ P\ C\ m) \wedge distinct\text{-}fst\ ms \wedge \\ &(C \neq Object \longrightarrow \\ &is\text{-}class\ P\ D \wedge \neg P \vdash D \preceq^* C \wedge \\ &(\forall (M,b,Ts,T,m) \in set\ ms. \\ &\quad \forall D'\ b'\ Ts'\ T'\ m'. P \vdash D\ sees\ M,b':Ts' \rightarrow T' = m' \text{ in } D' \longrightarrow \\ &\quad b = b' \wedge P \vdash Ts' [\leq] Ts \wedge P \vdash T \leq T')) \wedge \\ &wf\text{-}clinit\ ms \end{aligned}$$

definition $wf\text{-}syscls :: 'm\ prog \Rightarrow bool$

where

$$wf\text{-}syscls\ P \equiv \{Object\} \cup sys\text{-}xcpts \subseteq set(map\ fst\ P)$$

definition $wf\text{-}prog :: 'm\ wf\text{-}mdecl\text{-}test \Rightarrow 'm\ prog \Rightarrow bool$

where

$$wf\text{-}prog\ wf\text{-}md\ P \equiv wf\text{-}syscls\ P \wedge (\forall c \in set\ P.\ wf\text{-}cdecl\ wf\text{-}md\ P\ c) \wedge distinct\text{-}fst\ P$$

1.13.1 Well-formedness lemmas

lemma $class\text{-}wf$:

$$\llbracket class\ P\ C = Some\ c; wf\text{-}prog\ wf\text{-}md\ P \rrbracket \Longrightarrow wf\text{-}cdecl\ wf\text{-}md\ P\ (C,c)$$

lemma $class\text{-}Object$ [simp]:

$$wf\text{-}prog\ wf\text{-}md\ P \Longrightarrow \exists C\ fs\ ms.\ class\ P\ Object = Some\ (C,fs,ms)$$

lemma $is\text{-}class\text{-}Object$ [simp]:

$$wf\text{-}prog\ wf\text{-}md\ P \Longrightarrow is\text{-}class\ P\ Object$$

lemma $is\text{-}class\text{-}supclass$:

assumes wf : $wf\text{-}prog\ wf\text{-}md\ P$ **and** sub : $P \vdash C \preceq^* D$

shows $is\text{-}class\ P\ C \Longrightarrow is\text{-}class\ P\ D$

lemma $is\text{-}class\text{-}xcpt$:

$$\llbracket C \in sys\text{-}xcpts; wf\text{-}prog\ wf\text{-}md\ P \rrbracket \Longrightarrow is\text{-}class\ P\ C$$

lemma $subcls1\text{-}wfD$:

assumes $sub1$: $P \vdash C \prec^1 D$ **and** wf : $wf\text{-}prog\ wf\text{-}md\ P$

shows $D \neq C \wedge (D,C) \notin (subcls1\ P)^+$

lemma $wf\text{-}cdecl\text{-}supD$:

$$\llbracket wf\text{-}cdecl\ wf\text{-}md\ P\ (C,D,r); C \neq Object \rrbracket \Longrightarrow is\text{-}class\ P\ D$$

lemma $subcls\text{-}asym$:

$$\llbracket wf\text{-}prog\ wf\text{-}md\ P; (C,D) \in (subcls1\ P)^+ \rrbracket \Longrightarrow (D,C) \notin (subcls1\ P)^+$$

lemma $subcls\text{-}irrefl$:

$$\llbracket wf\text{-}prog\ wf\text{-}md\ P; (C,D) \in (subcls1\ P)^+ \rrbracket \Longrightarrow C \neq D$$

lemma $acyclic\text{-}subcls1$:

$$wf\text{-}prog\ wf\text{-}md\ P \Longrightarrow acyclic\ (subcls1\ P)$$

lemma $wf\text{-}subcls1$:

$wf\text{-prog } wf\text{-md } P \implies wf \ ((subcls1\ P)^{-1})$

lemma *single-valued-subcls1*:

$wf\text{-prog } wf\text{-md } G \implies single\text{-valued } (subcls1\ G)$

lemma *subcls-induct*:

$\llbracket wf\text{-prog } wf\text{-md } P; \bigwedge C. \forall D. (C,D) \in (subcls1\ P)^+ \longrightarrow Q\ D \implies Q\ C \rrbracket \implies Q\ C$

lemma *subcls1-induct-aux*:

assumes *is-class* $P\ C$ **and** $wf: wf\text{-prog } wf\text{-md } P$ **and** $QObj: Q\ Object$

shows

$\llbracket \bigwedge C\ D\ fs\ ms. \llbracket C \neq Object; is\text{-class } P\ C; class\ P\ C = Some\ (D,fs,ms) \wedge wf\text{-cdecl } wf\text{-md } P\ (C,D,fs,ms) \wedge P \vdash C \prec^1 D \wedge is\text{-class } P\ D \wedge Q\ D \rrbracket \implies Q\ C \rrbracket \implies Q\ C$

lemma *subcls1-induct* [*consumes 2, case-names Object Subcls*]:

$\llbracket wf\text{-prog } wf\text{-md } P; is\text{-class } P\ C; Q\ Object; \bigwedge C\ D. \llbracket C \neq Object; P \vdash C \prec^1 D; is\text{-class } P\ D; Q\ D \rrbracket \implies Q\ C \rrbracket \implies Q\ C$

lemma *subcls-C-Object*:

assumes *class: is-class* $P\ C$ **and** $wf: wf\text{-prog } wf\text{-md } P$

shows $P \vdash C \preceq^* Object$

lemma *is-type-pTs*:

assumes $wf\text{-prog } wf\text{-md } P$ **and** $(C,S,fs,ms) \in set\ P$ **and** $(M,b,Ts,T,m) \in set\ ms$

shows $set\ Ts \subseteq types\ P$

lemma *wf-supercls-distinct-app*:

assumes $wf: wf\text{-prog } wf\text{-md } P$

and $nObj: C \neq Object$ **and** $cls: class\ P\ C = \llbracket (D, fs, ms) \rrbracket$

and $super: supercls\text{-lst } P\ (C\#Cs)$ **and** $dist: distinct\ (C\#Cs)$

shows $distinct\ (D\#C\#Cs)$

proof –

have $\neg P \vdash D \preceq^* C$ **using** $subcls1\text{-wfD}[OF\ subcls1I[OF\ cls\ nObj]\ wf]$

by (*simp add: rtrancl-eq-or-trancl*)

then show *?thesis* **using** *assms* **by** *auto*

qed

1.13.2 Well-formedness and method lookup

lemma *sees-wf-mdecl*:

assumes $wf: wf\text{-prog } wf\text{-md } P$ **and** $sees: P \vdash C\ sees\ M,b:Ts \rightarrow T = m\ in\ D$

shows $wf\text{-mdecl } wf\text{-md } P\ D\ (M,b,Ts,T,m)$

lemma *sees-method-mono* [*rule-format (no-asm)*]:

assumes $sub: P \vdash C' \preceq^* C$ **and** $wf: wf\text{-prog } wf\text{-md } P$

shows $\forall D\ b\ Ts\ T\ m. P \vdash C\ sees\ M,b:Ts \rightarrow T = m\ in\ D \longrightarrow$

$(\exists D'\ Ts'\ T'\ m'. P \vdash C'\ sees\ M,b:Ts' \rightarrow T' = m'\ in\ D' \wedge P \vdash Ts \llbracket \leq \rrbracket Ts' \wedge P \vdash T' \leq T)$

lemma *sees-method-mono2*:

$\llbracket P \vdash C' \preceq^* C; wf\text{-prog } wf\text{-md } P;$

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

$\implies b = b' \wedge P \vdash Ts \llbracket \leq \rrbracket Ts' \wedge P \vdash T' \leq T$

lemma *mdecls-visible*:

assumes $wf: wf\text{-prog } wf\text{-md } P$ **and** $class: is\text{-class } P C$

shows $\bigwedge D fs ms. class P C = Some(D,fs,ms)$

$\implies \exists Mm. P \vdash C \text{ sees-methods } Mm \wedge (\forall (M,b,Ts,T,m) \in set\ ms. Mm\ M = Some((b,Ts,T,m),C))$

lemma *mdecl-visible*:

assumes $wf: wf\text{-prog } wf\text{-md } P$ **and** $C: (C,S,fs,ms) \in set\ P$ **and** $m: (M,b,Ts,T,m) \in set\ ms$

shows $P \vdash C \text{ sees } M,b:Ts \rightarrow T = m \text{ in } C$

lemma *Call-lemma*:

assumes $sees: P \vdash C \text{ sees } M,b:Ts \rightarrow T = m \text{ in } D$ **and** $sub: P \vdash C' \preceq^* C$ **and** $wf: wf\text{-prog } wf\text{-md } P$

shows $\exists D' Ts' T' m'$.

$P \vdash C' \text{ sees } M,b:Ts' \rightarrow T' = m' \text{ in } D' \wedge P \vdash Ts \preceq Ts' \wedge P \vdash T' \preceq T \wedge P \vdash C' \preceq^* D'$
 $\wedge is\text{-type } P T' \wedge (\forall T \in set\ Ts'. is\text{-type } P T) \wedge wf\text{-md } P D' (M,b,Ts',T',m')$

lemma *wf-prog-lift*:

assumes $wf: wf\text{-prog } (\lambda P C bd. A P C bd) P$

and *rule*:

$\bigwedge wf\text{-md } C M b Ts C T m bd.$

$wf\text{-prog } wf\text{-md } P \implies$

$P \vdash C \text{ sees } M,b:Ts \rightarrow T = m \text{ in } C \implies$

$set\ Ts \subseteq types\ P \implies$

$bd = (M,b,Ts,T,m) \implies$

$A P C bd \implies$

$B P C bd$

shows $wf\text{-prog } (\lambda P C bd. B P C bd) P$

lemma *wf-sees-clinit*:

assumes $wf: wf\text{-prog } wf\text{-md } P$ **and** $ex: class\ P\ C = Some\ a$

shows $\exists m. P \vdash C \text{ sees } clinit,Static:[] \rightarrow Void = m \text{ in } C$

proof –

from ex **obtain** $D fs ms$ **where** $a = (D,fs,ms)$ **by** (*cases a*)

then **have** $sP: (C, D, fs, ms) \in set\ P$ **using** $ex\ map\ of\ SomeD[of\ P\ C\ a]$ **by** (*simp add: class-def*)

then **have** $wf\text{-clinit } ms$ **using** $assms$ **by** (*unfold wf-prog-def wf-cdecl-def, auto*)

then **obtain** m **where** $sm: (clinit, Static, [], Void, m) \in set\ ms$ **by** (*meson wf-clinit-def*)

then **have** $P \vdash C \text{ sees } clinit,Static:[] \rightarrow Void = m \text{ in } C$

using $mdecl\ visible[OF\ wf\ sP\ sm]$ **by** *simp*

then **show** *?thesis* **by** (*rule exI*)

qed

lemma *wf-sees-clinit1*:

assumes $wf: wf\text{-prog } wf\text{-md } P$ **and** $ex: class\ P\ C = Some\ a$

and $P \vdash C \text{ sees } clinit,b:Ts \rightarrow T = m \text{ in } D$

shows $b = Static \wedge Ts = [] \wedge T = Void \wedge D = C$

proof –

obtain m' **where** $sees: P \vdash C \text{ sees } clinit,Static:[] \rightarrow Void = m' \text{ in } C$

using $wf\text{-sees-clinit}[OF\ wf\ ex]$ **by** *clarify*

then **show** *?thesis* **using** $sees\ wf$ **by** (*meson assms(3) sees-method-fun*)

qed

lemma *wf-NonStatic-nclinit*:

assumes $wf: wf\text{-prog } wf\text{-md } P$ **and** $meth: P \vdash C \text{ sees } M,NonStatic:Ts \rightarrow T = (m\ x\ s, m\ x\ l, ins, xt)$ **in** D

shows $M \neq clinit$

proof –

from $sees\ method\ is\ class[OF\ meth]$ **obtain** a **where** $cls: class\ P\ C = Some\ a$

by (*clarsimp simp: is-class-def*)

with *wf wf-sees-clinit*[*OF wf cls*]
obtain *m* **where** $P \vdash C \text{ sees } \text{clinit}, \text{Static}: [] \rightarrow \text{Void} = m \text{ in } C$ **by** *clarsimp*
with *meth* **show** *?thesis* **by**(*auto dest: sees-method-fun*)
qed

1.13.3 Well-formedness and field lookup

lemma *wf-Fields-Ex*:
assumes *wf: wf-prog wf-md P* **and** *is-class P C*
shows $\exists \text{FDTs}. P \vdash C \text{ has-fields FDTs}$

lemma *has-fields-types*:
 $\llbracket P \vdash C \text{ has-fields FDTs}; (FD, b, T) \in \text{set FDTs}; \text{wf-prog wf-md } P \rrbracket \implies \text{is-type } P \ T$
lemma *sees-field-is-type*:
 $\llbracket P \vdash C \text{ sees } F, b: T \text{ in } D; \text{wf-prog wf-md } P \rrbracket \implies \text{is-type } P \ T$

lemma *wf-syscls*:
 $\text{set SystemClasses} \subseteq \text{set } P \implies \text{wf-syscls } P$

1.13.4 Well-formedness and subclassing

lemma *wf-subcls-nCls*:
assumes *wf: wf-prog wf-md P* **and** *ns: $\neg \text{is-class } P \ C$*
shows $\llbracket P \vdash D \preceq^* D'; D \neq C \rrbracket \implies D' \neq C$
proof(*induct rule: rtrancl.induct*)
case (*rtrancl-into-rtrancl a b c*)
with *ns* **show** *?case* **by**(*clarsimp dest!: subcls1D wf-cdecl-supD[OF class-wf[OF - wf]]*)
qed(*simp*)

lemma *wf-subcls-nCls'*:
assumes *wf: wf-prog wf-md P* **and** *ns: $\neg \text{is-class } P \ C_0$*
shows $\bigwedge cd \ D'. cd \in \text{set } P \implies \neg P \vdash \text{fst } cd \preceq^* C_0$
proof –
fix *cd D'* **assume** *cd: cd \in set P*
then **have** *cls: is-class P (fst cd)* **using** *class-exists-equiv is-class-def* **by** *blast*
with *wf-subcls-nCls[OF wf ns]* *ns* **show** $\neg P \vdash \text{fst } cd \preceq^* C_0$ **by**(*cases fst cd = D', auto*)
qed

lemma *wf-nclass-nsub*:
 $\llbracket \text{wf-prog wf-md } P; \text{is-class } P \ C; \neg \text{is-class } P \ C' \rrbracket \implies \neg P \vdash C \preceq^* C'$
by(*rule notI, auto dest: wf-subcls-nCls[where C=C' and D=C]*)

lemma *wf-sys-xcpt-nsub-Start*:
assumes *wf: wf-prog wf-md P* **and** *ns: $\neg \text{is-class } P \ \text{Start}$* **and** *sx: C \in sys-xcpts*
shows $\neg P \vdash C \preceq^* \text{Start}$
proof –
have *Cns: C \neq Start* **using** *Start-nsys-xcpts sx* **by** *clarsimp*
show *?thesis* **using** *wf-subcls-nCls[OF wf ns - Cns]* **by** *auto*
qed

end

1.14 Weak well-formedness of Jinja programs

theory *WWellForm* **imports** *../Common/WellForm Expr* **begin**

definition *wwf-J-mdecl* :: *J-prog* \Rightarrow *cname* \Rightarrow *J-mb mdecl* \Rightarrow *bool*

where

wwf-J-mdecl *P C* \equiv $\lambda(M,b,Ts,T,(pns,body)).$

length *Ts* = *length* *pns* \wedge *distinct* *pns* \wedge \neg *sub-RI* *body* \wedge

(*case* *b* of

NonStatic \Rightarrow *this* \notin *set* *pns* \wedge *fv* *body* \subseteq {*this*} \cup *set* *pns*

| *Static* \Rightarrow *fv* *body* \subseteq *set* *pns*)

lemma *wwf-J-mdecl-NonStatic[simp]*:

wwf-J-mdecl *P C* (*M,NonStatic,Ts,T,pns,body*) =

(*length* *Ts* = *length* *pns* \wedge *distinct* *pns* \wedge \neg *sub-RI* *body* \wedge *this* \notin *set* *pns* \wedge *fv* *body* \subseteq {*this*} \cup *set* *pns*)

lemma *wwf-J-mdecl-Static[simp]*:

wwf-J-mdecl *P C* (*M,Static,Ts,T,pns,body*) =

(*length* *Ts* = *length* *pns* \wedge *distinct* *pns* \wedge \neg *sub-RI* *body* \wedge *fv* *body* \subseteq *set* *pns*)

abbreviation

wwf-J-prog :: *J-prog* \Rightarrow *bool* **where**

wwf-J-prog \equiv *wf-prog* *wwf-J-mdecl*

lemma *sees-wwf-nsub-RI*:

\llbracket *wwf-J-prog* *P*; *P* \vdash *C* *sees* *M,b : Ts* \rightarrow *T = (pns, body)* *in D* $\rrbracket \Longrightarrow \neg$ *sub-RI* *body*

end

1.15 Big Step Semantics

theory *BigStep* **imports** *Expr State WWellForm* **begin**

inductive

eval :: *J-prog* \Rightarrow *expr* \Rightarrow *state* \Rightarrow *expr* \Rightarrow *state* \Rightarrow *bool*

($\langle \cdot \vdash ((1 \langle \cdot, / - \rangle) \Rightarrow / (1 \langle \cdot, / - \rangle)) \rangle$ [51,0,0,0,0] 81)

and *evals* :: *J-prog* \Rightarrow *expr list* \Rightarrow *state* \Rightarrow *expr list* \Rightarrow *state* \Rightarrow *bool*

($\langle \cdot \vdash ((1 \langle \cdot, / - \rangle) [\Rightarrow] / (1 \langle \cdot, / - \rangle)) \rangle$ [51,0,0,0,0] 81)

for *P* :: *J-prog*

where

New:

\llbracket *sh* *C* = *Some* (*sfs, Done*); *new-Addr* *h* = *Some* *a*;

P \vdash *C* *has-fields* *FDTs*; *h'* = *h*(*a* \rightarrow *blank* *P C*) \rrbracket

\Longrightarrow *P* \vdash \langle *new* *C*, (*h*, *l*, *sh*) $\rangle \Rightarrow$ \langle *addr* *a*, (*h'*, *l*, *sh*) \rangle

| *NewFail*:

\llbracket *sh* *C* = *Some* (*sfs, Done*); *new-Addr* *h* = *None*; *is-class* *P C* $\rrbracket \Longrightarrow$

P \vdash \langle *new* *C*, (*h*, *l*, *sh*) $\rangle \Rightarrow$ \langle *THROW* *OutOfMemory*, (*h*, *l*, *sh*) \rangle

| *NewInit*:

\llbracket \nexists *sfs*. *sh* *C* = *Some* (*sfs, Done*); *P* \vdash \langle *INIT* *C* (*[C]*, *False*) \leftarrow *unit*, (*h*, *l*, *sh*) $\rangle \Rightarrow$ \langle *Val* *v'*, (*h'*, *l'*, *sh'*) \rangle ;

new-Addr *h'* = *Some* *a*; *P* \vdash *C* *has-fields* *FDTs*; *h''* = *h'*(*a* \rightarrow *blank* *P C*) \rrbracket

\Longrightarrow *P* \vdash \langle *new* *C*, (*h*, *l*, *sh*) $\rangle \Rightarrow$ \langle *addr* *a*, (*h''*, *l'*, *sh'*) \rangle

| *NewInitOOM*:

$$\begin{aligned} & \llbracket \# \text{sfs. sh } C = \text{Some}(\text{sfs}, \text{Done}); P \vdash \langle \text{INIT } C \ ([C], \text{False}) \leftarrow \text{unit}, (h, l, sh) \rangle \Rightarrow \langle \text{Val } v', (h', l', sh') \rangle; \\ & \quad \text{new-Addr } h' = \text{None}; \text{is-class } P \ C \rrbracket \\ \Rightarrow & P \vdash \langle \text{new } C, (h, l, sh) \rangle \Rightarrow \langle \text{THROW } \text{OutOfMemory}, (h', l', sh') \rangle \end{aligned}$$

| *NewInitThrow*:

$$\begin{aligned} & \llbracket \# \text{sfs. sh } C = \text{Some}(\text{sfs}, \text{Done}); P \vdash \langle \text{INIT } C \ ([C], \text{False}) \leftarrow \text{unit}, (h, l, sh) \rangle \Rightarrow \langle \text{throw } a, s^\wedge \rangle; \\ & \quad \text{is-class } P \ C \rrbracket \\ \Rightarrow & P \vdash \langle \text{new } C, (h, l, sh) \rangle \Rightarrow \langle \text{throw } a, s^\wedge \rangle \end{aligned}$$

| *Cast*:

$$\begin{aligned} & \llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, l, sh) \rangle; h \ a = \text{Some}(D, fs); P \vdash D \preceq^* C \rrbracket \\ \Rightarrow & P \vdash \langle \text{Cast } C \ e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, l, sh) \rangle \end{aligned}$$

| *CastNull*:

$$\begin{aligned} & P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \Rightarrow \\ & P \vdash \langle \text{Cast } C \ e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \end{aligned}$$

| *CastFail*:

$$\begin{aligned} & \llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, l, sh) \rangle; h \ a = \text{Some}(D, fs); \neg P \vdash D \preceq^* C \rrbracket \\ \Rightarrow & P \vdash \langle \text{Cast } C \ e, s_0 \rangle \Rightarrow \langle \text{THROW } \text{ClassCast}, (h, l, sh) \rangle \end{aligned}$$

| *CastThrow*:

$$\begin{aligned} & P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Rightarrow \\ & P \vdash \langle \text{Cast } C \ e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \end{aligned}$$

| *Val*:

$$P \vdash \langle \text{Val } v, s \rangle \Rightarrow \langle \text{Val } v, s \rangle$$

| *BinOp*:

$$\begin{aligned} & \llbracket P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v_2, s_2 \rangle; \text{binop}(bop, v_1, v_2) = \text{Some } v \rrbracket \\ \Rightarrow & P \vdash \langle e_1 \ \langle bop \rangle \ e_2, s_0 \rangle \Rightarrow \langle \text{Val } v, s_2 \rangle \end{aligned}$$

| *BinOpThrow1*:

$$\begin{aligned} & P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle \Rightarrow \\ & P \vdash \langle e_1 \ \langle bop \rangle \ e_2, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle \end{aligned}$$

| *BinOpThrow2*:

$$\begin{aligned} & \llbracket P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{throw } e, s_2 \rangle \rrbracket \\ \Rightarrow & P \vdash \langle e_1 \ \langle bop \rangle \ e_2, s_0 \rangle \Rightarrow \langle \text{throw } e, s_2 \rangle \end{aligned}$$

| *Var*:

$$\begin{aligned} & l \ V = \text{Some } v \Rightarrow \\ & P \vdash \langle \text{Var } V, (h, l, sh) \rangle \Rightarrow \langle \text{Val } v, (h, l, sh) \rangle \end{aligned}$$

| *LAss*:

$$\begin{aligned} & \llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, (h, l, sh) \rangle; l' = l(V \mapsto v) \rrbracket \\ \Rightarrow & P \vdash \langle V := e, s_0 \rangle \Rightarrow \langle \text{unit}, (h, l', sh) \rangle \end{aligned}$$

| *LAssThrow*:

$$\begin{aligned} & P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Rightarrow \\ & P \vdash \langle V := e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \end{aligned}$$

| *FAcc*:

$$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, l, sh) \rangle; h \ a = \text{Some}(C, fs);$$

$$\begin{aligned}
& P \vdash C \text{ has } F, \text{NonStatic}:t \text{ in } D; \\
& fs(F,D) = \text{Some } v \text{]} \\
\implies P \vdash \langle e \cdot F\{D\}, s_0 \rangle \Rightarrow \langle \text{Val } v, (h, l, sh) \rangle
\end{aligned}$$

| *FACcNull*:

$$\begin{aligned}
& P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \implies \\
& P \vdash \langle e \cdot F\{D\}, s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_1 \rangle
\end{aligned}$$

| *FACcThrow*:

$$\begin{aligned}
& P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \implies \\
& P \vdash \langle e \cdot F\{D\}, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle
\end{aligned}$$

| *FACcNone*:

$$\begin{aligned}
& \llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, l, sh) \rangle; h \ a = \text{Some}(C, fs); \\
& \quad \neg(\exists b \ t. P \vdash C \text{ has } F, b:t \text{ in } D) \rrbracket \\
\implies P \vdash \langle e \cdot F\{D\}, s_0 \rangle \Rightarrow \langle \text{THROW NoSuchFieldError}, (h, l, sh) \rangle
\end{aligned}$$

| *FACcStatic*:

$$\begin{aligned}
& \llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, l, sh) \rangle; h \ a = \text{Some}(C, fs); \\
& \quad P \vdash C \text{ has } F, \text{Static}:t \text{ in } D \rrbracket \\
\implies P \vdash \langle e \cdot F\{D\}, s_0 \rangle \Rightarrow \langle \text{THROW IncompatibleClassChangeError}, (h, l, sh) \rangle
\end{aligned}$$

| *SFACc*:

$$\begin{aligned}
& \llbracket P \vdash C \text{ has } F, \text{Static}:t \text{ in } D; \\
& \quad sh \ D = \text{Some} (sfs, Done); \\
& \quad sfs \ F = \text{Some } v \rrbracket \\
\implies P \vdash \langle C \cdot_s F\{D\}, (h, l, sh) \rangle \Rightarrow \langle \text{Val } v, (h, l, sh) \rangle
\end{aligned}$$

| *SFACcInit*:

$$\begin{aligned}
& \llbracket P \vdash C \text{ has } F, \text{Static}:t \text{ in } D; \\
& \quad \nexists sfs. sh \ D = \text{Some} (sfs, Done); P \vdash \langle \text{INIT } D \ ([D], \text{False}) \leftarrow \text{unit}, (h, l, sh) \rangle \Rightarrow \langle \text{Val } v', (h', l', sh') \rangle; \\
& \quad sh' \ D = \text{Some} (sfs, i); \\
& \quad sfs \ F = \text{Some } v \rrbracket \\
\implies P \vdash \langle C \cdot_s F\{D\}, (h, l, sh) \rangle \Rightarrow \langle \text{Val } v, (h', l', sh') \rangle
\end{aligned}$$

| *SFACcInitThrow*:

$$\begin{aligned}
& \llbracket P \vdash C \text{ has } F, \text{Static}:t \text{ in } D; \\
& \quad \nexists sfs. sh \ D = \text{Some} (sfs, Done); P \vdash \langle \text{INIT } D \ ([D], \text{False}) \leftarrow \text{unit}, (h, l, sh) \rangle \Rightarrow \langle \text{throw } a, s' \rangle \rrbracket \\
\implies P \vdash \langle C \cdot_s F\{D\}, (h, l, sh) \rangle \Rightarrow \langle \text{throw } a, s' \rangle
\end{aligned}$$

| *SFACcNone*:

$$\begin{aligned}
& \llbracket \neg(\exists b \ t. P \vdash C \text{ has } F, b:t \text{ in } D) \rrbracket \\
\implies P \vdash \langle C \cdot_s F\{D\}, s \rangle \Rightarrow \langle \text{THROW NoSuchFieldError}, s \rangle
\end{aligned}$$

| *SFACcNonStatic*:

$$\begin{aligned}
& \llbracket P \vdash C \text{ has } F, \text{NonStatic}:t \text{ in } D \rrbracket \\
\implies P \vdash \langle C \cdot_s F\{D\}, s \rangle \Rightarrow \langle \text{THROW IncompatibleClassChangeError}, s \rangle
\end{aligned}$$

| *FAss*:

$$\begin{aligned}
& \llbracket P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2, sh_2) \rangle; \\
& \quad h_2 \ a = \text{Some}(C, fs); P \vdash C \text{ has } F, \text{NonStatic}:t \text{ in } D; \\
& \quad fs' = fs((F, D) \mapsto v); h_2' = h_2(a \mapsto (C, fs')) \rrbracket \\
\implies P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{unit}, (h_2', l_2, sh_2) \rangle
\end{aligned}$$

- | *FAssNull*:

$$\llbracket P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, s_2 \rangle \rrbracket \Longrightarrow P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{THROW } \text{NullPointer}, s_2 \rangle$$
- | *FAssThrow1*:

$$P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$$
- | *FAssThrow2*:

$$\llbracket P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle \rrbracket \Longrightarrow P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle$$
- | *FAssNone*:

$$\llbracket P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2, sh_2) \rangle; h_2 \ a = \text{Some}(C, fs); \neg(\exists b \ t. P \vdash C \text{ has } F, b:t \text{ in } D) \rrbracket \Longrightarrow P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{THROW } \text{NoSuchFieldError}, (h_2, l_2, sh_2) \rangle$$
- | *FAssStatic*:

$$\llbracket P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2, sh_2) \rangle; h_2 \ a = \text{Some}(C, fs); P \vdash C \text{ has } F, \text{Static}:t \text{ in } D \rrbracket \Longrightarrow P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{THROW } \text{IncompatibleClassChangeError}, (h_2, l_2, sh_2) \rangle$$
- | *SFAss*:

$$\llbracket P \vdash \langle e_2, s_0 \rangle \Rightarrow \langle \text{Val } v, (h_1, l_1, sh_1) \rangle; P \vdash C \text{ has } F, \text{Static}:t \text{ in } D; sh_1 \ D = \text{Some}(sfs, \text{Done}); sfs' = sfs(F \mapsto v); sh_1' = sh_1(D \mapsto (sfs', \text{Done})) \rrbracket \Longrightarrow P \vdash \langle C \cdot_s F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{unit}, (h_1, l_1, sh_1') \rangle$$
- | *SFAssInit*:

$$\llbracket P \vdash \langle e_2, s_0 \rangle \Rightarrow \langle \text{Val } v, (h_1, l_1, sh_1) \rangle; P \vdash C \text{ has } F, \text{Static}:t \text{ in } D; \nexists sfs. sh_1 \ D = \text{Some}(sfs, \text{Done}); P \vdash \langle \text{INIT } D \ ([D], \text{False}) \leftarrow \text{unit}, (h_1, l_1, sh_1) \rangle \Rightarrow \langle \text{Val } v', (h', l', sh') \rangle; sh' \ D = \text{Some}(sfs, i); sfs' = sfs(F \mapsto v); sh'' = sh'(D \mapsto (sfs', i)) \rrbracket \Longrightarrow P \vdash \langle C \cdot_s F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{unit}, (h', l', sh'') \rangle$$
- | *SFAssInitThrow*:

$$\llbracket P \vdash \langle e_2, s_0 \rangle \Rightarrow \langle \text{Val } v, (h_1, l_1, sh_1) \rangle; P \vdash C \text{ has } F, \text{Static}:t \text{ in } D; \nexists sfs. sh_1 \ D = \text{Some}(sfs, \text{Done}); P \vdash \langle \text{INIT } D \ ([D], \text{False}) \leftarrow \text{unit}, (h_1, l_1, sh_1) \rangle \Rightarrow \langle \text{throw } a, s' \rangle \rrbracket \Longrightarrow P \vdash \langle C \cdot_s F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{throw } a, s' \rangle$$
- | *SFAssThrow*:

$$P \vdash \langle e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle \Longrightarrow P \vdash \langle C \cdot_s F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle$$
- | *SFAssNone*:

$$\llbracket P \vdash \langle e_2, s_0 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2, sh_2) \rangle; \neg(\exists b \ t. P \vdash C \text{ has } F, b:t \text{ in } D) \rrbracket \Longrightarrow P \vdash \langle C \cdot_s F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{THROW } \text{NoSuchFieldError}, (h_2, l_2, sh_2) \rangle$$
- | *SFAssNonStatic*:

$$\llbracket P \vdash \langle e_2, s_0 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2, sh_2) \rangle; P \vdash C \text{ has } F, \text{NonStatic}:t \text{ in } D \rrbracket$$

$$\implies P \vdash \langle C \cdot_s F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{THROW IncompatibleClassChangeError}, (h_2, l_2, sh_2) \rangle$$

| *CallObjThrow*:

$$P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \implies \\ P \vdash \langle e \cdot M(ps), s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$$

| *CallParamsThrow*:

$$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash \langle es, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs \ @ \ \text{throw } ex \ \# \ es', s_2 \rangle \rrbracket \\ \implies P \vdash \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle \text{throw } ex, s_2 \rangle$$

| *CallNull*:

$$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle; P \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, s_2 \rangle \rrbracket \\ \implies P \vdash \langle e \cdot M(ps), s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_2 \rangle$$

| *CallNone*:

$$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_2, l_2, sh_2) \rangle; \\ h_2 \ a = \text{Some}(C, fs); \neg(\exists b \ Ts \ T \ m \ D. P \vdash C \ \text{sees } M, b: Ts \rightarrow T = m \ \text{in } D) \rrbracket \\ \implies P \vdash \langle e \cdot M(ps), s_0 \rangle \Rightarrow \langle \text{THROW NoSuchMethodError}, (h_2, l_2, sh_2) \rangle$$

| *CallStatic*:

$$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_2, l_2, sh_2) \rangle; \\ h_2 \ a = \text{Some}(C, fs); P \vdash C \ \text{sees } M, \text{Static}: Ts \rightarrow T = m \ \text{in } D \rrbracket \\ \implies P \vdash \langle e \cdot M(ps), s_0 \rangle \Rightarrow \langle \text{THROW IncompatibleClassChangeError}, (h_2, l_2, sh_2) \rangle$$

| *Call*:

$$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_2, l_2, sh_2) \rangle; \\ h_2 \ a = \text{Some}(C, fs); P \vdash C \ \text{sees } M, \text{NonStatic}: Ts \rightarrow T = (pns, \text{body}) \ \text{in } D; \\ \text{length } vs = \text{length } pns; \ l_2' = [\text{this} \rightarrow \text{Addr } a, \ pns \mapsto vs]; \\ P \vdash \langle \text{body}, (h_2, l_2', sh_2) \rangle \Rightarrow \langle e', (h_3, l_3, sh_3) \rangle \rrbracket \\ \implies P \vdash \langle e \cdot M(ps), s_0 \rangle \Rightarrow \langle e', (h_3, l_3, sh_3) \rangle$$

| *SCallParamsThrow*:

$$\llbracket P \vdash \langle es, s_0 \rangle [\Rightarrow] \langle \text{map Val } vs \ @ \ \text{throw } ex \ \# \ es', s_2 \rangle \rrbracket \\ \implies P \vdash \langle C \cdot_s M(es), s_0 \rangle \Rightarrow \langle \text{throw } ex, s_2 \rangle$$

| *SCallNone*:

$$\llbracket P \vdash \langle ps, s_0 \rangle [\Rightarrow] \langle \text{map Val } vs, s_2 \rangle; \\ \neg(\exists b \ Ts \ T \ m \ D. P \vdash C \ \text{sees } M, b: Ts \rightarrow T = m \ \text{in } D) \rrbracket \\ \implies P \vdash \langle C \cdot_s M(ps), s_0 \rangle \Rightarrow \langle \text{THROW NoSuchMethodError}, s_2 \rangle$$

| *SCallNonStatic*:

$$\llbracket P \vdash \langle ps, s_0 \rangle [\Rightarrow] \langle \text{map Val } vs, s_2 \rangle; \\ P \vdash C \ \text{sees } M, \text{NonStatic}: Ts \rightarrow T = m \ \text{in } D \rrbracket \\ \implies P \vdash \langle C \cdot_s M(ps), s_0 \rangle \Rightarrow \langle \text{THROW IncompatibleClassChangeError}, s_2 \rangle$$

| *SCallInitThrow*:

$$\llbracket P \vdash \langle ps, s_0 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_1, l_1, sh_1) \rangle; \\ P \vdash C \ \text{sees } M, \text{Static}: Ts \rightarrow T = (pns, \text{body}) \ \text{in } D; \\ \nexists \text{sfs}. \ sh_1 \ D = \text{Some}(\text{sfs}, \text{Done}); M \neq \text{clinit}; \\ P \vdash \langle \text{INIT } D \ ([D], \text{False}) \leftarrow \text{unit}, (h_1, l_1, sh_1) \rangle \Rightarrow \langle \text{throw } a, s' \rangle \rrbracket \\ \implies P \vdash \langle C \cdot_s M(ps), s_0 \rangle \Rightarrow \langle \text{throw } a, s' \rangle$$

| *SCallInit*:

$$\llbracket P \vdash \langle ps, s_0 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_1, l_1, sh_1) \rangle; \rrbracket$$

$$\begin{aligned}
& P \vdash C \text{ sees } M, \text{Static: } Ts \rightarrow T = (pns, \text{body}) \text{ in } D; \\
& \# \text{sfs. } sh_1 \ D = \text{Some}(\text{sfs}, \text{Done}); M \neq \text{clinit}; \\
& P \vdash \langle \text{INIT } D \ ([D], \text{False}) \leftarrow \text{unit}, (h_1, l_1, sh_1) \rangle \Rightarrow \langle \text{Val } v', (h_2, l_2, sh_2) \rangle; \\
& \text{length } vs = \text{length } pns; \ l_2' = [pns[\mapsto]vs]; \\
& P \vdash \langle \text{body}, (h_2, l_2', sh_2) \rangle \Rightarrow \langle e', (h_3, l_3, sh_3) \rangle \] \\
\Rightarrow & P \vdash \langle C \cdot_s M(ps), s_0 \rangle \Rightarrow \langle e', (h_3, l_2, sh_3) \rangle
\end{aligned}$$

| *S*Call:

$$\begin{aligned}
& \llbracket P \vdash \langle ps, s_0 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_2, l_2, sh_2) \rangle; \\
& \quad P \vdash C \text{ sees } M, \text{Static: } Ts \rightarrow T = (pns, \text{body}) \text{ in } D; \\
& \quad sh_2 \ D = \text{Some}(\text{sfs}, \text{Done}) \vee (M = \text{clinit} \wedge sh_2 \ D = \text{Some}(\text{sfs}, \text{Processing})); \\
& \quad \text{length } vs = \text{length } pns; \ l_2' = [pns[\mapsto]vs]; \\
& \quad P \vdash \langle \text{body}, (h_2, l_2', sh_2) \rangle \Rightarrow \langle e', (h_3, l_3, sh_3) \rangle \] \\
\Rightarrow & P \vdash \langle C \cdot_s M(ps), s_0 \rangle \Rightarrow \langle e', (h_3, l_2, sh_3) \rangle
\end{aligned}$$

| *B*lock:

$$\begin{aligned}
& P \vdash \langle e_0, (h_0, l_0(V := \text{None}), sh_0) \rangle \Rightarrow \langle e_1, (h_1, l_1, sh_1) \rangle \Rightarrow \\
& P \vdash \langle \{V:T; e_0\}, (h_0, l_0, sh_0) \rangle \Rightarrow \langle e_1, (h_1, l_1(V := l_0 \ V), sh_1) \rangle
\end{aligned}$$

| *S*eq:

$$\begin{aligned}
& \llbracket P \vdash \langle e_0, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash \langle e_1, s_1 \rangle \Rightarrow \langle e_2, s_2 \rangle \] \\
\Rightarrow & P \vdash \langle e_0;;e_1, s_0 \rangle \Rightarrow \langle e_2, s_2 \rangle
\end{aligned}$$

| *S*eqThrow:

$$\begin{aligned}
& P \vdash \langle e_0, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle \Rightarrow \\
& P \vdash \langle e_0;;e_1, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle
\end{aligned}$$

| *C*ondT:

$$\begin{aligned}
& \llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle; P \vdash \langle e_1, s_1 \rangle \Rightarrow \langle e', s_2 \rangle \] \\
\Rightarrow & P \vdash \langle \text{if } (e) \ e_1 \ \text{else } e_2, s_0 \rangle \Rightarrow \langle e', s_2 \rangle
\end{aligned}$$

| *C*ondF:

$$\begin{aligned}
& \llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{false}, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle e', s_2 \rangle \] \\
\Rightarrow & P \vdash \langle \text{if } (e) \ e_1 \ \text{else } e_2, s_0 \rangle \Rightarrow \langle e', s_2 \rangle
\end{aligned}$$

| *C*ondThrow:

$$\begin{aligned}
& P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Rightarrow \\
& P \vdash \langle \text{if } (e) \ e_1 \ \text{else } e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle
\end{aligned}$$

| *W*hileF:

$$\begin{aligned}
& P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{false}, s_1 \rangle \Rightarrow \\
& P \vdash \langle \text{while } (e) \ c, s_0 \rangle \Rightarrow \langle \text{unit}, s_1 \rangle
\end{aligned}$$

| *W*hileT:

$$\begin{aligned}
& \llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle; P \vdash \langle c, s_1 \rangle \Rightarrow \langle \text{Val } v_1, s_2 \rangle; P \vdash \langle \text{while } (e) \ c, s_2 \rangle \Rightarrow \langle e_3, s_3 \rangle \] \\
\Rightarrow & P \vdash \langle \text{while } (e) \ c, s_0 \rangle \Rightarrow \langle e_3, s_3 \rangle
\end{aligned}$$

| *W*hileCondThrow:

$$\begin{aligned}
& P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Rightarrow \\
& P \vdash \langle \text{while } (e) \ c, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle
\end{aligned}$$

| *W*hileBodyThrow:

$$\begin{aligned}
& \llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle; P \vdash \langle c, s_1 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle \] \\
\Rightarrow & P \vdash \langle \text{while } (e) \ c, s_0 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle
\end{aligned}$$

| *Throw*:

$$\begin{aligned} P \vdash \langle e, s_0 \rangle &\Rightarrow \langle \text{addr } a, s_1 \rangle \Longrightarrow \\ P \vdash \langle \text{throw } e, s_0 \rangle &\Rightarrow \langle \text{Throw } a, s_1 \rangle \end{aligned}$$

| *ThrowNull*:

$$\begin{aligned} P \vdash \langle e, s_0 \rangle &\Rightarrow \langle \text{null}, s_1 \rangle \Longrightarrow \\ P \vdash \langle \text{throw } e, s_0 \rangle &\Rightarrow \langle \text{THROW NullPointer}, s_1 \rangle \end{aligned}$$

| *ThrowThrow*:

$$\begin{aligned} P \vdash \langle e, s_0 \rangle &\Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow \\ P \vdash \langle \text{throw } e, s_0 \rangle &\Rightarrow \langle \text{throw } e', s_1 \rangle \end{aligned}$$

| *Try*:

$$\begin{aligned} P \vdash \langle e_1, s_0 \rangle &\Rightarrow \langle \text{Val } v_1, s_1 \rangle \Longrightarrow \\ P \vdash \langle \text{try } e_1 \text{ catch}(C V) e_2, s_0 \rangle &\Rightarrow \langle \text{Val } v_1, s_1 \rangle \end{aligned}$$

| *TryCatch*:

$$\begin{aligned} \llbracket P \vdash \langle e_1, s_0 \rangle &\Rightarrow \langle \text{Throw } a, (h_1, l_1, sh_1) \rangle; h_1 a = \text{Some}(D, fs); P \vdash D \preceq^* C; \\ P \vdash \langle e_2, (h_1, l_1(V \mapsto \text{Addr } a), sh_1) \rangle &\Rightarrow \langle e_2', (h_2, l_2, sh_2) \rangle \rrbracket \\ \Longrightarrow P \vdash \langle \text{try } e_1 \text{ catch}(C V) e_2, s_0 \rangle &\Rightarrow \langle e_2', (h_2, l_2(V := l_1 V), sh_2) \rangle \end{aligned}$$

| *TryThrow*:

$$\begin{aligned} \llbracket P \vdash \langle e_1, s_0 \rangle &\Rightarrow \langle \text{Throw } a, (h_1, l_1, sh_1) \rangle; h_1 a = \text{Some}(D, fs); \neg P \vdash D \preceq^* C \rrbracket \\ \Longrightarrow P \vdash \langle \text{try } e_1 \text{ catch}(C V) e_2, s_0 \rangle &\Rightarrow \langle \text{Throw } a, (h_1, l_1, sh_1) \rangle \end{aligned}$$

| *Nil*:

$$P \vdash \langle [], s \rangle [\Rightarrow] \langle [], s \rangle$$

| *Cons*:

$$\begin{aligned} \llbracket P \vdash \langle e, s_0 \rangle &\Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash \langle es, s_1 \rangle [\Rightarrow] \langle es', s_2 \rangle \rrbracket \\ \Longrightarrow P \vdash \langle e \# es, s_0 \rangle [\Rightarrow] &\langle \text{Val } v \# es', s_2 \rangle \end{aligned}$$

| *ConsThrow*:

$$\begin{aligned} P \vdash \langle e, s_0 \rangle &\Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow \\ P \vdash \langle e \# es, s_0 \rangle [\Rightarrow] &\langle \text{throw } e' \# es, s_1 \rangle \end{aligned}$$

— init rules

| *InitFinal*:

$$P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \Longrightarrow P \vdash \langle \text{INIT } C (\text{Nil}, b) \leftarrow e, s \rangle \Rightarrow \langle e', s' \rangle$$

| *InitNone*:

$$\begin{aligned} \llbracket sh C = \text{None}; P \vdash \langle \text{INIT } C' (C \# Cs, \text{False}) \leftarrow e, (h, l, sh(C \mapsto (\text{sblank } P C, \text{Prepared}))) \rangle &\Rightarrow \langle e', s' \rangle \rrbracket \\ \Longrightarrow P \vdash \langle \text{INIT } C' (C \# Cs, \text{False}) \leftarrow e, (h, l, sh) \rangle &\Rightarrow \langle e', s' \rangle \end{aligned}$$

| *InitDone*:

$$\begin{aligned} \llbracket sh C = \text{Some}(sfs, \text{Done}); P \vdash \langle \text{INIT } C' (Cs, \text{True}) \leftarrow e, (h, l, sh) \rangle &\Rightarrow \langle e', s' \rangle \rrbracket \\ \Longrightarrow P \vdash \langle \text{INIT } C' (C \# Cs, \text{False}) \leftarrow e, (h, l, sh) \rangle &\Rightarrow \langle e', s' \rangle \end{aligned}$$

| *InitProcessing*:

$$\begin{aligned} \llbracket sh C = \text{Some}(sfs, \text{Processing}); P \vdash \langle \text{INIT } C' (Cs, \text{True}) \leftarrow e, (h, l, sh) \rangle &\Rightarrow \langle e', s' \rangle \rrbracket \\ \Longrightarrow P \vdash \langle \text{INIT } C' (C \# Cs, \text{False}) \leftarrow e, (h, l, sh) \rangle &\Rightarrow \langle e', s' \rangle \end{aligned}$$

— note that RI will mark all classes in the list Cs with the Error flag

| *InitError*:

$$\begin{aligned} & \llbracket sh\ C = Some(sfs, Error); \\ & \quad P \vdash \langle RI\ (C, THROW\ NoClassDefFoundError); Cs \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle \rrbracket \\ & \Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs, False) \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle \end{aligned}$$

| *InitObject*:

$$\begin{aligned} & \llbracket sh\ C = Some(sfs, Prepared); \\ & \quad C = Object; \\ & \quad sh' = sh(C \mapsto (sfs, Processing)); \\ & \quad P \vdash \langle INIT\ C'\ (C\#Cs, True) \leftarrow e, (h, l, sh') \rangle \Rightarrow \langle e', s' \rangle \rrbracket \\ & \Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs, False) \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle \end{aligned}$$

| *InitNonObject*:

$$\begin{aligned} & \llbracket sh\ C = Some(sfs, Prepared); \\ & \quad C \neq Object; \\ & \quad class\ P\ C = Some\ (D, r); \\ & \quad sh' = sh(C \mapsto (sfs, Processing)); \\ & \quad P \vdash \langle INIT\ C'\ (D\#C\#Cs, False) \leftarrow e, (h, l, sh') \rangle \Rightarrow \langle e', s' \rangle \rrbracket \\ & \Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs, False) \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle \end{aligned}$$

| *InitRInit*:

$$\begin{aligned} & P \vdash \langle RI\ (C, C \cdot_s\ clinit(\llbracket \rrbracket)); Cs \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle \\ & \Longrightarrow P \vdash \langle INIT\ C'\ (C\#Cs, True) \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle \end{aligned}$$

| *RInit*:

$$\begin{aligned} & \llbracket P \vdash \langle e', s \rangle \Rightarrow \langle Val\ v, (h', l', sh') \rangle; \\ & \quad sh' C = Some(sfs, i); sh'' = sh'(C \mapsto (sfs, Done)); \\ & \quad C' = last(C\#Cs); \\ & \quad P \vdash \langle INIT\ C'\ (Cs, True) \leftarrow e, (h', l', sh'') \rangle \Rightarrow \langle e_1, s_1 \rangle \rrbracket \\ & \Longrightarrow P \vdash \langle RI\ (C, e'); Cs \leftarrow e, s \rangle \Rightarrow \langle e_1, s_1 \rangle \end{aligned}$$

| *RInitInitFail*:

$$\begin{aligned} & \llbracket P \vdash \langle e', s \rangle \Rightarrow \langle throw\ a, (h', l', sh') \rangle; \\ & \quad sh' C = Some(sfs, i); sh'' = sh'(C \mapsto (sfs, Error)); \\ & \quad P \vdash \langle RI\ (D, throw\ a); Cs \leftarrow e, (h', l', sh'') \rangle \Rightarrow \langle e_1, s_1 \rangle \rrbracket \\ & \Longrightarrow P \vdash \langle RI\ (C, e'); D\#Cs \leftarrow e, s \rangle \Rightarrow \langle e_1, s_1 \rangle \end{aligned}$$

| *RInitFailFinal*:

$$\begin{aligned} & \llbracket P \vdash \langle e', s \rangle \Rightarrow \langle throw\ a, (h', l', sh') \rangle; \\ & \quad sh' C = Some(sfs, i); sh'' = sh'(C \mapsto (sfs, Error)) \rrbracket \\ & \Longrightarrow P \vdash \langle RI\ (C, e'); Nil \leftarrow e, s \rangle \Rightarrow \langle throw\ a, (h', l', sh'') \rangle \end{aligned}$$

1.15.1 Final expressions

lemma *eval-final*: $P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \Longrightarrow final\ e'$

and *evals-final*: $P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \Longrightarrow finals\ es'$

Only used later, in the small to big translation, but is already a good sanity check:

lemma *eval-finalId*: $final\ e \Longrightarrow P \vdash \langle e, s \rangle \Rightarrow \langle e, s \rangle$

lemma *eval-final-same*: $\llbracket P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle; final\ e \rrbracket \Longrightarrow e = e' \wedge s = s'$

lemma *eval-finalsId*:

assumes *finals*: *finals* es **shows** $P \vdash \langle es, s \rangle [\Rightarrow] \langle es, s \rangle$

lemma *evals-finals-same*:

assumes *finals*: *finals es*

shows $P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \Longrightarrow es = es' \wedge s = s'$

using *finals*

proof (*induct es arbitrary: es' type: list*)

case Nil then show *?case using evals-cases(1) by blast*

next

case (*Cons e es*)

have *IH*: $\bigwedge es'. P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \Longrightarrow \text{finals } es \Longrightarrow es = es' \wedge s = s'$

and *step*: $P \vdash \langle e \# es, s \rangle [\Rightarrow] \langle e' \# es', s' \rangle$ **and** *finals*: *finals (e # es)* **by** *fact+*

then obtain *e' es'' where es' = e' # es'' by* (*meson Cons.premis(1) evals-cases(2)*)

have *fe*: *final e using finals not-finals-ConsI by auto*

show *?case*

proof(*rule evals-cases(2)[OF step]*)

fix *v s1 es1 assume es1*: $es' = \text{Val } v \# es1$

and *estep*: $P \vdash \langle e, s \rangle \Rightarrow \langle \text{Val } v, s1 \rangle$ **and** *esstep*: $P \vdash \langle es, s1 \rangle [\Rightarrow] \langle es1, s' \rangle$

then have $e = \text{Val } v$ **using** *eval-final-same fe by auto*

then have *finals es using es' not-finals-ConsI2 finals by blast*

then show *?thesis using es' IH estep esstep es1 eval-final-same fe by blast*

next

fix *e' assume es1*: $es' = \text{throw } e' \# es$ **and** *estep*: $P \vdash \langle e, s \rangle \Rightarrow \langle \text{throw } e', s' \rangle$

then have $e = \text{throw } e'$ **using** *eval-final-same fe by auto*

then show *?thesis using es' estep es1 eval-final-same fe by blast*

qed

qed

1.15.2 Property preservation

lemma *evals-length*: $P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \Longrightarrow \text{length } es = \text{length } es'$

by(*induct es arbitrary:es' s s', auto elim: evals-cases*)

corollary *evals-empty*: $P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \Longrightarrow (es = []) = (es' = [])$

by(*drule evals-length, fastforce*)

theorem *eval-hext*: $P \vdash \langle e, (h, l, sh) \rangle \Rightarrow \langle e', (h', l', sh') \rangle \Longrightarrow h \sqsubseteq h'$

and *evals-hext*: $P \vdash \langle es, (h, l, sh) \rangle [\Rightarrow] \langle es', (h', l', sh') \rangle \Longrightarrow h \sqsubseteq h'$

lemma *eval-lcl-incr*: $P \vdash \langle e, (h_0, l_0, sh_0) \rangle \Rightarrow \langle e', (h_1, l_1, sh_1) \rangle \Longrightarrow \text{dom } l_0 \subseteq \text{dom } l_1$

and *evals-lcl-incr*: $P \vdash \langle es, (h_0, l_0, sh_0) \rangle [\Rightarrow] \langle es', (h_1, l_1, sh_1) \rangle \Longrightarrow \text{dom } l_0 \subseteq \text{dom } l_1$

lemma

shows *init-ri-same-loc*: $P \vdash \langle e, (h, l, sh) \rangle \Rightarrow \langle e', (h', l', sh') \rangle$

$\Longrightarrow (\bigwedge C Cs b M a. e = \text{INIT } C (Cs, b) \leftarrow \text{unit} \vee e = C \cdot_s M ([]) \vee e = \text{RI } (C, \text{Throw } a) ; Cs \leftarrow \text{unit}$

$\vee e = \text{RI } (C, C \cdot_s \text{clinit} ([])) ; Cs \leftarrow \text{unit}$

$\Longrightarrow l = l')$

and $P \vdash \langle es, (h, l, sh) \rangle [\Rightarrow] \langle es', (h', l', sh') \rangle \Longrightarrow \text{True}$

proof(*induct rule: eval-evals-inducts*)

case (*RInitInitFail e h l sh a'*)

then show *?case using eval-final[of - - - throw a']*

by(*fastforce dest: eval-final-same[of - Throw a]*)

next

case *RInitFailFinal then show ?case by*(*auto dest: eval-final-same*)

qed(*auto dest: evals-cases eval-cases(17) eval-final-same*)

lemma *init-same-loc*: $P \vdash \langle \text{INIT } C \ (Cs, b) \leftarrow \text{unit}, (h, l, sh) \rangle \Rightarrow \langle e', (h', l', sh') \rangle \Longrightarrow l = l'$
by(*simp add: init-ri-same-loc*)

lemma *assumes wf: wwf-J-prog P*

shows *eval-proc-pres'*: $P \vdash \langle e, (h, l, sh) \rangle \Rightarrow \langle e', (h', l', sh') \rangle$

$\Longrightarrow \text{not-init } C \ e \Longrightarrow \exists \text{ sfs. } sh \ C = \llbracket (\text{sfs}, \text{Processing}) \rrbracket \Longrightarrow \exists \text{ sfs}'. \ sh' \ C = \llbracket (\text{sfs}', \text{Processing}) \rrbracket$

and *evals-proc-pres'*: $P \vdash \langle es, (h, l, sh) \rangle [\Rightarrow] \langle es', (h', l', sh') \rangle$

$\Longrightarrow \text{not-inits } C \ es \Longrightarrow \exists \text{ sfs. } sh \ C = \llbracket (\text{sfs}, \text{Processing}) \rrbracket \Longrightarrow \exists \text{ sfs}'. \ sh' \ C = \llbracket (\text{sfs}', \text{Processing}) \rrbracket$

1.16 Definite assignment

theory *DefAss* **imports** *BigStep* **begin**

1.16.1 Hypersets

type-synonym *'a hyperset* = *'a set option*

definition *hyperUn* :: *'a hyperset* \Rightarrow *'a hyperset* \Rightarrow *'a hyperset* (**infixl** $\langle \sqcup \rangle$ 65)

where

$A \sqcup B \equiv \text{case } A \text{ of } \text{None} \Rightarrow \text{None} \mid \llbracket A \rrbracket \Rightarrow (\text{case } B \text{ of } \text{None} \Rightarrow \text{None} \mid \llbracket B \rrbracket \Rightarrow \llbracket A \cup B \rrbracket)$

definition *hyperInt* :: *'a hyperset* \Rightarrow *'a hyperset* \Rightarrow *'a hyperset* (**infixl** $\langle \sqcap \rangle$ 70)

where

$A \sqcap B \equiv \text{case } A \text{ of } \text{None} \Rightarrow B \mid \llbracket A \rrbracket \Rightarrow (\text{case } B \text{ of } \text{None} \Rightarrow \llbracket A \rrbracket \mid \llbracket B \rrbracket \Rightarrow \llbracket A \cap B \rrbracket)$

definition *hyperDiff1* :: *'a hyperset* \Rightarrow *'a* \Rightarrow *'a hyperset* (**infixl** $\langle \ominus \rangle$ 65)

where

$A \ominus a \equiv \text{case } A \text{ of } \text{None} \Rightarrow \text{None} \mid \llbracket A \rrbracket \Rightarrow \llbracket A - \{a\} \rrbracket$

definition *hyper-isin* :: *'a* \Rightarrow *'a hyperset* \Rightarrow *bool* (**infix** $\langle \in \in \rangle$ 50)

where

$a \in \in A \equiv \text{case } A \text{ of } \text{None} \Rightarrow \text{True} \mid \llbracket A \rrbracket \Rightarrow a \in A$

definition *hyper-subset* :: *'a hyperset* \Rightarrow *'a hyperset* \Rightarrow *bool* (**infix** $\langle \sqsubseteq \rangle$ 50)

where

$A \sqsubseteq B \equiv \text{case } B \text{ of } \text{None} \Rightarrow \text{True} \mid \llbracket B \rrbracket \Rightarrow (\text{case } A \text{ of } \text{None} \Rightarrow \text{False} \mid \llbracket A \rrbracket \Rightarrow A \subseteq B)$

lemmas *hyperset-defs* =

hyperUn-def hyperInt-def hyperDiff1-def hyper-isin-def hyper-subset-def

lemma [*simp*]: $\llbracket \{\} \rrbracket \sqcup A = A \wedge A \sqcup \llbracket \{\} \rrbracket = A$

lemma [*simp*]: $\llbracket A \rrbracket \sqcup \llbracket B \rrbracket = \llbracket A \cup B \rrbracket \wedge \llbracket A \rrbracket \ominus a = \llbracket A - \{a\} \rrbracket$

lemma [*simp*]: $\text{None} \sqcup A = \text{None} \wedge A \sqcup \text{None} = \text{None}$

lemma [*simp*]: $a \in \in \text{None} \wedge \text{None} \ominus a = \text{None}$

lemma *hyper-isin-union*: $x \in \in \llbracket A \rrbracket \Longrightarrow x \in \in \llbracket A \rrbracket \sqcup B$

by(*case-tac B, auto simp: hyper-isin-def*)

lemma *hyperUn-assoc*: $(A \sqcup B) \sqcup C = A \sqcup (B \sqcup C)$

lemma *hyper-insert-comm*: $A \sqcup [\{a\}] = [\{a\}] \sqcup A \wedge A \sqcup ([\{a\}] \sqcup B) = [\{a\}] \sqcup (A \sqcup B)$

lemma *hyper-comm*: $A \sqcup B = B \sqcup A \wedge A \sqcup B \sqcup C = B \sqcup A \sqcup C$

1.16.2 Definite assignment

primrec

$\mathcal{A} :: 'a \text{ exp} \Rightarrow 'a \text{ hyperset}$

and $\mathcal{A}s :: 'a \text{ exp list} \Rightarrow 'a \text{ hyperset}$

where

$\mathcal{A} (\text{new } C) = [\{\}]$
 $\mathcal{A} (\text{Cast } C \ e) = \mathcal{A} \ e$
 $\mathcal{A} (\text{Val } v) = [\{\}]$
 $\mathcal{A} (e_1 \ll\text{bop}\gg e_2) = \mathcal{A} \ e_1 \sqcup \mathcal{A} \ e_2$
 $\mathcal{A} (\text{Var } V) = [\{\}]$
 $\mathcal{A} (\text{LAss } V \ e) = [\{V\}] \sqcup \mathcal{A} \ e$
 $\mathcal{A} (e \cdot F\{D\}) = \mathcal{A} \ e$
 $\mathcal{A} (C \cdot_s F\{D\}) = [\{\}]$
 $\mathcal{A} (e_1 \cdot F\{D\} := e_2) = \mathcal{A} \ e_1 \sqcup \mathcal{A} \ e_2$
 $\mathcal{A} (C \cdot_s F\{D\} := e_2) = \mathcal{A} \ e_2$
 $\mathcal{A} (e \cdot M(es)) = \mathcal{A} \ e \sqcup \mathcal{A}s \ es$
 $\mathcal{A} (C \cdot_s M(es)) = \mathcal{A}s \ es$
 $\mathcal{A} (\{V:T; e\}) = \mathcal{A} \ e \ominus V$
 $\mathcal{A} (e_1;;e_2) = \mathcal{A} \ e_1 \sqcup \mathcal{A} \ e_2$
 $\mathcal{A} (\text{if } (e) \ e_1 \ \text{else } e_2) = \mathcal{A} \ e \sqcup (\mathcal{A} \ e_1 \sqcap \mathcal{A} \ e_2)$
 $\mathcal{A} (\text{while } (b) \ e) = \mathcal{A} \ b$
 $\mathcal{A} (\text{throw } e) = \text{None}$
 $\mathcal{A} (\text{try } e_1 \ \text{catch}(C \ V) \ e_2) = \mathcal{A} \ e_1 \sqcap (\mathcal{A} \ e_2 \ominus V)$
 $\mathcal{A} (\text{INIT } C \ (Cs,b) \leftarrow e) = [\{\}]$
 $\mathcal{A} (\text{RI } (C,e);Cs \leftarrow e') = \mathcal{A} \ e$

$\mathcal{A}s ([\]) = [\{\}]$

$\mathcal{A}s (e\#es) = \mathcal{A} \ e \sqcup \mathcal{A}s \ es$

primrec

$\mathcal{D} :: 'a \text{ exp} \Rightarrow 'a \text{ hyperset} \Rightarrow \text{bool}$

and $\mathcal{D}s :: 'a \text{ exp list} \Rightarrow 'a \text{ hyperset} \Rightarrow \text{bool}$

where

$\mathcal{D} (\text{new } C) \ A = \text{True}$
 $\mathcal{D} (\text{Cast } C \ e) \ A = \mathcal{D} \ e \ A$
 $\mathcal{D} (\text{Val } v) \ A = \text{True}$
 $\mathcal{D} (e_1 \ll\text{bop}\gg e_2) \ A = (\mathcal{D} \ e_1 \ A \wedge \mathcal{D} \ e_2 \ (A \sqcup \mathcal{A} \ e_1))$
 $\mathcal{D} (\text{Var } V) \ A = (V \in\in A)$
 $\mathcal{D} (\text{LAss } V \ e) \ A = \mathcal{D} \ e \ A$
 $\mathcal{D} (e \cdot F\{D\}) \ A = \mathcal{D} \ e \ A$
 $\mathcal{D} (C \cdot_s F\{D\}) \ A = \text{True}$
 $\mathcal{D} (e_1 \cdot F\{D\} := e_2) \ A = (\mathcal{D} \ e_1 \ A \wedge \mathcal{D} \ e_2 \ (A \sqcup \mathcal{A} \ e_1))$
 $\mathcal{D} (C \cdot_s F\{D\} := e_2) \ A = \mathcal{D} \ e_2 \ A$
 $\mathcal{D} (e \cdot M(es)) \ A = (\mathcal{D} \ e \ A \wedge \mathcal{D}s \ es \ (A \sqcup \mathcal{A} \ e))$
 $\mathcal{D} (C \cdot_s M(es)) \ A = \mathcal{D}s \ es \ A$
 $\mathcal{D} (\{V:T; e\}) \ A = \mathcal{D} \ e \ (A \ominus V)$
 $\mathcal{D} (e_1;;e_2) \ A = (\mathcal{D} \ e_1 \ A \wedge \mathcal{D} \ e_2 \ (A \sqcup \mathcal{A} \ e_1))$
 $\mathcal{D} (\text{if } (e) \ e_1 \ \text{else } e_2) \ A =$
 $(\mathcal{D} \ e \ A \wedge \mathcal{D} \ e_1 \ (A \sqcup \mathcal{A} \ e) \wedge \mathcal{D} \ e_2 \ (A \sqcup \mathcal{A} \ e))$

$\mathcal{D} (\text{while } (e) c) A = (\mathcal{D} e A \wedge \mathcal{D} c (A \sqcup \mathcal{A} e))$
 $\mathcal{D} (\text{throw } e) A = \mathcal{D} e A$
 $\mathcal{D} (\text{try } e_1 \text{ catch}(C V) e_2) A = (\mathcal{D} e_1 A \wedge \mathcal{D} e_2 (A \sqcup [\{V\}]))$
 $\mathcal{D} (\text{INIT } C (Cs, b) \leftarrow e) A = \mathcal{D} e A$
 $\mathcal{D} (\text{RI } (C, e); Cs \leftarrow e') A = (\mathcal{D} e A \wedge \mathcal{D} e' A)$

$\mathcal{D} s (\square) A = \text{True}$
 $\mathcal{D} s (e \# es) A = (\mathcal{D} e A \wedge \mathcal{D} s es (A \sqcup \mathcal{A} e))$

lemma *As-map-Val[simp]*: $\mathcal{A} s (\text{map Val } vs) = [\{\}]$

lemma *D-append[iff]*: $\bigwedge A. \mathcal{D} s (es @ es') A = (\mathcal{D} s es A \wedge \mathcal{D} s es' (A \sqcup \mathcal{A} s es))$

lemma *A-fv*: $\bigwedge A. \mathcal{A} e = [A] \implies A \subseteq \text{fv } e$

and $\bigwedge A. \mathcal{A} s es = [A] \implies A \subseteq \text{fvs } es$

lemma *sqUn-lem*: $A \sqsubseteq A' \implies A \sqcup B \sqsubseteq A' \sqcup B$

lemma *diff-lem*: $A \sqsubseteq A' \implies A \ominus b \sqsubseteq A' \ominus b$

lemma *D-mono*: $\bigwedge A A'. A \sqsubseteq A' \implies \mathcal{D} e A \implies \mathcal{D} (e :: 'a \text{ exp}) A'$

and *Ds-mono*: $\bigwedge A A'. A \sqsubseteq A' \implies \mathcal{D} s es A \implies \mathcal{D} s (es :: 'a \text{ exp list}) A'$

lemma *D-mono'*: $\mathcal{D} e A \implies A \sqsubseteq A' \implies \mathcal{D} e A'$

and *Ds-mono'*: $\mathcal{D} s es A \implies A \sqsubseteq A' \implies \mathcal{D} s es A'$

lemma *Ds-Vals*: $\mathcal{D} s (\text{map Val } vs) A \text{ by}(\text{induct } vs, \text{auto})$

end

1.17 Conformance Relations for Type Soundness Proofs

theory *Conform*

imports *Exceptions*

begin

definition *conf* :: $'m \text{ prog} \Rightarrow \text{heap} \Rightarrow \text{val} \Rightarrow \text{ty} \Rightarrow \text{bool}$ ($\langle -, - \vdash - : \leq \rangle$ [51,51,51,51] 50)

where

$P, h \vdash v : \leq T \equiv$
 $\exists T'. \text{typeof}_h v = \text{Some } T' \wedge P \vdash T' \leq T$

definition *oconf* :: $'m \text{ prog} \Rightarrow \text{heap} \Rightarrow \text{obj} \Rightarrow \text{bool}$ ($\langle -, - \vdash - \surd \rangle$ [51,51,51] 50)

where

$P, h \vdash \text{obj } \surd \equiv$
 $\text{let } (C, fs) = \text{obj in } \forall F D T. P \vdash C \text{ has } F, \text{NonStatic}:T \text{ in } D \longrightarrow$
 $(\exists v. fs(F, D) = \text{Some } v \wedge P, h \vdash v : \leq T)$

definition *soconf* :: $'m \text{ prog} \Rightarrow \text{heap} \Rightarrow \text{cname} \Rightarrow \text{sfields} \Rightarrow \text{bool}$ ($\langle -, - \vdash_s - \surd \rangle$ [51,51,51,51] 50)

where

$P, h, C \vdash_s sfs \surd \equiv$
 $\forall F T. P \vdash C \text{ has } F, \text{Static}:T \text{ in } C \longrightarrow$
 $(\exists v. sfs F = \text{Some } v \wedge P, h \vdash v : \leq T)$

definition *hconf* :: $'m \text{ prog} \Rightarrow \text{heap} \Rightarrow \text{bool}$ ($\langle - \vdash - \surd \rangle$ [51,51] 50)

where

$$P \vdash h \checkmark \equiv (\forall a \text{ obj. } h a = \text{Some obj} \longrightarrow P, h \vdash \text{obj} \checkmark) \wedge \text{preallocated } h$$

definition $shconf :: 'm \text{ prog} \Rightarrow \text{heap} \Rightarrow \text{sheap} \Rightarrow \text{bool}$ ($\langle -, - \vdash_s - \checkmark \rangle$ [51,51,51] 50)

where

$$P, h \vdash_s sh \checkmark \equiv (\forall C \text{ sfs } i. sh C = \text{Some}(sfs, i) \longrightarrow P, h, C \vdash_s sfs \checkmark)$$

definition $lconf :: 'm \text{ prog} \Rightarrow \text{heap} \Rightarrow (\text{vname} \rightarrow \text{val}) \Rightarrow (\text{vname} \rightarrow \text{ty}) \Rightarrow \text{bool}$ ($\langle -, - \vdash - '(\leq)' \rightarrow$ [51,51,51,51] 50)

where

$$P, h \vdash l (\leq) E \equiv \forall V v. l V = \text{Some } v \longrightarrow (\exists T. E V = \text{Some } T \wedge P, h \vdash v \leq T)$$

abbreviation

$$\begin{aligned} \text{confs} &:: 'm \text{ prog} \Rightarrow \text{heap} \Rightarrow \text{val list} \Rightarrow \text{ty list} \Rightarrow \text{bool} \\ &(\langle -, - \vdash - [:\leq] \rightarrow [51,51,51,51] 50) \text{ where} \\ P, h \vdash vs [:\leq] Ts &\equiv \text{list-all2 } (\text{conf } P h) vs Ts \end{aligned}$$

1.17.1 Value conformance $:\leq$

lemma $\text{conf-Null [simp]: } P, h \vdash \text{Null} \leq T = P \vdash NT \leq T$

lemma $\text{typeof-conf [simp]: } \text{typeof}_h v = \text{Some } T \Longrightarrow P, h \vdash v \leq T$

lemma $\text{typeof-lit-conf [simp]: } \text{typeof } v = \text{Some } T \Longrightarrow P, h \vdash v \leq T$

lemma $\text{defval-conf [simp]: } P, h \vdash \text{default-val } T \leq T$

lemma $\text{conf-upd-obj: } h a = \text{Some}(C, fs) \Longrightarrow (P, h(a \rightarrow (C, fs')) \vdash x \leq T) = (P, h \vdash x \leq T)$

lemma $\text{conf-widen: } P, h \vdash v \leq T \Longrightarrow P \vdash T \leq T' \Longrightarrow P, h \vdash v \leq T'$

lemma $\text{conf-hext: } h \leq h' \Longrightarrow P, h \vdash v \leq T \Longrightarrow P, h' \vdash v \leq T$

lemma $\text{conf-ClassD: } P, h \vdash v \leq \text{Class } C \Longrightarrow$

$$v = \text{Null} \vee (\exists a \text{ obj } T. v = \text{Addr } a \wedge h a = \text{Some obj} \wedge \text{obj-ty obj} = T \wedge P \vdash T \leq \text{Class } C)$$

lemma $\text{conf-NT [iff]: } P, h \vdash v \leq NT = (v = \text{Null})$

lemma $\text{non-npD: } \llbracket v \neq \text{Null}; P, h \vdash v \leq \text{Class } C \rrbracket$

$$\Longrightarrow \exists a C' fs. v = \text{Addr } a \wedge h a = \text{Some}(C', fs) \wedge P \vdash C' \leq^* C$$

1.17.2 Value list conformance $[:\leq]$

lemma $\text{confs-widens [trans]: } \llbracket P, h \vdash vs [:\leq] Ts; P \vdash Ts [:\leq] Ts' \rrbracket \Longrightarrow P, h \vdash vs [:\leq] Ts'$

lemma $\text{confs-rev: } P, h \vdash \text{rev } s [:\leq] t = (P, h \vdash s [:\leq] \text{rev } t)$

lemma confs-conv-map:

$$\bigwedge Ts'. P, h \vdash vs [:\leq] Ts' = (\exists Ts. \text{map } \text{typeof}_h vs = \text{map } \text{Some } Ts \wedge P \vdash Ts [:\leq] Ts')$$

lemma $\text{confs-hext: } P, h \vdash vs [:\leq] Ts \Longrightarrow h \leq h' \Longrightarrow P, h' \vdash vs [:\leq] Ts$

lemma $\text{confs-Cons2: } P, h \vdash xs [:\leq] y \# ys = (\exists z zs. xs = z \# zs \wedge P, h \vdash z \leq y \wedge P, h \vdash zs [:\leq] ys)$

1.17.3 Object conformance

lemma $\text{oconf-hext: } P, h \vdash \text{obj} \checkmark \Longrightarrow h \leq h' \Longrightarrow P, h' \vdash \text{obj} \checkmark$

lemma oconf-blank:

$$P \vdash C \text{ has-fields } FDTs \Longrightarrow P, h \vdash \text{blank } P C \checkmark$$

lemma $\text{oconf-fupd [intro?]:}$

$$\begin{aligned} \llbracket P \vdash C \text{ has } F, \text{NonStatic}:T \text{ in } D; P, h \vdash v \leq T; P, h \vdash (C, fs) \checkmark \rrbracket \\ \Longrightarrow P, h \vdash (C, fs((F, D) \mapsto v)) \checkmark \end{aligned}$$

1.17.4 Static object conformance

lemma $\text{soconf-hext: } P, h, C \vdash_s \text{obj} \checkmark \Longrightarrow h \leq h' \Longrightarrow P, h', C \vdash_s \text{obj} \checkmark$

lemma soconf-blank:

$P \vdash C \text{ has-fields FDTs} \implies P, h, C \vdash_s \text{blank } P \ C \ \checkmark$

lemma soconf-fupd [intro?]:

$\llbracket P \vdash C \text{ has } F, \text{Static:T in } C; P, h \vdash v : \leq T; P, h, C \vdash_s \text{sfs } \checkmark \rrbracket$
 $\implies P, h, C \vdash_s \text{sfs}(F \mapsto v) \ \checkmark$

1.17.5 Heap conformance

lemma hconfD: $\llbracket P \vdash h \ \checkmark; h \ a = \text{Some obj} \rrbracket \implies P, h \vdash \text{obj } \checkmark$

lemma hconf-new: $\llbracket P \vdash h \ \checkmark; h \ a = \text{None}; P, h \vdash \text{obj } \checkmark \rrbracket \implies P \vdash h(a \mapsto \text{obj}) \ \checkmark$

lemma hconf-upd-obj: $\llbracket P \vdash h \ \checkmark; h \ a = \text{Some}(C, \text{fs}); P, h \vdash (C, \text{fs}') \ \checkmark \rrbracket \implies P \vdash h(a \mapsto (C, \text{fs}')) \ \checkmark$

1.17.6 Class statics conformance

lemma shconfD: $\llbracket P, h \vdash_s \text{sh } \checkmark; \text{sh } C = \text{Some}(\text{sfs}, i) \rrbracket \implies P, h, C \vdash_s \text{sfs } \checkmark$

lemma shconf-upd-obj: $\llbracket P, h \vdash_s \text{sh } \checkmark; P, h, C \vdash_s \text{sfs}' \ \checkmark \rrbracket$

$\implies P, h \vdash_s \text{sh}(C \mapsto (\text{sfs}', i')) \ \checkmark$

lemma shconf-hnew: $\llbracket P, h \vdash_s \text{sh } \checkmark; h \ a = \text{None} \rrbracket \implies P, h(a \mapsto \text{obj}) \vdash_s \text{sh } \checkmark$

lemma shconf-hupd-obj: $\llbracket P, h \vdash_s \text{sh } \checkmark; h \ a = \text{Some}(C, \text{fs}) \rrbracket \implies P, h(a \mapsto (C, \text{fs}')) \vdash_s \text{sh } \checkmark$

1.17.7 Local variable conformance

lemma lconf-hext: $\llbracket P, h \vdash l \ (\leq) \ E; h \leq h' \rrbracket \implies P, h' \vdash l \ (\leq) \ E$

lemma lconf-upd:

$\llbracket P, h \vdash l \ (\leq) \ E; P, h \vdash v : \leq T; E \ V = \text{Some } T \rrbracket \implies P, h \vdash l(V \mapsto v) \ (\leq) \ E$

lemma lconf-empty[iff]: $P, h \vdash \text{Map.empty} \ (\leq) \ E$

lemma lconf-upd2: $\llbracket P, h \vdash l \ (\leq) \ E; P, h \vdash v : \leq T \rrbracket \implies P, h \vdash l(V \mapsto v) \ (\leq) \ E(V \mapsto T)$

end

1.18 Small Step Semantics

theory SmallStep

imports Expr State WWellForm

begin

fun blocks :: *vname list* * *ty list* * *val list* * *expr* \Rightarrow *expr*

where

$\text{blocks}(V \# Vs, T \# Ts, v \# vs, e) = \{ V:T := \text{Val } v; \text{blocks}(Vs, Ts, vs, e) \}$
 $\text{blocks}([], [], [], e) = e$

lemmas blocks-induct = *blocks.induct*[*split-format* (*complete*)]

lemma [*simp*]:

$\llbracket \text{size } vs = \text{size } Vs; \text{size } Ts = \text{size } Vs \rrbracket \implies \text{fv}(\text{blocks}(Vs, Ts, vs, e)) = \text{fv } e - \text{set } Vs$

lemma sub-RI-blocks-body[iff]: $\text{length } vs = \text{length } pns \implies \text{length } Ts = \text{length } pns$

$\implies \text{sub-RI}(\text{blocks}(pns, Ts, vs, \text{body})) \longleftrightarrow \text{sub-RI } \text{body}$

proof(*induct pns arbitrary: Ts vs*)

case Nil then show ?*case* **by simp**

next

case Cons then show ?*case* **by**(*cases vs; cases Ts*) *auto*

qed

definition *assigned* :: 'a ⇒ 'a exp ⇒ bool

where

assigned V e ≡ ∃ v e'. e = (V := Val v;; e')

— expression is okay to go the right side of *INIT* or *RI* ← or to have indicator Boolean be True (in latter case, given that class is also verified initialized)

fun *icheck* :: 'm prog ⇒ cname ⇒ 'a exp ⇒ bool **where**

icheck P C' (new C) = (C' = C) |

icheck P D' (C_sF{D}) = ((D' = D) ∧ (∃ T. P ⊢ C has F,Static:T in D)) |

icheck P D' (C_sF{D}:(Val v)) = ((D' = D) ∧ (∃ T. P ⊢ C has F,Static:T in D)) |

icheck P D (C_sM(es)) = ((∃ vs. es = map Val vs) ∧ (∃ Ts T m. P ⊢ C sees M,Static:Ts→T = m in D)) |

icheck P - = False

lemma *nichck-SFAss-nonVal*: val-of e₂ = None ⇒ ¬*icheck* P C' (C_sF{D} := (e₂::'a exp))

by(rule notI, cases e₂, auto)

inductive-set

red :: J-prog ⇒ ((expr × state × bool) × (expr × state × bool)) set

and *reds* :: J-prog ⇒ ((expr list × state × bool) × (expr list × state × bool)) set

and *red'* :: J-prog ⇒ expr ⇒ state ⇒ bool ⇒ expr ⇒ state ⇒ bool ⇒ bool

(⟨- ⊢ ((1⟨-,/,-/⟩) →/ (1⟨-,/,-/⟩))⟩ [51,0,0,0,0,0,0] 81)

and *reds'* :: J-prog ⇒ expr list ⇒ state ⇒ bool ⇒ expr list ⇒ state ⇒ bool ⇒ bool

(⟨- ⊢ ((1⟨-,/,-/⟩) [→]/ (1⟨-,/,-/⟩))⟩ [51,0,0,0,0,0,0] 81)

for P :: J-prog

where

P ⊢ ⟨e,s,b⟩ → ⟨e',s',b'⟩ ≡ ((e,s,b), e',s',b') ∈ *red* P

| P ⊢ ⟨es,s,b⟩ [→] ⟨es',s',b'⟩ ≡ ((es,s,b), es',s',b') ∈ *reds* P

| *RedNew*:

[[new-Addr h = Some a; P ⊢ C has-fields FDTs; h' = h(a→blank P C)]]

⇒ P ⊢ ⟨new C, (h,l,sh), True⟩ → ⟨addr a, (h',l,sh), False⟩

| *RedNewFail*:

[[new-Addr h = None; is-class P C]]

⇒ P ⊢ ⟨new C, (h,l,sh), True⟩ → ⟨THROW OutOfMemory, (h,l,sh), False⟩

| *NewInitDoneRed*:

sh C = Some (sfs, Done) ⇒

P ⊢ ⟨new C, (h,l,sh), False⟩ → ⟨new C, (h,l,sh), True⟩

| *NewInitRed*:

[[∄ sfs. sh C = Some (sfs, Done); is-class P C]]

⇒ P ⊢ ⟨new C,(h,l,sh),False⟩ → ⟨INIT C ([C],False) ← new C,(h,l,sh),False⟩

| *CastRed*:

P ⊢ ⟨e,s,b⟩ → ⟨e',s',b'⟩ ⇒

P ⊢ ⟨Cast C e, s, b⟩ → ⟨Cast C e', s', b'⟩

| *RedCastNull*:

P ⊢ ⟨Cast C null, s, b⟩ → ⟨null,s,b⟩

- | *RedCast*:

$$\llbracket h a = \text{Some}(D,fs); P \vdash D \preceq^* C \rrbracket$$

$$\implies P \vdash \langle \text{Cast } C \text{ (addr } a), (h,l,sh), b \rangle \rightarrow \langle \text{addr } a, (h,l,sh), b \rangle$$
- | *RedCastFail*:

$$\llbracket h a = \text{Some}(D,fs); \neg P \vdash D \preceq^* C \rrbracket$$

$$\implies P \vdash \langle \text{Cast } C \text{ (addr } a), (h,l,sh), b \rangle \rightarrow \langle \text{THROW ClassCast}, (h,l,sh), b \rangle$$
- | *BinOpRed1*:

$$P \vdash \langle e,s,b \rangle \rightarrow \langle e',s',b' \rangle \implies$$

$$P \vdash \langle e \text{ «bop» } e_2, s, b \rangle \rightarrow \langle e' \text{ «bop» } e_2, s', b' \rangle$$
- | *BinOpRed2*:

$$P \vdash \langle e,s,b \rangle \rightarrow \langle e',s',b' \rangle \implies$$

$$P \vdash \langle (\text{Val } v_1) \text{ «bop» } e, s, b \rangle \rightarrow \langle (\text{Val } v_1) \text{ «bop» } e', s', b' \rangle$$
- | *RedBinOp*:

$$\text{binop}(bop,v_1,v_2) = \text{Some } v \implies$$

$$P \vdash \langle (\text{Val } v_1) \text{ «bop» } (\text{Val } v_2), s, b \rangle \rightarrow \langle \text{Val } v, s, b \rangle$$
- | *RedVar*:

$$l V = \text{Some } v \implies$$

$$P \vdash \langle \text{Var } V, (h,l,sh), b \rangle \rightarrow \langle \text{Val } v, (h,l,sh), b \rangle$$
- | *LAssRed*:

$$P \vdash \langle e,s,b \rangle \rightarrow \langle e',s',b' \rangle \implies$$

$$P \vdash \langle V := e, s, b \rangle \rightarrow \langle V := e', s', b' \rangle$$
- | *RedLAss*:

$$P \vdash \langle V := (\text{Val } v), (h,l,sh), b \rangle \rightarrow \langle \text{unit}, (h,l(V \mapsto v), sh), b \rangle$$
- | *FAccRed*:

$$P \vdash \langle e,s,b \rangle \rightarrow \langle e',s',b' \rangle \implies$$

$$P \vdash \langle e \cdot F\{D\}, s, b \rangle \rightarrow \langle e' \cdot F\{D\}, s', b' \rangle$$
- | *RedFAcc*:

$$\llbracket h a = \text{Some}(C,fs); fs(F,D) = \text{Some } v;$$

$$P \vdash C \text{ has } F, \text{NonStatic}:t \text{ in } D \rrbracket$$

$$\implies P \vdash \langle (\text{addr } a) \cdot F\{D\}, (h,l,sh), b \rangle \rightarrow \langle \text{Val } v, (h,l,sh), b \rangle$$
- | *RedFAccNull*:

$$P \vdash \langle \text{null} \cdot F\{D\}, s, b \rangle \rightarrow \langle \text{THROW NullPointerException}, s, b \rangle$$
- | *RedFAccNone*:

$$\llbracket h a = \text{Some}(C,fs); \neg(\exists b t. P \vdash C \text{ has } F, b:t \text{ in } D) \rrbracket$$

$$\implies P \vdash \langle (\text{addr } a) \cdot F\{D\}, (h,l,sh), b \rangle \rightarrow \langle \text{THROW NoSuchFieldError}, (h,l,sh), b \rangle$$
- | *RedFAccStatic*:

$$\llbracket h a = \text{Some}(C,fs); P \vdash C \text{ has } F, \text{Static}:t \text{ in } D \rrbracket$$

$$\implies P \vdash \langle (\text{addr } a) \cdot F\{D\}, (h,l,sh), b \rangle \rightarrow \langle \text{THROW IncompatibleClassChangeError}, (h,l,sh), b \rangle$$
- | *RedSFAcc*:

$$\llbracket P \vdash C \text{ has } F, \text{Static}:t \text{ in } D;$$

$$sh D = \text{Some } (sfs,i);$$

$$\begin{array}{l} \text{sfs } F = \text{Some } v \text{]} \\ \implies P \vdash \langle C \cdot_s F\{D\}, (h, l, sh), \text{True} \rangle \rightarrow \langle \text{Val } v, (h, l, sh), \text{False} \rangle \end{array}$$

$$\begin{array}{l} | \text{SFAccInitDoneRed:} \\ \llbracket P \vdash C \text{ has } F, \text{Static}:t \text{ in } D; \\ \quad sh \ D = \text{Some } (\text{sfs}, \text{Done}) \rrbracket \\ \implies P \vdash \langle C \cdot_s F\{D\}, (h, l, sh), \text{False} \rangle \rightarrow \langle C \cdot_s F\{D\}, (h, l, sh), \text{True} \rangle \end{array}$$

$$\begin{array}{l} | \text{SFAccInitRed:} \\ \llbracket P \vdash C \text{ has } F, \text{Static}:t \text{ in } D; \\ \quad \nexists \text{sfs. } sh \ D = \text{Some } (\text{sfs}, \text{Done}) \rrbracket \\ \implies P \vdash \langle C \cdot_s F\{D\}, (h, l, sh), \text{False} \rangle \rightarrow \langle \text{INIT } D \ ([D], \text{False}) \leftarrow C \cdot_s F\{D\}, (h, l, sh), \text{False} \rangle \end{array}$$

$$\begin{array}{l} | \text{RedSFAccNone:} \\ \neg(\exists b \ t. P \vdash C \text{ has } F, b:t \text{ in } D) \\ \implies P \vdash \langle C \cdot_s F\{D\}, (h, l, sh), b \rangle \rightarrow \langle \text{THROW } \text{NoSuchFieldError}, (h, l, sh), \text{False} \rangle \end{array}$$

$$\begin{array}{l} | \text{RedSFAccNonStatic:} \\ P \vdash C \text{ has } F, \text{NonStatic}:t \text{ in } D \\ \implies P \vdash \langle C \cdot_s F\{D\}, (h, l, sh), b \rangle \rightarrow \langle \text{THROW } \text{IncompatibleClassChangeError}, (h, l, sh), \text{False} \rangle \end{array}$$

$$\begin{array}{l} | \text{FAssRed1:} \\ P \vdash \langle e, s, b \rangle \rightarrow \langle e', s', b' \rangle \implies \\ P \vdash \langle e \cdot F\{D\} := e_2, s, b \rangle \rightarrow \langle e' \cdot F\{D\} := e_2, s', b' \rangle \end{array}$$

$$\begin{array}{l} | \text{FAssRed2:} \\ P \vdash \langle e, s, b \rangle \rightarrow \langle e', s', b' \rangle \implies \\ P \vdash \langle \text{Val } v \cdot F\{D\} := e, s, b \rangle \rightarrow \langle \text{Val } v \cdot F\{D\} := e', s', b' \rangle \end{array}$$

$$\begin{array}{l} | \text{RedFAss:} \\ \llbracket P \vdash C \text{ has } F, \text{NonStatic}:t \text{ in } D; h \ a = \text{Some}(C, fs) \rrbracket \implies \\ P \vdash \langle (\text{addr } a) \cdot F\{D\} := (\text{Val } v), (h, l, sh), b \rangle \rightarrow \langle \text{unit}, (h(a \mapsto (C, fs((F, D) \mapsto v))), l, sh), b \rangle \end{array}$$

$$\begin{array}{l} | \text{RedFAssNull:} \\ P \vdash \langle \text{null} \cdot F\{D\} := \text{Val } v, s, b \rangle \rightarrow \langle \text{THROW } \text{NullPointer}, s, b \rangle \end{array}$$

$$\begin{array}{l} | \text{RedFAssNone:} \\ \llbracket h \ a = \text{Some}(C, fs); \neg(\exists b \ t. P \vdash C \text{ has } F, b:t \text{ in } D) \rrbracket \\ \implies P \vdash \langle (\text{addr } a) \cdot F\{D\} := (\text{Val } v), (h, l, sh), b \rangle \rightarrow \langle \text{THROW } \text{NoSuchFieldError}, (h, l, sh), b \rangle \end{array}$$

$$\begin{array}{l} | \text{RedFAssStatic:} \\ \llbracket h \ a = \text{Some}(C, fs); P \vdash C \text{ has } F, \text{Static}:t \text{ in } D \rrbracket \\ \implies P \vdash \langle (\text{addr } a) \cdot F\{D\} := (\text{Val } v), (h, l, sh), b \rangle \rightarrow \langle \text{THROW } \text{IncompatibleClassChangeError}, (h, l, sh), b \rangle \end{array}$$

$$\begin{array}{l} | \text{SFAssRed:} \\ P \vdash \langle e, s, b \rangle \rightarrow \langle e', s', b' \rangle \implies \\ P \vdash \langle C \cdot_s F\{D\} := e, s, b \rangle \rightarrow \langle C \cdot_s F\{D\} := e', s', b' \rangle \end{array}$$

$$\begin{array}{l} | \text{RedSFAss:} \\ \llbracket P \vdash C \text{ has } F, \text{Static}:t \text{ in } D; \\ \quad sh \ D = \text{Some}(\text{sfs}, i); \\ \quad \text{sfs}' = \text{sfs}(F \mapsto v); sh' = sh(D \mapsto (\text{sfs}', i)) \rrbracket \\ \implies P \vdash \langle C \cdot_s F\{D\} := (\text{Val } v), (h, l, sh), \text{True} \rangle \rightarrow \langle \text{unit}, (h, l, sh'), \text{False} \rangle \end{array}$$

- | *SFAssInitDoneRed*:
 $\llbracket P \vdash C \text{ has } F, \text{Static}:t \text{ in } D;$
 $\quad sh D = \text{Some}(sfs, \text{Done}) \rrbracket$
 $\implies P \vdash \langle C \cdot_s F \{D\} := (\text{Val } v), (h, l, sh), \text{False} \rangle \rightarrow \langle C \cdot_s F \{D\} := (\text{Val } v), (h, l, sh), \text{True} \rangle$
- | *SFAssInitRed*:
 $\llbracket P \vdash C \text{ has } F, \text{Static}:t \text{ in } D;$
 $\quad \nexists sfs. sh D = \text{Some}(sfs, \text{Done}) \rrbracket$
 $\implies P \vdash \langle C \cdot_s F \{D\} := (\text{Val } v), (h, l, sh), \text{False} \rangle \rightarrow \langle \text{INIT } D ([D], \text{False}) \leftarrow C \cdot_s F \{D\} := (\text{Val } v), (h, l, sh), \text{False} \rangle$
- | *RedSFAssNone*:
 $\neg(\exists b t. P \vdash C \text{ has } F, b:t \text{ in } D)$
 $\implies P \vdash \langle C \cdot_s F \{D\} := (\text{Val } v), s, b \rangle \rightarrow \langle \text{THROW NoSuchFieldError}, s, \text{False} \rangle$
- | *RedSFAssNonStatic*:
 $P \vdash C \text{ has } F, \text{NonStatic}:t \text{ in } D$
 $\implies P \vdash \langle C \cdot_s F \{D\} := (\text{Val } v), s, b \rangle \rightarrow \langle \text{THROW IncompatibleClassChangeError}, s, \text{False} \rangle$
- | *CallObj*:
 $P \vdash \langle e, s, b \rangle \rightarrow \langle e', s', b' \rangle \implies$
 $P \vdash \langle e \cdot M(es), s, b \rangle \rightarrow \langle e' \cdot M(es), s', b' \rangle$
- | *CallParams*:
 $P \vdash \langle es, s, b \rangle [\rightarrow] \langle es', s', b' \rangle \implies$
 $P \vdash \langle (\text{Val } v) \cdot M(es), s, b \rangle \rightarrow \langle (\text{Val } v) \cdot M(es'), s', b' \rangle$
- | *RedCall*:
 $\llbracket h a = \text{Some}(C.fs); P \vdash C \text{ sees } M, \text{NonStatic}:Ts \rightarrow T = (pns, \text{body}) \text{ in } D; \text{size } vs = \text{size } pns; \text{size } Ts = \text{size } pns \rrbracket$
 $\implies P \vdash \langle (\text{addr } a) \cdot M(\text{map Val } vs), (h, l, sh), b \rangle \rightarrow \langle \text{blocks}(\text{this}\#pns, \text{Class } D\#Ts, \text{Addr } a\#vs, \text{body}), (h, l, sh), b \rangle$
- | *RedCallNull*:
 $P \vdash \langle \text{null} \cdot M(\text{map Val } vs), s, b \rangle \rightarrow \langle \text{THROW NullPointer}, s, b \rangle$
- | *RedCallNone*:
 $\llbracket h a = \text{Some}(C.fs); \neg(\exists b Ts T m D. P \vdash C \text{ sees } M, b:Ts \rightarrow T = m \text{ in } D) \rrbracket$
 $\implies P \vdash \langle (\text{addr } a) \cdot M(\text{map Val } vs), (h, l, sh), b \rangle \rightarrow \langle \text{THROW NoSuchMethodError}, (h, l, sh), b \rangle$
- | *RedCallStatic*:
 $\llbracket h a = \text{Some}(C.fs); P \vdash C \text{ sees } M, \text{Static}:Ts \rightarrow T = m \text{ in } D \rrbracket$
 $\implies P \vdash \langle (\text{addr } a) \cdot M(\text{map Val } vs), (h, l, sh), b \rangle \rightarrow \langle \text{THROW IncompatibleClassChangeError}, (h, l, sh), b \rangle$
- | *SCallParams*:
 $P \vdash \langle es, s, b \rangle [\rightarrow] \langle es', s', b' \rangle \implies$
 $P \vdash \langle C \cdot_s M(es), s, b \rangle \rightarrow \langle C \cdot_s M(es'), s', b' \rangle$
- | *RedSCall*:
 $\llbracket P \vdash C \text{ sees } M, \text{Static}:Ts \rightarrow T = (pns, \text{body}) \text{ in } D;$
 $\quad \text{length } vs = \text{length } pns; \text{size } Ts = \text{size } pns \rrbracket$
 $\implies P \vdash \langle C \cdot_s M(\text{map Val } vs), s, \text{True} \rangle \rightarrow \langle \text{blocks}(pns, Ts, vs, \text{body}), s, \text{False} \rangle$
- | *SCallInitDoneRed*:
 $\llbracket P \vdash C \text{ sees } M, \text{Static}:Ts \rightarrow T = (pns, \text{body}) \text{ in } D;$

$$\begin{aligned} & \llbracket sh\ D = Some(sfs, Done) \vee (M = clinit \wedge sh\ D = Some(sfs, Processing)) \rrbracket \\ \implies & P \vdash \langle C \cdot_s M(\text{map Val } vs), (h, l, sh), False \rangle \rightarrow \langle C \cdot_s M(\text{map Val } vs), (h, l, sh), True \rangle \end{aligned}$$

| *SCallInitRed:*

$$\begin{aligned} & \llbracket P \vdash C \text{ sees } M, \text{Static}: Ts \rightarrow T = (pns, body) \text{ in } D; \\ & \quad \nexists sfs. sh\ D = Some(sfs, Done); M \neq clinit \rrbracket \\ \implies & P \vdash \langle C \cdot_s M(\text{map Val } vs), (h, l, sh), False \rangle \rightarrow \langle INIT\ D\ ([D], False) \leftarrow C \cdot_s M(\text{map Val } vs), (h, l, sh), False \rangle \end{aligned}$$

| *RedSCallNone:*

$$\begin{aligned} & \llbracket \neg(\exists b\ Ts\ T\ m\ D. P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D) \rrbracket \\ \implies & P \vdash \langle C \cdot_s M(\text{map Val } vs), s, b \rangle \rightarrow \langle THROW\ NoSuchMethodError, s, False \rangle \end{aligned}$$

| *RedSCallNonStatic:*

$$\begin{aligned} & \llbracket P \vdash C \text{ sees } M, \text{NonStatic}: Ts \rightarrow T = m \text{ in } D \rrbracket \\ \implies & P \vdash \langle C \cdot_s M(\text{map Val } vs), s, b \rangle \rightarrow \langle THROW\ IncompatibleClassChangeError, s, False \rangle \end{aligned}$$

| *BlockRedNone:*

$$\begin{aligned} & \llbracket P \vdash \langle e, (h, l(V := None), sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle; l' V = None; \neg \text{assigned } V\ e \rrbracket \\ \implies & P \vdash \langle \{V:T; e\}, (h, l, sh), b \rangle \rightarrow \langle \{V:T; e'\}, (h', l'(V := l V), sh'), b' \rangle \end{aligned}$$

| *BlockRedSome:*

$$\begin{aligned} & \llbracket P \vdash \langle e, (h, l(V := None), sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle; l' V = \text{Some } v; \neg \text{assigned } V\ e \rrbracket \\ \implies & P \vdash \langle \{V:T; e\}, (h, l, sh), b \rangle \rightarrow \langle \{V:T := \text{Val } v; e'\}, (h', l'(V := l V), sh'), b' \rangle \end{aligned}$$

| *InitBlockRed:*

$$\begin{aligned} & \llbracket P \vdash \langle e, (h, l(V \mapsto v), sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle; l' V = \text{Some } v' \rrbracket \\ \implies & P \vdash \langle \{V:T := \text{Val } v; e\}, (h, l, sh), b \rangle \rightarrow \langle \{V:T := \text{Val } v'; e'\}, (h', l'(V := l V), sh'), b' \rangle \end{aligned}$$

| *RedBlock:*

$$P \vdash \langle \{V:T; \text{Val } u\}, s, b \rangle \rightarrow \langle \text{Val } u, s, b \rangle$$

| *RedInitBlock:*

$$P \vdash \langle \{V:T := \text{Val } v; \text{Val } u\}, s, b \rangle \rightarrow \langle \text{Val } u, s, b \rangle$$

| *SeqRed:*

$$\begin{aligned} & P \vdash \langle e, s, b \rangle \rightarrow \langle e', s', b' \rangle \implies \\ & P \vdash \langle e;;e_2, s, b \rangle \rightarrow \langle e';;e_2, s', b' \rangle \end{aligned}$$

| *RedSeq:*

$$P \vdash \langle (\text{Val } v);;e_2, s, b \rangle \rightarrow \langle e_2, s, b \rangle$$

| *CondRed:*

$$\begin{aligned} & P \vdash \langle e, s, b \rangle \rightarrow \langle e', s', b' \rangle \implies \\ & P \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s, b \rangle \rightarrow \langle \text{if } (e') e_1 \text{ else } e_2, s', b' \rangle \end{aligned}$$

| *RedCondT:*

$$P \vdash \langle \text{if } (\text{true}) e_1 \text{ else } e_2, s, b \rangle \rightarrow \langle e_1, s, b \rangle$$

| *RedCondF:*

$$P \vdash \langle \text{if } (\text{false}) e_1 \text{ else } e_2, s, b \rangle \rightarrow \langle e_2, s, b \rangle$$

| *RedWhile:*

$$P \vdash \langle \text{while}(b) c, s, b' \rangle \rightarrow \langle \text{if}(b) (c;;\text{while}(b) c) \text{ else } \text{unit}, s, b' \rangle$$

| *ThrowRed*:

$$\begin{aligned} P \vdash \langle e, s, b \rangle &\rightarrow \langle e', s', b' \rangle \implies \\ P \vdash \langle \text{throw } e, s, b \rangle &\rightarrow \langle \text{throw } e', s', b' \rangle \end{aligned}$$

| *RedThrowNull*:

$$P \vdash \langle \text{throw null}, s, b \rangle \rightarrow \langle \text{THROW NullPointer}, s, b \rangle$$

| *TryRed*:

$$\begin{aligned} P \vdash \langle e, s, b \rangle &\rightarrow \langle e', s', b' \rangle \implies \\ P \vdash \langle \text{try } e \text{ catch}(C V) e_2, s, b \rangle &\rightarrow \langle \text{try } e' \text{ catch}(C V) e_2, s', b' \rangle \end{aligned}$$

| *RedTry*:

$$P \vdash \langle \text{try } (\text{Val } v) \text{ catch}(C V) e_2, s, b \rangle \rightarrow \langle \text{Val } v, s, b \rangle$$

| *RedTryCatch*:

$$\begin{aligned} \llbracket \text{hp } s \ a = \text{Some}(D, fs); P \vdash D \preceq^* C \rrbracket \\ \implies P \vdash \langle \text{try } (\text{Throw } a) \text{ catch}(C V) e_2, s, b \rangle &\rightarrow \langle \{V: \text{Class } C := \text{addr } a; e_2\}, s, b \rangle \end{aligned}$$

| *RedTryFail*:

$$\begin{aligned} \llbracket \text{hp } s \ a = \text{Some}(D, fs); \neg P \vdash D \preceq^* C \rrbracket \\ \implies P \vdash \langle \text{try } (\text{Throw } a) \text{ catch}(C V) e_2, s, b \rangle &\rightarrow \langle \text{Throw } a, s, b \rangle \end{aligned}$$

| *ListRed1*:

$$\begin{aligned} P \vdash \langle e, s, b \rangle &\rightarrow \langle e', s', b' \rangle \implies \\ P \vdash \langle e \# es, s, b \rangle [\rightarrow] &\langle e' \# es, s', b' \rangle \end{aligned}$$

| *ListRed2*:

$$\begin{aligned} P \vdash \langle es, s, b \rangle [\rightarrow] &\langle es', s', b' \rangle \implies \\ P \vdash \langle \text{Val } v \# es, s, b \rangle [\rightarrow] &\langle \text{Val } v \# es', s', b' \rangle \end{aligned}$$

— Initialization procedure

| *RedInit*:

$$\neg \text{sub-RI } e \implies P \vdash \langle \text{INIT } C (\text{Nil}, b) \leftarrow e, s, b' \rangle \rightarrow \langle e, s, \text{icheck } P C e \rangle$$

| *InitNoneRed*:

$$\begin{aligned} \text{sh } C = \text{None} \\ \implies P \vdash \langle \text{INIT } C' (C \# Cs, \text{False}) \leftarrow e, (h, l, \text{sh}), b \rangle &\rightarrow \langle \text{INIT } C' (C \# Cs, \text{False}) \leftarrow e, (h, l, \text{sh}(C \mapsto (\text{sblank } P C, \text{Prepared}))), b \rangle \end{aligned}$$

| *RedInitDone*:

$$\begin{aligned} \text{sh } C = \text{Some}(sfs, \text{Done}) \\ \implies P \vdash \langle \text{INIT } C' (C \# Cs, \text{False}) \leftarrow e, (h, l, \text{sh}), b \rangle &\rightarrow \langle \text{INIT } C' (Cs, \text{True}) \leftarrow e, (h, l, \text{sh}), b \rangle \end{aligned}$$

| *RedInitProcessing*:

$$\begin{aligned} \text{sh } C = \text{Some}(sfs, \text{Processing}) \\ \implies P \vdash \langle \text{INIT } C' (C \# Cs, \text{False}) \leftarrow e, (h, l, \text{sh}), b \rangle &\rightarrow \langle \text{INIT } C' (Cs, \text{True}) \leftarrow e, (h, l, \text{sh}), b \rangle \end{aligned}$$

| *RedInitError*:

$$\begin{aligned} \text{sh } C = \text{Some}(sfs, \text{Error}) \\ \implies P \vdash \langle \text{INIT } C' (C \# Cs, \text{False}) \leftarrow e, (h, l, \text{sh}), b \rangle &\rightarrow \langle \text{RI } (C, \text{THROW NoClassDefFoundError}); Cs \\ \leftarrow e, (h, l, \text{sh}), b \rangle \end{aligned}$$

| *InitObjectRed*:

$\llbracket sh\ C = Some(sfs, Prepared);$
 $\quad C = Object;$
 $\quad sh' = sh(C \mapsto (sfs, Processing)) \rrbracket$
 $\implies P \vdash \langle INIT\ C' (C \# Cs, False) \leftarrow e, (h, l, sh), b \rangle \rightarrow \langle INIT\ C' (C \# Cs, True) \leftarrow e, (h, l, sh'), b \rangle$

$| InitNonObjectSuperRed:$
 $\llbracket sh\ C = Some(sfs, Prepared);$
 $\quad C \neq Object;$
 $\quad class\ P\ C = Some\ (D, r);$
 $\quad sh' = sh(C \mapsto (sfs, Processing)) \rrbracket$
 $\implies P \vdash \langle INIT\ C' (C \# Cs, False) \leftarrow e, (h, l, sh), b \rangle \rightarrow \langle INIT\ C' (D \# C \# Cs, False) \leftarrow e, (h, l, sh'), b \rangle$

$| RedInitRInit:$
 $P \vdash \langle INIT\ C' (C \# Cs, True) \leftarrow e, (h, l, sh), b \rangle \rightarrow \langle RI\ (C, C \cdot_s\ clinit(\llbracket \rrbracket)); Cs \leftarrow e, (h, l, sh), b \rangle$

$| RInitRed:$
 $P \vdash \langle e, s, b \rangle \rightarrow \langle e', s', b' \rangle \implies$
 $P \vdash \langle RI\ (C, e); Cs \leftarrow e_0, s, b \rangle \rightarrow \langle RI\ (C, e'); Cs \leftarrow e_0, s', b' \rangle$

$| RedRInit:$
 $\llbracket sh\ C = Some\ (sfs, i);$
 $\quad sh' = sh(C \mapsto (sfs, Done));$
 $\quad C' = last(C \# Cs) \rrbracket \implies$
 $P \vdash \langle RI\ (C, Val\ v); Cs \leftarrow e, (h, l, sh), b \rangle \rightarrow \langle INIT\ C' (Cs, True) \leftarrow e, (h, l, sh'), b \rangle$

— Exception propagation

$| CastThrow: P \vdash \langle Cast\ C\ (throw\ e), s, b \rangle \rightarrow \langle throw\ e, s, b \rangle$
 $| BinOpThrow1: P \vdash \langle (throw\ e) \llbracket bop \rrbracket e_2, s, b \rangle \rightarrow \langle throw\ e, s, b \rangle$
 $| BinOpThrow2: P \vdash \langle (Val\ v_1) \llbracket bop \rrbracket (throw\ e), s, b \rangle \rightarrow \langle throw\ e, s, b \rangle$
 $| LAssThrow: P \vdash \langle V := (throw\ e), s, b \rangle \rightarrow \langle throw\ e, s, b \rangle$
 $| FAccThrow: P \vdash \langle (throw\ e) \cdot F\{D\}, s, b \rangle \rightarrow \langle throw\ e, s, b \rangle$
 $| FAssThrow1: P \vdash \langle (throw\ e) \cdot F\{D\} := e_2, s, b \rangle \rightarrow \langle throw\ e, s, b \rangle$
 $| FAssThrow2: P \vdash \langle Val\ v \cdot F\{D\} := (throw\ e), s, b \rangle \rightarrow \langle throw\ e, s, b \rangle$
 $| SFAssThrow: P \vdash \langle C \cdot_s F\{D\} := (throw\ e), s, b \rangle \rightarrow \langle throw\ e, s, b \rangle$
 $| CallThrowObj: P \vdash \langle (throw\ e) \cdot M(es), s, b \rangle \rightarrow \langle throw\ e, s, b \rangle$
 $| CallThrowParams: \llbracket es = map\ Val\ vs\ @\ throw\ e\ \# \ es' \rrbracket \implies P \vdash \langle (Val\ v) \cdot M(es), s, b \rangle \rightarrow \langle throw\ e, s, b \rangle$
 $| SCallThrowParams: \llbracket es = map\ Val\ vs\ @\ throw\ e\ \# \ es' \rrbracket \implies P \vdash \langle C \cdot_s M(es), s, b \rangle \rightarrow \langle throw\ e, s, b \rangle$
 $| BlockThrow: P \vdash \langle \{V:T; Throw\ a\}, s, b \rangle \rightarrow \langle Throw\ a, s, b \rangle$
 $| InitBlockThrow: P \vdash \langle \{V:T := Val\ v; Throw\ a\}, s, b \rangle \rightarrow \langle Throw\ a, s, b \rangle$
 $| SeqThrow: P \vdash \langle (throw\ e); e_2, s, b \rangle \rightarrow \langle throw\ e, s, b \rangle$
 $| CondThrow: P \vdash \langle if\ (throw\ e)\ e_1\ else\ e_2, s, b \rangle \rightarrow \langle throw\ e, s, b \rangle$
 $| ThrowThrow: P \vdash \langle throw(throw\ e), s, b \rangle \rightarrow \langle throw\ e, s, b \rangle$
 $| RInitInitThrow: \llbracket sh\ C = Some(sfs, i); sh' = sh(C \mapsto (sfs, Error)) \rrbracket \implies$
 $\quad P \vdash \langle RI\ (C, Throw\ a); D \# Cs \leftarrow e, (h, l, sh), b \rangle \rightarrow \langle RI\ (D, Throw\ a); Cs \leftarrow e, (h, l, sh'), b \rangle$
 $| RInitThrow: \llbracket sh\ C = Some(sfs, i); sh' = sh(C \mapsto (sfs, Error)) \rrbracket \implies$
 $\quad P \vdash \langle RI\ (C, Throw\ a); Nil \leftarrow e, (h, l, sh), b \rangle \rightarrow \langle Throw\ a, (h, l, sh'), b \rangle$

1.18.1 The reflexive transitive closure

abbreviation

$Step :: J\ prog \Rightarrow expr \Rightarrow state \Rightarrow bool \Rightarrow expr \Rightarrow state \Rightarrow bool \Rightarrow bool$

$(\langle \cdot \vdash ((1\langle -, /-, /- \rangle) \rightarrow^* / (1\langle -, /-, /- \rangle)) \rangle [51, 0, 0, 0, 0, 0, 0] 81)$
where $P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle \equiv ((e, s, b), e', s', b') \in (\text{red } P)^*$

abbreviation

$\text{Steps} :: J\text{-prog} \Rightarrow \text{expr list} \Rightarrow \text{state} \Rightarrow \text{bool} \Rightarrow \text{expr list} \Rightarrow \text{state} \Rightarrow \text{bool} \Rightarrow \text{bool}$
 $(\langle \cdot \vdash ((1\langle -, /-, /- \rangle) [\rightarrow]^* / (1\langle -, /-, /- \rangle)) \rangle [51, 0, 0, 0, 0, 0, 0] 81)$
where $P \vdash \langle es, s, b \rangle [\rightarrow]^* \langle es', s', b' \rangle \equiv ((es, s, b), es', s', b') \in (\text{reds } P)^*$

lemmas *converse-rtrancl-induct3* =

converse-rtrancl-induct [of (ax, ay, az) (bx, by, bz) , *split-format* (complete),
consumes 1, *case-names refl step*]

lemma *converse-rtrancl-induct-red*[*consumes 1*]:

assumes $P \vdash \langle e, (h, l, sh), b \rangle \rightarrow^* \langle e', (h', l', sh'), b' \rangle$

and $\bigwedge e h l sh b. R e h l sh b e h l sh b$

and $\bigwedge e_0 h_0 l_0 sh_0 b_0 e_1 h_1 l_1 sh_1 b_1 e' h' l' sh' b'$

$\llbracket P \vdash \langle e_0, (h_0, l_0, sh_0), b_0 \rangle \rightarrow \langle e_1, (h_1, l_1, sh_1), b_1 \rangle; R e_1 h_1 l_1 sh_1 b_1 e' h' l' sh' b' \rrbracket$
 $\implies R e_0 h_0 l_0 sh_0 b_0 e' h' l' sh' b'$

shows $R e h l sh b e' h' l' sh' b'$

1.18.2 Some easy lemmas

lemma [*iff*]: $\neg P \vdash \langle [], s, b \rangle [\rightarrow] \langle es', s', b' \rangle$

lemma [*iff*]: $\neg P \vdash \langle \text{Val } v, s, b \rangle \rightarrow \langle e', s', b' \rangle$

lemma *val-no-step*: $\text{val-of } e = \lfloor v \rfloor \implies \neg P \vdash \langle e, s, b \rangle \rightarrow \langle e', s', b' \rangle$

lemma [*iff*]: $\neg P \vdash \langle \text{Throw } a, s, b \rangle \rightarrow \langle e', s', b' \rangle$

lemma *map-Vals-no-step* [*iff*]: $\neg P \vdash \langle \text{map Val } vs, s, b \rangle [\rightarrow] \langle es', s', b' \rangle$

lemma *vals-no-step*: $\text{map-vals-of } es = \lfloor vs \rfloor \implies \neg P \vdash \langle es, s, b \rangle [\rightarrow] \langle es', s', b' \rangle$

lemma *vals-throw-no-step* [*iff*]: $\neg P \vdash \langle \text{map Val } vs @ \text{Throw } a \# es, s, b \rangle [\rightarrow] \langle es', s', b' \rangle$

lemma *lass-val-of-red*:

$\llbracket \text{lass-val-of } e = \lfloor a \rfloor; P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle \rrbracket$
 $\implies e' = \text{unit} \wedge h' = h \wedge l' = l(\text{fst } a \mapsto \text{snd } a) \wedge sh' = sh \wedge b = b'$

lemma *final-no-step* [*iff*]: $\text{final } e \implies \neg P \vdash \langle e, s, b \rangle \rightarrow \langle e', s', b' \rangle$

lemma *finals-no-step* [*iff*]: $\text{finals } es \implies \neg P \vdash \langle es, s, b \rangle [\rightarrow] \langle es', s', b' \rangle$

lemma *reds-final-same*:

$P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle \implies \text{final } e \implies e = e' \wedge s = s' \wedge b = b'$

proof(*induct rule: converse-rtrancl-induct3*)

case refl show ?*case by simp*

next

case (*step* $e_0 s_0 b_0 e_1 s_1 b_1$) **show** ?*case*

proof(*rule finalE[OF step.premis(1)]*)

fix v **assume** $e_0 = \text{Val } v$ **then show** ?*thesis using step by simp*

next

fix a **assume** $e_0 = \text{Throw } a$ **then show** ?*thesis using step by simp*

qed

qed

lemma *reds-throw*:

$P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle \implies (\bigwedge e_t. \text{throw-of } e = \lfloor e_t \rfloor \implies \exists e_t'. \text{throw-of } e' = \lfloor e_t' \rfloor)$

proof(*induct rule: converse-rtrancl-induct3*)

case refl then show ?*case by simp*

next

case (step e0 s0 b0 e1 s1 b1)
 then show ?case by(auto elim: red.cases)
 qed

lemma red-heat-incr: $P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle \implies h \sqsubseteq h'$
and reds-heat-incr: $P \vdash \langle es, (h, l, sh), b \rangle [\rightarrow] \langle es', (h', l', sh'), b' \rangle \implies h \sqsubseteq h'$

lemma red-lcl-incr: $P \vdash \langle e, (h_0, l_0, sh_0), b \rangle \rightarrow \langle e', (h_1, l_1, sh_1), b' \rangle \implies \text{dom } l_0 \subseteq \text{dom } l_1$

and reds-lcl-incr: $P \vdash \langle es, (h_0, l_0, sh_0), b \rangle [\rightarrow] \langle es', (h_1, l_1, sh_1), b' \rangle \implies \text{dom } l_0 \subseteq \text{dom } l_1$

lemma red-lcl-add: $P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle \implies (\bigwedge l_0. P \vdash \langle e, (h, l_0 ++ l, sh), b \rangle \rightarrow \langle e', (h', l_0 ++ l', sh'), b' \rangle)$

and reds-lcl-add: $P \vdash \langle es, (h, l, sh), b \rangle [\rightarrow] \langle es', (h', l', sh'), b' \rangle \implies (\bigwedge l_0. P \vdash \langle es, (h, l_0 ++ l, sh), b \rangle [\rightarrow] \langle es', (h', l_0 ++ l', sh'), b' \rangle)$

lemma Red-lcl-add:

assumes $P \vdash \langle e, (h, l, sh), b \rangle \rightarrow^* \langle e', (h', l', sh'), b' \rangle$ **shows** $P \vdash \langle e, (h, l_0 ++ l, sh), b \rangle \rightarrow^* \langle e', (h', l_0 ++ l', sh'), b' \rangle$

lemma **assumes** wf: wwf-J-prog P

shows red-proc-pres: $P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle$

$\implies \text{not-init } C e \implies sh C = [(sfs, Processing)] \implies \text{not-init } C e' \wedge (\exists sfs'. sh' C = [(sfs', Processing)])$

and reds-proc-pres: $P \vdash \langle es, (h, l, sh), b \rangle [\rightarrow] \langle es', (h', l', sh'), b' \rangle$

$\implies \text{not-inits } C es \implies sh C = [(sfs, Processing)] \implies \text{not-inits } C es' \wedge (\exists sfs'. sh' C = [(sfs', Processing)])$

1.19 Expression conformance properties

theory EConform

imports SmallStep BigStep

begin

lemma cons-to-append: $list \neq [] \longrightarrow (\exists ls. a \# list = ls @ [last list])$
by (metis append-butlast-last-id last-ConsR list.simps(3))

1.19.1 Initialization conformance

fun init-class :: 'm prog \Rightarrow 'a exp \Rightarrow cname option **where**

init-class P (new C) = Some C |

init-class P (C_sF{D}) = Some D |

init-class P (C_sF{D}:e₂) = Some D |

init-class P (C_sM(es)) = seeing-class P C M |

init-class - - = None

lemma ick-check-init-class: ick-check P C e \implies init-class P e = [C]

proof(induct e)

case (SFAss x1 x2 x3 e')

then show ?case by(case-tac e') auto

qed auto

— exp to take next small step (in particular, subexp that may contain initialization)

fun ss-exp :: 'a exp \Rightarrow 'a exp **and** ss-exps :: 'a exp list \Rightarrow 'a exp option **where**

ss-exp (new C) = new C

| ss-exp (Cast C e) = (case val-of e of Some v \Rightarrow Cast C e | - \Rightarrow ss-exp e)

| ss-exp (Val v) = Val v

$$\begin{aligned}
& | \text{ss-exp } (e_1 \ll \text{bop} \gg e_2) = (\text{case val-of } e_1 \text{ of Some } v \Rightarrow (\text{case val-of } e_2 \text{ of Some } v \Rightarrow e_1 \ll \text{bop} \gg e_2 \mid - \Rightarrow \\
& \text{ss-exp } e_2) \\
& \qquad \qquad \qquad | - \Rightarrow \text{ss-exp } e_1) \\
& | \text{ss-exp } (\text{Var } V) = \text{Var } V \\
& | \text{ss-exp } (\text{LAss } V e) = (\text{case val-of } e \text{ of Some } v \Rightarrow \text{LAss } V e \mid - \Rightarrow \text{ss-exp } e) \\
& | \text{ss-exp } (e \cdot F\{D\}) = (\text{case val-of } e \text{ of Some } v \Rightarrow e \cdot F\{D\} \mid - \Rightarrow \text{ss-exp } e) \\
& | \text{ss-exp } (C \cdot_s F\{D\}) = C \cdot_s F\{D\} \\
& | \text{ss-exp } (e_1 \cdot F\{D\} := e_2) = (\text{case val-of } e_1 \text{ of Some } v \Rightarrow (\text{case val-of } e_2 \text{ of Some } v \Rightarrow e_1 \cdot F\{D\} := e_2 \mid - \\
& \Rightarrow \text{ss-exp } e_2) \\
& \qquad \qquad \qquad | - \Rightarrow \text{ss-exp } e_1) \\
& | \text{ss-exp } (C \cdot_s F\{D\} := e_2) = (\text{case val-of } e_2 \text{ of Some } v \Rightarrow C \cdot_s F\{D\} := e_2 \mid - \Rightarrow \text{ss-exp } e_2) \\
& | \text{ss-exp } (e \cdot M(es)) = (\text{case val-of } e \text{ of Some } v \Rightarrow (\text{case map-vals-of } es \text{ of Some } t \Rightarrow e \cdot M(es) \mid - \Rightarrow \\
& \text{the(ss-exps } es)) \\
& \qquad \qquad \qquad | - \Rightarrow \text{ss-exp } e) \\
& | \text{ss-exp } (C \cdot_s M(es)) = (\text{case map-vals-of } es \text{ of Some } t \Rightarrow C \cdot_s M(es) \mid - \Rightarrow \text{the(ss-exps } es)) \\
& | \text{ss-exp } (\{V:T; e\}) = \text{ss-exp } e \\
& | \text{ss-exp } (e_1;;e_2) = (\text{case val-of } e_1 \text{ of Some } v \Rightarrow \text{ss-exp } e_2 \\
& \qquad \qquad \qquad | \text{None} \Rightarrow (\text{case lass-val-of } e_1 \text{ of Some } p \Rightarrow \text{ss-exp } e_2 \\
& \qquad \qquad \qquad \qquad \qquad \qquad | \text{None} \Rightarrow \text{ss-exp } e_1)) \\
& | \text{ss-exp } (\text{if } (b) e_1 \text{ else } e_2) = (\text{case bool-of } b \text{ of Some True} \Rightarrow \text{if } (b) e_1 \text{ else } e_2 \\
& \qquad \qquad \qquad | \text{Some False} \Rightarrow \text{if } (b) e_1 \text{ else } e_2 \\
& \qquad \qquad \qquad | - \Rightarrow \text{ss-exp } b) \\
& | \text{ss-exp } (\text{while } (b) e) = \text{while } (b) e \\
& | \text{ss-exp } (\text{throw } e) = (\text{case val-of } e \text{ of Some } v \Rightarrow \text{throw } e \mid - \Rightarrow \text{ss-exp } e) \\
& | \text{ss-exp } (\text{try } e_1 \text{ catch}(C V) e_2) = (\text{case val-of } e_1 \text{ of Some } v \Rightarrow \text{try } e_1 \text{ catch}(C V) e_2 \\
& \qquad \qquad \qquad | - \Rightarrow \text{ss-exp } e_1) \\
& | \text{ss-exp } (\text{INIT } C (Cs,b) \leftarrow e) = \text{INIT } C (Cs,b) \leftarrow e \\
& | \text{ss-exp } (\text{RI } (C,e);Cs \leftarrow e') = (\text{case val-of } e \text{ of Some } v \Rightarrow \text{RI } (C,e);Cs \leftarrow e' \mid - \Rightarrow \text{ss-exp } e) \\
& | \text{ss-exps}(\[]) = \text{None} \\
& | \text{ss-exps}(e\#es) = (\text{case val-of } e \text{ of Some } v \Rightarrow \text{ss-exps } es \mid - \Rightarrow \text{Some } (\text{ss-exp } e))
\end{aligned}$$

lemma *icheck-ss-exp*:

assumes *icheck* $P C e$ **shows** $\text{ss-exp } e = e$

using *assms*

proof(*cases* e)

case (*SFAss* $C F D e$) **then show** *?thesis* **using** *assms*

proof(*cases* e)**qed**(*auto*)

qed(*auto*)

lemma *ss-exps-Vals-None[simp]*:

$\text{ss-exps } (\text{map } \text{Val } vs) = \text{None}$

by(*induct* vs) (*auto*)

lemma *ss-exps-Vals-NoneI*:

$\text{ss-exps } es = \text{None} \implies \exists vs. es = \text{map } \text{Val } vs$

using *val-of-spec* **by**(*induct* es) (*auto*)

lemma *ss-exps-throw-nVal*:

$\ll \text{val-of } e = \text{None}; \text{ss-exps } (\text{map } \text{Val } vs @ \text{throw } e \# es') = [e'] \gg$

$\implies e' = \text{ss-exp } e$

by(*induct* vs) (*auto*)

lemma *ss-exps-throw-Val*:

$\ll \text{val-of } e = [a]; \text{ss-exps } (\text{map } \text{Val } vs @ \text{throw } e \# es') = [e'] \gg$

$\implies e' = \text{throw } e$
by(*induct vs*) (*auto*)

abbreviation *curr-init* :: 'm prog \Rightarrow 'a exp \Rightarrow cname option **where**

curr-init P e \equiv *init-class* P (ss-exp e)

abbreviation *curr-inits* :: 'm prog \Rightarrow 'a exp list \Rightarrow cname option **where**

curr-inits P es \equiv *case ss-exps es of Some e \Rightarrow init-class P e | - \Rightarrow None*

lemma *icheck-curr-init'*: $\bigwedge e'. \text{ss-exp } e = e' \implies \text{icheck } P \ C \ e' \implies \text{curr-init } P \ e = \lfloor C \rfloor$
and *icheck-curr-inits'*: $\bigwedge e. \text{ss-exps } es = \lfloor e \rfloor \implies \text{icheck } P \ C \ e \implies \text{curr-inits } P \ es = \lfloor C \rfloor$

proof(*induct rule: ss-exp-ss-exps-induct*)

qed(*simp-all add: icheck-init-class*)

lemma *icheck-curr-init*: *icheck* P C e' \implies *ss-exp* e = e' \implies *curr-init* P e = $\lfloor C \rfloor$

by(*rule icheck-curr-init'*)

lemma *icheck-curr-inits*: *icheck* P C e \implies *ss-exps* es = $\lfloor e \rfloor \implies$ *curr-inits* P es = $\lfloor C \rfloor$

by(*rule icheck-curr-inits'*)

definition *initPD* :: *sheap* \Rightarrow *cname* \Rightarrow *bool* **where**

initPD sh C \equiv \exists sfs i. sh C = *Some* (sfs, i) \wedge (i = *Done* \vee i = *Processing*)

— checks that *INIT* and *RI* conform and are only in the main computation

fun *iconf* :: *sheap* \Rightarrow 'a exp \Rightarrow *bool* **and** *iconfs* :: *sheap* \Rightarrow 'a exp list \Rightarrow *bool* **where**

iconf sh (*new* C) = *True*
| *iconf* sh (*Cast* C e) = *iconf* sh e
| *iconf* sh (*Val* v) = *True*
| *iconf* sh (e₁ «*bop*» e₂) = (*case val-of* e₁ of *Some* v \Rightarrow *iconf* sh e₂ | - \Rightarrow *iconf* sh e₁ \wedge \neg *sub-RI* e₂)
| *iconf* sh (*Var* V) = *True*
| *iconf* sh (*LAss* V e) = *iconf* sh e
| *iconf* sh (e·*F*{D}) = *iconf* sh e
| *iconf* sh (C·*sF*{D}) = *True*
| *iconf* sh (e₁·*F*{D}:=e₂) = (*case val-of* e₁ of *Some* v \Rightarrow *iconf* sh e₂ | - \Rightarrow *iconf* sh e₁ \wedge \neg *sub-RI* e₂)
| *iconf* sh (C·*sF*{D}:=e₂) = *iconf* sh e₂
| *iconf* sh (e·*M*(es)) = (*case val-of* e of *Some* v \Rightarrow *iconfs* sh es | - \Rightarrow *iconf* sh e \wedge \neg *sub-RIs* es)
| *iconf* sh (C·*sM*(es)) = *iconfs* sh es
| *iconf* sh ({V:T; e}) = *iconf* sh e
| *iconf* sh (e₁;;e₂) = (*case val-of* e₁ of *Some* v \Rightarrow *iconf* sh e₂
| *None* \Rightarrow (*case lass-val-of* e₁ of *Some* p \Rightarrow *iconf* sh e₂
| *None* \Rightarrow *iconf* sh e₁ \wedge \neg *sub-RI* e₂))
| *iconf* sh (*if* (b) e₁ *else* e₂) = (*iconf* sh b \wedge \neg *sub-RI* e₁ \wedge \neg *sub-RI* e₂)
| *iconf* sh (*while* (b) e) = (\neg *sub-RI* b \wedge \neg *sub-RI* e)
| *iconf* sh (*throw* e) = *iconf* sh e
| *iconf* sh (*try* e₁ *catch*(C V) e₂) = (*iconf* sh e₁ \wedge \neg *sub-RI* e₂)
| *iconf* sh (*INIT* C (Cs,b) \leftarrow e) = ((*case* Cs of *Nil* \Rightarrow *initPD* sh C | C'#Cs' \Rightarrow *last* Cs = C) \wedge \neg *sub-RI* e)
| *iconf* sh (*RI* (C,e);Cs \leftarrow e') = (*iconf* sh e \wedge \neg *sub-RI* e')
| *iconfs* sh ([]) = *True*
| *iconfs* sh (e#es) = (*case val-of* e of *Some* v \Rightarrow *iconfs* sh es | - \Rightarrow *iconf* sh e \wedge \neg *sub-RIs* es)

lemma *iconfs-map-throw*: *iconfs* sh (*map* *Val* vs @ *throw* e # es') \implies *iconf* sh e

by(*induct vs,auto*)

lemma *nsub-RI-icnf-aux*:

$(\neg \text{sub-RI } (e::'a \text{ exp}) \longrightarrow (\forall e'. e' \in \text{subexp } e \longrightarrow \neg \text{sub-RI } e' \longrightarrow \text{icnf sh } e') \longrightarrow \text{icnf sh } e)$
 $\wedge (\neg \text{sub-RIs } (es::'a \text{ exp list}) \longrightarrow (\forall e'. e' \in \text{subexps } es \longrightarrow \neg \text{sub-RI } e' \longrightarrow \text{icnf sh } e') \longrightarrow \text{iconfs sh } es)$

proof(*induct rule: sub-RI-sub-RIs.induct*) **qed**(*auto*)

lemma *nsub-RI-icnf-aux'*:

$(\bigwedge e'. \text{subexp-of } e' e \Longrightarrow \neg \text{sub-RI } e' \longrightarrow \text{icnf sh } e') \Longrightarrow (\neg \text{sub-RI } e \Longrightarrow \text{icnf sh } e)$

by(*simp add: nsub-RI-icnf-aux*)

lemma *nsub-RI-icnf*: $\neg \text{sub-RI } e \Longrightarrow \text{icnf sh } e$

and *nsub-RIs-iconfs*: $\neg \text{sub-RIs } es \Longrightarrow \text{iconfs sh } es$

proof –

let $?R = \lambda e. \neg \text{sub-RI } e \longrightarrow \text{icnf sh } e$

let $?Rs = \lambda es. \neg \text{sub-RIs } es \longrightarrow \text{iconfs sh } es$

have $(\forall e'. \text{subexp-of } e' e \longrightarrow ?R e') \wedge ?R e$

by(*rule subexp-induct[where ?Rs = ?Rs]; clarsimp simp: nsub-RI-icnf-aux*)

moreover have $(\forall e'. e' \in \text{subexps } es \longrightarrow ?R e') \wedge ?Rs es$

by(*rule subexps-induct; clarsimp simp: nsub-RI-icnf-aux*)

ultimately show $\neg \text{sub-RI } e \Longrightarrow \text{icnf sh } e$

and $\neg \text{sub-RIs } es \Longrightarrow \text{iconfs sh } es$ **by** *simp+*

qed

lemma *lass-val-of-icnf*: $\text{lass-val-of } e = \lfloor a \rfloor \Longrightarrow \text{icnf sh } e$

by(*drule lass-val-of-nsub-RI, erule nsub-RI-icnf*)

lemma *icheck-icnf*:

assumes *icheck P C e* **shows** *icnf sh e*

using *assms*

proof(*cases e*)

case (*SFAss C F D e*) **then show** *?thesis* **using** *assms*

proof(*cases e*)**qed**(*auto*)

next

case (*SCall C M es*) **then show** *?thesis* **using** *assms*

by (*auto simp: nsub-RIs-iconfs*)

next

qed(*auto*)

1.19.2 Indicator boolean conformance

definition *bconf* :: $'m \text{ prog} \Rightarrow \text{sheap} \Rightarrow 'a \text{ exp} \Rightarrow \text{bool} \Rightarrow \text{bool}$ ($\langle -, - \rangle \vdash_b '(-, -) \checkmark$) [51,51,0,0] 50)

where

$P, sh \vdash_b (e, b) \checkmark \equiv b \longrightarrow (\exists C. \text{icheck } P C (\text{ss-exp } e) \wedge \text{initPD } sh C)$

definition *bconfs* :: $'m \text{ prog} \Rightarrow \text{sheap} \Rightarrow 'a \text{ exp list} \Rightarrow \text{bool} \Rightarrow \text{bool}$ ($\langle -, - \rangle \vdash_b '(-, -) \checkmark$) [51,51,0,0] 50)

where

$P, sh \vdash_b (es, b) \checkmark \equiv b \longrightarrow (\exists C. (\text{icheck } P C (\text{the}(\text{ss-exps } es)) \wedge (\text{curr-inits } P es = \text{Some } C) \wedge \text{initPD } sh C))$

— *bconf* helper lemmas

lemma *bconf-nonVal[simp]*:

$P, sh \vdash_b (e, \text{True}) \checkmark \Longrightarrow \text{val-of } e = \text{None}$

by(cases e) (auto simp: bconf-def)

lemma bconfs-nonVals[simp]:

$P, sh \vdash_b (es, True) \checkmark \implies \text{map-vals-of } es = None$

by(induct es) (auto simp: bconfs-def)

lemma bconf-Cast[iff]:

$P, sh \vdash_b (Cast\ C\ e, b) \checkmark \iff P, sh \vdash_b (e, b) \checkmark$

by(cases b) (auto simp: bconf-def dest: val-of-spec)

lemma bconf-BinOp[iff]:

$P, sh \vdash_b (e1 \ll bop \gg e2, b) \checkmark$

$\iff (case\ \text{val-of } e1\ \text{of } Some\ v \Rightarrow P, sh \vdash_b (e2, b) \checkmark \mid - \Rightarrow P, sh \vdash_b (e1, b) \checkmark)$

by(cases b) (auto simp: bconf-def dest: val-of-spec)

lemma bconf-LAss[iff]:

$P, sh \vdash_b (LAss\ V\ e, b) \checkmark \iff P, sh \vdash_b (e, b) \checkmark$

by(cases b) (auto simp: bconf-def dest: val-of-spec)

lemma bconf-FAcc[iff]:

$P, sh \vdash_b (e \cdot F\{D\}, b) \checkmark \iff P, sh \vdash_b (e, b) \checkmark$

by(cases b) (auto simp: bconf-def dest: val-of-spec)

lemma bconf-FAss[iff]:

$P, sh \vdash_b (FAss\ e1\ F\ D\ e2, b) \checkmark$

$\iff (case\ \text{val-of } e1\ \text{of } Some\ v \Rightarrow P, sh \vdash_b (e2, b) \checkmark \mid - \Rightarrow P, sh \vdash_b (e1, b) \checkmark)$

by(cases b) (auto simp: bconf-def dest: val-of-spec)

lemma bconf-SFAss[iff]:

$\text{val-of } e2 = None \implies P, sh \vdash_b (SFAss\ C\ F\ D\ e2, b) \checkmark \iff P, sh \vdash_b (e2, b) \checkmark$

by(cases b) (auto simp: bconf-def)

lemma bconfs-Vals[iff]:

$P, sh \vdash_b (\text{map } Val\ vs, b) \checkmark \iff \neg b$

by(unfold bconfs-def) simp

lemma bconf-Call[iff]:

$P, sh \vdash_b (e \cdot M(es), b) \checkmark$

$\iff (case\ \text{val-of } e\ \text{of } Some\ v \Rightarrow P, sh \vdash_b (es, b) \checkmark \mid - \Rightarrow P, sh \vdash_b (e, b) \checkmark)$

proof(cases b)

case True

then show ?thesis

proof(cases ss-exps es)

case None

then obtain vs where es = map Val vs using ss-exps-Vals-NoneI by auto

then have mv: map-vals-of es = [vs] by simp

then show ?thesis by(auto simp: bconf-def) (simp add: bconfs-def)

next

case (Some a)

then show ?thesis by(auto simp: bconf-def) (auto simp: bconfs-def icheck-init-class)

qed

qed(simp add: bconf-def bconfs-def)

lemma bconf-SCall[iff]:

assumes *mvn*: *map-vals-of es = None*
shows $P, sh \vdash_b (C \cdot_s M(es), b) \checkmark \longleftrightarrow P, sh \vdash_b (es, b) \checkmark$
proof(*cases b*)
 case *True*
 then show *?thesis*
 proof(*cases ss-exps es*)
 case *None*
 then have $\exists vs. es = \text{map Val } vs$ **using** *ss-exps-Vals-NoneI* **by** *auto*
 then show *?thesis* **using** *mvn finals-def* **by** *clarsimp*
 next
 case (*Some a*)
 then show *?thesis* **by**(*auto simp: bconf-def*) (*auto simp: bconfs-def icheck-init-class*)
 qed
qed(*simp add: bconf-def bconfs-def*)

lemma *bconf-Cons*[*iff*]:
 $P, sh \vdash_b (e \# es, b) \checkmark$
 $\longleftrightarrow (\text{case val-of } e \text{ of } \text{Some } v \Rightarrow P, sh \vdash_b (es, b) \checkmark \mid - \Rightarrow P, sh \vdash_b (e, b) \checkmark)$
proof(*cases b*)
 case *True*
 then show *?thesis*
 proof(*cases ss-exps es*)
 case *None*
 then have $\exists vs. es = \text{map Val } vs$ **using** *ss-exps-Vals-NoneI* **by** *auto*
 then show *?thesis* **using** *None* **by**(*auto simp: bconf-def bconfs-def icheck-init-class*)
 next
 case (*Some a*)
 then show *?thesis* **by**(*auto simp: bconf-def bconfs-def icheck-init-class*)
 qed
qed(*simp add: bconf-def bconfs-def*)

lemma *bconf-InitBlock*[*iff*]:
 $P, sh \vdash_b (\{V:T; V:=\text{Val } v;; e_2\}, b) \checkmark \longleftrightarrow P, sh \vdash_b (e_2, b) \checkmark$
by(*cases b*) (*auto simp: bconf-def assigned-def*)

lemma *bconf-Block*[*iff*]:
 $P, sh \vdash_b (\{V:T; e\}, b) \checkmark \longleftrightarrow P, sh \vdash_b (e, b) \checkmark$
by(*cases b*) (*auto simp: bconf-def*)

lemma *bconf-Seq*[*iff*]:
 $P, sh \vdash_b (e1;;e2, b) \checkmark$
 $\longleftrightarrow (\text{case val-of } e1 \text{ of } \text{Some } v \Rightarrow P, sh \vdash_b (e2, b) \checkmark$
 $\quad \mid - \Rightarrow (\text{case lass-val-of } e1 \text{ of } \text{Some } p \Rightarrow P, sh \vdash_b (e2, b) \checkmark$
 $\quad \quad \quad \mid \text{None} \Rightarrow P, sh \vdash_b (e1, b) \checkmark))$
by(*cases b*) (*auto simp: bconf-def dest: val-of-spec lass-val-of-spec*)

lemma *bconf-Cond*[*iff*]:
 $P, sh \vdash_b (\text{if } (b) \ e_1 \text{ else } e_2, b') \checkmark \longleftrightarrow P, sh \vdash_b (b, b') \checkmark$
proof(*cases bool-of b*)
 case *None*
 then show *?thesis* **by**(*auto simp: bconf-def*)
next
 case (*Some a*)
 then show *?thesis* **by**(*case-tac a*) (*auto simp: bconf-def dest: bool-of-specT bool-of-specF*)

qed

lemma *bconf-While*[iff]:

$P, sh \vdash_b (\text{while } (b) \ e, b') \checkmark \longleftrightarrow \neg b'$
by(cases *b*) (auto simp: *bconf-def*)

lemma *bconf-Throw*[iff]:

$P, sh \vdash_b (\text{throw } e, b) \checkmark \longleftrightarrow P, sh \vdash_b (e, b) \checkmark$
by(cases *b*) (auto simp: *bconf-def* dest: *val-of-spec*)

lemma *bconf-Try*[iff]:

$P, sh \vdash_b (\text{try } e_1 \ \text{catch}(C \ V) \ e_2, b) \checkmark \longleftrightarrow P, sh \vdash_b (e_1, b) \checkmark$
by(cases *b*) (auto simp: *bconf-def* dest: *val-of-spec*)

lemma *bconf-INIT*[iff]:

$P, sh \vdash_b (\text{INIT } C \ (Cs, b') \leftarrow e, b) \checkmark \longleftrightarrow \neg b$
by(cases *b*) (auto simp: *bconf-def*)

lemma *bconf-RI*[iff]:

$P, sh \vdash_b (\text{RI}(C, e); Cs \leftarrow e', b) \checkmark \longleftrightarrow P, sh \vdash_b (e, b) \checkmark$
by(cases *b*) (auto simp: *bconf-def* dest: *val-of-spec*)

lemma *bconfs-map-throw*[iff]:

$P, sh \vdash_b (\text{map } \text{Val } vs \ @ \ \text{throw } e \ \# \ es', b) \checkmark \longleftrightarrow P, sh \vdash_b (e, b) \checkmark$
by(*induct vs*) auto

end

1.20 Progress of Small Step Semantics

theory *Progress*

imports *WellTypeRT DefAss ../Common/Conform EConform*

begin

lemma *final-addrE*:

$\llbracket P, E, h, sh \vdash e : \text{Class } C; \text{ final } e;$
 $\bigwedge a. e = \text{addr } a \implies R;$
 $\bigwedge a. e = \text{Throw } a \implies R \rrbracket \implies R$

lemma *finalRefE*:

$\llbracket P, E, h, sh \vdash e : T; \text{ is-refT } T; \text{ final } e;$
 $e = \text{null} \implies R;$
 $\bigwedge a \ C. \llbracket e = \text{addr } a; T = \text{Class } C \rrbracket \implies R;$
 $\bigwedge a. e = \text{Throw } a \implies R \rrbracket \implies R$

Derivation of new induction scheme for well typing:

inductive

$WTrt' :: [J\text{-prog}, \text{heap}, \text{sheap}, \text{env}, \text{expr}, \text{ty}] \Rightarrow \text{bool}$
and $WTrts' :: [J\text{-prog}, \text{heap}, \text{sheap}, \text{env}, \text{expr list}, \text{ty list}] \Rightarrow \text{bool}$
and $WTrt2' :: [J\text{-prog}, \text{env}, \text{heap}, \text{sheap}, \text{expr}, \text{ty}] \Rightarrow \text{bool}$
 $(\langle -, -, -, - \vdash - : '' \rightarrow [51, 51, 51, 51] 50 \rangle)$
and $WTrts2' :: [J\text{-prog}, \text{env}, \text{heap}, \text{sheap}, \text{expr list}, \text{ty list}] \Rightarrow \text{bool}$
 $(\langle -, -, -, - \vdash - : '' \rightarrow [51, 51, 51, 51] 50 \rangle)$
for $P :: J\text{-prog}$ **and** $h :: \text{heap}$ **and** $sh :: \text{sheap}$

where

$$\begin{aligned}
& P, E, h, sh \vdash e : ' T \equiv WTrt' P h sh E e T \\
& | P, E, h, sh \vdash es [:\] Ts \equiv WTrts' P h sh E es Ts \\
& | is-class P C \implies P, E, h, sh \vdash new C : ' Class C \\
& | [[P, E, h, sh \vdash e : ' T; is-refT T; is-class P C] \\
& \implies P, E, h, sh \vdash Cast C e : ' Class C \\
& | typeof_h v = Some T \implies P, E, h, sh \vdash Val v : ' T \\
& | E v = Some T \implies P, E, h, sh \vdash Var v : ' T \\
& | [[P, E, h, sh \vdash e_1 : ' T_1; P, E, h, sh \vdash e_2 : ' T_2] \\
& \implies P, E, h, sh \vdash e_1 \ll Eq \gg e_2 : ' Boolean \\
& | [[P, E, h, sh \vdash e_1 : ' Integer; P, E, h, sh \vdash e_2 : ' Integer] \\
& \implies P, E, h, sh \vdash e_1 \ll Add \gg e_2 : ' Integer \\
& | [[P, E, h, sh \vdash Var V : ' T; P, E, h, sh \vdash e : ' T'; P \vdash T' \leq T] \\
& \implies P, E, h, sh \vdash V := e : ' Void \\
& | [[P, E, h, sh \vdash e : ' Class C; P \vdash C has F, NonStatic: T in D] \implies P, E, h, sh \vdash e \cdot F\{D\} : ' T \\
& | P, E, h, sh \vdash e : ' NT \implies P, E, h, sh \vdash e \cdot F\{D\} : ' T \\
& | [[P \vdash C has F, Static: T in D] \implies P, E, h, sh \vdash C \cdot_s F\{D\} : ' T \\
& | [[P, E, h, sh \vdash e_1 : ' Class C; P \vdash C has F, NonStatic: T in D; \\
& P, E, h, sh \vdash e_2 : ' T_2; P \vdash T_2 \leq T] \\
& \implies P, E, h, sh \vdash e_1 \cdot F\{D\} := e_2 : ' Void \\
& | [[P, E, h, sh \vdash e_1 : ' NT; P, E, h, sh \vdash e_2 : ' T_2] \implies P, E, h, sh \vdash e_1 \cdot F\{D\} := e_2 : ' Void \\
& | [[P \vdash C has F, Static: T in D; \\
& P, E, h, sh \vdash e_2 : ' T_2; P \vdash T_2 \leq T] \\
& \implies P, E, h, sh \vdash C \cdot_s F\{D\} := e_2 : ' Void \\
& | [[P, E, h, sh \vdash e : ' Class C; P \vdash C sees M, NonStatic: Ts \to T = (pns, body) in D; \\
& P, E, h, sh \vdash es [:\] Ts'; P \vdash Ts' [\leq] Ts] \\
& \implies P, E, h, sh \vdash e \cdot M(es) : ' T \\
& | [[P, E, h, sh \vdash e : ' NT; P, E, h, sh \vdash es [:\] Ts] \implies P, E, h, sh \vdash e \cdot M(es) : ' T \\
& | [[P \vdash C sees M, Static: Ts \to T = (pns, body) in D; \\
& P, E, h, sh \vdash es [:\] Ts'; P \vdash Ts' [\leq] Ts; \\
& M = clinit \longrightarrow sh D = [(sfs, Processing)] \wedge es = map Val vs] \\
& \implies P, E, h, sh \vdash C \cdot_s M(es) : ' T \\
& | P, E, h, sh \vdash [] [:\] [] \\
& | [[P, E, h, sh \vdash e : ' T; P, E, h, sh \vdash es [:\] Ts] \implies P, E, h, sh \vdash e \# es [:\] T \# Ts \\
& | [[typeof_h v = Some T_1; P \vdash T_1 \leq T; P, E(V \mapsto T), h, sh \vdash e_2 : ' T_2] \\
& \implies P, E, h, sh \vdash \{V: T := Val v; e_2\} : ' T_2 \\
& | [[P, E(V \mapsto T), h, sh \vdash e : ' T'; \neg assigned V e] \implies P, E, h, sh \vdash \{V: T; e\} : ' T' \\
& | [[P, E, h, sh \vdash e_1 : ' T_1; P, E, h, sh \vdash e_2 : ' T_2] \implies P, E, h, sh \vdash e_1 ; e_2 : ' T_2 \\
& | [[P, E, h, sh \vdash e : ' Boolean; P, E, h, sh \vdash e_1 : ' T_1; P, E, h, sh \vdash e_2 : ' T_2; \\
& P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1; \\
& P \vdash T_1 \leq T_2 \longrightarrow T = T_2; P \vdash T_2 \leq T_1 \longrightarrow T = T_1] \\
& \implies P, E, h, sh \vdash if (e) e_1 else e_2 : ' T \\
& | [[P, E, h, sh \vdash e : ' Boolean; P, E, h, sh \vdash c : ' T] \\
& \implies P, E, h, sh \vdash while(e) c : ' Void \\
& | [[P, E, h, sh \vdash e : ' T_r; is-refT T_r] \implies P, E, h, sh \vdash throw e : ' T \\
& | [[P, E, h, sh \vdash e_1 : ' T_1; P, E(V \mapsto Class C), h, sh \vdash e_2 : ' T_2; P \vdash T_1 \leq T_2] \\
& \implies P, E, h, sh \vdash try e_1 catch(C V) e_2 : ' T_2 \\
& | [[P, E, h, sh \vdash e : ' T; \forall C' \in set (C \# Cs). is-class P C'; \neg sub-RI e; \\
& \forall C' \in set (tl Cs). \exists sfs. sh C' = [(sfs, Processing)]; \\
& b \longrightarrow (\forall C' \in set Cs. \exists sfs. sh C' = [(sfs, Processing)]); \\
& distinct Cs; supercls-1st P Cs] \implies P, E, h, sh \vdash INIT C (Cs, b) \leftarrow e : ' T \\
& | [[P, E, h, sh \vdash e : ' T; P, E, h, sh \vdash e' : ' T'; \forall C' \in set (C \# Cs). is-class P C'; \neg sub-RI e'; \\
& \forall C' \in set (C \# Cs). not-init C' e;
\end{aligned}$$

$$\begin{aligned} & \forall C' \in \text{set } Cs. \exists \text{sfs. sh } C' = [(sfs, Processing)]; \\ & \exists \text{sfs. sh } C = [(sfs, Processing)] \vee (\text{sh } C = [(sfs, Error)] \wedge e = \text{THROW NoClassDefFoundError}); \\ & \text{distinct } (C \# Cs); \text{supercls-1st } P (C \# Cs) \text{]} \\ \implies & P, E, h, sh \vdash RI(C, e); Cs \leftarrow e' : ' T' \end{aligned}$$

lemma [iff]: $P, E, h, sh \vdash e_1; e_2 : ' T_2 = (\exists T_1. P, E, h, sh \vdash e_1 : ' T_1 \wedge P, E, h, sh \vdash e_2 : ' T_2)$

lemma [iff]: $P, E, h, sh \vdash \text{Val } v : ' T = (\text{typeof}_h v = \text{Some } T)$

lemma [iff]: $P, E, h, sh \vdash \text{Var } v : ' T = (E v = \text{Some } T)$

lemma $wt\text{-}wt'$: $P, E, h, sh \vdash e : T \implies P, E, h, sh \vdash e : ' T$

and $wts\text{-}wts'$: $P, E, h, sh \vdash es [:] Ts \implies P, E, h, sh \vdash es [:\text{'}] Ts$

lemma $wt'\text{-}wt$: $P, E, h, sh \vdash e : ' T \implies P, E, h, sh \vdash e : T$

and $wts'\text{-}wts$: $P, E, h, sh \vdash es [:\text{'}] Ts \implies P, E, h, sh \vdash es [:] Ts$

corollary $wt'\text{-}iff\text{-}wt$: $(P, E, h, sh \vdash e : ' T) = (P, E, h, sh \vdash e : T)$

corollary $wts'\text{-}iff\text{-}wts$: $(P, E, h, sh \vdash es [:\text{'}] Ts) = (P, E, h, sh \vdash es [:] Ts)$

theorem assumes wf : $wwf\text{-}J\text{-prog } P$ **and** $hconf$: $P \vdash h \checkmark$ **and** $shconf$: $P, h \vdash_s sh \checkmark$

shows $progress$: $P, E, h, sh \vdash e : T \implies$

$(\bigwedge l. [\mathcal{D} e \mid \text{dom } l]; P, sh \vdash_b (e, b) \checkmark; \neg \text{final } e) \implies \exists e' s' b'. P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e', s', b' \rangle$

and $P, E, h, sh \vdash es [:] Ts \implies$

$(\bigwedge l. [\mathcal{D} s \text{ es } \mid \text{dom } l]; P, sh \vdash_b (es, b) \checkmark; \neg \text{finals } es) \implies \exists es' s' b'. P \vdash \langle es, (h, l, sh), b \rangle [\rightarrow] \langle es', s', b' \rangle$

end

1.21 Well-formedness Constraints

theory JWellForm

imports ../Common/WellForm WWellForm WellType DefAss

begin

definition $wf\text{-}J\text{-mdecl} :: J\text{-prog} \Rightarrow \text{cname} \Rightarrow J\text{-mb } mdecl \Rightarrow \text{bool}$

where

$wf\text{-}J\text{-mdecl } P C \equiv \lambda(M, b, Ts, T, (pns, body)).$

$\text{length } Ts = \text{length } pns \wedge$

$\text{distinct } pns \wedge$

$\neg \text{sub-RI } body \wedge$

(*case* b of

$\text{NonStatic} \Rightarrow \text{this} \notin \text{set } pns \wedge$

$(\exists T'. P, [\text{this} \mapsto \text{Class } C, pns[\mapsto] Ts] \vdash body :: T' \wedge P \vdash T' \leq T) \wedge$

$\mathcal{D} body [\{\text{this}\} \cup \text{set } pns]$

| $\text{Static} \Rightarrow (\exists T'. P, [pns[\mapsto] Ts] \vdash body :: T' \wedge P \vdash T' \leq T) \wedge$

$\mathcal{D} body [\text{set } pns])$

lemma $wf\text{-}J\text{-mdecl-NonStatic[simp]$:

$wf\text{-}J\text{-mdecl } P C (M, \text{NonStatic}, Ts, T, pns, body) \equiv$

$(\text{length } Ts = \text{length } pns \wedge$

$\text{distinct } pns \wedge$

$\neg \text{sub-RI } body \wedge$

$\text{this} \notin \text{set } pns \wedge$

$(\exists T'. P, [\text{this} \mapsto \text{Class } C, pns[\mapsto] Ts] \vdash body :: T' \wedge P \vdash T' \leq T) \wedge$

$\mathcal{D} body [\{\text{this}\} \cup \text{set } pns])$

lemma $wf\text{-}J\text{-mdecl-Static[simp]$:

$$\begin{aligned}
& wf\text{-}J\text{-}mdecl\ P\ C\ (M, Static, Ts, T, pns, body) \equiv \\
& (length\ Ts = length\ pns \wedge \\
& \text{distinct}\ pns \wedge \\
& \neg\text{sub}\text{-}RI\ body \wedge \\
& (\exists\ T'.\ P, [pns[\mapsto]Ts] \vdash body :: T' \wedge P \vdash T' \leq T) \wedge \\
& \mathcal{D}\ body\ [set\ pns])
\end{aligned}$$
abbreviation

$$\begin{aligned}
& wf\text{-}J\text{-}prog :: J\text{-}prog \Rightarrow bool\ \mathbf{where} \\
& wf\text{-}J\text{-}prog == wf\text{-}prog\ wf\text{-}J\text{-}mdecl
\end{aligned}$$
lemma *wf-J-prog-wf-J-mdecl*:
$$\begin{aligned}
& \llbracket wf\text{-}J\text{-}prog\ P; (C, D, fds, mths) \in set\ P; jmdcl \in set\ mths \rrbracket \\
& \implies wf\text{-}J\text{-}mdecl\ P\ C\ jmdcl
\end{aligned}$$
lemma *wf-mdecl-wwf-mdecl*: $wf\text{-}J\text{-}mdecl\ P\ C\ Md \implies wwf\text{-}J\text{-}mdecl\ P\ C\ Md$ **lemma** *wf-prog-wwf-prog*: $wf\text{-}J\text{-}prog\ P \implies wwf\text{-}J\text{-}prog\ P$

end

1.22 Type Safety Proof

theory *TypeSafe***imports** *Progress BigStep SmallStep JWellForm***begin****lemma** *red-shext-incr*: $P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle$ $\implies (\bigwedge E\ T.\ P, E, h, sh \vdash e : T \implies sh \leq_s sh')$ **and** *reds-shext-incr*: $P \vdash \langle es, (h, l, sh), b \rangle [\rightarrow] \langle es', (h', l', sh'), b' \rangle$ $\implies (\bigwedge E\ Ts.\ P, E, h, sh \vdash es\ [:]\ Ts \implies sh \leq_s sh')$ **lemma** *wf-types-clinit*:**assumes** $wf:wf\text{-}prog\ wf\text{-}md\ P$ **and** $ex: class\ P\ C = Some\ a$ **and** $proc: sh\ C = [(sfs, Processing)]$ **shows** $P, E, h, sh \vdash C \bullet_s clinit([]) : Void$ **proof** –**from** ex **obtain** $D\ fs\ ms$ **where** $a = (D, fs, ms)$ **by** (*cases a*)**then have** $sP: (C, D, fs, ms) \in set\ P$ **using** $ex\ map\text{-}of\ SomeD[of\ P\ C\ a]$ **by** (*simp add: class-def*)**then have** *wf-clinit ms* **using** *assms* **by** (*unfold wf-prog-def wf-cdecl-def, auto*)**then obtain** $pns\ body$ **where** $sm: (clinit, Static, [], Void, pns, body) \in set\ ms$ **by** (*unfold wf-clinit-def auto*)**then have** $P \vdash C\ sees\ clinit, Static: [] \rightarrow Void = (pns, body)$ **in** C **using** *mdecl-visible[OF wf sP sm]* **by** *simp***then show** *?thesis* **using** *WTrtSCall proc* **by** *simp***qed**

1.22.1 Basic preservation lemmas

First some easy preservation lemmas.

theorem *red-preserves-hconf*: $P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle \implies (\bigwedge T\ E.\ \llbracket P, E, h, sh \vdash e : T; P \vdash h \checkmark \rrbracket \implies P \vdash h' \checkmark)$ **and** *reds-preserves-hconf*: $P \vdash \langle es, (h, l, sh), b \rangle [\rightarrow] \langle es', (h', l', sh'), b' \rangle \implies (\bigwedge Ts\ E.\ \llbracket P, E, h, sh \vdash es\ [:]\ Ts; P \vdash h \checkmark \rrbracket \implies P \vdash h' \checkmark)$

theorem *red-preserves-lconf*:

$$P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle \implies (\bigwedge T E. \llbracket P, E, h, sh \vdash e : T; P, h \vdash l (: \leq) E \rrbracket \implies P, h' \vdash l' (: \leq) E)$$

and *reds-preserves-lconf*:

$$P \vdash \langle es, (h, l, sh), b \rangle [\rightarrow] \langle es', (h', l', sh'), b' \rangle \implies (\bigwedge Ts E. \llbracket P, E, h, sh \vdash es [:] Ts; P, h \vdash l (: \leq) E \rrbracket \implies P, h' \vdash l' (: \leq) E)$$

theorem *red-preserves-shconf*:

$$P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle \implies (\bigwedge T E. \llbracket P, E, h, sh \vdash e : T; P, h \vdash_s sh \checkmark \rrbracket \implies P, h' \vdash_s sh' \checkmark)$$

and *reds-preserves-shconf*:

$$P \vdash \langle es, (h, l, sh), b \rangle [\rightarrow] \langle es', (h', l', sh'), b' \rangle \implies (\bigwedge Ts E. \llbracket P, E, h, sh \vdash es [:] Ts; P, h \vdash_s sh \checkmark \rrbracket \implies P, h' \vdash_s sh' \checkmark)$$

theorem *assumes wf: wwf-J-prog P*

shows *red-preserves-icnf*:

$$P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle \implies icnf\ sh\ e \implies icnf\ sh'\ e'$$

and *reds-preserves-icnf*:

$$P \vdash \langle es, (h, l, sh), b \rangle [\rightarrow] \langle es', (h', l', sh'), b' \rangle \implies iconfs\ sh\ es \implies iconfs\ sh'\ es'$$

lemma *Seq-bconf-preserve-aux*:

assumes $P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle$ **and** $P, sh \vdash_b (e;; e_2, b) \checkmark$

and $P, sh \vdash_b (e::expr, b) \checkmark \longrightarrow P, sh' \vdash_b (e'::expr, b') \checkmark$

shows $P, sh' \vdash_b (e'; e_2, b') \checkmark$

proof(*cases val-of e*)

case *None* **show** *?thesis*

proof(*cases lass-val-of e*)

case *lNone: None* **show** *?thesis*

proof(*cases lass-val-of e'*)

case *lNone': None*

then show *?thesis* **using** *None assms lNone*

by(*auto dest: val-of-spec simp: bconf-def option.distinct(1)*)

next

case (*Some a*)

then show *?thesis* **using** *None assms lNone* **by**(*auto dest: lass-val-of-spec simp: bconf-def*)

qed

next

case (*Some a*)

then show *?thesis* **using** *None assms* **by**(*auto dest: lass-val-of-spec*)

qed

next

case (*Some a*)

then show *?thesis* **using** *assms* **by**(*auto dest: val-of-spec*)

qed

theorem *red-preserves-bconf*:

$$P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle \implies icnf\ sh\ e \implies P, sh \vdash_b (e, b) \checkmark \implies P, sh' \vdash_b (e', b') \checkmark$$

and *reds-preserves-bconf*:

$$P \vdash \langle es, (h, l, sh), b \rangle [\rightarrow] \langle es', (h', l', sh'), b' \rangle \implies iconfs\ sh\ es \implies P, sh \vdash_b (es, b) \checkmark \implies P, sh' \vdash_b (es', b') \checkmark$$

\checkmark

Preservation of definite assignment more complex and requires a few lemmas first.

lemma [*iff*]: $\bigwedge A. \llbracket \text{length } Vs = \text{length } Ts; \text{length } vs = \text{length } Ts \rrbracket \implies$

$\mathcal{D}(\text{blocks } (Vs, Ts, vs, e))\ A = \mathcal{D}\ e\ (A \sqcup [\text{set } Vs])$

lemma red-lA-incr: $P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle$

$$\implies \lfloor \text{dom } l \rfloor \sqcup \mathcal{A} e \sqsubseteq \lfloor \text{dom } l' \rfloor \sqcup \mathcal{A} e'$$

and reds-lA-incr: $P \vdash \langle es, (h, l, sh), b \rangle [\rightarrow] \langle es', (h', l', sh'), b' \rangle$

$$\implies \lfloor \text{dom } l \rfloor \sqcup \mathcal{A}s es \sqsubseteq \lfloor \text{dom } l' \rfloor \sqcup \mathcal{A}s es'$$

Now preservation of definite assignment.

lemma assumes wf: $wf\text{-}J\text{-prog } P$

shows red-preserves-defass:

$$P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle \implies \mathcal{D} e \lfloor \text{dom } l \rfloor \implies \mathcal{D} e' \lfloor \text{dom } l' \rfloor$$

and $P \vdash \langle es, (h, l, sh), b \rangle [\rightarrow] \langle es', (h', l', sh'), b' \rangle \implies \mathcal{D}s es \lfloor \text{dom } l \rfloor \implies \mathcal{D}s es' \lfloor \text{dom } l' \rfloor$

Combining conformance of heap, static heap, and local variables:

definition sconfg :: $J\text{-prog} \Rightarrow env \Rightarrow state \Rightarrow bool$ ($\langle -, - \vdash - \sqrt{} \rangle$ [51,51,51]50)

where

$$P, E \vdash s \sqrt{} \equiv \text{let } (h, l, sh) = s \text{ in } P \vdash h \sqrt{} \wedge P, h \vdash l (\leq) E \wedge P, h \vdash_s sh \sqrt{}$$

lemma red-preserves-sconfg:

$$\llbracket P \vdash \langle e, s, b \rangle \rightarrow \langle e', s', b' \rangle; P, E, hp s, shp s \vdash e : T; P, E \vdash s \sqrt{} \rrbracket \implies P, E \vdash s' \sqrt{}$$

lemma reds-preserves-sconfg:

$$\llbracket P \vdash \langle es, s, b \rangle [\rightarrow] \langle es', s', b' \rangle; P, E, hp s, shp s \vdash es [\cdot] Ts; P, E \vdash s \sqrt{} \rrbracket \implies P, E \vdash s' \sqrt{}$$

1.22.2 Subject reduction

lemma wt-blocks:

$$\bigwedge E. \llbracket \text{length } Vs = \text{length } Ts; \text{length } vs = \text{length } Ts \rrbracket \implies$$

$$(P, E, h, sh \vdash \text{blocks}(Vs, Ts, vs, e) : T) =$$

$$(P, E(Vs[\rightarrow]Ts), h, sh \vdash e : T \wedge (\exists Ts'. \text{map } (\text{typeof}_h) vs = \text{map } \text{Some } Ts' \wedge P \vdash Ts' [\leq] Ts))$$

theorem assumes wf: $wf\text{-}J\text{-prog } P$

shows subject-reduction2: $P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle \implies$

$$(\bigwedge E T. \llbracket P, E \vdash (h, l, sh) \sqrt{}; \text{iconfg } sh e; P, E, h, sh \vdash e : T \rrbracket$$

$$\implies \exists T'. P, E, h', sh' \vdash e' : T' \wedge P \vdash T' \leq T)$$

and subjects-reduction2: $P \vdash \langle es, (h, l, sh), b \rangle [\rightarrow] \langle es', (h', l', sh'), b' \rangle \implies$

$$(\bigwedge E Ts. \llbracket P, E \vdash (h, l, sh) \sqrt{}; \text{iconfgs } sh es; P, E, h, sh \vdash es [\cdot] Ts \rrbracket$$

$$\implies \exists Ts'. P, E, h', sh' \vdash es' [\cdot] Ts' \wedge P \vdash Ts' [\leq] Ts)$$

corollary subject-reduction:

$$\llbracket wf\text{-}J\text{-prog } P; P \vdash \langle e, s, b \rangle \rightarrow \langle e', s', b' \rangle; P, E \vdash s \sqrt{}; \text{iconfg } (shp s) e; P, E, hp s, shp s \vdash e : T \rrbracket$$

$$\implies \exists T'. P, E, hp s', shp s' \vdash e' : T' \wedge P \vdash T' \leq T$$

corollary subjects-reduction:

$$\llbracket wf\text{-}J\text{-prog } P; P \vdash \langle es, s, b \rangle [\rightarrow] \langle es', s', b' \rangle; P, E \vdash s \sqrt{}; \text{iconfgs } (shp s) es; P, E, hp s, shp s \vdash es [\cdot] Ts \rrbracket$$

$$\implies \exists Ts'. P, E, hp s', shp s' \vdash es' [\cdot] Ts' \wedge P \vdash Ts' [\leq] Ts$$

1.22.3 Lifting to \rightarrow^*

Now all these preservation lemmas are first lifted to the transitive closure ...

lemma Red-preserves-sconfg:

assumes wf: $wf\text{-}J\text{-prog } P$ **and Red:** $P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle$

shows $\bigwedge T. \llbracket P, E, hp s, shp s \vdash e : T; \text{iconfg } (shp s) e; P, E \vdash s \sqrt{} \rrbracket \implies P, E \vdash s' \sqrt{}$

lemma Red-preserves-iconfg:

assumes wf: $wf\text{-}J\text{-prog } P$ **and Red:** $P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle$

shows $\text{iconfg } (shp s) e \implies \text{iconfg } (shp s') e'$

lemma Reds-preserves-iconfg:

assumes wf: $wf\text{-}J\text{-prog } P$ **and Red:** $P \vdash \langle es, s, b \rangle [\rightarrow]^* \langle es', s', b' \rangle$

shows $\text{iconfgs } (shp s) es \implies \text{iconfgs } (shp s') es'$

lemma *Red-preserves-bconf*:

assumes *wf*: *wf-J-prog P* **and** *Red*: $P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle$

shows $\text{iconf } (\text{shp } s) e \Longrightarrow P, (\text{shp } s) \vdash_b (e, b) \checkmark \Longrightarrow P, (\text{shp } s') \vdash_b (e'::\text{expr}, b') \checkmark$

lemma *Reds-preserves-bconf*:

assumes *wf*: *wf-J-prog P* **and** *Red*: $P \vdash \langle es, s, b \rangle [\rightarrow]^* \langle es', s', b' \rangle$

shows $\text{iconfs } (\text{shp } s) es \Longrightarrow P, (\text{shp } s) \vdash_b (es, b) \checkmark \Longrightarrow P, (\text{shp } s') \vdash_b (es'::\text{expr list}, b') \checkmark$

lemma *Red-preserves-defass*:

assumes *wf*: *wf-J-prog P* **and** *reds*: $P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle$

shows $\mathcal{D} e \ [\text{dom}(\text{lcl } s)] \Longrightarrow \mathcal{D} e' \ [\text{dom}(\text{lcl } s')]$

using *reds*

proof (*induct rule:converse-rtrancl-induct3*)

case *refl* **thus** ?*case* .

next

case (*step e s b e' s' b'*) **thus** ?*case*

by(*cases s, cases s'*)(*auto dest:red-preserves-defass[OF wf]*)

qed

lemma *Red-preserves-type*:

assumes *wf*: *wf-J-prog P* **and** *Red*: $P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle$

shows !!*T*. $\llbracket P, E \vdash s \checkmark; \text{iconf } (\text{shp } s) e; P, E, \text{hp } s, \text{shp } s \vdash e : T \rrbracket$

$\Longrightarrow \exists T'. P \vdash T' \leq T \wedge P, E, \text{hp } s', \text{shp } s' \vdash e' : T'$

1.22.4 The final polish

The above preservation lemmas are now combined and packed nicely.

definition *wf-config* :: $J\text{-prog} \Rightarrow \text{env} \Rightarrow \text{state} \Rightarrow \text{expr} \Rightarrow \text{ty} \Rightarrow \text{bool}$ ($\langle -, -, - \vdash - : - \checkmark \rangle$ [51,0,0,0,0]50)

where

$P, E, s \vdash e : T \checkmark \equiv P, E \vdash s \checkmark \wedge \text{iconf } (\text{shp } s) e \wedge P, E, \text{hp } s, \text{shp } s \vdash e : T$

theorem *Subject-reduction*: **assumes** *wf*: *wf-J-prog P*

shows $P \vdash \langle e, s, b \rangle \rightarrow \langle e', s', b' \rangle \Longrightarrow P, E, s \vdash e : T \checkmark$

$\Longrightarrow \exists T'. P, E, s' \vdash e' : T' \checkmark \wedge P \vdash T' \leq T$

theorem *Subject-reductions*:

assumes *wf*: *wf-J-prog P* **and** *reds*: $P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle$

shows $\bigwedge T. P, E, s \vdash e : T \checkmark \Longrightarrow \exists T'. P, E, s' \vdash e' : T' \checkmark \wedge P \vdash T' \leq T$

corollary *Progress*: **assumes** *wf*: *wf-J-prog P*

shows $\llbracket P, E, s \vdash e : T \checkmark; \mathcal{D} e \ [\text{dom}(\text{lcl } s)]; P, \text{shp } s \vdash_b (e, b) \checkmark; \neg \text{final } e \rrbracket$

$\Longrightarrow \exists e' s' b'. P \vdash \langle e, s, b \rangle \rightarrow \langle e', s', b' \rangle$

corollary *TypeSafety*:

fixes *s*::*state* **and** *e*::*expr*

assumes *wf*: *wf-J-prog P* **and** *sconf*: $P, E \vdash s \checkmark$ **and** *wt*: $P, E \vdash e : T$

and *D*: $\mathcal{D} e \ [\text{dom}(\text{lcl } s)]$

and *iconf*: $\text{iconf } (\text{shp } s) e$ **and** *bconf*: $P, (\text{shp } s) \vdash_b (e, b) \checkmark$

and *steps*: $P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle$

and *nstep*: $\neg(\exists e'' s'' b''. P \vdash \langle e', s', b' \rangle \rightarrow \langle e'', s'', b'' \rangle)$

shows $(\exists v. e' = \text{Val } v \wedge P, \text{hp } s' \vdash v : \leq T) \vee$

$(\exists a. e' = \text{Throw } a \wedge a \in \text{dom}(\text{hp } s'))$

end

1.23 Equivalence of Big Step and Small Step Semantics

theory *Equivalence* imports *TypeSafe* *WWellForm* begin

1.23.1 Small steps simulate big step

Init

The reduction of initialization expressions doesn't touch or use their on-hold expressions (the subexpression to the right of \leftarrow) until the initialization operation completes. This function is used to prove this and related properties. It is then used for general reduction of initialization expressions separately from their on-hold expressions by using the on-hold expression *unit*, then putting the real on-hold expression back in at the end.

```
fun init-switch :: 'a exp  $\Rightarrow$  'a exp  $\Rightarrow$  'a exp where
init-switch (INIT C (Cs,b)  $\leftarrow$  ei) e = (INIT C (Cs,b)  $\leftarrow$  e) |
init-switch (RI(C,e');Cs  $\leftarrow$  ei) e = (RI(C,e');Cs  $\leftarrow$  e) |
init-switch e' e = e'
```

```
fun INIT-class :: 'a exp  $\Rightarrow$  cname option where
INIT-class (INIT C (Cs,b)  $\leftarrow$  e) = (if C = last (C#Cs) then Some C else None) |
INIT-class (RI(C,e0);Cs  $\leftarrow$  e) = Some (last (C#Cs)) |
INIT-class - = None
```

lemma *init-red-init*:

```
 $\llbracket$  init-exp-of e0 =  $\lfloor e \rfloor$ ; P  $\vdash$   $\langle e_0, s_0, b_0 \rangle \rightarrow \langle e_1, s_1, b_1 \rangle$   $\rrbracket$ 
 $\implies$  (init-exp-of e1 =  $\lfloor e \rfloor \wedge$  (INIT-class e0 =  $\lfloor C \rfloor \longrightarrow$  INIT-class e1 =  $\lfloor C \rfloor$ ))
 $\vee$  (e1 = e  $\wedge$  b1 = icheck P (the(INIT-class e0)) e)  $\vee$  ( $\exists a. e_1 =$  throw a)
by(erule red.cases, auto)
```

lemma *init-exp-switch*[*simp*]:

```
init-exp-of e0 =  $\lfloor e \rfloor \implies$  init-exp-of (init-switch e0 e') =  $\lfloor e' \rfloor$ 
by(cases e0, simp-all)
```

lemma *init-red-sync*:

```
 $\llbracket$  P  $\vdash$   $\langle e_0, s_0, b_0 \rangle \rightarrow \langle e_1, s_1, b_1 \rangle$ ; init-exp-of e0 =  $\lfloor e \rfloor$ ; e1  $\neq$  e  $\rrbracket$ 
 $\implies$  ( $\bigwedge e'. P \vdash \langle$  init-switch e0 e', s0, b0  $\rangle \rightarrow \langle$  init-switch e1 e', s1, b1  $\rangle$ )
```

proof(*induct rule: red.cases*) **qed**(*simp-all add: red-reds.intros*)

lemma *init-red-sync-end*:

```
 $\llbracket$  P  $\vdash$   $\langle e_0, s_0, b_0 \rangle \rightarrow \langle e_1, s_1, b_1 \rangle$ ; init-exp-of e0 =  $\lfloor e \rfloor$ ; e1 = e; throw-of e = None  $\rrbracket$ 
 $\implies$  ( $\bigwedge e'. \neg$  sub-RI e'  $\implies$  P  $\vdash$   $\langle$  init-switch e0 e', s0, b0  $\rangle \rightarrow \langle$  e', s1, icheck P (the(INIT-class e0)) e'  $\rangle$ )
```

proof(*induct rule: red.cases*) **qed**(*simp-all add: red-reds.intros*)

lemma *init-reds-sync-unit'*:

```
 $\llbracket$  P  $\vdash$   $\langle e_0, s_0, b_0 \rangle \rightarrow^* \langle$  Val v', s1, b1  $\rangle$ ; init-exp-of e0 =  $\lfloor$  unit  $\rfloor$ ; INIT-class e0 =  $\lfloor$  C  $\rfloor$   $\rrbracket$ 
 $\implies$  ( $\bigwedge e'. \neg$  sub-RI e'  $\implies$  P  $\vdash$   $\langle$  init-switch e0 e', s0, b0  $\rangle \rightarrow^* \langle$  e', s1, icheck P (the(INIT-class e0)) e'  $\rangle$ )
```

proof(*induct rule: converse-rtrancl-induct3*)

case *refl* **then show** ?*case* **by** *simp*

next

case (*step* e₀ s₀ b₀ e₁ s₁ b₁)

have (*init-exp-of* e₁ = \lfloor *unit* $\rfloor \wedge$ (*INIT-class* e₀ = \lfloor C $\rfloor \longrightarrow$ *INIT-class* e₁ = \lfloor C \rfloor))

\vee (e₁ = *unit* \wedge b₁ = *icheck* P (the(*INIT-class* e₀)) *unit*) \vee ($\exists a. e_1 =$ *throw* a)

using *init-red-init*[*OF step.premis(1) step.hyps(1)*] **by** *simp*

then show ?*case*

proof(rule *disjE*)
assume *assm*: $\text{init-exp-of } e1 = \lfloor \text{unit} \rfloor \wedge (\text{INIT-class } e0 = \lfloor C \rfloor \longrightarrow \text{INIT-class } e1 = \lfloor C \rfloor)$
then have *red*: $P \vdash \langle \text{init-switch } e0 \ e' \ s0 \ b0 \rangle \rightarrow \langle \text{init-switch } e1 \ e' \ s1 \ b1 \rangle$
using *init-red-sync*[*OF step.hyps(1) step.prem(1)*] **by** *simp*
have *reds*: $P \vdash \langle \text{init-switch } e1 \ e' \ s1 \ b1 \rangle \rightarrow^* \langle e' \ s1 \ \text{icheck } P \ (\text{the } (\text{INIT-class } e0)) \ e' \rangle$
using *step.hyps(3)*[*OF - - step.prem(3)*] *assm step.prem(2)* **by** *simp*
show *?thesis* **by**(rule *converse-rtrancl-into-rtrancl*[*OF red reds*])
next
assume $(e1 = \text{unit} \wedge b1 = \text{icheck } P \ (\text{the}(\text{INIT-class } e0)) \ \text{unit}) \vee (\exists a. e1 = \text{throw } a)$
then show *?thesis*
proof(rule *disjE*)
assume *assm*: $e1 = \text{unit} \wedge b1 = \text{icheck } P \ (\text{the}(\text{INIT-class } e0)) \ \text{unit}$ **then have** *e1*: $e1 = \text{unit}$
by *simp*
have *sb*: $s1 = s1 \ b1 = b1$ **using** *reds-final-same*[*OF step.hyps(2)*] *assm*
by(*simp-all add: final-def*)
then have *step'*: $P \vdash \langle \text{init-switch } e0 \ e' \ s0 \ b0 \rangle \rightarrow \langle e' \ s1 \ \text{icheck } P \ (\text{the } (\text{INIT-class } e0)) \ e' \rangle$
using *init-red-sync-end*[*OF step.hyps(1) step.prem(1) e1 - step.prem(3)*] **by** *auto*
then have $P \vdash \langle \text{init-switch } e0 \ e' \ s0 \ b0 \rangle \rightarrow^* \langle e' \ s1 \ \text{icheck } P \ (\text{the } (\text{INIT-class } e0)) \ e' \rangle$
using *r-into-rtrancl* **by** *auto*
then show *?thesis* **using** *step assm sb* **by** *simp*
next
assume $\exists a. e1 = \text{throw } a$ **then obtain** *a* **where** $e1 = \text{throw } a$ **by** *clarsimp*
then have *tof*: $\text{throw-of } e1 = \lfloor a \rfloor$ **by** *simp*
then show *?thesis* **using** *reds-throw*[*OF step.hyps(2) tof*] **by** *simp*
qed
qed
qed

lemma *init-reds-sync-unit-throw'*:
 $\llbracket P \vdash \langle e0 \ s0 \ b0 \rangle \rightarrow^* \langle \text{throw } a \ s1 \ b1 \rangle; \text{init-exp-of } e0 = \lfloor \text{unit} \rfloor \rrbracket$
 $\implies (\bigwedge e'. P \vdash \langle \text{init-switch } e0 \ e' \ s0 \ b0 \rangle \rightarrow^* \langle \text{throw } a \ s1 \ b1 \rangle)$

proof(*induct rule:converse-rtrancl-induct3*)
case *refl* **then show** *?case* **by** *simp*
next
case (*step e0 s0 b0 e1 s1 b1*)
have $\text{init-exp-of } e1 = \lfloor \text{unit} \rfloor \wedge (\forall C. \text{INIT-class } e0 = \lfloor C \rfloor \longrightarrow \text{INIT-class } e1 = \lfloor C \rfloor) \vee$
 $e1 = \text{unit} \wedge b1 = \text{icheck } P \ (\text{the } (\text{INIT-class } e0)) \ \text{unit} \vee (\exists a. e1 = \text{throw } a)$
using *init-red-init*[*OF step.prem(1) step.hyps(1)*] **by** *auto*
then show *?case*
proof(rule *disjE*)
assume *assm*: $\text{init-exp-of } e1 = \lfloor \text{unit} \rfloor \wedge (\forall C. \text{INIT-class } e0 = \lfloor C \rfloor \longrightarrow \text{INIT-class } e1 = \lfloor C \rfloor)$
then have $P \vdash \langle \text{init-switch } e0 \ e' \ s0 \ b0 \rangle \rightarrow \langle \text{init-switch } e1 \ e' \ s1 \ b1 \rangle$
using *step init-red-sync*[*OF step.hyps(1) step.prem*] **by** *simp*
then show *?thesis* **using** *step assm* **by** (*meson converse-rtrancl-into-rtrancl*)
next
assume $e1 = \text{unit} \wedge b1 = \text{icheck } P \ (\text{the } (\text{INIT-class } e0)) \ \text{unit} \vee (\exists a. e1 = \text{throw } a)$
then show *?thesis*
proof(rule *disjE*)
assume $e1 = \text{unit} \wedge b1 = \text{icheck } P \ (\text{the } (\text{INIT-class } e0)) \ \text{unit}$
then show *?thesis* **using** *step using final-def reds-final-same* **by** *blast*
next
assume *assm*: $\exists a. e1 = \text{throw } a$
then have $P \vdash \langle \text{init-switch } e0 \ e' \ s0 \ b0 \rangle \rightarrow \langle e1 \ s1 \ b1 \rangle$
using *init-red-sync*[*OF step.hyps(1) step.prem*] **by** *clarsimp*

then show *?thesis using step by simp*
 qed
 qed
 qed

lemma *init-reds-sync-unit*:

assumes $P \vdash \langle e_0, s_0, b_0 \rangle \rightarrow^* \langle \text{Val } v', s_1, b_1 \rangle$ **and** *init-exp-of* $e_0 = [\text{unit}]$ **and** *INIT-class* $e_0 = [C]$
and $\neg \text{sub-RI } e'$
shows $P \vdash \langle \text{init-switch } e_0 \ e', s_0, b_0 \rangle \rightarrow^* \langle e', s_1, \text{icheck } P \ (\text{the}(\text{INIT-class } e_0)) \ e' \rangle$
using *init-reds-sync-unit*'[*OF assms*] **by** *clarsimp*

lemma *init-reds-sync-unit-throw*:

assumes $P \vdash \langle e_0, s_0, b_0 \rangle \rightarrow^* \langle \text{throw } a, s_1, b_1 \rangle$ **and** *init-exp-of* $e_0 = [\text{unit}]$
shows $P \vdash \langle \text{init-switch } e_0 \ e', s_0, b_0 \rangle \rightarrow^* \langle \text{throw } a, s_1, b_1 \rangle$
using *init-reds-sync-unit-throw*'[*OF assms*] **by** *clarsimp*

— init reds lemmas

lemma *InitSeqReds*:

assumes $P \vdash \langle \text{INIT } C \ ([C], b) \leftarrow \text{unit}, s_0, b_0 \rangle \rightarrow^* \langle \text{Val } v', s_1, b_1 \rangle$
and $P \vdash \langle e, s_1, \text{icheck } P \ C \ e \rangle \rightarrow^* \langle e_2, s_2, b_2 \rangle$ **and** $\neg \text{sub-RI } e$
shows $P \vdash \langle \text{INIT } C \ ([C], b) \leftarrow e, s_0, b_0 \rangle \rightarrow^* \langle e_2, s_2, b_2 \rangle$
using *assms*
proof –
have $P \vdash \langle \text{init-switch } (\text{INIT } C \ ([C], b) \leftarrow \text{unit}) \ e, s_0, b_0 \rangle \rightarrow^* \langle e, s_1, \text{icheck } P \ C \ e \rangle$
using *init-reds-sync-unit*'[*OF assms*(1) - - *assms*(3)] **by** *simp*
then show *?thesis using assms*(2) **by** *simp*
 qed

lemma *InitSeqThrowReds*:

assumes $P \vdash \langle \text{INIT } C \ ([C], b) \leftarrow \text{unit}, s_0, b_0 \rangle \rightarrow^* \langle \text{throw } a, s_1, b_1 \rangle$
shows $P \vdash \langle \text{INIT } C \ ([C], b) \leftarrow e, s_0, b_0 \rangle \rightarrow^* \langle \text{throw } a, s_1, b_1 \rangle$
using *assms*
proof –
have $P \vdash \langle \text{init-switch } (\text{INIT } C \ ([C], b) \leftarrow \text{unit}) \ e, s_0, b_0 \rangle \rightarrow^* \langle \text{throw } a, s_1, b_1 \rangle$
using *init-reds-sync-unit-throw*'[*OF assms*(1)] **by** *simp*
then show *?thesis by simp*
 qed

lemma *InitNoneReds*:

$\llbracket \text{sh } C = \text{None};$
 $P \vdash \langle \text{INIT } C' \ (C \# Cs, \text{False}) \leftarrow e, (h, l, \text{sh}(C \mapsto (\text{sblank } P \ C, \text{Prepared}))), b \rangle \rightarrow^* \langle e', s', b' \rangle \rrbracket$
 $\implies P \vdash \langle \text{INIT } C' \ (C \# Cs, \text{False}) \leftarrow e, (h, l, \text{sh}), b \rangle \rightarrow^* \langle e', s', b' \rangle$

lemma *InitDoneReds*:

$\llbracket \text{sh } C = \text{Some}(sfs, \text{Done}); P \vdash \langle \text{INIT } C' \ (Cs, \text{True}) \leftarrow e, (h, l, \text{sh}), b \rangle \rightarrow^* \langle e', s', b' \rangle \rrbracket$
 $\implies P \vdash \langle \text{INIT } C' \ (C \# Cs, \text{False}) \leftarrow e, (h, l, \text{sh}), b \rangle \rightarrow^* \langle e', s', b' \rangle$

lemma *InitProcessingReds*:

$\llbracket \text{sh } C = \text{Some}(sfs, \text{Processing}); P \vdash \langle \text{INIT } C' \ (Cs, \text{True}) \leftarrow e, (h, l, \text{sh}), b \rangle \rightarrow^* \langle e', s', b' \rangle \rrbracket$
 $\implies P \vdash \langle \text{INIT } C' \ (C \# Cs, \text{False}) \leftarrow e, (h, l, \text{sh}), b \rangle \rightarrow^* \langle e', s', b' \rangle$

lemma *InitErrorReds*:

$\llbracket \text{sh } C = \text{Some}(sfs, \text{Error}); P \vdash \langle \text{RI } (C, \text{THROW } \text{NoClassDefFoundError}); Cs \leftarrow e, (h, l, \text{sh}), b \rangle \rightarrow^* \langle e', s', b' \rangle \rrbracket$

$\implies P \vdash \langle \text{INIT } C' \ (C \# Cs, \text{False}) \leftarrow e, (h, l, \text{sh}), b \rangle \rightarrow^* \langle e', s', b' \rangle$

lemma *InitObjectReds*:

$\llbracket sh\ C = Some(sfs, Prepared); C = Object; sh' = sh(C \mapsto (sfs, Processing));$
 $P \vdash \langle INIT\ C' (C \# Cs, True) \leftarrow e, (h, l, sh'), b \rangle \rightarrow^* \langle e', s', b' \rangle \rrbracket$
 $\implies P \vdash \langle INIT\ C' (C \# Cs, False) \leftarrow e, (h, l, sh), b \rangle \rightarrow^* \langle e', s', b' \rangle$

lemma *InitNonObjectReds*:

$\llbracket sh\ C = Some(sfs, Prepared); C \neq Object; class\ P\ C = Some\ (D, r);$
 $sh' = sh(C \mapsto (sfs, Processing));$
 $P \vdash \langle INIT\ C' (D \# C \# Cs, False) \leftarrow e, (h, l, sh'), b \rangle \rightarrow^* \langle e', s', b' \rangle \rrbracket$
 $\implies P \vdash \langle INIT\ C' (C \# Cs, False) \leftarrow e, (h, l, sh), b \rangle \rightarrow^* \langle e', s', b' \rangle$

lemma *RedsInitRInit*:

$P \vdash \langle RI\ (C, C \cdot s, clinit(\ [])); Cs \leftarrow e, (h, l, sh), b \rangle \rightarrow^* \langle e', s', b' \rangle$
 $\implies P \vdash \langle INIT\ C' (C \# Cs, True) \leftarrow e, (h, l, sh), b \rangle \rightarrow^* \langle e', s', b' \rangle$

lemmas *rtrancl-induct3* =

rtrancl-induct[of (ax, ay, az) (bx, by, bz) , *split-format* (*complete*), *consumes 1*, *case-names refl step*]

lemma *RInitReds*:

$P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle$
 $\implies P \vdash \langle RI\ (C, e); Cs \leftarrow e_0, s, b \rangle \rightarrow^* \langle RI\ (C, e'); Cs \leftarrow e_0, s', b' \rangle$

lemma *RedsRInit*:

$\llbracket P \vdash \langle e_0, s_0, b_0 \rangle \rightarrow^* \langle Val\ v, (h_1, l_1, sh_1), b_1 \rangle;$
 $sh_1\ C = Some\ (sfs, i); sh_2 = sh_1(C \mapsto (sfs, Done)); C' = last(C \# Cs);$
 $P \vdash \langle INIT\ C' (Cs, True) \leftarrow e, (h_1, l_1, sh_2), b_1 \rangle \rightarrow^* \langle e', s', b' \rangle \rrbracket$
 $\implies P \vdash \langle RI\ (C, e_0); Cs \leftarrow e, s_0, b_0 \rangle \rightarrow^* \langle e', s', b' \rangle$

lemma *RInitInitThrowReds*:

$\llbracket P \vdash \langle e, s, b \rangle \rightarrow^* \langle Throw\ a, (h', l', sh'), b' \rangle;$
 $sh'\ C = Some\ (sfs, i); sh'' = sh'(C \mapsto (sfs, Error));$
 $P \vdash \langle RI\ (D, Throw\ a); Cs \leftarrow e_0, (h', l', sh'), b' \rangle \rightarrow^* \langle e_1, s_1, b_1 \rangle \rrbracket$
 $\implies P \vdash \langle RI\ (C, e); D \# Cs \leftarrow e_0, s, b \rangle \rightarrow^* \langle e_1, s_1, b_1 \rangle$

lemma *RInitThrowReds*:

$\llbracket P \vdash \langle e, s, b \rangle \rightarrow^* \langle Throw\ a, (h', l', sh'), b' \rangle;$
 $sh'\ C = Some(sfs, i); sh'' = sh'(C \mapsto (sfs, Error)) \rrbracket$
 $\implies P \vdash \langle RI\ (C, e); Nil \leftarrow e_0, s, b \rangle \rightarrow^* \langle Throw\ a, (h', l', sh''), b' \rangle$

New

lemma *NewInitDoneReds*:

$\llbracket sh\ C = Some\ (sfs, Done); new-Addr\ h = Some\ a;$
 $P \vdash C\ has-fields\ FDTs; h' = h(a \mapsto blank\ P\ C) \rrbracket$
 $\implies P \vdash \langle new\ C, (h, l, sh), False \rangle \rightarrow^* \langle addr\ a, (h', l, sh), False \rangle$

lemma *NewInitDoneReds2*:

$\llbracket sh\ C = Some\ (sfs, Done); new-Addr\ h = None; is-class\ P\ C \rrbracket$
 $\implies P \vdash \langle new\ C, (h, l, sh), False \rangle \rightarrow^* \langle THROW\ OutOfMemory, (h, l, sh), False \rangle$

lemma *NewInitReds*:

assumes $nDone$: $\# sfs. shp\ s\ C = Some\ (sfs, Done)$
and *INIT-steps*: $P \vdash \langle INIT\ C\ ([C], False) \leftarrow unit, s, False \rangle \rightarrow^* \langle Val\ v', (h', l', sh'), b' \rangle$
and *Addr*: $new-Addr\ h' = Some\ a$ **and** *has*: $P \vdash C\ has-fields\ FDTs$
and h'' : $h'' = h'(a \mapsto blank\ P\ C)$ **and** *cls-C*: $is-class\ P\ C$
shows $P \vdash \langle new\ C, s, False \rangle \rightarrow^* \langle addr\ a, (h'', l', sh'), False \rangle$

lemma *NewInitOOMReds*:

assumes $nDone$: $\# sfs. shp\ s\ C = Some\ (sfs, Done)$
and *INIT-steps*: $P \vdash \langle INIT\ C\ ([C], False) \leftarrow unit, s, False \rangle \rightarrow^* \langle Val\ v', (h', l', sh'), b' \rangle$
and *Addr*: $new-Addr\ h' = None$ **and** *cls-C*: $is-class\ P\ C$
shows $P \vdash \langle new\ C, s, False \rangle \rightarrow^* \langle THROW\ OutOfMemory, (h', l', sh'), False \rangle$

lemma *NewInitThrowReds*:

assumes $nDone: \nexists sfs. shp\ s\ C = Some\ (sfs,\ Done)$
and $cls\ C: is\ class\ P\ C$
and $INIT\ steps: P \vdash \langle INIT\ C\ ([C],\ False) \leftarrow unit,\ s,\ False \rangle \rightarrow^* \langle throw\ a,\ s',\ b' \rangle$
shows $P \vdash \langle new\ C,\ s,\ False \rangle \rightarrow^* \langle throw\ a,\ s',\ b' \rangle$

Cast

lemma *CastReds*:

$$P \vdash \langle e,\ s,\ b \rangle \rightarrow^* \langle e',\ s',\ b' \rangle \implies P \vdash \langle Cast\ C\ e,\ s,\ b \rangle \rightarrow^* \langle Cast\ C\ e',\ s',\ b' \rangle$$

lemma *CastRedsNull*:

$$P \vdash \langle e,\ s,\ b \rangle \rightarrow^* \langle null,\ s',\ b' \rangle \implies P \vdash \langle Cast\ C\ e,\ s,\ b \rangle \rightarrow^* \langle null,\ s',\ b' \rangle$$

lemma *CastRedsAddr*:

$$\llbracket P \vdash \langle e,\ s,\ b \rangle \rightarrow^* \langle addr\ a,\ s',\ b' \rangle; hp\ s'\ a = Some(D,\ fs); P \vdash D \preceq^* C \rrbracket \implies P \vdash \langle Cast\ C\ e,\ s,\ b \rangle \rightarrow^* \langle addr\ a,\ s',\ b' \rangle$$

lemma *CastRedsFail*:

$$\llbracket P \vdash \langle e,\ s,\ b \rangle \rightarrow^* \langle addr\ a,\ s',\ b' \rangle; hp\ s'\ a = Some(D,\ fs); \neg P \vdash D \preceq^* C \rrbracket \implies P \vdash \langle Cast\ C\ e,\ s,\ b \rangle \rightarrow^* \langle THROW\ ClassCast,\ s',\ b' \rangle$$

lemma *CastRedsThrow*:

$$\llbracket P \vdash \langle e,\ s,\ b \rangle \rightarrow^* \langle throw\ a,\ s',\ b' \rangle \rrbracket \implies P \vdash \langle Cast\ C\ e,\ s,\ b \rangle \rightarrow^* \langle throw\ a,\ s',\ b' \rangle$$

LAss

lemma *LAssReds*:

$$P \vdash \langle e,\ s,\ b \rangle \rightarrow^* \langle e',\ s',\ b' \rangle \implies P \vdash \langle V := e,\ s,\ b \rangle \rightarrow^* \langle V := e',\ s',\ b' \rangle$$

lemma *LAssRedsVal*:

$$\llbracket P \vdash \langle e,\ s,\ b \rangle \rightarrow^* \langle Val\ v,\ (h',\ l',\ sh'),\ b' \rangle \rrbracket \implies P \vdash \langle V := e,\ s,\ b \rangle \rightarrow^* \langle unit,\ (h',\ l'((V \mapsto v),\ sh'),\ b' \rangle$$

lemma *LAssRedsThrow*:

$$\llbracket P \vdash \langle e,\ s,\ b \rangle \rightarrow^* \langle throw\ a,\ s',\ b' \rangle \rrbracket \implies P \vdash \langle V := e,\ s,\ b \rangle \rightarrow^* \langle throw\ a,\ s',\ b' \rangle$$

BinOp

lemma *BinOp1Reds*:

$$P \vdash \langle e,\ s,\ b \rangle \rightarrow^* \langle e',\ s',\ b' \rangle \implies P \vdash \langle e \ll bop \ e_2,\ s,\ b \rangle \rightarrow^* \langle e' \ll bop \ e_2,\ s',\ b' \rangle$$

lemma *BinOp2Reds*:

$$P \vdash \langle e,\ s,\ b \rangle \rightarrow^* \langle e',\ s',\ b' \rangle \implies P \vdash \langle (Val\ v) \ll bop \ e,\ s,\ b \rangle \rightarrow^* \langle (Val\ v) \ll bop \ e',\ s',\ b' \rangle$$

lemma *BinOpRedsVal*:

assumes $e_1\ steps: P \vdash \langle e_1,\ s_0,\ b_0 \rangle \rightarrow^* \langle Val\ v_1,\ s_1,\ b_1 \rangle$

and $e_2\ steps: P \vdash \langle e_2,\ s_1,\ b_1 \rangle \rightarrow^* \langle Val\ v_2,\ s_2,\ b_2 \rangle$

and $op: binop(bop,\ v_1,\ v_2) = Some\ v$

shows $P \vdash \langle e_1 \ll bop \ e_2,\ s_0,\ b_0 \rangle \rightarrow^* \langle Val\ v,\ s_2,\ b_2 \rangle$

lemma *BinOpRedsThrow1*:

$$P \vdash \langle e,\ s,\ b \rangle \rightarrow^* \langle throw\ e',\ s',\ b' \rangle \implies P \vdash \langle e \ll bop \ e_2,\ s,\ b \rangle \rightarrow^* \langle throw\ e',\ s',\ b' \rangle$$

lemma *BinOpRedsThrow2*:

assumes $e_1\ steps: P \vdash \langle e_1,\ s_0,\ b_0 \rangle \rightarrow^* \langle Val\ v_1,\ s_1,\ b_1 \rangle$

and $e_2\ steps: P \vdash \langle e_2,\ s_1,\ b_1 \rangle \rightarrow^* \langle throw\ e,\ s_2,\ b_2 \rangle$

shows $P \vdash \langle e_1 \ll bop \ e_2,\ s_0,\ b_0 \rangle \rightarrow^* \langle throw\ e,\ s_2,\ b_2 \rangle$

FAcc

lemma *FAccReds*:

$$P \vdash \langle e,\ s,\ b \rangle \rightarrow^* \langle e',\ s',\ b' \rangle \implies P \vdash \langle e \cdot F\{D\},\ s,\ b \rangle \rightarrow^* \langle e' \cdot F\{D\},\ s',\ b' \rangle$$

lemma *FAccRedsVal*:

$$\llbracket P \vdash \langle e,\ s,\ b \rangle \rightarrow^* \langle addr\ a,\ s',\ b' \rangle; hp\ s'\ a = Some(C,\ fs); fs(F,\ D) = Some\ v;$$

$$P \vdash C\ has\ F,\ NonStatic:t\ in\ D \rrbracket$$

$$\implies P \vdash \langle e \cdot F\{D\},\ s,\ b \rangle \rightarrow^* \langle Val\ v,\ s',\ b' \rangle$$

lemma *FAccRedsNull*:

$$P \vdash \langle e, s, b \rangle \rightarrow^* \langle \text{null}, s', b' \rangle \implies P \vdash \langle e \cdot F\{D\}, s, b \rangle \rightarrow^* \langle \text{THROW NullPointer}, s', b' \rangle$$

lemma *FAccRedsNone*:

$$\begin{aligned} & \llbracket P \vdash \langle e, s, b \rangle \rightarrow^* \langle \text{addr } a, s', b' \rangle; \\ & \quad \text{hp } s' a = \text{Some}(C, \text{fs}); \\ & \quad \neg(\exists b t. P \vdash C \text{ has } F, b:t \text{ in } D) \rrbracket \\ \implies & P \vdash \langle e \cdot F\{D\}, s, b \rangle \rightarrow^* \langle \text{THROW NoSuchFieldError}, s', b' \rangle \end{aligned}$$

lemma *FAccRedsStatic*:

$$\begin{aligned} & \llbracket P \vdash \langle e, s, b \rangle \rightarrow^* \langle \text{addr } a, s', b' \rangle; \\ & \quad \text{hp } s' a = \text{Some}(C, \text{fs}); \\ & \quad P \vdash C \text{ has } F, \text{Static}:t \text{ in } D \rrbracket \\ \implies & P \vdash \langle e \cdot F\{D\}, s, b \rangle \rightarrow^* \langle \text{THROW IncompatibleClassChangeError}, s', b' \rangle \end{aligned}$$

lemma *FAccRedsThrow*:

$$P \vdash \langle e, s, b \rangle \rightarrow^* \langle \text{throw } a, s', b' \rangle \implies P \vdash \langle e \cdot F\{D\}, s, b \rangle \rightarrow^* \langle \text{throw } a, s', b' \rangle$$

SFAcc

lemma *SFAccReds*:

$$\begin{aligned} & \llbracket P \vdash C \text{ has } F, \text{Static}:t \text{ in } D; \\ & \quad \text{shp } s D = \text{Some}(sfs, \text{Done}); \text{ sfs } F = \text{Some } v \rrbracket \\ \implies & P \vdash \langle C \cdot_s F\{D\}, s, \text{True} \rangle \rightarrow^* \langle \text{Val } v, s, \text{False} \rangle \end{aligned}$$

lemma *SFAccRedsNone*:

$$\begin{aligned} & \neg(\exists b t. P \vdash C \text{ has } F, b:t \text{ in } D) \\ \implies & P \vdash \langle C \cdot_s F\{D\}, s, b \rangle \rightarrow^* \langle \text{THROW NoSuchFieldError}, s, \text{False} \rangle \end{aligned}$$

lemma *SFAccRedsNonStatic*:

$$\begin{aligned} & P \vdash C \text{ has } F, \text{NonStatic}:t \text{ in } D \\ \implies & P \vdash \langle C \cdot_s F\{D\}, s, b \rangle \rightarrow^* \langle \text{THROW IncompatibleClassChangeError}, s, \text{False} \rangle \end{aligned}$$

lemma *SFAccInitDoneReds*:

assumes *cF*: $P \vdash C \text{ has } F, \text{Static}:t \text{ in } D$
and *shp*: $\text{shp } s D = \text{Some}(sfs, \text{Done})$ **and** *sfs*: $\text{sfs } F = \text{Some } v$
shows $P \vdash \langle C \cdot_s F\{D\}, s, b \rangle \rightarrow^* \langle \text{Val } v, s, \text{False} \rangle$

lemma *SFAccInitReds*:

assumes *cF*: $P \vdash C \text{ has } F, \text{Static}:t \text{ in } D$
and *nDone*: $\nexists \text{sfs}. \text{shp } s D = \text{Some}(sfs, \text{Done})$
and *INIT-steps*: $P \vdash \langle \text{INIT } D ([D], \text{False}) \leftarrow \text{unit}, s, \text{False} \rangle \rightarrow^* \langle \text{Val } v', s', b' \rangle$
and *shp'*: $\text{shp } s' D = \text{Some}(sfs, i)$ **and** *sfs*: $\text{sfs } F = \text{Some } v$
shows $P \vdash \langle C \cdot_s F\{D\}, s, \text{False} \rangle \rightarrow^* \langle \text{Val } v, s', \text{False} \rangle$

lemma *SFAccInitThrowReds*:

$$\begin{aligned} & \llbracket P \vdash C \text{ has } F, \text{Static}:t \text{ in } D; \\ & \quad \nexists \text{sfs}. \text{shp } s D = \text{Some}(sfs, \text{Done}); \\ & \quad P \vdash \langle \text{INIT } D ([D], \text{False}) \leftarrow \text{unit}, s, \text{False} \rangle \rightarrow^* \langle \text{throw } a, s', b' \rangle \rrbracket \\ \implies & P \vdash \langle C \cdot_s F\{D\}, s, \text{False} \rangle \rightarrow^* \langle \text{throw } a, s', b' \rangle \end{aligned}$$

FAss

lemma *FAssReds1*:

$$P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle \implies P \vdash \langle e \cdot F\{D\} := e_2, s, b \rangle \rightarrow^* \langle e' \cdot F\{D\} := e_2, s', b' \rangle$$

lemma *FAssReds2*:

$$P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle \implies P \vdash \langle \text{Val } v \cdot F\{D\} := e, s, b \rangle \rightarrow^* \langle \text{Val } v \cdot F\{D\} := e', s', b' \rangle$$

lemma *FAssRedsVal*:

assumes *e1-steps*: $P \vdash \langle e_1, s_0, b_0 \rangle \rightarrow^* \langle \text{addr } a, s_1, b_1 \rangle$
and *e2-steps*: $P \vdash \langle e_2, s_1, b_1 \rangle \rightarrow^* \langle \text{Val } v, (h_2, l_2, sh_2), b_2 \rangle$
and *cF*: $P \vdash C \text{ has } F, \text{NonStatic}:t \text{ in } D$ **and** *hC*: $\text{Some}(C, \text{fs}) = h_2 a$
shows $P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0, b_0 \rangle \rightarrow^* \langle \text{unit}, (h_2(a \mapsto (C, \text{fs}((F, D) \mapsto v))), l_2, sh_2), b_2 \rangle$

lemma *FAssRedsNull*:

assumes e_1 -steps: $P \vdash \langle e_1, s_0, b_0 \rangle \rightarrow^* \langle \text{null}, s_1, b_1 \rangle$

and e_2 -steps: $P \vdash \langle e_2, s_1, b_1 \rangle \rightarrow^* \langle \text{Val } v, s_2, b_2 \rangle$

shows $P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0, b_0 \rangle \rightarrow^* \langle \text{THROW NullPointer}, s_2, b_2 \rangle$

lemma *FAssRedsThrow1*:

$P \vdash \langle e, s, b \rangle \rightarrow^* \langle \text{throw } e', s', b' \rangle \implies P \vdash \langle e \cdot F\{D\} := e_2, s, b \rangle \rightarrow^* \langle \text{throw } e', s', b' \rangle$

lemma *FAssRedsThrow2*:

assumes e_1 -steps: $P \vdash \langle e_1, s_0, b_0 \rangle \rightarrow^* \langle \text{Val } v, s_1, b_1 \rangle$

and e_2 -steps: $P \vdash \langle e_2, s_1, b_1 \rangle \rightarrow^* \langle \text{throw } e, s_2, b_2 \rangle$

shows $P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0, b_0 \rangle \rightarrow^* \langle \text{throw } e, s_2, b_2 \rangle$

lemma *FAssRedsNone*:

assumes e_1 -steps: $P \vdash \langle e_1, s_0, b_0 \rangle \rightarrow^* \langle \text{addr } a, s_1, b_1 \rangle$

and e_2 -steps: $P \vdash \langle e_2, s_1, b_1 \rangle \rightarrow^* \langle \text{Val } v, (h_2, l_2, sh_2), b_2 \rangle$

and hC : $h_2 \ a = \text{Some}(C, fs)$ **and** ncF : $\neg(\exists b \ t. P \vdash C \ \text{has } F, b:t \ \text{in } D)$

shows $P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0, b_0 \rangle \rightarrow^* \langle \text{THROW NoSuchFieldError}, (h_2, l_2, sh_2), b_2 \rangle$

lemma *FAssRedsStatic*:

assumes e_1 -steps: $P \vdash \langle e_1, s_0, b_0 \rangle \rightarrow^* \langle \text{addr } a, s_1, b_1 \rangle$

and e_2 -steps: $P \vdash \langle e_2, s_1, b_1 \rangle \rightarrow^* \langle \text{Val } v, (h_2, l_2, sh_2), b_2 \rangle$

and hC : $h_2 \ a = \text{Some}(C, fs)$ **and** cF -Static: $P \vdash C \ \text{has } F, \text{Static}:t \ \text{in } D$

shows $P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0, b_0 \rangle \rightarrow^* \langle \text{THROW IncompatibleClassChangeError}, (h_2, l_2, sh_2), b_2 \rangle$

SFAss

lemma *SFAssReds*:

$P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle \implies P \vdash \langle C \cdot_s F\{D\} := e, s, b \rangle \rightarrow^* \langle C \cdot_s F\{D\} := e', s', b' \rangle$

lemma *SFAssRedsVal*:

assumes e_2 -steps: $P \vdash \langle e_2, s_0, b_0 \rangle \rightarrow^* \langle \text{Val } v, (h_2, l_2, sh_2), b_2 \rangle$

and cF : $P \vdash C \ \text{has } F, \text{Static}:t \ \text{in } D$

and shD : $sh_2 \ D = \lfloor (sfs, Done) \rfloor$

shows $P \vdash \langle C \cdot_s F\{D\} := e_2, s_0, b_0 \rangle \rightarrow^* \langle \text{unit}, (h_2, l_2, sh_2(D \mapsto (sfs(F \mapsto v), Done))), False \rangle$

lemma *SFAssRedsThrow*:

$\llbracket P \vdash \langle e_2, s_0, b_0 \rangle \rightarrow^* \langle \text{throw } e, s_2, b_2 \rangle \rrbracket$

$\implies P \vdash \langle C \cdot_s F\{D\} := e_2, s_0, b_0 \rangle \rightarrow^* \langle \text{throw } e, s_2, b_2 \rangle$

lemma *SFAssRedsNone*:

$\llbracket P \vdash \langle e_2, s_0, b_0 \rangle \rightarrow^* \langle \text{Val } v, (h_2, l_2, sh_2), b_2 \rangle; \neg(\exists b \ t. P \vdash C \ \text{has } F, b:t \ \text{in } D) \rrbracket \implies$

$P \vdash \langle C \cdot_s F\{D\} := e_2, s_0, b_0 \rangle \rightarrow^* \langle \text{THROW NoSuchFieldError}, (h_2, l_2, sh_2), False \rangle$

lemma *SFAssRedsNonStatic*:

$\llbracket P \vdash \langle e_2, s_0, b_0 \rangle \rightarrow^* \langle \text{Val } v, (h_2, l_2, sh_2), b_2 \rangle; P \vdash C \ \text{has } F, \text{NonStatic}:t \ \text{in } D \rrbracket \implies$

$P \vdash \langle C \cdot_s F\{D\} := e_2, s_0, b_0 \rangle \rightarrow^* \langle \text{THROW IncompatibleClassChangeError}, (h_2, l_2, sh_2), False \rangle$

lemma *SFAssInitReds*:

assumes e_2 -steps: $P \vdash \langle e_2, s_0, b_0 \rangle \rightarrow^* \langle \text{Val } v, (h_2, l_2, sh_2), False \rangle$

and cF : $P \vdash C \ \text{has } F, \text{Static}:t \ \text{in } D$

and $nDone$: $\nexists sfs. sh_2 \ D = \text{Some}(sfs, Done)$

and *INIT*-steps: $P \vdash \langle \text{INIT } D \ ([D], False) \leftarrow \text{unit}, (h_2, l_2, sh_2), False \rangle \rightarrow^* \langle \text{Val } v', (h', l', sh'), b' \rangle$

and $sh'D$: $sh' \ D = \text{Some}(sfs, i)$

and sfs' : $sfs' = sfs(F \mapsto v)$ **and** sh'' : $sh'' = sh'(D \mapsto (sfs', i))$

shows $P \vdash \langle C \cdot_s F\{D\} := e_2, s_0, b_0 \rangle \rightarrow^* \langle \text{unit}, (h', l', sh''), False \rangle$

lemma *SFAssInitThrowReds*:

assumes e_2 -steps: $P \vdash \langle e_2, s_0, b_0 \rangle \rightarrow^* \langle \text{Val } v, (h_2, l_2, sh_2), False \rangle$

and cF : $P \vdash C \ \text{has } F, \text{Static}:t \ \text{in } D$

and $nDone$: $\nexists sfs. sh_2 \ D = \text{Some}(sfs, Done)$

and *INIT*-steps: $P \vdash \langle \text{INIT } D \ ([D], False) \leftarrow \text{unit}, (h_2, l_2, sh_2), False \rangle \rightarrow^* \langle \text{throw } a, s', b' \rangle$

shows $P \vdash \langle C \cdot_s F\{D\} := e_2, s_0, b_0 \rangle \rightarrow^* \langle \text{throw } a, s', b' \rangle$

;;

lemma *SeqReds*:

$P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle \implies P \vdash \langle e;;e_2, s, b \rangle \rightarrow^* \langle e';e_2, s', b' \rangle$

lemma *SeqRedsThrow*:

$P \vdash \langle e, s, b \rangle \rightarrow^* \langle \text{throw } e', s', b' \rangle \implies P \vdash \langle e;;e_2, s, b \rangle \rightarrow^* \langle \text{throw } e', s', b' \rangle$

lemma *SeqReds2*:

assumes *e*-steps: $P \vdash \langle e_1, s_0, b_0 \rangle \rightarrow^* \langle \text{Val } v_1, s_1, b_1 \rangle$

and *e*₂-steps: $P \vdash \langle e_2, s_1, b_1 \rangle \rightarrow^* \langle e_2', s_2, b_2 \rangle$

shows $P \vdash \langle e_1;;e_2, s_0, b_0 \rangle \rightarrow^* \langle e_2', s_2, b_2 \rangle$

If

lemma *CondReds*:

$P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle \implies P \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s, b \rangle \rightarrow^* \langle \text{if } (e') e_1 \text{ else } e_2, s', b' \rangle$

lemma *CondRedsThrow*:

$P \vdash \langle e, s, b \rangle \rightarrow^* \langle \text{throw } a, s', b' \rangle \implies P \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s, b \rangle \rightarrow^* \langle \text{throw } a, s', b' \rangle$

lemma *CondReds2T*:

assumes *e*-steps: $P \vdash \langle e, s_0, b_0 \rangle \rightarrow^* \langle \text{true}, s_1, b_1 \rangle$

and *e*₁-steps: $P \vdash \langle e_1, s_1, b_1 \rangle \rightarrow^* \langle e', s_2, b_2 \rangle$

shows $P \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s_0, b_0 \rangle \rightarrow^* \langle e', s_2, b_2 \rangle$

lemma *CondReds2F*:

assumes *e*-steps: $P \vdash \langle e, s_0, b_0 \rangle \rightarrow^* \langle \text{false}, s_1, b_1 \rangle$

and *e*₂-steps: $P \vdash \langle e_2, s_1, b_1 \rangle \rightarrow^* \langle e', s_2, b_2 \rangle$

shows $P \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s_0, b_0 \rangle \rightarrow^* \langle e', s_2, b_2 \rangle$

While

lemma *WhileFReds*:

assumes *b*-steps: $P \vdash \langle b, s, b_0 \rangle \rightarrow^* \langle \text{false}, s', b' \rangle$

shows $P \vdash \langle \text{while } (b) c, s, b_0 \rangle \rightarrow^* \langle \text{unit}, s', b' \rangle$

lemma *WhileRedsThrow*:

assumes *b*-steps: $P \vdash \langle b, s, b_0 \rangle \rightarrow^* \langle \text{throw } e, s', b' \rangle$

shows $P \vdash \langle \text{while } (b) c, s, b_0 \rangle \rightarrow^* \langle \text{throw } e, s', b' \rangle$

lemma *WhileTReds*:

assumes *b*-steps: $P \vdash \langle b, s_0, b_0 \rangle \rightarrow^* \langle \text{true}, s_1, b_1 \rangle$

and *c*-steps: $P \vdash \langle c, s_1, b_1 \rangle \rightarrow^* \langle \text{Val } v_1, s_2, b_2 \rangle$

and *while*-steps: $P \vdash \langle \text{while } (b) c, s_2, b_2 \rangle \rightarrow^* \langle e, s_3, b_3 \rangle$

shows $P \vdash \langle \text{while } (b) c, s_0, b_0 \rangle \rightarrow^* \langle e, s_3, b_3 \rangle$

lemma *WhileTRedsThrow*:

assumes *b*-steps: $P \vdash \langle b, s_0, b_0 \rangle \rightarrow^* \langle \text{true}, s_1, b_1 \rangle$

and *c*-steps: $P \vdash \langle c, s_1, b_1 \rangle \rightarrow^* \langle \text{throw } e, s_2, b_2 \rangle$

shows $P \vdash \langle \text{while } (b) c, s_0, b_0 \rangle \rightarrow^* \langle \text{throw } e, s_2, b_2 \rangle$

Throw

lemma *ThrowReds*:

$P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle \implies P \vdash \langle \text{throw } e, s, b \rangle \rightarrow^* \langle \text{throw } e', s', b' \rangle$

lemma *ThrowRedsNull*:

$P \vdash \langle e, s, b \rangle \rightarrow^* \langle \text{null}, s', b' \rangle \implies P \vdash \langle \text{throw } e, s, b \rangle \rightarrow^* \langle \text{THROW } \text{NullPointer}, s', b' \rangle$

lemma *ThrowRedsThrow*:

$P \vdash \langle e, s, b \rangle \rightarrow^* \langle \text{throw } a, s', b' \rangle \implies P \vdash \langle \text{throw } e, s, b \rangle \rightarrow^* \langle \text{throw } a, s', b' \rangle$

InitBlock

lemma *InitBlockReds-aux*:

$$\begin{aligned} P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle &\implies \\ \forall h \ l \ sh \ h' \ l' \ sh' \ v. \ s = (h, l(V \mapsto v), sh) \longrightarrow s' = (h', l', sh') \longrightarrow \\ P \vdash \langle \{V:T := Val \ v; e\}, (h, l, sh), b \rangle \rightarrow^* \langle \{V:T := Val(the(l' \ V)); e'\}, (h', l'(V := (l \ V))), sh', b' \rangle \end{aligned}$$

lemma *InitBlockReds*:

$$\begin{aligned} P \vdash \langle e, (h, l(V \mapsto v), sh), b \rangle \rightarrow^* \langle e', (h', l', sh'), b' \rangle &\implies \\ P \vdash \langle \{V:T := Val \ v; e\}, (h, l, sh), b \rangle \rightarrow^* \langle \{V:T := Val(the(l' \ V)); e'\}, (h', l'(V := (l \ V))), sh', b' \rangle \end{aligned}$$

lemma *InitBlockRedsFinal*:

$$\begin{aligned} \llbracket P \vdash \langle e, (h, l(V \mapsto v), sh), b \rangle \rightarrow^* \langle e', (h', l', sh'), b' \rangle; \text{final } e' \rrbracket &\implies \\ P \vdash \langle \{V:T := Val \ v; e\}, (h, l, sh), b \rangle \rightarrow^* \langle e', (h', l'(V := l \ V), sh'), b' \rangle \end{aligned}$$

Block

lemmas *converse-rtranclE3 = converse-rtranclE* [of (xa, xb, xc) (za, zb, zc) , *split-rule*]

lemma *BlockRedsFinal*:

assumes *reds*: $P \vdash \langle e_0, s_0, b_0 \rangle \rightarrow^* \langle e_2, (h_2, l_2, sh_2), b_2 \rangle$ **and** *fin*: *final* e_2
shows $\bigwedge h_0 \ l_0 \ sh_0. \ s_0 = (h_0, l_0(V := None), sh_0) \implies P \vdash \langle \{V:T; e_0\}, (h_0, l_0, sh_0), b_0 \rangle \rightarrow^* \langle e_2, (h_2, l_2(V := l_0 \ V), sh_2), b_2 \rangle$

try-catch

lemma *TryReds*:

$$P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle \implies P \vdash \langle \text{try } e \text{ catch}(C \ V) \ e_2, s, b \rangle \rightarrow^* \langle \text{try } e' \text{ catch}(C \ V) \ e_2, s', b' \rangle$$

lemma *TryRedsVal*:

$$P \vdash \langle e, s, b \rangle \rightarrow^* \langle Val \ v, s', b' \rangle \implies P \vdash \langle \text{try } e \text{ catch}(C \ V) \ e_2, s, b \rangle \rightarrow^* \langle Val \ v, s', b' \rangle$$

lemma *TryCatchRedsFinal*:

assumes *e1-steps*: $P \vdash \langle e_1, s_0, b_0 \rangle \rightarrow^* \langle Throw \ a, (h_1, l_1, sh_1), b_1 \rangle$
and $h_1 \ a = Some(D, fs)$ **and** *sub*: $P \vdash D \preceq^* C$
and *e2-steps*: $P \vdash \langle e_2, (h_1, l_1(V \mapsto Addr \ a), sh_1), b_1 \rangle \rightarrow^* \langle e_2', (h_2, l_2, sh_2), b_2 \rangle$
and *final*: *final* e_2'

shows $P \vdash \langle \text{try } e_1 \text{ catch}(C \ V) \ e_2, s_0, b_0 \rangle \rightarrow^* \langle e_2', (h_2, (l_2::locals)(V := l_1 \ V), sh_2), b_2 \rangle$

lemma *TryRedsFail*:

$$\begin{aligned} \llbracket P \vdash \langle e, s, b \rangle \rightarrow^* \langle Throw \ a, (h, l, sh), b' \rangle; h \ a = Some(D, fs); \neg P \vdash D \preceq^* C \rrbracket \\ \implies P \vdash \langle \text{try } e_1 \text{ catch}(C \ V) \ e_2, s, b \rangle \rightarrow^* \langle Throw \ a, (h, l, sh), b' \rangle \end{aligned}$$

List

lemma *ListReds1*:

$$P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle \implies P \vdash \langle e \# es, s, b \rangle [\rightarrow]^* \langle e' \# es, s', b' \rangle$$

lemma *ListReds2*:

$$P \vdash \langle es, s, b \rangle [\rightarrow]^* \langle es', s', b' \rangle \implies P \vdash \langle Val \ v \# es, s, b \rangle [\rightarrow]^* \langle Val \ v \# es', s', b' \rangle$$

lemma *ListRedsVal*:

$$\begin{aligned} \llbracket P \vdash \langle e, s_0, b_0 \rangle \rightarrow^* \langle Val \ v, s_1, b_1 \rangle; P \vdash \langle es, s_1, b_1 \rangle [\rightarrow]^* \langle es', s_2, b_2 \rangle \rrbracket \\ \implies P \vdash \langle e \# es, s_0, b_0 \rangle [\rightarrow]^* \langle Val \ v \# es', s_2, b_2 \rangle \end{aligned}$$

Call

First a few lemmas on what happens to free variables during redction.

lemma **assumes** *wf*: *wf-J-prog* P

shows *Red-fv*: $P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle \implies fv \ e' \subseteq fv \ e$
and $P \vdash \langle es, (h, l, sh), b \rangle [\rightarrow] \langle es', (h', l', sh'), b' \rangle \implies fvs \ es' \subseteq fvs \ es$

lemma Red-dom-lcl:

$P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e', (h', l', sh'), b' \rangle \implies \text{dom } l' \subseteq \text{dom } l \cup \text{fv } e$ **and**
 $P \vdash \langle es, (h, l, sh), b \rangle [\mapsto] \langle es', (h', l', sh'), b' \rangle \implies \text{dom } l' \subseteq \text{dom } l \cup \text{fvs } es$

lemma Reds-dom-lcl:

assumes $wf: wuf\text{-}J\text{-prog } P$

shows $P \vdash \langle e, (h, l, sh), b \rangle \rightarrow^* \langle e', (h', l', sh'), b' \rangle \implies \text{dom } l' \subseteq \text{dom } l \cup \text{fv } e$

Now a few lemmas on the behaviour of blocks during reduction.

lemma override-on-upd-lemma:

$(\text{override-on } f (g(a \mapsto b)) A)(a := g a) = \text{override-on } f g (\text{insert } a A)$

lemma blocksReds:

$\wedge l. \llbracket \text{length } Vs = \text{length } Ts; \text{length } vs = \text{length } Ts; \text{distinct } Vs; \\ P \vdash \langle e, (h, l(Vs [\mapsto] vs), sh), b \rangle \rightarrow^* \langle e', (h', l', sh'), b' \rangle \rrbracket \\ \implies P \vdash \langle \text{blocks}(Vs, Ts, vs, e), (h, l, sh), b \rangle \rightarrow^* \langle \text{blocks}(Vs, Ts, \text{map } (the \circ l') Vs, e'), (h', \text{override-on } l' l (\text{set } Vs), sh'), b' \rangle$

lemma blocksFinal:

$\wedge l. \llbracket \text{length } Vs = \text{length } Ts; \text{length } vs = \text{length } Ts; \text{final } e \rrbracket \implies \\ P \vdash \langle \text{blocks}(Vs, Ts, vs, e), (h, l, sh), b \rangle \rightarrow^* \langle e, (h, l, sh), b \rangle$

lemma blocksRedsFinal:

assumes $wf: \text{length } Vs = \text{length } Ts \text{ length } vs = \text{length } Ts \text{ distinct } Vs$
and $\text{reds}: P \vdash \langle e, (h, l(Vs [\mapsto] vs), sh), b \rangle \rightarrow^* \langle e', (h', l', sh'), b' \rangle$
and $\text{fin}: \text{final } e' \text{ and } l'': l'' = \text{override-on } l' l (\text{set } Vs)$
shows $P \vdash \langle \text{blocks}(Vs, Ts, vs, e), (h, l, sh), b \rangle \rightarrow^* \langle e', (h', l'', sh'), b' \rangle$

An now the actual method call reduction lemmas.

lemma CallRedsObj:

$P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle \implies P \vdash \langle e \cdot M(es), s, b \rangle \rightarrow^* \langle e' \cdot M(es), s', b' \rangle$

lemma CallRedsParams:

$P \vdash \langle es, s, b \rangle [\mapsto]^* \langle es', s', b' \rangle \implies P \vdash \langle (Val v) \cdot M(es), s, b \rangle \rightarrow^* \langle (Val v) \cdot M(es'), s', b' \rangle$

lemma CallRedsFinal:

assumes $wuf: wuf\text{-}J\text{-prog } P$

and $P \vdash \langle e, s_0, b_0 \rangle \rightarrow^* \langle \text{addr } a, s_1, b_1 \rangle$

$P \vdash \langle es, s_1, b_1 \rangle [\mapsto]^* \langle \text{map } Val vs, (h_2, l_2, sh_2), b_2 \rangle$

$h_2 a = \text{Some}(C.fs) P \vdash C \text{ sees } M, \text{NonStatic}: Ts \rightarrow T = (\text{pns}, \text{body}) \text{ in } D$

$\text{size } vs = \text{size } \text{pns}$

and $l_2': l_2' = [\text{this} \mapsto \text{Addr } a, \text{pns}[\mapsto] vs]$

and $\text{body}: P \vdash \langle \text{body}, (h_2, l_2', sh_2), b_2 \rangle \rightarrow^* \langle ef, (h_3, l_3, sh_3), b_3 \rangle$

and $\text{final } ef$

shows $P \vdash \langle e \cdot M(es), s_0, b_0 \rangle \rightarrow^* \langle ef, (h_3, l_2, sh_3), b_3 \rangle$

lemma CallRedsThrowParams:

assumes $e\text{-steps}: P \vdash \langle e, s_0, b_0 \rangle \rightarrow^* \langle Val v, s_1, b_1 \rangle$

and $es\text{-steps}: P \vdash \langle es, s_1, b_1 \rangle [\mapsto]^* \langle \text{map } Val vs_1 @ \text{throw } a \# es_2, s_2, b_2 \rangle$

shows $P \vdash \langle e \cdot M(es), s_0, b_0 \rangle \rightarrow^* \langle \text{throw } a, s_2, b_2 \rangle$

lemma CallRedsThrowObj:

$P \vdash \langle e, s_0, b_0 \rangle \rightarrow^* \langle \text{throw } a, s_1, b_1 \rangle \implies P \vdash \langle e \cdot M(es), s_0, b_0 \rangle \rightarrow^* \langle \text{throw } a, s_1, b_1 \rangle$

lemma CallRedsNull:

assumes $e\text{-steps}$: $P \vdash \langle e, s_0, b_0 \rangle \rightarrow^* \langle \text{null}, s_1, b_1 \rangle$
and $es\text{-steps}$: $P \vdash \langle es, s_1, b_1 \rangle [\rightarrow]^* \langle \text{map Val } vs, s_2, b_2 \rangle$
shows $P \vdash \langle e \cdot M(es), s_0, b_0 \rangle \rightarrow^* \langle \text{THROW NullPointer}, s_2, b_2 \rangle$
lemma CallRedsNone :
assumes $e\text{-steps}$: $P \vdash \langle e, s, b \rangle \rightarrow^* \langle \text{addr } a, s_1, b_1 \rangle$
and $es\text{-steps}$: $P \vdash \langle es, s_1, b_1 \rangle [\rightarrow]^* \langle \text{map Val } vs, s_2, b_2 \rangle$
and hp_2a : $hp \ s_2 \ a = \text{Some}(C.fs)$
and ncM : $\neg(\exists b \ Ts \ T \ m \ D. \ P \vdash \ C \ \text{sees } M, b: Ts \rightarrow T = m \ \text{in } D)$
shows $P \vdash \langle e \cdot M(es), s, b \rangle \rightarrow^* \langle \text{THROW NoSuchMethodError}, s_2, b_2 \rangle$
lemma CallRedsStatic :
assumes $e\text{-steps}$: $P \vdash \langle e, s, b \rangle \rightarrow^* \langle \text{addr } a, s_1, b_1 \rangle$
and $es\text{-steps}$: $P \vdash \langle es, s_1, b_1 \rangle [\rightarrow]^* \langle \text{map Val } vs, s_2, b_2 \rangle$
and hp_2a : $hp \ s_2 \ a = \text{Some}(C.fs)$
and $cM\text{-Static}$: $P \vdash \ C \ \text{sees } M, \text{Static}: Ts \rightarrow T = m \ \text{in } D$
shows $P \vdash \langle e \cdot M(es), s, b \rangle \rightarrow^* \langle \text{THROW IncompatibleClassChangeError}, s_2, b_2 \rangle$

1.23.2 SCall

lemma SCallRedsParams :
 $P \vdash \langle es, s, b \rangle [\rightarrow]^* \langle es', s', b' \rangle \implies P \vdash \langle C \cdot_s M(es), s, b \rangle \rightarrow^* \langle C \cdot_s M(es'), s', b' \rangle$
lemma SCallRedsFinal :
assumes wwf : $wwf\text{-J-prog } P$
and $P \vdash \langle es, s_0, b_0 \rangle [\rightarrow]^* \langle \text{map Val } vs, (h_2, l_2, sh_2), b_2 \rangle$
 $P \vdash \ C \ \text{sees } M, \text{Static}: Ts \rightarrow T = (pns, body) \ \text{in } D$
 $sh_2 \ D = \text{Some}(sfs, Done) \vee (M = \text{clinit} \wedge sh_2 \ D = \lfloor (sfs, Processing) \rfloor)$
 $\text{size } vs = \text{size } pns$
and l_2' : $l_2' = [pns[\rightarrow]vs]$
and $body$: $P \vdash \langle body, (h_2, l_2', sh_2), False \rangle \rightarrow^* \langle ef, (h_3, l_3, sh_3), b_3 \rangle$
and $final \ ef$
shows $P \vdash \langle C \cdot_s M(es), s_0, b_0 \rangle \rightarrow^* \langle ef, (h_3, l_2, sh_3), b_3 \rangle$
lemma $\text{SCallRedsThrowParams}$:
 $\llbracket P \vdash \langle es, s_0, b_0 \rangle [\rightarrow]^* \langle \text{map Val } vs_1 \ @ \ \text{throw } a \ \# \ es_2, s_2, b_2 \rangle \rrbracket$
 $\implies P \vdash \langle C \cdot_s M(es), s_0, b_0 \rangle \rightarrow^* \langle \text{throw } a, s_2, b_2 \rangle$
lemma SCallRedsNone :
 $\llbracket P \vdash \langle es, s, b \rangle [\rightarrow]^* \langle \text{map Val } vs, s_2, False \rangle;$
 $\neg(\exists b \ Ts \ T \ m \ D. \ P \vdash \ C \ \text{sees } M, b: Ts \rightarrow T = m \ \text{in } D) \rrbracket$
 $\implies P \vdash \langle C \cdot_s M(es), s, b \rangle \rightarrow^* \langle \text{THROW NoSuchMethodError}, s_2, False \rangle$
lemma $\text{SCallRedsNonStatic}$:
 $\llbracket P \vdash \langle es, s, b \rangle [\rightarrow]^* \langle \text{map Val } vs, s_2, False \rangle;$
 $P \vdash \ C \ \text{sees } M, \text{NonStatic}: Ts \rightarrow T = m \ \text{in } D \rrbracket$
 $\implies P \vdash \langle C \cdot_s M(es), s, b \rangle \rightarrow^* \langle \text{THROW IncompatibleClassChangeError}, s_2, False \rangle$
lemma $\text{SCallInitThrowReds}$:
assumes wwf : $wwf\text{-J-prog } P$
and $P \vdash \langle es, s_0, b_0 \rangle [\rightarrow]^* \langle \text{map Val } vs, (h_1, l_1, sh_1), False \rangle$
 $P \vdash \ C \ \text{sees } M, \text{Static}: Ts \rightarrow T = (pns, body) \ \text{in } D$
 $\nexists sfs. \ sh_1 \ D = \text{Some}(sfs, Done)$
 $M \neq \text{clinit}$
and $P \vdash \langle \text{INIT } D \ ([D], False) \leftarrow \text{unit}, (h_1, l_1, sh_1), False \rangle \rightarrow^* \langle \text{throw } a, (h_2, l_2, sh_2), b_2 \rangle$
shows $P \vdash \langle C \cdot_s M(es), s_0, b_0 \rangle \rightarrow^* \langle \text{throw } a, (h_2, l_2, sh_2), b_2 \rangle$
lemma SCallInitReds :
assumes wwf : $wwf\text{-J-prog } P$
and $P \vdash \langle es, s_0, b_0 \rangle [\rightarrow]^* \langle \text{map Val } vs, (h_1, l_1, sh_1), False \rangle$
 $P \vdash \ C \ \text{sees } M, \text{Static}: Ts \rightarrow T = (pns, body) \ \text{in } D$
 $\nexists sfs. \ sh_1 \ D = \text{Some}(sfs, Done)$

$M \neq \text{clinit}$
and $P \vdash \langle \text{INIT } D ([D], \text{False}) \leftarrow \text{unit}, (h_1, l_1, sh_1), \text{False} \rangle \rightarrow^* \langle \text{Val } v', (h_2, l_2, sh_2), b_2 \rangle$
and $\text{size } vs = \text{size } pns$
and $l_2': l_2' = [pns[\mapsto]vs]$
and $\text{body}: P \vdash \langle \text{body}, (h_2, l_2', sh_2), \text{False} \rangle \rightarrow^* \langle \text{ef}, (h_3, l_3, sh_3), b_3 \rangle$
and $\text{final } \text{ef}$
shows $P \vdash \langle C \cdot_s M(es), s_0, b_0 \rangle \rightarrow^* \langle \text{ef}, (h_3, l_2, sh_3), b_3 \rangle$
lemma *SCallInitProcessingReds*:
assumes $wuf: wuf\text{-}J\text{-prog } P$
and $P \vdash \langle es, s_0, b_0 \rangle [\mapsto]^* \langle \text{map Val } vs, (h_2, l_2, sh_2), b_2 \rangle$
 $P \vdash C \text{ sees } M, \text{Static}: Ts \rightarrow T = (pns, \text{body}) \text{ in } D$
 $sh_2 \ D = \text{Some}(sfs, \text{Processing})$
and $\text{size } vs = \text{size } pns$
and $l_2': l_2' = [pns[\mapsto]vs]$
and $\text{body}: P \vdash \langle \text{body}, (h_2, l_2', sh_2), \text{False} \rangle \rightarrow^* \langle \text{ef}, (h_3, l_3, sh_3), b_3 \rangle$
and $\text{final } \text{ef}$
shows $P \vdash \langle C \cdot_s M(es), s_0, b_0 \rangle \rightarrow^* \langle \text{ef}, (h_3, l_2, sh_3), b_3 \rangle$

The main Theorem

lemma **assumes** $wuf: wuf\text{-}J\text{-prog } P$
shows *big-by-small*: $P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$
 $\Rightarrow (\bigwedge b. \text{iconf } (shp \ s) \ e \Rightarrow P, shp \ s \vdash_b (e, b) \ \checkmark \Rightarrow P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', \text{False} \rangle)$
and *big-by-small*: $P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle$
 $\Rightarrow (\bigwedge b. \text{iconfs } (shp \ s) \ es \Rightarrow P, shp \ s \vdash_b (es, b) \ \checkmark \Rightarrow P \vdash \langle es, s, b \rangle [\mapsto]^* \langle es', s', \text{False} \rangle)$

1.23.3 Big steps simulates small step

This direction was carried out by Norbert Schirmer and Daniel Wasserrab (and modified to include statics and DCI by Susannah Mansky).

The big step equivalent of *RedWhile*:

lemma *unfold-while*:

$$P \vdash \langle \text{while}(b) \ c, s \rangle \Rightarrow \langle e', s' \rangle = P \vdash \langle \text{if}(b) \ (c;; \text{while}(b) \ c) \ \text{else } (\text{unit}), s \rangle \Rightarrow \langle e', s' \rangle$$

lemma *blocksEval*:

$$\bigwedge Ts \ \text{vs} \ l \ l'. \llbracket \text{size } ps = \text{size } Ts; \text{size } ps = \text{size } vs; P \vdash \langle \text{blocks}(ps, Ts, vs, e), (h, l, sh) \rangle \Rightarrow \langle e', (h', l', sh') \rangle \rrbracket \\ \Rightarrow \exists l''. P \vdash \langle e, (h, l(ps[\mapsto]vs), sh) \rangle \Rightarrow \langle e', (h', l'', sh') \rangle$$

lemma

assumes $wf: wuf\text{-}J\text{-prog } P$

shows *eval-restrict-lcl*:

$$P \vdash \langle e, (h, l, sh) \rangle \Rightarrow \langle e', (h', l', sh') \rangle \Rightarrow (\bigwedge W. \text{fv } e \subseteq W \Rightarrow P \vdash \langle e, (h, l|'W, sh) \rangle \Rightarrow \langle e', (h', l'|'W, sh') \rangle)$$

and $P \vdash \langle es, (h, l, sh) \rangle [\Rightarrow] \langle es', (h', l', sh') \rangle \Rightarrow (\bigwedge W. \text{fvs } es \subseteq W \Rightarrow P \vdash \langle es, (h, l|'W, sh) \rangle [\Rightarrow] \langle es', (h', l'|'W, sh') \rangle)$

lemma *eval-notfree-unchanged*:

$$P \vdash \langle e, (h, l, sh) \rangle \Rightarrow \langle e', (h', l', sh') \rangle \Rightarrow (\bigwedge V. V \notin \text{fv } e \Rightarrow l' \ V = l \ V)$$

and $P \vdash \langle es, (h, l, sh) \rangle [\Rightarrow] \langle es', (h', l', sh') \rangle \Rightarrow (\bigwedge V. V \notin \text{fvs } es \Rightarrow l' \ V = l \ V)$

lemma *eval-closed-lcl-unchanged*:

$$\llbracket P \vdash \langle e, (h, l, sh) \rangle \Rightarrow \langle e', (h', l', sh') \rangle; \text{fv } e = \{\} \rrbracket \Rightarrow l' = l$$

lemma *list-eval-Throw*:

assumes *eval-e*: $P \vdash \langle \text{throw } x, s \rangle \Rightarrow \langle e', s' \rangle$

shows $P \vdash \langle \text{map Val vs } @ \text{ throw } x \# \text{ es}', s \rangle [\Rightarrow] \langle \text{map Val vs } @ \text{ e}' \# \text{ es}', s' \rangle$
 — separate evaluation of first subexp of a sequence

lemma *seq-ext*:

assumes $IH: \bigwedge e' s'. P \vdash \langle e'', s'' \rangle \Rightarrow \langle e', s' \rangle \Longrightarrow P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$

and $seq: P \vdash \langle e'' ;; e_0, s'' \rangle \Rightarrow \langle e', s' \rangle$

shows $P \vdash \langle e ;; e_0, s \rangle \Rightarrow \langle e', s' \rangle$

proof(*rule eval-cases(14)[OF seq]*) — Seq

fix $v' s_1$ assume $e'': P \vdash \langle e'', s'' \rangle \Rightarrow \langle \text{Val } v', s_1 \rangle$ and $estep: P \vdash \langle e_0, s_1 \rangle \Rightarrow \langle e', s' \rangle$

have $P \vdash \langle e, s \rangle \Rightarrow \langle \text{Val } v', s_1 \rangle$ using $e'' IH$ by *simp*

then show *?thesis* using *estep Seq* by *simp*

next

fix e_t assume $e'': P \vdash \langle e'', s'' \rangle \Rightarrow \langle \text{throw } e_t, s' \rangle$ and $e': e' = \text{throw } e_t$

have $P \vdash \langle e, s \rangle \Rightarrow \langle \text{throw } e_t, s' \rangle$ using $e'' IH$ by *simp*

then show *?thesis* using *eval-vals.SeqThrow e'* by *simp*

qed

— separate evaluation of *RI* subexp, val case

lemma *rinit-Val-ext*:

assumes $ri: P \vdash \langle RI (C, e'') ; Cs \leftarrow e_0, s'' \rangle \Rightarrow \langle \text{Val } v', s_1 \rangle$

and $IH: \bigwedge e' s'. P \vdash \langle e'', s'' \rangle \Rightarrow \langle e', s' \rangle \Longrightarrow P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$

shows $P \vdash \langle RI (C, e) ; Cs \leftarrow e_0, s \rangle \Rightarrow \langle \text{Val } v', s_1 \rangle$

proof(*rule eval-cases(20)[OF ri]*) — RI

fix $v'' h' l' sh' sfs i$

assume $e''step: P \vdash \langle e'', s'' \rangle \Rightarrow \langle \text{Val } v'', (h', l', sh') \rangle$

and $shC: sh' C = [(sfs, i)]$

and $init: P \vdash \langle \text{INIT (if } Cs = [] \text{ then } C \text{ else last } Cs) (Cs, True) \leftarrow e_0, (h', l', sh'(C \mapsto (sfs, Done))) \rangle$

\Rightarrow

$\langle \text{Val } v', s_1 \rangle$

have $P \vdash \langle e, s \rangle \Rightarrow \langle \text{Val } v'', (h', l', sh') \rangle$ using $IH[OF e''step]$ by *simp*

then show *?thesis* using *RInit init shC* by *auto*

next

fix $a h' l' sh' sfs i D Cs'$

assume $e''step: P \vdash \langle e'', s'' \rangle \Rightarrow \langle \text{throw } a, (h', l', sh') \rangle$

and $riD: P \vdash \langle RI (D, \text{throw } a) ; Cs' \leftarrow e_0, (h', l', sh'(C \mapsto (sfs, Error))) \rangle \Rightarrow \langle \text{Val } v', s_1 \rangle$

have $P \vdash \langle e, s \rangle \Rightarrow \langle \text{throw } a, (h', l', sh') \rangle$ using $IH[OF e''step]$ by *simp*

then show *?thesis* using riD *rinit-throwE* by *blast*

qed(*simp*)

— separate evaluation of *RI* subexp, throw case

lemma *rinit-throw-ext*:

assumes $ri: P \vdash \langle RI (C, e'') ; Cs \leftarrow e_0, s'' \rangle \Rightarrow \langle \text{throw } e_t, s' \rangle$

and $IH: \bigwedge e' s'. P \vdash \langle e'', s'' \rangle \Rightarrow \langle e', s' \rangle \Longrightarrow P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$

shows $P \vdash \langle RI (C, e) ; Cs \leftarrow e_0, s \rangle \Rightarrow \langle \text{throw } e_t, s' \rangle$

proof(*rule eval-cases(20)[OF ri]*) — RI

fix $v h' l' sh' sfs i$

assume $e''step: P \vdash \langle e'', s'' \rangle \Rightarrow \langle \text{Val } v, (h', l', sh') \rangle$

and $shC: sh' C = [(sfs, i)]$

and $init: P \vdash \langle \text{INIT (if } Cs = [] \text{ then } C \text{ else last } Cs) (Cs, True) \leftarrow e_0, (h', l', sh'(C \mapsto (sfs, Done))) \rangle$

\Rightarrow

$\langle \text{throw } e_t, s' \rangle$

have $P \vdash \langle e, s \rangle \Rightarrow \langle \text{Val } v, (h', l', sh') \rangle$ using $IH[OF e''step]$ by *simp*

then show *?thesis* using *RInit init shC* by *auto*

next

fix $a h' l' sh' sfs i D Cs'$

assume $e''step: P \vdash \langle e'', s'' \rangle \Rightarrow \langle throw\ a, (h', l', sh') \rangle$
and $shC: sh' C = [(sfs, i)]$
and $riD: P \vdash \langle RI\ (D, throw\ a) ; Cs' \leftarrow e_0, (h', l', sh'(C \mapsto (sfs, Error))) \rangle \Rightarrow \langle throw\ e_t, s' \rangle$
and $cons: Cs = D \# Cs'$
have $estep': P \vdash \langle e, s \rangle \Rightarrow \langle throw\ a, (h', l', sh') \rangle$ **using** $IH[OF\ e''step]$ **by** $simp$
then show $?thesis$ **using** $RInitInitFail\ cons\ riD\ shC$ **by** $simp$

next

fix $a\ h'\ l'\ sh'\ sfs\ i$
assume $throw\ e_t = throw\ a$
and $s' = (h', l', sh'(C \mapsto (sfs, Error)))$
and $P \vdash \langle e'', s'' \rangle \Rightarrow \langle throw\ a, (h', l', sh') \rangle$
and $sh' C = [(sfs, i)]$
and $Cs = []$
then show $?thesis$ **using** $RInitFailFinal\ IH$ **by** $auto$
qed

— separate evaluation of RI subexp

lemma $rinit-ext$:

assumes $IH: \bigwedge e' s'. P \vdash \langle e'', s'' \rangle \Rightarrow \langle e', s' \rangle \Longrightarrow P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$
shows $\bigwedge e' s'. P \vdash \langle RI\ (C, e'') ; Cs \leftarrow e_0, s'' \rangle \Rightarrow \langle e', s' \rangle$
 $\Longrightarrow P \vdash \langle RI\ (C, e) ; Cs \leftarrow e_0, s \rangle \Rightarrow \langle e', s' \rangle$

proof —

fix $e' s'$ **assume** $ri'': P \vdash \langle RI\ (C, e'') ; Cs \leftarrow e_0, s'' \rangle \Rightarrow \langle e', s' \rangle$
then have $final\ e'$ **using** $eval-final$ **by** $simp$
then show $P \vdash \langle RI\ (C, e) ; Cs \leftarrow e_0, s \rangle \Rightarrow \langle e', s' \rangle$
proof($rule\ finalE$)
fix v **assume** $e' = Val\ v$ **then show** $?thesis$ **using** $rinit-Val-ext[OF - IH]$ ri'' **by** $simp$
next
fix a **assume** $e' = throw\ a$ **then show** $?thesis$ **using** $rinit-throw-ext[OF - IH]$ ri'' **by** $simp$
qed
qed

— $INIT$ and RI return either Val with $Done$ or $Processing$ flag or $Throw$ with $Error$ flag

lemma

shows $eval-init-return: P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$
 $\Longrightarrow iconf\ (shp\ s)\ e$
 $\Longrightarrow (\exists Cs\ b. e = INIT\ C' (Cs, b) \leftarrow unit) \vee (\exists C\ e_0\ Cs\ e_i. e = RI(C, e_0); Cs@[C'] \leftarrow unit)$
 $\vee (\exists e_0. e = RI(C', e_0); Nil \leftarrow unit)$
 $\Longrightarrow (val-of\ e' = Some\ v \longrightarrow (\exists sfs\ i. shp\ s'\ C' = [(sfs, i)] \wedge (i = Done \vee i = Processing)))$
 $\wedge (throw-of\ e' = Some\ a \longrightarrow (\exists sfs\ i. shp\ s'\ C' = [(sfs, Error)]))$

and $P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \Longrightarrow True$

proof($induct\ rule: eval-evals.inducts$)

case ($InitFinal\ e\ s\ e'\ s'\ C\ b$) **then show** $?case$
by($fastforce\ simp: initPD-def\ dest: eval-final-same$)

next

case ($InitDone\ sh\ C\ sfs\ C'\ Cs\ e\ h\ l\ e'\ s'$)
then have $final\ e'$ **using** $eval-final$ **by** $simp$
then show $?case$
proof($rule\ finalE$)
fix v **assume** $e': e' = Val\ v$ **then show** $?thesis$ **using** $InitDone\ initPD-def$
proof($cases\ Cs$) **qed**($auto$)

next

fix a **assume** $e': e' = throw\ a$ **then show** $?thesis$ **using** $InitDone\ initPD-def$
proof($cases\ Cs$) **qed**($auto$)

```

qed
next
case (InitProcessing sh C sfs C' Cs e h l e' s')
then have final e' using eval-final by simp
then show ?case
proof(rule finalE)
  fix v assume e': e' = Val v then show ?thesis using InitProcessing initPD-def
  proof(cases Cs) qed(auto)
next
  fix a assume e': e' = throw a then show ?thesis using InitProcessing initPD-def
  proof(cases Cs) qed(auto)
qed
next
case (InitError sh C sfs Cs e h l e' s' C') show ?case
proof(cases Cs)
  case Nil then show ?thesis using InitError by simp
next
  case (Cons C2 list)
  then have final e' using InitError eval-final by simp
  then show ?thesis
  proof(rule finalE)
    fix v assume e': e' = Val v then show ?thesis
    using InitError
    proof -
      obtain ccss :: char list list and bb :: bool where
        INIT C' (C # Cs, False) ← e = INIT C' (ccss, bb) ← unit
        using InitError.premis(2) by blast
      then show ?thesis using InitError.hyps(2) e' by(auto dest!: rinit-throwE)
    qed
  next
    fix a assume e': e' = throw a
    then show ?thesis using Cons InitError cons-to-append[of list] by clarsimp
  qed
qed
next
case (InitRInit C Cs h l sh e' s' C') show ?case
proof(cases Cs)
  case Nil then show ?thesis using InitRInit by simp
next
  case (Cons C' list) then show ?thesis
  using InitRInit Cons cons-to-append[of list] by clarsimp
qed
next
case (RInit e s v h' l' sh' C sfs i sh'' C' Cs e' e1 s1)
then have final: final e1 using eval-final by simp
then show ?case
proof(cases Cs)
  case Nil show ?thesis using final
  proof(rule finalE)
    fix v assume e': e1 = Val v show ?thesis
    using RInit Nil by(auto simp: fun-upd-same initPD-def)
  next
    fix a assume e': e1 = throw a show ?thesis
    using RInit Nil by(auto simp: fun-upd-same initPD-def)
  qed

```

```

qed
next
case (Cons a list) show ?thesis
proof(rule finalE[OF final])
  fix v assume e': e1 = Val v then show ?thesis
  using RInit Cons by clarsimp (metis last.simps last-appendR list.distinct(1))
next
fix a assume e': e1 = throw a then show ?thesis
using RInit Cons by clarsimp (metis last.simps last-appendR list.distinct(1))
qed
qed
next
case (RInitInitFail e s a h' l' sh' C sfs i sh'' D Cs e' e1 s1)
then have final: final e1 using eval-final by simp
then show ?case
proof(rule finalE)
  fix v assume e': e1 = Val v then show ?thesis
  using RInitInitFail by clarsimp (meson exp.distinct(101) rinit-throwE)
next
fix a' assume e': e1 = Throw a'
then have iconf (sh'(C ↦ (sfs, Error))) a
  using RInitInitFail.hyps(1) eval-final by fastforce
then show ?thesis using RInitInitFail e'
  by clarsimp (meson Cons-eq-append-conv list.inject)
qed
qed(auto simp: fun-upd-same)

```

lemma *init-Val-PD*: $P \vdash \langle \text{INIT } C' (Cs, b) \leftarrow \text{unit}, s \rangle \Rightarrow \langle \text{Val } v, s \rangle$
 $\Rightarrow \text{iconf } (\text{shp } s) (\text{INIT } C' (Cs, b) \leftarrow \text{unit})$
 $\Rightarrow \exists \text{ sfs } i. \text{shp } s' C' = \lfloor (\text{sfs}, i) \rfloor \wedge (i = \text{Done} \vee i = \text{Processing})$
by(*drule-tac* $v = v$ **in** *eval-init-return*, *simp+*)

lemma *init-throw-PD*: $P \vdash \langle \text{INIT } C' (Cs, b) \leftarrow \text{unit}, s \rangle \Rightarrow \langle \text{throw } a, s \rangle$
 $\Rightarrow \text{iconf } (\text{shp } s) (\text{INIT } C' (Cs, b) \leftarrow \text{unit})$
 $\Rightarrow \exists \text{ sfs } i. \text{shp } s' C' = \lfloor (\text{sfs}, \text{Error}) \rfloor$
by(*drule-tac* $a = a$ **in** *eval-init-return*, *simp+*)

lemma *rinit-Val-PD*: $P \vdash \langle \text{RI}(C, e_0); Cs \leftarrow \text{unit}, s \rangle \Rightarrow \langle \text{Val } v, s \rangle$
 $\Rightarrow \text{iconf } (\text{shp } s) (\text{RI}(C, e_0); Cs \leftarrow \text{unit}) \Rightarrow \text{last}(C \# Cs) = C'$
 $\Rightarrow \exists \text{ sfs } i. \text{shp } s' C' = \lfloor (\text{sfs}, i) \rfloor \wedge (i = \text{Done} \vee i = \text{Processing})$
by(*auto dest!*: *eval-init-return* [**where** $C' = C'$]
append-butlast-last-id[*THEN sym*])

lemma *rinit-throw-PD*: $P \vdash \langle \text{RI}(C, e_0); Cs \leftarrow \text{unit}, s \rangle \Rightarrow \langle \text{throw } a, s \rangle$
 $\Rightarrow \text{iconf } (\text{shp } s) (\text{RI}(C, e_0); Cs \leftarrow \text{unit}) \Rightarrow \text{last}(C \# Cs) = C'$
 $\Rightarrow \exists \text{ sfs } i. \text{shp } s' C' = \lfloor (\text{sfs}, \text{Error}) \rfloor$
by(*auto dest!*: *eval-init-return* [**where** $C' = C'$]
append-butlast-last-id[*THEN sym*])

— combining the above to get evaluation of *INIT* in a sequence

lemma *eval-init-seq'*: $P \vdash \langle e, s \rangle \Rightarrow \langle e', s \rangle$
 $\Rightarrow (\exists C Cs b e_i. e = \text{INIT } C (Cs, b) \leftarrow e_i) \vee (\exists C e_0 Cs e_i. e = \text{RI}(C, e_0); Cs \leftarrow e_i)$

$\implies (\exists C Cs b e_i. e = \text{INIT } C (Cs, b) \leftarrow e_i \wedge P \vdash \langle (\text{INIT } C (Cs, b) \leftarrow \text{unit});; e_i, s \rangle \Rightarrow \langle e', s' \rangle)$
 $\vee (\exists C e_0 Cs e_i. e = \text{RI}(C, e_0); Cs \leftarrow e_i \wedge P \vdash \langle (\text{RI}(C, e_0); Cs \leftarrow \text{unit});; e_i, s \rangle \Rightarrow \langle e', s' \rangle)$
and $P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \implies \text{True}$
proof(*induct rule: eval-evals.inducts*)
case *InitFinal* **then show** *?case* **by**(*auto simp: Seq[OF eval-evals.InitFinal[OF Val[where v=Unit]]]*)
next
case (*InitNone sh*) **then show** *?case*
using *seq-ext[OF eval-evals.InitNone[where sh=sh and e=unit, OF InitNone.hyps(1)]]* **by** *fastforce*
next
case (*InitDone sh*) **then show** *?case*
using *seq-ext[OF eval-evals.InitDone[where sh=sh and e=unit, OF InitDone.hyps(1)]]* **by** *fastforce*
next
case (*InitProcessing sh*) **then show** *?case*
using *seq-ext[OF eval-evals.InitProcessing[where sh=sh and e=unit, OF InitProcessing.hyps(1)]]*
by *fastforce*
next
case (*InitError sh*) **then show** *?case*
using *seq-ext[OF eval-evals.InitError[where sh=sh and e=unit, OF InitError.hyps(1)]]* **by** *fastforce*
next
case (*InitObject sh*) **then show** *?case*
using *seq-ext[OF eval-evals.InitObject[where sh=sh and e=unit, OF InitObject.hyps(1)]]*
by *fastforce*
next
case (*InitNonObject sh*) **then show** *?case*
using *seq-ext[OF eval-evals.InitNonObject[where sh=sh and e=unit, OF InitNonObject.hyps(1)]]*
by *fastforce*
next
case (*InitRInit C Cs e h l sh e' s' C'*) **then show** *?case*
using *seq-ext[OF eval-evals.InitRInit[where e=unit]]* **by** *fastforce*
next
case *RInit* **then show** *?case*
using *seq-ext[OF eval-evals.RInit[where e=unit, OF RInit.hyps(1)]]* **by** *fastforce*
next
case *RInitInitFail* **then show** *?case*
using *seq-ext[OF eval-evals.RInitInitFail[where e=unit, OF RInitInitFail.hyps(1)]]* **by** *fastforce*
next
case *RInitFailFinal*
then show *?case* **using** *eval-evals.RInitFailFinal eval-evals.SeqThrow* **by** *auto*
qed(*auto*)

lemma *eval-init-seq*: $P \vdash \langle \text{INIT } C (Cs, b) \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle$
 $\implies P \vdash \langle (\text{INIT } C (Cs, b) \leftarrow \text{unit});; e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle$
by(*auto dest: eval-init-seq'*)

The key lemma:

lemma

assumes *wf: wwf-J-prog P*

shows *extend-1-eval*: $P \vdash \langle e, s, b \rangle \rightarrow \langle e'', s'', b'' \rangle \implies P, \text{shp } s \vdash_b (e, b) \checkmark$

$\implies (\bigwedge s' e'. P \vdash \langle e'', s'' \rangle \Rightarrow \langle e', s' \rangle \implies P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle)$

and *extend-1-evals*: $P \vdash \langle es, s, b \rangle [\rightarrow] \langle es'', s'', b'' \rangle \implies P, \text{shp } s \vdash_b (es, b) \checkmark$

$\implies (\bigwedge s' es'. P \vdash \langle es'', s'' \rangle [\Rightarrow] \langle es', s' \rangle \implies P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle)$

proof (*induct rule: red-reds.inducts*)

case (*RedNew h a C FDTs h' l sh*)

```

then have  $e':e' = \text{addr } a$  and  $s':s' = (h(a \mapsto \text{blank } P \ C), l, sh)$ 
  using eval-cases(3) by fastforce+
obtain  $sfs \ i$  where  $shC: sh \ C = [(sfs, i)]$  and  $i = \text{Done} \vee i = \text{Processing}$ 
  using RedNew by (clarsimp simp: bconf-def initPD-def)
then show ?case
proof(cases i)
  case Done then show ?thesis using RedNew shC e' s' New by simp
next
  case Processing
  then have  $shC': \nexists sfs. sh \ C = \text{Some}(sfs, \text{Done})$  and  $shP: sh \ C = \text{Some}(sfs, \text{Processing})$ 
    using shC by simp+
  then have  $init: P \vdash \langle \text{INIT } C \ ([C], \text{False}) \leftarrow unit, (h, l, sh) \rangle \Rightarrow \langle unit, (h, l, sh) \rangle$ 
    by(simp add: InitFinal InitProcessing Val)
  have  $P \vdash \langle \text{new } C, (h, l, sh) \rangle \Rightarrow \langle \text{addr } a, (h(a \mapsto \text{blank } P \ C), l, sh) \rangle$ 
    using RedNew shC' by(auto intro: NewInit[OF - init])
  then show ?thesis using  $e' \ s'$  by simp
qed(auto)
next
  case (RedNewFail h C l sh)
  then have  $e':e' = \text{THROW OutOfMemory}$  and  $s':s' = (h, l, sh)$ 
    using eval-final-same final-def by fastforce+
  obtain  $sfs \ i$  where  $shC: sh \ C = [(sfs, i)]$  and  $i = \text{Done} \vee i = \text{Processing}$ 
    using RedNewFail by (clarsimp simp: bconf-def initPD-def)
  then show ?case
  proof(cases i)
    case Done then show ?thesis using RedNewFail shC e' s' NewFail by simp
  next
    case Processing
    then have  $shC': \nexists sfs. sh \ C = \text{Some}(sfs, \text{Done})$  and  $shP: sh \ C = \text{Some}(sfs, \text{Processing})$ 
      using shC by simp+
    then have  $init: P \vdash \langle \text{INIT } C \ ([C], \text{False}) \leftarrow unit, (h, l, sh) \rangle \Rightarrow \langle unit, (h, l, sh) \rangle$ 
      by(simp add: InitFinal InitProcessing Val)
    have  $P \vdash \langle \text{new } C, (h, l, sh) \rangle \Rightarrow \langle \text{THROW OutOfMemory}, (h, l, sh) \rangle$ 
      using RedNewFail shC' by(auto intro: NewInitOOM[OF - init])
    then show ?thesis using  $e' \ s'$  by simp
    qed(auto)
  next
  case (NewInitRed sh C h l)
  then have  $seq: P \vdash \langle (\text{INIT } C \ ([C], \text{False}) \leftarrow unit);; \text{new } C, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle$ 
    using eval-init-seq by simp
  then show ?case
  proof(rule eval-cases(14)) — Seq
    fix  $v \ s_1$  assume  $init: P \vdash \langle \text{INIT } C \ ([C], \text{False}) \leftarrow unit, (h, l, sh) \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle$ 
      and  $new: P \vdash \langle \text{new } C, s_1 \rangle \Rightarrow \langle e', s' \rangle$ 
    obtain  $h_1 \ l_1 \ sh_1$  where  $s_1: s_1 = (h_1, l_1, sh_1)$  by(cases s_1)
    then obtain  $sfs \ i$  where  $shC: sh_1 \ C = [(sfs, i)]$  and  $iDP: i = \text{Done} \vee i = \text{Processing}$ 
      using init-Val-PD[OF init] by auto
    show ?thesis
  proof(rule eval-cases(1)[OF new]) — New
    fix  $sha \ ha \ a \ FDTs \ la$ 
    assume  $s_1 a: s_1 = (ha, la, sha)$  and  $e': e' = \text{addr } a$ 
      and  $s': s' = (ha(a \mapsto \text{blank } P \ C), la, sha)$ 
      and  $addr: \text{new-Addr } ha = [a]$  and  $fields: P \vdash C \text{ has-fields } FDTs$ 
    then show ?thesis using NewInit[OF - - addr fields] NewInitRed.hyps init by simp

```

```

next
  fix sha ha la
  assume s1 = (ha, la, sha) and e' = THROW OutOfMemory
  and s' = (ha, la, sha) and new-Addr ha = None
  then show ?thesis using NewInitOOM NewInitRed.hyps init by simp
next
  fix sha ha la v' h' l' sh' a FDTs
  assume s1a: s1 = (ha, la, sha) and e': e' = addr a
  and s': s' = (h'(a ↦ blank P C), l', sh')
  and shaC: ∀ sfs. sha C ≠ [(sfs, Done)]
  and init': P ⊢ ⟨INIT C ([C], False) ← unit, (ha, la, sha)⟩ ⇒ ⟨Val v', (h', l', sh')⟩
  and addr: new-Addr h' = [a] and fields: P ⊢ C has-fields FDTs
  then have i: i = Processing using iDP shC s1 by simp
  then have (h', l', sh') = (ha, la, sha) using init' init-ProcessingE s1 s1a shC by blast
  then show ?thesis using NewInit NewInitRed.hyps s1a addr fields init shaC e' s' by auto
next
  fix sha ha la v' h' l' sh'
  assume s1a: s1 = (ha, la, sha) and e': e' = THROW OutOfMemory
  and s': s' = (h', l', sh') and ∀ sfs. sha C ≠ [(sfs, Done)]
  and init': P ⊢ ⟨INIT C ([C], False) ← unit, (ha, la, sha)⟩ ⇒ ⟨Val v', (h', l', sh')⟩
  and addr: new-Addr h' = None
  then have i: i = Processing using iDP shC s1 by simp
  then have (h', l', sh') = (ha, la, sha) using init' init-ProcessingE s1 s1a shC by blast
  then show ?thesis
  using NewInitOOM NewInitRed.hyps e' addr s' s1a init by auto
next
  fix sha ha la a
  assume s1a: s1 = (ha, la, sha)
  and ∀ sfs. sha C ≠ [(sfs, Done)]
  and init': P ⊢ ⟨INIT C ([C], False) ← unit, (ha, la, sha)⟩ ⇒ ⟨throw a, s'⟩
  then have i: i = Processing using iDP shC s1 by simp
  then show ?thesis using init' init-ProcessingE s1 s1a shC by blast
qed
next
  fix e assume e': e' = throw e
  and init: P ⊢ ⟨INIT C ([C], False) ← unit, (h, l, sh)⟩ ⇒ ⟨throw e, s'⟩
  obtain h' l' sh' where s': s' = (h', l', sh') by (cases s')
  then obtain sfs i where shC: sh' C = [(sfs, i)] and iDP: i = Error
  using init-throw-PD[OF init] by auto
  then show ?thesis by (simp add: NewInitRed.hyps NewInitThrow e' init)
qed
next
  case CastRed then show ?case
  by (fastforce elim!: eval-cases intro: eval-vals.intros intro!: CastFail)
next
  case RedCastNull
  then show ?case
  by simp (iprover elim: eval-cases intro: eval-vals.intros)
next
  case RedCast
  then show ?case
  by (auto elim: eval-cases intro: eval-vals.intros)
next
  case RedCastFail

```

```

then show ?case
  by (auto elim!: eval-cases intro: eval-evals.intros)
next
  case BinOpRed1 then show ?case
    by(fastforce elim!: eval-cases intro: eval-evals.intros simp: val-no-step)
next
  case BinOpRed2
  thus ?case
    by (fastforce elim!: eval-cases intro: eval-evals.intros eval-finalId)
next
  case RedBinOp
  thus ?case
    by simp (iprover elim: eval-cases intro: eval-evals.intros)
next
  case RedVar
  thus ?case
    by (fastforce elim: eval-cases intro: eval-evals.intros)
next
  case LAssRed thus ?case
    by(fastforce elim: eval-cases intro: eval-evals.intros)
next
  case RedLAss
  thus ?case
    by (fastforce elim: eval-cases intro: eval-evals.intros)
next
  case FAccRed thus ?case
    by(fastforce elim: eval-cases intro: eval-evals.intros)
next
  case RedFAcc then show ?case
    by (fastforce elim: eval-cases intro: eval-evals.intros)
next
  case RedFAccNull then show ?case
    by (fastforce elim!: eval-cases intro: eval-evals.intros)
next
  case RedFAccNone thus ?case
    by(fastforce elim: eval-cases intro: eval-evals.intros)
next
  case RedFAccStatic thus ?case
    by(fastforce elim: eval-cases intro: eval-evals.intros)
next
  case (RedSFAcc C F t D sh sfs i v h l)
  then have  $e':e' = \text{Val } v$  and  $s':s' = (h, l, sh)$ 
    using eval-cases( $\beta$ ) by fastforce+
  have  $i = \text{Done} \vee i = \text{Processing}$  using RedSFAcc by (clarsimp simp: bconf-def initPD-def)
  then show ?case
  proof(cases i)
    case Done then show ?thesis using RedSFAcc  $e' s' \text{SFAcc}$  by simp
  next
    case Processing
  then have  $shC': \nexists sfs. sh D = \text{Some}(sfs, \text{Done})$  and  $shP: sh D = \text{Some}(sfs, \text{Processing})$ 
    using RedSFAcc by simp+
  then have  $\text{init}: P \vdash \langle \text{INIT } D ([D], \text{False}) \leftarrow \text{unit}, (h, l, sh) \rangle \Rightarrow \langle \text{unit}, (h, l, sh) \rangle$ 
    by(simp add: InitFinal InitProcessing Val)
  have  $P \vdash \langle C \cdot_s F \{D\}, (h, l, sh) \rangle \Rightarrow \langle \text{Val } v, (h, l, sh) \rangle$ 

```

```

    by(rule SFAccInit[OF RedSFAcc.hyps(1) shC' init shP RedSFAcc.hyps(3)])
    then show ?thesis using e' s' by simp
qed(auto)
next
case (SFAccInitRed C F t D sh h l)
then have seq:  $P \vdash \langle (INIT\ D\ ([D],False) \leftarrow unit); C \cdot_s F\{D\},(h, l, sh) \rangle \Rightarrow \langle e',s' \rangle$ 
  using eval-init-seq by simp
then show ?case
proof(rule eval-cases(14)) — Seq
  fix v s1 assume init:  $P \vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h, l, sh) \rangle \Rightarrow \langle Val\ v,s_1 \rangle$ 
    and acc:  $P \vdash \langle C \cdot_s F\{D\},s_1 \rangle \Rightarrow \langle e',s' \rangle$ 
  obtain h1 l1 sh1 where s1: s1 = (h1,l1,sh1) by(cases s1)
  then obtain sfs i where shD: sh1 D = [(sfs, i)] and iDP: i = Done  $\vee$  i = Processing
    using init-Val-PD[OF init] by auto
  show ?thesis
proof(rule eval-cases(8)[OF acc]) — SFAcc
  fix t sha sfs v ha la
  assume s1 = (ha, la, sha) and e' = Val v
    and s' = (ha, la, sha) and P  $\vdash$  C has F,Static:t in D
    and sha D = [(sfs, Done)] and sfs F = [v]
  then show ?thesis using SFAccInit SFAccInitRed.hyps(2) init by auto
next
  fix t sha ha la v' h' l' sh' sfs i' v
  assume s1a: s1 = (ha, la, sha) and e': e' = Val v
    and s': s' = (h', l', sh') and field: P  $\vdash$  C has F,Static:t in D
    and  $\forall$  sfs. sha D  $\neq$  [(sfs, Done)]
    and init': P  $\vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(ha, la, sha) \rangle \Rightarrow \langle Val\ v',(h', l', sh') \rangle$ 
    and shD': sh' D = [(sfs, i')] and sfsF: sfs F = [v]
  then have i: i = Processing using iDP shD s1 by simp
  then have (h', l', sh') = (ha, la, sha) using init' init-ProcessingE s1 s1a shD by blast
  then show ?thesis using SFAccInit SFAccInitRed.hyps(2) e' s' field init s1a sfsF shD' by auto
next
  fix t sha ha la a
  assume s1a: s1 = (ha, la, sha) and e' = throw a
    and P  $\vdash$  C has F,Static:t in D and  $\forall$  sfs. sha D  $\neq$  [(sfs, Done)]
    and init': P  $\vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(ha, la, sha) \rangle \Rightarrow \langle throw\ a,s' \rangle$ 
  then have i: i = Processing using iDP shD s1 by simp
  then show ?thesis using init' init-ProcessingE s1 s1a shD by blast
next
  assume  $\forall b\ t. \neg P \vdash C\ has\ F,b;t\ in\ D$ 
  then show ?thesis using SFAccInitRed.hyps(1) by blast
next
  fix t assume field: P  $\vdash$  C has F,NonStatic:t in D
  then show ?thesis using has-field-fun[OF SFAccInitRed.hyps(1) field] by simp
qed
next
fix e assume e': e' = throw e
  and init: P  $\vdash \langle INIT\ D\ ([D],False) \leftarrow unit,(h, l, sh) \rangle \Rightarrow \langle throw\ e,s' \rangle$ 
  obtain h' l' sh' where s': s' = (h',l',sh') by(cases s')
  then obtain sfs i where shC: sh' D = [(sfs, i)] and iDP: i = Error
    using init-throw-PD[OF init] by auto
  then show ?thesis
    using SFAccInitRed.hyps(1) SFAccInitRed.hyps(2) SFAccInitThrow e' init by auto
qed

```

```

next
  case RedSFAccNone thus ?case
    by(fastforce elim: eval-cases intro: eval-vals.intros)
next
  case RedSFAccNonStatic thus ?case
    by(fastforce elim: eval-cases intro: eval-vals.intros)
next
  case (FAssRed1 e s b e1 s1 b1 F D e2)
  obtain h' l' sh' where s'=(h',l',sh') by(cases s')
  with FAssRed1 show ?case
    by(fastforce elim!: eval-cases(9)[where e1=e1] intro: eval-vals.intros simp: val-no-step
      intro!: FAss FAssNull FAssNone FAssStatic FAssThrow2)
next
  case FAssRed2
  obtain h' l' sh' where s'=(h',l',sh') by(cases s')
  with FAssRed2 show ?case
    by(auto elim!: eval-cases intro: eval-vals.intros
      intro!: FAss FAssNull FAssNone FAssStatic FAssThrow2 Val)
next
  case RedFAss
  thus ?case
    by (fastforce elim!: eval-cases intro: eval-vals.intros)
next
  case RedFAssNull
  thus ?case
    by (fastforce elim!: eval-cases intro: eval-vals.intros)
next
  case RedFAssNone
  then show ?case
    by(auto elim!: eval-cases intro: eval-vals.intros eval-finalId)
next
  case RedFAssStatic
  then show ?case
    by(auto elim!: eval-cases intro: eval-vals.intros eval-finalId)
next
  case (SFAssRed e s b e'' s'' b'' C F D)
  obtain h l sh where [simp]: s = (h,l,sh) by(cases s)
  obtain h' l' sh' where [simp]: s'=(h',l',sh') by(cases s')
  have val-of e = None using val-no-step SFAssRed.hyps(1) by(meson option.exhaust)
  then have bconf: P,sh  $\vdash_b$  (e,b)  $\checkmark$  using SFAssRed by simp
  show ?case using SFAssRed.prem(2) SFAssRed bconf
  proof cases
    case 2 with SFAssRed bconf show ?thesis by(auto intro!: SFAssInit)
  next
    case 3 with SFAssRed bconf show ?thesis by(auto intro!: SFAssInitThrow)
  qed(auto intro: eval-vals.intros intro!: SFAss SFAssInit SFAssNone SFAssNonStatic)
next
  case (RedSFAss C F t D sh sfs i sfs' v sh' h l)
  let ?sfs' = sfs(F  $\mapsto$  v)
  have e':e' = unit and s':s' = (h, l, sh(D  $\mapsto$  (?sfs', i)))
    using RedSFAss eval-cases(3) by fastforce+
  have i = Done  $\vee$  i = Processing using RedSFAss by(clarsimp simp: bconf-def initPD-def)
  then show ?case
  proof(cases i)

```

case Done then show ?thesis using RedSFAss e' s' SFAss Val by auto
next
case Processing
then have shC': \nexists sfs. sh D = Some(sfs,Done) and shP: sh D = Some(sfs,Processing)
using RedSFAss by simp+
then have init: $P \vdash \langle \text{INIT } D ([D], \text{False}) \leftarrow \text{unit}, (h, l, sh) \rangle \Rightarrow \langle \text{unit}, (h, l, sh) \rangle$
by(simp add: InitFinal InitProcessing Val)
have $P \vdash \langle C \cdot_s F \{D\} := \text{Val } v, (h, l, sh) \rangle \Rightarrow \langle \text{unit}, (h, l, sh(D \mapsto (?sfs', i))) \rangle$
using Processing by(auto intro: SFAssInit[OF Val RedSFAss.hyps(1) shC' init shP])
then show ?thesis using e' s' by simp
qed(auto)
next
case (SFAssInitRed C F t D sh v h l)
then have seq: $P \vdash \langle (\text{INIT } D ([D], \text{False}) \leftarrow \text{unit}); C \cdot_s F \{D\} := \text{Val } v, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle$
using eval-init-seq by simp
then show ?case
proof(rule eval-cases(14)) — Seq
fix v' s₁ assume init: $P \vdash \langle \text{INIT } D ([D], \text{False}) \leftarrow \text{unit}, (h, l, sh) \rangle \Rightarrow \langle \text{Val } v', s_1 \rangle$
and acc: $P \vdash \langle C \cdot_s F \{D\} := \text{Val } v, s_1 \rangle \Rightarrow \langle e', s' \rangle$
obtain h₁ l₁ sh₁ where s₁: s₁ = (h₁, l₁, sh₁) by(cases s₁)
then obtain sfs i where shD: sh₁ D = [(sfs, i)] and iDP: i = Done \vee i = Processing
using init-Val-PD[OF init] by auto
show ?thesis
proof(rule eval-cases(10)[OF acc]) — SFAss
fix va h₁ l₁ sh₁ t sfs
assume e': e' = unit and s': s' = (h₁, l₁, sh₁(D \mapsto (sfs(F \mapsto va), Done)))
and val: $P \vdash \langle \text{Val } v, s_1 \rangle \Rightarrow \langle \text{Val } va, (h_1, l_1, sh_1) \rangle$
and field: $P \vdash C \text{ has } F, \text{Static}:t \text{ in } D$ and shD': sh₁ D = [(sfs, Done)]
have v = va and s₁ = (h₁, l₁, sh₁) using eval-final-same[OF val] by auto
then show ?thesis using SFAssInit field SFAssInitRed.hyps(2) shD' e' s' init val
by (metis eval-final eval-finalId)
next
fix va h₁ l₁ sh₁ t v' h' l' sh' sfs i'
assume e': e' = unit and s': s' = (h', l', sh'(D \mapsto (sfs(F \mapsto va), i')))
and val: $P \vdash \langle \text{Val } v, s_1 \rangle \Rightarrow \langle \text{Val } va, (h_1, l_1, sh_1) \rangle$
and field: $P \vdash C \text{ has } F, \text{Static}:t \text{ in } D$ and nDone: \forall sfs. sh₁ D \neq [(sfs, Done)]
and init': $P \vdash \langle \text{INIT } D ([D], \text{False}) \leftarrow \text{unit}, (h_1, l_1, sh_1) \rangle \Rightarrow \langle \text{Val } v', (h', l', sh') \rangle$
and shD': sh' D = [(sfs, i')]
have v: v = va and s₁a: s₁ = (h₁, l₁, sh₁) using eval-final-same[OF val] by auto
then have i: i = Processing using iDP shD s₁ nDone by simp
then have (h₁, l₁, sh₁) = (h', l', sh') using init' init-ProcessingE s₁ s₁a shD by blast
then show ?thesis using SFAssInit SFAssInitRed.hyps(2) e' s' field init v s₁a shD' val
by (metis eval-final eval-finalId)
next
fix va h₁ l₁ sh₁ t a
assume e' = throw a and val: $P \vdash \langle \text{Val } v, s_1 \rangle \Rightarrow \langle \text{Val } va, (h_1, l_1, sh_1) \rangle$
and $P \vdash C \text{ has } F, \text{Static}:t \text{ in } D$ and nDone: \forall sfs. sh₁ D \neq [(sfs, Done)]
and init': $P \vdash \langle \text{INIT } D ([D], \text{False}) \leftarrow \text{unit}, (h_1, l_1, sh_1) \rangle \Rightarrow \langle \text{throw } a, s' \rangle$
have v: v = va and s₁a: s₁ = (h₁, l₁, sh₁) using eval-final-same[OF val] by auto
then have i: i = Processing using iDP shD s₁ nDone by simp
then have (h₁, l₁, sh₁) = s' using init' init-ProcessingE s₁ s₁a shD by blast
then show ?thesis using init' init-ProcessingE s₁ s₁a shD i by blast
next
fix e'' assume val: $P \vdash \langle \text{Val } v, s_1 \rangle \Rightarrow \langle \text{throw } e'', s' \rangle$

```

    then show ?thesis using eval-final-same[OF val] by simp
  next
    assume  $\forall b t. \neg P \vdash C \text{ has } F, b:t \text{ in } D$ 
    then show ?thesis using SFAssInitRed.hyps(1) by blast
  next
    fix t assume field:  $P \vdash C \text{ has } F, \text{NonStatic}:t \text{ in } D$ 
    then show ?thesis using has-field-fun[OF SFAssInitRed.hyps(1) field] by simp
  qed
next
  fix e assume e':  $e' = \text{throw } e$ 
  and init:  $P \vdash \langle \text{INIT } D ([D], \text{False}) \leftarrow \text{unit}, (h, l, sh) \rangle \Rightarrow \langle \text{throw } e, s' \rangle$ 
  obtain h' l' sh' where s':  $s' = (h', l', sh')$  by (cases s')
  then obtain sfs i where shC:  $sh' D = [(sfs, i)]$  and iDP:  $i = \text{Error}$ 
  using init-throw-PD[OF init] by auto
  then show ?thesis using SFAssInitRed.hyps(1) SFAssInitRed.hyps(2) SFAssInitThrow Val
  by (metis e' init)
  qed
next
  case (RedSFAssNone C F D v s b) then show ?case
  by (cases s) (auto elim!: eval-cases intro: eval-vals.intros eval-finalId)
next
  case (RedSFAssNonStatic C F t D v s b) then show ?case
  by (cases s) (auto elim!: eval-cases intro: eval-vals.intros eval-finalId)
next
  case CallObj
  note val-no-step[simp]
  from CallObj.prem(2) CallObj show ?case
  proof cases
    case 2 with CallObj show ?thesis by (fastforce intro!: CallParamsThrow)
  next
    case 3 with CallObj show ?thesis by (fastforce intro!: CallNull)
  next
    case 4 with CallObj show ?thesis by (fastforce intro!: CallNone)
  next
    case 5 with CallObj show ?thesis by (fastforce intro!: CallStatic)
  qed (fastforce intro!: CallObjThrow Call)+
next
  case (CallParams es s b es'' s'' b'' v M s')
  then obtain h' l' sh' where s' = (h', l', sh') by (cases s')
  with CallParams show ?case
  by (auto elim!: eval-cases intro!: CallNone eval-finalId CallStatic Val)
  (auto intro!: CallParamsThrow CallNull Call Val)
next
  case (RedCall h a C fs M Ts T pns body D vs l sh b)
  have  $P \vdash \langle \text{addr } a, (h, l, sh) \rangle \Rightarrow \langle \text{addr } a, (h, l, sh) \rangle$  by (rule eval-vals.intros)
  moreover
  have finals:  $\text{finals}(\text{map } \text{Val } \text{vs})$  by simp
  with finals have  $P \vdash \langle \text{map } \text{Val } \text{vs}, (h, l, sh) \rangle [\Rightarrow] \langle \text{map } \text{Val } \text{vs}, (h, l, sh) \rangle$ 
  by (iprover intro: eval-finalsId)
  moreover have  $h a = \text{Some } (C, fs)$  using RedCall by simp
  moreover have method:  $P \vdash C \text{ sees } M, \text{NonStatic}: Ts \rightarrow T = (pns, \text{body}) \text{ in } D$  by fact
  moreover have same-len1:  $\text{length } Ts = \text{length } pns$ 
  and this-distinct:  $\text{this} \notin \text{set } pns$  and fv:  $\text{fv } (\text{body}) \subseteq \{\text{this}\} \cup \text{set } pns$ 
  using method wf by (fastforce dest!: sees-wf-mdecl simp: wf-mdecl-def)+

```

have same-len: $\text{length } vs = \text{length } pns$ **by fact**
moreover
obtain l_2' **where** $l_2': l_2' = [\text{this} \mapsto \text{Addr } a, pns \mapsto] vs$ **by simp**
moreover
obtain $h_3 \ l_3 \ sh_3$ **where** $s': s' = (h_3, l_3, sh_3)$ **by (cases s')**
have eval-blocks:
 $P \vdash \langle (\text{blocks } (\text{this} \# pns, \text{Class } D \# Ts, \text{Addr } a \# vs, \text{body})), (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle$ **by fact**
hence id: $l_3 = l$ **using** $fv \ s' \ \text{same-len}_1 \ \text{same-len}$
by (fastforce elim: eval-closed-lcl-unchanged)
from eval-blocks obtain l_3' **where** $P \vdash \langle \text{body}, (h, l_2', sh) \rangle \Rightarrow \langle e', (h_3, l_3', sh_3) \rangle$
proof –
from same-len₁ have $\text{length}(\text{this} \# pns) = \text{length}(\text{Class } D \# Ts)$ **by simp**
moreover from same-len₁ same-len
have same-len₂: $\text{length } (\text{this} \# pns) = \text{length } (\text{Addr } a \# vs)$ **by simp**
moreover from eval-blocks
have $P \vdash \langle \text{blocks } (\text{this} \# pns, \text{Class } D \# Ts, \text{Addr } a \# vs, \text{body}), (h, l, sh) \rangle$
 $\Rightarrow \langle e', (h_3, l_3, sh_3) \rangle$ **using** $s' \ \text{same-len}_1 \ \text{same-len}_2$ **by simp**
ultimately obtain l''
where $P \vdash \langle \text{body}, (h, l(\text{this} \# pns \mapsto \text{Addr } a \# vs), sh) \rangle \Rightarrow \langle e', (h_3, l'', sh_3) \rangle$
by (blast dest: blocksEval)
from eval-restrict-lcl [OF wf this fv] this-distinct same-len₁ same-len
have $P \vdash \langle \text{body}, (h, [\text{this} \# pns \mapsto] \text{Addr } a \# vs), sh \rangle \Rightarrow$
 $\langle e', (h_3, l'' \uparrow (\text{set } (\text{this} \# pns)), sh_3) \rangle$ **using** $wf \ \text{method}$
by (simp add: subset-insert-iff insert-Diff-if)
thus ?thesis by (fastforce intro!: that simp add: l₂')
qed
ultimately
have $P \vdash \langle (\text{addr } a) \cdot M(\text{map } Val \ vs), (h, l, sh) \rangle \Rightarrow \langle e', (h_3, l, sh_3) \rangle$ **by (rule Call)**
with s' id show ?case by simp
next
case RedCallNull
thus ?case
by (fastforce elim: eval-cases intro: eval-vals.intros eval-finalsId)
next
case (RedCallNone h a C fs M vs l sh b)
then have $tes: \text{THROW } \text{NoSuchMethodError} = e' \wedge (h, l, sh) = s'$
using eval-final-same by simp
have $P \vdash \langle \text{addr } a, (h, l, sh) \rangle \Rightarrow \langle \text{addr } a, (h, l, sh) \rangle$ **and** $P \vdash \langle \text{map } Val \ vs, (h, l, sh) \rangle [\Rightarrow] \langle \text{map } Val \ vs, (h, l, sh) \rangle$
using eval-finalId eval-finalsId by auto
then show ?case using RedCallNone CallNone tes by auto
next
case (RedCallStatic h a C fs M Ts T m D vs l sh b)
then have $tes: \text{THROW } \text{IncompatibleClassChangeError} = e' \wedge (h, l, sh) = s'$
using eval-final-same by simp
have $P \vdash \langle \text{addr } a, (h, l, sh) \rangle \Rightarrow \langle \text{addr } a, (h, l, sh) \rangle$ **and** $P \vdash \langle \text{map } Val \ vs, (h, l, sh) \rangle [\Rightarrow] \langle \text{map } Val \ vs, (h, l, sh) \rangle$
using eval-finalId eval-finalsId by auto
then show ?case using RedCallStatic CallStatic tes by fastforce
next
case (SCallParams es s b es'' s'' b' C M s')
obtain $h' \ l' \ sh'$ **where** $s'[\text{simp}]: s' = (h', l', sh')$ **by (cases s')**
obtain $h \ l \ sh$ **where** $s[\text{simp}]: s = (h, l, sh)$ **by (cases s)**
have $es: \text{map-vals-of } es = \text{None}$ **using** $\text{vals-no-step } \text{SCallParams.hyps}(1)$ **by (meson not-Some-eq)**

```

have bconf:  $P, sh \vdash_b (es, b) \surd$  using s SCallParams.prems(1) by (simp add: bconf-SCall[OF es])
from SCallParams.prems(2) SCallParams bconf show ?case
proof cases
  case 2 with SCallParams bconf show ?thesis by(auto intro!: SCallNone)
next
  case 3 with SCallParams bconf show ?thesis by(auto intro!: SCallNonStatic)
next
  case 4 with SCallParams bconf show ?thesis by(auto intro!: SCallInitThrow)
next
  case 5 with SCallParams bconf show ?thesis by(auto intro!: SCallInit)
qed(auto intro!: SCallParamsThrow SCall)
next
case (RedSCall C M Ts T pns body D vs s)
then obtain h l sh where  $s:s = (h, l, sh)$  by(cases s)
then obtain sfs i where  $shC: sh D = [(sfs, i)]$  and  $i = Done \vee i = Processing$ 
  using RedSCall by(auto simp: bconf-def initPD-def dest: sees-method-fun)
have fnals:  $fnals(\text{map Val } vs)$  by simp
with fnals have mVs:  $P \vdash \langle \text{map Val } vs, (h, l, sh) \rangle [\Rightarrow] \langle \text{map Val } vs, (h, l, sh) \rangle$ 
  by (iprover intro: eval-fnalsId)
obtain sfs i where  $shC: sh D = [(sfs, i)]$ 
  using RedSCall s by(auto simp: bconf-def initPD-def dest: sees-method-fun)
then have iDP:  $i = Done \vee i = Processing$  using RedSCall s
  by (auto simp: bconf-def initPD-def dest: sees-method-fun[OF RedSCall.hyps(1)])
have method:  $P \vdash C \text{ sees } M, \text{Static}: Ts \rightarrow T = (pns, \text{body}) \text{ in } D$  by fact
have same-len1:  $\text{length } Ts = \text{length } pns$  and  $fv(\text{body}) \subseteq \text{set } pns$ 
  using method wf by (fastforce dest!: sees-wf-mdecl simp: wf-mdecl-def) +
have same-len:  $\text{length } vs = \text{length } pns$  by fact
obtain  $l_2'$  where  $l_2': l_2' = [pns \mapsto vs]$  by simp
obtain  $h_3 l_3 sh_3$  where  $s': s' = (h_3, l_3, sh_3)$  by (cases s')
have eval-blocks:
   $P \vdash \langle (\text{blocks } (pns, Ts, vs, \text{body})), (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle$  using RedSCall.prems(2) s by simp
hence id:  $l_3 = l$  using fv s' same-len1 same-len
  by(fastforce elim: eval-closed-lcl-unchanged)
from eval-blocks obtain  $l_3'$  where  $\text{body}: P \vdash \langle \text{body}, (h, l_2', sh) \rangle \Rightarrow \langle e', (h_3, l_3', sh_3) \rangle$ 
proof –
  from eval-blocks
  have  $P \vdash \langle \text{blocks } (pns, Ts, vs, \text{body})), (h, l, sh) \rangle$ 
     $\Rightarrow \langle e', (h_3, l_3, sh_3) \rangle$  using  $s'$  same-len same-len1 by simp
  then obtain  $l''$ 
    where  $P \vdash \langle \text{body}, (h, l(pns \mapsto vs), sh) \rangle \Rightarrow \langle e', (h_3, l'', sh_3) \rangle$ 
    by (blast dest: blocksEval[OF same-len1[THEN sym] same-len[THEN sym]])
  from eval-restrict-lcl[OF wf this fv] same-len1 same-len
  have  $P \vdash \langle \text{body}, (h, [pns \mapsto vs], sh) \rangle \Rightarrow \langle e', (h_3, l'' | (\text{set } pns), sh_3) \rangle$  using wf method
    by(simp add: subset-insert-iff insert-Diff-if)
  thus ?thesis by(fastforce intro!: that simp add:  $l_2'$ )
qed
show ?case using iDP
proof(cases i)
  case Done
  then have  $shC'$ :  $sh D = [(sfs, Done)] \vee M = \text{clinit} \wedge sh D = [(sfs, Processing)]$ 
    using  $shC$  by simp
  have  $P \vdash \langle C \cdot_s M(\text{map Val } vs), (h, l, sh) \rangle \Rightarrow \langle e', (h_3, l, sh_3) \rangle$ 
    by (rule SCall[OF mVs method shC' same-len  $l_2'$  body])
  with  $s s'$  id show ?thesis by simp

```

```

next
  case Processing
  then have shC':  $\nexists$  sfs. sh D = Some(sfs,Done) and shP: sh D = Some(sfs,Processing)
    using shC by simp+
  then have init:  $P \vdash \langle \text{INIT } D ([D], \text{False}) \leftarrow \text{unit}, (h, l, sh) \rangle \Rightarrow \langle \text{unit}, (h, l, sh) \rangle$ 
    by (simp add: InitFinal InitProcessing Val)
  have P  $\vdash \langle C \cdot_s M(\text{map Val vs}), (h, l, sh) \rangle \Rightarrow \langle e', (h_3, l, sh_3) \rangle$ 
  proof (cases M = clinit)
    case False show ?thesis by (rule SCallInit[OF mVs method shC' False init same-len l2' body])
  next
    case True
    then have shC': sh D = [(sfs, Done)]  $\vee$  M = clinit  $\wedge$  sh D = [(sfs, Processing)]
      using shC Processing by simp
    have P  $\vdash \langle C \cdot_s M(\text{map Val vs}), (h, l, sh) \rangle \Rightarrow \langle e', (h_3, l, sh_3) \rangle$ 
      by (rule SCall[OF mVs method shC' same-len l2' body])
    with s s' id show ?thesis by simp
  qed
  with s s' id show ?thesis by simp
qed(auto)
next
case (SCallInitRed C M Ts T pns body D sh vs h l)
then have seq:  $P \vdash \langle (\text{INIT } D ([D], \text{False}) \leftarrow \text{unit}); C \cdot_s M(\text{map Val vs}), (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle$ 
  using eval-init-seq by simp
then show ?case
proof (rule eval-cases(14)) — Seq
  fix v' s1 assume init:  $P \vdash \langle \text{INIT } D ([D], \text{False}) \leftarrow \text{unit}, (h, l, sh) \rangle \Rightarrow \langle \text{Val } v', s_1 \rangle$ 
    and call:  $P \vdash \langle C \cdot_s M(\text{map Val vs}), s_1 \rangle \Rightarrow \langle e', s' \rangle$ 
  obtain h1 l1 sh1 where s1:  $s_1 = (h_1, l_1, sh_1)$  by (cases s1)
  then obtain sfs i where shD:  $sh_1 D = [(sfs, i)]$  and iDP:  $i = \text{Done} \vee i = \text{Processing}$ 
    using init-Val-PD[OF init] by auto
  show ?thesis
proof (rule eval-cases(12)[OF call]) — SCall
  fix vsa ex es' assume  $P \vdash \langle \text{map Val vs}, s_1 \rangle [\Rightarrow] \langle \text{map Val vsa } @ \text{ throw ex } \# \text{ es}', s' \rangle$ 
  then show ?thesis using evals-finals-same by (meson finals-def map-Val-nthrow-neq)
next
  assume  $\forall b Ts T a ba x. \neg P \vdash C \text{ sees } M, b : Ts \rightarrow T = (a, ba) \text{ in } x$ 
  then show ?thesis using SCallInitRed.hyps(1) by auto
next
  fix Ts T m D assume  $P \vdash C \text{ sees } M, \text{NonStatic} : Ts \rightarrow T = m \text{ in } D$ 
  then show ?thesis using sees-method-fun[OF SCallInitRed.hyps(1)] by blast
next
  fix vsa h1 l1 sh1 Ts T pns body D' a
  assume  $e' = \text{throw } a$  and vals:  $P \vdash \langle \text{map Val vs}, s_1 \rangle [\Rightarrow] \langle \text{map Val vsa}, (h_1, l_1, sh_1) \rangle$ 
    and method:  $P \vdash C \text{ sees } M, \text{Static} : Ts \rightarrow T = (pns, body) \text{ in } D'$ 
    and nDone:  $\forall sfs. sh_1 D' \neq [(sfs, \text{Done})]$ 
    and init':  $P \vdash \langle \text{INIT } D' ([D'], \text{False}) \leftarrow \text{unit}, (h_1, l_1, sh_1) \rangle \Rightarrow \langle \text{throw } a, s' \rangle$ 
  have vs:  $vs = \text{vsa}$  and s1a:  $s_1 = (h_1, l_1, sh_1)$ 
    using evals-finals-same[OF - vals] map-Val-eq by auto
  have D:  $D = D'$  using sees-method-fun[OF SCallInitRed.hyps(1) method] by simp
  then have i:  $i = \text{Processing}$  using iDP shD s1 s1a nDone by simp
  then show ?thesis using D init' init-ProcessingE s1 s1a shD by blast
next
  fix vsa h1 l1 sh1 Ts T pns' body' D' v' h2 l2 sh2 h3 l3 sh3
  assume s':  $s' = (h_3, l_2, sh_3)$ 

```

```

and vals:  $P \vdash \langle \text{map Val vs}, s_1 \rangle [\Rightarrow] \langle \text{map Val vs}, (h_1, l_1, sh_1) \rangle$ 
and method:  $P \vdash C \text{ sees } M, \text{ Static} : Ts \rightarrow T = (pns', \text{body}') \text{ in } D'$ 
and nDone:  $\forall sfs. sh_1 D' \neq [(sfs, \text{Done})]$ 
and init':  $P \vdash \langle \text{INIT } D' ([D], \text{False}) \leftarrow \text{unit}, (h_1, l_1, sh_1) \rangle \Rightarrow \langle \text{Val } v', (h_2, l_2, sh_2) \rangle$ 
and len:  $\text{length vs} = \text{length pns}'$ 
and bstep:  $P \vdash \langle \text{body}', (h_2, [pns' \mapsto] \text{vs}), sh_2 \rangle \Rightarrow \langle e', (h_3, l_3, sh_3) \rangle$ 
have vs:  $vs = \text{vs}$  and  $s_1 a: s_1 = (h_1, l_1, sh_1)$ 
using evals-finals-same[OF - vals] map-Val-eq by auto
have D:  $D = D'$  and pns:  $pns = pns'$  and body:  $\text{body} = \text{body}'$ 
using sees-method-fun[OF SCallInitRed.hypr(1) method] by auto
then have i:  $i = \text{Processing}$  using iDP shD  $s_1 s_1 a$  nDone by simp
then have s2:  $(h_2, l_2, sh_2) = s_1$  using D init' init-ProcessingE  $s_1 s_1 a$  shD by blast
then show ?thesis
using eval-finalId SCallInit[OF eval-finalsId[of map Val vs P (h,l,sh)]]
  SCallInitRed.hypr(1)] init init' len bstep nDone D pns body s'  $s_1 s_1 a$  shD vals vs
  SCallInitRed.hypr(2-3) s2 by auto
next
fix vsa  $h_2 l_2 sh_2 Ts T pns' \text{body}' D' sfs h_3 l_3 sh_3$ 
assume s':  $s' = (h_3, l_2, sh_3)$  and vals:  $P \vdash \langle \text{map Val vs}, s_1 \rangle [\Rightarrow] \langle \text{map Val vs}, (h_2, l_2, sh_2) \rangle$ 
and method:  $P \vdash C \text{ sees } M, \text{ Static} : Ts \rightarrow T = (pns', \text{body}') \text{ in } D'$ 
and sh2 D':  $[(sfs, \text{Done})] \vee M = \text{clinit} \wedge sh_2 D' = [(sfs, \text{Processing})]$ 
and len:  $\text{length vs} = \text{length pns}'$ 
and bstep:  $P \vdash \langle \text{body}', (h_2, [pns' \mapsto] \text{vs}), sh_2 \rangle \Rightarrow \langle e', (h_3, l_3, sh_3) \rangle$ 
have vs:  $vs = \text{vs}$  and  $s_1 a: s_1 = (h_2, l_2, sh_2)$ 
using evals-finals-same[OF - vals] map-Val-eq by auto
have D:  $D = D'$  and pns:  $pns = pns'$  and body:  $\text{body} = \text{body}'$ 
using sees-method-fun[OF SCallInitRed.hypr(1) method] by auto
then show ?thesis using SCallInit[OF eval-finalsId[of map Val vs P (h,l,sh)]]
  SCallInitRed.hypr(1)] bstep SCallInitRed.hypr(2-3) len s'  $s_1 a$  vals vs init by auto
qed
next
fix e assume e':  $e' = \text{throw } e$ 
and init:  $P \vdash \langle \text{INIT } D ([D], \text{False}) \leftarrow \text{unit}, (h, l, sh) \rangle \Rightarrow \langle \text{throw } e, s' \rangle$ 
obtain h' l' sh' where s':  $s' = (h', l', sh')$  by (cases s')
then obtain sfs i where shC:  $sh' D = [(sfs, i)]$  and iDP:  $i = \text{Error}$ 
using init-throw-PD[OF init] by auto
then show ?thesis using SCallInitRed.hypr(2-3) init e'
  SCallInitThrow[OF eval-finalsId[of map Val vs -] SCallInitRed.hypr(1)]
by auto
qed
next
case (RedSCallNone C M vs s b)
then have tes:  $\text{THROW NoSuchMethodError} = e' \wedge s = s'$ 
using eval-final-same by simp
have  $P \vdash \langle \text{map Val vs}, s \rangle [\Rightarrow] \langle \text{map Val vs}, s \rangle$  using eval-finalsId by simp
then show ?case using RedSCallNone eval-evals.SCallNone tes by auto
next
case (RedSCallNonStatic C M Ts T m D vs s b)
then have tes:  $\text{THROW IncompatibleClassChangeError} = e' \wedge s = s'$ 
using eval-final-same by simp
have  $P \vdash \langle \text{map Val vs}, s \rangle [\Rightarrow] \langle \text{map Val vs}, s \rangle$  using eval-finalsId by simp
then show ?case using RedSCallNonStatic eval-evals.SCallNonStatic tes by auto
next
case InitBlockRed

```

```

thus ?case
  by (fastforce elim!: eval-cases intro: eval-vals.intros eval-finalId
      simp: assigned-def map-upd-triv fun-upd-same)
next
  case (RedInitBlock V T v u s b)
  then have  $P \vdash \langle \text{Val } u, s \rangle \Rightarrow \langle e', s' \rangle$  by simp
  then obtain  $s': s'=s$  and  $e': e'=\text{Val } u$  by cases simp
  obtain  $h \ l \ sh$  where  $s = (h, l, sh)$  by (cases s)
  have  $P \vdash \langle \{V:T := \text{Val } v; \text{Val } u\}, (h, l, sh) \rangle \Rightarrow \langle \text{Val } u, (h, (l(V \mapsto v)))(V:=l \ V), sh \rangle$ 
  by (fastforce intro!: eval-vals.intros)
  then have  $P \vdash \langle \{V:T := \text{Val } v; \text{Val } u\}, s \rangle \Rightarrow \langle e', s' \rangle$  using  $s \ s' \ e'$  by simp
  then show ?case by simp
next
  case BlockRedNone
  thus ?case
  by (fastforce elim!: eval-cases intro: eval-vals.intros
      simp add: fun-upd-same fun-upd-idem)
next
  case BlockRedSome
  thus ?case
  by (fastforce elim!: eval-cases intro: eval-vals.intros
      simp add: fun-upd-same fun-upd-idem)
next
  case (RedBlock V T v s b)
  then have  $P \vdash \langle \text{Val } v, s \rangle \Rightarrow \langle e', s' \rangle$  by simp
  then obtain  $s': s'=s$  and  $e': e'=\text{Val } v$ 
  by cases simp
  obtain  $h \ l \ sh$  where  $s = (h, l, sh)$  by (cases s)
  have  $P \vdash \langle \text{Val } v, (h, l(V:=None), sh) \rangle \Rightarrow \langle \text{Val } v, (h, l(V:=None), sh) \rangle$ 
  by (rule eval-vals.intros)
  hence  $P \vdash \langle \{V:T; \text{Val } v\}, (h, l, sh) \rangle \Rightarrow \langle \text{Val } v, (h, (l(V:=None)))(V:=l \ V), sh \rangle$ 
  by (rule eval-vals.Block)
  then have  $P \vdash \langle \{V:T; \text{Val } v\}, s \rangle \Rightarrow \langle e', s' \rangle$  using  $s \ s' \ e'$  by simp
  then show ?case by simp
next
  case (SeqRed e s b e1 s1 b1 e2) show ?case
  proof(cases val-of e)
  case None show ?thesis
  proof(cases lass-val-of e)
  case lNone:None
  then have  $bconf: P, shp \ s \vdash_b (e, b) \checkmark$  using SeqRed.prem(1) None by simp
  then show ?thesis using SeqRed using seq-ext by fastforce
  next
  case (Some p)
  obtain  $V' \ v'$  where  $p: p = (V', v')$  and  $e: e = V' := \text{Val } v'$ 
  using lass-val-of-spec[OF Some] by(cases p, auto)
  obtain  $h \ l \ sh \ h' \ l' \ sh'$  where  $s = (h, l, sh)$  and  $s1: s1 = (h', l', sh')$  by(cases s, cases s1)
  then have  $red: P \vdash \langle e, (h, l, sh), b \rangle \rightarrow \langle e1, (h', l', sh'), b1 \rangle$  using SeqRed.hyps(1) by simp
  then have  $s1': e1 = \text{unit} \wedge h' = h \wedge l' = l(V' \mapsto v') \wedge sh' = sh$ 
  using lass-val-of-red[OF Some red] p e by simp
  then have  $eval: P \vdash \langle e, s \rangle \Rightarrow \langle e1, s1 \rangle$  using e s s1 LAss Val by auto
  then show ?thesis
  by (metis SeqRed.prem(2) eval-final eval-final-same seq-ext)
  qed

```

```

next
  case (Some a) then show ?thesis using SeqRed.hyps(1) val-no-step by blast
qed
next
  case RedSeq
  thus ?case
  by (fastforce elim: eval-cases intro: eval-vals.intros)
next
  case CondRed
  thus ?case
  by (fastforce elim: eval-cases intro: eval-vals.intros simp: val-no-step)
next
  case RedCondT
  thus ?case
  by (fastforce elim: eval-cases intro: eval-vals.intros)
next
  case RedCondF
  thus ?case
  by (fastforce elim: eval-cases intro: eval-vals.intros)
next
  case RedWhile
  thus ?case
  by (auto simp add: unfold-while intro:eval-vals.intros elim:eval-cases)
next
  case ThrowRed then show ?case by(fastforce elim: eval-cases simp: eval-vals.intros)
next
  case RedThrowNull
  thus ?case
  by (fastforce elim: eval-cases intro: eval-vals.intros)
next
  case TryRed thus ?case
  by(fastforce elim: eval-cases intro: eval-vals.intros)
next
  case RedTryCatch
  thus ?case
  by (fastforce elim: eval-cases intro: eval-vals.intros)
next
  case (RedTryFail s a D fs C V e2 b)
  thus ?case
  by (cases s)(auto elim!: eval-cases intro: eval-vals.intros)
next
  case ListRed1
  thus ?case
  by (fastforce elim: evals-cases intro: eval-vals.intros simp: val-no-step)
next
  case ListRed2
  thus ?case
  by (fastforce elim!: evals-cases eval-cases
      intro: eval-vals.intros eval-finalId)
next
  case (RedInit e1 C b s1 b') then show ?case using InitFinal by simp
next
  case (InitNoneRed sh C C' Cs e h l b)
  show ?case using InitNone InitNoneRed.hyps InitNoneRed.premis(2) by auto

```

```

next
  case (RedInitDone sh C sfs C' Cs e h l b)
  show ?case using InitDone RedInitDone.hyps RedInitDone.prem(2) by auto
next
  case (RedInitProcessing sh C sfs C' Cs e h l b)
  show ?case using InitProcessing RedInitProcessing.hyps RedInitProcessing.prem(2) by auto
next
  case (RedInitError sh C sfs C' Cs e h l b)
  show ?case using InitError RedInitError.hyps RedInitError.prem(2) by auto
next
  case (InitObjectRed sh C sfs sh' C' Cs e h l b) show ?case using InitObject InitObjectRed by auto
next
  case (InitNonObjectSuperRed sh C sfs D r sh' C' Cs e h l b)
  show ?case using InitNonObject InitNonObjectSuperRed by auto
next
  case (RedInitRInit C' C Cs e h l sh b)
  show ?case using InitRInit RedInitRInit by auto
next
  case (RInitRed e s b e'' s'' b'' C Cs e0)
  then have IH:  $\bigwedge e' s'. P \vdash \langle e'', s'' \rangle \Rightarrow \langle e', s' \rangle \implies P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$  by simp
  show ?case using RInitRed rinit-ext[OF IH] by simp
next
  case (RedRInit sh C sfs i sh' C' Cs v e h l b s' e')
  then have init:  $P \vdash \langle (INIT C' (Cs, True) \leftarrow e), (h, l, sh(C \mapsto (sfs, Done))) \rangle \Rightarrow \langle e', s' \rangle$ 
    using RedRInit by simp
  then show ?case using RInit RedRInit.hyps(1) RedRInit.hyps(3) Val by fastforce
next
  case BinOpThrow2
  thus ?case
  by (fastforce elim: eval-cases intro: eval-vals.intros)
next
  case FAssThrow2
  thus ?case
  by (fastforce elim: eval-cases intro: eval-vals.intros)
next
  case SFAssThrow
  then show ?case
  by (fastforce elim: eval-cases intro: eval-vals.intros)
next
  case (CallThrowParams es vs e es' v M s b)
  have val:  $P \vdash \langle Val v, s \rangle \Rightarrow \langle Val v, s \rangle$  by (rule eval-vals.intros)
  have eval-e:  $P \vdash \langle throw (e), s \rangle \Rightarrow \langle e', s' \rangle$  using CallThrowParams by simp
  then obtain xa where e':  $e' = Throw xa$  by (cases) (auto dest!: eval-final)
  with list-eval-Throw [OF eval-e]
  have vals:  $P \vdash \langle es, s \rangle [\Rightarrow] \langle map Val vs @ Throw xa \# es', s' \rangle$ 
    using CallThrowParams.hyps(1) eval-e list-eval-Throw by blast
  then have  $P \vdash \langle Val v \cdot M(es), s \rangle \Rightarrow \langle Throw xa, s' \rangle$ 
    using eval-vals.CallParamsThrow[OF val vals] by simp
  thus ?case using e' by simp
next
  case (SCallThrowParams es vs e es' C M s b)
  have eval-e:  $P \vdash \langle throw (e), s \rangle \Rightarrow \langle e', s' \rangle$  using SCallThrowParams by simp
  then obtain xa where e':  $e' = Throw xa$  by (cases) (auto dest!: eval-final)
  then have  $P \vdash \langle es, s \rangle [\Rightarrow] \langle map Val vs @ Throw xa \# es', s' \rangle$ 

```

```

  using SCallThrowParams.hyps(1) eval-e list-eval-Throw by blast
  then have  $P \vdash \langle C \cdot_s M(es), s \rangle \Rightarrow \langle \text{Throw } xa, s^\wedge \rangle$ 
    by (rule eval-vals.SCallParamsThrow)
  thus ?case using  $e'$  by simp
next
case (BlockThrow V T a s b)
then have  $P \vdash \langle \text{Throw } a, s \rangle \Rightarrow \langle e', s^\wedge \rangle$  by simp
then obtain  $s': s' = s$  and  $e': e' = \text{Throw } a$ 
  by cases (auto elim!: eval-cases)
obtain  $h l sh$  where  $s: s = (h, l, sh)$  by (cases s)
have  $P \vdash \langle \text{Throw } a, (h, l(V:=None), sh) \rangle \Rightarrow \langle \text{Throw } a, (h, l(V:=None), sh) \rangle$ 
  by (simp add: eval-vals.intros eval-finalId)
hence  $P \vdash \langle \{V:T; \text{Throw } a\}, (h, l, sh) \rangle \Rightarrow \langle \text{Throw } a, (h, l(V:=None))(V:=l V), sh \rangle$ 
  by (rule eval-vals.Block)
then have  $P \vdash \langle \{V:T; \text{Throw } a\}, s \rangle \Rightarrow \langle e', s^\wedge \rangle$  using  $s s' e'$  by simp
then show ?case by simp
next
case (InitBlockThrow V T v a s b)
then have  $P \vdash \langle \text{Throw } a, s \rangle \Rightarrow \langle e', s^\wedge \rangle$  by simp
then obtain  $s': s' = s$  and  $e': e' = \text{Throw } a$ 
  by cases (auto elim!: eval-cases)
obtain  $h l sh$  where  $s: s = (h, l, sh)$  by (cases s)
have  $P \vdash \langle \{V:T := \text{Val } v; \text{Throw } a\}, (h, l, sh) \rangle \Rightarrow \langle \text{Throw } a, (h, (l(V \mapsto v))(V:=l V), sh) \rangle$ 
  by (fastforce intro: eval-vals.intros)
then have  $P \vdash \langle \{V:T := \text{Val } v; \text{Throw } a\}, s \rangle \Rightarrow \langle e', s^\wedge \rangle$  using  $s s' e'$  by simp
then show ?case by simp
next
case (RInitInitThrow sh C sfs i sh' a D Cs e h l b)
have IH:  $\bigwedge e' s'. P \vdash \langle RI(D, \text{Throw } a); Cs \leftarrow e, (h, l, sh(C \mapsto (sfs, \text{Error}))) \rangle \Rightarrow \langle e', s^\wedge \rangle \Rightarrow$ 
 $P \vdash \langle RI(C, \text{Throw } a); D \# Cs \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s^\wedge \rangle$ 
  using RInitInitFail[OF eval-finalId] RInitInitThrow by simp
then show ?case using RInitInitThrow.hyps(2) RInitInitThrow.prem(2) by auto
next
case (RInitThrow sh C sfs i sh' a e h l b)
then have  $e': e' = \text{Throw } a$  and  $s': s' = (h, l, sh')$ 
  using eval-final-same final-def by fastforce+
show ?case using RInitFailFinal RInitThrow.hyps(1) RInitThrow.hyps(2)  $e'$  eval-finalId  $s'$  by auto
qed(auto elim: eval-cases simp: eval-vals.intros)

```

Its extension to \rightarrow^* :

lemma *extend-eval*:

assumes wf : *wf-J-prog* P

shows $\llbracket P \vdash \langle e, s, b \rangle \rightarrow^* \langle e'', s'', b'' \rangle; P \vdash \langle e'', s'' \rangle \Rightarrow \langle e', s^\wedge \rangle;$
 $\text{iconf } (shp s) e; P, shp s \vdash_b (e::\text{expr}, b) \checkmark \rrbracket$
 $\Rightarrow P \vdash \langle e, s \rangle \Rightarrow \langle e', s^\wedge \rangle$

lemma *extend-evals*:

assumes wf : *wf-J-prog* P

shows $\llbracket P \vdash \langle es, s, b \rangle [\rightarrow]^* \langle es'', s'', b'' \rangle; P \vdash \langle es'', s'' \rangle [\Rightarrow] \langle es', s^\wedge \rangle;$
 $\text{iconfs } (shp s) es; P, shp s \vdash_b (es::\text{expr list}, b) \checkmark \rrbracket$
 $\Rightarrow P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s^\wedge \rangle$

Finally, small step semantics can be simulated by big step semantics:

theorem**assumes** $wf: wuf\text{-}J\text{-}prog\ P$ **shows** *small-by-big*:

$$\llbracket P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', b' \rangle; iconf\ (shp\ s)\ e; P, shp\ s \vdash_b (e, b)\ \checkmark; final\ e' \rrbracket \\ \implies P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$$

and $\llbracket P \vdash \langle es, s, b \rangle [\rightarrow]^* \langle es', s', b' \rangle; iconfs\ (shp\ s)\ es; P, shp\ s \vdash_b (es, b)\ \checkmark; finals\ es' \rrbracket \\ \implies P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle$

1.23.4 Equivalence

And now, the crowning achievement:

corollary *big-iff-small*:

$$\llbracket wuf\text{-}J\text{-}prog\ P; iconf\ (shp\ s)\ e; P, shp\ s \vdash_b (e::expr, b)\ \checkmark \rrbracket \\ \implies P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle = (P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', False \rangle \wedge final\ e')$$

corollary *big-iff-small-WT*:

$$wuf\text{-}J\text{-}prog\ P \implies P, E \vdash e::T \implies P, shp\ s \vdash_b (e, b)\ \checkmark \implies \\ P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle = (P \vdash \langle e, s, b \rangle \rightarrow^* \langle e', s', False \rangle \wedge final\ e')$$

1.23.5 Lifting type safety to \Rightarrow

... and now to the big step semantics, just for fun.

lemma *eval-preserves-sconf*:**fixes** $s::state$ **and** $s'::state$

assumes $wf\text{-}J\text{-}prog\ P$ **and** $P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$ **and** $iconf\ (shp\ s)\ e$
and $P, E \vdash e::T$ **and** $P, E \vdash s\checkmark$

shows $P, E \vdash s'\checkmark$ **lemma** *eval-preserves-type*:**fixes** $s::state$ **assumes** $wf: wf\text{-}J\text{-}prog\ P$

and $P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$ **and** $P, E \vdash s\checkmark$ **and** $iconf\ (shp\ s)\ e$ **and** $P, E \vdash e::T$
shows $\exists T'. P \vdash T' \leq T \wedge P, E, hp\ s', shp\ s' \vdash e':T'$

end**1.24 Program annotation****theory** *Annotate* **imports** *WellType* **begin****inductive**

$$Anno :: [J\text{-}prog, env, expr\ \ \ , expr] \Rightarrow bool \\ (\langle -, - \vdash - \rightsquigarrow - \rangle [51, 0, 0, 51] 50)$$

$$\mathbf{and}\ Annos :: [J\text{-}prog, env, expr\ list, expr\ list] \Rightarrow bool \\ (\langle -, - \vdash - [\rightsquigarrow] - \rangle [51, 0, 0, 51] 50)$$

for $P :: J\text{-}prog$ **where**

$$AnnoNew: P, E \vdash new\ C \rightsquigarrow new\ C$$

$$| AnnoCast: P, E \vdash e \rightsquigarrow e' \implies P, E \vdash Cast\ C\ e \rightsquigarrow Cast\ C\ e'$$

$$| AnnoVal: P, E \vdash Val\ v \rightsquigarrow Val\ v$$

$$| AnnoVarVar: E\ V = [T] \implies P, E \vdash Var\ V \rightsquigarrow Var\ V$$

$$| AnnoVarField: \llbracket E\ V = None; E\ this = [Class\ C]; P \vdash C\ sees\ V, NonStatic: T\ in\ D \rrbracket \\ \implies P, E \vdash Var\ V \rightsquigarrow Var\ this \cdot V\{D\}$$

| *AnnoBinOp*:
 $\llbracket P, E \vdash e1 \rightsquigarrow e1'; P, E \vdash e2 \rightsquigarrow e2' \rrbracket$
 $\implies P, E \vdash e1 \llbracket \text{bop} \rrbracket e2 \rightsquigarrow e1' \llbracket \text{bop} \rrbracket e2'$

| *AnnoLAssVar*:
 $\llbracket E V = \llbracket T \rrbracket; P, E \vdash e \rightsquigarrow e' \rrbracket \implies P, E \vdash V := e \rightsquigarrow V := e'$

| *AnnoLAssField*:
 $\llbracket E V = \text{None}; E \text{ this} = \llbracket \text{Class } C \rrbracket; P \vdash C \text{ sees } V, \text{NonStatic}:T \text{ in } D; P, E \vdash e \rightsquigarrow e' \rrbracket$
 $\implies P, E \vdash V := e \rightsquigarrow \text{Var this} \cdot V \{D\} := e'$

| *AnnoFAcc*:
 $\llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash e' :: \text{Class } C; P \vdash C \text{ sees } F, \text{NonStatic}:T \text{ in } D \rrbracket$
 $\implies P, E \vdash e \cdot F \{\llbracket \rrbracket\} \rightsquigarrow e' \cdot F \{D\}$

| *AnnoSFAcc*:
 $\llbracket P \vdash C \text{ sees } F, \text{Static}:T \text{ in } D \rrbracket$
 $\implies P, E \vdash C \cdot_s F \{\llbracket \rrbracket\} \rightsquigarrow C \cdot_s F \{D\}$

| *AnnoFAss*: $\llbracket P, E \vdash e1 \rightsquigarrow e1'; P, E \vdash e2 \rightsquigarrow e2';$
 $P, E \vdash e1' :: \text{Class } C; P \vdash C \text{ sees } F, \text{NonStatic}:T \text{ in } D \rrbracket$
 $\implies P, E \vdash e1 \cdot F \{\llbracket \rrbracket\} := e2 \rightsquigarrow e1' \cdot F \{D\} := e2'$

| *AnnoSFAss*: $\llbracket P, E \vdash e2 \rightsquigarrow e2'; P \vdash C \text{ sees } F, \text{Static}:T \text{ in } D \rrbracket$
 $\implies P, E \vdash C \cdot_s F \{\llbracket \rrbracket\} := e2 \rightsquigarrow C \cdot_s F \{D\} := e2'$

| *AnnoCall*:
 $\llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash es \llbracket \rightsquigarrow \rrbracket es' \rrbracket$
 $\implies P, E \vdash \text{Call } e \text{ M } es \rightsquigarrow \text{Call } e' \text{ M } es'$

| *AnnoSCall*:
 $\llbracket P, E \vdash es \llbracket \rightsquigarrow \rrbracket es' \rrbracket$
 $\implies P, E \vdash \text{SCall } C \text{ M } es \rightsquigarrow \text{SCall } C \text{ M } es'$

| *AnnoBlock*:
 $P, E(V \mapsto T) \vdash e \rightsquigarrow e' \implies P, E \vdash \{V:T; e\} \rightsquigarrow \{V:T; e'\}$

| *AnnoComp*: $\llbracket P, E \vdash e1 \rightsquigarrow e1'; P, E \vdash e2 \rightsquigarrow e2' \rrbracket$
 $\implies P, E \vdash e1 ;; e2 \rightsquigarrow e1' ;; e2'$

| *AnnoCond*: $\llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash e1 \rightsquigarrow e1'; P, E \vdash e2 \rightsquigarrow e2' \rrbracket$
 $\implies P, E \vdash \text{if } (e) \ e1 \ \text{else } e2 \rightsquigarrow \text{if } (e') \ e1' \ \text{else } e2'$

| *AnnoLoop*: $\llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash c \rightsquigarrow c' \rrbracket$
 $\implies P, E \vdash \text{while } (e) \ c \rightsquigarrow \text{while } (e') \ c'$

| *AnnoThrow*: $P, E \vdash e \rightsquigarrow e' \implies P, E \vdash \text{throw } e \rightsquigarrow \text{throw } e'$

| *AnnoTry*: $\llbracket P, E \vdash e1 \rightsquigarrow e1'; P, E(V \mapsto \text{Class } C) \vdash e2 \rightsquigarrow e2' \rrbracket$
 $\implies P, E \vdash \text{try } e1 \ \text{catch}(C \ V) \ e2 \rightsquigarrow \text{try } e1' \ \text{catch}(C \ V) \ e2'$

| *AnnoNil*: $P, E \vdash \llbracket \rightsquigarrow \rrbracket \llbracket \rrbracket$

| *AnnoCons*: $\llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash es \llbracket \rightsquigarrow \rrbracket es' \rrbracket$
 $\implies P, E \vdash e \# es \llbracket \rightsquigarrow \rrbracket e' \# es'$

end

Chapter 2

Jinja Virtual Machine

2.1 State of the JVM

theory *JVMState* **imports** *../Common/Objects* **begin**

type-synonym

pc = *nat*

abbreviation *start-sheap* :: *sheap*

where *start-sheap* ≡ (λ*x*. *None*)(*Start* ↦ (*Map.empty*,*Done*))

definition *start-sheap-preloaded* :: '*m prog* ⇒ *sheap*

where

start-sheap-preloaded *P* ≡ *fold* (λ(*C*,*cl*) *f*. *f*(*C* := *Some* (*sblank* *P* *C*, *Prepared*))) *P* (λ*x*. *None*)

2.1.1 Frame Stack

datatype *init-call-status* = *No-ics* | *Calling* *cname* *cname list*
| *Called* *cname list* | *Throwing* *cname list* *addr*

- *No-ics* = not currently calling or waiting for the result of an initialization procedure call
- *Calling* *C Cs* = current instruction is calling for initialization of classes *C#Cs* (last class is the original) – still collecting classes to be initialized, *C* most recently collected
- *Called* *Cs* = current instruction called initialization and is waiting for the result – now initializing classes in the list
- *Throwing* *Cs a* = frame threw or was thrown an error causing erroneous end of initialization procedure for classes *Cs*

type-synonym

frame = *val list* × *val list* × *cname* × *mname* × *pc* × *init-call-status*

- operand stack
- registers (including this pointer, method parameters, and local variables)
- name of class where current method is defined
- current method
- program counter within frame
- indicates frame's initialization call status

translations

(*type*) *frame* <= (*type*) *val list* × *val list* × *char list* × *char list* × *nat* × *init-call-status*

fun *curr-stk* :: *frame* \Rightarrow *val list* **where**
curr-stk (*stk*, *loc*, *C*, *M*, *pc*, *ics*) = *stk*

fun *curr-class* :: *frame* \Rightarrow *cname* **where**
curr-class (*stk*, *loc*, *C*, *M*, *pc*, *ics*) = *C*

fun *curr-method* :: *frame* \Rightarrow *mname* **where**
curr-method (*stk*, *loc*, *C*, *M*, *pc*, *ics*) = *M*

fun *curr-pc* :: *frame* \Rightarrow *nat* **where**
curr-pc (*stk*, *loc*, *C*, *M*, *pc*, *ics*) = *pc*

fun *init-status* :: *frame* \Rightarrow *init-call-status* **where**
init-status (*stk*, *loc*, *C*, *M*, *pc*, *ics*) = *ics*

fun *ics-of* :: *frame* \Rightarrow *init-call-status* **where**
ics-of *fr* = *snd*(*snd*(*snd*(*snd*(*snd* *fr*))))

2.1.2 Runtime State

type-synonym

jvm-state = *addr option* \times *heap* \times *frame list* \times *sheap*
— exception flag, heap, frames, static heap

translations

(*type*) *jvm-state* \leq (*type*) *nat option* \times *heap* \times *frame list* \times *sheap*

fun *frames-of* :: *jvm-state* \Rightarrow *frame list* **where**
frames-of (*xp*, *h*, *frs*, *sh*) = *frs*

abbreviation *sheap* :: *jvm-state* \Rightarrow *sheap* **where**
sheap js \equiv *snd* (*snd* (*snd* *js*))

end

2.2 Instructions of the JVM

theory *JVMInstructions* **imports** *JVMState* **begin**

datatype

instr = *Load nat* — load from local variable
| *Store nat* — store into local variable
| *Push val* — push a value (constant)
| *New cname* — create object
| *Getfield vname cname* — Fetch field from object
| *Getstatic cname vname cname* — Fetch static field from class
| *Putfield vname cname* — Set field in object
| *Putstatic cname vname cname* — Set static field in class
| *Checkcast cname* — Check whether object is of given type
| *Invoke mname nat* — inv. instance meth of an object
| *Invokestatic cname mname nat* — inv. static method of a class
| *Return* — return from method
| *Pop* — pop top element from opstack

<i>IAdd</i>	— integer addition
<i>Goto int</i>	— goto relative address
<i>CmpEq</i>	— equality comparison
<i>IfFalse int</i>	— branch if top of stack false
<i>Throw</i>	— throw top of stack as exception

type-synonym

bytecode = *instr list*

type-synonym

ex-entry = *pc* × *pc* × *cname* × *pc* × *nat*
 — start-pc, end-pc, exception type, handler-pc, remaining stack depth

type-synonym

ex-table = *ex-entry list*

type-synonym

jvm-method = *nat* × *nat* × *bytecode* × *ex-table*
 — max stacksize
 — number of local variables. Add 1 + no. of parameters to get no. of registers
 — instruction sequence
 — exception handler table

type-synonym

jvm-prog = *jvm-method prog*

end

2.3 Exception handling in the JVM

theory *JVMExceptions* **imports** *../Common/Exceptions JVMInstructions*
begin

definition *matches-ex-entry* :: *'m prog* ⇒ *cname* ⇒ *pc* ⇒ *ex-entry* ⇒ *bool*
where

matches-ex-entry *P C pc xcp* ≡
 let (*s*, *e*, *C'*, *h*, *d*) = *xcp* in
 $s \leq pc \wedge pc < e \wedge P \vdash C \preceq^* C'$

primrec *match-ex-table* :: *'m prog* ⇒ *cname* ⇒ *pc* ⇒ *ex-table* ⇒ (*pc* × *nat*) *option*
where

match-ex-table *P C pc []* = *None*
 | *match-ex-table* *P C pc (e#es)* = (if *matches-ex-entry* *P C pc e*
 then *Some (snd(snd(snd e)))*
 else *match-ex-table* *P C pc es*)

abbreviation

ex-table-of :: *jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *ex-table* **where**
ex-table-of *P C M* == *snd (snd (snd (snd (snd (snd (method P C M))))))*)

fun *find-handler* :: *jvm-prog* ⇒ *addr* ⇒ *heap* ⇒ *frame list* ⇒ *sheap* ⇒ *jvm-state*

where

```

  find-handler P a h [] sh = (Some a, h, [], sh)
| find-handler P a h (fr#frs) sh =
  (let (stk,loc,C,M,pc,ics) = fr in
   case match-ex-table P (cname-of h a) pc (ex-table-of P C M) of
   None =>
     (case M = clinit of
      True => (case frs of (stk',loc',C',M',pc',ics')#frs'
                       => (case ics' of Called Cs => (None, h, (stk',loc',C',M',pc',Throwing
(C#Cs) a)#frs', sh)
                       | - => (None, h, (stk',loc',C',M',pc',ics')#frs', sh) — this won't
happen in wf code
                       )
      | [] => (Some a, h, [], sh)
      )
   | - => find-handler P a h frs sh
   )
| Some pc-d => (None, h, (Addr a # drop (size stk - snd pc-d) stk, loc, C, M, fst pc-d,
No-ics)#frs, sh))

```

lemma *find-handler-cases*:

```

find-handler P a h frs sh = js
=>> (∃ frs'. frs' ≠ [] ∧ js = (None, h, frs', sh)) ∨ (js = (Some a, h, [], sh))

```

proof(*induct P a h frs sh rule: find-handler.induct*)

case 1 then show ?case **by** *clarsimp*

next

case (2 P a h fr frs sh) **then show** ?case

by(*cases fr, auto split: bool.splits list.splits init-call-status.splits*)

qed

lemma *find-handler-heap[simp]*:

```

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

```

by(*auto dest: find-handler-cases*)

lemma *find-handler-sheap[simp]*:

```

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

```

by(*auto dest: find-handler-cases*)

lemma *find-handler-frames[simp]*:

```

find-handler P a h frs sh = (xp',h',frs',sh') =>> length frs' ≤ length frs

```

proof(*induct frs*)

case Nil then show ?case **by** *simp*

next

case (Cons a frs) **then show** ?case

by(*auto simp: split-beta split: bool.splits list.splits init-call-status.splits*)

qed

lemma *find-handler-None*:

```

find-handler P a h frs sh = (None, h, frs', sh') =>> frs' ≠ []

```

by (*drule find-handler-cases, clarsimp*)

lemma *find-handler-Some*:

```

find-handler P a h frs sh = (Some x, h, frs', sh') =>> frs' = []

```

by (*drule find-handler-cases, clarsimp*)

lemma *find-handler-Some-same-error-same-heap*[simp]:
find-handler P a h frs $sh = (Some\ x,\ h',\ frs',\ sh') \implies x = a \wedge h = h' \wedge sh = sh'$
by(*auto dest: find-handler-cases*)

lemma *find-handler-prealloc-pres*:
assumes *preallocated* h
and fh : *find-handler* P a h frs $sh = (xp',h',frs',sh')$
shows *preallocated* h'
using *assms find-handler-heap[OF fh]* **by** *simp*

lemma *find-handler-frs-tl-neq*:
ics-of $f \neq No\ ics$
 $\implies (xp,\ h,\ f\#\#frs,\ sh) \neq find\text{-}handler\ P\ xa\ h'\ (f'\ \#\#frs)\ sh'$
proof(*induct frs arbitrary: f f'*)
case *Nil* **then show** *?case* **by**(*auto simp: split-beta split: bool.splits*)
next
case (*Cons* a frs)
obtain $xp1\ h1\ frs1\ sh1$ **where** fh : *find-handler* P xa h' ($a\ \#\#frs$) $sh' = (xp1,h1,frs1,sh1)$
by(*cases find-handler* P xa h' ($a\ \#\#frs$) sh')
then have $length\ frs1 \leq length\ (a\ \#\#frs)$
by(*rule find-handler-frames[where P=P and a=xa and h=h' and frs=a#\#frs and sh=sh']*)
then have neq : $f\#\#a\#\#frs \neq frs1$ **by**(*clarsimp dest: impossible-Cons*)
then show *?case*
proof(*cases find-handler* P xa h' ($f'\ \#\#a\ \#\#frs$) $sh' = find\text{-}handler\ P\ xa\ h'\ (a\ \#\#frs)\ sh'$)
case *True* **then show** *?thesis* **using** $neq\ fh$ **by** *simp*
next
case *False* **then show** *?thesis* **using** *Cons.prem*
by(*fastforce simp: split-beta split: bool.splits init-call-status.splits list.splits*)
qed
qed
end

2.4 Program Execution in the JVM

theory *JVMExecInstr*
imports *JVMInstructions JVMExceptions*
begin

— frame calling the class initialization method for the given class in the given program

fun *create-init-frame* :: [*jvm-prog*, *cname*] \Rightarrow *frame* **where**
create-init-frame $P\ C =$
 (*let* ($D,b,Ts,T,(mxs,mxl_0,ins,xt)$) = *method* $P\ C\ clinit$
 in ($[],(replicate\ mxl_0\ undefined),D,clinit,0,No\ ics$)
)

primrec *exec-instr* :: [*instr*, *jvm-prog*, *heap*, *val list*, *val list*,
cname, *mname*, *pc*, *init-call-status*, *frame list*, *sheap*] \Rightarrow *jvm-state*

where

exec-instr-Load:
exec-instr (*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-instr-Store*:
exec-instr (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-instr-Push*:
exec-instr (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-instr-New*:
exec-instr (New C) P h stk loc C₀ M₀ pc ics frs sh =
(case (ics, sh C) of
(Called Cs, -) ⇒
(case new-Addr h of
None ⇒ ([addr-of-sys-xcpt OutOfMemory], h, (stk, loc, C₀, M₀, pc, No-ics)#frs, sh)
| Some a ⇒ (None, h(a→blank P C), (Addr a#stk, loc, C₀, M₀, Suc pc, No-ics)#frs, sh)
)
| (-, Some(obj, Done)) ⇒
(case new-Addr h of
None ⇒ ([addr-of-sys-xcpt OutOfMemory], h, (stk, loc, C₀, M₀, pc, ics)#frs, sh)
| Some a ⇒ (None, h(a→blank P C), (Addr a#stk, loc, C₀, M₀, Suc pc, ics)#frs, sh)
)
| - ⇒ (None, h, (stk, loc, C₀, M₀, pc, Calling C [])#frs, sh)
)

| *exec-instr-Getfield*:
exec-instr (Getfield F C) P h stk loc C₀ M₀ pc ics frs sh =
(let v = hd stk;
(D,fs) = the(h(the-Addr v));
(D',b,t) = field P C F;
xp' = if v=None then [addr-of-sys-xcpt NullPointer]
else if ¬(∃ t b. P ⊢ D has F,b:t in C)
then [addr-of-sys-xcpt NoSuchFieldError]
else case b of Static ⇒ [addr-of-sys-xcpt IncompatibleClassChangeError]
| NonStatic ⇒ None
in case xp' of None ⇒ (xp', h, (the(fs(F,C))#(tl stk), loc, C₀, M₀, pc+1, ics)#frs, sh)
| Some x ⇒ (xp', h, (stk, loc, C₀, M₀, pc, ics)#frs, sh))

| *exec-instr-Getstatic*:
exec-instr (Getstatic C F D) P h stk loc C₀ M₀ pc ics frs sh =
(let (D',b,t) = field P D F;
xp' = if ¬(∃ t b. P ⊢ C has F,b:t in D)
then [addr-of-sys-xcpt NoSuchFieldError]
else case b of NonStatic ⇒ [addr-of-sys-xcpt IncompatibleClassChangeError]
| Static ⇒ None
in (case (xp', ics, sh D') of
(Some a, -) ⇒ (xp', h, (stk, loc, C₀, M₀, pc, ics)#frs, sh)
| (-, Called Cs, -) ⇒ let (sfs, i) = the(sh D');
v = the(sfs F)
in (xp', h, (v#stk, loc, C₀, M₀, Suc pc, No-ics)#frs, sh)
| (-, -, Some (sfs, Done)) ⇒ let v = the (sfs F)
in (xp', h, (v#stk, loc, C₀, M₀, Suc pc, ics)#frs, sh)
| - ⇒ (xp', h, (stk, loc, C₀, M₀, pc, Calling D' [])#frs, sh)
)

```

)

| exec-instr-Putfield:
exec-instr (Putfield F C) P h stk loc C0 M0 pc ics frs sh =
  (let v    = hd stk;
      r    = hd (tl stk);
      a    = the-Addr r;
      (D,fs) = the (h a);
      (D',b,t) = field P C F;
      xp'   = if r=None then [addr-of-sys-xcpt NullPointer]
              else if ¬(∃ t b. P ⊢ D has F,b:t in C)
                  then [addr-of-sys-xcpt NoSuchFieldError]
                  else case b of Static ⇒ [addr-of-sys-xcpt IncompatibleClassChangeError]
                          | NonStatic ⇒ None;
      h'   = h(a ↦ (D, fs((F,C) ↦ v)))
  in case xp' of None ⇒ (xp', h', (tl (tl stk), loc, C0, M0, pc+1, ics)#frs, sh)
    | Some x ⇒ (xp', h, (stk, loc, C0, M0, pc, ics)#frs, sh)
)

| exec-instr-Putstatic:
exec-instr (Putstatic C F D) P h stk loc C0 M0 pc ics frs sh =
  (let (D',b,t) = field P D F;
      xp'   = if ¬(∃ t b. P ⊢ C has F,b:t in D)
              then [addr-of-sys-xcpt NoSuchFieldError]
              else case b of NonStatic ⇒ [addr-of-sys-xcpt IncompatibleClassChangeError]
                          | Static ⇒ None
  in (case (xp', ics, sh D') of
      (Some a, -) ⇒ (xp', h, (stk, loc, C0, M0, pc, ics)#frs, sh)
    | (-, Called Cs, -)
    ⇒ let (sfs, i) = the(sh D')
        in (xp', h, (tl stk, loc, C0, M0, Suc pc, No-ics)#frs, sh(D':=Some ((sfs(F ↦ hd stk)), i)))
    | (-, -, Some (sfs, Done))
    ⇒ (xp', h, (tl stk, loc, C0, M0, Suc pc, ics)#frs, sh(D':=Some ((sfs(F ↦ hd stk)), Done)))
    | - ⇒ (xp', h, (stk, loc, C0, M0, pc, Calling D' [])#frs, sh)
)

| exec-instr-Checkcast:
exec-instr (Checkcast C) P h stk loc C0 M0 pc ics frs sh =
  (if cast-ok P C h (hd stk)
   then (None, h, (stk, loc, C0, M0, Suc pc, ics)#frs, sh)
   else ([addr-of-sys-xcpt ClassCast], h, (stk, loc, C0, M0, pc, ics)#frs, sh)
)

| exec-instr-Invoke:
exec-instr (Invoke M n) P h stk loc C0 M0 pc ics frs sh =
  (let ps = take n stk;
      r   = stk!n;
      C   = fst(the(h(the-Addr r)));
      (D,b,Ts,T,mxs,mxl0,ins,xt) = method P C M;
      xp' = if r=None then [addr-of-sys-xcpt NullPointer]
            else if ¬(∃ Ts T m D b. P ⊢ C sees M,b:Ts → T = m in D)
                then [addr-of-sys-xcpt NoSuchMethodError]
                else case b of Static ⇒ [addr-of-sys-xcpt IncompatibleClassChangeError]
)

```

```

      | NonStatic ⇒ None;
    f' = ([, [r]@ (rev ps)@ (replicate mxl0 undefined), D, M, 0, No-ics)
  in case xp' of None ⇒ (xp', h, f'#(stk, loc, C0, M0, pc, ics)#frs, sh)
      | Some a ⇒ (xp', h, (stk, loc, C0, M0, pc, ics)#frs, sh)
)

| exec-instr-Invokestatic:
exec-instr Invokestatic C M n P h stk loc C0 M0 pc ics frs sh =
  (let ps = take n stk;
    (D, b, Ts, T, mxs, mxl0, ins, xt) = method P C M;
    xp' = if ¬(∃ Ts T m D b. P ⊢ C sees M, b: Ts → T = m in D)
      then [addr-of-sys-xcpt NoSuchMethodError]
      else case b of NonStatic ⇒ [addr-of-sys-xcpt IncompatibleClassChangeError]
      | Static ⇒ None;
    f' = ([, (rev ps)@ (replicate mxl0 undefined), D, M, 0, No-ics)
  in (case (xp', ics, sh D) of
    (Some a, -) ⇒ (xp', h, (stk, loc, C0, M0, pc, ics)#frs, sh)
    | (-, Called Cs, -) ⇒ (xp', h, f'#(stk, loc, C0, M0, pc, No-ics)#frs, sh)
    | (-, -, Some (sfs, Done)) ⇒ (xp', h, f'#(stk, loc, C0, M0, pc, ics)#frs, sh)
    | - ⇒ (xp', h, (stk, loc, C0, M0, pc, Calling D [])#frs, sh)
  )
)

| exec-instr-Return:
exec-instr Return P h stk0 loc0 C0 M0 pc ics frs sh =
  (case frs of
    [] ⇒ let sh' = (case M0 = clinit of True ⇒ sh(C0 := Some(fst(the(sh C0))), Done)
      | - ⇒ sh
    )
    in (None, h, [], sh')
  | (stk', loc', C', m', pc', ics')#frs'
    ⇒ let (D, b, Ts, T, (mxs, mxl0, ins, xt)) = method P C0 M0;
      offset = case b of NonStatic ⇒ 1 | Static ⇒ 0;
      (sh'', stk'', pc'') = (case M0 = clinit of True ⇒ (sh(C0 := Some(fst(the(sh C0))), Done),
stk', pc')
      | - ⇒ (sh, (hd stk0)#(drop (length Ts + offset) stk'), Suc pc')
    )
    in (None, h, (stk'', loc', C', m', pc'', ics')#frs', sh'')
  )

| exec-instr-Pop:
exec-instr Pop P h stk loc C0 M0 pc ics frs sh = (None, h, (tl stk, loc, C0, M0, Suc pc, ics)#frs, sh)

| exec-instr-IAdd:
exec-instr IAdd P h stk loc C0 M0 pc ics frs sh =
  (None, h, (Intg (the-Intg (hd (tl stk))) + the-Intg (hd stk))#(tl (tl stk)), loc, C0, M0, Suc pc,
ics)#frs, sh)

| exec-instr-IfFalse:
exec-instr IfFalse i P h stk loc C0 M0 pc ics frs sh =
  (let pc' = if hd stk = Bool False then nat(int pc+i) else pc+1
  in (None, h, (tl stk, loc, C0, M0, pc', ics)#frs, sh))

| exec-instr-CmpEq:

```

```
exec-instr CmpEq P h stk loc C0 M0 pc ics frs sh =
  (None, h, (Bool (hd (tl stk) = hd stk) # tl (tl stk), loc, C0, M0, Suc pc, ics)#frs, sh)
```

```
| exec-instr-Goto:
exec-instr (Goto i) P h stk loc C0 M0 pc ics frs sh =
  (None, h, (stk, loc, C0, M0, nat(int pc+i), ics)#frs, sh)
```

```
| exec-instr-Throw:
exec-instr Throw P h stk loc C0 M0 pc ics frs sh =
  (let xp' = if hd stk = Null then [addr-of-sys-xcpt NullPointer]
    else [the-Addr(hd stk)]
   in (xp', h, (stk, loc, C0, M0, pc, ics)#frs, sh))
```

Given a preallocated heap, a thrown exception is either a system exception or thrown directly by *Throw*.

lemma *exec-instr-xcpts*:

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

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

shows $i = (\text{JVMinstructions.Throw}) \vee a \in \{a. \exists x \in \text{sys-xcpts}. a = \text{addr-of-sys-xcpt } x\}$

using *assms*

proof(*cases i*)

case (*New C1*) **then show** *?thesis using assms*

proof(*cases sh C1*)

case (*Some a*)

then obtain *sfs i where sfsi: a = (sfs,i) by*(*cases a*)

then show *?thesis using Some New assms*

proof(*cases i*) **qed**(*cases ics', auto*)+

qed(*cases ics', auto*)

next

case (*Getfield F1 C1*)

obtain *D' b t where field: field P C1 F1 = (D',b,t) by*(*cases field P C1 F1*)

obtain *D fs where addr: the (h (the-Addr (hd stk))) = (D,fs) by*(*cases the (h (the-Addr (hd stk)))*)

show *?thesis using addr field Getfield assms*

proof(*cases hd stk = Null*)

case *nNull:False* **then show** *?thesis using addr field Getfield assms*

proof(*cases* $\nexists t \ b. P \vdash (\text{cname-of } h \ (\text{the-Addr} \ (\text{hd} \ \text{stk}))) \ \text{has } F1, b:t \ \text{in } C1$)

case *exists:False* **show** *?thesis*

proof(*cases fst(snd(field P C1 F1))*)

case *Static*

then show *?thesis using exists nNull addr field Getfield assms*

by(*auto simp: sys-xcpts-def split-beta split: staticb.splits*)

next

case *NonStatic*

then show *?thesis using exists nNull addr field Getfield assms*

by(*auto simp: sys-xcpts-def split-beta split: staticb.splits*)

qed

qed(*auto*)

qed(*auto*)

next

case (*Getstatic C1 F1 D1*)

obtain *D' b t where field: field P D1 F1 = (D',b,t) by*(*cases field P D1 F1*)

show *?thesis using field Getstatic assms*

proof(*cases* $\nexists t \ b. P \vdash C1 \ \text{has } F1, b:t \ \text{in } D1$)

case *exists:False* **then show** *?thesis using field Getstatic assms*

```

    proof(cases fst(snd(field P D1 F1)))
      case Static
        then obtain sfs i where the(sh D') = (sfs, i) by(cases the(sh D'))
        then show ?thesis using exists field Static Getstatic assms by(cases ics'; cases i, auto)
      qed(auto)
    qed(auto)
  next
  case (Putfield F1 C1)
  let ?r = hd(tl stk)
  obtain D' b t where field: field P C1 F1 = (D',b,t) by(cases field P C1 F1)
  obtain D fs where addr: the (h (the-Addr ?r)) = (D,fs)
  by(cases the (h (the-Addr ?r)))
  show ?thesis using addr field Putfield assms
  proof(cases ?r = Null)
    case nNull:False then show ?thesis using addr field Putfield assms
    proof(cases  $\nexists t b. P \vdash (\text{cname-of } h \text{ (the-Addr ?r)}) \text{ has } F1, b:t \text{ in } C1$ )
      case exists:False show ?thesis
      proof(cases b)
        case Static
          then show ?thesis using exists nNull addr field Putfield assms
          by(auto simp: sys-xcpts-def split-beta split: staticb.splits)
        next
        case NonStatic
          then show ?thesis using exists nNull addr field Putfield assms
          by(auto simp: sys-xcpts-def split-beta split: staticb.splits)
      qed
    qed(auto)
  qed(auto)
  next
  case (Putstatic C1 F1 D1)
  obtain D' b t where field: field P D1 F1 = (D',b,t) by(cases field P D1 F1)
  show ?thesis using field Putstatic assms
  proof(cases  $\nexists t b. P \vdash C1 \text{ has } F1, b:t \text{ in } D1$ )
    case exists:False then show ?thesis using field Putstatic assms
    proof(cases b)
      case Static
        then obtain sfs i where the(sh D') = (sfs, i) by(cases the(sh D'))
        then show ?thesis using exists field Static Putstatic assms by(cases ics'; cases i, auto)
      qed(auto)
    qed(auto)
  next
  case (Checkcast C1) then show ?thesis using assms by(cases cast-ok P C1 h (hd stk), auto)
  next
  case (Invoke M n)
  let ?r = stk!n
  let ?C = cname-of h (the-Addr ?r)
  show ?thesis using Invoke assms
  proof(cases ?r = Null)
    case nNull:False then show ?thesis using Invoke assms
    proof(cases  $\neg(\exists Ts T m D b. P \vdash ?C \text{ sees } M, b:Ts \rightarrow T = m \text{ in } D)$ )
      case exists:False then show ?thesis using nNull Invoke assms
      proof(cases fst(snd(method P ?C M)))
        case Static
          then show ?thesis using exists nNull Invoke assms

```

```

    by(auto simp: sys-xcpts-def split-beta split: staticb.splits)
  next
  case NonStatic
  then show ?thesis using exists nNull Invoke assms
    by(auto simp: sys-xcpts-def split-beta split: staticb.splits)
  qed
  qed(auto)
  qed(auto)
next
case (Invokestatic C1 M n)
show ?thesis using Invokestatic assms
proof(cases ¬(∃ Ts T m D b. P ⊢ C1 sees M,b:Ts → T = m in D))
  case exists:False then show ?thesis using Invokestatic assms
  proof(cases fst(snd(method P C1 M)))
    case Static
    then obtain sfs i where the(sh (fst(method P C1 M))) = (sfs, i)
      by(cases the(sh (fst(method P C1 M))))
    then show ?thesis using exists Static Invokestatic assms
      by(auto split: init-call-status.splits init-state.splits)
    qed(auto)
  qed(auto)
next
case Return then show ?thesis using assms
proof(cases frs)
  case (Cons f frs') then show ?thesis using Return assms
    by(cases f, cases method P C M, cases M=clinit, auto)
  qed(auto)
next
case (IfFalse x1γ) then show ?thesis using assms
proof(cases hd stk)
  case (Bool b) then show ?thesis using IfFalse assms by(cases b, auto)
  qed(auto)
qed(auto)

```

lemma exec-instr-prealloc-pres:

assumes preallocated h

and exec-instr i P h stk loc C₀ M₀ pc ics frs sh = (xp',h',frs',sh')

shows preallocated h'

using assms

proof(cases i)

case (New C1)

then obtain sfs i where sfsi: the(sh C1) = (sfs,i) by(cases the(sh C1))

then show ?thesis using preallocated-new[of h] New assms

by(cases blank P C1, auto dest: new-Addr-SomeD split: init-call-status.splits init-state.splits)

next

case (Getfield F1 C1) then show ?thesis using assms

by(cases the (h (the-Addr (hd stk))), cases field P C1 F1, auto)

next

case (Getstatic C1 F1 D1)

then obtain sfs i where sfsi: the(sh (fst (field P D1 F1))) = (sfs, i)

by(cases the(sh (fst (field P D1 F1))))

then show ?thesis using Getstatic assms

by(cases field P D1 F1, auto split: init-call-status.splits init-state.splits)

next

```

    case (Putfield F1 C1) then show ?thesis using preallocated-new preallocated-upd-obj assms
      by(cases the (h (the-Addr (hd (tl stk))))), cases field P C1 F1, auto, metis option.collapse)
  next
    case (Putstatic C1 F1 D1)
    then obtain sfs i where sfsi: the(sh (fst (field P D1 F1))) = (sfs, i)
      by(cases the(sh (fst (field P D1 F1))))
    then show ?thesis using Putstatic assms
      by(cases field P D1 F1, auto split: init-call-status.splits init-state.splits)
  next
    case (Checkcast C1)
    then show ?thesis using assms
    proof(cases hd stk = Null)
      case False then show ?thesis
        using Checkcast assms
        by(cases P ⊢ cname-of h (the-Addr (hd stk)) ≤* C1, auto simp: cast-ok-def)
    qed(simp add: cast-ok-def)
  next
    case (Invoke M n)
    then show ?thesis using assms by(cases method P (cname-of h (the-Addr (stk ! n))) M, auto)
  next
    case (Invokestatic C1 M n)
    show ?thesis
    proof(cases sh (fst (method P C1 M)))
      case None then show ?thesis using Invokestatic assms
        by(cases method P C1 M, auto split: init-call-status.splits)
    next
      case (Some a)
      then obtain sfs i where sfsi: a = (sfs, i) by(cases a)
      then show ?thesis using Some Invokestatic assms
      proof(cases i) qed(cases method P C1 M, auto split: init-call-status.splits)+
    qed
  next
    case Return
    then show ?thesis using assms by(cases method P C0 M0, auto simp: split-beta split: list.splits)
  next
    case (IfFalse x1γ) then show ?thesis using assms by(auto split: val.splits bool.splits)
  next
    case Throw then show ?thesis using assms by(auto split: val.splits)
  qed(auto)

end

```

2.5 Program Execution in the JVM in full small step style

```

theory JVMExec
imports JVMEExecInstr
begin

```

abbreviation

```

instrs-of :: jvm-prog ⇒ cname ⇒ mname ⇒ instr list where
instrs-of P C M == fst(snd(snd(snd(snd(snd(snd(method P C M)))))))

```

```

fun curr-instr :: jvm-prog ⇒ frame ⇒ instr where

```

$curr\text{-}instr\ P\ (stk,loc,C,M,pc,ics) = instrs\text{-}of\ P\ C\ M\ !\ pc$

— execution of single step of the initialization procedure

fun $exec\text{-}Calling :: [cname, cname\ list, jvm\text{-}prog, heap, val\ list, val\ list,$
 $cname, mname, pc, frame\ list, sheap] \Rightarrow jvm\text{-}state$

where

$exec\text{-}Calling\ C\ Cs\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
 $(case\ sh\ C\ of$
 $\quad None \Rightarrow (None, h, (stk, loc, C_0, M_0, pc, Calling\ C\ Cs)\#frs, sh(C := Some\ (sblank\ P\ C,$
 $Prepared)))$
 $\quad | Some\ (obj, iflag) \Rightarrow$
 $\quad\quad (case\ iflag\ of$
 $\quad\quad\quad Done \Rightarrow (None, h, (stk, loc, C_0, M_0, pc, Called\ Cs)\#frs, sh)$
 $\quad\quad\quad | Processing \Rightarrow (None, h, (stk, loc, C_0, M_0, pc, Called\ Cs)\#frs, sh)$
 $\quad\quad\quad | Error \Rightarrow (None, h, (stk, loc, C_0, M_0, pc,$
 $\quad\quad\quad\quad Throwing\ Cs\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NoClassDefFoundError))\#frs, sh)$
 $\quad\quad\quad | Prepared \Rightarrow$
 $\quad\quad\quad\quad let\ sh' = sh(C := Some(fst(the(sh\ C)), Processing));$
 $\quad\quad\quad\quad\quad D = fst(the(class\ P\ C))$
 $\quad\quad\quad\quad\quad in\ if\ C = Object$
 $\quad\quad\quad\quad\quad\quad then\ (None, h, (stk, loc, C_0, M_0, pc, Called\ (C\#Cs))\#frs, sh')$
 $\quad\quad\quad\quad\quad\quad else\ (None, h, (stk, loc, C_0, M_0, pc, Calling\ D\ (C\#Cs))\#frs, sh')$
 $\quad\quad\quad)$
 $\quad)$

— single step of execution without error handling

fun $exec\text{-}step :: [jvm\text{-}prog, heap, val\ list, val\ list,$
 $cname, mname, pc, init\text{-}call\text{-}status, frame\ list, sheap] \Rightarrow jvm\text{-}state$

where

$exec\text{-}step\ P\ h\ stk\ loc\ C\ M\ pc\ (Calling\ C'\ Cs)\ frs\ sh$
 $= exec\text{-}Calling\ C'\ Cs\ P\ h\ stk\ loc\ C\ M\ pc\ frs\ sh\ |$
 $exec\text{-}step\ P\ h\ stk\ loc\ C\ M\ pc\ (Called\ (C'\#Cs))\ frs\ sh$
 $= (None, h, create\text{-}init\text{-}frame\ P\ C'\#(stk, loc, C, M, pc, Called\ Cs)\#frs, sh) |$
 $exec\text{-}step\ P\ h\ stk\ loc\ C\ M\ pc\ (Throwing\ (C'\#Cs)\ a)\ frs\ sh$
 $= (None, h, (stk,loc,C,M,pc,Throwing\ Cs\ a)\#frs, sh(C' := Some(fst(the(sh\ C')), Error))) |$
 $exec\text{-}step\ P\ h\ stk\ loc\ C\ M\ pc\ (Throwing\ []\ a)\ frs\ sh$
 $= ([a], h, (stk,loc,C,M,pc,No\text{-}ics)\#frs, sh) |$
 $exec\text{-}step\ P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh$
 $= exec\text{-}instr\ (instrs\text{-}of\ P\ C\ M\ !\ pc)\ P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh$

— execution including error handling

fun $exec :: jvm\text{-}prog \times jvm\text{-}state \Rightarrow jvm\text{-}state\ option$ — single step execution **where**

$exec\ (P, None, h, (stk,loc,C,M,pc,ics)\#frs, sh) =$
 $(let\ (xp', h', frs', sh') = exec\text{-}step\ P\ h\ stk\ loc\ C\ M\ pc\ ics\ frs\ sh$
 $\quad in\ case\ xp'\ of$
 $\quad\quad None \Rightarrow Some\ (None, h', frs', sh')$
 $\quad\quad | Some\ x \Rightarrow Some\ (find\text{-}handler\ P\ x\ h\ ((stk,loc,C,M,pc,ics)\#frs)\ sh)$
 $\quad)$
 $| exec - = None$

— relational view

inductive-set

$exec\text{-}1 :: jvm\text{-}prog \Rightarrow (jvm\text{-}state \times jvm\text{-}state)\ set$

and $exec\text{-}1' :: jvm\text{-}prog \Rightarrow jvm\text{-}state \Rightarrow jvm\text{-}state \Rightarrow bool$

$(\langle - \vdash / - -jvm \rightarrow_1 / - \rangle [61,61,61] 60)$
for $P :: jvm\text{-prog}$
where
 $P \vdash \sigma -jvm \rightarrow_1 \sigma' \equiv (\sigma, \sigma') \in exec\text{-}1 P$
 $| exec\text{-}1I: exec (P, \sigma) = Some \sigma' \implies P \vdash \sigma -jvm \rightarrow_1 \sigma'$
— reflexive transitive closure:
definition $exec\text{-}all :: jvm\text{-prog} \Rightarrow jvm\text{-state} \Rightarrow jvm\text{-state} \Rightarrow bool$
 $(\langle - \vdash / - -jvm \rightarrow / - \rangle [61,61,61] 60)$ **where**
 $exec\text{-}all\text{-}def1: P \vdash \sigma -jvm \rightarrow \sigma' \longleftrightarrow (\sigma, \sigma') \in (exec\text{-}1 P)^*$
notation (*ASCII*)
 $exec\text{-}all (\langle - \vdash / - -jvm \rightarrow / - \rangle [61,61,61] 60)$

lemma $exec\text{-}1\text{-}eq$:
 $exec\text{-}1 P = \{(\sigma, \sigma'). exec (P, \sigma) = Some \sigma'\}$
lemma $exec\text{-}1\text{-}iff$:
 $P \vdash \sigma -jvm \rightarrow_1 \sigma' = (exec (P, \sigma) = Some \sigma')$
lemma $exec\text{-}all\text{-}def$:
 $P \vdash \sigma -jvm \rightarrow \sigma' = ((\sigma, \sigma') \in \{(\sigma, \sigma'). exec (P, \sigma) = Some \sigma'\}^*)$
lemma $jvm\text{-refl}[iff]$: $P \vdash \sigma -jvm \rightarrow \sigma$
lemma $jvm\text{-trans}[trans]$:
 $\llbracket P \vdash \sigma -jvm \rightarrow \sigma'; P \vdash \sigma' -jvm \rightarrow \sigma'' \rrbracket \implies P \vdash \sigma -jvm \rightarrow \sigma''$
lemma $jvm\text{-one-step}1[trans]$:
 $\llbracket P \vdash \sigma -jvm \rightarrow_1 \sigma'; P \vdash \sigma' -jvm \rightarrow \sigma'' \rrbracket \implies P \vdash \sigma -jvm \rightarrow \sigma''$
lemma $jvm\text{-one-step}2[trans]$:
 $\llbracket P \vdash \sigma -jvm \rightarrow \sigma'; P \vdash \sigma' -jvm \rightarrow_1 \sigma'' \rrbracket \implies P \vdash \sigma -jvm \rightarrow \sigma''$
lemma $exec\text{-}all\text{-}conf$:
 $\llbracket P \vdash \sigma -jvm \rightarrow \sigma'; P \vdash \sigma -jvm \rightarrow \sigma'' \rrbracket$
 $\implies P \vdash \sigma' -jvm \rightarrow \sigma'' \vee P \vdash \sigma'' -jvm \rightarrow \sigma'$
lemma $exec\text{-}1\text{-}exec\text{-}all\text{-}conf$:
 $\llbracket exec (P, \sigma) = Some \sigma'; P \vdash \sigma -jvm \rightarrow \sigma''; \sigma \neq \sigma'' \rrbracket$
 $\implies P \vdash \sigma' -jvm \rightarrow \sigma''$
by(*auto elim: converse-rtranclE simp: exec-1-eq exec-all-def*)

lemma $exec\text{-}all\text{-}finalD$: $P \vdash (x, h, [], sh) -jvm \rightarrow \sigma \implies \sigma = (x, h, [], sh)$
lemma $exec\text{-}all\text{-}deterministic$:
 $\llbracket P \vdash \sigma -jvm \rightarrow (x, h, [], sh); P \vdash \sigma -jvm \rightarrow \sigma' \rrbracket \implies P \vdash \sigma' -jvm \rightarrow (x, h, [], sh)$

2.5.1 Preservation of preallocated

lemma $exec\text{-}Calling\text{-}prealloc\text{-}pres$:
assumes $preallocated h$
and $exec\text{-}Calling C Cs P h stk loc C_0 M_0 pc frs sh = (xp', h', frs', sh')$
shows $preallocated h'$
using $assms$
proof(*cases sh C*)
case (*Some a*)
then obtain $sfs i$ **where** $sfsi:a = (sfs, i)$ **by**(*cases a*)
then show *?thesis* **using** *Some assms*
proof(*cases i*)
case *Prepared*
then show *?thesis* **using** $sfsi$ *Some assms* **by**(*cases method P C clinit, auto split: if-split-asm*)

```

next
  case Error
  then show ?thesis using sfsi Some assms by(cases method P C clinit, auto)
qed(auto)
qed(auto)

```

lemma *exec-step-prealloc-pres:*

assumes *pre: preallocated h*

and *exec-step P h stk loc C M pc ics frs sh = (xp',h',frs',sh')*

shows *preallocated h'*

proof(*cases ics*)

case *No-ics*

then show ?thesis **using** *exec-instr-prealloc-pres assms* **by** *auto*

next

case *Calling*

then show ?thesis **using** *exec-Calling-prealloc-pres assms* **by** *auto*

next

case (*Called Cs*)

then show ?thesis **using** *exec-instr-prealloc-pres assms* **by**(*cases Cs, auto*)

next

case (*Throwing Cs a*)

then show ?thesis **using** *assms* **by**(*cases Cs, auto*)

qed

lemma *exec-prealloc-pres:*

assumes *pre: preallocated h*

and *exec (P, xp, h, frs, sh) = Some(xp',h',frs',sh')*

shows *preallocated h'*

using *assms*

proof(*cases $\exists x. xp = [x] \vee frs = []$*)

case *False*

then obtain *f1 frs1* **where** *frs: frs = f1 # frs1* **by**(*cases frs, simp+*)

then obtain *stk1 loc1 C1 M1 pc1 ics1* **where** *f1: f1 = (stk1,loc1,C1,M1,pc1,ics1)* **by**(*cases f1*)

let *?i = instrs-of P C1 M1 ! pc1*

obtain *xp2 h2 frs2 sh2*

where *exec-step: exec-step P h stk1 loc1 C1 M1 pc1 ics1 frs1 sh = (xp2,h2,frs2,sh2)*

by(*cases exec-step P h stk1 loc1 C1 M1 pc1 ics1 frs1 sh*)

then show ?thesis **using** *exec-step-prealloc-pres[OF pre exec-step]* *f1 frs False assms*

proof(*cases xp2*)

case (*Some a*)

show ?thesis

using *find-handler-prealloc-pres[OF pre, where a=a]*

exec-step-prealloc-pres[OF pre]

exec-step f1 frs Some False assms

by(*auto split: bool.splits init-call-status.splits list.splits*)

qed(*auto split: init-call-status.splits*)

qed(*auto*)

2.5.2 Start state

The *Start* class is defined based on a given initial class and method. It has two methods: one that calls the initial method in the initial class, which is called by the starting program, and *clinit*, as required for the class to be well-formed.

definition *start-method* :: *cname* \Rightarrow *mname* \Rightarrow *jvm-method mdecl* **where**
start-method *C M* = (*start-m*, *Static*, [], *Void*, (1,0,[*Invokestatic C M 0,Return*],[]))

abbreviation *start-clinit* :: *jvm-method mdecl* **where**
start-clinit \equiv (*clinit*, *Static*, [], *Void*, (1,0,[*Push Unit,Return*],[]))

definition *start-class* :: *cname* \Rightarrow *mname* \Rightarrow *jvm-method cdecl* **where**
start-class *C M* = (*Start*, *Object*, [], [*start-method C M*, *start-clinit*])

The start configuration of the JVM in program *P*: in the start heap, we call the “start” method of the “Start”; this method performs *Invokestatic* on the class and method given to create the start program’s *Start* class. This allows the initialization procedure to be called on the initial class in a natural way before the initial method is performed. There is no *this* pointer of the frame as *start* is *Static*. The start sheap has every class pre-prepared; this decision is not necessary. The starting program includes the added *Start* class, given a method *M* of class *C*, added to program *P*.

definition *start-state* :: *jvm-prog* \Rightarrow *jvm-state* **where**
start-state *P* = (*None*, *start-heap P*, [([], [], *Start*, *start-m*, 0, *No-ics*)], *start-sheap*)

abbreviation *start-prog* :: *jvm-prog* \Rightarrow *cname* \Rightarrow *mname* \Rightarrow *jvm-prog* **where**
start-prog *P C M* \equiv *start-class C M* # *P*

end

2.6 A Defensive JVM

theory *JVMDefensive*
imports *JVMExec ../Common/Conform*
begin

Extend the state space by one element indicating a type error (or other abnormal termination)

datatype 'a *type-error* = *TypeError* | *Normal 'a*

fun *is-Addr* :: *val* \Rightarrow *bool* **where**
is-Addr (*Addr a*) \longleftrightarrow *True*
| *is-Addr* *v* \longleftrightarrow *False*

fun *is-Intg* :: *val* \Rightarrow *bool* **where**
is-Intg (*Intg i*) \longleftrightarrow *True*
| *is-Intg* *v* \longleftrightarrow *False*

fun *is-Bool* :: *val* \Rightarrow *bool* **where**
is-Bool (*Bool b*) \longleftrightarrow *True*
| *is-Bool* *v* \longleftrightarrow *False*

definition *is-Ref* :: *val* \Rightarrow *bool* **where**
is-Ref *v* \longleftrightarrow *v* = *Null* \vee *is-Addr v*

primrec *check-instr* :: [*instr*, *jvm-prog*, *heap*, *val list*, *val list*,
cname, *mname*, *pc*, *frame list*, *sheap*] \Rightarrow *bool* **where**
check-instr-Load:
check-instr (*Load n*) *P h stk loc C M₀ pc frs sh* =
(*n* < *length loc*)

| *check-instr-Store*:

check-instr (Store n) $P h stk loc C_0 M_0 pc frs sh =$
 $(0 < length\ stk \wedge n < length\ loc)$

| *check-instr-Push*:
check-instr (Push v) $P h stk loc C_0 M_0 pc frs sh =$
 $(\neg is-Addr\ v)$

| *check-instr-New*:
check-instr (New C) $P h stk loc C_0 M_0 pc frs sh =$
 $is-class\ P\ C$

| *check-instr-Getfield*:
check-instr (Getfield $F\ C$) $P h stk loc C_0 M_0 pc frs sh =$
 $(0 < length\ stk \wedge (\exists C' T. P \vdash C\ sees\ F, NonStatic:T\ in\ C') \wedge$
 $(let\ (C', b, T) = field\ P\ C\ F; ref = hd\ stk\ in$
 $C' = C \wedge is-Ref\ ref \wedge (ref \neq Null \longrightarrow$
 $h\ (the-Addr\ ref) \neq None \wedge$
 $(let\ (D, vs) = the\ (h\ (the-Addr\ ref))\ in$
 $P \vdash D \leq^* C \wedge vs\ (F, C) \neq None \wedge P, h \vdash the\ (vs\ (F, C))\ :\leq\ T))))$

| *check-instr-Getstatic*:
check-instr (Getstatic $C\ F\ D$) $P h stk loc C_0 M_0 pc frs sh =$
 $((\exists T. P \vdash C\ sees\ F, Static:T\ in\ D) \wedge$
 $(let\ (C', b, T) = field\ P\ C\ F\ in$
 $C' = D \wedge (sh\ D \neq None \longrightarrow$
 $(let\ (sfs, i) = the\ (sh\ D)\ in$
 $sfs\ F \neq None \wedge P, h \vdash the\ (sfs\ F)\ :\leq\ T))))$

| *check-instr-Putfield*:
check-instr (Putfield $F\ C$) $P h stk loc C_0 M_0 pc frs sh =$
 $(1 < length\ stk \wedge (\exists C' T. P \vdash C\ sees\ F, NonStatic:T\ in\ C') \wedge$
 $(let\ (C', b, T) = field\ P\ C\ F; v = hd\ stk; ref = hd\ (tl\ stk)\ in$
 $C' = C \wedge is-Ref\ ref \wedge (ref \neq Null \longrightarrow$
 $h\ (the-Addr\ ref) \neq None \wedge$
 $(let\ D = fst\ (the\ (h\ (the-Addr\ ref)))\ in$
 $P \vdash D \leq^* C \wedge P, h \vdash v\ :\leq\ T))))$

| *check-instr-Putstatic*:
check-instr (Putstatic $C\ F\ D$) $P h stk loc C_0 M_0 pc frs sh =$
 $(0 < length\ stk \wedge (\exists T. P \vdash C\ sees\ F, Static:T\ in\ D) \wedge$
 $(let\ (C', b, T) = field\ P\ C\ F; v = hd\ stk\ in$
 $C' = D \wedge P, h \vdash v\ :\leq\ T))$

| *check-instr-Checkcast*:
check-instr (Checkcast C) $P h stk loc C_0 M_0 pc frs sh =$
 $(0 < length\ stk \wedge is-class\ P\ C \wedge is-Ref\ (hd\ stk))$

| *check-instr-Invoke*:
check-instr (Invoke $M\ n$) $P h stk loc C_0 M_0 pc frs sh =$
 $(n < length\ stk \wedge is-Ref\ (stk!n) \wedge$
 $(stk!n \neq Null \longrightarrow$
 $(let\ a = the-Addr\ (stk!n);$
 $C = cname-of\ h\ a;$
 $Ts = fst\ (snd\ (snd\ (method\ P\ C\ M))))$

- $in\ h\ a \neq None \wedge P \vdash C\ has\ M, NonStatic \wedge$
 $P, h \vdash rev\ (take\ n\ stk)\ [:\leq]\ Ts))$
- | *check-instr-Invokestatic*:
 $check_instr\ (Invokestatic\ C\ M\ n)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
 $(n \leq length\ stk \wedge$
 $(let\ Ts = fst\ (snd\ (snd\ (method\ P\ C\ M)))$
 $in\ P \vdash C\ has\ M, Static \wedge$
 $P, h \vdash rev\ (take\ n\ stk)\ [:\leq]\ Ts))$
- | *check-instr-Return*:
 $check_instr\ Return\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
 $(case\ (M_0 = clinit)\ of\ False \Rightarrow (0 < length\ stk \wedge ((0 < length\ frs) \longrightarrow$
 $(\exists\ b. P \vdash C_0\ has\ M_0, b) \wedge$
 $(let\ v = hd\ stk;$
 $T = fst\ (snd\ (snd\ (snd\ (method\ P\ C_0\ M_0))))$
 $in\ P, h \vdash v :\leq T))$
 $| True \Rightarrow P \vdash C_0\ has\ M_0, Static)$
- | *check-instr-Pop*:
 $check_instr\ Pop\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
 $(0 < length\ stk)$
- | *check-instr-IAdd*:
 $check_instr\ IAdd\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
 $(1 < length\ stk \wedge is-Intg\ (hd\ stk) \wedge is-Intg\ (hd\ (tl\ stk)))$
- | *check-instr-IfFalse*:
 $check_instr\ (IfFalse\ b)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
 $(0 < length\ stk \wedge is-Bool\ (hd\ stk) \wedge 0 \leq int\ pc+b)$
- | *check-instr-CmpEq*:
 $check_instr\ CmpEq\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
 $(1 < length\ stk)$
- | *check-instr-Goto*:
 $check_instr\ (Goto\ b)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
 $(0 \leq int\ pc+b)$
- | *check-instr-Throw*:
 $check_instr\ Throw\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs\ sh =$
 $(0 < length\ stk \wedge is-Ref\ (hd\ stk))$

definition *check* :: *jvm-prog* \Rightarrow *jvm-state* \Rightarrow *bool* **where**

$check\ P\ \sigma = (let\ (xcpt, h, frs, sh) = \sigma\ in$
 $(case\ frs\ of\ [] \Rightarrow True\ | (stk, loc, C, M, pc, ics) \# frs' \Rightarrow$
 $\exists\ b. P \vdash C\ has\ M, b \wedge$
 $(let\ (C', b, Ts, T, mxs, mxl_0, ins, xt) = method\ P\ C\ M; i = ins!pc\ in$
 $pc < size\ ins \wedge size\ stk \leq mxs \wedge$
 $check_instr\ i\ P\ h\ stk\ loc\ C\ M\ pc\ frs'\ sh)))$

definition *exec-d* :: *jvm-prog* \Rightarrow *jvm-state* \Rightarrow *jvm-state* *option* *type-error* **where**

$exec-d\ P\ \sigma = (if\ check\ P\ \sigma\ then\ Normal\ (exec\ (P, \sigma))\ else\ TypeError)$

inductive-set

$exec-1-d :: jvm-prog \Rightarrow (jvm-state\ type-error \times jvm-state\ type-error)\ set$
and $exec-1-d' :: jvm-prog \Rightarrow jvm-state\ type-error \Rightarrow jvm-state\ type-error \Rightarrow bool$
 $(\langle - \vdash - \rangle - jvmd \rightarrow_1 \rightarrow [61,61,61]60)$

for $P :: jvm-prog$

where

$P \vdash \sigma -jvmd \rightarrow_1 \sigma' \equiv (\sigma, \sigma') \in exec-1-d\ P$
 $| exec-1-d-ErrorI: exec-d\ P\ \sigma = TypeError \Longrightarrow P \vdash Normal\ \sigma -jvmd \rightarrow_1\ TypeError$
 $| exec-1-d-NormalI: exec-d\ P\ \sigma = Normal\ (Some\ \sigma') \Longrightarrow P \vdash Normal\ \sigma -jvmd \rightarrow_1\ Normal\ \sigma'$

— reflexive transitive closure:

definition $exec-all-d :: jvm-prog \Rightarrow jvm-state\ type-error \Rightarrow jvm-state\ type-error \Rightarrow bool$

$(\langle - \vdash - \rangle - jvmd \rightarrow \rightarrow [61,61,61]60)$ **where**

$exec-all-d-def1: P \vdash \sigma -jvmd \rightarrow \sigma' \longleftrightarrow (\sigma, \sigma') \in (exec-1-d\ P)^*$

notation (ASCII)

$exec-all-d\ (\langle - | - \rangle - jvmd \rightarrow \rightarrow [61,61,61]60)$

lemma $exec-1-d-eq$:

$exec-1-d\ P = \{(s,t). \exists \sigma. s = Normal\ \sigma \wedge t = TypeError \wedge exec-d\ P\ \sigma = TypeError\} \cup$
 $\{(s,t). \exists \sigma\ \sigma'. s = Normal\ \sigma \wedge t = Normal\ \sigma' \wedge exec-d\ P\ \sigma = Normal\ (Some\ \sigma')\}$

by ($auto\ elim!$; $exec-1-d.cases\ intro!$; $exec-1-d.intros$)

declare $split-paired-All$ [$simp\ del$]

declare $split-paired-Ex$ [$simp\ del$]

lemma $if-neq$ [$dest!$]:

$(if\ P\ then\ A\ else\ B) \neq B \Longrightarrow P$
by ($cases\ P, auto$)

lemma $exec-d-no-errorI$ [$intro$]:

$check\ P\ \sigma \Longrightarrow exec-d\ P\ \sigma \neq TypeError$
by ($unfold\ exec-d-def$) $simp$

theorem $no-type-error-commutes$:

$exec-d\ P\ \sigma \neq TypeError \Longrightarrow exec-d\ P\ \sigma = Normal\ (exec\ (P, \sigma))$
by ($unfold\ exec-d-def, auto$)

lemma $defensive-imp-aggressive$:

$P \vdash (Normal\ \sigma) -jvmd \rightarrow (Normal\ \sigma') \Longrightarrow P \vdash \sigma -jvm \rightarrow \sigma'$
end

2.7 The Jinja Type System as a Semilattice

theory $SemiType$

imports $../Common/WellForm\ Jinja.Semilattices$

begin

definition $super :: 'a\ prog \Rightarrow cname \Rightarrow cname$

where $super\ P\ C \equiv fst\ (the\ (class\ P\ C))$

lemma *superI*:

$(C,D) \in subcls1\ P \implies super\ P\ C = D$
by (*unfold super-def*) (*auto dest: subcls1D*)

primrec *the-Class* :: $ty \Rightarrow cname$

where

$the-Class\ (Class\ C) = C$

definition *sup* :: $'c\ prog \Rightarrow ty \Rightarrow ty \Rightarrow ty\ err$

where

$sup\ P\ T_1\ T_2 \equiv$
if is-refT $T_1 \wedge is-refT\ T_2$ *then*
OK (*if* $T_1 = NT$ *then* T_2 *else*
if $T_2 = NT$ *then* T_1 *else*
 $(Class\ (exec-lub\ (subcls1\ P)\ (super\ P)\ (the-Class\ T_1)\ (the-Class\ T_2))))$
else
if $T_1 = T_2$ *then* *OK* T_1 *else* *Err*)

lemma *sup-def'*:

$sup\ P = (\lambda T_1\ T_2.$
if is-refT $T_1 \wedge is-refT\ T_2$ *then*
OK (*if* $T_1 = NT$ *then* T_2 *else*
if $T_2 = NT$ *then* T_1 *else*
 $(Class\ (exec-lub\ (subcls1\ P)\ (super\ P)\ (the-Class\ T_1)\ (the-Class\ T_2))))$
else
if $T_1 = T_2$ *then* *OK* T_1 *else* *Err*)
by (*simp add: sup-def fun-eq-iff*)

abbreviation

subtype :: $'c\ prog \Rightarrow ty \Rightarrow ty \Rightarrow bool$
where $subtype\ P \equiv widen\ P$

definition *esl* :: $'c\ prog \Rightarrow ty\ esl$

where

$esl\ P \equiv (types\ P,\ subtype\ P,\ sup\ P)$

lemma *is-class-is-subcls*:

$wf-prog\ m\ P \implies is-class\ P\ C = P \vdash C \preceq^* Object$

lemma *subcls-antisym*:

$\llbracket wf-prog\ m\ P; P \vdash C \preceq^* D; P \vdash D \preceq^* C \rrbracket \implies C = D$

lemma *widen-antisym*:

$\llbracket wf-prog\ m\ P; P \vdash T \leq U; P \vdash U \leq T \rrbracket \implies T = U$

lemma *order-widen* [*intro, simp*]:

$wf-prog\ m\ P \implies order\ (subtype\ P)\ (types\ P)$

lemma *NT-widen*:

$$P \vdash NT \leq T = (T = NT \vee (\exists C. T = \text{Class } C))$$

lemma *Class-widen2*: $P \vdash \text{Class } C \leq T = (\exists D. T = \text{Class } D \wedge P \vdash C \preceq^* D)$

lemma *wf-converse-subcls1-impl-acc-subtype*:

$$\text{wf } ((\text{subcls1 } P)^{\wedge -1}) \implies \text{acc } (\text{subtype } P)$$

lemma *wf-subtype-acc* [*intro*, *simp*]:

$$\text{wf-prog } \text{wf-mb } P \implies \text{acc } (\text{subtype } P)$$

lemma *exec-lub-refl* [*simp*]: $\text{exec-lub } r \ f \ T \ T = T$

lemma *closed-err-types*:

$$\text{wf-prog } \text{wf-mb } P \implies \text{closed } (\text{err } (\text{types } P)) \ (\text{lift2 } (\text{sup } P))$$

lemma *sup-subtype-greater*:

$$\llbracket \text{wf-prog } \text{wf-mb } P; \text{is-type } P \ t1; \text{is-type } P \ t2; \text{sup } P \ t1 \ t2 = \text{OK } s \rrbracket \\ \implies \text{subtype } P \ t1 \ s \wedge \text{subtype } P \ t2 \ s$$

lemma *sup-subtype-smallest*:

$$\llbracket \text{wf-prog } \text{wf-mb } P; \text{is-type } P \ a; \text{is-type } P \ b; \text{is-type } P \ c; \\ \text{subtype } P \ a \ c; \text{subtype } P \ b \ c; \text{sup } P \ a \ b = \text{OK } d \rrbracket \\ \implies \text{subtype } P \ d \ c$$

lemma *sup-exists*:

$$\llbracket \text{subtype } P \ a \ c; \text{subtype } P \ b \ c \rrbracket \implies \exists T. \text{sup } P \ a \ b = \text{OK } T$$

lemma *err-semilat-JType-esl*:

$$\text{wf-prog } \text{wf-mb } P \implies \text{err-semilat } (\text{esl } P)$$

end

2.8 The JVM Type System as Semilattice

theory *JVM-SemiType* **imports** *SemiType* **begin**

type-synonym $ty_l = \text{ty err list}$

type-synonym $ty_s = \text{ty list}$

type-synonym $ty_i = \text{ty}_s \times \text{ty}_l$

type-synonym $ty_i' = \text{ty}_i \ \text{option}$

type-synonym $ty_m = \text{ty}_i' \ \text{list}$

type-synonym $ty_P = \text{mname} \Rightarrow \text{cname} \Rightarrow \text{ty}_m$

definition $\text{stk-esl} :: 'c \ \text{prog} \Rightarrow \text{nat} \Rightarrow \text{ty}_s \ \text{esl}$

where

$$\text{stk-esl } P \ mxs \equiv \text{upto-esl } mxs \ (\text{SemiType.esl } P)$$

definition $\text{loc-sl} :: 'c \ \text{prog} \Rightarrow \text{nat} \Rightarrow \text{ty}_l \ \text{sl}$

where

$$\text{loc-sl } P \ mxl \equiv \text{Listn.sl } mxl \ (\text{Err.sl } (\text{SemiType.esl } P))$$

definition $\text{sl} :: 'c \ \text{prog} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{ty}_i' \ \text{err } \text{sl}$

where

$$\text{sl } P \ mxs \ mxl \equiv \\ \text{Err.sl}(\text{Opt.esl}(\text{Product.esl } (\text{stk-esl } P \ mxs) \ (\text{Err.esl}(\text{loc-sl } P \ mxl))))$$

definition $\text{states} :: 'c \ \text{prog} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{ty}_i' \ \text{err } \text{set}$

where $states\ P\ mxs\ mxl \equiv fst(sl\ P\ mxs\ mxl)$

definition $le :: 'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i' err\ ord$

where

$le\ P\ mxs\ mxl \equiv fst(snd(sl\ P\ mxs\ mxl))$

definition $sup :: 'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i' err\ binop$

where

$sup\ P\ mxs\ mxl \equiv snd(snd(sl\ P\ mxs\ mxl))$

definition $sup\text{-}ty\text{-}opt :: ['c\ prog, ty\ err, ty\ err] \Rightarrow bool$

$(\langle - \vdash - \leq_{\top} \rangle \rightarrow [71, 71, 71] 70)$

where

$sup\text{-}ty\text{-}opt\ P \equiv Err.le\ (subtype\ P)$

definition $sup\text{-}state :: ['c\ prog, ty_i, ty_i] \Rightarrow bool$

$(\langle - \vdash - \leq_i \rangle \rightarrow [71, 71, 71] 70)$

where

$sup\text{-}state\ P \equiv Product.le\ (Listn.le\ (subtype\ P))\ (Listn.le\ (sup\text{-}ty\text{-}opt\ P))$

definition $sup\text{-}state\text{-}opt :: ['c\ prog, ty_i', ty_i'] \Rightarrow bool$

$(\langle - \vdash - \leq'' \rangle \rightarrow [71, 71, 71] 70)$

where

$sup\text{-}state\text{-}opt\ P \equiv Opt.le\ (sup\text{-}state\ P)$

abbreviation

$sup\text{-}loc :: ['c\ prog, ty_i, ty_i] \Rightarrow bool\ (\langle - \vdash - [\leq_{\top}] \rangle \rightarrow [71, 71, 71] 70)$

where $P \vdash LT [\leq_{\top}] LT' \equiv list\text{-}all2\ (sup\text{-}ty\text{-}opt\ P)\ LT\ LT'$

notation (*ASCII*)

$sup\text{-}ty\text{-}opt\ (\langle - \vdash - \leq=T \rangle \rightarrow [71, 71, 71] 70)$ **and**

$sup\text{-}state\ (\langle - \vdash - \leq=i \rangle \rightarrow [71, 71, 71] 70)$ **and**

$sup\text{-}state\text{-}opt\ (\langle - \vdash - \leq=' \rangle \rightarrow [71, 71, 71] 70)$ **and**

$sup\text{-}loc\ (\langle - \vdash - [\leq=T] \rangle \rightarrow [71, 71, 71] 70)$

2.8.1 Unfolding

lemma *JVM-states-unfold*:

$states\ P\ mxs\ mxl \equiv err(opt((Union\ \{nlists\ n\ (types\ P)\ |\ n.\ n \leq mxs\ \}) \times nlists\ mxl\ (err(types\ P))))$

lemma *JVM-le-unfold*:

$le\ P\ m\ n \equiv$

$Err.le(Opt.le(Product.le(Listn.le(subtype\ P))(Listn.le(Err.le(subtype\ P))))))$

lemma *sl-def2*:

$JVM\text{-}SemiType.sl\ P\ mxs\ mxl \equiv$

$(states\ P\ mxs\ mxl, JVM\text{-}SemiType.le\ P\ mxs\ mxl, JVM\text{-}SemiType.sup\ P\ mxs\ mxl)$

lemma *JVM-le-conv*:

$le\ P\ m\ n\ (OK\ t1)\ (OK\ t2) = P \vdash t1 \leq' t2$

lemma *JVM-le-Err-conv*:

$le\ P\ m\ n = Err.le\ (sup\text{-}state\text{-}opt\ P)$

lemma *err-le-unfold* [*iff*]:

$Err.le\ r\ (OK\ a)\ (OK\ b) = r\ a\ b$

2.8.2 Semilattice

lemma *order-sup-state-opt'* [intro, simp]:

wf-prog wf-mb P \implies
 $\text{order } (\text{sup-state-opt } P) (\text{opt } ((\bigcup \{ \text{nlists } n (\text{types } P) \mid n. n \leq \text{mxs} \}) \times \text{nlists } (\text{Suc } (\text{length } Ts + \text{mxl}_0))) (\text{err } (\text{types } P))))$

2.9 Effect of Instructions on the State Type

theory *Effect*

imports *JVM-SemiType ../JVM/JVMExceptions*

begin

— FIXME

locale *prog* =

fixes *P* :: 'a *prog*

locale *jvm-method* = *prog* +

fixes *mxs* :: nat

fixes *mxl₀* :: nat

fixes *Ts* :: ty list

fixes *T_r* :: ty

fixes *is* :: instr list

fixes *xt* :: ex-table

fixes *mxl* :: nat

defines *mxl-def*: $\text{mxl} \equiv 1 + \text{size } Ts + \text{mxl}_0$

Program counter of successor instructions:

primrec *succs* :: instr \Rightarrow ty_{*i*} \Rightarrow pc \Rightarrow pc list **where**

succs (*Load idx*) τ pc = [pc+1]
| *succs* (*Store idx*) τ pc = [pc+1]
| *succs* (*Push v*) τ pc = [pc+1]
| *succs* (*Getfield F C*) τ pc = [pc+1]
| *succs* (*Getstatic C F D*) τ pc = [pc+1]
| *succs* (*Putfield F C*) τ pc = [pc+1]
| *succs* (*Putstatic C F D*) τ pc = [pc+1]
| *succs* (*New C*) τ pc = [pc+1]
| *succs* (*Checkcast C*) τ pc = [pc+1]
| *succs* *Pop* τ pc = [pc+1]
| *succs* *IAdd* τ pc = [pc+1]
| *succs* *CmpEq* τ pc = [pc+1]
| *succs-IfFalse*:
 succs (*IfFalse b*) τ pc = [pc+1, nat (int pc + b)]
| *succs-Goto*:
 succs (*Goto b*) τ pc = [nat (int pc + b)]
| *succs-Return*:
 succs *Return* τ pc = []
| *succs-Invoke*:
 succs (*Invoke M n*) τ pc = (if (fst τ)!n = NT then [] else [pc+1])
| *succs-Invokestatic*:
 succs (*Invokestatic C M n*) τ pc = [pc+1]
| *succs-Throw*:
 succs *Throw* τ pc = []

Effect of instruction on the state type:

fun *the-class* :: $ty \Rightarrow cname$ **where**

the-class (Class C) = C

fun eff_i :: $instr \times 'm\ prog \times ty_i \Rightarrow ty_i$ **where**

eff_i-Load:

eff_i (Load n , P , (ST , LT)) = ($ok\text{-}val$ (LT ! n) # ST , LT)

| *eff_i-Store*:

eff_i (Store n , P , (T # ST , LT)) = (ST , $LT[n:= OK\ T]$)

| *eff_i-Push*:

eff_i (Push v , P , (ST , LT)) = (the ($typeof\ v$) # ST , LT)

| *eff_i-Getfield*:

eff_i (Getfield $F\ C$, P , (T # ST , LT)) = (snd (snd ($field\ P\ C\ F$)) # ST , LT)

| *eff_i-Getstatic*:

eff_i (Getstatic $C\ F\ D$, P , (ST , LT)) = (snd (snd ($field\ P\ C\ F$)) # ST , LT)

| *eff_i-Putfield*:

eff_i (Putfield $F\ C$, P , (T_1 # T_2 # ST , LT)) = (ST , LT)

| *eff_i-Putstatic*:

eff_i (Putstatic $C\ F\ D$, P , (T # ST , LT)) = (ST , LT)

| *eff_i-New*:

eff_i (New C , P , (ST , LT)) = ($Class\ C$ # ST , LT)

| *eff_i-Checkcast*:

eff_i (Checkcast C , P , (T # ST , LT)) = ($Class\ C$ # ST , LT)

| *eff_i-Pop*:

eff_i (Pop, P , (T # ST , LT)) = (ST , LT)

| *eff_i-IAdd*:

eff_i (IAdd, P , (T_1 # T_2 # ST , LT)) = ($Integer$ # ST , LT)

| *eff_i-CmpEq*:

eff_i (CmpEq, P , (T_1 # T_2 # ST , LT)) = ($Boolean$ # ST , LT)

| *eff_i-IfFalse*:

eff_i (IfFalse b , P , (T_1 # ST , LT)) = (ST , LT)

| *eff_i-Invoke*:

eff_i (Invoke $M\ n$, P , (ST , LT)) =
 ($let\ C = the\text{-}class\ (ST!n); (D, b, Ts, T_r, m) = method\ P\ C\ M$
 in (T_r # $drop\ (n+1)\ ST$, LT))

| *eff_i-Invokestatic*:

eff_i (Invokestatic $C\ M\ n$, P , (ST , LT)) =
 ($let\ (D, b, Ts, T_r, m) = method\ P\ C\ M$
 in (T_r # $drop\ n\ ST$, LT))

| *eff_i-Goto*:

eff_i (Goto n , P , s) = s

fun *is-relevant-class* :: $instr \Rightarrow 'm\ prog \Rightarrow cname \Rightarrow bool$ **where**

rel-Getfield:

is-relevant-class (Getfield $F\ D$)
 = ($\lambda P\ C. P \vdash NullPointer \preceq^* C \vee P \vdash NoSuchFieldError \preceq^* C$
 $\vee P \vdash IncompatibleClassChangeError \preceq^* C$)

| *rel-Getstatic*:

is-relevant-class (Getstatic $C\ F\ D$)
 = ($\lambda P\ C. True$)

| *rel-Putfield*:

is-relevant-class (Putfield $F\ D$)
 = ($\lambda P\ C. P \vdash NullPointer \preceq^* C \vee P \vdash NoSuchFieldError \preceq^* C$
 $\vee P \vdash IncompatibleClassChangeError \preceq^* C$)

| *rel-Putstatic*:
 is-relevant-class (*Putstatic* *C F D*)
 = ($\lambda P C. \text{True}$)

| *rel-Checkcast*:
 is-relevant-class (*Checkcast* *D*) = ($\lambda P C. P \vdash \text{ClassCast} \preceq^* C$)

| *rel-New*:
 is-relevant-class (*New* *D*) = ($\lambda P C. \text{True}$)

| *rel-Throw*:
 is-relevant-class *Throw* = ($\lambda P C. \text{True}$)

| *rel-Invoke*:
 is-relevant-class (*Invoke* *M n*) = ($\lambda P C. \text{True}$)

| *rel-Invokestatic*:
 is-relevant-class (*Invokestatic* *C M n*) = ($\lambda P C. \text{True}$)

| *rel-default*:
 is-relevant-class *i* = ($\lambda P C. \text{False}$)

definition *is-relevant-entry* :: 'm prog \Rightarrow instr \Rightarrow pc \Rightarrow ex-entry \Rightarrow bool **where**
is-relevant-entry *P i pc e* \longleftrightarrow (let (*f,t,C,h,d*) = *e* in *is-relevant-class* *i P C* \wedge *pc* \in {*f..<t*})

definition *relevant-entries* :: 'm prog \Rightarrow instr \Rightarrow pc \Rightarrow ex-table \Rightarrow ex-table **where**
relevant-entries *P i pc* = filter (*is-relevant-entry* *P i pc*)

definition *xcpt-eff* :: instr \Rightarrow 'm prog \Rightarrow pc \Rightarrow ty_{*i*}
 \Rightarrow ex-table \Rightarrow (pc \times ty_{*i*}[']) list **where**
xcpt-eff *i P pc* τ *et* = (let (*ST,LT*) = τ in
 map ($\lambda(f,t,C,h,d). (h, \text{Some} (\text{Class } C \# \text{drop} (\text{size } ST - d) ST, LT)))$ (*relevant-entries* *P i pc et*))

definition *norm-eff* :: instr \Rightarrow 'm prog \Rightarrow nat \Rightarrow ty_{*i*} \Rightarrow (pc \times ty_{*i*}[']) list **where**
norm-eff *i P pc* τ = map ($\lambda pc'. (pc', \text{Some} (\text{eff}_i (i, P, \tau)))$) (*succs* *i* τ *pc*)

definition *eff* :: instr \Rightarrow 'm prog \Rightarrow pc \Rightarrow ex-table \Rightarrow ty_{*i*}['] \Rightarrow (pc \times ty_{*i*}[']) list **where**
eff *i P pc et t* = (case *t* of
 None \Rightarrow []
 | Some $\tau \Rightarrow$ (*norm-eff* *i P pc* τ) @ (*xcpt-eff* *i P pc* τ *et*))

lemma *eff-None*:
eff *i P pc xt* None = []
by (*simp* add: *eff-def*)

lemma *eff-Some*:
eff *i P pc xt* (Some τ) = *norm-eff* *i P pc* τ @ *xcpt-eff* *i P pc* τ *xt*
by (*simp* add: *eff-def*)

Conditions under which *eff* is applicable:

fun *app_{*i*}* :: instr \times 'm prog \times pc \times nat \times ty \times ty_{*i*} \Rightarrow bool **where**
*app_{*i*}-Load*:
*app_{*i*}* (*Load* *n, P, pc, mxs, T_r, (ST,LT)*) =
 (*n* < length *LT* \wedge *LT* ! *n* \neq Err \wedge length *ST* < *mxs*)

| *app_{*i*}-Store*:
*app_{*i*}* (*Store* *n, P, pc, mxs, T_r, (T#ST, LT)*) =
 (*n* < length *LT*)

| *app_{*i*}-Push*:
*app_{*i*}* (*Push* *v, P, pc, mxs, T_r, (ST,LT)*) =

- ($\text{length } ST < \text{maxs} \wedge \text{typeof } v \neq \text{None}$)
- | *app_i-Getfield*:
 $\text{app}_i (\text{Getfield } F C, P, pc, \text{maxs}, T_r, (T\#ST, LT)) =$
 $(\exists T_f. P \vdash C \text{ sees } F, \text{NonStatic}:T_f \text{ in } C \wedge P \vdash T \leq \text{Class } C)$
- | *app_i-Getstatic*:
 $\text{app}_i (\text{Getstatic } C F D, P, pc, \text{maxs}, T_r, (ST, LT)) =$
 $(\text{length } ST < \text{maxs} \wedge (\exists T_f. P \vdash C \text{ sees } F, \text{Static}:T_f \text{ in } D))$
- | *app_i-Putfield*:
 $\text{app}_i (\text{Putfield } F C, P, pc, \text{maxs}, T_r, (T_1\#T_2\#ST, LT)) =$
 $(\exists T_f. P \vdash C \text{ sees } F, \text{NonStatic}:T_f \text{ in } C \wedge P \vdash T_2 \leq (\text{Class } C) \wedge P \vdash T_1 \leq T_f)$
- | *app_i-Putstatic*:
 $\text{app}_i (\text{Putstatic } C F D, P, pc, \text{maxs}, T_r, (T\#ST, LT)) =$
 $(\exists T_f. P \vdash C \text{ sees } F, \text{Static}:T_f \text{ in } D \wedge P \vdash T \leq T_f)$
- | *app_i-New*:
 $\text{app}_i (\text{New } C, P, pc, \text{maxs}, T_r, (ST, LT)) =$
 $(\text{is-class } P C \wedge \text{length } ST < \text{maxs})$
- | *app_i-Checkcast*:
 $\text{app}_i (\text{Checkcast } C, P, pc, \text{maxs}, T_r, (T\#ST, LT)) =$
 $(\text{is-class } P C \wedge \text{is-refT } T)$
- | *app_i-Pop*:
 $\text{app}_i (\text{Pop}, P, pc, \text{maxs}, T_r, (T\#ST, LT)) =$
 True
- | *app_i-IAdd*:
 $\text{app}_i (\text{IAdd}, P, pc, \text{maxs}, T_r, (T_1\#T_2\#ST, LT)) = (T_1 = T_2 \wedge T_1 = \text{Integer})$
- | *app_i-CmpEq*:
 $\text{app}_i (\text{CmpEq}, P, pc, \text{maxs}, T_r, (T_1\#T_2\#ST, LT)) =$
 $(T_1 = T_2 \vee \text{is-refT } T_1 \wedge \text{is-refT } T_2)$
- | *app_i-IfFalse*:
 $\text{app}_i (\text{IfFalse } b, P, pc, \text{maxs}, T_r, (\text{Boolean}\#ST, LT)) =$
 $(0 \leq \text{int } pc + b)$
- | *app_i-Goto*:
 $\text{app}_i (\text{Goto } b, P, pc, \text{maxs}, T_r, s) =$
 $(0 \leq \text{int } pc + b)$
- | *app_i-Return*:
 $\text{app}_i (\text{Return}, P, pc, \text{maxs}, T_r, (T\#ST, LT)) =$
 $(P \vdash T \leq T_r)$
- | *app_i-Throw*:
 $\text{app}_i (\text{Throw}, P, pc, \text{maxs}, T_r, (T\#ST, LT)) =$
 $\text{is-refT } T$
- | *app_i-Invoke*:
 $\text{app}_i (\text{Invoke } M n, P, pc, \text{maxs}, T_r, (ST, LT)) =$
 $(n < \text{length } ST \wedge$
 $(ST!n \neq NT \rightarrow$
 $(\exists C D Ts T m. ST!n = \text{Class } C \wedge P \vdash C \text{ sees } M, \text{NonStatic}:Ts \rightarrow T = m \text{ in } D \wedge$
 $P \vdash \text{rev } (\text{take } n ST) [\leq] Ts)))$
- | *app_i-Invokestatic*:
 $\text{app}_i (\text{Invokestatic } C M n, P, pc, \text{maxs}, T_r, (ST, LT)) =$
 $(\text{length } ST - n < \text{maxs} \wedge n \leq \text{length } ST \wedge M \neq \text{clinit} \wedge$
 $(\exists D Ts T m. P \vdash C \text{ sees } M, \text{Static}:Ts \rightarrow T = m \text{ in } D \wedge$
 $P \vdash \text{rev } (\text{take } n ST) [\leq] Ts))$
- | *app_i-default*:
 $\text{app}_i (i, P, pc, \text{maxs}, T_r, s) = \text{False}$

definition $xcpt\text{-}app :: instr \Rightarrow 'm\ prog \Rightarrow pc \Rightarrow nat \Rightarrow ex\text{-}table \Rightarrow ty_i \Rightarrow bool$ **where**

$xcpt\text{-}app\ i\ P\ pc\ mxs\ xt\ \tau \longleftrightarrow (\forall (f,t,C,h,d) \in set\ (relevant\text{-}entries\ P\ i\ pc\ xt). is\text{-}class\ P\ C \wedge d \leq size\ (fst\ \tau) \wedge d < mxs)$

definition $app :: instr \Rightarrow 'm\ prog \Rightarrow nat \Rightarrow ty \Rightarrow nat \Rightarrow nat \Rightarrow ex\text{-}table \Rightarrow ty_i' \Rightarrow bool$ **where**

$app\ i\ P\ mxs\ T_r\ pc\ mpc\ xt\ t = (case\ t\ of\ None \Rightarrow True\ |\ Some\ \tau \Rightarrow app_i\ (i,P,pc,mxs,T_r,\tau) \wedge xcpt\text{-}app\ i\ P\ pc\ mxs\ xt\ \tau \wedge (\forall (pc',\tau') \in set\ (eff\ i\ P\ pc\ xt\ t). pc' < mpc))$

lemma $app\text{-}Some$:

$app\ i\ P\ mxs\ T_r\ pc\ mpc\ xt\ (Some\ \tau) = (app_i\ (i,P,pc,mxs,T_r,\tau) \wedge xcpt\text{-}app\ i\ P\ pc\ mxs\ xt\ \tau \wedge (\forall (pc',s') \in set\ (eff\ i\ P\ pc\ xt\ (Some\ \tau)). pc' < mpc))$

by ($simp\ add$: $app\text{-}def$)

locale $eff = jvm\text{-}method +$

fixes eff_i **and** app_i **and** eff **and** app

fixes $norm\text{-}eff$ **and** $xcpt\text{-}app$ **and** $xcpt\text{-}eff$

fixes mpc

defines $mpc \equiv size\ is$

defines $eff_i\ i\ \tau \equiv Effect.eff_i\ (i,P,\tau)$

notes $eff_i\text{-}simps\ [simp] = Effect.eff_i.simps$ [**where** $P = P$, $folded\ eff_i\text{-}def$]

defines $app_i\ i\ pc\ \tau \equiv Effect.app_i\ (i, P, pc, mxs, T_r, \tau)$

notes $app_i\text{-}simps\ [simp] = Effect.app_i.simps$ [**where** $P=P$ **and** $mxs=mxs$ **and** $T_r=T_r$, $folded\ app_i\text{-}def$]

defines $xcpt\text{-}eff\ i\ pc\ \tau \equiv Effect.xcpt\text{-}eff\ i\ P\ pc\ \tau\ xt$

notes $xcpt\text{-}eff = Effect.xcpt\text{-}eff\text{-}def$ [$of - P - -\ xt$, $folded\ xcpt\text{-}eff\text{-}def$]

defines $norm\text{-}eff\ i\ pc\ \tau \equiv Effect.norm\text{-}eff\ i\ P\ pc\ \tau$

notes $norm\text{-}eff = Effect.norm\text{-}eff\text{-}def$ [$of - P$, $folded\ norm\text{-}eff\text{-}def\ eff_i\text{-}def$]

defines $eff\ i\ pc \equiv Effect.eff\ i\ P\ pc\ xt$

notes $eff = Effect.eff\text{-}def$ [$of - P - -\ xt$, $folded\ eff\text{-}def\ norm\text{-}eff\text{-}def\ xcpt\text{-}eff\text{-}def$]

defines $xcpt\text{-}app\ i\ pc\ \tau \equiv Effect.xcpt\text{-}app\ i\ P\ pc\ mxs\ xt\ \tau$

notes $xcpt\text{-}app = Effect.xcpt\text{-}app\text{-}def$ [$of - P - mxs\ xt$, $folded\ xcpt\text{-}app\text{-}def$]

defines $app\ i\ pc \equiv Effect.app\ i\ P\ mxs\ T_r\ pc\ mpc\ xt$

notes $app = Effect.app\text{-}def$ [$of - P\ mxs\ T_r - mpc\ xt$, $folded\ app\text{-}def\ xcpt\text{-}app\text{-}def\ app_i\text{-}def\ eff\text{-}def$]

lemma $length\text{-}cases2$:

assumes $\bigwedge LT. P\ (\[],LT)$

assumes $\bigwedge l\ ST\ LT. P\ (l\#\!ST,LT)$

shows $P\ s$

by ($cases\ s$, $cases\ fst\ s$) ($auto\ intro!$: $assms$)

lemma *length-cases3*:

assumes $\bigwedge LT. P ([], LT)$
 assumes $\bigwedge l LT. P ([l], LT)$
 assumes $\bigwedge l ST LT. P (l\#ST, LT)$
 shows $P s$

lemma *length-cases4*:

assumes $\bigwedge LT. P ([], LT)$
 assumes $\bigwedge l LT. P ([l], LT)$
 assumes $\bigwedge l l' LT. P ([l, l'], LT)$
 assumes $\bigwedge l l' ST LT. P (l\#l'\#ST, LT)$
 shows $P s$

simp rules for *app*

lemma *appNone[simp]*: $app\ i\ P\ mxs\ T_r\ pc\ mpc\ et\ None = True$
 by (*simp add: app-def*)

lemma *appLoad[simp]*:

$app_i\ (Load\ idx, P, T_r, mxs, pc, s) = (\exists ST\ LT. s = (ST, LT) \wedge idx < length\ LT \wedge LT!idx \neq Err \wedge length\ ST < mxs)$
 by (*cases s, simp*)

lemma *appStore[simp]*:

$app_i\ (Store\ idx, P, pc, mxs, T_r, s) = (\exists ts\ ST\ LT. s = (ts\#ST, LT) \wedge idx < length\ LT)$
 by (*rule length-cases2, auto*)

lemma *appPush[simp]*:

$app_i\ (Push\ v, P, pc, mxs, T_r, s) =$
 $(\exists ST\ LT. s = (ST, LT) \wedge length\ ST < mxs \wedge typeof\ v \neq None)$
 by (*cases s, simp*)

lemma *appGetField[simp]*:

$app_i\ (Getfield\ F\ C, P, pc, mxs, T_r, s) =$
 $(\exists oT\ vT\ ST\ LT. s = (oT\#ST, LT) \wedge$
 $P \vdash C\ sees\ F, NonStatic:vT\ in\ C \wedge P \vdash oT \leq (Class\ C))$
 by (*rule length-cases2 [of - s] auto*)

lemma *appGetStatic[simp]*:

$app_i\ (Getstatic\ C\ F\ D, P, pc, mxs, T_r, s) =$
 $(\exists vT\ ST\ LT. s = (ST, LT) \wedge length\ ST < mxs \wedge P \vdash C\ sees\ F, Static:vT\ in\ D)$
 by (*rule length-cases2 [of - s] auto*)

lemma *appPutField[simp]*:

$app_i\ (Putfield\ F\ C, P, pc, mxs, T_r, s) =$
 $(\exists vT\ vT'\ oT\ ST\ LT. s = (vT\#oT\#ST, LT) \wedge$
 $P \vdash C\ sees\ F, NonStatic:vT'\ in\ C \wedge P \vdash oT \leq (Class\ C) \wedge P \vdash vT \leq vT')$
 by (*rule length-cases4 [of - s], auto*)

lemma *appPutstatic[simp]*:

$app_i\ (Putstatic\ C\ F\ D, P, pc, mxs, T_r, s) =$
 $(\exists vT\ vT'\ ST\ LT. s = (vT\#ST, LT) \wedge$
 $P \vdash C\ sees\ F, Static:vT'\ in\ D \wedge P \vdash vT \leq vT')$
 by (*rule length-cases4 [of - s], auto*)

lemma *appNew*[simp]:

$app_i (New\ C,P,pc,mxs,T_r,s) =$
 $(\exists ST\ LT. s=(ST,LT) \wedge is-class\ P\ C \wedge length\ ST < mxs)$
by (*cases* *s*, *simp*)

lemma *appCheckcast*[simp]:

$app_i (Checkcast\ C,P,pc,mxs,T_r,s) =$
 $(\exists T\ ST\ LT. s = (T\#\!ST,LT) \wedge is-class\ P\ C \wedge is-refT\ T)$
by (*cases* *s*, *cases* *fst* *s*, *simp* *add*: *app-def*) (*cases* *hd* (*fst* *s*), *auto*)

lemma *app_iPop*[simp]:

$app_i (Pop,P,pc,mxs,T_r,s) = (\exists ts\ ST\ LT. s = (ts\#\!ST,LT))$
by (*rule* *length-cases2*, *auto*)

lemma *appIAdd*[simp]:

$app_i (IAdd,P,pc,mxs,T_r,s) = (\exists ST\ LT. s = (Integer\#\!Integer\#\!ST,LT))$

lemma *appIfFalse* [simp]:

$app_i (IfFalse\ b,P,pc,mxs,T_r,s) =$
 $(\exists ST\ LT. s = (Boolean\#\!ST,LT) \wedge 0 \leq int\ pc + b)$

lemma *appCmpEq*[simp]:

$app_i (CmpEq,P,pc,mxs,T_r,s) =$
 $(\exists T_1\ T_2\ ST\ LT. s = (T_1\#\!T_2\#\!ST,LT) \wedge (\neg is-refT\ T_1 \wedge T_2 = T_1 \vee is-refT\ T_1 \wedge is-refT\ T_2))$
by (*rule* *length-cases4*, *auto*)

lemma *appReturn*[simp]:

$app_i (Return,P,pc,mxs,T_r,s) = (\exists T\ ST\ LT. s = (T\#\!ST,LT) \wedge P \vdash T \leq T_r)$
by (*rule* *length-cases2*, *auto*)

lemma *appThrow*[simp]:

$app_i (Throw,P,pc,mxs,T_r,s) = (\exists T\ ST\ LT. s=(T\#\!ST,LT) \wedge is-refT\ T)$
by (*rule* *length-cases2*, *auto*)

lemma *effNone*:

$(pc', s') \in set\ (eff\ i\ P\ pc\ et\ None) \implies s' = None$
by (*auto* *simp* *add*: *eff-def* *xcpt-eff-def* *norm-eff-def*)

some helpers to make the specification directly executable:

lemma *relevant-entries-append* [simp]:

$relevant-entries\ P\ i\ pc\ (xt\ @\ xt') = relevant-entries\ P\ i\ pc\ xt\ @\ relevant-entries\ P\ i\ pc\ xt'$
by (*unfold* *relevant-entries-def*) *simp*

lemma *xcpt-app-append* [iff]:

$xcpt-app\ i\ P\ pc\ mxs\ (xt\ @\ xt')\ \tau = (xcpt-app\ i\ P\ pc\ mxs\ xt\ \tau \wedge xcpt-app\ i\ P\ pc\ mxs\ xt'\ \tau)$
by (*unfold* *xcpt-app-def*) *fastforce*

lemma *xcpt-eff-append* [simp]:

$xcpt-eff\ i\ P\ pc\ \tau\ (xt\ @\ xt') = xcpt-eff\ i\ P\ pc\ \tau\ xt\ @\ xcpt-eff\ i\ P\ pc\ \tau\ xt'$
by (*unfold* *xcpt-eff-def*, *cases* τ) *simp*

lemma *app-append* [simp]:

$app\ i\ P\ pc\ T\ mxs\ mpc\ (xt\ @\ xt')\ \tau = (app\ i\ P\ pc\ T\ mxs\ mpc\ xt\ \tau \wedge app\ i\ P\ pc\ T\ mxs\ mpc\ xt'\ \tau)$
by (*unfold* *app-def* *eff-def*) *auto*

end

2.10 Monotonicity of eff and app

theory *EffectMono* imports *Effect* begin

declare *not-Err-eq* [*iff*]

lemma *app_i-mono*:

assumes *wf*: *wf-prog* *p P*

assumes *less*: $P \vdash \tau \leq_i \tau'$

shows $app_i (i, P, mxs, mpc, rT, \tau') \implies app_i (i, P, mxs, mpc, rT, \tau)$

lemma *succs-mono*:

assumes *wf*: *wf-prog* *p P* and *app_i*: $app_i (i, P, mxs, mpc, rT, \tau')$

shows $P \vdash \tau \leq_i \tau' \implies set (succs\ i\ \tau\ pc) \subseteq set (succs\ i\ \tau'\ pc)$

lemma *app-mono*:

assumes *wf*: *wf-prog* *p P*

assumes *less'*: $P \vdash \tau \leq' \tau'$

shows $app\ i\ P\ m\ rT\ pc\ mpc\ xt\ \tau' \implies app\ i\ P\ m\ rT\ pc\ mpc\ xt\ \tau$

lemma *eff_i-mono*:

assumes *wf*: *wf-prog* *p P*

assumes *less*: $P \vdash \tau \leq_i \tau'$

assumes *app_i*: $app\ i\ P\ m\ rT\ pc\ mpc\ xt\ (Some\ \tau')$

assumes *succs*: $succs\ i\ \tau\ pc \neq [] \quad succs\ i\ \tau'\ pc \neq []$

shows $P \vdash eff_i (i, P, \tau) \leq_i eff_i (i, P, \tau')$

end

2.11 The Bytecode Verifier

theory *BVSpec*

imports *Effect*

begin

This theory contains a specification of the BV. The specification describes correct typings of method bodies; it corresponds to type *checking*.

definition

— The method type only contains declared classes:

$check_types :: 'm\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i' \ err\ list \Rightarrow bool$

where

$check_types\ P\ mxs\ mxl\ \tau s \equiv set\ \tau s \subseteq states\ P\ mxs\ mxl$

— An instruction is welltyped if it is applicable and its effect

— is compatible with the type at all successor instructions:

definition

$wt_instr :: ['m\ prog, ty, nat, pc, ex_table, instr, pc, ty_m] \Rightarrow bool$

$(\langle -, -, -, - \vdash -, - \rangle \rightarrow [60, 0, 0, 0, 0, 0, 61]\ 60)$

where

$P, T, mxs, mpc, xt \vdash i, pc :: \tau s \equiv$

$app\ i\ P\ mxs\ T\ pc\ mpc\ xt\ (\tau s!pc) \wedge$

$(\forall (pc', \tau') \in set (eff\ i\ P\ pc\ xt\ (\tau s!pc)). P \vdash \tau' \leq' \tau s!pc')$

— The type at $pc=0$ conforms to the method calling convention:

definition $wt\text{-}start :: [m\ prog, cname, staticb, ty\ list, nat, ty_m] \Rightarrow bool$

where

$wt\text{-}start\ P\ C\ b\ Ts\ mxl_0\ \tau s \equiv$
 $case\ b\ of\ NonStatic \Rightarrow P \vdash Some\ (\ [], OK\ (Class\ C)\#map\ OK\ Ts@replicate\ mxl_0\ Err) \leq' \tau s!0$
 $\quad | Static \Rightarrow P \vdash Some\ (\ [], map\ OK\ Ts@replicate\ mxl_0\ Err) \leq' \tau s!0$

— A method is welltyped if the body is not empty,
 — if the method type covers all instructions and mentions
 — declared classes only, if the method calling convention is respected, and
 — if all instructions are welltyped.

definition $wt\text{-}method :: [m\ prog, cname, staticb, ty\ list, ty, nat, nat, instr\ list,$
 $ex\ table, ty_m] \Rightarrow bool$

where

$wt\text{-}method\ P\ C\ b\ Ts\ T_r\ mxs\ mxl_0\ is\ xt\ \tau s \equiv (b = Static \vee b = NonStatic) \wedge$
 $0 < size\ is \wedge size\ \tau s = size\ is \wedge$
 $check\ types\ P\ mxs\ ((case\ b\ of\ Static \Rightarrow 0 \mid NonStatic \Rightarrow 1) + size\ Ts + mxl_0)\ (map\ OK\ \tau s) \wedge$
 $wt\text{-}start\ P\ C\ b\ Ts\ mxl_0\ \tau s \wedge$
 $(\forall pc < size\ is.\ P, T_r, mxs, size\ is, xt \vdash is!pc, pc :: \tau s)$

— A program is welltyped if it is wellformed and all methods are welltyped

definition $wf\text{-}jvm\text{-}prog\text{-}phi :: ty_P \Rightarrow jvm\text{-}prog \Rightarrow bool\ (\langle wf'\text{-}jvm'\text{-}prog.\rangle)$

where

$wf\text{-}jvm\text{-}prog_{\Phi} \equiv$
 $wf\text{-}prog\ (\lambda P\ C\ (M, b, Ts, T_r, (mxs, mxl_0, is, xt)).$
 $wt\text{-}method\ P\ C\ b\ Ts\ T_r\ mxs\ mxl_0\ is\ xt\ (\Phi\ C\ M))$

definition $wf\text{-}jvm\text{-}prog :: jvm\text{-}prog \Rightarrow bool$

where

$wf\text{-}jvm\text{-}prog\ P \equiv \exists \Phi.\ wf\text{-}jvm\text{-}prog_{\Phi}\ P$

lemma $wt\text{-}jvm\text{-}progD$:

$wf\text{-}jvm\text{-}prog_{\Phi}\ P \Longrightarrow \exists wt.\ wf\text{-}prog\ wt\ P$

lemma $wt\text{-}jvm\text{-}prog\text{-}impl\text{-}wt\text{-}instr$:

assumes wf : $wf\text{-}jvm\text{-}prog_{\Phi}\ P$ **and**

$sees$: $P \vdash C\ sees\ M, b: Ts \rightarrow T = (mxs, mxl_0, ins, xt)$ in C **and**
 pc : $pc < size\ ins$

shows $P, T, mxs, size\ ins, xt \vdash ins!pc, pc :: \Phi\ C\ M$

lemma $wt\text{-}jvm\text{-}prog\text{-}impl\text{-}wt\text{-}start$:

assumes wf : $wf\text{-}jvm\text{-}prog_{\Phi}\ P$ **and**

$sees$: $P \vdash C\ sees\ M, b: Ts \rightarrow T = (mxs, mxl_0, ins, xt)$ in C

shows $0 < size\ ins \wedge wt\text{-}start\ P\ C\ b\ Ts\ mxl_0\ (\Phi\ C\ M)$

lemma $wf\text{-}jvm\text{-}prog\text{-}nclinit$:

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

and $meth$: $P \vdash C\ sees\ M, b : Ts \rightarrow T = (mxs, mxl_0, ins, xt)$ in D

and wt : $P, T, mxs, size\ ins, xt \vdash ins!pc, pc :: \Phi\ C\ M$

and pc : $pc < length\ ins$ **and** Φ : $\Phi\ C\ M ! pc = Some(ST, LT)$

and ins : $ins ! pc = Invokestatic\ C_0\ M_0\ n$

shows $M_0 \neq clinit$

using $assms$ **by** ($simp\ add$: $wf\text{-}jvm\text{-}prog\text{-}phi\text{-}def\ wt\text{-}instr\text{-}def\ app\text{-}def$)

end

2.12 The Typing Framework for the JVM

```

theory TF-JVM
imports Jinja.Typing-Framework-err EffectMono BVSpec
begin

definition exec :: jvm-prog  $\Rightarrow$  nat  $\Rightarrow$  ty  $\Rightarrow$  ex-table  $\Rightarrow$  instr list  $\Rightarrow$  tyi' err step-type
where
  exec G mxs rT et bs  $\equiv$ 
  err-step (size bs) ( $\lambda$ pc. app (bs!pc) G mxs rT pc (size bs) et)
    ( $\lambda$ pc. eff (bs!pc) G pc et)

locale JVM-sl =
  fixes P :: jvm-prog and mxs and mxl0 and n
  fixes b and Ts :: ty list and is and xt and Tr

  fixes mxl and A and r and f and app and eff and step
  defines [simp]: mxl  $\equiv$  (case b of Static  $\Rightarrow$  0 | NonStatic  $\Rightarrow$  1)+size Ts+mxl0
  defines [simp]: A  $\equiv$  states P mxs mxl
  defines [simp]: r  $\equiv$  JVM-SemiType.le P mxs mxl
  defines [simp]: f  $\equiv$  JVM-SemiType.sup P mxs mxl

  defines [simp]: app  $\equiv$   $\lambda$ pc. Effect.app (is!pc) P mxs Tr pc (size is) xt
  defines [simp]: eff  $\equiv$   $\lambda$ pc. Effect.eff (is!pc) P pc xt
  defines [simp]: step  $\equiv$  err-step (size is) app eff

  defines [simp]: n  $\equiv$  size is
  assumes staticb: b = Static  $\vee$  b = NonStatic

locale start-context = JVM-sl +
  fixes p and C

  assumes wf: wf-prog p P
  assumes C: is-class P C
  assumes Ts: set Ts  $\subseteq$  types P

  fixes first :: tyi' and start
  defines [simp]:
    first  $\equiv$  Some ([],(case b of Static  $\Rightarrow$  [] | NonStatic  $\Rightarrow$  [OK (Class C)])) @ map OK Ts @ replicate
    mxl0 Err)
  defines [simp]:
    start  $\equiv$  (OK first) # replicate (size is - 1) (OK None)
thm start-context.intro

lemma start-context-intro-auxi:
  fixes P b Ts p C
  assumes b = Static  $\vee$  b = NonStatic
    and wf-prog p P
    and is-class P C
    and set Ts  $\subseteq$  types P
  shows start-context P b Ts p C
using start-context.intro[OF JVM-sl.intro] start-context-axioms-def assms by auto

```

2.12.1 Connecting JVM and Framework

lemma (in *start-context*) *semi: semilat* (A, r, f)
apply (*insert semilat-JVM*[*OF wf*])
apply (*unfold A-def r-def f-def JVM-SemiType.le-def JVM-SemiType.sup-def states-def*)
apply *auto*
done

lemma (in *JVM-sl*) *step-def-exec: step* \equiv *exec* P *mxs* T_r *xt is*
by (*simp add: exec-def*)

lemma *special-ex-swap-lemma* [*iff*]:
 $(? X. (? n. X = A\ n \ \& \ P\ n) \ \& \ Q\ X) = (? n. Q(A\ n) \ \& \ P\ n)$
by *blast*

lemma *ex-in-list* [*iff*]:
 $(\exists n. ST \in nlists\ n\ A \wedge n \leq mxs) = (set\ ST \subseteq A \wedge size\ ST \leq mxs)$
by (*unfold nlists-def*) *auto*

lemma *singleton-nlists*:
 $(\exists n. [Class\ C] \in nlists\ n\ (types\ P) \wedge n \leq mxs) = (is-class\ P\ C \wedge 0 < mxs)$
by *auto*

lemma *set-drop-subset*:
 $set\ xs \subseteq A \implies set\ (drop\ n\ xs) \subseteq A$
by (*auto dest: in-set-dropD*)

lemma *Suc-minus-minus-le*:
 $n < mxs \implies Suc\ (n - (n - b)) \leq mxs$
by *arith*

lemma *in-nlistsE*:
 $\llbracket xs \in nlists\ n\ A; \llbracket size\ xs = n; set\ xs \subseteq A \rrbracket \implies P \rrbracket \implies P$
by (*unfold nlists-def*) *blast*

declare *is-relevant-entry-def* [*simp*]
declare *set-drop-subset* [*simp*]

theorem (in *start-context*) *exec-pres-type*:
pres-type *step* (*size is*) A
declare *is-relevant-entry-def* [*simp del*]
declare *set-drop-subset* [*simp del*]

lemma *lesubstep-type-simple*:
 $xs \llbracket \sqsubseteq_{Product.le} (=)\ r \rrbracket ys \implies set\ xs \ \{\llbracket \sqsubseteq_r \rrbracket\} \ set\ ys$
declare *is-relevant-entry-def* [*simp del*]

lemma *conjI2*: $\llbracket A; A \implies B \rrbracket \implies A \wedge B$ **by** *blast*

lemma (in *JVM-sl*) *eff-mono*:
assumes *wf: wf-prog* $p\ P$ **and** $pc < length\ is$ **and**
lesub: s $\llbracket \sqsubseteq_{sup-state-opt}\ P\ t \rrbracket$ **and** *app: app* $pc\ t$

shows $set (eff\ pc\ s) \{\sqsubseteq_{sup\ state\ opt}\ P\} set (eff\ pc\ t)$

lemma (in *JVM-sl*) *bounded-step*: *bounded step (size is)*

theorem (in *JVM-sl*) *step-mono*:

$wf\ prog\ wf\ mb\ P \implies mono\ r\ step\ (size\ is)\ A$

lemma (in *start-context*) *first-in-A* [*iff*]: *OK first* $\in A$

using *Ts C* **by** (*cases b*; *force intro!*; *nlists-appendI simp add: JVM-states-unfold*)

lemma (in *JVM-sl*) *wt-method-def2*:

$wt\ method\ P\ C'\ b\ Ts\ T_r\ mxs\ mxl_0\ is\ xt\ \tau s =$

$(is \neq [] \wedge$

$(b = Static \vee b = NonStatic) \wedge$

$size\ \tau s = size\ is \wedge$

$OK\ 'set\ \tau s \subseteq states\ P\ mxs\ mxl \wedge$

$wt\ start\ P\ C'\ b\ Ts\ mxl_0\ \tau s \wedge$

$wt\ app\ eff\ (sup\ state\ opt\ P)\ app\ eff\ \tau s)$

end

2.13 Kildall for the JVM

theory *BVExec*

imports *Jinja.Abstract-BV TF-JVM Jinja.Typing-Framework-2*

begin

definition *kiljvm* :: $jvm\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty \Rightarrow$

$instr\ list \Rightarrow ex\ table \Rightarrow ty_i' err\ list \Rightarrow ty_i' err\ list$

where

$kiljvm\ P\ mxs\ mxl\ T_r\ is\ xt \equiv$

$kildall\ (JVM\ SemiType.le\ P\ mxs\ mxl)\ (JVM\ SemiType.sup\ P\ mxs\ mxl)$

$(exec\ P\ mxs\ T_r\ xt\ is)$

definition *wt-kildall* :: $jvm\ prog \Rightarrow cname \Rightarrow staticb \Rightarrow ty\ list \Rightarrow ty \Rightarrow nat \Rightarrow nat \Rightarrow$

$instr\ list \Rightarrow ex\ table \Rightarrow bool$

where

$wt\ kildall\ P\ C'\ b\ Ts\ T_r\ mxs\ mxl_0\ is\ xt \equiv (b = Static \vee b = NonStatic) \wedge$

$0 < size\ is \wedge$

$(let\ first = Some\ ([], (case\ b\ of\ Static \Rightarrow [] \mid NonStatic \Rightarrow [OK\ (Class\ C')]))$

$\quad @(\map\ OK\ Ts) @(\replicate\ mxl_0\ Err));$

$start = (OK\ first) \# (\replicate\ (size\ is - 1)\ (OK\ None));$

$result = kiljvm\ P\ mxs$

$\quad ((case\ b\ of\ Static \Rightarrow 0 \mid NonStatic \Rightarrow 1) + size\ Ts + mxl_0)$

$\quad T_r\ is\ xt\ start$

$in\ \forall n < size\ is. result!n \neq Err)$

definition *wf-jvm-prog_k* :: $jvm\ prog \Rightarrow bool$

where

$wf\ jvm\ prog_k\ P \equiv$

$wf\ prog\ (\lambda P\ C'\ (M, b, Ts, T_r, (mxs, mxl_0, is, xt)). wt\ kildall\ P\ C'\ b\ Ts\ T_r\ mxs\ mxl_0\ is\ xt)\ P$

context *start-context*

begin

lemma *Cons-less-Conss3* [simp]:

$x\#xs \sqsubset_r y\#ys = (x \sqsubset_r y \wedge xs \sqsubset_r ys \vee x = y \wedge xs \sqsubset_r ys)$

unfolding *lesssub-def*

by (*metis Cons-le-Cons JVM-le-Err-conv lesssub-def list.inject r-def sup-state-opt-err*)

lemma *acc-le-listI3* [intro!]:

$acc\ r \implies acc\ (Listn.le\ r)$

unfolding *acc-def*

apply (*subgoal-tac*

$wf(UN\ n.\ \{(ys, xs).\ size\ xs = n \wedge size\ ys = n \wedge xs <-(Listn.le\ r)\ ys\})$)

apply (*erule wf-subset*)

apply (*blast intro: lesssub-lengthD*)

apply (*rule wf-UN*)

prefer 2

apply *force*

apply (*rename-tac n*)

apply (*induct-tac n*)

apply (*simp add: lesssub-def cong: conj-cong*)

apply (*rename-tac k*)

apply (*simp add: wf-eq-minimal del: r-def f-def step-def A-def*)

apply (*simp (no-asm) add: length-Suc-conv cong: conj-cong del: r-def f-def step-def A-def*)

apply *clarify*

apply (*rename-tac M m*)

apply (*case-tac* $\exists x\ xs.\ size\ xs = k \wedge x\#xs \in M$)

prefer 2

apply *blast*

apply (*erule-tac* $x = \{a.\ \exists xs.\ size\ xs = k \wedge a\#xs:M\}$ **in** *allE*)

apply (*erule impE*)

apply *blast*

apply (*thin-tac* $\exists x\ xs.\ P\ x\ xs$ **for** *P*)

apply *clarify*

apply (*rename-tac maxA xs*)

apply (*erule-tac* $x = \{ys.\ size\ ys = size\ xs \wedge maxA\#ys \in M\}$ **in** *allE*)

apply (*erule impE*)

apply *blast*

apply *clarify*

using *Cons-less-Conss3* **by** *blast*

lemma *wf-jvm*: $wf\ \{(ss', ss).\ ss \sqsubset_r ss'\}$

using *acc-le-listI3 acc-JVM [OF wf]* **by** (*simp add: acc-def r-def*)

lemma *iter-properties-bv*[*rule-format*]:

shows $\llbracket \forall p \in w0.\ p < n;\ ss0 \in nlists\ n\ A;\ \forall p < n.\ p \notin w0 \implies stable\ r\ step\ ss0\ p \rrbracket \implies$

$iter\ f\ step\ ss0\ w0 = (ss', w') \longrightarrow$

$ss' \in nlists\ n\ A \wedge stables\ r\ step\ ss' \wedge ss0 \sqsubset_r ss' \wedge$

$(\forall ts \in nlists\ n\ A.\ ss0 \sqsubset_r ts \wedge stables\ r\ step\ ts \longrightarrow ss' \sqsubset_r ts)$

lemma *kildall-properties-bv*:

shows $\llbracket ss0 \in nlists\ n\ A \rrbracket \implies$

$kildall\ r\ f\ step\ ss0 \in nlists\ n\ A \wedge$

$stables\ r\ step\ (kildall\ r\ f\ step\ ss0) \wedge$

$$ss0 \llbracket _ \rrbracket_r \text{ kildall } r \text{ f step } ss0 \wedge$$

$$(\forall ts \in nlists \ n \ A. \ ss0 \llbracket _ \rrbracket_r \ ts \wedge \text{ stables } r \text{ step } ts \longrightarrow$$

$$\text{ kildall } r \text{ f step } ss0 \llbracket _ \rrbracket_r \ ts)$$

2.14 LBV for the JVM

theory *LBVJVM*

imports *Jinja.Abstract-BV TF-JVM*

begin

type-synonym *prog-cert* = *cname* \Rightarrow *mname* \Rightarrow *ty_i' err list*

definition *check-cert* :: *jvm-prog* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *ty_i' err list* \Rightarrow *bool*

where

$$\text{check-cert } P \text{ mxs mxl } n \text{ cert} \equiv \text{check-types } P \text{ mxs mxl } \text{cert} \wedge \text{size cert} = n+1 \wedge$$

$$(\forall i < n. \text{cert}!i \neq \text{Err}) \wedge \text{cert}!n = \text{OK None}$$

definition *lbvjvm* :: *jvm-prog* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *ty* \Rightarrow *ex-table* \Rightarrow

$$\text{ty}_i' \text{ err list} \Rightarrow \text{instr list} \Rightarrow \text{ty}_i' \text{ err} \Rightarrow \text{ty}_i' \text{ err}$$

where

$$\text{lbvjvm } P \text{ mxs mxr } T_r \text{ et } \text{cert } \text{bs} \equiv$$

$$\text{wtl-inst-list } \text{bs } \text{cert} \ (\text{JVM-SemiType.sup } P \text{ mxs mxr}) \ (\text{JVM-SemiType.le } P \text{ mxs mxr}) \ \text{Err} \ (\text{OK None}) \ (\text{exec } P \text{ mxs } T_r \ \text{et } \text{bs}) \ 0$$

definition *wt-lbv* :: *jvm-prog* \Rightarrow *cname* \Rightarrow *staticb* \Rightarrow *ty list* \Rightarrow *ty* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow

$$\text{ex-table} \Rightarrow \text{ty}_i' \text{ err list} \Rightarrow \text{instr list} \Rightarrow \text{bool}$$

where

$$\text{wt-lbv } P \ C \ b \ T_s \ T_r \ \text{mxs mxl}_0 \ \text{et } \text{cert } \text{ins} \equiv (b = \text{Static} \vee b = \text{NonStatic}) \wedge$$

$$\text{check-cert } P \ \text{mxs} \ ((\text{case } b \ \text{of } \text{Static} \Rightarrow 0 \mid \text{NonStatic} \Rightarrow 1) + \text{size } T_s + \text{mxl}_0) \ (\text{size } \text{ins}) \ \text{cert} \wedge$$

$$0 < \text{size } \text{ins} \wedge$$

$$(\text{let } \text{start} = \text{Some} \ (\[], (\text{case } b \ \text{of } \text{Static} \Rightarrow [] \mid \text{NonStatic} \Rightarrow [\text{OK} \ (\text{Class } C)]))$$

$$\ @((\text{map } \text{OK } T_s)) @(\text{replicate } \text{mxl}_0 \ \text{Err});$$

$$\text{result} = \text{lbvjvm } P \ \text{mxs} \ ((\text{case } b \ \text{of } \text{Static} \Rightarrow 0 \mid \text{NonStatic} \Rightarrow 1) + \text{size } T_s + \text{mxl}_0) \ T_r \ \text{et } \text{cert } \text{ins}$$

$$(\text{OK } \text{start})$$

$$\text{in } \text{result} \neq \text{Err})$$

definition *wt-jvm-prog-lbv* :: *jvm-prog* \Rightarrow *prog-cert* \Rightarrow *bool*

where

$$\text{wt-jvm-prog-lbv } P \ \text{cert} \equiv$$

$$\text{wf-prog} \ (\lambda P \ C \ (\text{mn}, b, T_s, T_r, (\text{mxs}, \text{mxl}_0, \text{ins}, \text{et})) . \ \text{wt-lbv } P \ C \ b \ T_s \ T_r \ \text{mxs} \ \text{mxl}_0 \ \text{et} \ (\text{cert } C \ \text{mn}) \ \text{ins}) \ P$$

definition *mk-cert* :: *jvm-prog* \Rightarrow *nat* \Rightarrow *ty* \Rightarrow *ex-table* \Rightarrow *instr list*

$$\Rightarrow \text{ty}_m \Rightarrow \text{ty}_i' \text{ err list}$$

where

$$\text{mk-cert } P \ \text{mxs} \ T_r \ \text{et } \text{bs } \text{phi} \equiv \text{make-cert} \ (\text{exec } P \ \text{mxs} \ T_r \ \text{et } \text{bs}) \ (\text{map } \text{OK } \text{phi}) \ (\text{OK None})$$

definition *prg-cert* :: *jvm-prog* \Rightarrow *ty_P* \Rightarrow *prog-cert*

where

$$\text{prg-cert } P \ \text{phi} \ C \ \text{mn} \equiv \text{let } (C, b, T_s, T_r, (\text{mxs}, \text{mxl}_0, \text{ins}, \text{et})) = \text{method } P \ C \ \text{mn}$$

$$\text{in } \text{mk-cert } P \ \text{mxs} \ T_r \ \text{et } \text{ins} \ (\text{phi } C \ \text{mn})$$

lemma *check-certD* [*intro?*]:

$$\text{check-cert } P \ \text{mxs} \ \text{mxl} \ n \ \text{cert} \Longrightarrow \text{cert-ok } \text{cert } n \ \text{Err} \ (\text{OK None}) \ (\text{states } P \ \text{mxs} \ \text{mxl})$$

by (unfold cert-ok-def check-cert-def check-types-def) auto

lemma (in start-context) wt-lbv-wt-step:

assumes lbv: wt-lbv $P C b Ts T_r mxs mxl_0$ *at cert is*

shows $\exists \tau s \in nlists$ (size is) A . wt-step r Err step $\tau s \wedge OK$ first $\sqsubseteq_r \tau s!0$

lemma (in start-context) wt-lbv-wt-method:

assumes lbv: wt-lbv $P C b Ts T_r mxs mxl_0$ *at cert is*

shows $\exists \tau s$. wt-method $P C b Ts T_r mxs mxl_0$ *is at* τs

lemma (in start-context) wt-method-wt-lbv:

assumes wt: wt-method $P C b Ts T_r mxs mxl_0$ *is at* τs

defines [simp]: cert \equiv mk-cert $P mxs T_r$ *at is* τs

shows wt-lbv $P C b Ts T_r mxs mxl_0$ *at cert is*

theorem jvm-lbv-correct:

wt-jvm-prog-lbv $P Cert \implies$ wf-jvm-prog P

theorem jvm-lbv-complete:

assumes wt: wf-jvm-prog $_{\Phi}$ P

shows wt-jvm-prog-lbv P (prg-cert $P \Phi$)

end

2.15 BV Type Safety Invariant

theory BVConform

imports BVSpec ../JVM/JVMExec ../Common/Conform

begin

2.15.1 correct-state definitions

definition confT :: 'c prog \Rightarrow heap \Rightarrow val \Rightarrow ty err \Rightarrow bool

($\langle -, \vdash - : \leq_{\top} \rightarrow [51, 51, 51, 51] 50$)

where

$P, h \vdash v : \leq_{\top} E \equiv$ case E of Err \Rightarrow True | OK $T \Rightarrow P, h \vdash v : \leq T$

notation (ASCII)

confT ($\langle -, \vdash - : \leq_{\top} \rightarrow [51, 51, 51, 51] 50$)

abbreviation

confTs :: 'c prog \Rightarrow heap \Rightarrow val list \Rightarrow ty_l \Rightarrow bool

($\langle -, \vdash - [: \leq_{\top}] \rightarrow [51, 51, 51, 51] 50$) **where**

$P, h \vdash vs [: \leq_{\top}] Ts \equiv$ list-all2 (confT $P h$) $vs Ts$

notation (ASCII)

confTs ($\langle -, \vdash - [: \leq_{\top}] \rightarrow [51, 51, 51, 51] 50$)

fun Called-context :: jvm-prog \Rightarrow cname \Rightarrow instr \Rightarrow bool **where**

Called-context $P C_0$ (New C') = ($C_0 = C'$) |

Called-context $P C_0$ (Getstatic $C F D$) = ($(C_0 = D) \wedge (\exists t. P \vdash C$ has $F, Static: t$ in D)) |

Called-context $P C_0$ (Putstatic $C F D$) = ($(C_0 = D) \wedge (\exists t. P \vdash C$ has $F, Static: t$ in D)) |

Called-context $P C_0$ (Invokestatic $C M n$)

$= (\exists Ts T m D. (C_0=D) \wedge P \vdash C \text{ sees } M, \text{Static}:Ts \rightarrow T = m \text{ in } D) \mid$
Called-context $P \text{ - -} = \text{False}$

abbreviation *Called-set* :: *instr set* **where**

Called-set $\equiv \{i. \exists C. i = \text{New } C\} \cup \{i. \exists C M n. i = \text{Invokestatic } C M n\}$
 $\cup \{i. \exists C F D. i = \text{Getstatic } C F D\} \cup \{i. \exists C F D. i = \text{Putstatic } C F D\}$

lemma *Called-context-Called-set*:

Called-context $P D i \implies i \in \text{Called-set}$ **by** (*cases* i , *auto*)

fun *valid-ics* :: *jvm-prog* \Rightarrow *heap* \Rightarrow *sheap* \Rightarrow *cname* \times *mname* \times *pc* \times *init-call-status* \Rightarrow *bool*

($\langle -, -, - \vdash_i \rightarrow [51, 51, 51, 51] \ 50 \rangle$ **where**
valid-ics $P h sh (C, M, pc, \text{Calling } C' Cs)$
 $= (\text{let } ins = \text{instrs-of } P C M \text{ in } \text{Called-context } P (\text{last } (C' \# Cs)) (ins!pc)$
 $\wedge \text{is-class } P C') \mid$
valid-ics $P h sh (C, M, pc, \text{Throwing } Cs a)$
 $= (\text{let } ins = \text{instrs-of } P C M \text{ in } \exists C1. \text{Called-context } P C1 (ins!pc)$
 $\wedge (\exists obj. h a = \text{Some } obj)) \mid$
valid-ics $P h sh (C, M, pc, \text{Called } Cs)$
 $= (\text{let } ins = \text{instrs-of } P C M$
 $\text{in } \exists C1 \text{ subj. } \text{Called-context } P C1 (ins!pc) \wedge sh C1 = \text{Some } subj) \mid$
valid-ics $P \text{ - -} = \text{True}$

definition *conf-f* :: *jvm-prog* \Rightarrow *heap* \Rightarrow *sheap* \Rightarrow *ty_i* \Rightarrow *bytecode* \Rightarrow *frame* \Rightarrow *bool*
where

conf-f $P h sh \equiv \lambda(ST, LT) \text{ is } (stk, loc, C, M, pc, ics).$
 $P, h \vdash stk [:\leq] ST \wedge P, h \vdash loc [:\leq_{\top}] LT \wedge pc < \text{size } is \wedge P, h, sh \vdash_i (C, M, pc, ics)$

lemma *conf-f-def2*:

conf-f $P h sh (ST, LT) \text{ is } (stk, loc, C, M, pc, ics) \equiv$
 $P, h \vdash stk [:\leq] ST \wedge P, h \vdash loc [:\leq_{\top}] LT \wedge pc < \text{size } is \wedge P, h, sh \vdash_i (C, M, pc, ics)$
by (*simp add: conf-f-def*)

primrec *conf-fs* :: [*jvm-prog*, *heap*, *sheap*, *ty_P*, *cname*, *mname*, *nat*, *ty*, *frame list*] \Rightarrow *bool*
where

conf-fs $P h sh \Phi C_0 M_0 n_0 T_0 [] = \text{True}$
 $\mid \text{conf-fs } P h sh \Phi C_0 M_0 n_0 T_0 (f \# frs) =$
 $(\text{let } (stk, loc, C, M, pc, ics) = f \text{ in}$
 $(\exists ST LT b Ts T mxs mxl_0 \text{ is } xt.$
 $\Phi C M ! pc = \text{Some } (ST, LT) \wedge$
 $(P \vdash C \text{ sees } M, b:Ts \rightarrow T = (mxs, mxl_0, is, xt) \text{ in } C) \wedge$
 $((\exists D Ts' T' m D'. M_0 \neq \text{clinit} \wedge ics = \text{No-ics} \wedge$
 $is!pc = \text{Invoke } M_0 n_0 \wedge ST!n_0 = \text{Class } D \wedge$
 $P \vdash D \text{ sees } M_0, \text{NonStatic}:Ts' \rightarrow T' = m \text{ in } D' \wedge P \vdash C_0 \preceq^* D' \wedge P \vdash T_0 \leq T') \vee$
 $(\exists D Ts' T' m. M_0 \neq \text{clinit} \wedge ics = \text{No-ics} \wedge$
 $is!pc = \text{Invokestatic } D M_0 n_0 \wedge$
 $P \vdash D \text{ sees } M_0, \text{Static}:Ts' \rightarrow T' = m \text{ in } C_0 \wedge P \vdash T_0 \leq T') \vee$
 $(M_0 = \text{clinit} \wedge (\exists Cs. ics = \text{Called } Cs))) \wedge$
 $\text{conf-f } P h sh (ST, LT) \text{ is } f \wedge \text{conf-fs } P h sh \Phi C M (\text{size } Ts) T \text{ frs}))$

fun *ics-classes* :: *init-call-status* \Rightarrow *cname list* **where**

ics-classes (*Calling* $C Cs$) = $Cs \mid$
ics-classes (*Throwing* $Cs a$) = $Cs \mid$
ics-classes (*Called* Cs) = $Cs \mid$

ics-classes - = []

fun *frame-clinit-classes* :: *frame* \Rightarrow *cname list* **where**
frame-clinit-classes (*stk,loc,C,M,pc,ics*) = (if *M=clinit* then [*C*] else []) @ *ics-classes ics*

abbreviation *clinit-classes* :: *frame list* \Rightarrow *cname list* **where**
clinit-classes frs \equiv *concat (map frame-clinit-classes frs)*

definition *distinct-clinit* :: *frame list* \Rightarrow *bool* **where**
distinct-clinit frs \equiv *distinct (clinit-classes frs)*

definition *conf-clinit* :: *jvm-prog* \Rightarrow *sheap* \Rightarrow *frame list* \Rightarrow *bool* **where**
conf-clinit P sh frs
 \equiv *distinct-clinit frs* \wedge
 $(\forall C \in \text{set}(\text{clinit-classes } frs). \text{is-class } P \ C \wedge (\exists sfs. sh \ C = \text{Some}(sfs, \text{Processing})))$

definition *correct-state* :: [*jvm-prog,ty_P,jvm-state*] \Rightarrow *bool* ($\langle -, - \vdash - \sqrt{\ } \rangle$ [61,0,0] 61)
where

correct-state P Φ \equiv $\lambda(xp,h,frs,sh).$
case xp of
None \Rightarrow (*case frs of*
 [] \Rightarrow *True*
 | (*f#fs*) \Rightarrow *P* \vdash *h* $\sqrt{\ } \wedge$ *P, h* \vdash_s *sh* $\sqrt{\ } \wedge$ *conf-clinit P sh frs* \wedge
 (*let (stk,loc,C,M,pc,ics) = f*
in $\exists b \ Ts \ T \ mxs \ mxl_0 \ is \ xt \ \tau.$
 (*P* \vdash *C* *sees* *M, b:Ts* \rightarrow *T = (mxs,mxl₀,is,xt) in C*) \wedge
 Φ C M ! pc = Some τ \wedge
conf-f P h sh τ is f \wedge *conf-fs P h sh Φ C M (size Ts) T fs*)
 | *Some x* \Rightarrow *frs = []*

notation

correct-state ($\langle -, - \mid - - [ok] \rangle$ [61,0,0] 61)

2.15.2 Values and \top

lemma *confT-Err [iff]*: *P, h* \vdash *x* : \leq_{\top} *Err*
by (*simp add: confT-def*)

lemma *confT-OK [iff]*: *P, h* \vdash *x* : \leq_{\top} *OK T = (P, h* \vdash *x* : \leq *T)*
by (*simp add: confT-def*)

lemma *confT-cases*:
P, h \vdash *x* : \leq_{\top} *X = (X = Err* \vee ($\exists T. X = OK \ T \wedge P, h \vdash x : \leq T$))
by (*cases X*) *auto*

lemma *confT-hext [intro?, trans]*:
 $\llbracket P, h \vdash x : \leq_{\top} T; h \sqsubseteq h' \rrbracket \Longrightarrow P, h' \vdash x : \leq_{\top} T$
by (*cases T*) (*blast intro: conf-hext*) $+$

lemma *confT-widen [intro?, trans]*:
 $\llbracket P, h \vdash x : \leq_{\top} T; P \vdash T \leq_{\top} T' \rrbracket \Longrightarrow P, h \vdash x : \leq_{\top} T'$
by (*cases T'*, *auto intro: conf-widen*)

2.15.3 Stack and Registers

lemmas *confTs-Cons1* [iff] = *list-all2-Cons1* [of *confT P h*] **for** $P h$

lemma *confTs-confT-sup*:

assumes *confTs*: $P, h \vdash \text{loc} [\leq_{\top}] LT$ **and** $n: n < \text{size } LT$ **and**

$LTn: LT!n = OK T$ **and** *subtype*: $P \vdash T \leq T'$

shows $P, h \vdash (\text{loc!}n) :\leq T'$

lemma *confTs-heat* [intro?]:

$P, h \vdash \text{loc} [\leq_{\top}] LT \implies h \leq h' \implies P, h' \vdash \text{loc} [\leq_{\top}] LT$

by (*fast elim: list-all2-mono confT-heat*)

lemma *confTs-widen* [intro?, trans]:

$P, h \vdash \text{loc} [\leq_{\top}] LT \implies P \vdash LT [\leq_{\top}] LT' \implies P, h \vdash \text{loc} [\leq_{\top}] LT'$

by (*rule list-all2-trans, rule confT-widen*)

lemma *confTs-map* [iff]:

$\bigwedge vs. (P, h \vdash vs [\leq_{\top}] \text{map } OK Ts) = (P, h \vdash vs [\leq] Ts)$

by (*induct Ts*) (*auto simp: list-all2-Cons2*)

lemma *reg-widen-Err* [iff]:

$\bigwedge LT. (P \vdash \text{replicate } n \text{ Err} [\leq_{\top}] LT) = (LT = \text{replicate } n \text{ Err})$

by (*induct n*) (*auto simp: list-all2-Cons1*)

lemma *confTs-Err* [iff]:

$P, h \vdash \text{replicate } n v [\leq_{\top}] \text{replicate } n \text{ Err}$

by (*induct n*) *auto*

2.15.4 valid init-call-status

lemma *valid-ics-shupd*:

assumes $P, h, sh \vdash_i (C, M, pc, ics)$ **and** *distinct* ($C' \# ics\text{-classes } ics$)

shows $P, h, sh(C' \mapsto (sfs, i')) \vdash_i (C, M, pc, ics)$

using *assms* **by**(*cases ics; clarsimp simp: fun-upd-apply*) *fastforce*

2.15.5 correct-frame

lemma *conf-f-Throwing*:

assumes *conf-f* $P h sh (ST, LT)$ *is* (*stk, loc, C, M, pc, Called Cs*)

and *is-class* $P C'$ **and** $h xcp = \text{Some } \text{obj}$ **and** $sh C' = \text{Some}(sfs, \text{Processing})$

shows *conf-f* $P h sh (ST, LT)$ *is* (*stk, loc, C, M, pc, Throwing (C' # Cs) xcp*)

using *assms* **by**(*auto simp: conf-f-def2*)

lemma *conf-f-shupd*:

assumes *conf-f* $P h sh (ST, LT)$ *ins* f

and $i = \text{Processing}$

$\vee (\text{distinct } (C \# ics\text{-classes } (ics\text{-of } f)) \wedge (\text{curr-method } f = \text{clinit} \longrightarrow C \neq \text{curr-class } f))$

shows *conf-f* $P h (sh(C \mapsto (sfs, i))) (ST, LT)$ *ins* f

using *assms*

by(*cases f, cases ics-of f; clarsimp simp: conf-f-def2 fun-upd-apply*) *fastforce+*

lemma *conf-f-shupd'*:

assumes *conf-f* $P h sh (ST, LT)$ *ins* f

and $sh C = \text{Some}(sfs, i)$

shows *conf-f* $P h (sh(C \mapsto (sfs', i))) (ST, LT)$ *ins* f

using *assms*

by(*cases f*, *cases ics-of f*; *clarsimp simp*: *conf-f-def2 fun-upd-apply*) *fastforce+*

2.15.6 correct-frames

lemmas [*simp del*] = *fun-upd-apply*

lemma *conf-fs-hext*:

$\bigwedge C M n T_r.$

$\llbracket \text{conf-fs } P h sh \Phi C M n T_r \text{ frs}; h \sqsubseteq h' \rrbracket \implies \text{conf-fs } P h' sh \Phi C M n T_r \text{ frs}$

lemma *conf-fs-shupd*:

assumes *conf-fs* $P h sh \Phi C_0 M n T$ *frs*

and *dist*: *distinct* ($C \# \text{clinit-classes } frs$)

shows *conf-fs* $P h (sh(C \mapsto (sfs, i))) \Phi C_0 M n T$ *frs*

using *assms* **proof**(*induct frs arbitrary*: $C_0 C M n T$)

case (*Cons f' frs'*)

then obtain $stk' loc' C' M' pc' ics'$ **where** $f': f' = (stk', loc', C', M', pc', ics')$ **by**(*cases f'*)

with *assms* *Cons* **obtain** $ST LT b Ts T1 mxs mxl_0 ins xt$ **where**

ty: $\Phi C' M' ! pc' = \text{Some}(ST, LT)$ **and**

meth: $P \vdash C'$ *sees* $M', b: Ts \rightarrow T1 = (mxs, mxl_0, ins, xt)$ **in** C' **and**

conf: *conf-f* $P h sh (ST, LT) ins f'$ **and**

confs: *conf-fs* $P h sh \Phi C' M' (size Ts) T1 frs'$ **by** *clarsimp*

from f' *Cons.prem*s(2) **have**

distinct ($C \# \text{ics-classes } (ics\text{-of } f')$) \wedge (*curr-method* $f' = \text{clinit} \rightarrow C \neq \text{curr-class } f'$)

by *fastforce*

with *conf-f-shupd*[**where** $C=C$, *OF conf*] **have**

conf': *conf-f* $P h (sh(C \mapsto (sfs, i))) (ST, LT) ins f'$ **by** *simp*

from *Cons.prem*s(2) **have** *dist'*: *distinct* ($C \# \text{clinit-classes } frs'$)

by(*auto simp*: *distinct-length-2-or-more*)

from *Cons.hyps*[*OF confs dist'*] **have**

confs': *conf-fs* $P h (sh(C \mapsto (sfs, i))) \Phi C' M' (length Ts) T1 frs'$ **by** *simp*

from *conf'* *confs'* *ty meth f' Cons.prem*s **show** *?case* **by**(*fastforce dest*: *sees-method-fun*)

qed(*simp*)

lemma *conf-fs-shupd'*:

assumes *conf-fs* $P h sh \Phi C_0 M n T$ *frs*

and *shC*: $sh C = \text{Some}(sfs, i)$

shows *conf-fs* $P h (sh(C \mapsto (sfs', i))) \Phi C_0 M n T$ *frs*

using *assms* **proof**(*induct frs arbitrary*: $C_0 C M n T sfs i sfs'$)

case (*Cons f' frs'*)

then obtain $stk' loc' C' M' pc' ics'$ **where** $f': f' = (stk', loc', C', M', pc', ics')$ **by**(*cases f'*)

with *assms* *Cons* **obtain** $ST LT b Ts T1 mxs mxl_0 ins xt$ **where**

ty: $\Phi C' M' ! pc' = \text{Some}(ST, LT)$ **and**

meth: $P \vdash C'$ *sees* $M', b: Ts \rightarrow T1 = (mxs, mxl_0, ins, xt)$ **in** C' **and**

conf: *conf-f* $P h sh (ST, LT) ins f'$ **and**

confs: *conf-fs* $P h sh \Phi C' M' (size Ts) T1 frs'$ **and**

shC': $sh C = \text{Some}(sfs, i)$ **by** *clarsimp*

have *conf'*: *conf-f* $P h (sh(C \mapsto (sfs', i))) (ST, LT) ins f'$ **by**(*rule conf-f-shupd'*[*OF conf shC'*])

from *Cons.hyps*[*OF confs shC'*] **have**
confs': *conf-fs P h (sh(C ↦ (sfs', i)))* Φ *C' M' (length Ts) T1 frs'* **by** *simp*

from *conf' confs' ty meth f' Cons.prem*s **show** *?case* **by**(*fastforce dest: sees-method-fun*)
qed(*simp*)

2.15.7 correctness wrt *clinit* use

lemma *conf-clinit-Cons*:
assumes *conf-clinit P sh (f#frs)*
shows *conf-clinit P sh frs*
proof –
from *assms* **have** *dist: distinct-clinit (f#frs)*
by(*cases curr-method f = clinit, auto simp: conf-clinit-def*)
then **have** *dist': distinct-clinit frs* **by**(*simp add: distinct-clinit-def*)

with *assms* **show** *?thesis* **by**(*cases frs; fastforce simp: conf-clinit-def*)
qed

lemma *conf-clinit-Cons-Cons*:
conf-clinit P sh (f'#f#frs) ⇒ conf-clinit P sh (f'#frs)
by(*auto simp: conf-clinit-def distinct-clinit-def*)

lemma *conf-clinit-diff*:
assumes *conf-clinit P sh ((stk,loc,C,M,pc,ics)#frs)*
shows *conf-clinit P sh ((stk',loc',C,M,pc',ics)#frs)*
using *assms* **by**(*cases M = clinit, simp-all add: conf-clinit-def distinct-clinit-def*)

lemma *conf-clinit-diff'*:
assumes *conf-clinit P sh ((stk,loc,C,M,pc,ics)#frs)*
shows *conf-clinit P sh ((stk',loc',C,M,pc',No-ics)#frs)*
using *assms* **by**(*cases M = clinit, simp-all add: conf-clinit-def distinct-clinit-def*)

lemma *conf-clinit-Called-Throwing*:
conf-clinit P sh ((stk',loc',C',clinit,pc',ics') # (stk,loc,C,M,pc,Called Cs) # fs)
 \Rightarrow *conf-clinit P sh ((stk,loc,C,M,pc,Throwing (C'#Cs) xcp) # fs)*
by(*simp add: conf-clinit-def distinct-clinit-def*)

lemma *conf-clinit-Throwing*:
conf-clinit P sh ((stk,loc,C,M,pc,Throwing (C'#Cs) xcp) # fs)
 \Rightarrow *conf-clinit P sh ((stk,loc,C,M,pc,Throwing Cs xcp) # fs)*
by(*simp add: conf-clinit-def distinct-clinit-def*)

lemma *conf-clinit-Called*:
 \llbracket *conf-clinit P sh ((stk,loc,C,M,pc,Called (C'#Cs)) # frs);*
P ⊢ C' sees clinit,Static: [] → Void=(mxs',mxl',ins',xt') in C' \rrbracket
 \Rightarrow *conf-clinit P sh (create-init-frame P C' # (stk,loc,C,M,pc,Called Cs) # frs)*
by(*simp add: conf-clinit-def distinct-clinit-def*)

lemma *conf-clinit-Cons-nclinit*:
assumes *conf-clinit P sh frs* **and** *nclinit: M ≠ clinit*
shows *conf-clinit P sh ((stk,loc,C,M,pc,No-ics) # frs)*
proof –
from *nclinit*

have *clinit-classes* $((stk, loc, C, M, pc, No-ics) \# frs) = clinit-classes\ frs$ **by** *simp*
with *assms* **show** *?thesis* **by**(*simp add: conf-clinit-def distinct-clinit-def*)
qed

lemma *conf-clinit-Invoke:*

assumes *conf-clinit P sh* $((stk, loc, C, M, pc, ics) \# frs)$ **and** $M' \neq clinit$
shows *conf-clinit P sh* $((stk', loc', C', M', pc', No-ics) \# (stk, loc, C, M, pc, No-ics) \# frs)$
using *assms conf-clinit-Cons-nclinit conf-clinit-diff'* **by** *auto*

lemma *conf-clinit-nProc-dist:*

assumes *conf-clinit P sh frs*
and $\forall sfs. sh\ C \neq Some(sfs, Processing)$
shows *distinct* $(C \# clinit-classes\ frs)$
using *assms* **by**(*auto simp: conf-clinit-def distinct-clinit-def*)

lemma *conf-clinit-shupd:*

assumes *conf-clinit P sh frs*
and *dist: distinct* $(C \# clinit-classes\ frs)$
shows *conf-clinit P* $(sh(C \mapsto (sfs, i)))\ frs$
using *assms* **by**(*simp add: conf-clinit-def fun-upd-apply*)

lemma *conf-clinit-shupd':*

assumes *conf-clinit P sh frs*
and $sh\ C = Some(sfs, i)$
shows *conf-clinit P* $(sh(C \mapsto (sfs', i)))\ frs$
using *assms* **by**(*fastforce simp: conf-clinit-def fun-upd-apply*)

lemma *conf-clinit-shupd-Called:*

assumes *conf-clinit P sh* $((stk, loc, C, M, pc, Calling\ C'\ Cs) \# frs)$
and *dist: distinct* $(C' \# clinit-classes\ ((stk, loc, C, M, pc, Calling\ C'\ Cs) \# frs))$
and *cls: is-class P C'*
shows *conf-clinit P* $(sh(C' \mapsto (sfs, Processing)))\ ((stk, loc, C, M, pc, Called\ (C' \# Cs)) \# frs)$
using *assms* **by**(*clarsimp simp: conf-clinit-def fun-upd-apply distinct-clinit-def*)

lemma *conf-clinit-shupd-Calling:*

assumes *conf-clinit P sh* $((stk, loc, C, M, pc, Calling\ C'\ Cs) \# frs)$
and *dist: distinct* $(C' \# clinit-classes\ ((stk, loc, C, M, pc, Calling\ C'\ Cs) \# frs))$
and *cls: is-class P C'*
shows *conf-clinit P* $(sh(C' \mapsto (sfs, Processing)))\ ((stk, loc, C, M, pc, Calling\ (fst(the(class\ P\ C')))\ (C' \# Cs)) \# frs)$
using *assms* **by**(*clarsimp simp: conf-clinit-def fun-upd-apply distinct-clinit-def*)

2.15.8 correct state

lemma *correct-state-Cons:*

assumes *cr: P, Φ |-* $(xp, h, f \# frs, sh)$ *[ok]*
shows $P, \Phi |-$ (xp, h, frs, sh) *[ok]*

proof –

from *cr* **have** *dist: conf-clinit P sh* $(f \# frs)$
by(*simp add: correct-state-def*)
then **have** *conf-clinit P sh frs* **by**(*rule conf-clinit-Cons*)

with *cr* **show** *?thesis* **by**(*cases frs; fastforce simp: correct-state-def*)

qed

lemma *correct-state-shupd*:

assumes *cs*: $P, \Phi \vdash (xp, h, frs, sh)$ [ok] and *shC*: $sh\ C = Some(sfs, i)$

and *dist*: *distinct* ($C \# clinit\ classes\ frs$)

shows $P, \Phi \vdash (xp, h, frs, sh(C \mapsto (sfs, i)))$ [ok]

using *assms*

proof(*cases xp*)

case *None* with *assms* show ?thesis

proof(*cases frs*)

case (*Cons f' frs'*)

let ?sh = $sh(C \mapsto (sfs, i'))$

obtain *stk' loc' C' M' pc' ics'* where $f' = (stk', loc', C', M', pc', ics')$ by(*cases f'*)

with *cs Cons None* obtain *b Ts T mxs mxl₀ ins xt ST LT* where

meth: $P \vdash C'$ sees $M', b: Ts \rightarrow T = (mxs, mxl_0, ins, xt)$ in C'

and *ty*: $\Phi\ C'\ M'\ !\ pc' = Some(ST, LT)$ and *conf*: *conf-f* $P\ h\ sh\ (ST, LT)$ *ins f'*

and *confs*: *conf-fs* $P\ h\ sh\ \Phi\ C'\ M'$ (*size Ts*) *T frs'*

and *confc*: *conf-clinit* $P\ sh\ frs$

and *h-ok*: $P \vdash h \checkmark$ and *sh-ok*: $P, h \vdash_s sh \checkmark$

by(*auto simp: correct-state-def*)

from *Cons dist* have *dist'*: *distinct* ($C \# clinit\ classes\ frs'$)

by(*auto simp: distinct-length-2-or-more*)

from *shconf-upd-obj*[*OF sh-ok shconfD*[*OF sh-ok shC*]] have *sh-ok'*: $P, h \vdash_s ?sh \checkmark$

by *simp*

from *conf f' valid-ics-shupd Cons dist* have *conf'*: *conf-f* $P\ h\ ?sh\ (ST, LT)$ *ins f'*

by(*auto simp: conf-f-def2 fun-upd-apply*)

have *confs'*: *conf-fs* $P\ h\ ?sh\ \Phi\ C'\ M'$ (*size Ts*) *T frs'* by(*rule conf-fs-shupd*[*OF confs dist'*])

have *confc'*: *conf-clinit* $P\ ?sh\ frs$ by(*rule conf-clinit-shupd*[*OF confc dist*])

with *h-ok sh-ok' meth ty conf' confs' f' Cons None* show ?thesis

by(*fastforce simp: correct-state-def*)

qed(*simp add: correct-state-def*)

qed(*simp add: correct-state-def*)

lemma *correct-state-Throwing-ex*:

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

shows $\bigwedge Cs\ a.\ ics = Throwing\ Cs\ a \implies \exists\ obj.\ h\ a = Some\ obj$

using *correct* by(*clarsimp simp: correct-state-def conf-f-def*)

end

2.16 Property preservation under *class-add*

theory *ClassAdd*

imports *BVConform*

begin

lemma *err-mono*: $A \subseteq B \implies \text{err } A \subseteq \text{err } B$
by(*unfold err-def*) *auto*

lemma *opt-mono*: $A \subseteq B \implies \text{opt } A \subseteq \text{opt } B$
by(*unfold opt-def*) *auto*

— adding a class in the simplest way

abbreviation *class-add* :: *jvm-prog* \Rightarrow *jvm-method cdecl* \Rightarrow *jvm-prog* **where**
class-add *P cd* \equiv *cd#P*

2.16.1 Fields

lemma *class-add-has-fields*:
assumes *fs*: $P \vdash D$ *has-fields FDTs* **and** *nc*: $\neg \text{is-class } P C$
shows *class-add* *P (C, cdec)* $\vdash D$ *has-fields FDTs*
using *assms*
proof(*induct rule: Fields.induct*)
 case (*has-fields-Object D fs ms FDTs*)
 from *has-fields-is-class-Object[OF fs]* *nc* **have** $C \neq \text{Object}$ **by** *fast*
 with *has-fields-Object* **show** *?case*
 by(*auto simp: class-def fun-upd-apply intro!: TypeRel.has-fields-Object*)
next
 case *rec: (has-fields-rec C1 D fs ms FDTs FDTs')*
 with *has-fields-is-class* **have** [*simp*]: $D \neq C$ **by** *auto*
 with *rec* **have** $C1 \neq C$ **by**(*clarsimp simp: is-class-def*)
 with *rec* **show** *?case*
 by(*auto simp: class-def fun-upd-apply intro: TypeRel.has-fields-rec*)
qed

lemma *class-add-has-fields-rev*:
 $\llbracket \text{class-add } P (C, cdec) \vdash D \text{ has-fields FDTs}; \neg P \vdash D \preceq^* C \rrbracket$
 $\implies P \vdash D \text{ has-fields FDTs}$
proof(*induct rule: Fields.induct*)
 case (*has-fields-Object D fs ms FDTs*)
 then **show** *?case* **by**(*auto simp: class-def fun-upd-apply intro!: TypeRel.has-fields-Object*)
next
 case *rec: (has-fields-rec C1 D fs ms FDTs FDTs')*
 then **have** *sub1*: $P \vdash C1 \prec^1 D$
 by(*auto simp: class-def fun-upd-apply intro!: subcls1I split: if-split-asm*)
 with *rec.prem*s **have** *cls*: $\neg P \vdash D \preceq^* C$ **by** (*meson converse-rtrancl-into-rtrancl*)
 with *cls rec* **show** *?case*
 by(*auto simp: class-def fun-upd-apply*
 intro: TypeRel.has-fields-rec split: if-split-asm)
qed

lemma *class-add-has-field*:
assumes $P \vdash C_0$ *has F,b:T in D* **and** $\neg \text{is-class } P C$
shows *class-add* *P (C, cdec)* $\vdash C_0$ *has F,b:T in D*
using *assms* **by**(*auto simp: has-field-def dest!: class-add-has-fields[of P C₀]*)

lemma *class-add-has-field-rev*:
assumes *has*: *class-add* *P (C, cdec)* $\vdash C_0$ *has F,b:T in D*

and $ncp: \bigwedge D'. P \vdash C_0 \preceq^* D' \implies D' \neq C$
shows $P \vdash C_0$ has $F, b: T$ in D
using *assms* **by**(*auto simp: has-field-def dest!: class-add-has-fields-rev*)

lemma *class-add-sees-field*:
assumes $P \vdash C_0$ sees $F, b: T$ in D **and** \neg *is-class* $P C$
shows *class-add* $P (C, cdec) \vdash C_0$ sees $F, b: T$ in D
using *assms* **by**(*auto simp: sees-field-def dest!: class-add-has-fields[of P C_0]*)

lemma *class-add-sees-field-rev*:
assumes *has: class-add* $P (C, cdec) \vdash C_0$ sees $F, b: T$ in D
and $ncp: \bigwedge D'. P \vdash C_0 \preceq^* D' \implies D' \neq C$
shows $P \vdash C_0$ sees $F, b: T$ in D
using *assms* **by**(*auto simp: sees-field-def dest!: class-add-has-fields-rev*)

lemma *class-add-field*:
assumes *fd: P* $\vdash C_0$ sees $F, b: T$ in D **and** \neg *is-class* $P C$
shows *field* $P C_0 F = \text{field } (\text{class-add } P (C, cdec)) C_0 F$
using *class-add-sees-field[OF assms, of cdec]* *fd* **by** *simp*

2.16.2 Methods

lemma *class-add-sees-methods*:
assumes *ms: P* $\vdash D$ sees *methods* Mm **and** *nc:* \neg *is-class* $P C$
shows *class-add* $P (C, cdec) \vdash D$ sees *methods* Mm
using *assms*
proof(*induct rule: Methods.induct*)
 case (*sees-methods-Object* $D fs ms Mm$)
 from *sees-methods-is-class-Object[OF ms]* *nc* **have** $C \neq \text{Object}$ **by** *fast*
 with *sees-methods-Object* **show** *?case*
 by(*auto simp: class-def fun-upd-apply intro!: TypeRel.sees-methods-Object*)
next
 case *rec: (sees-methods-rec* $C1 D fs ms Mm Mm')$
 with *sees-methods-is-class* **have** [*simp*]: $D \neq C$ **by** *auto*
 with *rec* **have** $C1 \neq C$ **by**(*clarsimp simp: is-class-def*)
 with *rec* **show** *?case*
 by(*auto simp: class-def fun-upd-apply intro: TypeRel.sees-methods-rec*)
qed

lemma *class-add-sees-methods-rev*:
 $\llbracket \text{class-add } P (C, cdec) \vdash D$ sees *methods* Mm ;
 $\bigwedge D'. P \vdash D \preceq^* D' \implies D' \neq C \rrbracket$
 $\implies P \vdash D$ sees *methods* Mm
proof(*induct rule: Methods.induct*)
 case (*sees-methods-Object* $D fs ms Mm$)
 then show *?case*
 by(*auto simp: class-def fun-upd-apply intro!: TypeRel.sees-methods-Object*)
next
 case *rec: (sees-methods-rec* $C1 D fs ms Mm Mm')$
 then have *sub1: P* $\vdash C1 \prec^1 D$
 by(*auto simp: class-def fun-upd-apply intro!: subcls1I*)
 have *cls: \bigwedge D'. P* $\vdash D \preceq^* D' \implies D' \neq C$
 proof –
 fix D' **assume** $P \vdash D \preceq^* D'$

with *sub1* **have** $P \vdash C1 \preceq^* D'$ **by** *simp*
with *rec.prem*s **show** $D' \neq C$ **by** *simp*
qed
with *cls rec* **show** *?case*
by(*auto simp: class-def fun-upd-apply intro: TypeRel.sees-methods-rec*)
qed

lemma *class-add-sees-methods-Obj*:

assumes $P \vdash \text{Object sees-methods } Mm$ **and** $nObj: C \neq \text{Object}$

shows $\text{class-add } P (C, cdec) \vdash \text{Object sees-methods } Mm$

proof –

from *assms* **obtain** $C' fs ms$ **where** $cls: \text{class } P \text{ Object} = \text{Some}(C', fs, ms)$
by(*auto elim!: Methods.cases*)
with $nObj$ **have** $cls': \text{class} (\text{class-add } P (C, cdec)) \text{ Object} = \text{Some}(C', fs, ms)$
by(*simp add: class-def fun-upd-apply*)
from *assms cls* **have** $Mm = \text{map-option } (\lambda m. (m, \text{Object})) \circ \text{map-of } ms$ **by**(*auto elim!: Methods.cases*)
with *assms cls'* **show** *?thesis*
by(*auto simp: is-class-def fun-upd-apply intro!: sees-methods-Object*)
qed

lemma *class-add-sees-methods-rev-Obj*:

assumes $\text{class-add } P (C, cdec) \vdash \text{Object sees-methods } Mm$ **and** $nObj: C \neq \text{Object}$

shows $P \vdash \text{Object sees-methods } Mm$

proof –

from *assms* **obtain** $C' fs ms$ **where** $cls: \text{class} (\text{class-add } P (C, cdec)) \text{ Object} = \text{Some}(C', fs, ms)$
by(*auto elim!: Methods.cases*)
with $nObj$ **have** $cls': \text{class } P \text{ Object} = \text{Some}(C', fs, ms)$
by(*simp add: class-def fun-upd-apply*)
from *assms cls* **have** $Mm = \text{map-option } (\lambda m. (m, \text{Object})) \circ \text{map-of } ms$ **by**(*auto elim!: Methods.cases*)
with *assms cls'* **show** *?thesis*
by(*auto simp: is-class-def fun-upd-apply intro!: sees-methods-Object*)
qed

lemma *class-add-sees-method*:

assumes $P \vdash C_0 \text{ sees } M_0, b : Ts \rightarrow T = m \text{ in } D$ **and** $\neg \text{is-class } P C$

shows $\text{class-add } P (C, cdec) \vdash C_0 \text{ sees } M_0, b : Ts \rightarrow T = m \text{ in } D$

using *assms* **by**(*auto simp: Method-def dest!: class-add-sees-methods[of P C₀]*)

lemma *class-add-method*:

assumes $md: P \vdash C_0 \text{ sees } M_0, b : Ts \rightarrow T = m \text{ in } D$ **and** $\neg \text{is-class } P C$

shows $\text{method } P C_0 M_0 = \text{method} (\text{class-add } P (C, cdec)) C_0 M_0$

using *class-add-sees-method[OF assms, of cdec]* *md* **by** *simp*

lemma *class-add-sees-method-rev*:

$\llbracket \text{class-add } P (C, cdec) \vdash C_0 \text{ sees } M_0, b : Ts \rightarrow T = m \text{ in } D;$

$\neg P \vdash C_0 \preceq^* C \rrbracket$

$\implies P \vdash C_0 \text{ sees } M_0, b : Ts \rightarrow T = m \text{ in } D$

by(*auto simp: Method-def dest!: class-add-sees-methods-rev*)

lemma *class-add-sees-method-Obj*:

$\llbracket P \vdash \text{Object sees } M_0, b : Ts \rightarrow T = m \text{ in } D; C \neq \text{Object} \rrbracket$

$\implies \text{class-add } P (C, cdec) \vdash \text{Object sees } M_0, b : Ts \rightarrow T = m \text{ in } D$

by(*auto simp: Method-def dest!: class-add-sees-methods-Obj*[**where** $P=P$])

lemma *class-add-sees-method-rev-Obj*:

$\llbracket \text{class-add } P (C, \text{cdec}) \vdash \text{Object sees } M_0, b : Ts \rightarrow T = m \text{ in } D; C \neq \text{Object} \rrbracket$
 $\implies P \vdash \text{Object sees } M_0, b : Ts \rightarrow T = m \text{ in } D$

by(*auto simp: Method-def dest!: class-add-sees-methods-rev-Obj*[**where** $P=P$])

2.16.3 Types and states

lemma *class-add-is-type*:

is-type $P T \implies \text{is-type} (\text{class-add } P (C, \text{cdec})) T$

by(*cases cdec, simp add: is-type-def is-class-def class-def fun-upd-apply split: ty.splits*)

lemma *class-add-types*:

types $P \subseteq \text{types} (\text{class-add } P (C, \text{cdec}))$

using *class-add-is-type* **by**(*cases cdec, clarsimp*)

lemma *class-add-states*:

states $P \text{ mxs } \text{m} \times \text{l} \subseteq \text{states} (\text{class-add } P (C, \text{cdec})) \text{ mxs } \text{m} \times \text{l}$

proof –

let $?A = \text{types } P$ **and** $?B = \text{types} (\text{class-add } P (C, \text{cdec}))$

have $ab: ?A \subseteq ?B$ **by**(*rule class-add-types*)

moreover have $\bigwedge n. \text{nlists } n ?A \subseteq \text{nlists } n ?B$ **using** ab **by**(*rule nlists-mono*)

moreover have $\text{nlists } \text{m} \times \text{l} (\text{err } ?A) \subseteq \text{nlists } \text{m} \times \text{l} (\text{err } ?B)$ **using** *err-mono[OF ab]* **by**(*rule nlists-mono*)

ultimately show *?thesis* **by**(*auto simp: JVM-states-unfold intro!: err-mono opt-mono*)

qed

lemma *class-add-check-types*:

check-types $P \text{ mxs } \text{m} \times \text{l} \tau s \implies \text{check-types} (\text{class-add } P (C, \text{cdec})) \text{ mxs } \text{m} \times \text{l} \tau s$

using *class-add-states* **by**(*fastforce simp: check-types-def*)

2.16.4 Subclasses and subtypes

lemma *class-add-subcls*:

$\llbracket P \vdash D \preceq^* D'; \neg \text{is-class } P C \rrbracket$

$\implies \text{class-add } P (C, \text{cdec}) \vdash D \preceq^* D'$

proof(*induct rule: rtrancl.induct*)

case (*rtrancl-into-rtrancl* $a b c$)

then have $b \neq C$ **by**(*clarsimp simp: is-class-def dest!: subcls1D*)

with *rtrancl-into-rtrancl* **show** *?case*

by(*fastforce dest!: subcls1D simp: class-def fun-upd-apply*

intro!: rtrancl-trans[of a b] subcls1I)

qed(*simp*)

lemma *class-add-subcls-rev*:

$\llbracket \text{class-add } P (C, \text{cdec}) \vdash D \preceq^* D'; \neg P \vdash D \preceq^* C \rrbracket$

$\implies P \vdash D \preceq^* D'$

proof(*induct rule: rtrancl.induct*)

case (*rtrancl-into-rtrancl* $a b c$)

then have $b \neq C$ **by**(*clarsimp simp: is-class-def dest!: subcls1D*)

with *rtrancl-into-rtrancl* **show** *?case*

by(*fastforce dest!: subcls1D simp: class-def fun-upd-apply*

intro!: rtrancl-trans[of a b] subcls1I)

qed(*simp*)

lemma *class-add-subtype*:

$\llbracket \text{subtype } P \ x \ y; \neg \text{is-class } P \ C \rrbracket$
 $\implies \text{subtype } (\text{class-add } P \ (C, \text{cdec})) \ x \ y$

proof(*induct rule: widen.induct*)

case (*widen-subcls C D*)
then show *?case using class-add-subcls by simp*
qed(*simp+*)

lemma *class-add-widens*:

$\llbracket P \vdash Ts \ [\leq] \ Ts'; \neg \text{is-class } P \ C \rrbracket$
 $\implies (\text{class-add } P \ (C, \text{cdec})) \vdash Ts \ [\leq] \ Ts'$

using *class-add-subtype by (metis (no-types) list-all2-mono)*

lemma *class-add-sup-ty-opt*:

$\llbracket P \vdash l1 \leq_{\top} l2; \neg \text{is-class } P \ C \rrbracket$
 $\implies \text{class-add } P \ (C, \text{cdec}) \vdash l1 \leq_{\top} l2$

using *class-add-subtype by (auto simp: sup-ty-opt-def Err.le-def lesub-def split: err.splits)*

lemma *class-add-sup-loc*:

$\llbracket P \vdash LT \ [\leq_{\top}] \ LT'; \neg \text{is-class } P \ C \rrbracket$
 $\implies \text{class-add } P \ (C, \text{cdec}) \vdash LT \ [\leq_{\top}] \ LT'$

using *class-add-sup-ty-opt[where P=P and C=C] by (simp add: list.rel-mono-strong)*

lemma *class-add-sup-state*:

$\llbracket P \vdash \tau \leq_i \tau'; \neg \text{is-class } P \ C \rrbracket$
 $\implies \text{class-add } P \ (C, \text{cdec}) \vdash \tau \leq_i \tau'$

using *class-add-subtype class-add-sup-ty-opt*

by(*auto simp: sup-state-def Listn.le-def Product.le-def lesub-def class-add-widens*
class-add-sup-ty-opt list-all2-mono)

lemma *class-add-sup-state-opt*:

$\llbracket P \vdash \tau \leq' \tau'; \neg \text{is-class } P \ C \rrbracket$
 $\implies \text{class-add } P \ (C, \text{cdec}) \vdash \tau \leq' \tau'$

by(*auto simp: sup-state-opt-def Opt.le-def lesub-def class-add-widens*
class-add-sup-ty-opt list-all2-mono)

2.16.5 Effect

lemma *class-add-is-relevant-class*:

$\llbracket \text{is-relevant-class } i \ P \ C_0; \neg \text{is-class } P \ C \rrbracket$
 $\implies \text{is-relevant-class } i \ (\text{class-add } P \ (C, \text{cdec})) \ C_0$

by(*cases i, auto simp: class-add-subcls*)

lemma *class-add-is-relevant-class-rev*:

assumes *irc: is-relevant-class i (class-add P (C, cdec)) C₀*

and *nep: $\bigwedge cd \ D'. cd \in \text{set } P \implies \neg P \vdash \text{fst } cd \preceq^* C$*

and *wfxp: wf-syscls P*

shows *is-relevant-class i P C₀*

using *assms*

proof(*cases i*)

case (*Getfield F D*) **with** *assms*

show *?thesis by (fastforce simp: wf-syscls-def sys-xcpts-def dest!: class-add-subcls-rev)*

next

```

  case (Putfield F D) with assms
  show ?thesis by(fastforce simp: wf-syscls-def sys-xcpts-def dest!: class-add-subcls-rev)
next
  case (Checkcast D) with assms
  show ?thesis by(fastforce simp: wf-syscls-def sys-xcpts-def dest!: class-add-subcls-rev)
qed(simp-all)

```

lemma *class-add-is-relevant-entry*:

```

[[ is-relevant-entry P i pc e; ¬ is-class P C ]]
  ⇒ is-relevant-entry (class-add P (C, cdec)) i pc e
by(clarsimp simp: is-relevant-entry-def class-add-is-relevant-class)

```

lemma *class-add-is-relevant-entry-rev*:

```

[[ is-relevant-entry (class-add P (C, cdec)) i pc e;
  ∧ cd D'. cd ∈ set P ⇒ ¬P ⊢ fst cd ≼* C;
  wf-syscls P ]]
  ⇒ is-relevant-entry P i pc e
by(auto simp: is-relevant-entry-def dest!: class-add-is-relevant-class-rev)

```

lemma *class-add-relevant-entries*:

```

¬ is-class P C
  ⇒ set (relevant-entries P i pc xt) ⊆ set (relevant-entries (class-add P (C, cdec)) i pc xt)
by(clarsimp simp: relevant-entries-def class-add-is-relevant-entry)

```

lemma *class-add-relevant-entries-eq*:

```

assumes wf: wf-prog wf-md P and nclass: ¬ is-class P C
shows relevant-entries P i pc xt = relevant-entries (class-add P (C, cdec)) i pc xt
proof -
  have ncp: ∧ cd D'. cd ∈ set P ⇒ ¬P ⊢ fst cd ≼* C
  by(rule wf-subcls-nCls'[OF assms])
  moreover from wf have wfsys: wf-syscls P by(simp add: wf-prog-def)
  moreover
  note class-add-is-relevant-entry[OF - nclass, of i pc - cdec]
      class-add-is-relevant-entry-rev[OF - ncp wfsys, of cdec i pc]
  ultimately show ?thesis by (metis filter-cong relevant-entries-def)
qed

```

lemma *class-add-norm-eff-pc*:

```

assumes ne: ∀ (pc',τ') ∈ set (norm-eff i P pc τ). pc' < mpc
shows ∀ (pc',τ') ∈ set (norm-eff i (class-add P (C, cdec)) pc τ). pc' < mpc
using assms by(cases i, auto simp: norm-eff-def)

```

lemma *class-add-norm-eff-sup-state-opt*:

```

assumes ne: ∀ (pc',τ') ∈ set (norm-eff i P pc τ). P ⊢ τ' ≤' τs!pc'
  and nclass: ¬ is-class P C and app: appi (i, P, pc, mxs, T, τ)
shows ∀ (pc',τ') ∈ set (norm-eff i (class-add P (C, cdec)) pc τ). (class-add P (C, cdec)) ⊢ τ' ≤' τs!pc'
proof -
  obtain ST LT where τ = (ST,LT) by(cases τ)
  with assms show ?thesis proof(cases i)
  qed(fastforce simp: norm-eff-def
    dest!: class-add-field[where cdec=cdec] class-add-method[where cdec=cdec]
    class-add-sup-loc[OF - nclass] class-add-subtype[OF - nclass]
    class-add-widens[OF - nclass] class-add-sup-state-opt[OF - nclass])

```

qed

lemma *class-add-xcpt-eff-eq*:

assumes *wf*: *wf-prog wf-md P* **and** *nclass*: \neg *is-class P C*

shows *xcpt-eff i P pc τ xt = xcpt-eff i (class-add P (C, cdec)) pc τ xt*

using *class-add-relevant-entries-eq*[*OF assms, of i pc xt cdec*] **by**(*cases τ , simp add: xcpt-eff-def*)

lemma *class-add-eff-pc*:

assumes *eff*: $\forall (pc', \tau') \in \text{set } (\text{eff } i P pc xt (\text{Some } \tau)). pc' < mpc$

and *wf*: *wf-prog wf-md P* **and** *nclass*: \neg *is-class P C*

shows $\forall (pc', \tau') \in \text{set } (\text{eff } i (\text{class-add } P (C, cdec)) pc xt (\text{Some } \tau)). pc' < mpc$

using *eff class-add-norm-eff-pc class-add-xcpt-eff-eq*[*OF wf nclass*]

by(*auto simp: norm-eff-def eff-def*)

lemma *class-add-eff-sup-state-opt*:

assumes *eff*: $\forall (pc', \tau') \in \text{set } (\text{eff } i P pc xt (\text{Some } \tau)). P \vdash \tau' \leq' \tau s ! pc'$

and *wf*: *wf-prog wf-md P* **and** *nclass*: \neg *is-class P C*

and *app*: *app_i (i, P, pc, mxs, T, τ)*

shows $\forall (pc', \tau') \in \text{set } (\text{eff } i (\text{class-add } P (C, cdec)) pc xt (\text{Some } \tau)).$

$(\text{class-add } P (C, cdec)) \vdash \tau' \leq' \tau s ! pc'$

proof –

from *eff have ne*: $\forall (pc', \tau') \in \text{set } (\text{norm-eff } i P pc \tau). P \vdash \tau' \leq' \tau s ! pc'$

by(*simp add: norm-eff-def eff-def*)

from *eff have* $\forall (pc', \tau') \in \text{set } (\text{xcpt-eff } i P pc \tau xt). P \vdash \tau' \leq' \tau s ! pc'$

by(*simp add: xcpt-eff-def eff-def*)

with *class-add-norm-eff-sup-state-opt*[*OF ne nclass app*]

class-add-xcpt-eff-eq[*OF wf nclass*]*class-add-sup-state-opt*[*OF - nclass*]

show *?thesis* **by**(*cases cdec, auto simp: eff-def norm-eff-def xcpt-app-def*)

qed

lemma *class-add-app_i*:

assumes *app_i (i, P, pc, mxs, T_r, ST, LT)* **and** \neg *is-class P C*

shows *app_i (i, class-add P (C, cdec), pc, mxs, T_r, ST, LT)*

using *assms*

proof(*cases i*)

case *New* **then show** *?thesis* **using** *assms* **by**(*fastforce simp: is-class-def class-def fun-upd-apply*)

next

case *Getfield* **then show** *?thesis* **using** *assms*

by(*auto simp: class-add-subtype dest!: class-add-sees-field*[**where** *P=P*])

next

case *Getstatic* **then show** *?thesis* **using** *assms* **by**(*auto dest!: class-add-sees-field*[**where** *P=P*])

next

case *Putfield* **then show** *?thesis* **using** *assms*

by(*auto dest!: class-add-subtype*[**where** *P=P*] *class-add-sees-field*[**where** *P=P*])

next

case *Putstatic* **then show** *?thesis* **using** *assms*

by(*auto dest!: class-add-subtype*[**where** *P=P*] *class-add-sees-field*[**where** *P=P*])

next

case *Checkcast* **then show** *?thesis* **using** *assms*

by(*clarsimp simp: is-class-def class-def fun-upd-apply*)

next

case *Invoke* **then show** *?thesis* **using** *assms*

by(*fastforce dest!: class-add-widens*[**where** *P=P*] *class-add-sees-method*[**where** *P=P*])

next

case *Invokestatic* **then show** *?thesis* **using** *assms*
by(*fastforce dest!*: *class-add-widens*[**where** $P=P$] *class-add-sees-method*[**where** $P=P$])
next
case *Return* **then show** *?thesis* **using** *assms* **by**(*clarsimp simp*: *class-add-subtype*)
qed(*simp+*)

lemma *class-add-xcpt-app*:
assumes *xa*: *xcpt-app* $i P pc mxs xt \tau$
and *wf*: *wf-prog wf-md* P **and** *nclass*: \neg *is-class* $P C$
shows *xcpt-app* i (*class-add* $P (C, cdec)$) $pc mxs xt \tau$
using *xa class-add-relevant-entries-eq*[*OF wf nclass*] *nclass*
by(*auto simp*: *xcpt-app-def is-class-def class-def fun-upd-apply*) *auto*

lemma *class-add-app*:
assumes *app*: *app* $i P mxs T pc mpc xt t$
and *wf*: *wf-prog wf-md* P **and** *nclass*: \neg *is-class* $P C$
shows *app* i (*class-add* $P (C, cdec)$) $mxs T pc mpc xt t$
proof(*cases t*)
case (*Some* τ)
let $?P =$ *class-add* $P (C, cdec)$
from *assms Some* **have** *eff*: $\forall (pc', \tau') \in \text{set } (eff\ i\ P\ pc\ xt\ [\tau]). pc' < mpc$ **by**(*simp add*: *app-def*)
from *assms Some* **have** *app_i*: *app_i* (i, P, pc, mxs, T, τ) **by**(*simp add*: *app-def*)
with *class-add-app_i*[*OF - nclass*] *Some* **have** *app_i* $(i, ?P, pc, mxs, T, \tau)$ **by**(*cases \tau, simp*)
moreover
from *app class-add-xcpt-app*[*OF - wf nclass*] *Some*
have *xcpt-app* $i ?P pc mxs xt \tau$ **by**(*simp add*: *app-def del*: *xcpt-app-def*)
moreover
from *app class-add-eff-pc*[*OF eff wf nclass*] *Some*
have $\forall (pc', \tau') \in \text{set } (eff\ i\ ?P\ pc\ xt\ t). pc' < mpc$ **by** *auto*
moreover note *app Some*
ultimately show *?thesis* **by**(*simp add*: *app-def*)
qed(*simp*)

2.16.6 Well-formedness and well-typedness

lemma *class-add-wf-mdecl*:
 \llbracket *wf-mdecl wf-md* $P C_0 md$;
 $\bigwedge C_0 md. wf-md\ P\ C_0\ md \implies wf-md\ (class-add\ P\ (C, cdec))\ C_0\ md \rrbracket$
 $\implies wf-mdecl\ wf-md\ (class-add\ P\ (C, cdec))\ C_0\ md$
by(*clarsimp simp*: *wf-mdecl-def class-add-is-type*)

lemma *class-add-wf-mdecl'*:
assumes *wfd*: *wf-mdecl wf-md* $P C_0 md$
and *ms*: $(C_0, S, fs, ms) \in \text{set } P$ **and** *md*: $md \in \text{set } ms$
and *wf-md'*: $\bigwedge C_0 S fs ms m. \llbracket (C_0, S, fs, ms) \in \text{set } P; m \in \text{set } ms \rrbracket \implies wf-md' (class-add\ P\ (C, cdec))\ C_0\ m$
shows *wf-mdecl wf-md'* (*class-add* $P (C, cdec)$) $C_0 md$
using *assms* **by**(*clarsimp simp*: *wf-mdecl-def class-add-is-type*)

lemma *class-add-wf-cdecl*:
assumes *wfcd*: *wf-cdecl wf-md* $P cd$ **and** *cdP*: $cd \in \text{set } P$
and *nep*: $\neg P \vdash fst\ cd \preceq^* C$ **and** *dist*: *distinct-fst* P
and *wfmd*: $\bigwedge C_0 md. wf-md\ P\ C_0\ md \implies wf-md\ (class-add\ P\ (C, cdec))\ C_0\ md$
and *nclass*: \neg *is-class* $P C$

shows *wf-cdecl wf-md (class-add P (C, cdec)) cd*

proof –

let $?P = \text{class-add } P \ (C, \text{cdec})$

obtain $C1 \ D \ fs \ ms$ where $[simp]: cd = (C1, (D, fs, ms))$ by (cases cd)

from *wfcd*

have $\forall f \in \text{set } fs. \text{wf-fdecl } ?P \ f$ by (auto simp: *wf-cdecl-def wf-fdecl-def class-add-is-type*)

moreover

from *wfcd wfmd class-add-wf-mdecl*

have $\forall m \in \text{set } ms. \text{wf-mdecl } \text{wf-md } ?P \ C1 \ m$ by (auto simp: *wf-cdecl-def*)

moreover

have $C1 \neq \text{Object} \implies \text{is-class } ?P \ D \wedge \neg ?P \vdash D \preceq^* C1$

$\wedge (\forall (M, b, Ts, T, m) \in \text{set } ms.$

$\forall D' \ b' \ Ts' \ T' \ m'. ?P \vdash D \text{ sees } M, b': Ts' \rightarrow T' = m' \text{ in } D' \longrightarrow$

$b = b' \wedge ?P \vdash Ts' [\leq] Ts \wedge ?P \vdash T \leq T')$

proof –

assume $nObj[simp]: C1 \neq \text{Object}$

with $cdP \ dist$ have $sub1: P \vdash C1 \prec^1 D$ by (auto simp: *class-def intro!: subcls1I map-of-SomeI*)

with ncp have $ncp': \neg P \vdash D \preceq^* C$ by (auto simp: *converse-rtrancl-into-rtrancl*)

with *wfcd*

have $clsD: \text{is-class } ?P \ D$

by (auto simp: *wf-cdecl-def is-class-def class-def fun-upd-apply*)

moreover

from *wfcd sub1*

have $\neg ?P \vdash D \preceq^* C1$ by (auto simp: *wf-cdecl-def dest!: class-add-subcls-rev[OF - ncp']*)

moreover

have $\bigwedge M \ b \ Ts \ T \ m \ D' \ b' \ Ts' \ T' \ m'. (M, b, Ts, T, m) \in \text{set } ms$

$\implies ?P \vdash D \text{ sees } M, b': Ts' \rightarrow T' = m' \text{ in } D'$

$\implies b = b' \wedge ?P \vdash Ts' [\leq] Ts \wedge ?P \vdash T \leq T'$

proof –

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

assume $ms: (M, b, Ts, T, m) \in \text{set } ms$ and $meth': ?P \vdash D \text{ sees } M, b': Ts' \rightarrow T' = m' \text{ in } D'$

with *sub1*

have $P \vdash D \text{ sees } M, b': Ts' \rightarrow T' = m' \text{ in } D'$

by (fastforce *dest!: class-add-sees-method-rev[OF - ncp']*)

moreover

with *wfcd ms meth'*

have $b = b' \wedge P \vdash Ts' [\leq] Ts \wedge P \vdash T \leq T'$

by (cases m' , fastforce *simp: wf-cdecl-def elim!: ballE[where $x=(M, b, Ts, T, m)$]*)

ultimately show $b = b' \wedge ?P \vdash Ts' [\leq] Ts \wedge ?P \vdash T \leq T'$

by (auto *dest!: class-add-subtype[OF - nclass] class-add-widens[OF - nclass]*)

qed

ultimately show *?thesis* by *clarsimp*

qed

moreover note *wfcd*

ultimately show *?thesis* by (*simp add: wf-cdecl-def*)

qed

lemma *class-add-wf-cdecl'*:

assumes *wfcd: wf-cdecl wf-md P cd* and $cdP: cd \in \text{set } P$

and $ncp: \neg P \vdash \text{fst } cd \preceq^* C$ and $dist: \text{distinct-fst } P$

and $wfmd: \bigwedge C_0 \ S \ fs \ ms \ m. [(C_0, S, fs, ms) \in \text{set } P; m \in \text{set } ms] \implies \text{wf-md}' (\text{class-add } P \ (C, \text{cdec}))$

$C_0 \ m$

and $nclass: \neg \text{is-class } P \ C$

shows *wf-cdecl wf-md' (class-add P (C, cdec)) cd*

proof –

let $?P = \text{class-add } P (C, \text{cdec})$
obtain $C1\ D\ fs\ ms$ **where** $[simp]: cd = (C1, (D, fs, ms))$ **by** $(\text{cases } cd)$
from $wfcd$
have $\forall f \in \text{set } fs. \text{wf-fdecl } ?P\ f$ **by** $(\text{auto } simp: \text{wf-cdecl-def } \text{wf-fdecl-def } \text{class-add-is-type})$
moreover
from $cdP\ wfcd\ wfmd$
have $\forall m \in \text{set } ms. \text{wf-mdecl } wf\text{-}md' ?P\ C1\ m$
by $(\text{auto } simp: \text{wf-cdecl-def } \text{wf-mdecl-def } \text{class-add-is-type})$
moreover
have $C1 \neq \text{Object} \implies \text{is-class } ?P\ D \wedge \neg ?P \vdash D \preceq^* C1$
 $\wedge (\forall (M, b, Ts, T, m) \in \text{set } ms.$
 $\quad \forall D' b' Ts' T' m'. ?P \vdash D \text{ sees } M, b': Ts' \rightarrow T' = m' \text{ in } D' \longrightarrow$
 $\quad b = b' \wedge ?P \vdash Ts' [\leq] Ts \wedge ?P \vdash T \leq T')$

proof –

assume $nObj[simp]: C1 \neq \text{Object}$
with $cdP\ \text{dist}$ **have** $sub1: P \vdash C1 \prec^1 D$ **by** $(\text{auto } simp: \text{class-def intro!}: \text{subcls1I } \text{map-of-SomeI})$
with ncp **have** $ncp': \neg P \vdash D \preceq^* C$ **by** $(\text{auto } simp: \text{converse-rtrancl-into-rtrancl})$
with $wfcd$
have $clsD: \text{is-class } ?P\ D$
by $(\text{auto } simp: \text{wf-cdecl-def } \text{is-class-def } \text{class-def fun-upd-apply})$
moreover
from $wfcd\ sub1$
have $\neg ?P \vdash D \preceq^* C1$ **by** $(\text{auto } simp: \text{wf-cdecl-def } \text{dest!}: \text{class-add-subcls-rev}[OF - ncp'])$
moreover
have $\bigwedge M\ b\ Ts\ T\ m\ D'\ b'\ Ts'\ T'\ m'. (M, b, Ts, T, m) \in \text{set } ms$
 $\implies ?P \vdash D \text{ sees } M, b': Ts' \rightarrow T' = m' \text{ in } D'$
 $\implies b = b' \wedge ?P \vdash Ts' [\leq] Ts \wedge ?P \vdash T \leq T'$

proof –

fix $M\ b\ Ts\ T\ m\ D'\ b'\ Ts'\ T'\ m'$
assume $ms: (M, b, Ts, T, m) \in \text{set } ms$ **and** $\text{meth}': ?P \vdash D \text{ sees } M, b': Ts' \rightarrow T' = m' \text{ in } D'$
with $sub1$
have $P \vdash D \text{ sees } M, b': Ts' \rightarrow T' = m' \text{ in } D'$
by $(\text{fastforce } \text{dest!}: \text{class-add-sees-method-rev}[OF - ncp'])$
moreover
with $wfcd\ ms\ \text{meth}'$
have $b = b' \wedge P \vdash Ts' [\leq] Ts \wedge P \vdash T \leq T'$
by $(\text{cases } m', \text{fastforce } simp: \text{wf-cdecl-def } \text{elim!}: \text{ballE}[\text{where } x = (M, b, Ts, T, m)])$
ultimately show $b = b' \wedge ?P \vdash Ts' [\leq] Ts \wedge ?P \vdash T \leq T'$
by $(\text{auto } \text{dest!}: \text{class-add-subtype}[OF - nclass] \text{class-add-widens}[OF - nclass])$

qed

ultimately show $?thesis$ **by** clarsimp

qed

moreover note $wfcd$

ultimately show $?thesis$ **by** $(simp\ \text{add}: \text{wf-cdecl-def})$

qed

lemma $\text{class-add-wt-start}$:

$\llbracket \text{wt-start } P\ C_0\ b\ Ts\ mxl\ \tau s; \neg \text{is-class } P\ C \rrbracket$
 $\implies \text{wt-start } (\text{class-add } P (C, \text{cdec}))\ C_0\ b\ Ts\ mxl\ \tau s$

using $\text{class-add-sup-state-opt}$ **by** $(\text{clarsimp } simp: \text{wt-start-def } \text{split}: \text{staticb.splits})$

lemma $\text{class-add-wt-instr}$:

assumes $wti: P, T, mxs, mpc, xt \vdash i, pc :: \tau s$

and *wf*: *wf-prog wf-md P* **and** *nclass*: \neg *is-class P C*
shows *class-add P (C, cdec), T, mxs, mpc, xt* \vdash *i, pc* :: τs
proof –
let *?P* = *class-add P (C, cdec)*
from *wti* **have** *eff*: $\forall (pc', \tau') \in \text{set} (\text{eff } i P pc xt (\tau s ! pc)). P \vdash \tau' \leq' \tau s ! pc'$
by(*simp add: wt-instr-def*)
from *wti* **have** *app_i*: $\tau s ! pc \neq \text{None} \implies \text{app}_i (i, P, pc, mxs, T, \text{the } (\tau s ! pc))$
by(*simp add: wt-instr-def app-def*)
from *wti* *class-add-app[OF - wf nclass]*
have *app i ?P mxs T pc mpc xt* $(\tau s ! pc)$ **by**(*simp add: wt-instr-def*)
moreover
have $\forall (pc', \tau') \in \text{set} (\text{eff } i ?P pc xt (\tau s ! pc)). ?P \vdash \tau' \leq' \tau s ! pc'$
proof(*cases* $\tau s ! pc$)
case *Some* **with** *eff class-add-eff-sup-state-opt[OF - wf nclass app_i]* **show** *?thesis* **by** *auto*
qed(*simp add: eff-def*)
moreover **note** *wti*
ultimately **show** *?thesis* **by**(*clarsimp simp: wt-instr-def*)
qed

lemma *class-add-wt-method*:

assumes *wtm*: *wt-method P C₀ b Ts T_r mxs mxl₀ is xt* $(\Phi C_0 M_0)$
and *wf*: *wf-prog wf-md P* **and** *nclass*: \neg *is-class P C*
shows *wt-method (class-add P (C, cdec)) C₀ b Ts T_r mxs mxl₀ is xt* $(\Phi C_0 M_0)$
proof –
let *?P* = *class-add P (C, cdec)*
let *? τs* = $\Phi C_0 M_0$
from *wtm* *class-add-check-types*
have *check-types ?P mxs* $((\text{case } b \text{ of } \text{Static} \Rightarrow 0 \mid \text{NonStatic} \Rightarrow 1) + \text{size } Ts + mxl_0)$ $(\text{map } OK ?\tau s)$
by(*simp add: wt-method-def*)
moreover
from *wtm* *class-add-wt-start nclass*
have *wt-start ?P C₀ b Ts mxl₀ ? τs* **by**(*simp add: wt-method-def*)
moreover
from *wtm* *class-add-wt-instr[OF - wf nclass]*
have $\forall pc < \text{size } is. ?P, T_r, mxs, \text{size } is, xt \vdash is ! pc, pc :: ?\tau s$ **by**(*clarsimp simp: wt-method-def*)
moreover **note** *wtm*
ultimately
show *?thesis* **by**(*clarsimp simp: wt-method-def*)
qed

lemma *class-add-wt-method'*:

$\llbracket (\lambda P C (M, b, Ts, T_r, (mxs, mxl_0, is, xt)). \text{wt-method } P C b Ts T_r mxs mxl_0 is xt (\Phi C M)) P C_0 md;$
 $\text{wf-prog wf-md } P; \neg \text{is-class } P C \rrbracket$
 $\implies (\lambda P C (M, b, Ts, T_r, (mxs, mxl_0, is, xt)). \text{wt-method } P C b Ts T_r mxs mxl_0 is xt (\Phi C M))$
 $(\text{class-add } P (C, cdec)) C_0 md$
by(*clarsimp simp: class-add-wt-method*)

2.16.7 distinct-fst

lemma *class-add-distinct-fst*:

$\llbracket \text{distinct-fst } P; \neg \text{is-class } P C \rrbracket$
 $\implies \text{distinct-fst } (\text{class-add } P (C, cdec))$
by(*clarsimp simp: distinct-fst-def is-class-def class-def*)

2.16.8 Conformance

lemma *class-add-conf*:

$\llbracket P, h \vdash v : \leq T; \neg \text{is-class } P \ C \rrbracket$
 $\implies \text{class-add } P \ (C, \text{cdec}), h \vdash v : \leq T$
by(*clarsimp simp: conf-def class-add-subtype*)

lemma *class-add-oconf*:

fixes *obj::obj*

assumes *oc*: $P, h \vdash \text{obj} \checkmark$ **and** *ns*: $\neg \text{is-class } P \ C$

and *ncp*: $\bigwedge D'. P \vdash \text{fst}(\text{obj}) \preceq^* D' \implies D' \neq C$

shows $(\text{class-add } P \ (C, \text{cdec})), h \vdash \text{obj} \checkmark$

proof –

obtain $C_0 \text{ fs}$ **where** [*simp*]: $\text{obj} = (C_0, \text{fs})$ **by**(*cases obj*)

from *oc* **have**

oc' : $\bigwedge F \ D \ T. P \vdash C_0 \text{ has } F, \text{NonStatic}: T \text{ in } D \implies (\exists v. \text{fs } (F, D) = [v] \wedge P, h \vdash v : \leq T)$

by(*simp add: oconf-def*)

have $\bigwedge F \ D \ T. \text{class-add } P \ (C, \text{cdec}) \vdash C_0 \text{ has } F, \text{NonStatic}: T \text{ in } D$

$\implies \exists v. \text{fs}(F, D) = \text{Some } v \wedge \text{class-add } P \ (C, \text{cdec}), h \vdash v : \leq T$

proof –

fix $F \ D \ T$ **assume** $\text{class-add } P \ (C, \text{cdec}) \vdash C_0 \text{ has } F, \text{NonStatic}: T \text{ in } D$

with *class-add-has-field-rev*[*OF - ncp*] **have** *meth*: $P \vdash C_0 \text{ has } F, \text{NonStatic}: T \text{ in } D$ **by** *simp*

then show $\exists v. \text{fs}(F, D) = \text{Some } v \wedge \text{class-add } P \ (C, \text{cdec}), h \vdash v : \leq T$

using *oc'*[*OF meth*] *class-add-conf*[*OF - ns*] **by**(*fastforce simp: oconf-def*)

qed

then show *?thesis* **by**(*simp add: oconf-def*)

qed

lemma *class-add-soconf*:

assumes *soc*: $P, h, C_0 \vdash_s \text{sfs} \checkmark$ **and** *ns*: $\neg \text{is-class } P \ C$

and *ncp*: $\bigwedge D'. P \vdash C_0 \preceq^* D' \implies D' \neq C$

shows $(\text{class-add } P \ (C, \text{cdec})), h, C_0 \vdash_s \text{sfs} \checkmark$

proof –

from *soc* **have**

oc' : $\bigwedge F \ T. P \vdash C_0 \text{ has } F, \text{Static}: T \text{ in } C_0 \implies (\exists v. \text{sfs } F = [v] \wedge P, h \vdash v : \leq T)$

by(*simp add: soconf-def*)

have $\bigwedge F \ T. \text{class-add } P \ (C, \text{cdec}) \vdash C_0 \text{ has } F, \text{Static}: T \text{ in } C_0$

$\implies \exists v. \text{sfs } F = \text{Some } v \wedge \text{class-add } P \ (C, \text{cdec}), h \vdash v : \leq T$

proof –

fix $F \ T$ **assume** $\text{class-add } P \ (C, \text{cdec}) \vdash C_0 \text{ has } F, \text{Static}: T \text{ in } C_0$

with *class-add-has-field-rev*[*OF - ncp*] **have** *meth*: $P \vdash C_0 \text{ has } F, \text{Static}: T \text{ in } C_0$ **by** *simp*

then show $\exists v. \text{sfs } F = \text{Some } v \wedge \text{class-add } P \ (C, \text{cdec}), h \vdash v : \leq T$

using *oc'*[*OF meth*] *class-add-conf*[*OF - ns*] **by**(*fastforce simp: soconf-def*)

qed

then show *?thesis* **by**(*simp add: soconf-def*)

qed

lemma *class-add-hconf*:

assumes $P \vdash h \checkmark$ **and** $\neg \text{is-class } P \ C$

and $\bigwedge a \ \text{obj } D'. h \ a = \text{Some } \text{obj} \implies P \vdash \text{fst}(\text{obj}) \preceq^* D' \implies D' \neq C$

shows $\text{class-add } P \ (C, \text{cdec}) \vdash h \checkmark$

using *assms* **by**(*auto simp: hconf-def intro!: class-add-oconf*)

lemma *class-add-hconf-wf*:

assumes $wf: wf\text{-prog } wf\text{-md } P$ **and** $P \vdash h \checkmark$ **and** $\neg is\text{-class } P C$
and $\bigwedge a \text{ obj. } h a = \text{Some obj} \implies fst(obj) \neq C$
shows $class\text{-add } P (C, cdec) \vdash h \checkmark$
using $wf\text{-subcls-nCls}[OF wf]$ **assms** **by**($fastforce simp: hconf\text{-def intro!}: class\text{-add-oconf}$)

lemma $class\text{-add-shconf}$:
assumes $P, h \vdash_s sh \checkmark$ **and** $ns: \neg is\text{-class } P C$
and $\bigwedge C \text{ subj } D'. sh C = \text{Some subj} \implies P \vdash C \preceq^* D' \implies D' \neq C$
shows $class\text{-add } P (C, cdec), h \vdash_s sh \checkmark$
using **assms** **by**($fastforce simp: shconf\text{-def}$)

lemma $class\text{-add-shconf-wf}$:
assumes $wf: wf\text{-prog } wf\text{-md } P$ **and** $P, h \vdash_s sh \checkmark$ **and** $\neg is\text{-class } P C$
and $\bigwedge C \text{ subj. } sh C = \text{Some subj} \implies C \neq C$
shows $class\text{-add } P (C, cdec), h \vdash_s sh \checkmark$
using $wf\text{-subcls-nCls}[OF wf]$ **assms** **by**($fastforce simp: shconf\text{-def}$)

end

2.17 Properties and types of the starting program

theory $StartProg$
imports $ClassAdd$
begin

lemmas $wt\text{-defs} = correct\text{-state-def conf-f-def wt-instr-def eff-def norm-eff-def app-def xcpt-app-def}$

declare $wt\text{-defs}$ [$simp$] — removed from $simp$ at the end of file
declare $start\text{-class-def}$ [$simp$]

2.17.1 Types

abbreviation $start\text{-}\varphi_m :: ty_m$ **where**
 $start\text{-}\varphi_m \equiv [Some([], []), Some([Void], [])]$

fun $\Phi\text{-start} :: ty_P \Rightarrow ty_P$ **where**
 $\Phi\text{-start } \Phi C M = (if C = Start \wedge (M = start\text{-}m \vee M = clinit) then start\text{-}\varphi_m else \Phi C M)$

lemma $\Phi\text{-start}$: $\bigwedge C. C \neq Start \implies \Phi\text{-start } \Phi C = \Phi C$
 $\Phi\text{-start } \Phi Start start\text{-}m = start\text{-}\varphi_m$ $\Phi\text{-start } \Phi Start clinit = start\text{-}\varphi_m$
by $auto$

lemma $check\text{-types-}\varphi_m$: $check\text{-types} (start\text{-prog } P C M) 1 0 (map OK start\text{-}\varphi_m)$
by ($auto simp: check\text{-types-def JVM-states-unfold}$)

2.17.2 Some simple properties

lemma $preallocated\text{-start-state}$: $start\text{-state } P = \sigma \implies preallocated (fst(snd \sigma))$
using $preallocated\text{-start}[of P]$ **by**($auto simp: start\text{-state-def split-beta}$)

lemma $start\text{-prog-Start-super}$: $start\text{-prog } P C M \vdash Start \prec^1 Object$
by($auto intro!: subcls1I simp: class\text{-def fun-upd-apply}$)

lemma *start-prog-Start-fields*:

start-prog $P\ C\ M \vdash$ *Start has-fields FDTs* \implies *map-of FDTs* (F, Start) = *None*
by(*drule* *Fields.cases*, *auto simp: class-def fun-upd-apply Object-fields*)

lemma *start-prog-Start-soconf*:

(*start-prog* $P\ C\ M$), $h, \text{Start} \vdash_s$ *Map.empty* \checkmark
by(*simp add: soconf-def has-field-def start-prog-Start-fields*)

lemma *start-prog-start-shconf*:

start-prog $P\ C\ M, \text{start-heap } P \vdash_s$ *start-sheap* \checkmark

2.17.3 Well-typed and well-formed

lemma *start-wt-method*:

assumes $P \vdash C$ *sees* $M, \text{Static} : [] \rightarrow \text{Void} = m$ *in* D **and** $M \neq \text{clinit}$ **and** \neg *is-class* P *Start*
shows *wt-method* (*start-prog* $P\ C\ M$) *Start* $\text{Static}\ []\ \text{Void}\ 1\ 0$ [*Invokestatic* $C\ M\ 0, \text{Return}$] $[]$ *start- φ_m*
(*is wt-method* $?P\ ?C\ ?b\ ?Ts\ ?T_r\ ?m_x_s\ ?m_x_l_0\ ?is\ ?x_t\ ?\tau_s$)

proof –

let $?cdec = (\text{Object}, [], [\text{start-method } C\ M, \text{start-clinit}])$
obtain $m_x_s\ m_x_l\ ins\ x_t$ **where** $m: m = (m_x_s, m_x_l, ins, x_t)$ **by**(*cases* m)
have *ca-sees*: *class-add* P (*Start*, $?cdec$) $\vdash C$ *sees* $M, \text{Static} : [] \rightarrow \text{Void} = m$ *in* D
by(*rule class-add-sees-method[OF assms(1,3)]*)
have $\bigwedge pc. pc < \text{size } ?is \implies ?P, ?T_r, ?m_x_s, \text{size } ?is, ?x_t \vdash ?is!pc, pc :: ?\tau_s$

proof –

fix pc **assume** $pc < \text{size } ?is$
then show $?P, ?T_r, ?m_x_s, \text{size } ?is, ?x_t \vdash ?is!pc, pc :: ?\tau_s$
proof(*cases* $pc = 0$)
case *True* **with** *assms* m *ca-sees* **show** *thesis*
by(*fastforce simp: wt-method-def wt-start-def relevant-entries-def*
is-relevant-entry-def xcpt-eff-def)

next

case *False* **with** pc **show** *thesis*
by(*simp add: wt-method-def wt-start-def relevant-entries-def*
is-relevant-entry-def xcpt-eff-def)

qed

qed

with *assms* *check-types- φ_m* **show** *thesis* **by**(*simp add: wt-method-def wt-start-def*)

qed

lemma *start-clinit-wt-method*:

assumes $P \vdash C$ *sees* $M, \text{Static} : [] \rightarrow \text{Void} = m$ *in* D **and** $M \neq \text{clinit}$ **and** \neg *is-class* P *Start*
shows *wt-method* (*start-prog* $P\ C\ M$) *Start* $\text{Static}\ []\ \text{Void}\ 1\ 0$ [*Push Unit, Return*] $[]$ *start- φ_m*
(*is wt-method* $?P\ ?C\ ?b\ ?Ts\ ?T_r\ ?m_x_s\ ?m_x_l_0\ ?is\ ?x_t\ ?\tau_s$)

proof –

let $?cdec = (\text{Object}, [], [\text{start-method } C\ M, \text{start-clinit}])$
obtain $m_x_s\ m_x_l\ ins\ x_t$ **where** $m: m = (m_x_s, m_x_l, ins, x_t)$ **by**(*cases* m)
have *ca-sees*: *class-add* P (*Start*, $?cdec$) $\vdash C$ *sees* $M, \text{Static} : [] \rightarrow \text{Void} = m$ *in* D
by(*rule class-add-sees-method[OF assms(1,3)]*)
have $\bigwedge pc. pc < \text{size } ?is \implies ?P, ?T_r, ?m_x_s, \text{size } ?is, ?x_t \vdash ?is!pc, pc :: ?\tau_s$

proof –

fix pc **assume** $pc < \text{size } ?is$
then show $?P, ?T_r, ?m_x_s, \text{size } ?is, ?x_t \vdash ?is!pc, pc :: ?\tau_s$
proof(*cases* $pc = 0$)
case *True* **with** *assms* m *ca-sees* **show** *thesis*

```

  by(fastforce simp: wt-method-def wt-start-def relevant-entries-def
      is-relevant-entry-def xcpt-eff-def)
next
case False with pc show ?thesis
  by(simp add: wt-method-def wt-start-def relevant-entries-def
      is-relevant-entry-def xcpt-eff-def)
qed
qed
with assms check-types- $\varphi_m$  show ?thesis by(simp add: wt-method-def wt-start-def)
qed

```

lemma *start-class-wf*:

assumes $P \vdash C$ sees M , $Static : [] \rightarrow Void = m$ in D

and $M \neq clinit$ **and** $\neg is-class P Start$

and $\Phi Start$ $start-m = start-\varphi_m$ **and** $\Phi Start$ $clinit = start-\varphi_m$

and $is-class P Object$

and $\bigwedge b' Ts' T' m' D'. P \vdash Object$ sees $start-m$, $b' : Ts' \rightarrow T' = m'$ in D'

$\implies b' = Static \wedge Ts' = [] \wedge T' = Void$

and $\bigwedge b' Ts' T' m' D'. P \vdash Object$ sees $clinit$, $b' : Ts' \rightarrow T' = m'$ in D'

$\implies b' = Static \wedge Ts' = [] \wedge T' = Void$

shows $wf-cdecl (\lambda P C (M, b, Ts, T_r, (mxs, mxl_0, is, xt)). wt-method P C b Ts T_r mxs mxl_0 is xt (\Phi C M))$

$(start-prog P C M) (start-class C M)$

proof –

from *assms start-wt-method start-clinit-wt-method class-add-sees-method-rev-Obj* **where** $P=P$ **and** $C=Start$

show *?thesis*

by(*auto simp: start-method-def wf-cdecl-def wf-fdecl-def wf-mdecl-def*
is-class-def class-def fun-upd-apply wf-clinit-def) *fast+*

qed

lemma *start-prog-wf-jvm-prog-phi*:

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

and $nstart: \neg is-class P Start$

and $meth: P \vdash C$ sees M , $Static : [] \rightarrow Void = m$ in D **and** $nclinit: M \neq clinit$

and $\Phi: \bigwedge C. C \neq Start \implies \Phi' C = \Phi C$

and $\Phi': \Phi' Start$ $start-m = start-\varphi_m$ $\Phi' Start$ $clinit = start-\varphi_m$

and $Obj-start-m: \bigwedge b' Ts' T' m' D'. P \vdash Object$ sees $start-m$, $b' : Ts' \rightarrow T' = m'$ in D'

$\implies b' = Static \wedge Ts' = [] \wedge T' = Void$

shows $wf-jvm-prog_{\Phi'} (start-prog P C M)$

proof –

let $?wf-md = (\lambda P C (M, b, Ts, T_r, (mxs, mxl_0, is, xt)). wt-method P C b Ts T_r mxs mxl_0 is xt (\Phi C M))$

let $?wf-md' = (\lambda P C (M, b, Ts, T_r, (mxs, mxl_0, is, xt)). wt-method P C b Ts T_r mxs mxl_0 is xt (\Phi' C M))$

from *wtp* **have** $wf: wf-prog ?wf-md P$ **by**(*simp add: wf-jvm-prog-phi-def*)

from *wf-subcls-nCls'* $[OF wf nstart]$

have $nsp: \bigwedge cd D'. cd \in set P \implies \neg P \vdash fst cd \preceq^* Start$ **by** *simp*

have $wf-md'$:

$\bigwedge C_0 S fs ms m. (C_0, S, fs, ms) \in set P \implies m \in set ms \implies ?wf-md' (start-prog P C M) C_0 m$

proof –

fix $C_0 S fs ms m$ **assume** $asms: (C_0, S, fs, ms) \in set P$ $m \in set ms$

with $nstart$ **have** $ns: C_0 \neq Start$ **by**(*auto simp: is-class-def class-def dest: weak-map-of-SomeI*)

from *wf asms* **have** $?wf-md P C_0 m$ **by**(*auto simp: wf-prog-def wf-cdecl-def wf-mdecl-def*)

with $\Phi[OF\ ns]$ *class-add-wt-method*[$OF - wf\ nstart$]
show $?wf\ md' (start\ prog\ P\ C\ M)\ C_0\ m$ **by** *fastforce*
qed
from *wtp* **have** $a1: is\ class\ P\ Object$ **by** (*simp add: wf-jvm-prog-phi-def*)
with *wf-sees-clinit*[**where** $P=P$ **and** $C=Object$] *wtp*
have $a2: \bigwedge b' Ts' T' m' D'. P \vdash Object\ sees\ clinit, b' : Ts' \rightarrow T' = m' in\ D'$
 $\implies b' = Static \wedge Ts' = [] \wedge T' = Void$
by(*fastforce simp: wf-jvm-prog-phi-def is-class-def dest: sees-method-fun*)
from *wf* **have** *dist: distinct-fst P* **by** (*simp add: wf-prog-def*)
with *class-add-distinct-fst*[$OF - nstart$] **have** *distinct-fst (start-prog P C M)* **by** *simp*
moreover from *wf* **have** *wf-syscls (start-prog P C M)* **by**(*simp add: wf-prog-def wf-syscls-def*)
moreover
from *class-add-wf-cdecl'*[**where** $wf\ md'=?wf\ md', OF - - nsp\ dist$] *wf-md' nstart wf*
have $\bigwedge c. c \in set\ P \implies wf\ cdecl\ ?wf\ md' (start\ prog\ P\ C\ M)\ c$ **by**(*fastforce simp: wf-prog-def*)
moreover from *start-class-wf*[$OF\ meth$] *nclinit nstart Φ' a1 Obj-start-m a2*
have *wf-cdecl ?wf-md' (start-prog P C M) (start-class C M)* **by** *simp*
ultimately show *?thesis* **by**(*simp add: wf-jvm-prog-phi-def wf-prog-def*)
qed

lemma *start-prog-wf-jvm-prog:*

assumes *wf: wf-jvm-prog P*

and *nstart: $\neg is\ class\ P\ Start$*

and *meth: $P \vdash C\ sees\ M, Static : [] \rightarrow Void = m\ in\ D$ **and** *nclinit: $M \neq clinit$**

and *Obj-start-m: $\bigwedge b' Ts' T' m' D'. P \vdash Object\ sees\ start\ m, b' : Ts' \rightarrow T' = m' in\ D'$*
 $\implies b' = Static \wedge Ts' = [] \wedge T' = Void$

shows *wf-jvm-prog (start-prog P C M)*

proof –

from *wf* **obtain** Φ **where** *wtp: wf-jvm-prog Φ P* **by**(*clarsimp simp: wf-jvm-prog-def*)

let $?\Phi' = \lambda C f. if\ C = Start \wedge (f = start\ m \vee f = clinit)$ **then** *start- φ_m* **else** $\Phi\ C\ f$

from *start-prog-wf-jvm-prog-phi*[$OF\ wtp\ nstart\ meth\ nclinit - - - Obj\ start\ m$] **have**

wf-jvm-prog $_{\Phi'}$ (start-prog P C M) **by** *simp*

then show *?thesis* **by**(*auto simp: wf-jvm-prog-def*)

qed

2.17.4 Methods and instructions

lemma *start-prog-Start-sees-methods:*

$P \vdash Object\ sees\ methods\ Mm$

$\implies start\ prog\ P\ C\ M \vdash$

*Start sees methods Mm ++ (map-option ($\lambda m. (m, Start)$) \circ map-of [*start-method C M, start-clinit*])*

by (*auto simp: class-def fun-upd-apply*

dest!: class-add-sees-methods-Obj[**where** $P=P$ **and** $C=Start$] *intro: sees-methods-rec*)

lemma *start-prog-Start-sees-start-method:*

$P \vdash Object\ sees\ methods\ Mm$

$\implies start\ prog\ P\ C\ M \vdash$

Start sees start-m, Static : $[] \rightarrow Void = (1, 0, [Invokestatic\ C\ M\ 0, Return], [])$ in Start

by(*auto simp: start-method-def Method-def fun-upd-apply*

dest!: start-prog-Start-sees-methods)

lemma *wf-start-prog-Start-sees-start-method:*

```

assumes wf: wf-prog wf-md P
shows start-prog P C M  $\vdash$ 
  Start sees start-m, Static : []  $\rightarrow$  Void = (1, 0, [Invokestatic C M 0, Return], []) in Start
proof –
  from wf have is-class P Object by simp
  with sees-methods-Object obtain Mm where P  $\vdash$  Object sees-methods Mm
  by(fastforce simp: is-class-def dest: sees-methods-Object)
  then show ?thesis by(rule start-prog-Start-sees-start-method)
qed

```

```

lemma start-prog-start-m-instrs:
assumes wf: wf-prog wf-md P
shows (instrs-of (start-prog P C M) Start start-m) = [Invokestatic C M 0, Return]
proof –
  from wf-start-prog-Start-sees-start-method[OF wf]
  have start-prog P C M  $\vdash$  Start sees start-m, Static :
    []  $\rightarrow$  Void = (1, 0, [Invokestatic C M 0, Return], []) in Start by simp
  then show ?thesis by simp
qed

```

```

declare wt-defs [simp del]

```

```

end

```

2.18 BV Type Safety Proof

```

theory BVSpecTypeSafe
imports BVConform StartProg
begin

```

This theory contains proof that the specification of the bytecode verifier only admits type safe programs.

2.18.1 Preliminaries

Simp and intro setup for the type safety proof:

```

lemmas defs1 = correct-state-def conf-f-def wt-instr-def eff-def norm-eff-def app-def xcpt-app-def

```

```

lemmas widen-rules [intro] = conf-widen confT-widen confs-widens confTs-widen

```

2.18.2 Exception Handling

For the *Invoke* instruction the BV has checked all handlers that guard the current *pc*.

```

lemma Invoke-handlers:
  match-ex-table P C pc xt = Some (pc', d')  $\implies$ 
   $\exists (f, t, D, h, d) \in \text{set } (\text{relevant-entries } P \text{ (Invoke } n \ M) \text{ } pc \ xt).$ 
  P  $\vdash$  C  $\preceq^*$  D  $\wedge$  pc  $\in$  {f.. $\langle$ t}  $\wedge$  pc' = h  $\wedge$  d' = d
  by (induct xt) (auto simp: relevant-entries-def matches-ex-entry-def
    is-relevant-entry-def split: if-split-asm)

```

For the *Invokestatic* instruction the BV has checked all handlers that guard the current *pc*.

lemma *Invokestatic-handlers*:

match-ex-table $P\ C\ pc\ xt = \text{Some}\ (pc', d') \implies$
 $\exists (f, t, D, h, d) \in \text{set}\ (\text{relevant-entries}\ P\ (\text{Invokestatic}\ C_0\ n\ M)\ pc\ xt).$
 $P \vdash C \preceq^* D \wedge pc \in \{f..<t\} \wedge pc' = h \wedge d' = d$
by (*induct xt*) (*auto simp: relevant-entries-def matches-ex-entry-def*
is-relevant-entry-def split: if-split-asm)

For the instrs in *Called-set* the BV has checked all handlers that guard the current *pc*.

lemma *Called-set-handlers*:

match-ex-table $P\ C\ pc\ xt = \text{Some}\ (pc', d') \implies i \in \text{Called-set} \implies$
 $\exists (f, t, D, h, d) \in \text{set}\ (\text{relevant-entries}\ P\ i\ pc\ xt).$
 $P \vdash C \preceq^* D \wedge pc \in \{f..<t\} \wedge pc' = h \wedge d' = d$
by (*induct xt*) (*auto simp: relevant-entries-def matches-ex-entry-def*
is-relevant-entry-def split: if-split-asm)

We can prove separately that the recursive search for exception handlers (*find-handler*) in the frame stack results in a conforming state (if there was no matching exception handler in the current frame). We require that the exception is a valid heap address, and that the state before the exception occurred conforms.

lemma *uncaught-xcpt-correct*:

assumes *wt*: $wf\text{-jvm-prog}_{\Phi}\ P$
assumes *h*: $h\ xcp = \text{Some}\ \text{obj}$
shows $\bigwedge f. P, \Phi \vdash (\text{None}, h, f\#\text{frs}, sh)\checkmark$
 $\implies \text{curr-method}\ f \neq \text{clinit} \implies P, \Phi \vdash \text{find-handler}\ P\ xcp\ h\ \text{frs}\ sh\ \checkmark$
(is $\bigwedge f. ?\text{correct}\ (\text{None}, h, f\#\text{frs}, sh) \implies ?\text{prem}\ f \implies ?\text{correct}\ (?\text{find}\ \text{frs}))$

The requirement of lemma *uncaught-xcpt-correct* (that the exception is a valid reference on the heap) is always met for welltyped instructions and conformant states:

lemma *exec-instr-xcpt-h*:

$\llbracket \text{fst}\ (\text{exec-instr}\ (\text{ins!pc})\ P\ h\ \text{stk}\ \text{vars}\ C\ M\ pc\ \text{ics}\ \text{frs}\ sh) = \text{Some}\ xcp;$
 $P, T, \text{mxs}, \text{size}\ \text{ins}, xt \vdash \text{ins!pc}, pc :: \Phi\ C\ M;$
 $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, pc, \text{ics})\#\text{frs}, sh)\checkmark \rrbracket$
 $\implies \exists \text{obj}. h\ xcp = \text{Some}\ \text{obj}$
(is $\llbracket ?\text{xcpt}; ?\text{wt}; ?\text{correct} \rrbracket \implies ?\text{thesis}$)

lemma *exec-step-xcpt-h*:

assumes *xcpt*: $\text{fst}\ (\text{exec-step}\ P\ h\ \text{stk}\ \text{vars}\ C\ M\ pc\ \text{ics}\ \text{frs}\ sh) = \text{Some}\ xcp$
and *ins*: $\text{instrs-of}\ P\ C\ M = \text{ins}$
and *wti*: $P, T, \text{mxs}, \text{size}\ \text{ins}, xt \vdash \text{ins!pc}, pc :: \Phi\ C\ M$
and *correct*: $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, pc, \text{ics})\#\text{frs}, sh)\checkmark$
shows $\exists \text{obj}. h\ xcp = \text{Some}\ \text{obj}$
proof –
from *correct* **have** *pre*: *preallocated h* **by** (*simp add: defs1 hconf-def*)
{ **fix** $C' Cs$ **assume** $[\text{simp}]: \text{ics} = \text{Calling}\ C'\ Cs$
with *xcpt* **have** $xcp = \text{addr-of-sys-xcpt}\ \text{NoClassDefFoundError}$
by (*cases ics, auto simp: split-beta split: init-state.splits if-split-asm*)
with *pre* **have** *?thesis* **using** *preallocated-def* **by** *force*
}
moreover
{ **fix** $Cs\ a$ **assume** $[\text{simp}]: \text{ics} = \text{Throwing}\ Cs\ a$
with *xcpt* **have** $eq: a = xcp$ **by** (*cases Cs; simp*)

```

from correct have  $P, h, sh \vdash_i (C, M, pc, ics)$  by (auto simp: defs1)
with eq have ?thesis by simp
}
moreover
{ fix Cs assume ics: ics = No-ics  $\vee$  ics = Called Cs
  with exec-instr-xcpt-h[OF - wti correct] xcpt ins have ?thesis by (cases Cs, auto)
}
ultimately show ?thesis by (cases ics, auto)
qed

```

lemma *conf-sys-xcpt:*

```

[[preallocated h; C  $\in$  sys-xcpts]]  $\implies P, h \vdash \text{Addr } (\text{addr-of-sys-xcpt } C) \leq \text{Class } C$ 
by (auto elim: preallocatedE)

```

lemma *match-ex-table-SomeD:*

```

match-ex-table P C pc xt = Some (pc', d')  $\implies$ 
 $\exists (f, t, D, h, d) \in \text{set } xt. \text{matches-ex-entry } P C pc (f, t, D, h, d) \wedge h = pc' \wedge d = d'$ 
by (induct xt) (auto split: if-split-asm)

```

Finally we can state that, whenever an exception occurs, the next state always conforms:

lemma *xcpt-correct:*

```

fixes  $\sigma' :: \text{jvm-state}$ 
assumes wtp: wf-jvm-prog $\Phi$  P
assumes meth: P  $\vdash$  C sees M, b: Ts  $\rightarrow$  T = (m $\bar{x}$ s, m $\bar{x}$ l $\bar{o}$ , ins, xt) in C
assumes wt: P, T, m $\bar{x}$ s, size ins, xt  $\vdash$  ins!pc, pc ::  $\Phi$  C M
assumes xp: fst (exec-step P h stk loc C M pc ics frs sh) = Some xcp
assumes s': Some  $\sigma' = \text{exec } (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, pc, ics) \# \text{frs}, sh)$ 
assumes correct: P,  $\Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, pc, ics) \# \text{frs}, sh) \checkmark$ 
shows  $P, \Phi \vdash \sigma' \checkmark$ 

```

declare *defs1* [*simp*]

2.18.3 Initialization procedure steps

In this section we prove that, for states that result in a step of the initialization procedure rather than an instruction execution, the state after execution of the step still conforms.

lemma *Calling-correct:*

```

fixes  $\sigma' :: \text{jvm-state}$ 
assumes wtprog: wf-jvm-prog $\Phi$  P
assumes mC: P  $\vdash$  C sees M, b: Ts  $\rightarrow$  T = (m $\bar{x}$ s, m $\bar{x}$ l $\bar{o}$ , ins, xt) in C
assumes s': Some  $\sigma' = \text{exec } (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, pc, ics) \# \text{frs}, sh)$ 
assumes cf: P,  $\Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, pc, ics) \# \text{frs}, sh) \checkmark$ 
assumes xc: fst (exec-step P h stk loc C M pc ics frs sh) = None
assumes ics: ics = Calling C' Cs

```

shows $P, \Phi \vdash \sigma' \checkmark$

proof –

```

from wtprog obtain wfmb where wf: wf-prog wfmb P
by (simp add: wf-jvm-prog-phi-def)

```

from *mC cf* **obtain** *ST LT* **where**

```

h-ok: P  $\vdash$  h  $\checkmark$  and
sh-ok: P, h  $\vdash_s$  sh  $\checkmark$  and
 $\Phi: \Phi C M ! pc = \text{Some } (ST, LT)$  and

```

$stk: P, h \vdash stk \[:\leq] ST$ **and** $loc: P, h \vdash loc \[:\leq_{\top}] LT$ **and**
 $pc: pc < size \ ins$ **and**
 $frame: conf-f \ P \ h \ sh \ (ST, LT) \ ins \ (stk, loc, C, M, pc, ics)$ **and**
 $fs: conf-fs \ P \ h \ sh \ \Phi \ C \ M \ (size \ Ts) \ T \ frs$ **and**
 $confc: conf-clinit \ P \ sh \ ((stk, loc, C, M, pc, ics) \# frs)$ **and**
 $vics: P, h, sh \vdash_i \ (C, M, pc, ics)$
by (*fastforce dest: sees-method-fun*)

with ics **have** $confc_0: conf-clinit \ P \ sh \ ((stk, loc, C, M, pc, Calling \ C' \ Cs) \# frs)$ **by** *simp*

from $vics \ ics$ **have** $cls': is-class \ P \ C'$ **by** *auto*

{ assume $None: sh \ C' = None$

let $?sh = sh(C' \mapsto (sblank \ P \ C', Prepared))$

obtain *FDTs* **where**
 $flds: P \vdash C' \ has-fields \ FDTs$ **using** *wf-Fields-Ex*[*OF wf cls'*] **by** *clarsimp*

from $shconf-upd-obj$ [**where** $C=C'$, *OF sh-ok soconf-sblank*[*OF flds*]]
have $sh-ok': P, h \vdash_s \ ?sh \ \checkmark$ **by** *simp*

from $None$ **have** $\forall sfs. sh \ C' \neq Some(sfs, Processing)$ **by** *simp*
with *conf-clinit-nProc-dist*[*OF confc*] **have**
 $dist': distinct \ (C' \# clinit-classes \ ((stk, loc, C, M, pc, ics) \# frs))$ **by** *simp*
then have $dist'': distinct \ (C' \# clinit-classes \ frs)$ **by** *simp*

have $confc': conf-clinit \ P \ ?sh \ ((stk, loc, C, M, pc, ics) \# frs)$
by(*rule conf-clinit-shupd*[*OF confc dist'*])
have $fs': conf-fs \ P \ h \ ?sh \ \Phi \ C \ M \ (size \ Ts) \ T \ frs$ **by**(*rule conf-fs-shupd*[*OF fs dist''*])
from $vics \ ics$ **have** $vics': P, h, ?sh \vdash_i \ (C, M, pc, ics)$ **by** *auto*

from $s' \ ics \ None$ **have** $\sigma' = (None, h, (stk, loc, C, M, pc, ics) \# frs, ?sh)$ **by** *auto*

with $mC \ h-ok \ sh-ok' \ \Phi \ stk \ loc \ pc \ fs' \ confc \ vics' \ confc' \ frame \ None$
have *?thesis* **by** *fastforce*

}
moreover
{ fix a **assume** $sh \ C' = Some \ a$
then obtain $sfs \ i$ **where** $shC'[simp]: sh \ C' = Some(sfs, i)$ **by**(*cases a, simp*)

from $confc \ ics$ **have** $last: \exists sobj. sh \ (last(C' \# Cs)) = Some \ sobj$
by(*fastforce simp: conf-clinit-def*)

let $?f = \lambda ics'. (stk, loc, C, M, pc, ics'::init-call-status)$

{ assume $i: i = Done \vee i = Processing$
let $?ics = Called \ Cs$

from $last \ vics \ ics$ **have** $vics': P, h, sh \vdash_i \ (C, M, pc, ?ics)$ **by** *auto*
from $confc \ ics$ **have** $confc': conf-clinit \ P \ sh \ (?f \ ?ics \# frs)$
by(*cases M=clinit; clarsimp simp: conf-clinit-def distinct-clinit-def*)

from $i \ s' \ ics$ **have** $\sigma' = (None, h, ?f \ ?ics \# frs, sh)$ **by** *auto*

```

  with mC h-ok sh-ok  $\Phi$  stk loc pc fs confc' vics' frame ics
  have ?thesis by fastforce
}
moreover
{ assume i[simp]: i = Error
  let ?a = addr-of-sys-xcpt NoClassDefFoundError
  let ?ics = Throwing Cs ?a

  from h-ok have preh: preallocated h by (simp add: hconf-def)
  then obtain obj where ha: h ?a = Some obj by (clarsimp simp: preallocated-def sys-xcpts-def)
  with vics ics have vics': P,h,sh  $\vdash_i$  (C, M, pc, ?ics) by auto

  from confc ics have confc'': conf-clinit P sh (?f ?ics#frs)
    by (cases M=clinit; clarsimp simp: conf-clinit-def distinct-clinit-def)

  from s' ics have  $\sigma'$ :  $\sigma' = (None, h, ?f ?ics#frs, sh)$  by auto

  from mC h-ok sh-ok  $\Phi$  stk loc pc fs confc'' vics  $\sigma'$  ics ha
  have ?thesis by fastforce
}
moreover
{ assume i[simp]: i = Prepared
  let ?sh = sh(C'  $\mapsto$  (sfs,Processing))
  let ?D = fst(the(class P C'))
  let ?ics = if C' = Object then Called (C'#Cs) else Calling ?D (C'#Cs)

  from shconf-upd-obj[where C=C', OF sh-ok shconfD[OF sh-ok shC']]
  have sh-ok': P,h  $\vdash_s$  ?sh  $\checkmark$  by simp

  from cls' have C'  $\neq$  Object  $\implies$  P  $\vdash$  C'  $\preceq^*$  ?D by (auto simp: is-class-def intro!: subclsII)
  with is-class-supclass[OF wf - cls'] have D: C'  $\neq$  Object  $\implies$  is-class P ?D by simp

  from i have  $\forall$  sfs. sh C'  $\neq$  Some(sfs,Processing) by simp
  with conf-clinit-nProc-dist[OF confc0] have
    dist': distinct (C' # clinit-classes ((stk, loc, C, M, pc, Calling C' Cs) # frs)) by fast
  then have dist'': distinct (C' # clinit-classes frs) by simp

  from conf-clinit-shupd-Calling[OF confc0 dist' cls']
    conf-clinit-shupd-Called[OF confc0 dist' cls']
  have confc': conf-clinit P ?sh (?f ?ics#frs) by clarsimp
  with last ics have  $\exists$  sobj. ?sh (last(C'#Cs)) = Some sobj
    by (auto simp: conf-clinit-def fun-upd-apply)
  with D vics ics have vics': P,h,?sh  $\vdash_i$  (C, M, pc, ?ics) by auto

  have fs': conf-fs P h ?sh  $\Phi$  C M (size Ts) T frs by (rule conf-fs-shupd[OF fs dist'])

  from frame vics' have frame': conf-f P h ?sh (ST, LT) ins (?f ?ics) by simp

  from i s' ics have  $\sigma' = (None, h, ?f ?ics#frs, ?sh)$  by (auto simp: if-split-asm)

  with mC h-ok sh-ok'  $\Phi$  stk loc pc fs' confc' frame' ics
  have ?thesis by fastforce
}

```

ultimately have *?thesis* **by**(cases *i*, *auto*)
 }
 ultimately show *?thesis* **by**(cases *sh C'*, *auto*)
qed

lemma *Throwing-correct*:

fixes $\sigma' :: \text{jvm-state}$

assumes *wfprog*: $\text{wf-jvm-prog}_{\Phi} P$

assumes *mC*: $P \vdash C \text{ sees } M, b: Ts \rightarrow T = (\text{mcs}, \text{mcl}_0, \text{ins}, \text{xt}) \text{ in } C$

assumes *s'*: $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics}) \# \text{frs}, \text{sh})$

assumes *cf*: $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics}) \# \text{frs}, \text{sh}) \checkmark$

assumes *xc*: $\text{fst } (\text{exec-step } P h \text{ stk loc } C M \text{ pc ics frs sh}) = \text{None}$

assumes *ics*: $\text{ics} = \text{Throwing } (C' \# Cs) a$

shows $P, \Phi \vdash \sigma' \checkmark$

proof –

from *wfprog* **obtain** *wfmb* **where** *wf*: $\text{wf-prog } \text{wfmb } P$

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

from *mC cf* **obtain** *ST LT* **where**

h-ok: $P \vdash h \checkmark$ **and**

sh-ok: $P, h \vdash_s \text{sh} \checkmark$ **and**

Φ : $\Phi C M ! \text{pc} = \text{Some } (ST, LT)$ **and**

stk: $P, h \vdash \text{stk} [:\leq] ST$ **and** *loc*: $P, h \vdash \text{loc} [:\leq_{\top}] LT$ **and**

pc: $\text{pc} < \text{size ins}$ **and**

frame: $\text{conf-f } P h \text{ sh } (ST, LT) \text{ ins } (\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics})$ **and**

fs: $\text{conf-fs } P h \text{ sh } \Phi C M (\text{size } Ts) T \text{ frs}$ **and**

confc: $\text{conf-clinit } P \text{ sh } ((\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics}) \# \text{frs})$ **and**

vics: $P, h, \text{sh} \vdash_i (C, M, \text{pc}, \text{ics})$

by (*fastforce dest*: *sees-method-fun*)

with *ics* **have** *confc₀*: $\text{conf-clinit } P \text{ sh } ((\text{stk}, \text{loc}, C, M, \text{pc}, \text{Throwing } (C' \# Cs) a) \# \text{frs})$ **by** *simp*

from *frame ics mC* **have**

cc: $\exists C1. \text{Called-context } P C1 (\text{ins} ! \text{pc})$ **by**(*clarsimp simp*: *conf-f-def2*)

from *frame ics* **obtain** *obj* **where** *ha*: $h a = \text{Some } \text{obj}$ **by**(*auto simp*: *conf-f-def2*)

from *confc ics* **obtain** *sfs i* **where** *shC'*: $\text{sh } C' = \text{Some}(sfs, i)$ **by**(*clarsimp simp*: *conf-clinit-def*)

then have *sfs*: $P, h, C' \vdash_s \text{sfs} \checkmark$ **by**(*rule shconfD[OF sh-ok]*)

from *s' ics*

have σ' : $\sigma' = (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}, \text{Throwing } Cs a) \# \text{frs}, \text{sh}(C' \mapsto (\text{fst}(\text{the}(\text{sh } C')), \text{Error})))$

(**is** $\sigma' = (\text{None}, h, ?f' \# \text{frs}, ?sh')$)

by *simp*

from *confc ics* **have** *dist*: $\text{distinct } (C' \# \text{clinit-classes } (?f' \# \text{frs}))$

by (*simp add*: *conf-clinit-def distinct-clinit-def*)

then have *dist'*: $\text{distinct } (C' \# \text{clinit-classes } \text{frs})$ **by** *simp*

from *conf-clinit-Throwing confc ics* **have** *confc'*: $\text{conf-clinit } P \text{ sh } (?f' \# \text{frs})$ **by** *simp*

from *shconf-upd-obj[OF sh-ok sfs] shC'* **have** $P, h \vdash_s ?sh' \checkmark$ **by** *simp*
moreover

have *conf-fs* $P\ h\ ?sh'\ \Phi\ C\ M\ (\text{length } Ts)\ T\ \text{frs}$ **by** (*rule conf-fs-shupd*[*OF fs dist'*])
moreover
have *conf-clinit* $P\ ?sh'\ (?f'\ \#\ \text{frs})$ **by** (*rule conf-clinit-shupd*[*OF confc' dist*])
moreover note $\sigma'\ h\text{-ok}\ mC\ \Phi\ pc\ stk\ loc\ ha\ cc$
ultimately show $P, \Phi \vdash \sigma' \checkmark$ **by** *fastforce*
qed

lemma *Called-correct*:

fixes $\sigma' :: \text{jvm-state}$
assumes *wtprog*: *wf-jvm-prog* $\Phi\ P$
assumes *mC*: $P \vdash C\ \text{sees } M, b: Ts \rightarrow T = (m\text{xs}, m\text{x}l_0, \text{ins}, \text{xt})$ *in* C
assumes *s'*: *Some* $\sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc, ics) \#\ \text{frs}, sh)$
assumes *cf*: $P, \Phi \vdash (\text{None}, h, (stk, loc, C, M, pc, ics) \#\ \text{frs}, sh) \checkmark$
assumes *xc*: *fst* (*exec-step* $P\ h\ stk\ loc\ C\ M\ pc\ ics\ \text{frs}\ sh$) = *None*
assumes *ics[simp]*: $ics = \text{Called } (C' \#\ Cs)$

shows $P, \Phi \vdash \sigma' \checkmark$

proof –

from *wtprog* **obtain** *wfmb* **where** *wf*: *wf-prog* *wfmb* P
by (*simp add*: *wf-jvm-prog-phi-def*)

from *mC* *cf* **obtain** *ST* *LT* **where**

h-ok: $P \vdash h \checkmark$ **and**
sh-ok: $P, h \vdash_s sh \checkmark$ **and**
 Φ : $\Phi\ C\ M\ !\ pc = \text{Some } (ST, LT)$ **and**
stk: $P, h \vdash stk\ [:\leq] ST$ **and** *loc*: $P, h \vdash loc\ [:\leq\top] LT$ **and**
pc: $pc < \text{size ins}$ **and**
frame: *conf-f* $P\ h\ sh\ (ST, LT)\ \text{ins } (stk, loc, C, M, pc, ics)$ **and**
fs: *conf-fs* $P\ h\ sh\ \Phi\ C\ M\ (\text{size } Ts)\ T\ \text{frs}$ **and**
confc: *conf-clinit* $P\ sh\ ((stk, loc, C, M, pc, ics) \#\ \text{frs})$ **and**
vics: $P, h, sh \vdash_i (C, M, pc, ics)$
by (*fastforce dest*: *sees-method-fun*)

then have *confc₀*: *conf-clinit* $P\ sh\ ((stk, loc, C, M, pc, \text{Called } (C' \#\ Cs)) \#\ \text{frs})$ **by** *simp*

from *frame* *mC* **obtain** *C1* *sobj* **where**

ss: *Called-context* $P\ C1\ (\text{ins}\ !\ pc)$ **and**
shC1: $sh\ C1 = \text{Some } sobj$ **by** (*clarsimp simp*: *conf-f-def2*)

from *confc* *wf-sees-clinit*[*OF wf*] **obtain** *mxs'* *mxl'* *ins'* *xt'* **where**

clinit: $P \vdash C'\ \text{sees } clinit, \text{Static}: [] \rightarrow \text{Void} = (mxs', mxl', ins', xt')$ *in* C'
by (*fastforce simp*: *conf-clinit-def is-class-def*)

let $?loc' = \text{replicate } mxl'\ \text{undefined}$

from *s' clinit*

have $\sigma': \sigma' = (\text{None}, h, ([], ?loc', C', clinit, 0, \text{No-ics}) \#\ (stk, loc, C, M, pc, \text{Called } Cs) \#\ \text{frs}, sh)$
(is $\sigma' = (\text{None}, h, ?if\ \#\ ?f'\ \#\ \text{frs}, sh)$
by *simp*

with *wtprog clinit*

obtain *start*: *wt-start* $P\ C'\ \text{Static } []\ mxl'\ (\Phi\ C'\ clinit)$ **and** *ins'*: $ins' \neq []$

by (*auto dest*: *wt-jvm-prog-impl-wt-start*)

then obtain *LT₀* **where** $LT_0: \Phi\ C'\ clinit\ !\ 0 = \text{Some } ([], LT_0)$

by (*clarsimp simp: wt-start-def defs1 sup-state-opt-any-Some split: staticb.splits*)
moreover
have *conf-f P h sh* (\square , LT_0) *ins' ?if*
proof –
 let $?LT = \text{replicate } m\lambda' \text{ Err}$
have $P, h \vdash ?loc' [:\leq_{\top}] ?LT$ **by** *simp*
also from *start LT_0 have $P \vdash \dots [:\leq_{\top}] LT_0$ by (simp add: wt-start-def)*
finally have $P, h \vdash ?loc' [:\leq_{\top}] LT_0$.
thus *?thesis using ins' by simp*
qed
moreover
from *conf-clinit-Called confc clinit have conf-clinit P sh (?if # ?f' # frs) by simp*
moreover note $\sigma' \text{ h-ok sh-ok } mC \Phi \text{ pc stk loc clinit ss shC1 fs}$
ultimately show $P, \Phi \vdash \sigma' \checkmark$ **by** *fastforce*
qed

2.18.4 Single Instructions

In this section we prove for each single (welltyped) instruction that the state after execution of the instruction still conforms. Since we have already handled exceptions above, we can now assume that no exception occurs in this step. For instructions that may call the initialization procedure, we cover the calling and non-calling cases separately.

lemma *Invoke-correct:*

fixes $\sigma' :: \text{jvm-state}$
assumes *wtprog: wf-jvm-prog $_{\Phi}$ P*
assumes *meth-C: P \vdash C sees M, b: Ts \rightarrow T = (m λ s, m λ l $_0$, ins, xt) in C*
assumes *ins: ins ! pc = Invoke M' n*
assumes *wti: P, T, m λ s, size ins, xt \vdash ins!pc, pc :: Φ C M*
assumes σ' : *Some $\sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics}) \# \text{frs}, \text{sh})$*
assumes *approx: P, $\Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics}) \# \text{frs}, \text{sh}) \checkmark$*
assumes *no-xcp: fst (exec-step P h stk loc C M pc ics frs sh) = None*
shows $P, \Phi \vdash \sigma' \checkmark$

lemma *Invokestatic-nInit-correct:*

fixes $\sigma' :: \text{jvm-state}$
assumes *wtprog: wf-jvm-prog $_{\Phi}$ P*
assumes *meth-C: P \vdash C sees M, b: Ts \rightarrow T = (m λ s, m λ l $_0$, ins, xt) in C*
assumes *ins: ins ! pc = Invokestatic D M' n and nclinit: M' \neq clinit*
assumes *wti: P, T, m λ s, size ins, xt \vdash ins!pc, pc :: Φ C M*
assumes σ' : *Some $\sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics}) \# \text{frs}, \text{sh})$*
assumes *approx: P, $\Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics}) \# \text{frs}, \text{sh}) \checkmark$*
assumes *no-xcp: fst (exec-step P h stk loc C M pc ics frs sh) = None*
assumes *cs: ics = Called $\square \vee (\text{ics} = \text{No-ics} \wedge (\exists \text{sfs}. \text{sh} (\text{fst}(\text{method } P \text{ D } M')) = \text{Some}(\text{sfs}, \text{Done})))$*
shows $P, \Phi \vdash \sigma' \checkmark$

lemma *Invokestatic-Init-correct:*

fixes $\sigma' :: \text{jvm-state}$
assumes *wtprog: wf-jvm-prog $_{\Phi}$ P*
assumes *meth-C: P \vdash C sees M, b: Ts \rightarrow T = (m λ s, m λ l $_0$, ins, xt) in C*
assumes *ins: ins ! pc = Invokestatic D M' n and nclinit: M' \neq clinit*
assumes *wti: P, T, m λ s, size ins, xt \vdash ins!pc, pc :: Φ C M*
assumes σ' : *Some $\sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}, \text{No-ics}) \# \text{frs}, \text{sh})$*
assumes *approx: P, $\Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}, \text{No-ics}) \# \text{frs}, \text{sh}) \checkmark$*
assumes *no-xcp: fst (exec-step P h stk loc C M pc No-ics frs sh) = None*
assumes *nDone: $\forall \text{sfs}. \text{sh} (\text{fst}(\text{method } P \text{ D } M')) \neq \text{Some}(\text{sfs}, \text{Done})$*

shows $P, \Phi \vdash \sigma' \surd$
declare *list-all2-Cons2* [iff]

lemma *Return-correct*:

fixes $\sigma' :: \text{jvm-state}$
assumes *wt-prog*: $\text{wf-jvm-prog}_{\Phi} P$
assumes *meth*: $P \vdash C \text{ sees } M, b: Ts \rightarrow T = (m\text{x}s, m\text{x}l_0, \text{ins}, \text{xt}) \text{ in } C$
assumes *ins*: $\text{ins} ! \text{pc} = \text{Return}$
assumes *wt*: $P, T, m\text{x}s, \text{size } \text{ins}, \text{xt} \vdash \text{ins} ! \text{pc}, \text{pc} :: \Phi C M$
assumes *s'*: $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics}) \# \text{frs}, \text{sh})$
assumes *correct*: $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics}) \# \text{frs}, \text{sh}) \surd$

shows $P, \Phi \vdash \sigma' \surd$
declare *sup-state-opt-any-Some* [iff]
declare *not-Err-eq* [iff]

lemma *Load-correct*:

assumes *wf-prog wt P and*
 $mC: P \vdash C \text{ sees } M, b: Ts \rightarrow T = (m\text{x}s, m\text{x}l_0, \text{ins}, \text{xt}) \text{ in } C$ **and**
 $i: \text{ins} ! \text{pc} = \text{Load } \text{idx}$ **and**
 $P, T, m\text{x}s, \text{size } \text{ins}, \text{xt} \vdash \text{ins} ! \text{pc}, \text{pc} :: \Phi C M$ **and**
 $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics}) \# \text{frs}, \text{sh})$ **and**
 $cf: P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics}) \# \text{frs}, \text{sh}) \surd$
shows $P, \Phi \vdash \sigma' [ok]$
declare [[*simproc del: list-to-set-comprehension*]]

lemma *Store-correct*:

assumes *wf-prog wt P and*
 $mC: P \vdash C \text{ sees } M, b: Ts \rightarrow T = (m\text{x}s, m\text{x}l_0, \text{ins}, \text{xt}) \text{ in } C$ **and**
 $i: \text{ins} ! \text{pc} = \text{Store } \text{idx}$ **and**
 $P, T, m\text{x}s, \text{size } \text{ins}, \text{xt} \vdash \text{ins} ! \text{pc}, \text{pc} :: \Phi C M$ **and**
 $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics}) \# \text{frs}, \text{sh})$ **and**
 $cf: P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics}) \# \text{frs}, \text{sh}) \surd$
shows $P, \Phi \vdash \sigma' [ok]$

lemma *Push-correct*:

assumes *wf-prog wt P and*
 $mC: P \vdash C \text{ sees } M, b: Ts \rightarrow T = (m\text{x}s, m\text{x}l_0, \text{ins}, \text{xt}) \text{ in } C$ **and**
 $i: \text{ins} ! \text{pc} = \text{Push } v$ **and**
 $P, T, m\text{x}s, \text{size } \text{ins}, \text{xt} \vdash \text{ins} ! \text{pc}, \text{pc} :: \Phi C M$ **and**
 $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics}) \# \text{frs}, \text{sh})$ **and**
 $cf: P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}, \text{ics}) \# \text{frs}, \text{sh}) \surd$
shows $P, \Phi \vdash \sigma' [ok]$

2.19 Welltyped Programs produce no Type Errors

theory *BVNoTypeError*

imports *../JVM/JVMDefensive BVSpecTypeSafe*
begin

lemma *has-methodI*:

$P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D \implies P \vdash C \text{ has } M, b$
by (*unfold has-method-def*) *blast*

Some simple lemmas about the type testing functions of the defensive JVM:

lemma *typeof-NoneD* [*simp, dest*]: $\text{typeof } v = \text{Some } x \implies \neg \text{is-Addr } v$
by (*cases v*) *auto*

lemma *is-Ref-def2*:
 $\text{is-Ref } v = (v = \text{Null} \vee (\exists a. v = \text{Addr } a))$
by (*cases v*) (*auto simp add: is-Ref-def*)

lemma [*iff*]: *is-Ref Null* **by** (*simp add: is-Ref-def2*)

lemma *is-RefI* [*intro, simp*]: $P, h \vdash v : \leq T \implies \text{is-refT } T \implies \text{is-Ref } v$

lemma *is-IntgI* [*intro, simp*]: $P, h \vdash v : \leq \text{Integer} \implies \text{is-Intg } v$

lemma *is-BoolI* [*intro, simp*]: $P, h \vdash v : \leq \text{Boolean} \implies \text{is-Bool } v$

declare *defs1* [*simp del*]

lemma *wt-jvm-prog-states-NonStatic*:

assumes *wf*: *wf-jvm-prog Φ P*

and *mC*: $P \vdash C \text{ sees } M, \text{NonStatic}: Ts \rightarrow T = (m\text{xs}, m\text{x}l, \text{ins}, \text{et}) \text{ in } C$

and $\Phi: \Phi C M ! pc = \tau$ **and** *pc*: $pc < \text{size ins}$

shows *OK* $\tau \in \text{states } P \text{ m}\text{xs} (1 + \text{size } Ts + m\text{x}l)$

lemma *wt-jvm-prog-states-Static*:

assumes *wf*: *wf-jvm-prog Φ P*

and *mC*: $P \vdash C \text{ sees } M, \text{Static}: Ts \rightarrow T = (m\text{xs}, m\text{x}l, \text{ins}, \text{et}) \text{ in } C$

and $\Phi: \Phi C M ! pc = \tau$ **and** *pc*: $pc < \text{size ins}$

shows *OK* $\tau \in \text{states } P \text{ m}\text{xs} (\text{size } Ts + m\text{x}l)$

The main theorem: welltyped programs do not produce type errors if they are started in a conformant state.

theorem *no-type-error*:

fixes $\sigma :: \text{jvm-state}$

assumes *welldtyped*: *wf-jvm-prog Φ P* **and** *conforms*: $P, \Phi \vdash \sigma \checkmark$

shows *exec-d* $P \sigma \neq \text{TypeError}$

The theorem above tells us that, in welltyped programs, the defensive machine reaches the same result as the aggressive one (after arbitrarily many steps).

theorem *welldtyped-aggressive-imp-defensive*:

$wf\text{-jvm-prog}\Phi P \implies P, \Phi \vdash \sigma \checkmark \implies P \vdash \sigma \text{-jvm} \rightarrow \sigma'$

$\implies P \vdash (\text{Normal } \sigma) \text{-jvmd} \rightarrow (\text{Normal } \sigma')$

As corollary we get that the aggressive and the defensive machine are equivalent for welltyped programs (if started in a conformant state or in the canonical start state)

corollary *welldtyped-commutes*:

fixes $\sigma :: \text{jvm-state}$

assumes *wf*: *wf-jvm-prog Φ P* **and** *conforms*: $P, \Phi \vdash \sigma \checkmark$

shows $P \vdash (\text{Normal } \sigma) \text{-jvmd} \rightarrow (\text{Normal } \sigma') = P \vdash \sigma \text{-jvm} \rightarrow \sigma'$

proof(*rule iffI*)

assume $P \vdash \text{Normal } \sigma \text{-jvmd} \rightarrow \text{Normal } \sigma'$ **then show** $P \vdash \sigma \text{-jvm} \rightarrow \sigma'$

by (*rule defensive-imp-aggressive*)

next

assume $P \vdash \sigma \text{-jvm} \rightarrow \sigma'$ **then show** $P \vdash \text{Normal } \sigma \text{-jvmd} \rightarrow \text{Normal } \sigma'$

by (*rule welldtyped-aggressive-imp-defensive [OF wf conforms]*)

qed

corollary *welltyped-initial-commutes*:

assumes *wf*: *wf-jvm-prog* *P*
assumes *nstart*: \neg *is-class* *P* *Start*
assumes *meth*: $P \vdash C$ *sees* $M, \text{Static} : [] \rightarrow \text{Void} = b$ *in* *C*
assumes *nclinit*: $M \neq \text{clinit}$
assumes *Obj-start-m*:
 $(\bigwedge b' Ts' T' m' D'. P \vdash \text{Object sees start-m}, b' : Ts' \rightarrow T' = m' \text{ in } D'$
 $\implies b' = \text{Static} \wedge Ts' = [] \wedge T' = \text{Void})$
defines *start*: $\sigma \equiv \text{start-state } P$
shows *start-prog* $P C M \vdash (\text{Normal } \sigma) -\text{jvmd} \rightarrow (\text{Normal } \sigma') = \text{start-prog } P C M \vdash \sigma -\text{jvm} \rightarrow \sigma'$
proof –
from *wf* **obtain** Φ **where** *wf'*: *wf-jvm-prog* $_{\Phi}$ *P* **by** (*auto simp*: *wf-jvm-prog-def*)
let $?\Phi = \Phi\text{-start } \Phi$
from *start-prog-wf-jvm-prog-phi* **where** $\Phi' = ?\Phi$, *OF* *wf'* *nstart* *meth* *nclinit* $\Phi\text{-start}$ *Obj-start-m*
have *wf-jvm-prog* $?_{\Phi}(\text{start-prog } P C M)$ **by** *simp*
moreover
from *wf'* *nstart* *meth* *nclinit* $\Phi\text{-start}(2)$ **have** *start-prog* $P C M, ?\Phi \vdash \sigma \checkmark$
unfolding *start* **by** (*rule BV-correct-initial*)
ultimately show *?thesis* **by** (*rule welltyped-commutes*)
qed

lemma *not-TypeError-eq* [*iff*]:

$x \neq \text{TypeError} = (\exists t. x = \text{Normal } t)$
by (*cases* *x*) *auto*

locale *cnf* =

fixes *P* **and** Φ **and** σ
assumes *wf*: *wf-jvm-prog* $_{\Phi}$ *P*
assumes *cnf*: *correct-state* $P \Phi \sigma$

theorem (**in** *cnf*) *no-type-errors*:

$P \vdash (\text{Normal } \sigma) -\text{jvmd} \rightarrow \sigma' \implies \sigma' \neq \text{TypeError}$

proof –

assume $P \vdash (\text{Normal } \sigma) -\text{jvmd} \rightarrow \sigma'$
then have $(\text{Normal } \sigma, \sigma') \in (\text{exec-1-d } P)^*$ **by** (*unfold exec-all-d-def1*) *simp*
then show *?thesis* **proof**(*induct rule*: *rtrancl-induct*)
case (*step* *y z*)
then obtain y_n **where** [*simp*]: $y = \text{Normal } y_n$ **by** *clarsimp*
have $n\sigma y: P \vdash \text{Normal } \sigma -\text{jvmd} \rightarrow \text{Normal } y_n$ **using** *step.hyps(1)*
by (*fold exec-all-d-def1*) *simp*
have $\sigma y: P \vdash \sigma -\text{jvm} \rightarrow y_n$ **using** *defensive-imp-aggressive*[*OF* $n\sigma y$] **by** *simp*
show *?case* **using** *step no-type-error*[*OF* *wf BV-correct*[*OF* *wf* σy *cnf*]]
by (*auto simp* *add*: *exec-1-d-eq*)
qed *simp*
qed

locale *start* =

fixes *P* **and** *C* **and** *M* **and** σ **and** *T* **and** *b* **and** P_0
assumes *wf*: *wf-jvm-prog* *P*
assumes *nstart*: \neg *is-class* *P* *Start*
assumes *sees*: $P \vdash C$ *sees* $M, \text{Static} : [] \rightarrow \text{Void} = b$ *in* *C*
assumes *nclinit*: $M \neq \text{clinit}$
assumes *Obj-start-m*: $(\bigwedge b' Ts' T' m' D'. P \vdash \text{Object sees start-m}, b' : Ts' \rightarrow T' = m' \text{ in } D'$

$\implies b' = \text{Static} \wedge Ts' = [] \wedge T' = \text{Void}$
defines $\sigma \equiv \text{Normal}$ (*start-state P*)
defines [*simp*]: $P_0 \equiv \text{start-prog } P \ C \ M$

corollary (*in start*) *bv-no-type-error*:

shows $P_0 \vdash \sigma \text{ -jvmd} \rightarrow \sigma' \implies \sigma' \neq \text{TypeError}$

proof –

from *wf* **obtain** Φ **where** wf' : *wf-jvm-prog* Φ *P* **by** (*auto simp: wf-jvm-prog-def*)

let $?\Phi = \Phi\text{-start } \Phi$

from *start-prog-wf-jvm-prog-phi*[**where** $\Phi' = ?\Phi$, *OF wf' nstart sees nclinit* $\Phi\text{-start}$ *Obj-start-m*]

have *wf-jvm-prog* $?\Phi P_0$ **by** *simp*

moreover

from *BV-correct-initial*[**where** $\Phi' = ?\Phi$, *OF wf' nstart sees nclinit* $\Phi\text{-start}$ (2)]

have *correct-state* P_0 $?\Phi$ (*start-state P*) **by** *simp*

ultimately have *cnf* P_0 $?\Phi$ (*start-state P*) **by** (*rule cnf.intro*)

moreover assume $P_0 \vdash \sigma \text{ -jvmd} \rightarrow \sigma'$

ultimately show *?thesis* **by** (*unfold* $\sigma\text{-def}$) (*rule cnf.no-type-errors*)

qed

end

Chapter 3

Compilation

3.1 An Intermediate Language

theory *J1* imports *../J/BigStep* begin

type-synonym *expr*₁ = *nat exp*
type-synonym *J*₁-*prog* = *expr*₁ *prog*
type-synonym *state*₁ = *heap* × (*val list*) × *sheap*

definition *hp*₁ :: *state*₁ ⇒ *heap*

where

*hp*₁ ≡ *fst*

definition *lcl*₁ :: *state*₁ ⇒ *val list*

where

*lcl*₁ ≡ *fst* ∘ *snd*

definition *shp*₁ :: *state*₁ ⇒ *sheap*

where

*shp*₁ ≡ *snd* ∘ *snd*

primrec

max-vars :: 'a *exp* ⇒ *nat*

and *max-varss* :: 'a *exp list* ⇒ *nat*

where

max-vars(*new C*) = 0
| *max-vars*(*Cast C e*) = *max-vars e*
| *max-vars*(*Val v*) = 0
| *max-vars*(*e*₁ «*bop*» *e*₂) = *max* (*max-vars e*₁) (*max-vars e*₂)
| *max-vars*(*Var V*) = 0
| *max-vars*(*V:=e*) = *max-vars e*
| *max-vars*(*e*·*F*{*D*}) = *max-vars e*
| *max-vars*(*C*·*s**F*{*D*}) = 0
| *max-vars*(*FAss e*₁ *F D e*₂) = *max* (*max-vars e*₁) (*max-vars e*₂)
| *max-vars*(*SFAss C F D e*₂) = *max-vars e*₂
| *max-vars*(*e*·*M*(*es*)) = *max* (*max-vars e*) (*max-varss es*)
| *max-vars*(*C*·*s**M*(*es*)) = *max-varss es*
| *max-vars*({*V:T*; *e*}) = *max-vars e* + 1
| *max-vars*(*e*₁;;*e*₂) = *max* (*max-vars e*₁) (*max-vars e*₂)
| *max-vars*(*if* (*e*) *e*₁ *else e*₂) =
 max (*max-vars e*) (*max* (*max-vars e*₁) (*max-vars e*₂))
| *max-vars*(*while* (*b*) *e*) = *max* (*max-vars b*) (*max-vars e*)

| $max\text{-vars}(throw\ e) = max\text{-vars}\ e$
| $max\text{-vars}(try\ e_1\ catch(C\ V)\ e_2) = max\ (max\text{-vars}\ e_1)\ (max\text{-vars}\ e_2 + 1)$
| $max\text{-vars}(INIT\ C\ (Cs,b)\ \leftarrow\ e) = max\text{-vars}\ e$
| $max\text{-vars}(RI(C,e);Cs\ \leftarrow\ e') = max\ (max\text{-vars}\ e)\ (max\text{-vars}\ e')$

| $max\text{-varss}\ [] = 0$
| $max\text{-varss}\ (e\#\!es) = max\ (max\text{-vars}\ e)\ (max\text{-varss}\ es)$

inductive

$eval_1 :: J_1\text{-prog} \Rightarrow expr_1 \Rightarrow state_1 \Rightarrow expr_1 \Rightarrow state_1 \Rightarrow bool$
 $(\leftarrow \vdash_1 ((I\langle -,/- \rangle) \Rightarrow / (I\langle -,/- \rangle))) \triangleright [51,0,0,0,0] 81$

and $evals_1 :: J_1\text{-prog} \Rightarrow expr_1\ list \Rightarrow state_1 \Rightarrow expr_1\ list \Rightarrow state_1 \Rightarrow bool$
 $(\leftarrow \vdash_1 ((I\langle -,/- \rangle) [\Rightarrow] / (I\langle -,/- \rangle))) \triangleright [51,0,0,0,0] 81$

for $P :: J_1\text{-prog}$

where

$New_1:$

$\llbracket sh\ C = Some\ (sfs,\ Done); new\text{-Addr}\ h = Some\ a;$
 $P \vdash C\ has\text{-fields}\ FDTs; h' = h(a \mapsto blank\ P\ C) \rrbracket$
 $\implies P \vdash_1 \langle new\ C, (h,l,sh) \rangle \Rightarrow \langle addr\ a, (h',l,sh) \rangle$

| $NewFail_1:$

$\llbracket sh\ C = Some\ (sfs,\ Done); new\text{-Addr}\ h = None \rrbracket \implies$
 $P \vdash_1 \langle new\ C, (h,l,sh) \rangle \Rightarrow \langle THROW\ OutOfMemory, (h,l,sh) \rangle$

| $NewInit_1:$

$\llbracket \#sfs.\ sh\ C = Some\ (sfs,\ Done); P \vdash_1 \langle INIT\ C\ ([C],False) \leftarrow unit, (h,l,sh) \rangle \Rightarrow \langle Val\ v', (h',l',sh') \rangle;$
 $new\text{-Addr}\ h' = Some\ a; P \vdash C\ has\text{-fields}\ FDTs; h'' = h'(a \mapsto blank\ P\ C) \rrbracket$
 $\implies P \vdash_1 \langle new\ C, (h,l,sh) \rangle \Rightarrow \langle addr\ a, (h'',l',sh') \rangle$

| $NewInitOOM_1:$

$\llbracket \#sfs.\ sh\ C = Some\ (sfs,\ Done); P \vdash_1 \langle INIT\ C\ ([C],False) \leftarrow unit, (h,l,sh) \rangle \Rightarrow \langle Val\ v', (h',l',sh') \rangle;$
 $new\text{-Addr}\ h' = None; is\text{-class}\ P\ C \rrbracket$
 $\implies P \vdash_1 \langle new\ C, (h,l,sh) \rangle \Rightarrow \langle THROW\ OutOfMemory, (h',l',sh') \rangle$

| $NewInitThrow_1:$

$\llbracket \#sfs.\ sh\ C = Some\ (sfs,\ Done); P \vdash_1 \langle INIT\ C\ ([C],False) \leftarrow unit, (h,l,sh) \rangle \Rightarrow \langle throw\ a, s' \rangle;$
 $is\text{-class}\ P\ C \rrbracket$
 $\implies P \vdash_1 \langle new\ C, (h,l,sh) \rangle \Rightarrow \langle throw\ a, s' \rangle$

| $Cast_1:$

$\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle addr\ a, (h,l,sh) \rangle; h\ a = Some(D,fs); P \vdash D \preceq^* C \rrbracket$
 $\implies P \vdash_1 \langle Cast\ C\ e, s_0 \rangle \Rightarrow \langle addr\ a, (h,l,sh) \rangle$

| $CastNull_1:$

$P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle null, s_1 \rangle \implies$
 $P \vdash_1 \langle Cast\ C\ e, s_0 \rangle \Rightarrow \langle null, s_1 \rangle$

| $CastFail_1:$

$\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle addr\ a, (h,l,sh) \rangle; h\ a = Some(D,fs); \neg P \vdash D \preceq^* C \rrbracket$
 $\implies P \vdash_1 \langle Cast\ C\ e, s_0 \rangle \Rightarrow \langle THROW\ ClassCast, (h,l,sh) \rangle$

| $CastThrow_1:$

$P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle throw\ e', s_1 \rangle \implies$
 $P \vdash_1 \langle Cast\ C\ e, s_0 \rangle \Rightarrow \langle throw\ e', s_1 \rangle$

| $Val_1:$

$P \vdash_1 \langle Val\ v, s \rangle \Rightarrow \langle Val\ v, s \rangle$

| $BinOp_1:$

$\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle Val\ v_1, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle Val\ v_2, s_2 \rangle; binop(bop, v_1, v_2) = Some\ v \rrbracket$

$$\begin{aligned} & \Longrightarrow P \vdash_1 \langle e_1 \text{ «bop» } e_2, s_0 \rangle \Rightarrow \langle \text{Val } v, s_2 \rangle \\ | \text{BinOpThrow}_{11}: & \\ & P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle \Longrightarrow \\ & P \vdash_1 \langle e_1 \text{ «bop» } e_2, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle \\ | \text{BinOpThrow}_{21}: & \\ & \llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{throw } e, s_2 \rangle \rrbracket \\ & \Longrightarrow P \vdash_1 \langle e_1 \text{ «bop» } e_2, s_0 \rangle \Rightarrow \langle \text{throw } e, s_2 \rangle \\ | \text{Var}_1: & \\ & \llbracket \text{ls!}i = v; i < \text{size } \text{ls} \rrbracket \Longrightarrow \\ & P \vdash_1 \langle \text{Var } i, (h, \text{ls}, \text{sh}) \rangle \Rightarrow \langle \text{Val } v, (h, \text{ls}, \text{sh}) \rangle \\ | \text{LAss}_1: & \\ & \llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, (h, \text{ls}, \text{sh}) \rangle; i < \text{size } \text{ls}; \text{ls}' = \text{ls}[i := v] \rrbracket \\ & \Longrightarrow P \vdash_1 \langle i := e, s_0 \rangle \Rightarrow \langle \text{unit}, (h, \text{ls}', \text{sh}) \rangle \\ | \text{LAssThrow}_1: & \\ & P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow \\ & P \vdash_1 \langle i := e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \\ | \text{FAcc}_1: & \\ & \llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, \text{ls}, \text{sh}) \rangle; h a = \text{Some}(C, \text{fs}); \\ & \quad P \vdash C \text{ has } F, \text{NonStatic}:t \text{ in } D; \\ & \quad \text{fs}(F, D) = \text{Some } v \rrbracket \\ & \Longrightarrow P \vdash_1 \langle e \cdot F\{D\}, s_0 \rangle \Rightarrow \langle \text{Val } v, (h, \text{ls}, \text{sh}) \rangle \\ | \text{FAccNull}_1: & \\ & P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \Longrightarrow \\ & P \vdash_1 \langle e \cdot F\{D\}, s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_1 \rangle \\ | \text{FAccThrow}_1: & \\ & P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow \\ & P \vdash_1 \langle e \cdot F\{D\}, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \\ | \text{FAccNone}_1: & \\ & \llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, \text{ls}, \text{sh}) \rangle; h a = \text{Some}(C, \text{fs}); \\ & \quad \neg(\exists b t. P \vdash C \text{ has } F, b:t \text{ in } D) \rrbracket \\ & \Longrightarrow P \vdash_1 \langle e \cdot F\{D\}, s_0 \rangle \Rightarrow \langle \text{THROW NoSuchFieldError}, (h, \text{ls}, \text{sh}) \rangle \\ | \text{FAccStatic}_1: & \\ & \llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, \text{ls}, \text{sh}) \rangle; h a = \text{Some}(C, \text{fs}); \\ & \quad P \vdash C \text{ has } F, \text{Static}:t \text{ in } D \rrbracket \\ & \Longrightarrow P \vdash_1 \langle e \cdot F\{D\}, s_0 \rangle \Rightarrow \langle \text{THROW IncompatibleClassChangeError}, (h, \text{ls}, \text{sh}) \rangle \\ | \text{SFAcc}_1: & \\ & \llbracket P \vdash C \text{ has } F, \text{Static}:t \text{ in } D; \\ & \quad \text{sh } D = \text{Some}(\text{sfs}, \text{Done}); \\ & \quad \text{sfs } F = \text{Some } v \rrbracket \\ & \Longrightarrow P \vdash_1 \langle C \cdot_s F\{D\}, (h, \text{ls}, \text{sh}) \rangle \Rightarrow \langle \text{Val } v, (h, \text{ls}, \text{sh}) \rangle \\ | \text{SFAccInit}_1: & \\ & \llbracket P \vdash C \text{ has } F, \text{Static}:t \text{ in } D; \\ & \quad \# \text{sfs}. \text{sh } D = \text{Some}(\text{sfs}, \text{Done}); P \vdash_1 \langle \text{INIT } D ([D], \text{False}) \leftarrow \text{unit}, (h, \text{ls}, \text{sh}) \rangle \Rightarrow \langle \text{Val } v', (h', \text{ls}', \text{sh}') \rangle; \\ & \quad \text{sh}' D = \text{Some}(\text{sfs}, i); \\ & \quad \text{sfs } F = \text{Some } v \rrbracket \\ & \Longrightarrow P \vdash_1 \langle C \cdot_s F\{D\}, (h, \text{ls}, \text{sh}) \rangle \Rightarrow \langle \text{Val } v, (h', \text{ls}', \text{sh}') \rangle \\ | \text{SFAccInitThrow}_1: & \\ & \llbracket P \vdash C \text{ has } F, \text{Static}:t \text{ in } D; \\ & \quad \# \text{sfs}. \text{sh } D = \text{Some}(\text{sfs}, \text{Done}); P \vdash_1 \langle \text{INIT } D ([D], \text{False}) \leftarrow \text{unit}, (h, \text{ls}, \text{sh}) \rangle \Rightarrow \langle \text{throw } a, s^\wedge \rangle \rrbracket \\ & \Longrightarrow P \vdash_1 \langle C \cdot_s F\{D\}, (h, \text{ls}, \text{sh}) \rangle \Rightarrow \langle \text{throw } a, s^\wedge \rangle \end{aligned}$$

$|$ *SFAccNone*₁:
 $\llbracket \neg(\exists b t. P \vdash C \text{ has } F, b:t \text{ in } D) \rrbracket$
 $\implies P \vdash_1 \langle C \cdot_s F\{D\}, s \rangle \Rightarrow \langle \text{THROW NoSuchFieldError}, s \rangle$

$|$ *SFAccNonStatic*₁:
 $\llbracket P \vdash C \text{ has } F, \text{NonStatic}:t \text{ in } D \rrbracket$
 $\implies P \vdash_1 \langle C \cdot_s F\{D\}, s \rangle \Rightarrow \langle \text{THROW IncompatibleClassChangeError}, s \rangle$

$|$ *FAss*₁:
 $\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2, sh_2) \rangle;$
 $h_2 a = \text{Some}(C, fs); P \vdash C \text{ has } F, \text{NonStatic}:t \text{ in } D;$
 $fs' = fs(F, D) \mapsto v; h_2' = h_2(a \mapsto (C, fs')) \rrbracket$
 $\implies P \vdash_1 \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{unit}, (h_2', l_2, sh_2) \rangle$

$|$ *FAssNull*₁:
 $\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, s_2 \rangle \rrbracket \implies$
 $P \vdash_1 \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_2 \rangle$

$|$ *FAssThrow*₁₁:
 $P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \implies$
 $P \vdash_1 \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$

$|$ *FAssThrow*₂₁:
 $\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle \rrbracket$
 $\implies P \vdash_1 \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle$

$|$ *FAssNone*₁:
 $\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2, sh_2) \rangle;$
 $h_2 a = \text{Some}(C, fs); \neg(\exists b t. P \vdash C \text{ has } F, b:t \text{ in } D) \rrbracket$
 $\implies P \vdash_1 \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{THROW NoSuchFieldError}, (h_2, l_2, sh_2) \rangle$

$|$ *FAssStatic*₁:
 $\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2, sh_2) \rangle;$
 $h_2 a = \text{Some}(C, fs); P \vdash C \text{ has } F, \text{Static}:t \text{ in } D \rrbracket$
 $\implies P \vdash_1 \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{THROW IncompatibleClassChangeError}, (h_2, l_2, sh_2) \rangle$

$|$ *SFAss*₁:
 $\llbracket P \vdash_1 \langle e_2, s_0 \rangle \Rightarrow \langle \text{Val } v, (h_1, l_1, sh_1) \rangle;$
 $P \vdash C \text{ has } F, \text{Static}:t \text{ in } D;$
 $sh_1 D = \text{Some}(sfs, \text{Done}); sfs' = sfs(F \mapsto v); sh_1' = sh_1(D \mapsto (sfs', \text{Done})) \rrbracket$
 $\implies P \vdash_1 \langle C \cdot_s F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{unit}, (h_1, l_1, sh_1') \rangle$

$|$ *SFAssInit*₁:
 $\llbracket P \vdash_1 \langle e_2, s_0 \rangle \Rightarrow \langle \text{Val } v, (h_1, l_1, sh_1) \rangle;$
 $P \vdash C \text{ has } F, \text{Static}:t \text{ in } D;$
 $\nexists sfs. sh_1 D = \text{Some}(sfs, \text{Done}); P \vdash_1 \langle \text{INIT } D ([D], \text{False}) \leftarrow \text{unit}, (h_1, l_1, sh_1) \rangle \Rightarrow \langle \text{Val } v', (h', l', sh') \rangle;$
 $sh' D = \text{Some}(sfs, i);$
 $sfs' = sfs(F \mapsto v); sh'' = sh'(D \mapsto (sfs', i)) \rrbracket$
 $\implies P \vdash_1 \langle C \cdot_s F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{unit}, (h', l', sh'') \rangle$

$|$ *SFAssInitThrow*₁:
 $\llbracket P \vdash_1 \langle e_2, s_0 \rangle \Rightarrow \langle \text{Val } v, (h_1, l_1, sh_1) \rangle;$
 $P \vdash C \text{ has } F, \text{Static}:t \text{ in } D;$
 $\nexists sfs. sh_1 D = \text{Some}(sfs, \text{Done}); P \vdash_1 \langle \text{INIT } D ([D], \text{False}) \leftarrow \text{unit}, (h_1, l_1, sh_1) \rangle \Rightarrow \langle \text{throw } a, s^\wedge \rangle \rrbracket$
 $\implies P \vdash_1 \langle C \cdot_s F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{throw } a, s^\wedge \rangle$

$|$ *SFAssThrow*₁:
 $P \vdash_1 \langle e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle$
 $\implies P \vdash_1 \langle C \cdot_s F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle$

$|$ *SFAssNone*₁:
 $\llbracket P \vdash_1 \langle e_2, s_0 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2, sh_2) \rangle;$
 $\neg(\exists b t. P \vdash C \text{ has } F, b:t \text{ in } D) \rrbracket$

$\Rightarrow P \vdash_1 \langle C \cdot_s F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{THROW NoSuchFieldError}, (h_2, l_2, sh_2) \rangle$
 | *SFAssNonStatic*₁:
 $\llbracket P \vdash_1 \langle e_2, s_0 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2, sh_2) \rangle; \quad P \vdash C \text{ has } F, \text{NonStatic}: t \text{ in } D \rrbracket$
 $\Rightarrow P \vdash_1 \langle C \cdot_s F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{THROW IncompatibleClassChangeError}, (h_2, l_2, sh_2) \rangle$

| *CallObjThrow*₁:
 $P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Rightarrow$
 $P \vdash_1 \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$

| *CallNull*₁:
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle; P \vdash_1 \langle es, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, s_2 \rangle \rrbracket$
 $\Rightarrow P \vdash_1 \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_2 \rangle$

| *Call*₁:
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash_1 \langle es, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_2, ls_2, sh_2) \rangle;$
 $h_2 \ a = \text{Some}(C.fs); P \vdash C \text{ sees } M, \text{NonStatic}: Ts \rightarrow T = \text{body in } D;$
 $\text{size } vs = \text{size } Ts; ls_2' = (\text{Addr } a) \# vs \ @ \ \text{replicate } (\text{max-vars body}) \ \text{undefined};$
 $P \vdash_1 \langle \text{body}, (h_2, ls_2', sh_2) \rangle \Rightarrow \langle e', (h_3, ls_3, sh_3) \rangle \rrbracket$
 $\Rightarrow P \vdash_1 \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle e', (h_3, ls_2, sh_3) \rangle$

| *CallParamsThrow*₁:
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash_1 \langle es, s_1 \rangle [\Rightarrow] \langle es', s_2 \rangle;$
 $es' = \text{map Val } vs \ @ \ \text{throw } ex \ \# \ es_2 \rrbracket$
 $\Rightarrow P \vdash_1 \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle \text{throw } ex, s_2 \rangle$

| *CallNone*₁:
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash_1 \langle ps, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_2, ls_2, sh_2) \rangle;$
 $h_2 \ a = \text{Some}(C.fs); \neg(\exists b \ Ts \ T \ \text{body } D. P \vdash C \text{ sees } M, b: Ts \rightarrow T = \text{body in } D) \rrbracket$
 $\Rightarrow P \vdash_1 \langle e \cdot M(ps), s_0 \rangle \Rightarrow \langle \text{THROW NoSuchMethodError}, (h_2, ls_2, sh_2) \rangle$

| *CallStatic*₁:
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash_1 \langle ps, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_2, ls_2, sh_2) \rangle;$
 $h_2 \ a = \text{Some}(C.fs); P \vdash C \text{ sees } M, \text{Static}: Ts \rightarrow T = \text{body in } D \rrbracket$
 $\Rightarrow P \vdash_1 \langle e \cdot M(ps), s_0 \rangle \Rightarrow \langle \text{THROW IncompatibleClassChangeError}, (h_2, ls_2, sh_2) \rangle$

| *SCallParamsThrow*₁:
 $\llbracket P \vdash_1 \langle es, s_0 \rangle [\Rightarrow] \langle es', s_2 \rangle; es' = \text{map Val } vs \ @ \ \text{throw } ex \ \# \ es_2 \rrbracket$
 $\Rightarrow P \vdash_1 \langle C \cdot_s M(es), s_0 \rangle \Rightarrow \langle \text{throw } ex, s_2 \rangle$

| *SCallNone*₁:
 $\llbracket P \vdash_1 \langle ps, s_0 \rangle [\Rightarrow] \langle \text{map Val } vs, s_2 \rangle;$
 $\neg(\exists b \ Ts \ T \ \text{body } D. P \vdash C \text{ sees } M, b: Ts \rightarrow T = \text{body in } D) \rrbracket$
 $\Rightarrow P \vdash_1 \langle C \cdot_s M(ps), s_0 \rangle \Rightarrow \langle \text{THROW NoSuchMethodError}, s_2 \rangle$

| *SCallNonStatic*₁:
 $\llbracket P \vdash_1 \langle ps, s_0 \rangle [\Rightarrow] \langle \text{map Val } vs, s_2 \rangle;$
 $P \vdash C \text{ sees } M, \text{NonStatic}: Ts \rightarrow T = \text{body in } D \rrbracket$
 $\Rightarrow P \vdash_1 \langle C \cdot_s M(ps), s_0 \rangle \Rightarrow \langle \text{THROW IncompatibleClassChangeError}, s_2 \rangle$

| *SCallInitThrow*₁:
 $\llbracket P \vdash_1 \langle ps, s_0 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_1, ls_1, sh_1) \rangle;$
 $P \vdash C \text{ sees } M, \text{Static}: Ts \rightarrow T = \text{body in } D;$
 $\nexists \text{sfs. } sh_1 \ D = \text{Some}(\text{sfs}, \text{Done}); M \neq \text{clinit};$
 $P \vdash_1 \langle \text{INIT } D \ ([D], \text{False}) \leftarrow \text{unit}, (h_1, ls_1, sh_1) \rangle \Rightarrow \langle \text{throw } a, s^\wedge \rangle \rrbracket$
 $\Rightarrow P \vdash_1 \langle C \cdot_s M(ps), s_0 \rangle \Rightarrow \langle \text{throw } a, s^\wedge \rangle$

| *SCallInit*₁:
 $\llbracket P \vdash_1 \langle ps, s_0 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_1, ls_1, sh_1) \rangle;$
 $P \vdash C \text{ sees } M, \text{Static}: Ts \rightarrow T = \text{body in } D;$
 $\nexists \text{sfs. } sh_1 \ D = \text{Some}(\text{sfs}, \text{Done}); M \neq \text{clinit};$
 $P \vdash_1 \langle \text{INIT } D \ ([D], \text{False}) \leftarrow \text{unit}, (h_1, ls_1, sh_1) \rangle \Rightarrow \langle \text{Val } v', (h_2, ls_2, sh_2) \rangle;$
 $\text{size } vs = \text{size } Ts; ls_2' = vs \ @ \ \text{replicate } (\text{max-vars body}) \ \text{undefined};$

$$\begin{aligned}
& P \vdash_1 \langle \text{body}, (h_2, ls_2', sh_2) \rangle \Rightarrow \langle e', (h_3, ls_3, sh_3) \rangle \text{]} \\
& \Longrightarrow P \vdash_1 \langle C \cdot_s M(ps), s_0 \rangle \Rightarrow \langle e', (h_3, ls_2, sh_3) \rangle \\
| \text{SCall}_1: \\
& \text{[} P \vdash_1 \langle ps, s_0 \rangle \text{ [}\Rightarrow\text{]} \langle \text{map Val } vs, (h_2, ls_2, sh_2) \rangle; \\
& \quad P \vdash C \text{ sees } M, \text{Static: } Ts \rightarrow T = \text{body in } D; \\
& \quad sh_2 D = \text{Some}(sfs, Done) \vee (M = \text{clinit} \wedge sh_2 D = \text{[(sfs, Processing)]}); \\
& \quad \text{size } vs = \text{size } Ts; ls_2' = vs @ \text{replicate (max-vars body) undefined}; \\
& \quad P \vdash_1 \langle \text{body}, (h_2, ls_2', sh_2) \rangle \Rightarrow \langle e', (h_3, ls_3, sh_3) \rangle \text{]} \\
& \Longrightarrow P \vdash_1 \langle C \cdot_s M(ps), s_0 \rangle \Rightarrow \langle e', (h_3, ls_2, sh_3) \rangle \\
| \text{Block}_1: \\
& P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle e', s_1 \rangle \Longrightarrow P \vdash_1 \langle \text{Block } i \ T \ e, s_0 \rangle \Rightarrow \langle e', s_1 \rangle \\
| \text{Seq}_1: \\
& \text{[} P \vdash_1 \langle e_0, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash_1 \langle e_1, s_1 \rangle \Rightarrow \langle e_2, s_2 \rangle \text{]} \\
& \Longrightarrow P \vdash_1 \langle e_0;; e_1, s_0 \rangle \Rightarrow \langle e_2, s_2 \rangle \\
| \text{SeqThrow}_1: \\
& P \vdash_1 \langle e_0, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle \Longrightarrow \\
& P \vdash_1 \langle e_0;; e_1, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle \\
| \text{CondT}_1: \\
& \text{[} P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle; P \vdash_1 \langle e_1, s_1 \rangle \Rightarrow \langle e', s_2 \rangle \text{]} \\
& \Longrightarrow P \vdash_1 \langle \text{if } (e) \ e_1 \ \text{else } e_2, s_0 \rangle \Rightarrow \langle e', s_2 \rangle \\
| \text{CondF}_1: \\
& \text{[} P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{false}, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle e', s_2 \rangle \text{]} \\
& \Longrightarrow P \vdash_1 \langle \text{if } (e) \ e_1 \ \text{else } e_2, s_0 \rangle \Rightarrow \langle e', s_2 \rangle \\
| \text{CondThrow}_1: \\
& P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow \\
& P \vdash_1 \langle \text{if } (e) \ e_1 \ \text{else } e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \\
| \text{WhileF}_1: \\
& P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{false}, s_1 \rangle \Longrightarrow \\
& P \vdash_1 \langle \text{while } (e) \ c, s_0 \rangle \Rightarrow \langle \text{unit}, s_1 \rangle \\
| \text{WhileT}_1: \\
& \text{[} P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle; P \vdash_1 \langle c, s_1 \rangle \Rightarrow \langle \text{Val } v_1, s_2 \rangle; \\
& \quad P \vdash_1 \langle \text{while } (e) \ c, s_2 \rangle \Rightarrow \langle e_3, s_3 \rangle \text{]} \\
& \Longrightarrow P \vdash_1 \langle \text{while } (e) \ c, s_0 \rangle \Rightarrow \langle e_3, s_3 \rangle \\
| \text{WhileCondThrow}_1: \\
& P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow \\
& P \vdash_1 \langle \text{while } (e) \ c, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \\
| \text{WhileBodyThrow}_1: \\
& \text{[} P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle; P \vdash_1 \langle c, s_1 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle \text{]} \\
& \Longrightarrow P \vdash_1 \langle \text{while } (e) \ c, s_0 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle \\
| \text{Throw}_1: \\
& P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle \Longrightarrow \\
& P \vdash_1 \langle \text{throw } e, s_0 \rangle \Rightarrow \langle \text{Throw } a, s_1 \rangle \\
| \text{ThrowNull}_1: \\
& P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \Longrightarrow \\
& P \vdash_1 \langle \text{throw } e, s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_1 \rangle \\
| \text{ThrowThrow}_1: \\
& P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow \\
& P \vdash_1 \langle \text{throw } e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle
\end{aligned}$$

| *Try*₁:
 $P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle \Longrightarrow$
 $P \vdash_1 \langle \text{try } e_1 \text{ catch}(C \ i) \ e_2, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle$

| *TryCatch*₁:
 $\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Throw } a, (h_1, ls_1, sh_1) \rangle;$
 $h_1 \ a = \text{Some}(D, fs); P \vdash D \preceq^* C; i < \text{length } ls_1;$
 $P \vdash_1 \langle e_2, (h_1, ls_1[i := \text{Addr } a], sh_1) \rangle \Rightarrow \langle e_2', (h_2, ls_2, sh_2) \rangle \rrbracket$
 $\Longrightarrow P \vdash_1 \langle \text{try } e_1 \text{ catch}(C \ i) \ e_2, s_0 \rangle \Rightarrow \langle e_2', (h_2, ls_2, sh_2) \rangle$

| *TryThrow*₁:
 $\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Throw } a, (h_1, ls_1, sh_1) \rangle; h_1 \ a = \text{Some}(D, fs); \neg P \vdash D \preceq^* C \rrbracket$
 $\Longrightarrow P \vdash_1 \langle \text{try } e_1 \text{ catch}(C \ i) \ e_2, s_0 \rangle \Rightarrow \langle \text{Throw } a, (h_1, ls_1, sh_1) \rangle$

| *Nil*₁:
 $P \vdash_1 \langle [], s \rangle [\Rightarrow] \langle [], s \rangle$

| *Cons*₁:
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash_1 \langle es, s_1 \rangle [\Rightarrow] \langle es', s_2 \rangle \rrbracket$
 $\Longrightarrow P \vdash_1 \langle e \# es, s_0 \rangle [\Rightarrow] \langle \text{Val } v \# es', s_2 \rangle$

| *ConsThrow*₁:
 $P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow$
 $P \vdash_1 \langle e \# es, s_0 \rangle [\Rightarrow] \langle \text{throw } e' \# es, s_1 \rangle$

— init rules

| *InitFinal*₁:
 $P \vdash_1 \langle e, s \rangle \Rightarrow \langle e', s' \rangle \Longrightarrow P \vdash_1 \langle \text{INIT } C \ (\text{Nil}, b) \leftarrow e, s \rangle \Rightarrow \langle e', s' \rangle$

| *InitNone*₁:
 $\llbracket sh \ C = \text{None}; P \vdash_1 \langle \text{INIT } C' \ (C \# Cs, \text{False}) \leftarrow e, (h, l, sh(C \mapsto (\text{sblank } P \ C, \text{Prepared}))) \rangle \Rightarrow \langle e', s' \rangle \rrbracket$
 $\Longrightarrow P \vdash_1 \langle \text{INIT } C' \ (C \# Cs, \text{False}) \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle$

| *InitDone*₁:
 $\llbracket sh \ C = \text{Some}(sfs, \text{Done}); P \vdash_1 \langle \text{INIT } C' \ (Cs, \text{True}) \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle \rrbracket$
 $\Longrightarrow P \vdash_1 \langle \text{INIT } C' \ (C \# Cs, \text{False}) \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle$

| *InitProcessing*₁:
 $\llbracket sh \ C = \text{Some}(sfs, \text{Processing}); P \vdash_1 \langle \text{INIT } C' \ (Cs, \text{True}) \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle \rrbracket$
 $\Longrightarrow P \vdash_1 \langle \text{INIT } C' \ (C \# Cs, \text{False}) \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle$

| *InitError*₁:
 $\llbracket sh \ C = \text{Some}(sfs, \text{Error});$
 $P \vdash_1 \langle \text{RI } (C, \text{THROW } \text{NoClassDefFoundError}); Cs \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle \rrbracket$
 $\Longrightarrow P \vdash_1 \langle \text{INIT } C' \ (C \# Cs, \text{False}) \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle$

| *InitObject*₁:
 $\llbracket sh \ C = \text{Some}(sfs, \text{Prepared});$
 $C = \text{Object};$
 $sh' = sh(C \mapsto (sfs, \text{Processing}));$
 $P \vdash_1 \langle \text{INIT } C' \ (C \# Cs, \text{True}) \leftarrow e, (h, l, sh') \rangle \Rightarrow \langle e', s' \rangle \rrbracket$
 $\Longrightarrow P \vdash_1 \langle \text{INIT } C' \ (C \# Cs, \text{False}) \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle$

| *InitNonObject*₁:
 $\llbracket sh \ C = \text{Some}(sfs, \text{Prepared});$
 $C \neq \text{Object};$
 $\text{class } P \ C = \text{Some } (D, r);$
 $sh' = sh(C \mapsto (sfs, \text{Processing}));$
 $P \vdash_1 \langle \text{INIT } C' \ (D \# C \# Cs, \text{False}) \leftarrow e, (h, l, sh') \rangle \Rightarrow \langle e', s' \rangle \rrbracket$
 $\Longrightarrow P \vdash_1 \langle \text{INIT } C' \ (C \# Cs, \text{False}) \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle$

| *InitRInit*₁:

$$P \vdash_1 \langle RI (C, C \cdot_s \text{clinit}(\square)); Cs \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle \\ \Longrightarrow P \vdash_1 \langle INIT C' (C \# Cs, True) \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', s' \rangle$$

| *RInit*₁:

$$\llbracket P \vdash_1 \langle e, s \rangle \Rightarrow \langle Val v, (h', l', sh') \rangle; \\ sh' C = Some(sfs, i); sh'' = sh'(C \mapsto (sfs, Done)); \\ C' = last(C \# Cs); \\ P \vdash_1 \langle INIT C' (Cs, True) \leftarrow e', (h', l', sh'') \rangle \Rightarrow \langle e_1, s_1 \rangle \rrbracket \\ \Longrightarrow P \vdash_1 \langle RI (C, e); Cs \leftarrow e', s \rangle \Rightarrow \langle e_1, s_1 \rangle$$

| *RInitInitFail*₁:

$$\llbracket P \vdash_1 \langle e, s \rangle \Rightarrow \langle throw a, (h', l', sh') \rangle; \\ sh' C = Some(sfs, i); sh'' = sh'(C \mapsto (sfs, Error)); \\ P \vdash_1 \langle RI (D, throw a); Cs \leftarrow e', (h', l', sh'') \rangle \Rightarrow \langle e_1, s_1 \rangle \rrbracket \\ \Longrightarrow P \vdash_1 \langle RI (C, e); D \# Cs \leftarrow e', s \rangle \Rightarrow \langle e_1, s_1 \rangle$$

| *RInitFailFinal*₁:

$$\llbracket P \vdash_1 \langle e, s \rangle \Rightarrow \langle throw a, (h', l', sh') \rangle; \\ sh' C = Some(sfs, i); sh'' = sh'(C \mapsto (sfs, Error)) \rrbracket \\ \Longrightarrow P \vdash_1 \langle RI (C, e); Nil \leftarrow e', s \rangle \Rightarrow \langle throw a, (h', l', sh'') \rangle$$

inductive-cases *eval*₁-cases [cases set]:

$$P \vdash_1 \langle new C, s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle Cast C e, s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle Val v, s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle e_1 \ll \text{bop} \gg e_2, s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle Var v, s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle V := e, s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle e \cdot F \{D\}, s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle C \cdot_s F \{D\}, s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle e_1 \cdot F \{D\} := e_2, s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle C \cdot_s F \{D\} := e_2, s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle e \cdot M(es), s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle C \cdot_s M(es), s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle \{V:T; e_1\}, s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle e_1 ;; e_2, s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle if (e) e_1 else e_2, s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle while (b) c, s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle throw e, s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle try e_1 catch (C V) e_2, s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle INIT C (Cs, b) \leftarrow e, s \rangle \Rightarrow \langle e', s' \rangle \\ P \vdash_1 \langle RI (C, e); Cs \leftarrow e_0, s \rangle \Rightarrow \langle e', s' \rangle$$

inductive-cases *evals*₁-cases [cases set]:

$$P \vdash_1 \langle \square, s \rangle [\Rightarrow] \langle e', s' \rangle \\ P \vdash_1 \langle e \# es, s \rangle [\Rightarrow] \langle e', s' \rangle$$

lemma *eval*₁-final: $P \vdash_1 \langle e, s \rangle \Rightarrow \langle e', s' \rangle \Longrightarrow \text{final } e'$
and *evals*₁-final: $P \vdash_1 \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \Longrightarrow \text{finals } es'$

lemma *eval*₁-final-same:

assumes *eval*: $P \vdash_1 \langle e, s \rangle \Rightarrow \langle e', s' \rangle$ **shows** *final* $e \Longrightarrow e = e' \wedge s = s'$

3.1.1 Property preservation

lemma *eval₁-preserves-len*:

$$P \vdash_1 \langle e_0, (h_0, ls_0, sh_0) \rangle \Rightarrow \langle e_1, (h_1, ls_1, sh_1) \rangle \Longrightarrow \text{length } ls_0 = \text{length } ls_1$$

and *evals₁-preserves-len*:

$$P \vdash_1 \langle es_0, (h_0, ls_0, sh_0) \rangle [\Rightarrow] \langle es_1, (h_1, ls_1, sh_1) \rangle \Longrightarrow \text{length } ls_0 = \text{length } ls_1$$

lemma *evals₁-preserves-len*:

$$\bigwedge es' s'. P \vdash_1 \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \Longrightarrow \text{length } es = \text{length } es'$$

lemma *clinit₁-loc-pres*:

$$P \vdash_1 \langle C_0 \cdot_s \text{clinit}(\square), (h, l, sh) \rangle \Rightarrow \langle e', (h', l', sh') \rangle \Longrightarrow l = l'$$

by(*auto elim!*: *eval₁-cases*(12) *evals₁-cases*(1))

lemma

shows *init₁-ri₁-same-loc*: $P \vdash_1 \langle e, (h, l, sh) \rangle \Rightarrow \langle e', (h', l', sh') \rangle$

$$\begin{aligned} \Longrightarrow & (\bigwedge C Cs b M a. e = \text{INIT } C (Cs, b) \leftarrow \text{unit} \vee e = C \cdot_s M(\square) \vee e = \text{RI } (C, \text{Throw } a) ; Cs \leftarrow \text{unit} \\ & \vee e = \text{RI } (C, C \cdot_s \text{clinit}(\square)) ; Cs \leftarrow \text{unit} \\ \Longrightarrow & l = l') \end{aligned}$$

and $P \vdash_1 \langle es, (h, l, sh) \rangle [\Rightarrow] \langle es', (h', l', sh') \rangle \Longrightarrow \text{True}$

proof(*induct rule*: *eval₁-evals₁-inducts*)

case (*RInitInitFail₁* *e h l sh a'*)

then show *?case using eval₁-final*[*of - - - throw a'*]

by(*fastforce dest*: *eval₁-final-same*[*of - Throw a'*])

next

case *RInitFailFinal₁* **then show** *?case by*(*auto dest*: *eval₁-final-same*)

qed(*auto dest*: *evals₁-cases eval₁-cases*(17) *eval₁-final-same*)

lemma *init₁-same-loc*: $P \vdash_1 \langle \text{INIT } C (Cs, b) \leftarrow \text{unit}, (h, l, sh) \rangle \Rightarrow \langle e', (h', l', sh') \rangle \Longrightarrow l = l'$

by(*simp add*: *init₁-ri₁-same-loc*)

theorem *eval₁-hext*: $P \vdash_1 \langle e, (h, l, sh) \rangle \Rightarrow \langle e', (h', l', sh') \rangle \Longrightarrow h \trianglelefteq h'$

and *evals₁-hext*: $P \vdash_1 \langle es, (h, l, sh) \rangle [\Rightarrow] \langle es', (h', l', sh') \rangle \Longrightarrow h \trianglelefteq h'$

3.1.2 Initialization

lemma *rinit₁-throw*:

$$P_1 \vdash_1 \langle \text{RI } (D, \text{Throw } xa) ; Cs \leftarrow e, (h, l, sh) \rangle \Rightarrow \langle e', (h', l', sh') \rangle$$

$$\Longrightarrow e' = \text{Throw } xa$$

proof(*induct* *Cs arbitrary*: *D h l sh h' l' sh'*)

case *Nil* **then show** *?case*

proof(*rule eval₁-cases*(20)) **qed**(*auto elim*: *eval₁-cases*)

next

case (*Cons C Cs*) **show** *?case using Cons.prem*s

proof(*induct rule*: *eval₁-cases*(20))

case *RInit₁*: 1

then show *?case using Cons.hyps by*(*auto elim*: *eval₁-cases*)

next

case *RInitInitFail₁*: 2

then show *?case using Cons.hyps eval₁-final-same final-def by blast*

next

case *RInitFailFinal₁*: 3 **then show** *?case by simp*

qed

qed

lemma *rinit₁-throwE*:

$P \vdash_1 \langle RI (C, \text{throw } e) ; Cs \leftarrow e_0, s \rangle \Rightarrow \langle e', s' \rangle$
 $\implies \exists a s_t. e' = \text{throw } a \wedge P \vdash_1 \langle \text{throw } e, s \rangle \Rightarrow \langle \text{throw } a, s_t \rangle$

proof(*induct Cs arbitrary: C e s*)

case *Nil*

then show *?case*

proof(*rule eval₁-cases(20)*) — RI

fix *v h' l' sh'* **assume** $P \vdash_1 \langle \text{throw } e, s \rangle \Rightarrow \langle \text{Val } v, (h', l', sh') \rangle$

then show *?case using eval₁-cases(17) by blast*

qed(*auto*)

next

case (*Cons C' Cs'*)

show *?case using Cons.prem(1)*

proof(*rule eval₁-cases(20)*) — RI

fix *v h' l' sh'* **assume** $P \vdash_1 \langle \text{throw } e, s \rangle \Rightarrow \langle \text{Val } v, (h', l', sh') \rangle$

then show *?case using eval₁-cases(17) by blast*

next

fix *a h' l' sh' sfs i D Cs''*

assume *e''step*: $P \vdash_1 \langle \text{throw } e, s \rangle \Rightarrow \langle \text{throw } a, (h', l', sh') \rangle$

and *shC*: $sh' C = [(sfs, i)]$

and *riD*: $P \vdash_1 \langle RI (D, \text{throw } a) ; Cs'' \leftarrow e_0, (h', l', sh'(C \mapsto (sfs, \text{Error}))) \rangle \Rightarrow \langle e', s' \rangle$

and $C' \# Cs' = D \# Cs''$

then show *?thesis using Cons.hyps eval₁-final eval₁-final-same by blast*

qed(*simp*)

qed

end

3.2 Well-Formedness of Intermediate Language

theory *J1WellForm*

imports *../J/JWellForm J1*

begin

3.2.1 Well-Typedness

type-synonym

$env_1 = ty\ list$ — type environment indexed by variable number

inductive

$WT_1 :: [J_1\text{-prog}, env_1, expr_1, ty] \Rightarrow bool$

$\langle \langle -, \vdash_1 / - :: - \rangle \rangle [51, 51, 51] 50$

and $WTS_1 :: [J_1\text{-prog}, env_1, expr_1\ list, ty\ list] \Rightarrow bool$

$\langle \langle -, \vdash_1 / - [:: -] \rangle \rangle [51, 51, 51] 50$

for $P :: J_1\text{-prog}$

where

$WTNew_1:$

is-class $P C \implies$

$P, E \vdash_1 \text{new } C :: \text{Class } C$

| $WTCast_1:$

$\llbracket P, E \vdash_1 e :: \text{Class } D; \text{is-class } P C; P \vdash C \preceq^* D \vee P \vdash D \preceq^* C \rrbracket$

$$\Longrightarrow P, E \vdash_1 \text{Cast } C \ e :: \text{Class } C$$

| *WTVal*₁:
 $\text{typeof } v = \text{Some } T \Longrightarrow$
 $P, E \vdash_1 \text{Val } v :: T$

| *WTVar*₁:
 $\llbracket E!i = T; i < \text{size } E \rrbracket$
 $\Longrightarrow P, E \vdash_1 \text{Var } i :: T$

| *WTBinOp*₁:
 $\llbracket P, E \vdash_1 e_1 :: T_1; P, E \vdash_1 e_2 :: T_2;$
 $\text{case bop of Eq} \Rightarrow (P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1) \wedge T = \text{Boolean}$
 $\quad \quad \quad \text{Add} \Rightarrow T_1 = \text{Integer} \wedge T_2 = \text{Integer} \wedge T = \text{Integer} \rrbracket$
 $\Longrightarrow P, E \vdash_1 e_1 \llbracket \text{bop} \rrbracket e_2 :: T$

| *WTLAss*₁:
 $\llbracket E!i = T; i < \text{size } E; P, E \vdash_1 e :: T'; P \vdash T' \leq T \rrbracket$
 $\Longrightarrow P, E \vdash_1 i := e :: \text{Void}$

| *WTFAcc*₁:
 $\llbracket P, E \vdash_1 e :: \text{Class } C; P \vdash C \text{ sees } F, \text{NonStatic}:T \text{ in } D \rrbracket$
 $\Longrightarrow P, E \vdash_1 e \cdot F\{D\} :: T$

| *WTSFAcc*₁:
 $\llbracket P \vdash C \text{ sees } F, \text{Static}:T \text{ in } D \rrbracket$
 $\Longrightarrow P, E \vdash_1 C \cdot_s F\{D\} :: T$

| *WTFAss*₁:
 $\llbracket P, E \vdash_1 e_1 :: \text{Class } C; P \vdash C \text{ sees } F, \text{NonStatic}:T \text{ in } D; P, E \vdash_1 e_2 :: T'; P \vdash T' \leq T \rrbracket$
 $\Longrightarrow P, E \vdash_1 e_1 \cdot F\{D\} := e_2 :: \text{Void}$

| *WTSFAss*₁:
 $\llbracket P \vdash C \text{ sees } F, \text{Static}:T \text{ in } D; P, E \vdash_1 e_2 :: T'; P \vdash T' \leq T \rrbracket$
 $\Longrightarrow P, E \vdash_1 C \cdot_s F\{D\} := e_2 :: \text{Void}$

| *WTCall*₁:
 $\llbracket P, E \vdash_1 e :: \text{Class } C; P \vdash C \text{ sees } M, \text{NonStatic}:Ts' \rightarrow T = m \text{ in } D;$
 $P, E \vdash_1 es \llbracket :: \rrbracket Ts; P \vdash Ts \llbracket \leq \rrbracket Ts' \rrbracket$
 $\Longrightarrow P, E \vdash_1 e \cdot M(es) :: T$

| *WTSCall*₁:
 $\llbracket P \vdash C \text{ sees } M, \text{Static}:Ts \rightarrow T = m \text{ in } D;$
 $P, E \vdash_1 es \llbracket :: \rrbracket Ts'; P \vdash Ts' \llbracket \leq \rrbracket Ts; M \neq \text{clinit} \rrbracket$
 $\Longrightarrow P, E \vdash_1 C \cdot_s M(es) :: T$

| *WTBlock*₁:
 $\llbracket \text{is-type } P \ T; P, E@[T] \vdash_1 e :: T' \rrbracket$
 $\Longrightarrow P, E \vdash_1 \{i:T; e\} :: T'$

| *WTSeq*₁:
 $\llbracket P, E \vdash_1 e_1 :: T_1; P, E \vdash_1 e_2 :: T_2 \rrbracket$
 $\Longrightarrow P, E \vdash_1 e_1;;e_2 :: T_2$

| *WTCond*₁:
 $\llbracket P, E \vdash_1 e :: \text{Boolean}; P, E \vdash_1 e_1 :: T_1; P, E \vdash_1 e_2 :: T_2;$
 $P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1; P \vdash T_1 \leq T_2 \longrightarrow T = T_2; P \vdash T_2 \leq T_1 \longrightarrow T = T_1 \rrbracket$
 $\Longrightarrow P, E \vdash_1 \text{if } (e) e_1 \text{ else } e_2 :: T$

| *WTWhile*₁:
 $\llbracket P, E \vdash_1 e :: \text{Boolean}; P, E \vdash_1 c :: T \rrbracket$
 $\Longrightarrow P, E \vdash_1 \text{while } (e) c :: \text{Void}$

| *WTThrow*₁:
 $P, E \vdash_1 e :: \text{Class } C \Longrightarrow$
 $P, E \vdash_1 \text{throw } e :: \text{Void}$

| *WTTry*₁:
 $\llbracket P, E \vdash_1 e_1 :: T; P, E@[\text{Class } C] \vdash_1 e_2 :: T; \text{is-class } P C \rrbracket$
 $\Longrightarrow P, E \vdash_1 \text{try } e_1 \text{ catch}(C i) e_2 :: T$

| *WTNil*₁:
 $P, E \vdash_1 [] [::] []$

| *WTCons*₁:
 $\llbracket P, E \vdash_1 e :: T; P, E \vdash_1 es [::] Ts \rrbracket$
 $\Longrightarrow P, E \vdash_1 e \# es [::] T \# Ts$

lemma *init-nWT*₁ [*simp*]: $\neg P, E \vdash_1 \text{INIT } C (Cs, b) \leftarrow e :: T$
by(*auto elim: WT*₁.*cases*)

lemma *rinit-nWT*₁ [*simp*]: $\neg P, E \vdash_1 \text{RI}(C, e); Cs \leftarrow e' :: T$
by(*auto elim: WT*₁.*cases*)

lemma *WT*_{s₁}-*same-size*: $\bigwedge Ts. P, E \vdash_1 es [::] Ts \Longrightarrow \text{size } es = \text{size } Ts$

lemma *WT*₁-*unique*:
 $P, E \vdash_1 e :: T_1 \Longrightarrow (\bigwedge T_2. P, E \vdash_1 e :: T_2 \Longrightarrow T_1 = T_2)$ **and**
 $\text{WT}_{s_1}\text{-unique}: P, E \vdash_1 es [::] Ts_1 \Longrightarrow (\bigwedge Ts_2. P, E \vdash_1 es [::] Ts_2 \Longrightarrow Ts_1 = Ts_2)$

lemma *assumes wf: wf-prog p P*
shows *WT*₁-*is-type*: $P, E \vdash_1 e :: T \Longrightarrow \text{set } E \subseteq \text{types } P \Longrightarrow \text{is-type } P T$
and $P, E \vdash_1 es [::] Ts \Longrightarrow \text{True}$
lemma *WT*₁-*nsub-RI*: $P, E \vdash_1 e :: T \Longrightarrow \neg \text{sub-RI } e$
and *WT*_{s₁}-*nsub-RIs*: $P, E \vdash_1 es [::] Ts \Longrightarrow \neg \text{sub-RIs } es$
proof(*induct rule: WT*₁-*WT*_{s₁}.*inducts*) **qed**(*simp-all*)

3.2.2 Runtime Well-Typedness

inductive

$\text{WTrt}_1 :: J_1\text{-prog} \Rightarrow \text{heap} \Rightarrow \text{sheap} \Rightarrow \text{env}_1 \Rightarrow \text{expr}_1 \Rightarrow \text{ty} \Rightarrow \text{bool}$
and $\text{WTrts}_1 :: J_1\text{-prog} \Rightarrow \text{heap} \Rightarrow \text{sheap} \Rightarrow \text{env}_1 \Rightarrow \text{expr}_1 \text{ list} \Rightarrow \text{ty list} \Rightarrow \text{bool}$
and $\text{WTrt2}_1 :: [J_1\text{-prog}, \text{env}_1, \text{heap}, \text{sheap}, \text{expr}_1, \text{ty}] \Rightarrow \text{bool}$
 $(\langle -, -, - \vdash_1 - : - \rangle [51, 51, 51, 51] 50)$
and $\text{WTrts2}_1 :: [J_1\text{-prog}, \text{env}_1, \text{heap}, \text{sheap}, \text{expr}_1 \text{ list}, \text{ty list}] \Rightarrow \text{bool}$
 $(\langle -, -, - \vdash_1 - [] \rightarrow [51, 51, 51, 51] 50)$
for $P :: J_1\text{-prog}$ **and** $h :: \text{heap}$ **and** $sh :: \text{sheap}$
where

- $P, E, h, sh \vdash_1 e : T \equiv WTrt_1 P h sh E e T$
 $| P, E, h, sh \vdash_1 es[:]Ts \equiv WTrts_1 P h sh E es Ts$
- $| WTrtNew_1:$
 $is-class P C \implies$
 $P, E, h, sh \vdash_1 new C : Class C$
- $| WTrtCast_1:$
 $\llbracket P, E, h, sh \vdash_1 e : T; is-refT T; is-class P C \rrbracket$
 $\implies P, E, h, sh \vdash_1 Cast C e : Class C$
- $| WTrtVal_1:$
 $typeof_h v = Some T \implies$
 $P, E, h, sh \vdash_1 Val v : T$
- $| WTrtVar_1:$
 $\llbracket E!i = T; i < size E \rrbracket \implies$
 $P, E, h, sh \vdash_1 Var i : T$
- $| WTrtBinOpEq_1:$
 $\llbracket P, E, h, sh \vdash_1 e_1 : T_1; P, E, h, sh \vdash_1 e_2 : T_2 \rrbracket$
 $\implies P, E, h, sh \vdash_1 e_1 \llbracket Eq \rrbracket e_2 : Boolean$
- $| WTrtBinOpAdd_1:$
 $\llbracket P, E, h, sh \vdash_1 e_1 : Integer; P, E, h, sh \vdash_1 e_2 : Integer \rrbracket$
 $\implies P, E, h, sh \vdash_1 e_1 \llbracket Add \rrbracket e_2 : Integer$
- $| WTrtLAss_1:$
 $\llbracket E!i = T; i < size E; P, E, h, sh \vdash_1 e : T'; P \vdash T' \leq T \rrbracket$
 $\implies P, E, h, sh \vdash_1 i:=e : Void$
- $| WTrtFAcc_1:$
 $\llbracket P, E, h, sh \vdash_1 e : Class C; P \vdash C has F, NonStatic:T in D \rrbracket \implies$
 $P, E, h, sh \vdash_1 e \cdot F\{D\} : T$
- $| WTrtFAccNT_1:$
 $P, E, h, sh \vdash_1 e : NT \implies$
 $P, E, h, sh \vdash_1 e \cdot F\{D\} : T$
- $| WTrtSFAcc_1:$
 $\llbracket P \vdash C has F, Static:T in D \rrbracket \implies$
 $P, E, h, sh \vdash_1 C \cdot_s F\{D\} : T$
- $| WTrtFAss_1:$
 $\llbracket P, E, h, sh \vdash_1 e_1 : Class C; P \vdash C has F, NonStatic:T in D; P, E, h, sh \vdash_1 e_2 : T_2; P \vdash T_2 \leq T \rrbracket$
 $\implies P, E, h, sh \vdash_1 e_1 \cdot F\{D\} := e_2 : Void$
- $| WTrtFAssNT_1:$
 $\llbracket P, E, h, sh \vdash_1 e_1 : NT; P, E, h, sh \vdash_1 e_2 : T_2 \rrbracket$
 $\implies P, E, h, sh \vdash_1 e_1 \cdot F\{D\} := e_2 : Void$
- $| WTrtSFAss_1:$
 $\llbracket P, E, h, sh \vdash_1 e_2 : T_2; P \vdash C has F, Static:T in D; P \vdash T_2 \leq T \rrbracket$
 $\implies P, E, h, sh \vdash_1 C \cdot_s F\{D\} := e_2 : Void$

- | *WTrtCall₁*:

$$\begin{aligned} & \llbracket P, E, h, sh \vdash_1 e : \text{Class } C; P \vdash C \text{ sees } M, \text{NonStatic}: Ts \rightarrow T = m \text{ in } D; \\ & \quad P, E, h, sh \vdash_1 es \text{ [:] } Ts'; P \vdash Ts' \leq Ts \rrbracket \\ & \implies P, E, h, sh \vdash_1 e \cdot M(es) : T \end{aligned}$$
- | *WTrtCallNT₁*:

$$\begin{aligned} & \llbracket P, E, h, sh \vdash_1 e : NT; P, E, h, sh \vdash_1 es \text{ [:] } Ts \rrbracket \\ & \implies P, E, h, sh \vdash_1 e \cdot M(es) : T \end{aligned}$$
- | *WTrtSCall₁*:

$$\begin{aligned} & \llbracket P \vdash C \text{ sees } M, \text{Static}: Ts \rightarrow T = m \text{ in } D; \\ & \quad P, E, h, sh \vdash_1 es \text{ [:] } Ts'; P \vdash Ts' \leq Ts; \\ & \quad M = \text{clinit} \rightarrow sh D = \llbracket (sfs, \text{Processing}) \rrbracket \wedge es = \text{map Val vs} \rrbracket \\ & \implies P, E, h, sh \vdash_1 C \cdot_s M(es) : T \end{aligned}$$
- | *WTrtBlock₁*:

$$\begin{aligned} & P, E @ [T], h, sh \vdash_1 e : T' \implies \\ & P, E, h, sh \vdash_1 \{i: T; e\} : T' \end{aligned}$$
- | *WTrtSeq₁*:

$$\begin{aligned} & \llbracket P, E, h, sh \vdash_1 e_1 : T_1; P, E, h, sh \vdash_1 e_2 : T_2 \rrbracket \\ & \implies P, E, h, sh \vdash_1 e_1 ;; e_2 : T_2 \end{aligned}$$
- | *WTrtCond₁*:

$$\begin{aligned} & \llbracket P, E, h, sh \vdash_1 e : \text{Boolean}; P, E, h, sh \vdash_1 e_1 : T_1; P, E, h, sh \vdash_1 e_2 : T_2; \\ & \quad P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1; P \vdash T_1 \leq T_2 \rightarrow T = T_2; P \vdash T_2 \leq T_1 \rightarrow T = T_1 \rrbracket \\ & \implies P, E, h, sh \vdash_1 \text{if } (e) e_1 \text{ else } e_2 : T \end{aligned}$$
- | *WTrtWhile₁*:

$$\begin{aligned} & \llbracket P, E, h, sh \vdash_1 e : \text{Boolean}; P, E, h, sh \vdash_1 c : T \rrbracket \\ & \implies P, E, h, sh \vdash_1 \text{while}(e) c : \text{Void} \end{aligned}$$
- | *WTrtThrow₁*:

$$\begin{aligned} & \llbracket P, E, h, sh \vdash_1 e : T_r; \text{is-refT } T_r \rrbracket \implies \\ & P, E, h, sh \vdash_1 \text{throw } e : T \end{aligned}$$
- | *WTrtTry₁*:

$$\begin{aligned} & \llbracket P, E, h, sh \vdash_1 e_1 : T_1; P, E @ [\text{Class } C], h, sh \vdash_1 e_2 : T_2; P \vdash T_1 \leq T_2 \rrbracket \\ & \implies P, E, h, sh \vdash_1 \text{try } e_1 \text{ catch}(C i) e_2 : T_2 \end{aligned}$$
- | *WTrtInit₁*:

$$\begin{aligned} & \llbracket P, E, h, sh \vdash_1 e : T; \forall C' \in \text{set } (C \# Cs). \text{is-class } P C'; \neg \text{sub-RI } e; \\ & \quad \forall C' \in \text{set } (tl Cs). \exists sfs. sh C' = \llbracket (sfs, \text{Processing}) \rrbracket; \\ & \quad b \rightarrow (\forall C' \in \text{set } Cs. \exists sfs. sh C' = \llbracket (sfs, \text{Processing}) \rrbracket); \\ & \quad \text{distinct } Cs; \text{supercls-1st } P Cs \rrbracket \\ & \implies P, E, h, sh \vdash_1 \text{INIT } C (Cs, b) \leftarrow e : T \end{aligned}$$
- | *WTrtRI₁*:

$$\begin{aligned} & \llbracket P, E, h, sh \vdash_1 e : T; P, E, h, sh \vdash_1 e' : T'; \forall C' \in \text{set } (C \# Cs). \text{is-class } P C'; \neg \text{sub-RI } e'; \\ & \quad \forall C' \in \text{set } (C \# Cs). \text{not-init } C' e; \\ & \quad \forall C' \in \text{set } Cs. \exists sfs. sh C' = \llbracket (sfs, \text{Processing}) \rrbracket; \\ & \quad \exists sfs. sh C = \llbracket (sfs, \text{Processing}) \rrbracket \vee (sh C = \llbracket (sfs, \text{Error}) \rrbracket \wedge e = \text{THROW NoClassDefFoundError}); \\ & \quad \text{distinct } (C \# Cs); \text{supercls-1st } P (C \# Cs) \rrbracket \end{aligned}$$

$$\Longrightarrow P, E, h, sh \vdash_1 RI(C, e); Cs \leftarrow e' : T'$$

— well-typed expression lists

$$\begin{array}{l} | WTrtNil_1: \\ P, E, h, sh \vdash_1 [] [:] [] \end{array}$$

$$\begin{array}{l} | WTrtCons_1: \\ \llbracket P, E, h, sh \vdash_1 e : T; P, E, h, sh \vdash_1 es [:] Ts \rrbracket \\ \Longrightarrow P, E, h, sh \vdash_1 e \# es [:] T \# Ts \end{array}$$

lemma $WT_1\text{-implies-}WTrt_1: P, E \vdash_1 e :: T \Longrightarrow P, E, h, sh \vdash_1 e : T$
and $WTs_1\text{-implies-}WTrts_1: P, E \vdash_1 es [::] Ts \Longrightarrow P, E, h, sh \vdash_1 es [:] Ts$

3.2.3 Well-formedness

primrec $\mathcal{B} :: \text{expr}_1 \Rightarrow \text{nat} \Rightarrow \text{bool}$
and $\mathcal{B}s :: \text{expr}_1 \text{ list} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**

$$\begin{array}{l} \mathcal{B} (\text{new } C) i = \text{True} \mid \\ \mathcal{B} (\text{Cast } C e) i = \mathcal{B} e i \mid \\ \mathcal{B} (\text{Val } v) i = \text{True} \mid \\ \mathcal{B} (e_1 \ll \text{bop} \gg e_2) i = (\mathcal{B} e_1 i \wedge \mathcal{B} e_2 i) \mid \\ \mathcal{B} (\text{Var } j) i = \text{True} \mid \\ \mathcal{B} (e \cdot F\{D\}) i = \mathcal{B} e i \mid \\ \mathcal{B} (C \cdot_s F\{D\}) i = \text{True} \mid \\ \mathcal{B} (j := e) i = \mathcal{B} e i \mid \\ \mathcal{B} (e_1 \cdot F\{D\} := e_2) i = (\mathcal{B} e_1 i \wedge \mathcal{B} e_2 i) \mid \\ \mathcal{B} (C \cdot_s F\{D\} := e_2) i = \mathcal{B} e_2 i \mid \\ \mathcal{B} (e \cdot M(es)) i = (\mathcal{B} e i \wedge \mathcal{B}s es i) \mid \\ \mathcal{B} (C \cdot_s M(es)) i = \mathcal{B}s es i \mid \\ \mathcal{B} (\{j:T ; e\}) i = (i = j \wedge \mathcal{B} e (i+1)) \mid \\ \mathcal{B} (e_1 ;; e_2) i = (\mathcal{B} e_1 i \wedge \mathcal{B} e_2 i) \mid \\ \mathcal{B} (\text{if } (e) e_1 \text{ else } e_2) i = (\mathcal{B} e i \wedge \mathcal{B} e_1 i \wedge \mathcal{B} e_2 i) \mid \\ \mathcal{B} (\text{throw } e) i = \mathcal{B} e i \mid \\ \mathcal{B} (\text{while } (e) c) i = (\mathcal{B} e i \wedge \mathcal{B} c i) \mid \\ \mathcal{B} (\text{try } e_1 \text{ catch } (C j) e_2) i = (\mathcal{B} e_1 i \wedge i=j \wedge \mathcal{B} e_2 (i+1)) \mid \\ \mathcal{B} (\text{INIT } C (Cs, b) \leftarrow e) i = \mathcal{B} e i \mid \\ \mathcal{B} (RI(C, e); Cs \leftarrow e') i = (\mathcal{B} e i \wedge \mathcal{B} e' i) \mid \end{array}$$

$$\begin{array}{l} \mathcal{B}s [] i = \text{True} \mid \\ \mathcal{B}s (e \# es) i = (\mathcal{B} e i \wedge \mathcal{B}s es i) \end{array}$$

definition $wf\text{-}J_1\text{-mdecl} :: J_1\text{-prog} \Rightarrow \text{cname} \Rightarrow \text{expr}_1 \text{ mdecl} \Rightarrow \text{bool}$
where

$$\begin{array}{l} wf\text{-}J_1\text{-mdecl } P C \equiv \lambda(M, b, Ts, T, \text{body}). \\ \quad \neg \text{sub-RI } \text{body} \wedge \\ \text{(case } b \text{ of} \\ \quad \text{NonStatic} \Rightarrow \\ \quad \quad (\exists T'. P, \text{Class } C \# Ts \vdash_1 \text{body} :: T' \wedge P \vdash T' \leq T) \wedge \\ \quad \quad \mathcal{D} \text{body} [\{\dots \text{size } Ts\}] \wedge \mathcal{B} \text{body} (\text{size } Ts + 1) \\ \quad \mid \text{Static} \Rightarrow (\exists T'. P, Ts \vdash_1 \text{body} :: T' \wedge P \vdash T' \leq T) \wedge \\ \quad \quad \mathcal{D} \text{body} [\{\dots < \text{size } Ts\}] \wedge \mathcal{B} \text{body} (\text{size } Ts)) \end{array}$$

lemma *wf-J₁-mdecl-NonStatic[simp]*:

wf-J₁-mdecl $P\ C\ (M, NonStatic, Ts, T, body) \equiv$
 $(\neg sub\text{-}RI\ body \wedge$
 $(\exists T'. P, Class\ C \# Ts \vdash_1\ body :: T' \wedge P \vdash T' \leq T) \wedge$
 $D\ body\ [\{..size\ Ts\}] \wedge \mathcal{B}\ body\ (size\ Ts + 1))$

lemma *wf-J₁-mdecl-Static[simp]*:

wf-J₁-mdecl $P\ C\ (M, Static, Ts, T, body) \equiv$
 $(\neg sub\text{-}RI\ body \wedge$
 $(\exists T'. P, Ts \vdash_1\ body :: T' \wedge P \vdash T' \leq T) \wedge$
 $D\ body\ [\{..<size\ Ts\}] \wedge \mathcal{B}\ body\ (size\ Ts))$

abbreviation *wf-J₁-prog* == *wf-prog wf-J₁-mdecl*

lemma *sees-wf₁-nsub-RI*:

assumes *wf*: *wf-J₁-prog* P **and** *cM*: $P \vdash C\ sees\ M, b : Ts \rightarrow T = body\ in\ D$

shows $\neg sub\text{-}RI\ body$

using *sees-wf-mdecl*[*OF wf cM*] **by**(*simp add: wf-J₁-mdecl-def wf-mdecl-def*)

lemma *wf₁-types-clinit*:

assumes *wf*: *wf-prog wf-md* P **and** *ex*: *class* $P\ C = Some\ a$ **and** *proc*: *sh* $C = [(sfs, Processing)]$

shows $P, E, h, sh \vdash_1\ C \cdot_s\ clinit(\[]) : Void$

proof –

from *ex* **obtain** $D\ fs\ ms$ **where** $a = (D, fs, ms)$ **by**(*cases a*)

then have $sP : (C, D, fs, ms) \in set\ P$ **using** *ex map-of-SomeD*[*of P C a*] **by**(*simp add: class-def*)

then have *wf-clinit* ms **using** *assms* **by**(*unfold wf-prog-def wf-cdecl-def, auto*)

then obtain m **where** $sm : (clinit, Static, [], Void, m) \in set\ ms$

by(*unfold wf-clinit-def*) *auto*

then have $P \vdash C\ sees\ clinit, Static : [] \rightarrow Void = m\ in\ C$

using *mdecl-visible*[*OF wf sP sm*] **by** *simp*

then show *?thesis* **using** *WTrtSCall₁* *proc* **by** *blast*

qed

lemma **assumes** *wf*: *wf-J₁-prog* P

shows *eval₁-proc-pres*: $P \vdash_1\ \langle e, (h, l, sh) \rangle \Rightarrow \langle e', (h', l', sh') \rangle$

$\Rightarrow not\text{-}init\ C\ e \Rightarrow \exists sfs. sh\ C = [(sfs, Processing)] \Rightarrow \exists sfs'. sh'\ C = [(sfs', Processing)]$

and *evals₁-proc-pres*: $P \vdash_1\ \langle es, (h, l, sh) \rangle [\Rightarrow] \langle es', (h', l', sh') \rangle$

$\Rightarrow not\text{-}inits\ C\ es \Rightarrow \exists sfs. sh\ C = [(sfs, Processing)] \Rightarrow \exists sfs'. sh'\ C = [(sfs', Processing)]$

lemma *clinit₁-proc-pres*:

$[[\ iwf\text{-}J_1\text{-}prog\ P; P \vdash_1\ \langle C_0 \cdot_s\ clinit(\[]), (h, l, sh) \rangle \Rightarrow \langle e', (h', l', sh') \rangle;$

$sh\ C' = [(sfs, Processing)]\]]$

$\Rightarrow \exists sfs. sh'\ C' = [(sfs, Processing)]$

by(*auto dest: eval₁-proc-pres*)

end

3.3 Program Compilation

theory *PCompiler*

imports *../Common/WellForm*

begin

definition *compM* :: $(staticb \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a\ mdecl \Rightarrow 'b\ mdecl$

where

$$\text{compM } f \equiv \lambda(M, b, Ts, T, m). (M, b, Ts, T, f b m)$$

definition $\text{compC} :: (\text{staticb} \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \text{ cdecl} \Rightarrow 'b \text{ cdecl}$

where

$$\text{compC } f \equiv \lambda(C, D, Fdecls, Mdecls). (C, D, Fdecls, \text{map } (\text{compM } f) \text{ Mdecls})$$

definition $\text{compP} :: (\text{staticb} \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \text{ prog} \Rightarrow 'b \text{ prog}$

where

$$\text{compP } f \equiv \text{map } (\text{compC } f)$$

Compilation preserves the program structure. Therefore lookup functions either commute with compilation (like method lookup) or are preserved by it (like the subclass relation).

lemma map-of-map4 :

$$\begin{aligned} \text{map-of } (\text{map } (\lambda(x, a, b, c). (x, a, b, f c)) \text{ ts}) &= \\ \text{map-option } (\lambda(a, b, c). (a, b, f c)) \circ (\text{map-of } \text{ts}) & \end{aligned}$$

lemma map-of-map245 :

$$\begin{aligned} \text{map-of } (\text{map } (\lambda(x, a, b, c, d). (x, a, b, c, f a c d)) \text{ ts}) &= \\ \text{map-option } (\lambda(a, b, c, d). (a, b, c, f a c d)) \circ (\text{map-of } \text{ts}) & \end{aligned}$$

lemma class-compP :

$$\begin{aligned} \text{class } P \ C = \text{Some } (D, fs, ms) & \\ \implies \text{class } (\text{compP } f \ P) \ C = \text{Some } (D, fs, \text{map } (\text{compM } f) \ ms) & \end{aligned}$$

lemma class-compPD :

$$\begin{aligned} \text{class } (\text{compP } f \ P) \ C = \text{Some } (D, fs, cms) & \\ \implies \exists ms. \text{class } P \ C = \text{Some}(D, fs, ms) \wedge cms = \text{map } (\text{compM } f) \ ms & \end{aligned}$$

lemma $[\text{simp}]$: $\text{is-class } (\text{compP } f \ P) \ C = \text{is-class } P \ C$

lemma $[\text{simp}]$: $\text{class } (\text{compP } f \ P) \ C = \text{map-option } (\lambda c. \text{snd}(\text{compC } f \ (C, c))) \ (\text{class } P \ C)$

lemma $\text{sees-methods-compP}$:

$$\begin{aligned} P \vdash C \text{ sees-methods } Mm \implies & \\ \text{compP } f \ P \vdash C \text{ sees-methods } (\text{map-option } (\lambda((b, Ts, T, m), D). ((b, Ts, T, f b m), D)) \circ Mm) & \end{aligned}$$

lemma sees-method-compP :

$$\begin{aligned} P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D \implies & \\ \text{compP } f \ P \vdash C \text{ sees } M, b: Ts \rightarrow T = (f b m) \text{ in } D & \end{aligned}$$

lemma $[\text{simp}]$:

$$\begin{aligned} P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D \implies & \\ \text{method } (\text{compP } f \ P) \ C \ M = (D, b, Ts, T, f b m) & \end{aligned}$$

lemma $\text{sees-methods-compPD}$:

$$\begin{aligned} \llbracket cP \vdash C \text{ sees-methods } Mm'; cP = \text{compP } f \ P \rrbracket \implies & \\ \exists Mm. P \vdash C \text{ sees-methods } Mm \wedge & \\ Mm' = (\text{map-option } (\lambda((b, Ts, T, m), D). ((b, Ts, T, f b m), D)) \circ Mm) & \end{aligned}$$

lemma $\text{sees-method-compPD}$:

$$\begin{aligned} \text{compP } f \ P \vdash C \text{ sees } M, b: Ts \rightarrow T = fm \text{ in } D \implies & \\ \exists m. P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } D \wedge f b m = fm & \end{aligned}$$

lemma $[\text{simp}]$: $\text{subcls1 } (\text{compP } f \ P) = \text{subcls1 } P$

lemma *compP-widen*[simp]: $(\text{compP } f \ P \vdash T \leq T') = (P \vdash T \leq T')$

lemma [simp]: $(\text{compP } f \ P \vdash Ts \leq Ts') = (P \vdash Ts \leq Ts')$

lemma [simp]: *is-type* (compP f P) T = *is-type* P T

lemma [simp]: $(\text{compP } (f :: \text{staticb} \Rightarrow 'a \Rightarrow 'b) \ P \vdash C \text{ has-fields FDTs}) = (P \vdash C \text{ has-fields FDTs})$

lemma *fields-compP* [simp]: *fields* (compP f P) C = *fields* P C

lemma *ifields-compP* [simp]: *ifields* (compP f P) C = *ifields* P C

lemma *blank-compP* [simp]: *blank* (compP f P) C = *blank* P C

lemma *isfields-compP* [simp]: *isfields* (compP f P) C = *isfields* P C

lemma *sblank-compP* [simp]: *sblank* (compP f P) C = *sblank* P C

lemma *sees-fields-compP* [simp]: $(\text{compP } f \ P \vdash C \text{ sees } F, b: T \text{ in } D) = (P \vdash C \text{ sees } F, b: T \text{ in } D)$

lemma *has-field-compP* [simp]: $(\text{compP } f \ P \vdash C \text{ has } F, b: T \text{ in } D) = (P \vdash C \text{ has } F, b: T \text{ in } D)$

lemma *field-compP* [simp]: *field* (compP f P) F D = *field* P F D

3.3.1 Invariance of wf-prog under compilation

lemma [iff]: *distinct-fst* (compP f P) = *distinct-fst* P

lemma [iff]: *distinct-fst* (map (compM f) ms) = *distinct-fst* ms

lemma [iff]: *wf-syscls* (compP f P) = *wf-syscls* P

lemma [iff]: *wf-fdecl* (compP f P) = *wf-fdecl* P

lemma *wf-clinit-compM* [iff]: *wf-clinit* (map (compM f) ms) = *wf-clinit* ms

lemma *set-compP*:

$((C, D, fs, ms') \in \text{set}(\text{compP } f \ P)) =$
 $(\exists ms. (C, D, fs, ms) \in \text{set } P \wedge ms' = \text{map } (\text{compM } f) \ ms)$

lemma *wf-cdecl-compPI*:

$\llbracket \bigwedge C \ M \ b \ Ts \ T \ m.$
 $\llbracket \text{wf-mdecl } wf_1 \ P \ C \ (M, b, Ts, T, m); P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } C \rrbracket$
 $\implies \text{wf-mdecl } wf_2 \ (\text{compP } f \ P) \ C \ (M, b, Ts, T, f \ b \ m);$
 $\forall x \in \text{set } P. \text{wf-cdecl } wf_1 \ P \ x; x \in \text{set } (\text{compP } f \ P); \text{wf-prog } p \ P \rrbracket$
 $\implies \text{wf-cdecl } wf_2 \ (\text{compP } f \ P) \ x$

lemma *wf-prog-compPI*:

assumes *lift*:

$\bigwedge C \ M \ b \ Ts \ T \ m.$
 $\llbracket P \vdash C \text{ sees } M, b: Ts \rightarrow T = m \text{ in } C; \text{wf-mdecl } wf_1 \ P \ C \ (M, b, Ts, T, m) \rrbracket$
 $\implies \text{wf-mdecl } wf_2 \ (\text{compP } f \ P) \ C \ (M, b, Ts, T, f \ b \ m)$

and *wf*: *wf-prog* wf₁ P

shows *wf-prog* wf₂ (compP f P)

end

theory *Hidden*

imports *List-Index.List-Index*

begin

definition *hidden* :: 'a list \Rightarrow nat \Rightarrow bool **where**

hidden xs i \equiv i < size xs \wedge xs!i \in set(drop (i+1) xs)

lemma *hidden-last-index*: $x \in \text{set } xs \implies \text{hidden } (xs @ [x]) (\text{last-index } xs \ x)$
by(*auto simp add: hidden-def nth-append rev-nth[symmetric]*
dest: last-index-less[OF - le-refl])

lemma *hidden-inacc*: $\text{hidden } xs \ i \implies \text{last-index } xs \ x \neq i$
by(*auto simp add: hidden-def last-index-drop last-index-less-size-conv*)

lemma [*simp*]: $\text{hidden } xs \ i \implies \text{hidden } (xs@[x]) \ i$
by(*auto simp add: hidden-def nth-append*)

lemma *fun-upds-apply*:
 $(m(xs[\mapsto]ys)) \ x =$
 $(\text{let } xs' = \text{take } (\text{size } ys) \ xs$
 $\text{in if } x \in \text{set } xs' \text{ then } \text{Some}(ys \ ! \ \text{last-index } xs' \ x) \ \text{else } m \ x)$
proof(*induct xs arbitrary: m ys*)
case Nil then show *?case* **by**(*simp add: Let-def*)
next
case Cons show *?case*
proof(*cases ys*)
case Nil
then show *?thesis* **by**(*simp add: Let-def*)
next
case Cons': Cons
then show *?thesis* **using** *Cons* **by**(*simp add: Let-def last-index-Cons*)
qed
qed

lemma *map-upds-apply-eq-Some*:
 $((m(xs[\mapsto]ys)) \ x = \text{Some } y) =$
 $(\text{let } xs' = \text{take } (\text{size } ys) \ xs$
 $\text{in if } x \in \text{set } xs' \text{ then } ys \ ! \ \text{last-index } xs' \ x = y \ \text{else } m \ x = \text{Some } y)$
by(*simp add: fun-upds-apply Let-def*)

lemma *map-upds-upd-conv-last-index*:
 $\llbracket x \in \text{set } xs; \text{size } xs \leq \text{size } ys \rrbracket$
 $\implies m(xs[\mapsto]ys, x \mapsto y) = m(xs[\mapsto]ys[\text{last-index } xs \ x := y])$
by(*rule ext*) (*simp add: fun-upds-apply eq-sym-conv Let-def*)

end

3.4 Compilation Stage 1

theory *Compiler1* **imports** *PCompiler J1 Hidden* **begin**

Replacing variable names by indices.

primrec *compE₁* $:: \text{vname list} \Rightarrow \text{expr} \Rightarrow \text{expr}_1$
and *compEs₁* $:: \text{vname list} \Rightarrow \text{expr list} \Rightarrow \text{expr}_1 \text{ list}$ **where**
compE₁ $Vs \ (\text{new } C) = \text{new } C$

```

| compE1 Vs (Cast C e) = Cast C (compE1 Vs e)
| compE1 Vs (Val v) = Val v
| compE1 Vs (e1 «bop» e2) = (compE1 Vs e1) «bop» (compE1 Vs e2)
| compE1 Vs (Var V) = Var(last-index Vs V)
| compE1 Vs (V:=e) = (last-index Vs V):= (compE1 Vs e)
| compE1 Vs (e.F{D}) = (compE1 Vs e).F{D}
| compE1 Vs (CsF{D}) = CsF{D}
| compE1 Vs (e1.F{D};:=e2) = (compE1 Vs e1).F{D} := (compE1 Vs e2)
| compE1 Vs (CsF{D};:=e2) = CsF{D} := (compE1 Vs e2)
| compE1 Vs (e.M(es)) = (compE1 Vs e).M(compEs1 Vs es)
| compE1 Vs (CsM(es)) = CsM(compEs1 Vs es)
| compE1 Vs {V:T; e} = {(size Vs):T; compE1 (Vs@[V]) e}
| compE1 Vs (e1;;e2) = (compE1 Vs e1);;(compE1 Vs e2)
| compE1 Vs (if (e) e1 else e2) = if (compE1 Vs e) (compE1 Vs e1) else (compE1 Vs e2)
| compE1 Vs (while (e) c) = while (compE1 Vs e) (compE1 Vs c)
| compE1 Vs (throw e) = throw (compE1 Vs e)
| compE1 Vs (try e1 catch(C V) e2) =
  try(compE1 Vs e1) catch(C (size Vs)) (compE1 (Vs@[V]) e2)
| compE1 Vs (INIT C (Cs,b) ← e) = INIT C (Cs,b) ← (compE1 Vs e)
| compE1 Vs (RI(C,e);Cs ← e') = RI(C,(compE1 Vs e));Cs ← (compE1 Vs e')

| compEs1 Vs [] = []
| compEs1 Vs (e#es) = compE1 Vs e # compEs1 Vs es

```

lemma [simp]: $compEs_1 Vs es = map (compE_1 Vs) es$
lemma [simp]: $\bigwedge Vs. sub-RI (compE_1 Vs e) = sub-RI e$
and [simp]: $\bigwedge Vs. sub-RIs (compEs_1 Vs es) = sub-RIs es$
proof(induct rule: sub-RI-sub-RIs-induct) **qed**(auto)

primrec $fin_1 :: expr \Rightarrow expr_1$ **where**
 $fin_1 (Val v) = Val v$
 $fin_1 (throw e) = throw(fin_1 e)$

lemma *comp-final*: $final e \Longrightarrow compE_1 Vs e = fin_1 e$

lemma [simp]:
 $\bigwedge Vs. max-vars (compE_1 Vs e) = max-vars e$
and $\bigwedge Vs. max-varss (compEs_1 Vs es) = max-varss es$

Compiling programs:

definition $compP_1 :: J-prog \Rightarrow J_1-prog$
where

$compP_1 \equiv compP (\lambda b (pns,body). compE_1 (case b of NonStatic \Rightarrow this\#pns \mid Static \Rightarrow pns) body)$

end

3.5 Correctness of Stage 1

theory *Correctness1*
imports *J1WellForm Compiler1*
begin

3.5.1 Correctness of program compilation

primrec $unmod :: expr_1 \Rightarrow nat \Rightarrow bool$

and $unmods :: expr_1 list \Rightarrow nat \Rightarrow bool$ **where**

$unmod (new C) i = True$ |
 $unmod (Cast C e) i = unmod e i$ |
 $unmod (Val v) i = True$ |
 $unmod (e_1 \ll bop \gg e_2) i = (unmod e_1 i \wedge unmod e_2 i)$ |
 $unmod (Var i) j = True$ |
 $unmod (i:=e) j = (i \neq j \wedge unmod e j)$ |
 $unmod (e \cdot F\{D\}) i = unmod e i$ |
 $unmod (C \cdot_s F\{D\}) i = True$ |
 $unmod (e_1 \cdot F\{D\} := e_2) i = (unmod e_1 i \wedge unmod e_2 i)$ |
 $unmod (C \cdot_s F\{D\} := e_2) i = unmod e_2 i$ |
 $unmod (e \cdot M(es)) i = (unmod e i \wedge unmods es i)$ |
 $unmod (C \cdot_s M(es)) i = unmods es i$ |
 $unmod \{j:T; e\} i = unmod e i$ |
 $unmod (e_1;;e_2) i = (unmod e_1 i \wedge unmod e_2 i)$ |
 $unmod (if (e) e_1 else e_2) i = (unmod e i \wedge unmod e_1 i \wedge unmod e_2 i)$ |
 $unmod (while (e) c) i = (unmod e i \wedge unmod c i)$ |
 $unmod (throw e) i = unmod e i$ |
 $unmod (try e_1 catch (C i) e_2) j = (unmod e_1 j \wedge (if i=j then False else unmod e_2 j))$ |
 $unmod (INIT C (Cs,b) \leftarrow e) i = unmod e i$ |
 $unmod (RI(C,e);Cs \leftarrow e') i = (unmod e i \wedge unmod e' i)$ |

$unmods ([]) i = True$ |

$unmods (e\#es) i = (unmod e i \wedge unmods es i)$

lemma $hidden-unmod: \bigwedge Vs. hidden Vs i \implies unmod (compE_1 Vs e) i$ **and**
 $\bigwedge Vs. hidden Vs i \implies unmods (compEs_1 Vs es) i$

lemma $eval_1-preserves-unmod:$

$\llbracket P \vdash_1 \langle e, (h, ls, sh) \rangle \Rightarrow \langle e', (h', ls', sh') \rangle; unmod e i; i < size ls \rrbracket$
 $\implies ls ! i = ls' ! i$

and $\llbracket P \vdash_1 \langle es, (h, ls, sh) \rangle [\Rightarrow] \langle es', (h', ls', sh') \rangle; unmods es i; i < size ls \rrbracket$
 $\implies ls ! i = ls' ! i$

lemma $LAss-lem:$

$\llbracket x \in set xs; size xs \leq size ys \rrbracket$

$\implies m_1 \subseteq_m m_2(xs[\mapsto]ys) \implies m_1(x \mapsto y) \subseteq_m m_2(xs[\mapsto]ys[last-index xs x := y])$ **lemma** $Block-lem:$

fixes $l :: 'a \rightarrow 'b$

assumes $0: l \subseteq_m [Vs [\mapsto] ls]$

and $1: l' \subseteq_m [Vs [\mapsto] ls', V \mapsto v]$

and $hidden: V \in set Vs \implies ls ! last-index Vs V = ls' ! last-index Vs V$

and $size: size ls = size ls' \quad size Vs < size ls'$

shows $l'(V := l V) \subseteq_m [Vs [\mapsto] ls']$

The main theorem:

theorem **assumes** $wf: wuf-J-prog P$

shows $eval_1-eval: P \vdash \langle e, (h, l, sh) \rangle \Rightarrow \langle e', (h', l', sh') \rangle$

$\implies (\bigwedge Vs ls. \llbracket fv e \subseteq set Vs; l \subseteq_m [Vs[\mapsto]ls]; size Vs + max-vars e \leq size ls \rrbracket$

$\implies \exists ls'. compP_1 P \vdash_1 \langle compE_1 Vs e, (h, ls, sh) \rangle \Rightarrow \langle fin_1 e', (h', ls', sh') \rangle \wedge l' \subseteq_m [Vs[\mapsto]ls']$)

and $evals_1-evals: P \vdash \langle es, (h, l, sh) \rangle [\Rightarrow] \langle es', (h', l', sh') \rangle$

$\implies (\bigwedge Vs ls. \llbracket fvs es \subseteq set Vs; l \subseteq_m [Vs[\mapsto]ls]; size Vs + max-varss es \leq size ls \rrbracket$

$\implies \exists ls'. compP_1 P \vdash_1 \langle compEs_1 Vs es, (h, ls, sh) \rangle [\Rightarrow] \langle compEs_1 Vs es', (h', ls', sh') \rangle \wedge$

$$l' \subseteq_m [Vs[\mapsto]ls')$$

3.5.2 Preservation of well-formedness

The compiler preserves well-formedness. Is less trivial than it may appear. We start with two simple properties: preservation of well-typedness

lemma *compE₁-pres-wt*: $\bigwedge Vs Ts U.$

$$\llbracket P, [Vs[\mapsto]Ts] \vdash e :: U; \text{size } Ts = \text{size } Vs \rrbracket$$

$$\implies \text{compP } f P, Ts \vdash_1 \text{compE}_1 Vs e :: U$$

and $\bigwedge Vs Ts Us.$

$$\llbracket P, [Vs[\mapsto]Ts] \vdash es [::] Us; \text{size } Ts = \text{size } Vs \rrbracket$$

$$\implies \text{compP } f P, Ts \vdash_1 \text{compEs}_1 Vs es [::] Us$$

and the correct block numbering:

lemma \mathcal{B} : $\bigwedge Vs n. \text{size } Vs = n \implies \mathcal{B} (\text{compE}_1 Vs e) n$

and $\mathcal{B}s$: $\bigwedge Vs n. \text{size } Vs = n \implies \mathcal{B}s (\text{compEs}_1 Vs es) n$

The main complication is preservation of definite assignment \mathcal{D} .

lemma *image-last-index*: $A \subseteq \text{set}(xs@[x]) \implies \text{last-index } (xs @ [x]) \text{ ' } A =$
(if $x \in A$ *then* $\text{insert } (\text{size } xs) (\text{last-index } xs \text{ ' } (A - \{x\}))$ *else* $\text{last-index } xs \text{ ' } A$ *)*

lemma *A-compE₁-None[simp]*:

$$\bigwedge Vs. \mathcal{A} e = \text{None} \implies \mathcal{A} (\text{compE}_1 Vs e) = \text{None}$$

and $\bigwedge Vs. \mathcal{A}s es = \text{None} \implies \mathcal{A}s (\text{compEs}_1 Vs es) = \text{None}$

lemma *A-compE₁*:

$$\bigwedge A Vs. \llbracket \mathcal{A} e = [A]; \text{fv } e \subseteq \text{set } Vs \rrbracket \implies \mathcal{A} (\text{compE}_1 Vs e) = \lfloor \text{last-index } Vs \text{ ' } A \rfloor$$

and $\bigwedge A Vs. \llbracket \mathcal{A}s es = [A]; \text{fvs } es \subseteq \text{set } Vs \rrbracket \implies \mathcal{A}s (\text{compEs}_1 Vs es) = \lfloor \text{last-index } Vs \text{ ' } A \rfloor$

lemma *D-None[iff]*: $\mathcal{D} (e::'a \text{ exp}) \text{ None}$ **and** *[iff]*: $\mathcal{D}s (es::'a \text{ exp list}) \text{ None}$

lemma *D-last-index-compE₁*:

$$\bigwedge A Vs. \llbracket A \subseteq \text{set } Vs; \text{fv } e \subseteq \text{set } Vs \rrbracket \implies$$

$$\mathcal{D} e [A] \implies \mathcal{D} (\text{compE}_1 Vs e) \lfloor \text{last-index } Vs \text{ ' } A \rfloor$$

and $\bigwedge A Vs. \llbracket A \subseteq \text{set } Vs; \text{fvs } es \subseteq \text{set } Vs \rrbracket \implies$

$$\mathcal{D}s es [A] \implies \mathcal{D}s (\text{compEs}_1 Vs es) \lfloor \text{last-index } Vs \text{ ' } A \rfloor$$

lemma *last-index-image-set*: $\text{distinct } xs \implies \text{last-index } xs \text{ ' set } xs = \{..<\text{size } xs\}$

lemma *D-compE₁*:

$$\llbracket \mathcal{D} e [set Vs]; \text{fv } e \subseteq \text{set } Vs; \text{distinct } Vs \rrbracket \implies \mathcal{D} (\text{compE}_1 Vs e) \lfloor \{..<\text{length } Vs\} \rfloor$$

lemma *D-compE₁'*:

assumes $\mathcal{D} e [set(V\#Vs)]$ **and** $\text{fv } e \subseteq \text{set}(V\#Vs)$ **and** $\text{distinct}(V\#Vs)$

shows $\mathcal{D} (\text{compE}_1 (V\#Vs) e) \lfloor \{..length Vs\} \rfloor$

lemma *compP₁-pres-wf*: $\text{wf-J-prog } P \implies \text{wf-J}_1\text{-prog } (\text{compP}_1 P)$

end

3.6 Compilation Stage 2

theory *Compiler2*

imports *PCompiler J1 ../JVM/JVMExec*

begin

lemma *bop-expr-length-aux* [*simp*]:

length (case bop of Eq ⇒ [CmpEq] | Add ⇒ [IAdd]) = Suc 0
by(*cases bop, simp+*)

primrec *compE₂* :: *expr₁ ⇒ instr list*

and *compEs₂* :: *expr₁ list ⇒ instr list* **where**

compE₂ (*new C*) = [*New C*]
| *compE₂* (*Cast C e*) = *compE₂* *e* @ [*Checkcast C*]
| *compE₂* (*Val v*) = [*Push v*]
| *compE₂* (*e₁ «bop» e₂*) = *compE₂* *e₁* @ *compE₂* *e₂* @
(*case bop of Eq ⇒ [CmpEq]*
| *Add ⇒ [IAdd]*)
| *compE₂* (*Var i*) = [*Load i*]
| *compE₂* (*i:=e*) = *compE₂* *e* @ [*Store i, Push Unit*]
| *compE₂* (*e·F{D}*) = *compE₂* *e* @ [*Getfield F D*]
| *compE₂* (*C·_sF{D}*) = [*Getstatic C F D*]
| *compE₂* (*e₁·F{D} := e₂*) =
compE₂ *e₁* @ *compE₂* *e₂* @ [*Putfield F D, Push Unit*]
| *compE₂* (*C·_sF{D} := e₂*) =
compE₂ *e₂* @ [*Putstatic C F D, Push Unit*]
| *compE₂* (*e·M(es)*) = *compE₂* *e* @ *compEs₂* *es* @ [*Invoke M (size es)*]
| *compE₂* (*C·_sM(es)*) = *compEs₂* *es* @ [*Invokestatic C M (size es)*]
| *compE₂* (*{i:T; e}*) = *compE₂* *e*
| *compE₂* (*e₁;;e₂*) = *compE₂* *e₁* @ [*Pop*] @ *compE₂* *e₂*
| *compE₂* (*if (e) e₁ else e₂*) =
(*let cnd = compE₂* *e*;
thn = compE₂ *e₁*;
els = compE₂ *e₂*;
test = IfFalse (int(size thn + 2));
thnex = Goto (int(size els + 1))
in cnd @ [test] @ thn @ [thnex] @ els)
| *compE₂* (*while (e) c*) =
(*let cnd = compE₂* *e*;
bdy = compE₂ *c*;
test = IfFalse (int(size bdy + 3));
loop = Goto (-int(size bdy + size cnd + 2))
in cnd @ [test] @ bdy @ [Pop] @ [loop] @ [Push Unit])
| *compE₂* (*throw e*) = *compE₂* *e* @ [*instr.Throw*]
| *compE₂* (*try e₁ catch(C i) e₂*) =
(*let catch = compE₂* *e₂*
in compE₂ *e₁* @ [*Goto (int(size catch)+2), Store i*] @ *catch*)
| *compE₂* (*INIT C (Cs,b) ← e*) = []
| *compE₂* (*RI(C,e);Cs ← e'*) = []

| *compEs₂* [] = []
| *compEs₂* (*e#es*) = *compE₂* *e* @ *compEs₂* *es*

Compilation of exception table. Is given start address of code to compute absolute addresses necessary in exception table.

primrec *compxE₂* :: *expr₁ ⇒ pc ⇒ nat ⇒ ex-table*

and *compxEs₂* :: *expr₁ list ⇒ pc ⇒ nat ⇒ ex-table* **where**

compxE₂ (*new C*) *pc d* = []

$$\begin{aligned}
& | \text{compxE}_2 (\text{Cast } C \ e) \ pc \ d = \text{compxE}_2 \ e \ pc \ d \\
& | \text{compxE}_2 (\text{Val } v) \ pc \ d = [] \\
& | \text{compxE}_2 (e_1 \ll\text{bop}\gg e_2) \ pc \ d = \\
& \quad \text{compxE}_2 \ e_1 \ pc \ d @ \text{compxE}_2 \ e_2 \ (pc + \text{size}(\text{compE}_2 \ e_1)) \ (d+1) \\
& | \text{compxE}_2 (\text{Var } i) \ pc \ d = [] \\
& | \text{compxE}_2 (i:=e) \ pc \ d = \text{compxE}_2 \ e \ pc \ d \\
& | \text{compxE}_2 (e \cdot F\{D\}) \ pc \ d = \text{compxE}_2 \ e \ pc \ d \\
& | \text{compxE}_2 (C \cdot_s F\{D\}) \ pc \ d = [] \\
& | \text{compxE}_2 (e_1 \cdot F\{D\} := e_2) \ pc \ d = \\
& \quad \text{compxE}_2 \ e_1 \ pc \ d @ \text{compxE}_2 \ e_2 \ (pc + \text{size}(\text{compE}_2 \ e_1)) \ (d+1) \\
& | \text{compxE}_2 (C \cdot_s F\{D\} := e_2) \ pc \ d = \text{compxE}_2 \ e_2 \ pc \ d \\
& | \text{compxE}_2 (e \cdot M(es)) \ pc \ d = \\
& \quad \text{compxE}_2 \ e \ pc \ d @ \text{compxEs}_2 \ es \ (pc + \text{size}(\text{compE}_2 \ e)) \ (d+1) \\
& | \text{compxE}_2 (C \cdot_s M(es)) \ pc \ d = \text{compxEs}_2 \ es \ pc \ d \\
& | \text{compxE}_2 (\{i:T; e\}) \ pc \ d = \text{compxE}_2 \ e \ pc \ d \\
& | \text{compxE}_2 (e_1;;e_2) \ pc \ d = \\
& \quad \text{compxE}_2 \ e_1 \ pc \ d @ \text{compxE}_2 \ e_2 \ (pc + \text{size}(\text{compE}_2 \ e_1) + 1) \ d \\
& | \text{compxE}_2 (\text{if } (e) \ e_1 \ \text{else } e_2) \ pc \ d = \\
& \quad (\text{let } pc_1 = pc + \text{size}(\text{compE}_2 \ e) + 1; \\
& \quad \quad pc_2 = pc_1 + \text{size}(\text{compE}_2 \ e_1) + 1 \\
& \quad \text{in } \text{compxE}_2 \ e \ pc \ d @ \text{compxE}_2 \ e_1 \ pc_1 \ d @ \text{compxE}_2 \ e_2 \ pc_2 \ d) \\
& | \text{compxE}_2 (\text{while } (b) \ e) \ pc \ d = \\
& \quad \text{compxE}_2 \ b \ pc \ d @ \text{compxE}_2 \ e \ (pc + \text{size}(\text{compE}_2 \ b) + 1) \ d \\
& | \text{compxE}_2 (\text{throw } e) \ pc \ d = \text{compxE}_2 \ e \ pc \ d \\
& | \text{compxE}_2 (\text{try } e_1 \ \text{catch}(C \ i) \ e_2) \ pc \ d = \\
& \quad (\text{let } pc_1 = pc + \text{size}(\text{compE}_2 \ e_1) \\
& \quad \text{in } \text{compxE}_2 \ e_1 \ pc \ d @ \text{compxE}_2 \ e_2 \ (pc_1 + 2) \ d @ [(pc, pc_1, C, pc_1 + 1, d)]) \\
& | \text{compxE}_2 (\text{INIT } C \ (Cs, b) \leftarrow e) \ pc \ d = [] \\
& | \text{compxE}_2 (\text{RI}(C, e); Cs \leftarrow e^\wedge) \ pc \ d = [] \\
& | \text{compxEs}_2 [] \ pc \ d = [] \\
& | \text{compxEs}_2 (e\#es) \ pc \ d = \text{compxE}_2 \ e \ pc \ d @ \text{compxEs}_2 \ es \ (pc + \text{size}(\text{compE}_2 \ e)) \ (d+1)
\end{aligned}$$

primrec *max-stack* :: *expr*₁ ⇒ *nat*

and *max-stacks* :: *expr*₁ *list* ⇒ *nat* **where**

max-stack (*new* *C*) = 1

| *max-stack* (*Cast* *C* *e*) = *max-stack* *e*

| *max-stack* (*Val* *v*) = 1

| *max-stack* (*e*₁ «*bop*» *e*₂) = *max* (*max-stack* *e*₁) (*max-stack* *e*₂) + 1

| *max-stack* (*Var* *i*) = 1

| *max-stack* (*i*:=*e*) = *max-stack* *e*

| *max-stack* (*e* · *F*{*D*}) = *max-stack* *e*

| *max-stack* (*C* ·_s *F*{*D*}) = 1

| *max-stack* (*e*₁ · *F*{*D*} := *e*₂) = *max* (*max-stack* *e*₁) (*max-stack* *e*₂) + 1

| *max-stack* (*C* ·_s *F*{*D*} := *e*₂) = *max-stack* *e*₂

| *max-stack* (*e* · *M*(*es*)) = *max* (*max-stack* *e*) (*max-stacks* *es*) + 1

| *max-stack* (*C* ·_s *M*(*es*)) = *max-stacks* *es* + 1

| *max-stack* ({*i*:*T*; *e*}) = *max-stack* *e*

| *max-stack* (*e*₁;;*e*₂) = *max* (*max-stack* *e*₁) (*max-stack* *e*₂)

| *max-stack* (*if* (*e*) *e*₁ *else* *e*₂) =
max (*max-stack* *e*) (*max* (*max-stack* *e*₁) (*max-stack* *e*₂))

| *max-stack* (*while* (*e*) *c*) = *max* (*max-stack* *e*) (*max-stack* *c*)

| *max-stack* (*throw* *e*) = *max-stack* *e*

| *max-stack* (*try* *e*₁ *catch*(*C* *i*) *e*₂) = *max* (*max-stack* *e*₁) (*max-stack* *e*₂)

| $\text{max-stacks } [] = 0$
| $\text{max-stacks } (e\#es) = \text{max } (\text{max-stack } e) (1 + \text{max-stacks } es)$

lemma $\text{max-stack1}'$: $\neg \text{sub-RI } e \implies 1 \leq \text{max-stack } e$

lemma $\text{compE}_2\text{-not-Nil}'$: $\neg \text{sub-RI } e \implies \text{compE}_2 e \neq []$

lemma $\text{compE}_2\text{-nRet}$: $\bigwedge i. i \in \text{set } (\text{compE}_2 e_1) \implies i \neq \text{Return}$

and $\bigwedge i. i \in \text{set } (\text{compEs}_2 es_1) \implies i \neq \text{Return}$

by($\text{induct rule: compE}_2.\text{induct compEs}_2.\text{induct}$, $\text{auto simp: nth-append split: bop.splits}$)

definition compMb_2 :: $\text{staticb} \Rightarrow \text{expr}_1 \Rightarrow \text{jvm-method}$

where

$\text{compMb}_2 \equiv \lambda b \text{ body.}$

$\text{let } \text{ins} = \text{compE}_2 \text{ body } @ [\text{Return}];$

$\text{xt} = \text{compxE}_2 \text{ body } 0 0$

$\text{in } (\text{max-stack body}, \text{max-vars body}, \text{ins}, \text{xt})$

definition compP_2 :: $J_1\text{-prog} \Rightarrow \text{jvm-prog}$

where

$\text{compP}_2 \equiv \text{compP } \text{compMb}_2$

lemma compMb_2 [simp]:

$\text{compMb}_2 b e = (\text{max-stack } e, \text{max-vars } e,$
 $\text{compE}_2 e @ [\text{Return}], \text{compxE}_2 e 0 0)$

end

3.7 Correctness of Stage 2

theory Correctness2

imports $\text{HOL-Library.Sublist Compiler2 J1WellForm ../J/EConform}$

begin

3.7.1 Instruction sequences

How to select individual instructions and subsequences of instructions from a program given the class, method and program counter.

definition before :: $\text{jvm-prog} \Rightarrow \text{cname} \Rightarrow \text{mname} \Rightarrow \text{nat} \Rightarrow \text{instr list} \Rightarrow \text{bool}$

$\langle \langle (-, -, -, / \triangleright -) \rangle [51, 0, 0, 0, 51] 50 \rangle$ **where**

$P, C, M, pc \triangleright is \longleftrightarrow \text{prefix } is (\text{drop } pc (\text{instrs-of } P C M))$

definition at :: $\text{jvm-prog} \Rightarrow \text{cname} \Rightarrow \text{mname} \Rightarrow \text{nat} \Rightarrow \text{instr} \Rightarrow \text{bool}$

$\langle \langle (-, -, -, / \triangleright -) \rangle [51, 0, 0, 0, 51] 50 \rangle$ **where**

$P, C, M, pc \triangleright i \longleftrightarrow (\exists is. \text{drop } pc (\text{instrs-of } P C M) = i\#is)$

lemma [simp]: $P, C, M, pc \triangleright []$

lemma [simp]: $P, C, M, pc \triangleright (i\#is) = (P, C, M, pc \triangleright i \wedge P, C, M, pc + 1 \triangleright is)$

lemma [simp]: $P, C, M, pc \triangleright (is_1 @ is_2) = (P, C, M, pc \triangleright is_1 \wedge P, C, M, pc + \text{size } is_1 \triangleright is_2)$

lemma [simp]: $P, C, M, pc \triangleright i \implies \text{instrs-of } P C M ! pc = i$

lemma *beforeM*:

$$P \vdash C \text{ sees } M, b: Ts \rightarrow T = \text{body in } D \implies \\ \text{comp}P_2 P, D, M, 0 \triangleright \text{comp}E_2 \text{ body } @ [\text{Return}]$$

This lemma executes a single instruction by rewriting:

lemma [*simp*]:

$$P, C, M, pc \triangleright \text{instr} \implies \\ (P \vdash (\text{None}, h, (vs, ls, C, M, pc, ics) \# \text{frs}, sh) -jvm \rightarrow \sigma') = \\ ((\text{None}, h, (vs, ls, C, M, pc, ics) \# \text{frs}, sh) = \sigma' \vee \\ (\exists \sigma. \text{exec}(P, (\text{None}, h, (vs, ls, C, M, pc, ics) \# \text{frs}, sh)) = \text{Some } \sigma \wedge P \vdash \sigma -jvm \rightarrow \sigma'))$$

3.7.2 Exception tables

definition *pcs* :: *ex-table* \Rightarrow *nat set*

where

$$\text{pcs } xt \equiv \bigcup (f, t, C, h, d) \in \text{set } xt. \{f \dots < t\}$$

lemma *pcs-subset*:

$$\text{shows } (\bigwedge pc \ d. \text{pcs}(\text{comp}xE_2 \ e \ pc \ d) \subseteq \{pc \dots < pc + \text{size}(\text{comp}E_2 \ e)\}) \\ \text{and } (\bigwedge pc \ d. \text{pcs}(\text{comp}xEs_2 \ es \ pc \ d) \subseteq \{pc \dots < pc + \text{size}(\text{comp}Es_2 \ es)\})$$

lemma [*simp*]: $\text{pcs } [] = \{\}$

lemma [*simp*]: $\text{pcs } (x \# xt) = \{fst \ x \dots < fst(\text{snd } x)\} \cup \text{pcs } xt$

lemma [*simp*]: $\text{pcs}(xt_1 @ xt_2) = \text{pcs } xt_1 \cup \text{pcs } xt_2$

lemma [*simp*]: $pc < pc_0 \vee pc_0 + \text{size}(\text{comp}E_2 \ e) \leq pc \implies pc \notin \text{pcs}(\text{comp}xE_2 \ e \ pc_0 \ d)$

lemma [*simp*]: $pc < pc_0 \vee pc_0 + \text{size}(\text{comp}Es_2 \ es) \leq pc \implies pc \notin \text{pcs}(\text{comp}xEs_2 \ es \ pc_0 \ d)$

lemma [*simp*]: $pc_1 + \text{size}(\text{comp}E_2 \ e_1) \leq pc_2 \implies \text{pcs}(\text{comp}xE_2 \ e_1 \ pc_1 \ d_1) \cap \text{pcs}(\text{comp}xE_2 \ e_2 \ pc_2 \ d_2) = \{\}$

lemma [*simp*]: $pc_1 + \text{size}(\text{comp}E_2 \ e) \leq pc_2 \implies \text{pcs}(\text{comp}xE_2 \ e \ pc_1 \ d_1) \cap \text{pcs}(\text{comp}xEs_2 \ es \ pc_2 \ d_2) = \{\}$

lemma [*simp*]:

$$pc \notin \text{pcs } xt_0 \implies \text{match-ex-table } P \ C \ pc \ (xt_0 @ xt_1) = \text{match-ex-table } P \ C \ pc \ xt_1$$

lemma [*simp*]: $\llbracket x \in \text{set } xt; pc \notin \text{pcs } xt \rrbracket \implies \neg \text{matches-ex-entry } P \ D \ pc \ x$

lemma [*simp*]:

assumes *xe*: $xe \in \text{set}(\text{comp}xE_2 \ e \ pc \ d)$ **and** *outside*: $pc' < pc \vee pc + \text{size}(\text{comp}E_2 \ e) \leq pc'$
shows $\neg \text{matches-ex-entry } P \ C \ pc' \ xe$

lemma [*simp*]:

assumes *xe*: $xe \in \text{set}(\text{comp}xEs_2 \ es \ pc \ d)$ **and** *outside*: $pc' < pc \vee pc + \text{size}(\text{comp}Es_2 \ es) \leq pc'$
shows $\neg \text{matches-ex-entry } P \ C \ pc' \ xe$

lemma *match-ex-table-app*[*simp*]:

$$\forall xte \in \text{set } xt_1. \neg \text{matches-ex-entry } P \ D \ pc \ xte \implies \\ \text{match-ex-table } P \ D \ pc \ (xt_1 @ xt) = \text{match-ex-table } P \ D \ pc \ xt$$

lemma [simp]:

$\forall x \in \text{set } xtab. \neg \text{matches-ex-entry } P \ C \ pc \ x \implies$
 $\text{match-ex-table } P \ C \ pc \ xtab = \text{None}$

lemma match-ex-entry:

$\text{matches-ex-entry } P \ C \ pc \ (\text{start}, \text{end}, \text{catch-type}, \text{handler}) =$
 $(\text{start} \leq pc \wedge pc < \text{end} \wedge P \vdash C \preceq^* \text{catch-type})$

definition caught :: $\text{jvm-prog} \Rightarrow pc \Rightarrow \text{heap} \Rightarrow \text{addr} \Rightarrow \text{ex-table} \Rightarrow \text{bool}$ **where**

$\text{caught } P \ pc \ h \ a \ xt \longleftrightarrow$
 $(\exists \text{entry} \in \text{set } xt. \text{matches-ex-entry } P \ (\text{cname-of } h \ a) \ pc \ \text{entry})$

definition beforex :: $\text{jvm-prog} \Rightarrow \text{cname} \Rightarrow \text{mname} \Rightarrow \text{ex-table} \Rightarrow \text{nat set} \Rightarrow \text{nat} \Rightarrow \text{bool}$

$(\langle (2, -, / - \triangleright / - / / -, / -) \rangle [51, 0, 0, 0, 0, 51] \ 50)$ **where**
 $P, C, M \triangleright xt / I, d \longleftrightarrow$
 $(\exists xt_0 \ xt_1. \text{ex-table-of } P \ C \ M = xt_0 \ @ \ xt \ @ \ xt_1 \wedge \text{pcs } xt_0 \cap I = \{\} \wedge \text{pcs } xt \subseteq I \wedge$
 $(\forall pc \in I. \forall C \ pc' \ d'. \text{match-ex-table } P \ C \ pc \ xt_1 = \lfloor (pc', d') \rfloor \longrightarrow d' \leq d))$

definition dummyx :: $\text{jvm-prog} \Rightarrow \text{cname} \Rightarrow \text{mname} \Rightarrow \text{ex-table} \Rightarrow \text{nat set} \Rightarrow \text{nat} \Rightarrow \text{bool}$ $(\langle (2, -, - \triangleright / - / / -, -) \rangle [51, 0, 0, 0, 0, 51] \ 50)$ **where**

$P, C, M \triangleright xt / I, d \longleftrightarrow P, C, M \triangleright xt / I, d$

abbreviation

$\text{beforex}_0 \ P \ C \ M \ d \ I \ xt \ xt_0 \ xt_1$
 $\equiv \text{ex-table-of } P \ C \ M = xt_0 \ @ \ xt \ @ \ xt_1 \wedge \text{pcs } xt_0 \cap I = \{\}$
 $\wedge \text{pcs } xt \subseteq I \wedge (\forall pc \in I. \forall C \ pc' \ d'. \text{match-ex-table } P \ C \ pc \ xt_1 = \lfloor (pc', d') \rfloor \longrightarrow d' \leq d)$

lemma beforex-beforex₀-eq:

$P, C, M \triangleright xt / I, d \equiv \exists xt_0 \ xt_1. \text{beforex}_0 \ P \ C \ M \ d \ I \ xt \ xt_0 \ xt_1$
using beforex-def **by** auto

lemma beforexD1: $P, C, M \triangleright xt / I, d \implies \text{pcs } xt \subseteq I$

lemma beforex-mono: $\llbracket P, C, M \triangleright xt / I, d'; d' \leq d \rrbracket \implies P, C, M \triangleright xt / I, d$

lemma [simp]: $P, C, M \triangleright xt / I, d \implies P, C, M \triangleright xt / I, \text{Suc } d$

lemma beforex-append[simp]:

$\text{pcs } xt_1 \cap \text{pcs } xt_2 = \{\} \implies$
 $P, C, M \triangleright xt_1 \ @ \ xt_2 / I, d =$
 $(P, C, M \triangleright xt_1 / I - \text{pcs } xt_2, d \wedge P, C, M \triangleright xt_2 / I - \text{pcs } xt_1, d \wedge P, C, M \triangleright xt_1 @ xt_2 / I, d)$

lemma beforex-appendD1:

assumes $bx: P, C, M \triangleright xt_1 \ @ \ xt_2 \ @ \ [(f, t, D, h, d)] / I, d$
and $\text{pcs}: \text{pcs } xt_1 \subseteq J$ **and** $JI: J \subseteq I$ **and** $J\text{pcs}: J \cap \text{pcs } xt_2 = \{\}$
shows $P, C, M \triangleright xt_1 / J, d$

lemma beforex-appendD2:

assumes $bx: P, C, M \triangleright xt_1 \ @ \ xt_2 \ @ \ [(f, t, D, h, d)] / I, d$
and $\text{pcs}: \text{pcs } xt_2 \subseteq J$ **and** $JI: J \subseteq I$ **and** $J\text{pcs}: J \cap \text{pcs } xt_1 = \{\}$
shows $P, C, M \triangleright xt_2 / J, d$

lemma beforexM:

$P \vdash C \text{ sees } M, b: Ts \rightarrow T = \text{body in } D \implies \text{compP}_2 \ P, D, M \triangleright \text{compxE}_2 \ \text{body } 0 \ 0 / \{.. < \text{size}(\text{compE}_2)$

body}),0

lemma *match-ex-table-SomeD2*:

assumes *met*: *match-ex-table* $P D pc$ (*ex-table-of* $P C M$) = $\lfloor (pc',d') \rfloor$

and *bx*: $P, C, M \triangleright xt / I, d$

and *nmet*: $\forall x \in set\ xt. \neg matches\text{-}ex\text{-}entry\ P\ D\ pc\ x$ **and** *pcI*: $pc \in I$

shows $d' \leq d$

lemma *match-ex-table-SomeD1*:

$\llbracket match\text{-}ex\text{-}table\ P\ D\ pc\ (ex\text{-}table\text{-}of\ P\ C\ M) = \lfloor (pc',d') \rfloor;$

$P, C, M \triangleright xt / I, d; pc \in I; pc \notin pcs\ xt \rrbracket \implies d' \leq d$

3.7.3 The correctness proof

definition

handle :: *jvm-prog* \Rightarrow *cname* \Rightarrow *mname* \Rightarrow *addr* \Rightarrow *heap* \Rightarrow *val list* \Rightarrow *val list* \Rightarrow *nat* \Rightarrow *init-call-status*
 \Rightarrow *frame list* \Rightarrow *sheap*

\Rightarrow *jvm-state* **where**

handle $P C M a h vs ls pc ics frs sh = find\text{-}handler\ P\ a\ h\ ((vs, ls, C, M, pc, ics) \# frs)\ sh$

lemma *aux-isin[simp]*: $\llbracket B \subseteq A; a \in B \rrbracket \implies a \in A$

lemma *handle-frs-tl-neq*:

ics-of $f \neq No\text{-}ics$

$\implies (xp, h, f \# frs, sh) \neq handle\ P\ C\ M\ xa\ h' vs\ l\ pc\ ics\ frs\ sh'$

by (*simp* *add*: *handle-def* *find-handler-frs-tl-neq* *del*: *find-handler.simps*)

Correctness proof inductive hypothesis

fun *calling-to-called* :: *frame* \Rightarrow *frame* **where**

calling-to-called $(stk, loc, D, M, pc, ics) = (stk, loc, D, M, pc, case\ ics\ of\ Calling\ C\ Cs \Rightarrow Called\ (C \# Cs))$

fun *calling-to-scalled* :: *frame* \Rightarrow *frame* **where**

calling-to-scalled $(stk, loc, D, M, pc, ics) = (stk, loc, D, M, pc, case\ ics\ of\ Calling\ C\ Cs \Rightarrow Called\ Cs)$

fun *calling-to-calling* :: *frame* \Rightarrow *cname* \Rightarrow *frame* **where**

calling-to-calling $(stk, loc, D, M, pc, ics) C' = (stk, loc, D, M, pc, case\ ics\ of\ Calling\ C\ Cs \Rightarrow Calling\ C' (C \# Cs))$

fun *calling-to-throwing* :: *frame* \Rightarrow *addr* \Rightarrow *frame* **where**

calling-to-throwing $(stk, loc, D, M, pc, ics) a = (stk, loc, D, M, pc, case\ ics\ of\ Calling\ C\ Cs \Rightarrow Throwing\ (C \# Cs)\ a)$

fun *calling-to-sthrowing* :: *frame* \Rightarrow *addr* \Rightarrow *frame* **where**

calling-to-sthrowing $(stk, loc, D, M, pc, ics) a = (stk, loc, D, M, pc, case\ ics\ of\ Calling\ C\ Cs \Rightarrow Throwing\ Cs\ a)$

— pieces of the correctness proof's inductive hypothesis, which depend on the expression being compiled)

fun *Jcc-cond* :: $J_1\text{-prog}$ \Rightarrow *ty list* \Rightarrow *cname* \Rightarrow *mname* \Rightarrow *val list* \Rightarrow *pc* \Rightarrow *init-call-status*

\Rightarrow *nat set* \Rightarrow *heap* \Rightarrow *sheap* \Rightarrow *expr*₁ \Rightarrow *bool* **where**

Jcc-cond $P E C M vs pc ics I h sh (INIT\ C_0\ (Cs, b) \leftarrow e')$

$$\begin{aligned}
&= ((\exists T. P, E, h, sh \vdash_1 \text{INIT } C_0 (Cs, b) \leftarrow e' : T) \wedge \text{unit} = e' \wedge \text{ics} = \text{No-ics}) \mid \\
&\text{Jcc-cond } P E C M \text{ vs } pc \text{ ics } I h sh (RI(C', e_0); Cs \leftarrow e') \\
&= (((e_0 = C' \cdot_s \text{clinit}(\square)) \wedge (\exists T. P, E, h, sh \vdash_1 RI(C', e_0); Cs \leftarrow e' : T)) \\
&\quad \vee ((\exists a. e_0 = \text{Throw } a) \wedge (\forall C \in \text{set}(C' \# Cs). \text{is-class } P C))) \\
&\quad \wedge \text{unit} = e' \wedge \text{ics} = \text{No-ics}) \mid \\
&\text{Jcc-cond } P E C M \text{ vs } pc \text{ ics } I h sh (C' \cdot_s M'(es)) \\
&= (\text{let } e = (C' \cdot_s M'(es)) \\
&\quad \text{in if } M' = \text{clinit} \wedge es = \square \text{ then } (\exists T. P, E, h, sh \vdash_1 e : T) \wedge (\exists Cs. \text{ics} = \text{Called } Cs) \\
&\quad \text{else } (\text{comp}P_2 P, C, M, pc \triangleright \text{comp}E_2 e \wedge \text{comp}P_2 P, C, M \triangleright \text{comp}x E_2 e \text{ pc } (\text{size } vs) / I, \text{size } vs) \\
&\quad \quad \wedge \{pc.. < pc + \text{size}(\text{comp}E_2 e)\} \subseteq I \wedge \neg \text{sub-RI } e \wedge \text{ics} = \text{No-ics}) \\
&)\mid \\
&\text{Jcc-cond } P E C M \text{ vs } pc \text{ ics } I h sh e \\
&= (\text{comp}P_2 P, C, M, pc \triangleright \text{comp}E_2 e \wedge \text{comp}P_2 P, C, M \triangleright \text{comp}x E_2 e \text{ pc } (\text{size } vs) / I, \text{size } vs) \\
&\quad \wedge \{pc.. < pc + \text{size}(\text{comp}E_2 e)\} \subseteq I \wedge \neg \text{sub-RI } e \wedge \text{ics} = \text{No-ics})
\end{aligned}$$

fun *Jcc-frames* :: *jvm-prog* \Rightarrow *cname* \Rightarrow *mname* \Rightarrow *val list* \Rightarrow *val list* \Rightarrow *pc* \Rightarrow *init-call-status*
 \Rightarrow *frame list* \Rightarrow *expr₁* \Rightarrow *frame list* **where**

$$\begin{aligned}
&\text{Jcc-frames } P C M \text{ vs } ls \text{ pc ics frs } (\text{INIT } C_0 (C' \# Cs, b) \leftarrow e') \\
&= (\text{case } b \text{ of } \text{False} \Rightarrow (vs, ls, C, M, pc, \text{Calling } C' Cs) \# \text{frs} \\
&\quad \mid \text{True} \Rightarrow (vs, ls, C, M, pc, \text{Called } (C' \# Cs)) \# \text{frs} \\
&)\mid
\end{aligned}$$

$$\begin{aligned}
&\text{Jcc-frames } P C M \text{ vs } ls \text{ pc ics frs } (\text{INIT } C_0 (\text{Nil}, b) \leftarrow e') \\
&= (vs, ls, C, M, pc, \text{Called } \square) \# \text{frs} \mid
\end{aligned}$$

$$\begin{aligned}
&\text{Jcc-frames } P C M \text{ vs } ls \text{ pc ics frs } (RI(C', e_0); Cs \leftarrow e') \\
&= (\text{case } e_0 \text{ of } \text{Throw } a \Rightarrow (vs, ls, C, M, pc, \text{Throwing } (C' \# Cs) a) \# \text{frs} \\
&\quad \mid - \Rightarrow (vs, ls, C, M, pc, \text{Called } (C' \# Cs)) \# \text{frs}) \mid
\end{aligned}$$

$$\begin{aligned}
&\text{Jcc-frames } P C M \text{ vs } ls \text{ pc ics frs } (C' \cdot_s M'(es)) \\
&= (\text{if } M' = \text{clinit} \wedge es = \square \\
&\quad \text{then } \text{create-init-frame } P C' \# (vs, ls, C, M, pc, \text{ics}) \# \text{frs} \\
&\quad \text{else } (vs, ls, C, M, pc, \text{ics}) \# \text{frs} \\
&)\mid
\end{aligned}$$

$$\begin{aligned}
&\text{Jcc-frames } P C M \text{ vs } ls \text{ pc ics frs } e \\
&= (vs, ls, C, M, pc, \text{ics}) \# \text{frs}
\end{aligned}$$

fun *Jcc-rhs* :: *J₁-prog* \Rightarrow *ty list* \Rightarrow *cname* \Rightarrow *mname* \Rightarrow *val list* \Rightarrow *val list* \Rightarrow *pc* \Rightarrow *init-call-status*
 \Rightarrow *frame list* \Rightarrow *heap* \Rightarrow *val list* \Rightarrow *sheap* \Rightarrow *val* \Rightarrow *expr₁* \Rightarrow *jvm-state* **where**

$$\begin{aligned}
&\text{Jcc-rhs } P E C M \text{ vs } ls \text{ pc ics frs } h' ls' sh' v (\text{INIT } C_0 (Cs, b) \leftarrow e') \\
&= (\text{None}, h', (vs, ls, C, M, pc, \text{Called } \square) \# \text{frs}, sh') \mid
\end{aligned}$$

$$\begin{aligned}
&\text{Jcc-rhs } P E C M \text{ vs } ls \text{ pc ics frs } h' ls' sh' v (RI(C', e_0); Cs \leftarrow e') \\
&= (\text{None}, h', (vs, ls, C, M, pc, \text{Called } \square) \# \text{frs}, sh') \mid
\end{aligned}$$

$$\begin{aligned}
&\text{Jcc-rhs } P E C M \text{ vs } ls \text{ pc ics frs } h' ls' sh' v (C' \cdot_s M'(es)) \\
&= (\text{let } e = (C' \cdot_s M'(es)) \\
&\quad \text{in if } M' = \text{clinit} \wedge es = \square \\
&\quad \text{then } (\text{None}, h', (vs, ls, C, M, pc, \text{ics}) \# \text{frs}, sh' (C' \mapsto (\text{fst}(\text{the}(sh' C')), \text{Done}))) \\
&\quad \text{else } (\text{None}, h', (v \# vs, ls', C, M, pc + \text{size}(\text{comp}E_2 e), \text{ics}) \# \text{frs}, sh') \\
&)\mid
\end{aligned}$$

$$\begin{aligned}
&\text{Jcc-rhs } P E C M \text{ vs } ls \text{ pc ics frs } h' ls' sh' v e \\
&= (\text{None}, h', (v \# vs, ls', C, M, pc + \text{size}(\text{comp}E_2 e), \text{ics}) \# \text{frs}, sh')
\end{aligned}$$

fun *Jcc-err* :: *jvm-prog* \Rightarrow *cname* \Rightarrow *mname* \Rightarrow *heap* \Rightarrow *val list* \Rightarrow *val list* \Rightarrow *pc* \Rightarrow *init-call-status*
 \Rightarrow *frame list* \Rightarrow *sheap* \Rightarrow *nat set* \Rightarrow *heap* \Rightarrow *val list* \Rightarrow *sheap* \Rightarrow *addr* \Rightarrow *expr₁*
 \Rightarrow *bool* **where**

$$\text{Jcc-err } P C M h \text{ vs } ls \text{ pc ics frs } sh I h' ls' sh' xa (\text{INIT } C_0 (Cs, b) \leftarrow e')$$

$$\begin{aligned}
&= (\exists vs'. P \vdash (None, h, Jcc\text{-frames } P \ C \ M \ vs \ ls \ pc \ ics \ frs \ (INIT \ C_0 \ (Cs, b) \leftarrow e'), sh) \\
&\quad -jvm \rightarrow handle \ P \ C \ M \ xa \ h' \ (vs' @ vs) \ ls \ pc \ ics \ frs \ sh') \mid \\
&Jcc\text{-err } P \ C \ M \ h \ vs \ ls \ pc \ ics \ frs \ sh \ I \ h' \ ls' \ sh' \ xa \ (RI(C', e_0); Cs \leftarrow e') \\
&= (\exists vs'. P \vdash (None, h, Jcc\text{-frames } P \ C \ M \ vs \ ls \ pc \ ics \ frs \ (RI(C', e_0); Cs \leftarrow e'), sh) \\
&\quad -jvm \rightarrow handle \ P \ C \ M \ xa \ h' \ (vs' @ vs) \ ls \ pc \ ics \ frs \ sh') \mid \\
&Jcc\text{-err } P \ C \ M \ h \ vs \ ls \ pc \ ics \ frs \ sh \ I \ h' \ ls' \ sh' \ xa \ (C' \cdot_s M'(es)) \\
&= (let \ e = (C' \cdot_s M'(es)) \\
&\quad in \ if \ M' = clinit \wedge \ es = [] \\
&\quad \quad then \ case \ ics \ of \\
&\quad \quad \quad Called \ Cs \Rightarrow P \vdash (None, h, Jcc\text{-frames } P \ C \ M \ vs \ ls \ pc \ ics \ frs \ e, sh) \\
&\quad \quad \quad \quad -jvm \rightarrow (None, h', (vs, ls, C, M, pc, Throwing \ Cs \ xa) \# frs, (sh'(C' \mapsto (fst(the(sh' \\
&C')), Error)))) \\
&\quad \quad \quad else \ (\exists pc_1. \ pc \leq pc_1 \wedge pc_1 < pc + size(compE_2 \ e) \wedge \\
&\quad \quad \quad \quad \neg caught \ P \ pc_1 \ h' \ xa \ (compE_2 \ e \ pc \ (size \ vs)) \wedge \\
&\quad \quad \quad \quad (\exists vs'. P \vdash (None, h, Jcc\text{-frames } P \ C \ M \ vs \ ls \ pc \ ics \ frs \ e, sh) \\
&\quad \quad \quad \quad -jvm \rightarrow handle \ P \ C \ M \ xa \ h' \ (vs' @ vs) \ ls' \ pc_1 \ ics \ frs \ sh')) \\
&\quad \quad \quad) \mid \\
&Jcc\text{-err } P \ C \ M \ h \ vs \ ls \ pc \ ics \ frs \ sh \ I \ h' \ ls' \ sh' \ xa \ e \\
&= (\exists pc_1. \ pc \leq pc_1 \wedge pc_1 < pc + size(compE_2 \ e) \wedge \\
&\quad \neg caught \ P \ pc_1 \ h' \ xa \ (compE_2 \ e \ pc \ (size \ vs)) \wedge \\
&\quad (\exists vs'. P \vdash (None, h, Jcc\text{-frames } P \ C \ M \ vs \ ls \ pc \ ics \ frs \ e, sh) \\
&\quad \quad -jvm \rightarrow handle \ P \ C \ M \ xa \ h' \ (vs' @ vs) \ ls' \ pc_1 \ ics \ frs \ sh'))
\end{aligned}$$

fun *Jcc-pieces* :: $J_1\text{-prog} \Rightarrow ty \ list \Rightarrow cname \Rightarrow mname \Rightarrow heap \Rightarrow val \ list \Rightarrow val \ list \Rightarrow pc \Rightarrow$
init-call-status

$\Rightarrow frame \ list \Rightarrow sheap \Rightarrow nat \ set \Rightarrow heap \Rightarrow val \ list \Rightarrow sheap \Rightarrow val \Rightarrow addr \Rightarrow expr_1$
 $\Rightarrow bool \times frame \ list \times jvm\text{-state} \times bool$ **where**

Jcc-pieces $P \ E \ C \ M \ h \ vs \ ls \ pc \ ics \ frs \ sh \ I \ h' \ ls' \ sh' \ v \ xa \ e$
 $= (Jcc\text{-cond } P \ E \ C \ M \ vs \ pc \ ics \ I \ h \ sh \ e, Jcc\text{-frames } (compP_2 \ P) \ C \ M \ vs \ ls \ pc \ ics \ frs \ e,$
 $Jcc\text{-rhs } P \ E \ C \ M \ vs \ ls \ pc \ ics \ frs \ h' \ ls' \ sh' \ v \ e,$
 $Jcc\text{-err } (compP_2 \ P) \ C \ M \ h \ vs \ ls \ pc \ ics \ frs \ sh \ I \ h' \ ls' \ sh' \ xa \ e)$

— *Jcc-pieces* lemmas

lemma *nsub-RI-Jcc-pieces*:

assumes [*simp*]: $P \equiv compP_2 \ P_1$

and *nsub*: $\neg sub\text{-RI } e$

shows *Jcc-pieces* $P_1 \ E \ C \ M \ h \ vs \ ls \ pc \ ics \ frs \ sh \ I \ h' \ ls' \ sh' \ v \ xa \ e$

$= (let \ cond = P, C, M, pc \triangleright compE_2 \ e \wedge P, C, M \triangleright compE_2 \ e \ pc \ (size \ vs) / I, size \ vs$
 $\wedge \{pc.. < pc + size(compE_2 \ e)\} \subseteq I \wedge ics = No\text{-ics};$

$frs' = (vs, ls, C, M, pc, ics) \# frs;$

$rhs = (None, h', (v \# vs, ls', C, M, pc + size(compE_2 \ e), ics) \# frs, sh');$

$err = (\exists pc_1. \ pc \leq pc_1 \wedge pc_1 < pc + size(compE_2 \ e) \wedge$

$\neg caught \ P \ pc_1 \ h' \ xa \ (compE_2 \ e \ pc \ (size \ vs)) \wedge$

$(\exists vs'. P \vdash (None, h, frs', sh) -jvm \rightarrow handle \ P \ C \ M \ xa \ h' \ (vs' @ vs) \ ls' \ pc_1 \ ics \ frs \ sh'))$

$in \ (cond, frs', rhs, err)$

)

proof —

have *NC*: $\forall C'. e \neq C' \cdot_s clinit([])$ **using** *assms*(2) **proof**(*cases e*) **qed**(*simp-all*)

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

proof(*cases e*)

case (*S*Call *C M es*)

then have $M \neq clinit$ **using** *nsub* **by** *simp*

then show *?thesis* **using** *S*Call *nsub* **proof**(*cases es*) **qed**(*simp-all*)

qed(*simp-all*)
qed

lemma *Jcc-pieces-Cast*:

assumes [*simp*]: $P \equiv \text{comp}P_2 P_1$

and *Jcc-pieces* $P_1 E C M h_0 vs ls_0 pc ics frs sh_0 I h_1 ls_1 sh_1 v xa$ (*Cast* $C' e$)
= (*True*, frs_0 , $(xp', h', (v \# vs', ls', C_0, M', pc', ics') \# frs', sh')$, *err*)

shows *Jcc-pieces* $P_1 E C M h_0 vs ls_0 pc ics frs sh_0 I h_1 ls_1 sh_1 v' xa e$

= (*True*, frs_0 , $(xp', h', (v \# vs', ls', C_0, M', pc' - 1, ics') \# frs', sh')$,
($\exists pc_1. pc \leq pc_1 \wedge pc_1 < pc + \text{size}(\text{comp}E_2 e) \wedge$
 $\neg \text{caught } P pc_1 h_1 xa (\text{comp}xE_2 e pc (\text{size } vs)) \wedge$
($\exists vs'. P \vdash (\text{None}, h_0, frs_0, sh_0) -jvm \rightarrow \text{handle } P C M xa h_1 (vs' @ vs) ls_1 pc_1 ics frs sh_1$)))

proof –

have $pc: \{pc.. < pc + \text{length}(\text{comp}E_2 e)\} \subseteq I$ **using** *assms* **by** *clarsimp*

show *?thesis* **using** *assms* *nsub-RI-Jcc-pieces*[**where** $e=e$] pc **by** *clarsimp*

qed

lemma *Jcc-pieces-BinOp1*:

assumes

Jcc-pieces $P E C M h_0 vs ls_0 pc ics frs sh_0 I h_2 ls_2 sh_2 v xa$ ($e \llbracket \text{bop} \rrbracket e'$)

= (*True*, frs_0 , $(xp', h', (v \# vs', ls', C_0, M', pc', ics') \# frs', sh')$, *err*)

shows $\exists \text{err}. \text{Jcc-pieces } P E C M h_0 vs ls_0 pc ics frs sh_0$

($I - \text{pcs}(\text{comp}xE_2 e' (pc + \text{length}(\text{comp}E_2 e)) (\text{Suc}(\text{length } vs')))) h_1 ls_1 sh_1 v' xa e$

= (*True*, frs_0 , $(xp', h_1, (v \# vs', ls_1, C_0, M', pc' - \text{size}(\text{comp}E_2 e) - 1, ics') \# frs', sh_1)$, *err*)

proof –

have *bef*: $\text{comp}P \text{comp}Mb_2 P, C_0, M' \triangleright \text{comp}xE_2 e pc (\text{length } vs)$

/ $I - \text{pcs}(\text{comp}xE_2 e' (pc + \text{length}(\text{comp}E_2 e)) (\text{Suc}(\text{length } vs'))), \text{length } vs$

using *assms* **by** *clarsimp*

have $vs: vs = vs'$ **using** *assms* **by** *simp*

show *?thesis* **using** *assms* *nsub-RI-Jcc-pieces*[**where** $e=e$] *bef* vs **by** *clarsimp*

qed

lemma *Jcc-pieces-BinOp2*:

assumes [*simp*]: $P \equiv \text{comp}P_2 P_1$

and *Jcc-pieces* $P_1 E C M h_0 vs ls_0 pc ics frs sh_0 I h_2 ls_2 sh_2 v xa$ ($e \llbracket \text{bop} \rrbracket e'$)

= (*True*, frs_0 , $(xp', h', (v \# vs', ls', C_0, M', pc', ics') \# frs', sh')$, *err*)

shows $\exists \text{err}. \text{Jcc-pieces } P_1 E C M h_1 (v_1 \# vs) ls_1 (pc + \text{size}(\text{comp}E_2 e)) ics frs sh_1$

($I - \text{pcs}(\text{comp}xE_2 e pc (\text{length } vs')) h_2 ls_2 sh_2 v' xa e'$

= (*True*, $(v_1 \# vs, ls_1, C, M, pc + \text{size}(\text{comp}E_2 e), ics) \# frs$,

$(xp', h', (v \# v_1 \# vs', ls', C_0, M', pc' - 1, ics') \# frs', sh')$,

($\exists pc_1. pc + \text{size}(\text{comp}E_2 e) \leq pc_1 \wedge pc_1 < pc + \text{size}(\text{comp}E_2 e) + \text{length}(\text{comp}E_2 e') \wedge$

$\neg \text{caught } P pc_1 h_2 xa (\text{comp}xE_2 e' (pc + \text{size}(\text{comp}E_2 e)) (\text{Suc}(\text{length } vs))) \wedge$

($\exists vs'. P \vdash (\text{None}, h_1, (v_1 \# vs, ls_1, C, M, pc + \text{size}(\text{comp}E_2 e), ics) \# frs, sh_1)$

$-jvm \rightarrow \text{handle } P C M xa h_2 (vs' @ v_1 \# vs) ls_2 pc_1 ics frs sh_2$)))

proof –

have *bef*: $\text{comp}P \text{comp}Mb_2 P_1, C_0, M' \triangleright \text{comp}xE_2 e pc (\text{length } vs)$

/ $I - \text{pcs}(\text{comp}xE_2 e' (pc + \text{length}(\text{comp}E_2 e)) (\text{Suc}(\text{length } vs'))), \text{length } vs$

using *assms* **by** *clarsimp*

have $vs: vs = vs'$ **using** *assms* **by** *simp*

show *?thesis* **using** *assms* *nsub-RI-Jcc-pieces*[**where** $e=e'$] *bef* vs **by** *clarsimp*

qed

lemma *Jcc-pieces-FAcc*:

assumes

$Jcc\text{-pieces } P E C M h_0 vs ls_0 pc ics frs sh_0 I h_1 ls_1 sh_1 v xa (e \cdot F\{D\})$
 $= (True, frs_0, (xp', h', (v \# vs', ls', C_0, M', pc', ics') \# frs', sh'), err)$
shows $\exists err. Jcc\text{-pieces } P E C M h_0 vs ls_0 pc ics frs sh_0 I h_1 ls_1 sh_1 v' xa e$
 $= (True, frs_0, (xp', h', (v \# vs', ls', C_0, M', pc' - 1, ics') \# frs', sh'), err)$
proof –
have $pc: \{pc.. < pc + length(compE_2 e)\} \subseteq I$ **using** *assms* **by** *clarsimp*
then show *?thesis* **using** *assms nsub-RI-Jcc-pieces* [**where** $e=e$] **by** *clarsimp*
qed

lemma *Jcc-pieces-LAss*:

assumes [*simp*]: $P \equiv compP_2 P_1$
and $Jcc\text{-pieces } P_1 E C M h_0 vs ls_0 pc ics frs sh_0 I h_1 ls_1 sh_1 v xa (i:=e)$
 $= (True, frs_0, (xp', h', (v \# vs', ls', C_0, M', pc', ics') \# frs', sh'), err)$
shows $Jcc\text{-pieces } P_1 E C M h_0 vs ls_0 pc ics frs sh_0 I h_1 ls_1 sh_1 v' xa e$
 $= (True, frs_0, (xp', h', (v \# vs', ls', C_0, M', pc' - 2, ics') \# frs', sh'),$
 $(\exists pc_1. pc \leq pc_1 \wedge pc_1 < pc + size(compE_2 e) \wedge$
 $\neg caught P pc_1 h_1 xa (compxE_2 e pc (size vs)) \wedge$
 $(\exists vs'. P \vdash (None, h_0, frs_0, sh_0) -jvm \rightarrow handle P C M xa h_1 (vs' @ vs) ls_1 pc_1 ics frs sh_1)))$
proof –
have $pc: \{pc.. < pc + length(compE_2 e)\} \subseteq I$ **using** *assms* **by** *clarsimp*
show *?thesis* **using** *assms nsub-RI-Jcc-pieces* [**where** $e=e$] pc **by** *clarsimp*
qed

lemma *Jcc-pieces-FAss1*:

assumes
 $Jcc\text{-pieces } P E C M h_0 vs ls_0 pc ics frs sh_0 I h_2 ls_2 sh_2 v xa (e \cdot F\{D\} := e')$
 $= (True, frs_0, (xp', h', (v \# vs', ls', C_0, M', pc', ics') \# frs', sh'), err)$
shows $\exists err. Jcc\text{-pieces } P E C M h_0 vs ls_0 pc ics frs sh_0$
 $(I - pcs (compxE_2 e' (pc + length(compE_2 e)) (Suc (length vs')))) h_1 ls_1 sh_1 v' xa e$
 $= (True, frs_0, (xp', h_1, (v \# vs', ls_1, C_0, M', pc' - size(compE_2 e') - 2, ics') \# frs', sh_1), err)$
proof –
show *?thesis* **using** *assms nsub-RI-Jcc-pieces* [**where** $e=e$] **by** *clarsimp*
qed

lemma *Jcc-pieces-FAss2*:

assumes
 $Jcc\text{-pieces } P E C M h_0 vs ls_0 pc ics frs sh_0 I h_2 ls_2 sh_2 v xa (e \cdot F\{D\} := e')$
 $= (True, frs_0, (xp', h', (v \# vs', ls', C_0, M', pc', ics') \# frs', sh'), err)$
shows $Jcc\text{-pieces } P E C M h_1 (v_1 \# vs) ls_1 (pc + size(compE_2 e)) ics frs sh_1$
 $(I - pcs (compxE_2 e pc (length vs')) h_2 ls_2 sh_2 v' xa e'$
 $= (True, (v_1 \# vs, ls_1, C, M, pc + size(compE_2 e), ics) \# frs,$
 $(xp', h', (v \# v_1 \# vs', ls', C_0, M', pc' - 2, ics') \# frs', sh'),$
 $(\exists pc_1. (pc + size(compE_2 e)) \leq pc_1 \wedge pc_1 < pc + size(compE_2 e) + size(compE_2 e') \wedge$
 $\neg caught (compP_2 P) pc_1 h_2 xa (compxE_2 e' (pc + size(compE_2 e)) (size (v_1 \# vs))) \wedge$
 $(\exists vs'. (compP_2 P) \vdash (None, h_1, (v_1 \# vs, ls_1, C, M, pc + size(compE_2 e), ics) \# frs, sh_1)$
 $-jvm \rightarrow handle (compP_2 P) C M xa h_2 (vs' @ v_1 \# vs) ls_2 pc_1 ics frs sh_2)))$
proof –
show *?thesis* **using** *assms nsub-RI-Jcc-pieces* [**where** $e=e'$] **by** *clarsimp*
qed

lemma *Jcc-pieces-SFAss*:

assumes
 $Jcc\text{-pieces } P E C M h_0 vs ls_0 pc ics frs sh_0 I h' ls' sh' v xa (C' \cdot_s F\{D\} := e)$
 $= (True, frs_0, (xp', h', (v \# vs', ls', C_0, M', pc', ics') \# frs', sh'), err)$

shows $\exists err. Jcc\text{-pieces } P E C M h_0 vs ls_0 pc ics frs sh_0 I h_1 ls_1 sh_1 v' xa e$
 $= (True, frs_0, (xp', h_1, (v \# vs', ls_1, C_0, M', pc' - 2, ics') \# frs', sh_1), err)$

proof –

have $pc: \{pc..<pc + length (compE_2 e)\} \subseteq I$ **using** *assms* **by** *clarsimp*
show *?thesis* **using** *assms nsub-RI-Jcc-pieces* [**where** $e=e$] pc **by** *clarsimp*

qed

lemma *Jcc-pieces-Call1*:

assumes

Jcc-pieces $P E C M h_0 vs ls_0 pc ics frs sh_0 I h_3 ls_3 sh_3 v xa (e \cdot M_0(es))$
 $= (True, frs_0, (xp', h', (v \# vs', ls', C', M', pc', ics') \# frs', sh'), err)$

shows $\exists err. Jcc\text{-pieces } P E C M h_0 vs ls_0 pc ics frs sh_0$

$(I - pcs (compEs_2 es (pc + length (compE_2 e)) (Suc (length vs')))) h_1 ls_1 sh_1 v' xa e$
 $= (True, frs_0,$
 $(xp', h_1, (v \# vs', ls_1, C', M', pc' - size (compEs_2 es) - 1, ics') \# frs', sh_1), err)$

proof –

show *?thesis* **using** *assms nsub-RI-Jcc-pieces* [**where** $e=e$] **by** *clarsimp*

qed

lemma *Jcc-pieces-clinit*:

assumes [*simp*]: $P \equiv compP_2 P_1$

and *cond*: *Jcc-cond* $P_1 E C M vs pc ics I h sh (C1 \cdot_s clinit(\square))$

shows *Jcc-pieces* $P_1 E C M h vs ls pc ics frs sh I h' ls' sh' v xa (C1 \cdot_s clinit(\square))$

$= (True, create\text{-init}\text{-frame } P C1 \# (vs, ls, C, M, pc, ics) \# frs,$
 $(None, h', (vs, ls, C, M, pc, ics) \# frs, sh'(C1 \mapsto (fst(the(sh' C1)), Done))),$
 $P \vdash (None, h, create\text{-init}\text{-frame } P C1 \# (vs, ls, C, M, pc, ics) \# frs, sh) \text{--}jvm \rightarrow$
 $(case\ ics\ of\ Called\ Cs \Rightarrow (None, h', (vs, ls, C, M, pc, Throwing\ Cs\ xa) \# frs, (sh'(C1 \mapsto (fst(the(sh'$
 $C1)), Error))))))$

using *assms* **by** (*auto split: init-call-status.splits list.splits bool.splits*)

lemma *Jcc-pieces-SCall-clinit-body*:

assumes [*simp*]: $P \equiv compP_2 P_1$ **and** *wf*: *wf-J₁-prog* P_1

and *Jcc-pieces* $P_1 E C M h_0 vs ls_0 pc ics frs sh_0 I h_3 ls_2 sh_3 v xa (C1 \cdot_s clinit(\square))$
 $= (True, frs', rhs', err')$

and *method*: $P_1 \vdash C1$ *sees* *clinit, Static*: $\square \rightarrow Void = body$ *in* D

shows *Jcc-pieces* $P_1 \square D clinit h_2 \square (replicate (max\text{-vars } body) undefined) 0$

No-ics $(tl\ frs') sh_2 \{..<length (compE_2 body)\} h_3 ls_3 sh_3 v xa body$
 $= (True, frs',$
 $(None, h_3, ([v], ls_3, D, clinit, size (compE_2 body), No\text{-ics}) \# tl\ frs', sh_3),$
 $\exists pc_1. 0 \leq pc_1 \wedge pc_1 < size (compE_2 body) \wedge$
 $\neg caught\ P\ pc_1\ h_3\ xa\ (compE_2\ body\ 0\ 0) \wedge$
 $(\exists vs'. P \vdash (None, h_2, frs', sh_2) \text{--}jvm \rightarrow handle\ P\ D\ clinit\ xa\ h_3\ vs'\ ls_3\ pc_1$
 $No\text{-ics}\ (tl\ frs')\ sh_3))$

proof –

have *M-in-D*: $P_1 \vdash D$ *sees* *clinit, Static*: $\square \rightarrow Void = body$ *in* D

using *method* **by** (*rule sees-method-idemp*)

hence *M-code*: $compP_2 P_1, D, clinit, 0 \triangleright compE_2 body @ [Return]$

and *M-xtab*: $compP_2 P_1, D, clinit \triangleright compE_2 body\ 0\ 0 / \{..<size (compE_2 body)\}, 0$
by (*rule beforeM, rule beforexM*)

have *nsub*: $\neg sub\text{-RI } body$ **by** (*rule sees-wf₁-nsub-RI* [*OF wf method*])

then show *?thesis* **using** *assms nsub-RI-Jcc-pieces* *M-code* *M-xtab* **by** *clarsimp*

qed

lemma *Jcc-pieces-Cons*:

assumes $[simp]: P \equiv compP_2 P_1$
and $P, C, M, pc \triangleright compEs_2 (e\#es)$ **and** $P, C, M \triangleright compxEs_2 (e\#es) pc (size\ vs)/I, size\ vs$
and $\{pc..<pc+size(compEs_2 (e\#es))\} \subseteq I$
and $ics = No-ics$
and $\neg sub-RIs (e\#es)$
shows $Jcc-pieces\ P_1\ E\ C\ M\ h\ vs\ ls\ pc\ ics\ frs\ sh$
 $(I - pcs (compxEs_2\ es\ (pc + length (compE_2\ e)) (Suc (length\ vs))))\ h'\ ls'\ sh'\ v\ xa\ e$
 $= (True, (vs, ls, C, M, pc, ics) \# frs,$
 $(None, h', (v\#vs, ls', C, M, pc + length (compE_2\ e), ics) \# frs, sh'),$
 $\exists pc_1 \geq pc. pc_1 < pc + length (compE_2\ e) \wedge \neg caught\ P\ pc_1\ h'\ xa\ (compxE_2\ e\ pc\ (length\ vs))$
 $\wedge (\exists vs'. P \vdash (None, h, (vs, ls, C, M, pc, ics) \# frs, sh)$
 $\quad -jvm \rightarrow handle\ P\ C\ M\ xa\ h'\ (vs'@vs)\ ls'\ pc_1\ ics\ frs\ sh')$
proof –
show *?thesis* **using** *assms nsub-RI-Jcc-pieces* **[where** $e=e$ **]** **by** *auto*
qed

lemma *Jcc-pieces-InitNone*:

assumes $[simp]: P \equiv compP_2 P_1$
and $Jcc-pieces\ P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ sh\ I\ h'\ l'\ sh'\ v\ xa\ (INIT\ C'\ (C_0 \# Cs, False) \leftarrow e)$
 $= (True, frs', (None, h', (vs, l, C, M, pc, Called\ [])) \# frs, sh'), err)$
shows
 $Jcc-pieces\ P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ (sh(C_0 \mapsto (sblank\ P\ C_0, Prepared)))$
 $I\ h'\ l'\ sh'\ v\ xa\ (INIT\ C'\ (C_0 \# Cs, False) \leftarrow e)$
 $= (True, frs', (None, h', (vs, l, C, M, pc, Called\ [])) \# frs, sh'),$
 $\exists vs'. P \vdash (None, h, frs', (sh(C_0 \mapsto (sblank\ P_1\ C_0, Prepared))))$
 $\quad -jvm \rightarrow handle\ P\ C\ M\ xa\ h'\ (vs'@vs)\ l\ pc\ ics\ frs\ sh')$
proof –
have $Jcc-cond\ P_1\ E\ C\ M\ vs\ pc\ ics\ I\ h\ sh\ (INIT\ C'\ (C_0 \# Cs, False) \leftarrow e)$ **using** *assms* **by** *simp*
then obtain T **where** $P_1, E, h, sh \vdash_1 INIT\ C'\ (C_0 \# Cs, False) \leftarrow unit : T$ **by** *fastforce*
then have $P_1, E, h, sh(C_0 \mapsto (sblank\ P_1\ C_0, Prepared)) \vdash_1 INIT\ C'\ (C_0 \# Cs, False) \leftarrow unit : T$
by *(auto simp: fun-upd-apply)*
then have $Ex\ (WTrt2_1\ P_1\ E\ h\ (sh(C_0 \mapsto (sblank\ P_1\ C_0, Prepared)))\ (INIT\ C'\ (C_0 \# Cs, False)$
 $\leftarrow unit))$
by *(simp only: exI)*
then show *?thesis* **using** *assms* **by** *clarsimp*
qed

lemma *Jcc-pieces-InitDP*:

assumes $[simp]: P \equiv compP_2 P_1$
and $Jcc-pieces\ P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ sh\ I\ h'\ l'\ sh'\ v\ xa\ (INIT\ C'\ (C_0 \# Cs, False) \leftarrow e)$
 $= (True, frs', (None, h', (vs, l, C, M, pc, Called\ [])) \# frs, sh'), err)$
shows
 $Jcc-pieces\ P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ sh\ I\ h'\ l'\ sh'\ v\ xa\ (INIT\ C'\ (Cs, True) \leftarrow e)$
 $= (True, (calling-to-scalled\ (hd\ frs')) \# (tl\ frs'),$
 $(None, h', (vs, l, C, M, pc, Called\ [])) \# frs, sh'),$
 $\exists vs'. P \vdash (None, h, calling-to-scalled\ (hd\ frs') \# (tl\ frs'), sh)$
 $\quad -jvm \rightarrow handle\ P\ C\ M\ xa\ h'\ (vs'@vs)\ l\ pc\ ics\ frs\ sh')$
proof –
have $Jcc-cond\ P_1\ E\ C\ M\ vs\ pc\ ics\ I\ h\ sh\ (INIT\ C'\ (C_0 \# Cs, False) \leftarrow e)$ **using** *assms* **by** *simp*
then obtain T **where** $P_1, E, h, sh \vdash_1 INIT\ C'\ (C_0 \# Cs, False) \leftarrow unit : T$ **by** *fastforce*
then have $P_1, E, h, sh \vdash_1 INIT\ C'\ (Cs, True) \leftarrow unit : T$
by *(auto; metis list.sel(2) list.set-sel(2))*
then have $wtrt: Ex\ (WTrt2_1\ P_1\ E\ h\ sh\ (INIT\ C'\ (Cs, True) \leftarrow unit))$ **by** *(simp only: exI)*
show *?thesis* **using** *assms wtrt*

```

proof(cases Cs)
  case (Cons C1 Cs1)
  then show ?thesis using assms wrt
    by(case-tac method P C1 clinit) clarsimp
qed(clarsimp)
qed

```

lemma *Jcc-pieces-InitError*:

```

assumes [simp]: P  $\equiv$  compP2 P1
and Jcc-pieces P1 E C M h vs l pc ics frs sh I h' l' sh' v xa (INIT C' (C0 # Cs,False)  $\leftarrow$  e)
  = (True, frs', (None, h', (vs, l, C, M, pc, Called [])) # frs, sh'), err)
and err: sh C0 = Some(sfs,Error)
shows
  Jcc-pieces P1 E C M h vs l pc ics frs sh I h' l' sh' v xa (RI (C0, THROW NoClassDefFoundError);Cs
 $\leftarrow$  e)
  = (True, (calling-to-throwing (hd frs') (addr-of-sys-xcpt NoClassDefFoundError))#(tl frs'),
    (None, h', (vs, l, C, M, pc, Called [])) # frs, sh'),
     $\exists$  vs'. P  $\vdash$  (None,h, (calling-to-throwing (hd frs') (addr-of-sys-xcpt NoClassDefFoundError))#(tl frs'),sh)
    -jvm $\rightarrow$  handle P C M xa h' (vs'@vs) l pc ics frs sh')

```

proof –

```

show ?thesis using assms
proof(cases Cs)
  case (Cons C1 Cs1)
  then show ?thesis using assms
    by(case-tac method P C1 clinit, case-tac method P C0 clinit) clarsimp
qed(clarsimp)
qed

```

lemma *Jcc-pieces-InitObj*:

```

assumes [simp]: P  $\equiv$  compP2 P1
and Jcc-pieces P1 E C M h vs l pc ics frs sh I h' l' (sh(C0  $\mapsto$  (sfs,Processing))) v xa (INIT C' (C0 # Cs,False)  $\leftarrow$  e)
  = (True, frs', (None, h', (vs, l, C, M, pc, Called [])) # frs, sh'), err)
shows
  Jcc-pieces P1 E C M h vs l pc ics frs (sh(C0  $\mapsto$  (sfs,Processing))) I h' l' sh'' v xa (INIT C' (C0 # Cs,True)  $\leftarrow$  e)
  = (True, calling-to-called (hd frs')#(tl frs'),
    (None, h', (vs, l, C, M, pc, Called [])) # frs, sh''),
     $\exists$  vs'. P  $\vdash$  (None,h,calling-to-called (hd frs')#(tl frs'),sh')
    -jvm $\rightarrow$  handle P C M xa h' (vs'@vs) l pc ics frs sh'')

```

proof –

```

have Jcc-cond P1 E C M vs pc ics I h sh (INIT C' (C0 # Cs,False)  $\leftarrow$  e) using assms by simp
then obtain T where P1,E,h,sh  $\vdash_1$  INIT C' (C0 # Cs,False)  $\leftarrow$  unit : T by fastforce
then have P1,E,h,sh(C0  $\mapsto$  (sfs,Processing))  $\vdash_1$  INIT C' (C0 # Cs,True)  $\leftarrow$  unit : T
  using assms by clarsimp (auto simp: fun-upd-apply)
then have wrt: Ex (WTrt21 P1 E h (sh(C0  $\mapsto$  (sfs,Processing))) (INIT C' (C0 # Cs,True)  $\leftarrow$  unit))
  by(simp only: exI)
show ?thesis using assms wrt by clarsimp
qed

```

lemma *Jcc-pieces-InitNonObj*:

```

assumes [simp]: P  $\equiv$  compP2 P1

```

and *is-class* $P_1 D$ **and** $D \notin \text{set}(C_0 \# Cs)$ **and** $\forall C \in \text{set}(C_0 \# Cs). P_1 \vdash C \preceq^* D$
and *pcs*: *Jcc-pieces* $P_1 E C M h vs l pc ics frs sh I h' l' (sh(C_0 \mapsto (sfs, Processing))) v xa (INIT C' (C_0 \# Cs, False) \leftarrow e)$

$= (True, frs', (None, h', (vs, l, C, M, pc, Called [])) \# frs, sh'), err)$

shows

Jcc-pieces $P_1 E C M h vs l pc ics frs (sh(C_0 \mapsto (sfs, Processing))) I h' l' sh'' v xa (INIT C' (D \# C_0 \# Cs, False) \leftarrow e)$

$= (True, \text{calling-to-calling } (hd \text{ frs}') D \# (tl \text{ frs}'),$
 $(None, h', (vs, l, C, M, pc, Called [])) \# frs, sh'),$
 $\exists vs'. P \vdash (None, h, \text{calling-to-calling } (hd \text{ frs}') D \# (tl \text{ frs}'), sh')$
 $-jvm \rightarrow \text{handle } P C M xa h' (vs' @ vs) l pc ics frs sh'')$

proof –

have *Jcc-cond* $P_1 E C M vs pc ics I h sh (INIT C' (C_0 \# Cs, False) \leftarrow e)$ **using** *assms* **by** *simp*
then obtain T **where** $P_1, E, h, sh \vdash_1 INIT C' (C_0 \# Cs, False) \leftarrow unit : T$ **by** *fastforce*
then have $P_1, E, h, sh(C_0 \mapsto (sfs, Processing)) \vdash_1 INIT C' (D \# C_0 \# Cs, False) \leftarrow unit : T$
using *assms* **by** *clarsimp* (*auto simp: fun-upd-apply*)
then have *wtrt*: $Ex (WTrt2_1 P_1 E h (sh(C_0 \mapsto (sfs, Processing))) (INIT C' (D \# C_0 \# Cs, False) \leftarrow unit))$

by (*simp only: exI*)

show *?thesis* **using** *assms* **wtrt** **by** *clarsimp*

qed

lemma *Jcc-pieces-InitRInit*:

assumes [*simp*]: $P \equiv \text{comp} P_2 P_1$ **and** *wf*: *wf-J₁-prog* P_1

and *Jcc-pieces* $P_1 E C M h vs l pc ics frs sh I h' l' sh' v xa (INIT C' (C_0 \# Cs, True) \leftarrow e)$
 $= (True, frs', (None, h', (vs, l, C, M, pc, Called [])) \# frs, sh'), err)$

shows

Jcc-pieces $P_1 E C M h vs l pc ics frs sh I h' l' sh' v xa (RI (C_0, C_0 \cdot_s \text{clinit}([])); Cs \leftarrow e)$
 $= (True, frs',$
 $(None, h', (vs, l, C, M, pc, Called [])) \# frs, sh'),$
 $\exists vs'. P \vdash (None, h, frs', sh)$
 $-jvm \rightarrow \text{handle } P C M xa h' (vs' @ vs) l pc ics frs sh')$

proof –

have *cond*: *Jcc-cond* $P_1 E C M vs pc ics I h sh (INIT C' (C_0 \# Cs, True) \leftarrow e)$ **using** *assms* **by** *simp*

then have *clinit*: $\exists T. P_1, E, h, sh \vdash_1 C_0 \cdot_s \text{clinit}([]) : T$ **using** *wf*

by *clarsimp* (*auto simp: is-class-def intro: wf₁-types-clinit*)

then obtain T **where** $cT: P_1, E, h, sh \vdash_1 C_0 \cdot_s \text{clinit}([]) : T$ **by** *blast*

obtain T **where** $P_1, E, h, sh \vdash_1 INIT C' (C_0 \# Cs, True) \leftarrow unit : T$ **using** *cond* **by** *fastforce*

then have $P_1, E, h, sh \vdash_1 RI (C_0, C_0 \cdot_s \text{clinit}([])); Cs \leftarrow unit : T$

using *assms* **by** (*auto intro: cT*)

then have *wtrt*: $Ex (WTrt2_1 P_1 E h sh (RI (C_0, C_0 \cdot_s \text{clinit}([])); Cs \leftarrow unit))$

by (*simp only: exI*)

then show *?thesis* **using** *assms* **by** *simp*

qed

lemma *Jcc-pieces-RInit-clinit*:

assumes [*simp*]: $P \equiv \text{comp} P_2 P_1$ **and** *wf*: *wf-J₁-prog* P_1

and *Jcc-pieces* $P_1 E C M h vs l pc ics frs sh I h_1 l_1 sh_1 v xa (RI (C_0, C_0 \cdot_s \text{clinit}([])); Cs \leftarrow e)$
 $= (True, frs',$

$(None, h_1, (vs, l, C, M, pc, Called [])) \# frs, sh_1), err)$

shows

Jcc-pieces $P_1 E C M h vs l pc (Called Cs) (tl \text{ frs}') sh I h' l' sh' v xa (C_0 \cdot_s \text{clinit}([]))$
 $= (True, \text{create-init-frame } P C_0 \# (vs, l, C, M, pc, Called Cs) \# tl \text{ frs}',$

$(None, h', (vs, l, C, M, pc, Called\ Cs) \# tl\ frs', sh'(C_0 \mapsto (fst(the(sh' C_0)), Done e))),$
 $P \vdash (None, h, create-init-frame\ P\ C_0 \# (vs, l, C, M, pc, Called\ Cs) \# tl\ frs', sh)$
 $-jvm \rightarrow (None, h', (vs, l, C, M, pc, Throwing\ Cs\ xa) \# tl\ frs', sh'(C_0 \mapsto (fst(the(sh' C_0)), Error))))$
proof –
have *cond*: $Jcc-cond\ P_1\ E\ C\ M\ vs\ pc\ ics\ I\ h\ sh\ (RI\ (C_0, C_0.sclinit(\ [])); Cs \leftarrow e)$ **using** *assms* **by** *simp*
then have *wtrt*: $\exists T. P_1, E, h, sh \vdash_1 C_0.sclinit(\ []) : T$ **using** *wf*
by *clarsimp* (*auto simp: is-class-def intro: wf₁-types-clinit*)
then show *?thesis* **using** *assms* **by** *clarsimp*
qed

lemma *Jcc-pieces-RInit-Init*:

assumes [*simp*]: $P \equiv compP_2\ P_1$ **and** *wf*: $wf-J_1-prog\ P_1$

and *proc*: $\forall C' \in set\ Cs. \exists sfs. sh''\ C' = \lfloor (sfs, Processing) \rfloor$

and *Jcc-pieces* $P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ sh\ I\ h_1\ l_1\ sh_1\ v\ xa\ (RI\ (C_0, C_0.sclinit(\ [])); Cs \leftarrow e)$
 $= (True, frs',$
 $(None, h_1, (vs, l, C, M, pc, Called\ []) \# frs, sh_1), err)$

shows

$Jcc-pieces\ P_1\ E\ C\ M\ h'\ vs\ l\ pc\ ics\ frs\ sh''\ I\ h_1\ l_1\ sh_1\ v\ xa\ (INIT\ (last\ (C_0 \# Cs))\ (Cs, True) \leftarrow e)$
 $= (True, (vs, l, C, M, pc, Called\ Cs) \# frs,$
 $(None, h_1, (vs, l, C, M, pc, Called\ []) \# frs, sh_1),$
 $\exists vs'. P \vdash (None, h', (vs, l, C, M, pc, Called\ Cs) \# frs, sh'')$
 $-jvm \rightarrow handle\ P\ C\ M\ xa\ h_1\ (vs' @ vs)\ l\ pc\ ics\ frs\ sh_1)$

proof –

have *Jcc-cond* $P_1\ E\ C\ M\ vs\ pc\ ics\ I\ h\ sh\ (RI\ (C_0, C_0.sclinit(\ [])); Cs \leftarrow e)$ **using** *assms* **by** *simp*

then have *Ex* $(WTrt_2_1\ P_1\ E\ h\ sh\ (RI\ (C_0, C_0.sclinit(\ [])); Cs \leftarrow unit))$ **by** *simp*

then obtain *T* **where** *riwt*: $P_1, E, h, sh \vdash_1 RI\ (C_0, C_0.sclinit(\ [])); Cs \leftarrow unit : T$ **by** *meson*

then have $P_1, E, h', sh'' \vdash_1 INIT\ (last\ (C_0 \# Cs))\ (Cs, True) \leftarrow unit : T$ **using** *proc*

proof(*cases Cs*) **qed**(*auto*)

then have *wtrt*: $Ex\ (WTrt_2_1\ P_1\ E\ h'\ sh''\ (INIT\ (last\ (C_0 \# Cs))\ (Cs, True) \leftarrow unit))$ **by**(*simp only:*

exI)

show *?thesis* **using** *assms* *wtrt*

proof(*cases Cs*)

case (*Cons C1 Cs1*)

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

by(*case-tac method P C1 clinit*) *clarsimp*

qed(*clarsimp*)

qed

lemma *Jcc-pieces-RInit-RInit*:

assumes [*simp*]: $P \equiv compP_2\ P_1$

and *Jcc-pieces* $P_1\ E\ C\ M\ h\ vs\ l\ pc\ ics\ frs\ sh\ I\ h_1\ l_1\ sh_1\ v\ xa\ (RI\ (C_0, e); D \# Cs \leftarrow e')$
 $= (True, frs', rhs, err)$

and *hd*: $hd\ frs' = (vs1, l1, C1, M1, pc1, ics1)$

shows

$Jcc-pieces\ P_1\ E\ C\ M\ h'\ vs\ l\ pc\ ics\ frs\ sh''\ I\ h_1\ l_1\ sh_1\ v\ xa\ (RI\ (D, Throw\ xa) ; Cs \leftarrow e')$
 $= (True, (vs1, l1, C1, M1, pc1, Throwing\ (D \# Cs)\ xa) \# tl\ frs',$
 $(None, h_1, (vs, l, C, M, pc, Called\ []) \# frs, sh_1),$

$\exists vs'. P \vdash (None, h', (vs1, l1, C1, M1, pc1, Throwing\ (D \# Cs)\ xa) \# tl\ frs', sh'')$
 $-jvm \rightarrow handle\ P\ C\ M\ xa\ h_1\ (vs' @ vs)\ l\ pc\ ics\ frs\ sh_1)$

using *assms* **by**(*case-tac method P D clinit, cases e = C₀.sclinit(\ [])*) *clarsimp*+

JVM stepping lemmas

lemma *jvm-Invoke*:

assumes [*simp*]: $P \equiv \text{comp}P_2 P_1$
and $P, C, M, pc \triangleright \text{Invoke } M' \text{ (length } Ts)$
and $ha: h_2 a = [(Ca, fs)]$ **and** *method*: $P_1 \vdash Ca \text{ sees } M', \text{ NonStatic} : Ts \rightarrow T = \text{body in } D$
and *len*: $\text{length } pvs = \text{length } Ts$ **and** $ls_2' = \text{Addr } a \# pvs @ \text{replicate (max-vars body) undefined}$
shows $P \vdash (\text{None}, h_2, (\text{rev } pvs @ \text{Addr } a \# vs, ls_2, C, M, pc, \text{No-ics}) \# \text{frs}, sh_2) \text{ -jvm} \rightarrow$
 $(\text{None}, h_2, ([], ls_2', D, M', 0, \text{No-ics}) \# (\text{rev } pvs @ \text{Addr } a \# vs, ls_2, C, M, pc, \text{No-ics}) \# \text{frs}, sh_2)$
proof –
have *cname*: $\text{cname-of } h_2 (\text{the-Addr } ((\text{rev } pvs @ \text{Addr } a \# vs) ! \text{length } Ts)) = Ca$
using *ha method len* **by** (*auto simp: nth-append*)
have *r*: $(\text{rev } pvs @ \text{Addr } a \# vs) ! (\text{length } Ts) = \text{Addr } a$ **using** *len* **by** (*auto simp: nth-append*)
have *exm*: $\exists Ts T m D b. P \vdash Ca \text{ sees } M', b: Ts \rightarrow T = m \text{ in } D$
using *sees-method-compP[OF method]* **by** *fastforce*
show *?thesis* **using** *assms cname r exm* **by** *simp*
qed

lemma *jvm-Invokestatic*:

assumes [*simp*]: $P \equiv \text{comp}P_2 P_1$
and $P, C, M, pc \triangleright \text{Invokestatic } C' M' \text{ (length } Ts)$
and *sh*: $sh_2 D = \text{Some}(sfs, \text{Done})$
and *method*: $P_1 \vdash C' \text{ sees } M', \text{ Static} : Ts \rightarrow T = \text{body in } D$
and *len*: $\text{length } pvs = \text{length } Ts$ **and** $ls_2' = pvs @ \text{replicate (max-vars body) undefined}$
shows $P \vdash (\text{None}, h_2, (\text{rev } pvs @ vs, ls_2, C, M, pc, \text{No-ics}) \# \text{frs}, sh_2) \text{ -jvm} \rightarrow$
 $(\text{None}, h_2, ([], ls_2', D, M', 0, \text{No-ics}) \# (\text{rev } pvs @ vs, ls_2, C, M, pc, \text{No-ics}) \# \text{frs}, sh_2)$
proof –
have *exm*: $\exists Ts T m D b. P \vdash C' \text{ sees } M', b: Ts \rightarrow T = m \text{ in } D$
using *sees-method-compP[OF method]* **by** *fastforce*
show *?thesis* **using** *assms exm* **by** *simp*
qed

lemma *jvm-Invokestatic-Called*:

assumes [*simp*]: $P \equiv \text{comp}P_2 P_1$
and $P, C, M, pc \triangleright \text{Invokestatic } C' M' \text{ (length } Ts)$
and *sh*: $sh_2 D = \text{Some}(sfs, i)$
and *method*: $P_1 \vdash C' \text{ sees } M', \text{ Static} : Ts \rightarrow T = \text{body in } D$
and *len*: $\text{length } pvs = \text{length } Ts$ **and** $ls_2' = pvs @ \text{replicate (max-vars body) undefined}$
shows $P \vdash (\text{None}, h_2, (\text{rev } pvs @ vs, ls_2, C, M, pc, \text{Called } []) \# \text{frs}, sh_2) \text{ -jvm} \rightarrow$
 $(\text{None}, h_2, ([], ls_2', D, M', 0, \text{No-ics}) \# (\text{rev } pvs @ vs, ls_2, C, M, pc, \text{No-ics}) \# \text{frs}, sh_2)$
proof –
have *exm*: $\exists Ts T m D b. P \vdash C' \text{ sees } M', b: Ts \rightarrow T = m \text{ in } D$
using *sees-method-compP[OF method]* **by** *fastforce*
show *?thesis* **using** *assms exm* **by** *simp*
qed

lemma *jvm-Return-Init*:

$P, D, \text{clinit}, 0 \triangleright \text{comp}E_2 \text{ body } @ [\text{Return}]$
 $\implies P \vdash (\text{None}, h, (vs, ls, D, \text{clinit}, \text{size}(\text{comp}E_2 \text{ body}), \text{No-ics}) \# \text{frs}, sh)$
 $\text{ -jvm} \rightarrow (\text{None}, h, \text{frs}, sh(D \mapsto (\text{fst}(\text{the}(sh D))), \text{Done}))$
(is ?P $\implies P \vdash ?s1 \text{ -jvm} \rightarrow ?s2$)
proof –
assume *?P*
then **have** *exec* $(P, ?s1) = [?s2]$ **by** (*cases frs*) *auto*

then have $(?s1, ?s2) \in (exec-1 P)^*$
by(rule *exec-1I*[*THEN r-into-rtrancl*])
then show *?thesis* **by**(*simp add: exec-all-def1*)
qed

lemma *jvm-InitNone*:

\llbracket *ics-of f = Calling C Cs*;
sh C = None \rrbracket
 $\implies P \vdash (None, h, f \# frs, sh) -jvm \rightarrow (None, h, f \# frs, sh(C \mapsto (sblank P C, Prepared)))$
(*is* $\llbracket ?P; ?Q \rrbracket \implies P \vdash ?s1 -jvm \rightarrow ?s2$)

proof –

assume *assms: ?P ?Q*
then obtain *stk1 loc1 C1 M1 pc1 ics1* **where** $f = (stk1, loc1, C1, M1, pc1, ics1)$
by(*cases f*) *simp*
then have $exec(P, ?s1) = \llbracket ?s2 \rrbracket$ **using** *assms*
by(*case-tac ics1*) *simp-all*
then have $(?s1, ?s2) \in (exec-1 P)^*$
by(rule *exec-1I*[*THEN r-into-rtrancl*])
then show *?thesis* **by**(*simp add: exec-all-def1*)
qed

lemma *jvm-InitDP*:

\llbracket *ics-of f = Calling C Cs*;
sh C = \llbracket (sfs, i) \rrbracket; i = Done \vee i = Processing \rrbracket
 $\implies P \vdash (None, h, f \# frs, sh) -jvm \rightarrow (None, h, (calling-to-scalled f) \# frs, sh)$
(*is* $\llbracket ?P; ?Q; ?R \rrbracket \implies P \vdash ?s1 -jvm \rightarrow ?s2$)

proof –

assume *assms: ?P ?Q ?R*
then obtain *stk1 loc1 C1 M1 pc1 ics1* **where** $f = (stk1, loc1, C1, M1, pc1, ics1)$
by(*cases f*) *simp*
then have $exec(P, ?s1) = \llbracket ?s2 \rrbracket$ **using** *assms*
by(*case-tac i*) *simp-all*
then have $(?s1, ?s2) \in (exec-1 P)^*$
by(rule *exec-1I*[*THEN r-into-rtrancl*])
then show *?thesis* **by**(*simp add: exec-all-def1*)
qed

lemma *jvm-InitError*:

sh C = \llbracket (sfs, Error) \rrbracket
 $\implies P \vdash (None, h, (vs, ls, C_0, M, pc, Calling C Cs) \# frs, sh)$
 $-jvm \rightarrow (None, h, (vs, ls, C_0, M, pc, Throwing Cs (addr-of-sys-xcpt NoClassDefFoundError)) \# frs, sh)$
by(*clarsimp simp: exec-all-def1 intro!: r-into-rtrancl exec-1I*)

lemma *exec-ErrorThrowing*:

sh C = \llbracket (sfs, Error) \rrbracket
 $\implies exec(P, (None, h, calling-to-throwing (stk, loc, D, M, pc, Calling C Cs) a \# frs, sh))$
 $= Some (None, h, calling-to-throwing (stk, loc, D, M, pc, Calling C Cs) a \# frs, sh)$
by(*clarsimp simp: exec-all-def1 fun-upd-idem-iff intro!: r-into-rtrancl exec-1I*)

lemma *jvm-InitObj*:

\llbracket *sh C = Some(sfs, Prepared)*;
C = Object;
sh' = sh(C \mapsto (sfs, Processing)) \rrbracket
 $\implies P \vdash (None, h, (vs, ls, C_0, M, pc, Calling C Cs) \# frs, sh) -jvm \rightarrow$

(None, h, (vs,ls,C₀,M,pc,Called (C#Cs))#frs,sh[^])
 (is [?P; ?Q; ?R] $\implies P \vdash ?s1 \text{ -jvm} \rightarrow ?s2$)

proof –

assume *assms*: ?P ?Q ?R
then have *exec* (P, ?s1) = [?s2]
by(*case-tac method P C clinit simp*)
then have (?s1, ?s2) \in (*exec-1 P*)^{*}
by(*rule exec-1I[THEN r-into-rtrancl]*)
then show ?thesis **by**(*simp add: exec-all-def1*)

qed

lemma *jvm-InitNonObj*:

[sh C = Some(sfs,Prepared);
 C \neq Object;
 class P C = Some (D,r);
 sh' = sh(C \mapsto (sfs,Processing))]
 $\implies P \vdash$ (None, h, (vs,ls,C₀,M,pc,Calling C Cs)#frs, sh) $\text{-jvm} \rightarrow$
 (None, h, (vs,ls,C₀,M,pc,Calling D (C#Cs))#frs, sh[^])
 (is [?P; ?Q; ?R; ?S] $\implies P \vdash ?s1 \text{ -jvm} \rightarrow ?s2$)

proof –

assume *assms*: ?P ?Q ?R ?S
then have *exec* (P, ?s1) = [?s2]
by(*case-tac method P C clinit simp*)
then have (?s1, ?s2) \in (*exec-1 P*)^{*}
by(*rule exec-1I[THEN r-into-rtrancl]*)
then show ?thesis **by**(*simp add: exec-all-def1*)

qed

lemma *jvm-RInit-throw*:

P \vdash (None,h,(vs,l,C,M,pc,Throwing [] xa) # frs,sh)
 $\text{-jvm} \rightarrow$ handle P C M xa h vs l pc No-ics frs sh
 (is P \vdash ?s1 $\text{-jvm} \rightarrow$?s2)

proof –

have *exec* (P, ?s1) = [?s2]
by(*simp add: handle-def split: bool.splits*)
then have (?s1, ?s2) \in (*exec-1 P*)^{*}
by(*rule exec-1I[THEN r-into-rtrancl]*)
then show ?thesis **by**(*simp add: exec-all-def1*)

qed

lemma *jvm-RInit-throw'*:

P \vdash (None,h,(vs,l,C,M,pc,Throwing [C'] xa) # frs,sh)
 $\text{-jvm} \rightarrow$ handle P C M xa h vs l pc No-ics frs (sh(C':=Some(fst(the(sh C')), Error)))
 (is P \vdash ?s1 $\text{-jvm} \rightarrow$?s2)

proof –

let ?sy = (None,h,(vs,l,C,M,pc,Throwing [] xa) # frs,sh(C':=Some(fst(the(sh C')), Error)))
have *exec* (P, ?s1) = [?sy] **by** *simp*
then have (?s1, ?sy) \in (*exec-1 P*)^{*}
by(*rule exec-1I[THEN r-into-rtrancl]*)
also have (?sy, ?s2) \in (*exec-1 P*)^{*}
using *jvm-RInit-throw* **by**(*simp add: exec-all-def1*)
ultimately show ?thesis **by**(*simp add: exec-all-def1*)

qed

lemma *jvm-Called*:

$P \vdash (\text{None}, h, (vs, l, C, M, pc, \text{Called } (C_0 \# Cs)) \# frs, sh) \text{--jvm--}$
 $(\text{None}, h, \text{create-init-frame } P \ C_0 \# (vs, l, C, M, pc, \text{Called } Cs) \# frs, sh)$
by(*simp add: exec-all-def1 r-into-rtrancl exec-1I*)

lemma *jvm-Throwing*:

$P \vdash (\text{None}, h, (vs, l, C, M, pc, \text{Throwing } (C_0 \# Cs) \ x a') \# frs, sh) \text{--jvm--}$
 $(\text{None}, h, (vs, l, C, M, pc, \text{Throwing } Cs \ x a') \# frs, sh(C_0 \mapsto (\text{fst } (\text{the } (sh \ C_0))), \text{Error})))$
by(*simp add: exec-all-def1 r-into-rtrancl exec-1I*)

Other lemmas for correctness proof

lemma *assumes wf:wf-prog wf-md P*

and *ex: class P C = Some a*

shows *create-init-frame-wf-eq: create-init-frame (compP₂ P) C = (stk,loc,D,M,pc,ics) \implies D=C*
using *wf-sees-clinit[OF wf ex]* **by**(*cases method P C clinit, auto*)

lemma *beforex-try*:

assumes *pcI: {pc..<pc+size(compE₂(try e₁ catch(Ci i) e₂))} \subseteq I*
and *bx: P,C,M \triangleright compxE₂ (try e₁ catch(Ci i) e₂) pc (size vs) / I, size vs*
shows *P,C,M \triangleright compxE₂ e₁ pc (size vs) / {pc..<pc + length (compE₂ e₁)}, size vs*
proof –
obtain *xt₀ xt₁ where*
beforex₀ P C M (size vs) I (compxE₂ (try e₁ catch(Ci i) e₂) pc (size vs)) xt₀ xt₁
using *bx by(clarsimp simp:beforex-def)*
then have $\exists xt_1. \text{beforex}_0 \ P \ C \ M \ (size \ vs) \ \{pc..<pc + length \ (compE_2 \ e_1)\}$
 $(compxE_2 \ e_1 \ pc \ (size \ vs)) \ xt_0 \ xt_1$
using *pcI pcs-subset(1) atLeastLessThan-iff by simp blast*
then show *?thesis using beforex-def by blast*
qed

— Evaluation of initialization expressions

lemma

shows *eval₁-init-return: P \vdash_1 $\langle e, s \rangle \Rightarrow \langle e', s' \rangle$*
 $\implies \text{iconf } (shp_1 \ s) \ e$
 $\implies (\exists Cs \ b. e = \text{INIT } C' \ (Cs, b) \leftarrow \text{unit}) \vee (\exists C \ e_0 \ Cs \ e_i. e = \text{RI}(C, e_0); Cs @ [C'] \leftarrow \text{unit})$
 $\vee (\exists e_0. e = \text{RI}(C', e_0); Nil \leftarrow \text{unit})$
 $\implies (\text{val-of } e' = \text{Some } v \longrightarrow (\exists sfs \ i. shp_1 \ s' \ C' = [(sfs, i)] \wedge (i = \text{Done} \vee i = \text{Processing})))$
 $\wedge (\text{throw-of } e' = \text{Some } a \longrightarrow (\exists sfs \ i. shp_1 \ s' \ C' = [(sfs, \text{Error})]))$

and $P \vdash_1 \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \implies \text{True}$

proof(*induct rule: eval₁-evals₁.inducts*)

case (*InitFinal₁ e s e' s' C b*) **then show** *?case*

by(*auto simp: initPD-def dest: eval₁-final-same*)

next

case (*InitDone₁ sh C sfs C' Cs e h l e' s'*)

then have *final e' using eval₁-final by simp*

then show *?case*

proof(*rule finalE*)

fix *v assume e': e' = Val v then show ?thesis using InitDone₁ initPD-def*

proof(*cases Cs*) **qed**(*auto*)

next

fix *a assume e': e' = throw a then show ?thesis using InitDone₁ initPD-def*

```

    proof(cases Cs) qed(auto)
  qed
next
case (InitProcessing1 sh C sfs C' Cs e h l e' s')
then have final e' using eval1-final by simp
then show ?case
proof(rule finalE)
  fix v assume e': e' = Val v then show ?thesis using InitProcessing1 initPD-def
  proof(cases Cs) qed(auto)
next
  fix a assume e': e' = throw a then show ?thesis using InitProcessing1 initPD-def
  proof(cases Cs) qed(auto)
qed
next
case (InitError1 sh C sfs Cs e h l e' s' C') show ?case
proof(cases Cs)
  case Nil then show ?thesis using InitError1 by simp
next
  case (Cons C2 list)
  then have final e' using InitError1 eval1-final by simp
  then show ?thesis
  proof(rule finalE)
    fix v assume e': e' = Val v show ?thesis
    using InitError1.hypos(2) e' rinit1-throwE by blast
  next
    fix a assume e': e' = throw a
    then show ?thesis using Cons InitError1 cons-to-append[of list] by clarsimp
  qed
qed
next
case (InitRInit1 C Cs h l sh e' s' C') show ?case
proof(cases Cs)
  case Nil then show ?thesis using InitRInit1 by simp
next
  case (Cons C' list) then show ?thesis
  using InitRInit1 Cons cons-to-append[of list] by clarsimp
qed
next
case (RInit1 e s v h' l' sh' C sfs i sh'' C' Cs e' e1 s1)
then have final: final e1 using eval1-final by simp
then show ?case
proof(cases Cs)
  case Nil show ?thesis using final
  proof(rule finalE)
    fix v assume e': e1 = Val v show ?thesis
    using RInit1 Nil by(clarsimp, meson fun-upd-same initPD-def)
  next
    fix a assume e': e1 = throw a show ?thesis
    using RInit1 Nil by(clarsimp, meson fun-upd-same initPD-def)
  qed
qed
next
case (Cons a list) show ?thesis using final
proof(rule finalE)
  fix v assume e': e1 = Val v then show ?thesis

```

```

    using RInit1 Cons by(clarsimp, metis last.simps last-appendR list.distinct(1))
  next
    fix a assume e': e1 = throw a then show ?thesis
    using RInit1 Cons by(clarsimp, metis last.simps last-appendR list.distinct(1))
  qed
qed
next
case (RInitInitFail1 e s a h' l' sh' C sfs i sh'' D Cs e' e1 s1)
then have final: final e1 using eval1-final by simp
then show ?case
proof(rule finalE)
  fix v assume e': e1 = Val v then show ?thesis
  using RInitInitFail1 by(clarsimp, meson exp.distinct(101) rinit1-throwE)
next
  fix a' assume e': e1 = Throw a'
  then have iconf (sh'(C ↦ (sfs, Error))) a
    using RInitInitFail1.hyps(1) eval1-final by fastforce
  then show ?thesis using RInitInitFail1 e'
    by(clarsimp, meson Cons-eq-append-conv list.inject)
qed
qed(auto simp: fun-upd-same)

```

lemma *init₁-Val-PD*: $P \vdash_1 \langle \text{INIT } C' (Cs, b) \leftarrow \text{unit}, s \rangle \Rightarrow \langle \text{Val } v, s \rangle$
 $\Rightarrow \text{iconf } (\text{shp}_1 s) (\text{INIT } C' (Cs, b) \leftarrow \text{unit})$
 $\Rightarrow \exists \text{ sfs } i. \text{shp}_1 s' C' = \lfloor (\text{sfs}, i) \rfloor \wedge (i = \text{Done} \vee i = \text{Processing})$
by(drule-tac $v = v$ **in** eval₁-init-return, simp+)

lemma *init₁-throw-PD*: $P \vdash_1 \langle \text{INIT } C' (Cs, b) \leftarrow \text{unit}, s \rangle \Rightarrow \langle \text{throw } a, s \rangle$
 $\Rightarrow \text{iconf } (\text{shp}_1 s) (\text{INIT } C' (Cs, b) \leftarrow \text{unit})$
 $\Rightarrow \exists \text{ sfs } i. \text{shp}_1 s' C' = \lfloor (\text{sfs}, \text{Error}) \rfloor$
by(drule-tac $a = a$ **in** eval₁-init-return, simp+)

lemma *rinit₁-Val-PD*:
assumes *eval*: $P \vdash_1 \langle \text{RI}(C, e_0); Cs \leftarrow \text{unit}, s \rangle \Rightarrow \langle \text{Val } v, s \rangle$
and *iconf*: $\text{iconf } (\text{shp}_1 s) (\text{RI}(C, e_0); Cs \leftarrow \text{unit})$ **and** *last*: $\text{last}(C \# Cs) = C'$
shows $\exists \text{ sfs } i. \text{shp}_1 s' C' = \lfloor (\text{sfs}, i) \rfloor \wedge (i = \text{Done} \vee i = \text{Processing})$
proof(cases Cs)
 case Nil
 then show ?thesis using eval₁-init-return[OF eval iconf] last **by** simp
next
 case (Cons a list)
 then have nNil: $Cs \neq []$ **by** simp
 then have $\exists Cs'. Cs = Cs' @ [C']$ **using** last append-butlast-last-id[OF nNil]
 by(rule-tac $x = \text{butlast } Cs$ **in** exI) simp
 then show ?thesis using eval₁-init-return[OF eval iconf] **by** simp
qed

lemma *rinit₁-throw-PD*:
assumes *eval*: $P \vdash_1 \langle \text{RI}(C, e_0); Cs \leftarrow \text{unit}, s \rangle \Rightarrow \langle \text{throw } a, s \rangle$
and *iconf*: $\text{iconf } (\text{shp}_1 s) (\text{RI}(C, e_0); Cs \leftarrow \text{unit})$ **and** *last*: $\text{last}(C \# Cs) = C'$
shows $\exists \text{ sfs } i. \text{shp}_1 s' C' = \lfloor (\text{sfs}, \text{Error}) \rfloor$
proof(cases Cs)
 case Nil
 then show ?thesis using eval₁-init-return[OF eval iconf] last **by** simp

next

case (*Cons a list*)

then have $nNil: Cs \neq []$ **by** *simp*

then have $\exists Cs'. Cs = Cs' @ [C']$ **using** *last append-butlast-last-id[OF nNil]*

by(*rule-tac x=butlast Cs in exI*) *simp*

then show *?thesis* **using** *eval₁-init-return[OF eval iconf]* **by** *simp*

qed

The proof

lemma fixes P_1 **defines** [*simp*]: $P \equiv compP_2 P_1$

assumes *wf*: *wf-J₁-prog P₁*

shows $Jcc: P_1 \vdash_1 \langle e, (h_0, ls_0, sh_0) \rangle \Rightarrow \langle ef, (h_1, ls_1, sh_1) \rangle \Longrightarrow$

$(\bigwedge E C M pc ics v xa vs frs I.$

$\llbracket Jcc\text{-cond } P_1 E C M vs pc ics I h_0 sh_0 e \rrbracket \Longrightarrow$

$(ef = Val v \longrightarrow$

$P \vdash (None, h_0, Jcc\text{-frames } P C M vs ls_0 pc ics frs e, sh_0)$

$\text{-jvm} \rightarrow Jcc\text{-rhs } P_1 E C M vs ls_0 pc ics frs h_1 ls_1 sh_1 v e)$

\wedge

$(ef = Throw xa \longrightarrow Jcc\text{-err } P C M h_0 vs ls_0 pc ics frs sh_0 I h_1 ls_1 sh_1 xa e)$

) **and** $P_1 \vdash_1 \langle es, (h_0, ls_0, sh_0) \rangle [\Rightarrow] \langle fs, (h_1, ls_1, sh_1) \rangle \Longrightarrow$

$(\bigwedge C M pc ics ws xa es' vs frs I.$

$\llbracket P, C, M, pc \triangleright compEs_2 es; P, C, M \triangleright compxEs_2 es pc (size vs)/I, size vs;$

$\{pc..<pc+size(compEs_2 es)\} \subseteq I; ics = No\text{-ics};$

$\neg sub\text{-RIs } es \rrbracket \Longrightarrow$

$(fs = map Val ws \longrightarrow$

$P \vdash (None, h_0, (vs, ls_0, C, M, pc, ics) \# frs, sh_0) \text{-jvm} \rightarrow$

$(None, h_1, (rev ws @ vs, ls_1, C, M, pc+size(compEs_2 es), ics) \# frs, sh_1))$

\wedge

$(fs = map Val ws @ Throw xa \# es' \longrightarrow$

$(\exists pc_1. pc \leq pc_1 \wedge pc_1 < pc + size(compEs_2 es) \wedge$

$\neg caught P pc_1 h_1 xa (compxEs_2 es pc (size vs)) \wedge$

$(\exists vs'. P \vdash (None, h_0, (vs, ls_0, C, M, pc, ics) \# frs, sh_0)$

$\text{-jvm} \rightarrow handle P C M xa h_1 (vs' @ vs) ls_1 pc_1 ics frs sh_1))))$

lemma *atLeast0AtMost*[*simp*]: $\{0::nat..n\} = \{..n\}$

by *auto*

lemma *atLeast0LessThan*[*simp*]: $\{0::nat..<n\} = \{..<n\}$

by *auto*

fun *exception* :: '*a exp* \Rightarrow *addr option* **where**

exception (*Throw a*) = *Some a*

| *exception e* = *None*

lemma *comp₂-correct*:

assumes *wf*: *wf-J₁-prog P₁*

and *method*: $P_1 \vdash C \text{ sees } M, b: Ts \rightarrow T = \text{body in } C$

and *eval*: $P_1 \vdash_1 \langle \text{body}, (h, ls, sh) \rangle \Rightarrow \langle e', (h', ls', sh') \rangle$

and *nclinit*: $M \neq \text{clinit}$

shows $compP_2 P_1 \vdash (None, h, ([], ls, C, M, 0, No\text{-ics}), sh) \text{-jvm} \rightarrow (\text{exception } e', h', [], sh')$

end

3.8 Combining Stages 1 and 2

theory *Compiler*

imports *Correctness1 Correctness2*

begin

definition *J2JVM* :: *J-prog* \Rightarrow *jvm-prog*

where

J2JVM \equiv *compP2* \circ *compP1*

theorem *comp-correct-NonStatic*:

assumes *wf*: *wf-J-prog P*

and method: *P* \vdash *C* sees *M*, *NonStatic*: *Ts* \rightarrow *T* = (*pns*, *body*) in *C*

and eval: *P* \vdash \langle *body*, (*h*, [*this*#*pns* \mapsto *vs*], *sh*) $\rangle \Rightarrow$ \langle *e'*, (*h'*, *l'*, *sh'*) \rangle

and sizes: *size vs* = *size pns* + 1 *size rest* = *max-vars body*

shows *J2JVM P* \vdash (*None*, *h*, ([], *vs*@*rest*, *C*, *M*, 0, *No-ics*), *sh*) $\text{-jvm}\rightarrow$ (*exception e'*, *h'*, [], *sh'*)

theorem *comp-correct-Static*:

assumes *wf*: *wf-J-prog P*

and method: *P* \vdash *C* sees *M*, *Static*: *Ts* \rightarrow *T* = (*pns*, *body*) in *C*

and eval: *P* \vdash \langle *body*, (*h*, [*pns* \mapsto *vs*], *sh*) $\rangle \Rightarrow$ \langle *e'*, (*h'*, *l'*, *sh'*) \rangle

and sizes: *size vs* = *size pns* *size rest* = *max-vars body*

and nclinit: *M* \neq *clinit*

shows *J2JVM P* \vdash (*None*, *h*, ([], *vs*@*rest*, *C*, *M*, 0, *No-ics*), *sh*) $\text{-jvm}\rightarrow$ (*exception e'*, *h'*, [], *sh'*)

end

3.9 Preservation of Well-Typedness

theory *TypeComp*

imports *Compiler ../BV/BVSpec*

begin

lemma *max-stack1*: *P, E* \vdash_1 *e* :: *T* \Longrightarrow $1 \leq$ *max-stack e*

locale *TC0* =

fixes *P* :: *J1-prog* **and** *maxl* :: *nat*

begin

definition *ty E e* = (*THE T. P, E* \vdash_1 *e* :: *T*)

definition *ty_l E A'* = *map* ($\lambda i. \text{if } i \in A' \wedge i < \text{size } E \text{ then } \text{OK}(E!i) \text{ else } \text{Err}$) [*0..<maxl*]

definition *ty_i' ST E A* = (*case A of None* \Rightarrow *None* | [*A'*] \Rightarrow *Some*(*ST, ty_l E A'*))

definition *after E A ST e* = *ty_i'* (*ty E e* # *ST*) *E* (*A* \sqcup *A e*)

end

lemma (**in** *TC0*) *ty-def2* [*simp*]: *P, E* \vdash_1 *e* :: *T* \Longrightarrow *ty E e* = *T*

lemma (**in** *TC0*) [*simp*]: *ty_i' ST E None* = *None*

lemma (**in** *TC0*) *ty_l-app-diff* [*simp*]:

ty_l (E@[T]) (A - {size E}) = *ty_l E A*

lemma (**in** *TC0*) *ty_i'-app-diff* [*simp*]:

ty_i' ST (E @ [T]) (A \ominus size E) = *ty_i' ST E A*

lemma (in *TC0*) *ty_l-antimono*:

$$A \subseteq A' \implies P \vdash ty_l E A' [\leq_{\top}] ty_l E A$$

lemma (in *TC0*) *ty_i'-antimono*:

$$A \subseteq A' \implies P \vdash ty_i' ST E [A'] \leq' ty_i' ST E [A]$$

lemma (in *TC0*) *ty_l-env-antimono*:

$$P \vdash ty_l (E@[T]) A [\leq_{\top}] ty_l E A$$

lemma (in *TC0*) *ty_i'-env-antimono*:

$$P \vdash ty_i' ST (E@[T]) A \leq' ty_i' ST E A$$

lemma (in *TC0*) *ty_i'-incr*:

$$P \vdash ty_i' ST (E @ [T]) [insert (size E) A] \leq' ty_i' ST E [A]$$

lemma (in *TC0*) *ty_l-incr*:

$$P \vdash ty_l (E @ [T]) (insert (size E) A) [\leq_{\top}] ty_l E A$$

lemma (in *TC0*) *ty_l-in-types*:

$$set E \subseteq types P \implies ty_l E A \in nlists\ max\ (err\ (types\ P))$$

locale *TC1 = TC0*

begin

primrec *compT* :: *ty list* \Rightarrow *nat hyperset* \Rightarrow *ty list* \Rightarrow *expr₁* \Rightarrow *ty_i' list* **and**

compTs :: *ty list* \Rightarrow *nat hyperset* \Rightarrow *ty list* \Rightarrow *expr₁ list* \Rightarrow *ty_i' list* **where**

$$compT E A ST (new C) = []$$

$$| compT E A ST (Cast C e) =$$

$$compT E A ST e @ [after E A ST e]$$

$$| compT E A ST (Val v) = []$$

$$| compT E A ST (e_1 \ll bop \gg e_2) =$$

$$(let ST_1 = ty E e_1 \# ST; A_1 = A \sqcup \mathcal{A} e_1 \text{ in}$$

$$compT E A ST e_1 @ [after E A ST e_1] @$$

$$compT E A_1 ST_1 e_2 @ [after E A_1 ST_1 e_2])$$

$$| compT E A ST (Var i) = []$$

$$| compT E A ST (i := e) = compT E A ST e @$$

$$[after E A ST e, ty_i' ST E (A \sqcup \mathcal{A} e \sqcup [\{i\}])]$$

$$| compT E A ST (e.F\{D\}) =$$

$$compT E A ST e @ [after E A ST e]$$

$$| compT E A ST (C.sF\{D\}) = []$$

$$| compT E A ST (e_1.F\{D\} := e_2) =$$

$$(let ST_1 = ty E e_1 \# ST; A_1 = A \sqcup \mathcal{A} e_1; A_2 = A_1 \sqcup \mathcal{A} e_2 \text{ in}$$

$$compT E A ST e_1 @ [after E A ST e_1] @$$

$$compT E A_1 ST_1 e_2 @ [after E A_1 ST_1 e_2] @$$

$$[ty_i' ST E A_2])$$

$$| compT E A ST (C.sF\{D\} := e_2) = compT E A ST e_2 @ [after E A ST e_2] @ [ty_i' ST E (A \sqcup \mathcal{A} e_2)]$$

$$| compT E A ST \{i:T; e\} = compT (E@[T]) (A \ominus i) ST e$$

$$| compT E A ST (e_1;;e_2) =$$

$$(let A_1 = A \sqcup \mathcal{A} e_1 \text{ in}$$

$$compT E A ST e_1 @ [after E A ST e_1, ty_i' ST E A_1] @$$

$$compT E A_1 ST e_2)$$

$$| compT E A ST (if (e) e_1 else e_2) =$$

$$(let A_0 = A \sqcup \mathcal{A} e; \tau = ty_i' ST E A_0 \text{ in}$$

$$compT E A ST e @ [after E A ST e, \tau] @$$

$$\begin{aligned}
& \text{compT } E \ A_0 \ ST \ e_1 \ @ \ [\text{after } E \ A_0 \ ST \ e_1, \ \tau] \ @ \\
& \text{compT } E \ A_0 \ ST \ e_2) \\
| \text{compT } E \ A \ ST \ (\text{while } (e) \ c) = & \\
& (\text{let } A_0 = A \sqcup \mathcal{A} \ e; \ A_1 = A_0 \sqcup \mathcal{A} \ c; \ \tau = \text{ty}_i' \ ST \ E \ A_0 \ \text{in} \\
& \text{compT } E \ A \ ST \ e \ @ \ [\text{after } E \ A \ ST \ e, \ \tau] \ @ \\
& \text{compT } E \ A_0 \ ST \ c \ @ \ [\text{after } E \ A_0 \ ST \ c, \ \text{ty}_i' \ ST \ E \ A_1, \ \text{ty}_i' \ ST \ E \ A_0]) \\
| \text{compT } E \ A \ ST \ (\text{throw } e) = & \text{compT } E \ A \ ST \ e \ @ \ [\text{after } E \ A \ ST \ e] \\
| \text{compT } E \ A \ ST \ (e.M(es)) = & \\
& \text{compT } E \ A \ ST \ e \ @ \ [\text{after } E \ A \ ST \ e] \ @ \\
& \text{compTs } E \ (A \sqcup \mathcal{A} \ e) \ (\text{ty } E \ e \ # \ ST) \ es \\
| \text{compT } E \ A \ ST \ (C.sM(es)) = & \text{compTs } E \ A \ ST \ es \\
| \text{compT } E \ A \ ST \ (\text{try } e_1 \ \text{catch}(C \ i) \ e_2) = & \\
& \text{compT } E \ A \ ST \ e_1 \ @ \ [\text{after } E \ A \ ST \ e_1] \ @ \\
& [\text{ty}_i' \ (\text{Class } C \ # \ ST) \ E \ A, \ \text{ty}_i' \ ST \ (E @ [\text{Class } C]) \ (A \sqcup \{\{i\}\})] \ @ \\
& \text{compT } (E @ [\text{Class } C]) \ (A \sqcup \{\{i\}\}) \ ST \ e_2 \\
| \text{compT } E \ A \ ST \ (\text{INIT } C \ (Cs, b) \ \leftarrow \ e) = & [] \\
| \text{compT } E \ A \ ST \ (RI(C, e'); Cs \ \leftarrow \ e) = & [] \\
| \text{compTs } E \ A \ ST \ [] = & [] \\
| \text{compTs } E \ A \ ST \ (e \ # \ es) = & \text{compT } E \ A \ ST \ e \ @ \ [\text{after } E \ A \ ST \ e] \ @ \\
& \text{compTs } E \ (A \sqcup (\mathcal{A} \ e)) \ (\text{ty } E \ e \ # \ ST) \ es
\end{aligned}$$

definition $\text{compT}_a :: \text{ty list} \Rightarrow \text{nat hyperset} \Rightarrow \text{ty list} \Rightarrow \text{expr}_1 \Rightarrow \text{ty}_i' \ \text{list}$ **where**
 $\text{compT}_a \ E \ A \ ST \ e = \text{compT } E \ A \ ST \ e \ @ \ [\text{after } E \ A \ ST \ e]$

end

lemma $\text{compE}_2\text{-not-Nil}[\text{simp}]$: $P, E \vdash_1 e :: T \Longrightarrow \text{compE}_2 \ e \neq []$

lemma (in *TC1*) $\text{compT}\text{-sizes}'$:

shows $\bigwedge E \ A \ ST. \neg \text{sub-RI } e \Longrightarrow \text{size}(\text{compT } E \ A \ ST \ e) = \text{size}(\text{compE}_2 \ e) - 1$

and $\bigwedge E \ A \ ST. \neg \text{sub-RIs } es \Longrightarrow \text{size}(\text{compTs } E \ A \ ST \ es) = \text{size}(\text{compEs}_2 \ es)$

lemma (in *TC1*) $\text{compT}\text{-sizes}[\text{simp}]$:

shows $\bigwedge E \ A \ ST. P, E \vdash_1 e :: T \Longrightarrow \text{size}(\text{compT } E \ A \ ST \ e) = \text{size}(\text{compE}_2 \ e) - 1$

and $\bigwedge E \ A \ ST. P, E \vdash_1 es [::] Ts \Longrightarrow \text{size}(\text{compTs } E \ A \ ST \ es) = \text{size}(\text{compEs}_2 \ es)$

lemma (in *TC1*) $[\text{simp}]$: $\bigwedge ST \ E. [\tau] \notin \text{set}(\text{compT } E \ \text{None} \ ST \ e)$

and $[\text{simp}]$: $\bigwedge ST \ E. [\tau] \notin \text{set}(\text{compTs } E \ \text{None} \ ST \ es)$

lemma (in *TC0*) $\text{pair-eq-ty}_i'\text{-conv}$:

$([\text{ST}, \text{LT}]) = \text{ty}_i' \ ST_0 \ E \ A) =$

(case A of $\text{None} \Rightarrow \text{False} \mid \text{Some } A \Rightarrow (\text{ST} = \text{ST}_0 \wedge \text{LT} = \text{ty}_l \ E \ A)$)

lemma (in *TC0*) $\text{pair-conv-ty}_i'$:

$[\text{ST}, \text{ty}_l \ E \ A] = \text{ty}_i' \ ST \ E \ [A]$

lemma (in *TC1*) $\text{compT}\text{-LT}\text{-prefix}$:

$\bigwedge E \ A \ ST_0. \llbracket [\text{ST}, \text{LT}] \in \text{set}(\text{compT } E \ A \ ST_0 \ e); \mathcal{B} \ e \ (\text{size } E) \rrbracket$

$\Longrightarrow P \vdash [\text{ST}, \text{LT}] \leq' \text{ty}_i' \ ST \ E \ A$

and

$\bigwedge E \ A \ ST_0. \llbracket [\text{ST}, \text{LT}] \in \text{set}(\text{compTs } E \ A \ ST_0 \ es); \mathcal{B} \ es \ (\text{size } E) \rrbracket$

$\Longrightarrow P \vdash [\text{ST}, \text{LT}] \leq' \text{ty}_i' \ ST \ E \ A$

lemma $[\text{iff}]$: $OK \ \text{None} \in \text{states } P \ \text{mxs} \ \text{m}xl$

lemma (in *TC0*) after-in-states :

assumes wf : $wf\text{-prog } p \ P$ **and** wt : $P, E \vdash_1 e :: T$

and $Etypes$: set $E \subseteq types\ P$ **and** $STtypes$: set $ST \subseteq types\ P$
and $stack$: size $ST + max-stack\ e \leq mxs$
shows OK (after $E\ A\ ST\ e$) $\in states\ P\ mxs\ mxl$

lemma (in $TC0$) $OK-ty_i'-in-statesI[simp]$:
 $\llbracket set\ E \subseteq types\ P; set\ ST \subseteq types\ P; size\ ST \leq mxs \rrbracket$
 $\implies OK\ (ty_i'\ ST\ E\ A) \in states\ P\ mxs\ mxl$

lemma $is-class-type-aux$: $is-class\ P\ C \implies is-type\ P$ (Class C)

theorem (in $TC1$) $compT-states$:

assumes wf : $wf-prog\ p\ P$

shows $\bigwedge E\ T\ A\ ST$.

$\llbracket P, E \vdash_1 e :: T; set\ E \subseteq types\ P; set\ ST \subseteq types\ P;$
 $size\ ST + max-stack\ e \leq mxs; size\ E + max-vars\ e \leq mxl \rrbracket$
 $\implies OK\ 'set(compT\ E\ A\ ST\ e) \subseteq states\ P\ mxs\ mxl$

and $\bigwedge E\ Ts\ A\ ST$.

$\llbracket P, E \vdash_1 es[::]Ts; set\ E \subseteq types\ P; set\ ST \subseteq types\ P;$
 $size\ ST + max-stacks\ es \leq mxs; size\ E + max-varss\ es \leq mxl \rrbracket$
 $\implies OK\ 'set(compTs\ E\ A\ ST\ es) \subseteq states\ P\ mxs\ mxl$

definition $shift :: nat \Rightarrow ex-table \Rightarrow ex-table$

where

$shift\ n\ xt \equiv map\ (\lambda(from, to, C, handler, depth). (from+n, to+n, C, handler+n, depth))\ xt$

lemma $[simp]$: $shift\ 0\ xt = xt$

lemma $[simp]$: $shift\ n\ [] = []$

lemma $[simp]$: $shift\ n\ (xt_1 @ xt_2) = shift\ n\ xt_1 @ shift\ n\ xt_2$

lemma $[simp]$: $shift\ m\ (shift\ n\ xt) = shift\ (m+n)\ xt$

lemma $[simp]$: $pcs\ (shift\ n\ xt) = \{pc+n \mid pc. pc \in pcs\ xt\}$

lemma $shift-compxE_2$:

shows $\bigwedge pc\ pc'\ d. shift\ pc\ (compxE_2\ e\ pc'\ d) = compxE_2\ e\ (pc' + pc)\ d$

and $\bigwedge pc\ pc'\ d. shift\ pc\ (compxEs_2\ es\ pc'\ d) = compxEs_2\ es\ (pc' + pc)\ d$

lemma $compxE_2-size-convs[simp]$:

shows $n \neq 0 \implies compxE_2\ e\ n\ d = shift\ n\ (compxE_2\ e\ 0\ d)$

and $n \neq 0 \implies compxEs_2\ es\ n\ d = shift\ n\ (compxEs_2\ es\ 0\ d)$

locale $TC2 = TC1 +$

fixes $T_r :: ty$ **and** $mxs :: pc$

begin

definition

$wt-instrs :: instr\ list \Rightarrow ex-table \Rightarrow ty_i'\ list \Rightarrow bool$

$(\langle \vdash -, - / [::] / - \rangle [0, 0, 51] 50)$ **where**

$\vdash is, xt [::] \tau s \iff size\ is < size\ \tau s \wedge pcs\ xt \subseteq \{0..<size\ is\} \wedge$

$(\forall pc < size\ is. P, T_r, mxs, size\ \tau s, xt \vdash is!pc, pc :: \tau s)$

end

notation $TC2.wt-instrs (\langle \vdash -, - / [::] / - \rangle [50, 50, 50, 50, 50, 51] 50)$

lemma (in $TC2$) $[simp]$: $\tau s \neq [] \implies \vdash [], [] [::] \tau s$

lemma $[simp]$: $eff\ i\ P\ pc\ et\ None = []$

lemma *wt-instr-appR*:

$\llbracket P, T, m, mpc, xt \vdash is!pc, pc :: \tau s;$
 $pc < size\ is; size\ is < size\ \tau s; mpc \leq size\ \tau s; mpc \leq mpc' \rrbracket$
 $\implies P, T, m, mpc', xt \vdash is!pc, pc :: \tau s @ \tau s'$

lemma *relevant-entries-shift* [*simp*]:

relevant-entries $P\ i\ (pc+n)\ (shift\ n\ xt) = shift\ n\ (relevant-entries\ P\ i\ pc\ xt)$

lemma [*simp*]:

xcpt-eff $i\ P\ (pc+n)\ \tau\ (shift\ n\ xt) =$
 $map\ (\lambda(pc, \tau). (pc + n, \tau))\ (xcpt-eff\ i\ P\ pc\ \tau\ xt)$

lemma [*simp*]:

app_i $(i, P, pc, m, T, \tau) \implies$
 $eff\ i\ P\ (pc+n)\ (shift\ n\ xt)\ (Some\ \tau) =$
 $map\ (\lambda(pc, \tau). (pc+n, \tau))\ (eff\ i\ P\ pc\ xt\ (Some\ \tau))$

lemma [*simp*]:

xcpt-app $i\ P\ (pc+n)\ m\ x\ s\ (shift\ n\ xt)\ \tau = xcpt-app\ i\ P\ pc\ m\ x\ s\ xt\ \tau$

lemma *wt-instr-appL*:

assumes $P, T, m, mpc, xt \vdash i, pc :: \tau s$ **and** $pc < size\ \tau s$ **and** $mpc \leq size\ \tau s$
shows $P, T, m, mpc + size\ \tau s', shift\ (size\ \tau s')\ xt \vdash i, pc + size\ \tau s' :: \tau s' @ \tau s$

lemma *wt-instr-Cons*:

assumes *wti*: $P, T, m, mpc - 1, [] \vdash i, pc - 1 :: \tau s$
and *pcl*: $0 < pc$ **and** *mpcl*: $0 < mpc$
and *pcu*: $pc < size\ \tau s + 1$ **and** *mpcu*: $mpc \leq size\ \tau s + 1$
shows $P, T, m, mpc, [] \vdash i, pc :: \tau \# \tau s$

lemma *wt-instr-append*:

assumes *wti*: $P, T, m, mpc - size\ \tau s', [] \vdash i, pc - size\ \tau s' :: \tau s$
and *pcl*: $size\ \tau s' \leq pc$ **and** *mpcl*: $size\ \tau s' \leq mpc$
and *pcu*: $pc < size\ \tau s + size\ \tau s'$ **and** *mpcu*: $mpc \leq size\ \tau s + size\ \tau s'$
shows $P, T, m, mpc, [] \vdash i, pc :: \tau s' @ \tau s$

lemma *xcpt-app-pcs*:

$pc \notin pcs\ xt \implies xcpt-app\ i\ P\ pc\ m\ x\ s\ xt\ \tau$

lemma *xcpt-eff-pcs*:

$pc \notin pcs\ xt \implies xcpt-eff\ i\ P\ pc\ \tau\ xt = []$

lemma *pcs-shift*:

$pc < n \implies pc \notin pcs\ (shift\ n\ xt)$

lemma *wt-instr-appRx*:

$\llbracket P, T, m, mpc, xt \vdash is!pc, pc :: \tau s; pc < size\ is; size\ is < size\ \tau s; mpc \leq size\ \tau s \rrbracket$
 $\implies P, T, m, mpc, xt @ shift\ (size\ is)\ xt' \vdash is!pc, pc :: \tau s$

lemma *wt-instr-appLx*:

$\llbracket P, T, m, mpc, xt \vdash i, pc :: \tau s; pc \notin pcs\ xt' \rrbracket$
 $\implies P, T, m, mpc, xt' @ xt \vdash i, pc :: \tau s$

lemma (**in** *TC2*) *wt-instrs-extR*:

$\vdash is,xt [::] \tau s \implies \vdash is,xt [::] \tau s @ \tau s'$

lemma (in *TC2*) *wt-instrs-ext*:

assumes $wt_1: \vdash is_1,xt_1 [::] \tau s_1 @ \tau s_2$ **and** $wt_2: \vdash is_2,xt_2 [::] \tau s_2$
and $\tau s\text{-size}: size \tau s_1 = size is_1$

shows $\vdash is_1 @ is_2, xt_1 @ shift (size is_1) xt_2 [::] \tau s_1 @ \tau s_2$

corollary (in *TC2*) *wt-instrs-ext2*:

$\llbracket \vdash is_2,xt_2 [::] \tau s_2; \vdash is_1,xt_1 [::] \tau s_1 @ \tau s_2; size \tau s_1 = size is_1 \rrbracket$
 $\implies \vdash is_1 @ is_2, xt_1 @ shift (size is_1) xt_2 [::] \tau s_1 @ \tau s_2$

corollary (in *TC2*) *wt-instrs-ext-prefix* [*trans*]:

$\llbracket \vdash is_1,xt_1 [::] \tau s_1 @ \tau s_2; \vdash is_2,xt_2 [::] \tau s_3;$
 $size \tau s_1 = size is_1; prefix \tau s_3 \tau s_2 \rrbracket$
 $\implies \vdash is_1 @ is_2, xt_1 @ shift (size is_1) xt_2 [::] \tau s_1 @ \tau s_2$

corollary (in *TC2*) *wt-instrs-app*:

assumes $is_1: \vdash is_1,xt_1 [::] \tau s_1 @ [\tau]$
assumes $is_2: \vdash is_2,xt_2 [::] \tau \# \tau s_2$
assumes $s: size \tau s_1 = size is_1$
shows $\vdash is_1 @ is_2, xt_1 @ shift (size is_1) xt_2 [::] \tau s_1 @ \tau \# \tau s_2$

corollary (in *TC2*) *wt-instrs-app-last* [*trans*]:

assumes $\vdash is_2,xt_2 [::] \tau \# \tau s_2 \vdash is_1,xt_1 [::] \tau s_1$
 $last \tau s_1 = \tau \ size \tau s_1 = size is_1 + 1$
shows $\vdash is_1 @ is_2, xt_1 @ shift (size is_1) xt_2 [::] \tau s_1 @ \tau s_2$

corollary (in *TC2*) *wt-instrs-append-last* [*trans*]:

assumes $wtis: \vdash is,xt [::] \tau s$ **and** $wti: P, T_r, max, mpc, [] \vdash i, pc :: \tau s$
and $pc: pc = size is$ **and** $mpc: mpc = size \tau s$ **and** $is\text{-}\tau s: size is + 1 < size \tau s$
shows $\vdash is @ [i], xt [::] \tau s$

corollary (in *TC2*) *wt-instrs-app2*:

$\llbracket \vdash is_2,xt_2 [::] \tau' \# \tau s_2; \vdash is_1,xt_1 [::] \tau \# \tau s_1 @ [\tau'];$
 $xt' = xt_1 @ shift (size is_1) xt_2; size \tau s_1 + 1 = size is_1 \rrbracket$
 $\implies \vdash is_1 @ is_2, xt' [::] \tau \# \tau s_1 @ \tau' \# \tau s_2$

corollary (in *TC2*) *wt-instrs-app2-simp* [*trans*, *simp*]:

$\llbracket \vdash is_2,xt_2 [::] \tau' \# \tau s_2; \vdash is_1,xt_1 [::] \tau \# \tau s_1 @ [\tau']; size \tau s_1 + 1 = size is_1 \rrbracket$
 $\implies \vdash is_1 @ is_2, xt_1 @ shift (size is_1) xt_2 [::] \tau \# \tau s_1 @ \tau' \# \tau s_2$

corollary (in *TC2*) *wt-instrs-Cons* [*simp*]:

$\llbracket \tau s \neq []; \vdash [i], [] [::] [\tau, \tau']; \vdash is,xt [::] \tau' \# \tau s \rrbracket$
 $\implies \vdash i \# is, shift 1 xt [::] \tau \# \tau' \# \tau s$

theory *JinjaDCI*

imports

J/Equivalence
J/Annotate
JVM/JVMDefensive
BV/BVExec
BV/LBVJVM
BV/BVNoTypeError
Compiler/TypeComp

begin

end

Bibliography

- [1] S. Mansky and E. L. Gunter. Dynamic class initialization semantics: A jinja extension. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 209–221, New York, NY, USA, 2019. Association for Computing Machinery.