

A Machine-Checked Model for a Java-like Language,
Virtual Machine and Compiler

Gerwin Klein

Tobias Nipkow

May 26, 2024

Contents

1	Preface	5
1.1	Theory Dependencies	5
2	Jinja Source Language	7
2.1	Auxiliary Definitions	7
2.2	Jinja types	8
2.3	Class Declarations and Programs	9
2.4	Relations between Jinja Types	10
2.5	Jinja Values	16
2.6	Objects and the Heap	16
2.7	Exceptions	18
2.8	Expressions	19
2.9	Program State	21
2.10	Big Step Semantics	21
2.11	Small Step Semantics	26
2.12	System Classes	30
2.13	Generic Well-formedness of programs	31
2.14	Weak well-formedness of Jinja programs	34
2.15	Equivalence of Big Step and Small Step Semantics	34
2.16	Well-typedness of Jinja expressions	40
2.17	Runtime Well-typedness	42
2.18	Definite assignment	44
2.19	Conformance Relations for Type Soundness Proofs	46
2.20	Progress of Small Step Semantics	48
2.21	Well-formedness Constraints	50
2.22	Type Safety Proof	50
2.23	Program annotation	53
2.24	Example Expressions	54
2.25	Code Generation For BigStep	56
2.26	Code Generation For WellType	58
3	Jinja Virtual Machine	61
3.1	State of the JVM	61
3.2	Instructions of the JVM	61
3.3	JVM Instruction Semantics	62
3.4	Exception handling in the JVM	65
3.5	Program Execution in the JVM	66

3.6	A Defensive JVM	67
3.7	Example for generating executable code from JVM semantics	70
4	Bytecode Verifier	73
4.1	Semilattices	73
4.2	The Error Type	76
4.3	More about Options	78
4.4	Products as Semilattices	78
4.5	Fixed Length Lists	79
4.6	Typing and Dataflow Analysis Framework	80
4.7	More on Semilattices	81
4.8	Lifting the Typing Framework to <code>err</code> , <code>app</code> , and <code>eff</code>	82
4.9	Kildall's Algorithm	84
4.10	Kildall's Algorithm	84
4.11	The Lightweight Bytecode Verifier	86
4.12	Correctness of the LBV	90
4.13	Completeness of the LBV	91
4.14	The Jinja Type System as a Semilattice	92
4.15	The JVM Type System as Semilattice	94
4.16	Effect of Instructions on the State Type	96
4.17	Monotonicity of <code>eff</code> and <code>app</code>	102
4.18	The Bytecode Verifier	102
4.19	The Typing Framework for the JVM	103
4.20	Typing and Dataflow Analysis Framework	105
4.21	Kildall for the JVM	106
4.22	LBV for the JVM	107
4.23	BV Type Safety Invariant	108
4.24	BV Type Safety Proof	110
4.25	Welltyped Programs produce no Type Errors	116
4.26	Example Welltypings	117
5	Compilation	125
5.1	An Intermediate Language	125
5.2	Well-Formedness of Intermediate Language	129
5.3	Program Compilation	131
5.4	Compilation Stage 1	134
5.5	Correctness of Stage 1	135
5.6	Compilation Stage 2	137
5.7	Correctness of Stage 2	139
5.8	Combining Stages 1 and 2	143
5.9	Preservation of Well-Typedness	143

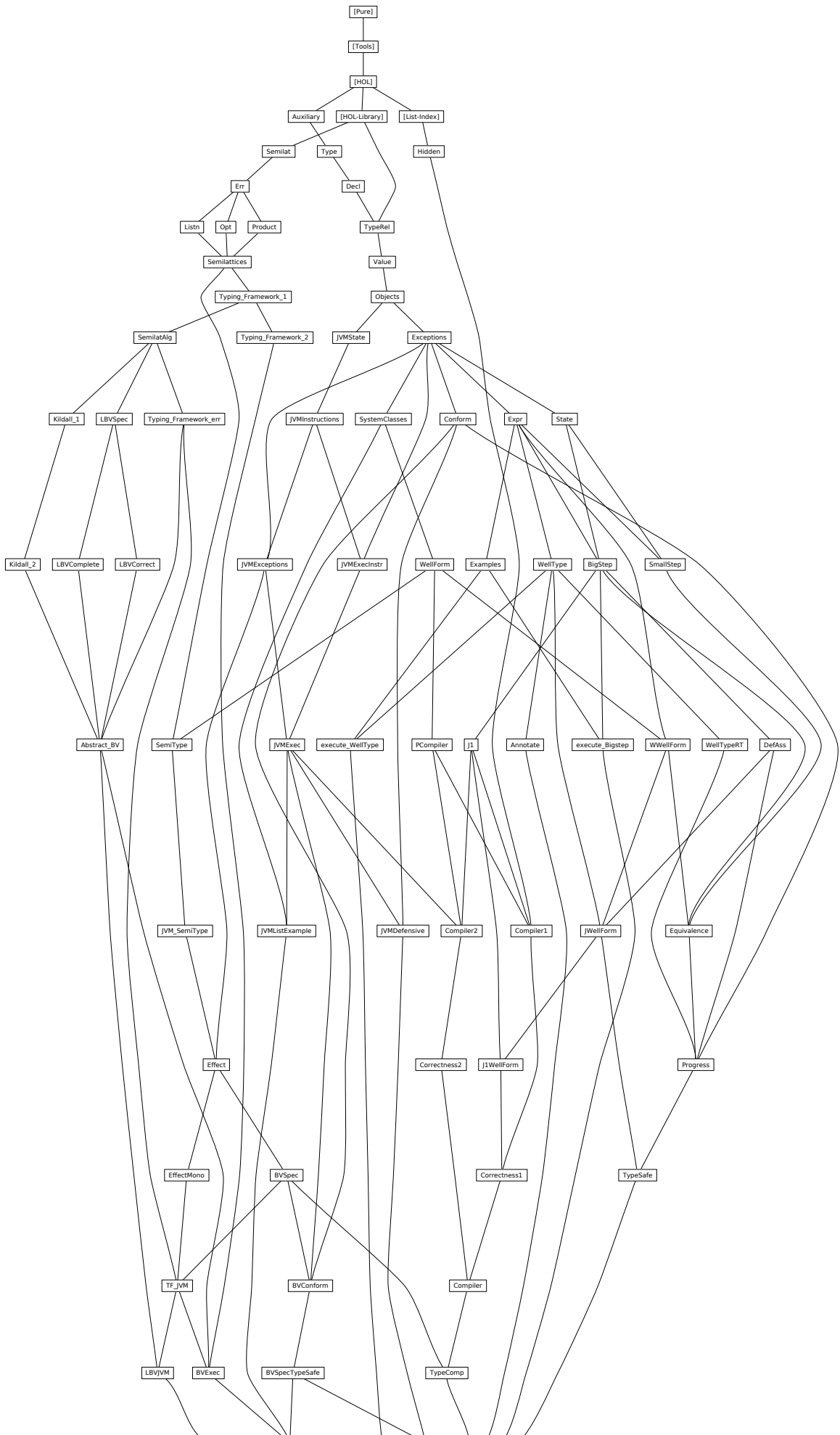
Chapter 1

Preface

This document contains the automatically generated listings of the Isabelle sources for the theories defining and analysing Jinja (a Java-like programming language), the Jinja Virtual Machine, and the compiler. To shorten the document, all proofs have been hidden. For a detailed exposition of these theories see the paper by Klein and Nipkow [1, 2].

1.1 Theory Dependencies

Figure 1.1 shows the dependencies between the Isabelle theories in the following sections.



Chapter 2

Jinja Source Language

2.1 Auxiliary Definitions

theory *Auxiliary* imports *Main* begin

lemma *nat-add-max-le[simp]*:

$$((n::nat) + \max i j \leq m) = (n + i \leq m \wedge n + j \leq m)$$

\langle *proof* \rangle

lemma *Suc-add-max-le[simp]*:

$$(Suc(n + \max i j) \leq m) = (Suc(n + i) \leq m \wedge Suc(n + j) \leq m) \langle$$
proof \rangle

notation *Some* ($([-])$)

2.1.1 *distinct-fst*

definition *distinct-fst* :: ('a × 'b) list ⇒ bool

where

$$distinct-fst \equiv distinct \circ map\ fst$$

lemma *distinct-fst-Nil [simp]*:

$$distinct-fst []$$

\langle *proof* \rangle

lemma *distinct-fst-Cons [simp]*:

$$distinct-fst ((k,x)\#kxs) = (distinct-fst kxs \wedge (\forall y. (k,y) \notin set\ kxs)) \langle$$
proof \rangle

lemma *map-of-SomeI*:

$$[[distinct-fst\ kxs; (k,x) \in set\ kxs]] \implies map-of\ kxs\ k = Some\ x \langle$$
proof \rangle

2.1.2 Using *list-all2* for relations

definition *fun-of* :: ('a × 'b) set ⇒ 'a ⇒ 'b ⇒ bool

where

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

Convenience lemmas

lemma *rel-list-all2-Cons [iff]*:

$$list-all2\ (fun-of\ S)\ (x\#xs)\ (y\#ys) =$$

$$((x,y) \in S \wedge list-all2\ (fun-of\ S)\ xs\ ys)$$

\langle *proof* \rangle

lemma *rel-list-all2-Cons1*:

$list\text{-}all2\ (fun\text{-}of\ S)\ (x\#xs)\ ys =$
 $(\exists z\ zs.\ ys = z\#zs \wedge (x,z) \in S \wedge list\text{-}all2\ (fun\text{-}of\ S)\ xs\ zs)$
<proof>

lemma *rel-list-all2-Cons2*:

$list\text{-}all2\ (fun\text{-}of\ S)\ xs\ (y\#ys) =$
 $(\exists z\ zs.\ xs = z\#zs \wedge (z,y) \in S \wedge list\text{-}all2\ (fun\text{-}of\ S)\ zs\ ys)$
<proof>

lemma *rel-list-all2-refl*:

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

lemma *rel-list-all2-antisym*:

$\llbracket (\bigwedge x\ y.\ \llbracket (x,y) \in S; (y,x) \in T \rrbracket \implies x = y);$
 $list\text{-}all2\ (fun\text{-}of\ S)\ xs\ ys; list\text{-}all2\ (fun\text{-}of\ T)\ ys\ xs \rrbracket \implies xs = ys$
<proof>

lemma *rel-list-all2-trans*:

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

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

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

lemma *rel-list-all2-nthD*:

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

lemma *rel-list-all2I*:

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

end

2.2 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

$this \equiv "this"$

— types

datatype *ty*

= *Void* — type of statements

| *Boolean*

| *Integer*

| *NT* — null type
| *Class cname* — class type

definition *is-refT* :: *ty* ⇒ *bool*

where

is-refT T ≡ *T* = *NT* ∨ (∃ *C*. *T* = *Class C*)

lemma [*iff*]: *is-refT NT* ⟨*proof*⟩

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

lemma *refTE*:

[[*is-refT T*; *T* = *NT* ⇒ *P*; ∧*C*. *T* = *Class C* ⇒ *P*]] ⇒ *P* ⟨*proof*⟩

lemma *not-refTE*:

[[¬*is-refT T*; *T* = *Void* ∨ *T* = *Boolean* ∨ *T* = *Integer* ⇒ *P*]] ⇒ *P* ⟨*proof*⟩

end

2.3 Class Declarations and Programs

theory *Decl* **imports** *Type* **begin**

type-synonym

fdecl = *vname* × *ty* — field declaration

type-synonym

'm mdecl = *mname* × *ty list* × *ty* × *'m* — method = name, 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*

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}*

⟨*proof*⟩

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* ⟨*proof*⟩

abbreviation

types P == *Collect (is-type P)*

end

2.4 Relations between Jinja Types

theory *TypeRel* imports

HOL-Library.Transitive-Closure-Table

Decl

begin

2.4.1 The subclass relations

inductive-set

subcls1 :: 'm prog \Rightarrow (cname \times cname) set

and *subcls1'* :: 'm prog \Rightarrow [cname, cname] \Rightarrow bool (- \vdash - \prec^1 - [71,71,71] 70)

for *P* :: 'm prog

where

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

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

abbreviation

subcls :: 'm prog \Rightarrow [cname, cname] \Rightarrow bool (- \vdash - \preceq^* - [71,71,71] 70)

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

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

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

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

lemma *subcls1-def2*:

subcls1 *P* =

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

lemma *finite-subcls1*: *finite* (*subcls1* *P*) $\langle \text{proof} \rangle$

2.4.2 The subtype relations

inductive

widen :: 'm prog \Rightarrow ty \Rightarrow ty \Rightarrow bool (- \vdash - \leq - [71,71,71] 70)

for *P* :: 'm prog

where

widen-refl[*iff*]: $P \vdash T \leq T$

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

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

abbreviation

widens :: 'm prog \Rightarrow ty list \Rightarrow ty list \Rightarrow bool

(- \vdash - [\leq] - [71,71,71] 70) **where**

widens *P* *Ts* *Ts'* $\equiv \text{list-all2 } (\text{widen } P) \ Ts \ Ts'$

lemma [*iff*]: $(P \vdash T \leq \text{Void}) = (T = \text{Void}) \langle \text{proof} \rangle$

lemma [*iff*]: $(P \vdash T \leq \text{Boolean}) = (T = \text{Boolean}) \langle \text{proof} \rangle$

lemma [*iff*]: $(P \vdash T \leq \text{Integer}) = (T = \text{Integer}) \langle \text{proof} \rangle$

lemma [*iff*]: $(P \vdash \text{Void} \leq T) = (T = \text{Void}) \langle \text{proof} \rangle$

lemma [*iff*]: $(P \vdash \text{Boolean} \leq T) = (T = \text{Boolean}) \langle \text{proof} \rangle$

lemma [*iff*]: $(P \vdash \text{Integer} \leq T) = (T = \text{Integer}) \langle \text{proof} \rangle$

lemma *Class-widen*: $P \vdash \text{Class } C \leq T \Longrightarrow \exists D. T = \text{Class } D \langle \text{proof} \rangle$

lemma *[iff]*: $(P \vdash T \leq NT) = (T = NT)\langle proof \rangle$

lemma *Class-widen-Class [iff]*: $(P \vdash \text{Class } C \leq \text{Class } D) = (P \vdash C \preceq^* D)\langle proof \rangle$

lemma *widen-Class*: $(P \vdash T \leq \text{Class } C) = (T = NT \vee (\exists D. T = \text{Class } D \wedge P \vdash D \preceq^* C))\langle proof \rangle$

lemma *widen-trans[trans]*: $\llbracket P \vdash S \leq U; P \vdash U \leq T \rrbracket \Longrightarrow P \vdash S \leq T\langle proof \rangle$

lemma *widens-trans [trans]*: $\llbracket P \vdash Ss \leq Ts; P \vdash Ts \leq Us \rrbracket \Longrightarrow P \vdash Ss \leq Us\langle proof \rangle$

2.4.3 Method lookup

inductive

Methods :: $['m \text{ prog}, \text{cname}, \text{mname} \rightarrow (\text{ty list} \times \text{ty} \times 'm) \times \text{cname}] \Rightarrow \text{bool}$
 $(- \vdash - \text{sees}'\text{-methods} - [51,51,51] 50)$

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

where

sees-methods-Object:

$\llbracket \text{class } P \text{ Object} = \text{Some}(D,fs,ms); Mm = \text{map-option } (\lambda m. (m, \text{Object})) \circ \text{map-of } ms \rrbracket$
 $\Longrightarrow P \vdash \text{Object sees-methods } Mm$

| *sees-methods-rec*:

$\llbracket \text{class } P \text{ C} = \text{Some}(D,fs,ms); C \neq \text{Object}; P \vdash D \text{ sees-methods } Mm;$
 $Mm' = Mm ++ (\text{map-option } (\lambda m. (m, C)) \circ \text{map-of } ms) \rrbracket$
 $\Longrightarrow P \vdash C \text{ sees-methods } Mm'$

lemma *sees-methods-fun*:

assumes $1: P \vdash C \text{ sees-methods } Mm$

shows $\bigwedge Mm'. P \vdash C \text{ sees-methods } Mm' \Longrightarrow Mm' = Mm$

$\langle proof \rangle$

lemma *visible-methods-exist*:

$P \vdash C \text{ sees-methods } Mm \Longrightarrow Mm M = \text{Some}(m, D) \Longrightarrow$

$(\exists D' fs ms. \text{class } P \text{ D} = \text{Some}(D', fs, ms) \wedge \text{map-of } ms M = \text{Some } m)$

$\langle proof \rangle$

lemma *sees-methods-decl-above*:

assumes $C \text{ sees}: P \vdash C \text{ sees-methods } Mm$

shows $Mm M = \text{Some}(m, D) \Longrightarrow P \vdash C \preceq^* D$

$\langle proof \rangle$

lemma *sees-methods-idemp*:

assumes $C \text{ methods}: P \vdash C \text{ sees-methods } Mm$

shows $\bigwedge m D. Mm M = \text{Some}(m, D) \Longrightarrow$

$\exists Mm'. (P \vdash D \text{ sees-methods } Mm') \wedge Mm' M = \text{Some}(m, D)\langle proof \rangle$

lemma *sees-methods-decl-mono*:

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

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

$\exists Mm' Mm_2. P \vdash C' \text{ sees-methods } Mm' \wedge Mm' = Mm ++ Mm_2 \wedge$
 $(\forall M m D. Mm_2 M = \text{Some}(m, D) \longrightarrow P \vdash D \preceq^* C)\langle proof \rangle$

definition *Method* :: $'m \text{ prog} \Rightarrow \text{cname} \Rightarrow \text{mname} \Rightarrow \text{ty list} \Rightarrow \text{ty} \Rightarrow 'm \Rightarrow \text{cname} \Rightarrow \text{bool}$

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

where

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

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

definition *has-method* :: $'m \text{ prog} \Rightarrow \text{cname} \Rightarrow \text{mname} \Rightarrow \text{bool} (- \vdash - \text{has} - [51,0,51] 50)$

where

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

lemma sees-method-fun:

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

$\langle \text{proof} \rangle$

lemma sees-method-decl-above:

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

$\langle \text{proof} \rangle$

lemma visible-method-exists:

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

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

lemma sees-method-idemp:

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

$\langle \text{proof} \rangle$

lemma sees-method-decl-mono:

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

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

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

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

$\langle \text{proof} \rangle$

lemma sees-method-is-class:

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

2.4.4 Field lookup

inductive

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

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

where

$\text{has-fields-rec}:$

$\llbracket \text{class } P \ C = \text{Some}(D, fs, ms); C \neq \text{Object}; P \vdash D \text{ has-fields } \text{FDTs};$

$\text{FDTs}' = \text{map } (\lambda(F, T). ((F, C), T)) \ fs \ @ \ \text{FDTs} \rrbracket$

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

$| \text{has-fields-Object}:$

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

$\implies P \vdash \text{Object} \text{ has-fields } \text{FDTs}$

lemma has-fields-fun:

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

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

$\langle \text{proof} \rangle$

lemma all-fields-in-has-fields:

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

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

$\implies ((F, D), T) \in \text{set } \text{FDTs} \langle \text{proof} \rangle$

lemma has-fields-decl-above:

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

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

lemma subcls-notin-has-fields:

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

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

lemma *has-fields-mono-lem*:

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

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

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

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

where

$P \vdash C \text{ has } F:T \text{ in } D \equiv$

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

lemma *has-field-mono*:

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

shows $P \vdash C' \text{ has } F:T \text{ in } D \langle \text{proof} \rangle$

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

where

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

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

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

lemma *map-of-remap-SomeD*:

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

lemma *has-visible-field*:

$P \vdash C \text{ sees } F:T \text{ in } D \implies P \vdash C \text{ has } F:T \text{ in } D \langle \text{proof} \rangle$

lemma *sees-field-fun*:

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

lemma *sees-field-decl-above*:

$P \vdash C \text{ sees } F:T \text{ in } D \implies P \vdash C \preceq^* D \langle \text{proof} \rangle$

lemma *sees-field-idemp*:

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

shows $P \vdash D \text{ sees } F:T \text{ in } D \langle \text{proof} \rangle$

2.4.5 Functional lookup

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

where

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

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

where

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

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

where

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

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

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

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

2.4.6 Code generator setup

code-pred

(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}$, $i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$)
subcls1p

$\langle \text{proof} \rangle$

declare *subcls1-def* [*code-pred-def*]

code-pred

(*modes*: $i \Rightarrow i \times o \Rightarrow \text{bool}$, $i \Rightarrow i \times i \Rightarrow \text{bool}$)
 [*inductify*]
subcls1

$\langle \text{proof} \rangle$

definition *subcls'* **where** *subcls'* $G = (\text{subcls1p } G) \hat{**}$

code-pred

(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}$, $i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$)
 [*inductify*]
subcls'

$\langle \text{proof} \rangle$

lemma *subcls-conv-subcls'* [*code-unfold*]:

$(\text{subcls1 } G) \hat{*} = \{(C, D). \text{subcls}' G \ C \ D\}$
 $\langle \text{proof} \rangle$

code-pred

(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}$)
widen

$\langle \text{proof} \rangle$

code-pred

(*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$)
Fields

$\langle \text{proof} \rangle$

lemma *has-field-code* [*code-pred-intro*]:

$\llbracket P \vdash C \text{ has-fields } FDTs; \text{map-of } FDTs (F, D) = \lfloor T \rfloor \rrbracket$
 $\implies P \vdash C \text{ has } F:T \text{ in } D$

$\langle \text{proof} \rangle$

code-pred

(*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow \text{bool}$, $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}$)
has-field

$\langle \text{proof} \rangle$

lemma *sees-field-code* [*code-pred-intro*]:

$\llbracket P \vdash C \text{ has-fields } FDTs; \text{map-of } (\text{map } (\lambda((F, D), T). (F, D, T)) \ FDTs) \ F = \lfloor (D, T) \rfloor \rrbracket$
 $\implies P \vdash C \text{ sees } F:T \text{ in } D$

$\langle \text{proof} \rangle$

code-pred

(modes: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow \text{bool}$, $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow i \Rightarrow \text{bool}$,
 $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$, $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}$)
 sees-field
 ⟨proof⟩

code-pred

(modes: $i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$)
 Methods
 ⟨proof⟩

lemma *Method-code* [code-pred-intro]:

$\llbracket P \vdash C \text{ sees-methods } Mm; Mm \ M = \llbracket ((Ts, T, m), D) \rrbracket \rrbracket$
 $\implies P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } D$
 ⟨proof⟩

code-pred

(modes: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow \text{bool}$,
 $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}$)
 Method
 ⟨proof⟩

lemma *eval-Method-i-i-i-o-o-o-o-conv*:

$\text{Predicate.eval } (\text{Method-i-i-i-o-o-o-o } P \ C \ M) = (\lambda(Ts, T, m, D). P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } D)$
 ⟨proof⟩

lemma *method-code* [code]:

$\text{method } P \ C \ M =$
 $\text{Predicate.the } (\text{Predicate.bind } (\text{Method-i-i-i-o-o-o-o } P \ C \ M) (\lambda(Ts, T, m, D). \text{Predicate.single } (D, Ts,$
 $T, m)))$
 ⟨proof⟩

lemma *eval-Fields-conv*:

$\text{Predicate.eval } (\text{Fields-i-i-o } P \ C) = (\lambda FDTs. P \vdash C \text{ has-fields } FDTs)$
 ⟨proof⟩

lemma *fields-code* [code]:

$\text{fields } P \ C = \text{Predicate.the } (\text{Fields-i-i-o } P \ C)$
 ⟨proof⟩

lemma *eval-sees-field-i-i-i-o-o-conv*:

$\text{Predicate.eval } (\text{sees-field-i-i-i-o-o } P \ C \ F) = (\lambda(T, D). P \vdash C \text{ sees } F:T \text{ in } D)$
 ⟨proof⟩

lemma *eval-sees-field-i-i-i-o-i-conv*:

$\text{Predicate.eval } (\text{sees-field-i-i-i-o-i } P \ C \ F \ D) = (\lambda T. P \vdash C \text{ sees } F:T \text{ in } D)$
 ⟨proof⟩

lemma *field-code* [code]:

$\text{field } P \ C \ F = \text{Predicate.the } (\text{Predicate.bind } (\text{sees-field-i-i-i-o-o } P \ C \ F) (\lambda(T, D). \text{Predicate.single}$
 $(D, T)))$
 ⟨proof⟩

2.5 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* :: *val* \Rightarrow *int* **where**

the-Intg (*Intg i*) = *i*

primrec *the-Addr* :: *val* \Rightarrow *addr* **where**

the-Addr (*Addr a*) = *a*

primrec *default-val* :: *ty* \Rightarrow *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

2.6 Objects and the Heap

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

2.6.1 Objects

type-synonym

fields = *vname* \times *cname* \rightarrow *val* — field name, defining class, value

type-synonym

obj = *cname* \times *fields* — class instance with class name and fields

definition *obj-ty* :: *obj* \Rightarrow *ty*

where

obj-ty obj \equiv *Class (fst obj)*

definition *init-fields* :: ((*vname* \times *cname*) \times *ty*) *list* \Rightarrow *fields*

where

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

— a new, blank object with default values in all fields:

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

where

blank P C \equiv (*C, init-fields (fields P C)*)

lemma [*simp*]: *obj-ty (C, fs)* = *Class C* {*proof*}

2.6.2 Heap

type-synonym $heap = addr \rightarrow obj$

abbreviation

$cname-of :: heap \Rightarrow addr \Rightarrow cname$ **where**
 $cname-of\ hp\ a == fst\ (the\ (hp\ a))$

definition $new-Addr :: heap \Rightarrow addr\ option$

where

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

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

where

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

definition $heqt :: heap \Rightarrow heap \Rightarrow bool\ (- \trianglelefteq - [51,51]\ 50)$

where

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

primrec $typeof-h :: heap \Rightarrow val \Rightarrow ty\ option\ (typeof-)$

where

$typeof_h\ Unit = Some\ Void$
 $|\ typeof_h\ Null = Some\ NT$
 $| \ typeof_h\ (Bool\ b) = Some\ Boolean$
 $| \ typeof_h\ (Intg\ i) = Some\ Integer$
 $| \ typeof_h\ (Addr\ a) = (case\ h\ a\ of\ None \Rightarrow None\ | \ Some(C,fs) \Rightarrow Some(Class\ C))$

lemma $new-Addr-SomeD:$

$new-Addr\ h = Some\ a \Longrightarrow h\ a = None$

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma $[simp]: (typeof_h\ v = Some\ Integer) = (\exists i. v = Intg\ i) \langle proof \rangle$

lemma $[simp]: (typeof_h\ v = Some\ NT) = (v = Null)$

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma $[simp]: h\ a = Some(C,fs) \Longrightarrow typeof_{(h(a \mapsto (C,fs')))}\ v = typeof_h\ v$

$\langle proof \rangle$

For literal values the first parameter of $typeof$ can be set to $\lambda x. None$ because they do not contain addresses:

abbreviation

$typeof :: val \Rightarrow ty\ option$ **where**
 $typeof\ v == typeof-h\ Map.empty\ v$

lemma $typeof-lit-typeof:$

$typeof\ v = Some\ T \Longrightarrow typeof_h\ v = Some\ T$

$\langle proof \rangle$

lemma $typeof-lit-is-type:$

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

$\langle proof \rangle$

2.6.3 Heap extension \trianglelefteq

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

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

lemma *hext-refl* [*iff*]: $h \trianglelefteq h \langle \text{proof} \rangle$

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

lemma *hext-trans*: $\llbracket h \trianglelefteq h'; h' \trianglelefteq h'' \rrbracket \Longrightarrow h \trianglelefteq h'' \langle \text{proof} \rangle$

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

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

Code generator setup for *new-Addr*

definition *gen-new-Addr* :: *heap* \Rightarrow *addr* \Rightarrow *addr option*

where *gen-new-Addr* *h n* \equiv *if* $\exists a. a \geq n \wedge h a = \text{None}$ *then* *Some*(*LEAST* *a. a* $\geq n \wedge h a = \text{None}$) *else* *None*

lemma *new-Addr-code-code* [*code*]:

new-Addr *h* = *gen-new-Addr* *h* 0

$\langle \text{proof} \rangle$

lemma *gen-new-Addr-code* [*code*]:

gen-new-Addr *h n* = (*if* *h n* = *None* *then* *Some n* *else* *gen-new-Addr* *h* (*Suc n*))

$\langle \text{proof} \rangle$

end

2.7 Exceptions

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

definition *NullPointer* :: *cname*

where

NullPointer \equiv "*NullPointer*"

definition *ClassCast* :: *cname*

where

ClassCast \equiv "*ClassCast*"

definition *OutOfMemory* :: *cname*

where

OutOfMemory \equiv "*OutOfMemory*"

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

where

sys-xcpts \equiv {*NullPointer*, *ClassCast*, *OutOfMemory*}

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* *undefined*

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*)

definition *preallocated* :: *heap* \Rightarrow *bool*

where

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

2.7.1 System exceptions

lemma [*simp*]: *NullPointer* $\in \text{sys-xcpts} \wedge \text{OutOfMemory} \in \text{sys-xcpts} \wedge \text{ClassCast} \in \text{sys-xcpts}$ ⟨*proof*⟩

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

$\llbracket C \in \text{sys-xcpts}; P \text{ NullPointer}; P \text{ OutOfMemory}; P \text{ ClassCast} \rrbracket \Longrightarrow P C$ ⟨*proof*⟩

2.7.2 preallocated

lemma *preallocated-dom* [*simp*]:

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

lemma *preallocatedD*:

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

lemma *preallocatedE* [*elim?*]:

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

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

$\llbracket \text{preallocated } h; C \in \text{sys-xcpts} \rrbracket \Longrightarrow \text{cname-of } h (\text{addr-of-sys-xcpt } C) = C$ ⟨*proof*⟩

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

preallocated *h* $\Longrightarrow \text{typeof}_h (\text{Addr}(\text{addr-of-sys-xcpt } \text{ClassCast})) = \text{Some}(\text{Class } \text{ClassCast})$ ⟨*proof*⟩

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

preallocated *h* $\Longrightarrow \text{typeof}_h (\text{Addr}(\text{addr-of-sys-xcpt } \text{OutOfMemory})) = \text{Some}(\text{Class } \text{OutOfMemory})$ ⟨*proof*⟩

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

preallocated *h* $\Longrightarrow \text{typeof}_h (\text{Addr}(\text{addr-of-sys-xcpt } \text{NullPointer})) = \text{Some}(\text{Class } \text{NullPointer})$ ⟨*proof*⟩

lemma *preallocated-hext*:

$\llbracket \text{preallocated } h; h \trianglelefteq h' \rrbracket \Longrightarrow \text{preallocated } h'$ ⟨*proof*⟩

lemma *preallocated-start*:

preallocated (*start-heap* *P*)⟨*proof*⟩

end

2.8 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
- | $BinOp\ ('a\ exp)\ bop\ ('a\ exp)$ ($- \llcorner -$ - $[80,0,81]\ 80$) — binary operation
- | $Var\ 'a$ — local variable (incl. parameter)
- | $LAss\ 'a\ ('a\ exp)$ ($- := -$ $[90,90]\ 90$) — local assignment
- | $FAcc\ ('a\ exp)\ vname\ cname$ ($--\{-\}$ $[10,90,99]\ 90$) — field access
- | $FAss\ ('a\ exp)\ vname\ cname\ ('a\ exp)$ ($--\{-\}\ := -$ $[10,90,99,90]\ 90$) — field assignment
- | $Call\ ('a\ exp)\ mname\ ('a\ exp\ list)$ ($--\{-\}\ [90,99,0]\ 90$) — method call
- | $Block\ 'a\ ty\ ('a\ exp)$ ($\{-\}; -$)
- | $Seq\ ('a\ exp)\ ('a\ exp)$ ($- ; / -$ $[61,60]\ 60$)
- | $Cond\ ('a\ exp)\ ('a\ exp)\ ('a\ exp)$ ($if\ '(-)\ - / else -$ $[80,79,79]\ 70$)
- | $While\ ('a\ exp)\ ('a\ exp)$ ($while\ '(-)\ -$ $[80,79]\ 70$)
- | $throw\ ('a\ exp)$
- | $TryCatch\ ('a\ exp)\ cname\ 'a\ ('a\ exp)$ ($try\ - / catch\ '(-)\ -$ $[0,99,80,79]\ 70$)

type-synonym

$expr = vname\ exp$ — Jinja expression

type-synonym

$J\text{-}mb = vname\ list \times expr$ — Jinja method body: parameter names and expression

type-synonym

$J\text{-}prog = J\text{-}mb\ prog$ — Jinja program

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 $[simp]$:

$(binop(Add, v_1, v_2) = Some\ v) = (\exists i_1\ i_2. v_1 = Intg\ i_1 \wedge v_2 = Intg\ i_2 \wedge v = Intg\ (i_1 + i_2)) \langle proof \rangle$

2.8.1 Syntactic sugar

abbreviation $(input)$

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

abbreviation $unit$ **where** $unit == Val\ Unit$

abbreviation $null$ **where** $null == Val\ Null$

abbreviation $addr\ a == Val(Addr\ a)$

abbreviation $true == Val(Bool\ True)$

abbreviation $false == Val(Bool\ False)$

abbreviation

$Throw :: addr \Rightarrow 'a\ exp$ **where**
 $Throw\ a == throw(Val(Addr\ a))$

abbreviation

$THROW :: cname \Rightarrow 'a\ exp$ **where**
 $THROW\ xc == Throw(addr\ of\ sys\ xcpt\ xc)$

2.8.2 Free Variables

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

```

   $fv(\text{new } C) = \{\}$ 
|  $fv(\text{Cast } C \ e) = fv \ e$ 
|  $fv(\text{Val } v) = \{\}$ 
|  $fv(e_1 \ll \text{bop} \gg e_2) = fv \ e_1 \cup fv \ e_2$ 
|  $fv(\text{Var } V) = \{V\}$ 
|  $fv(\text{LAss } V \ e) = \{V\} \cup fv \ e$ 
|  $fv(e \cdot F\{D\}) = fv \ e$ 
|  $fv(e_1 \cdot F\{D\};=e_2) = fv \ e_1 \cup fv \ e_2$ 
|  $fv(e \cdot M(es)) = fv \ e \cup fvs \ es$ 
|  $fv(\{V:T; e\}) = fv \ e - \{V\}$ 
|  $fv(e_1;;e_2) = fv \ e_1 \cup fv \ e_2$ 
|  $fv(\text{if } (b) \ e_1 \ \text{else } e_2) = fv \ b \cup fv \ e_1 \cup fv \ e_2$ 
|  $fv(\text{while } (b) \ e) = fv \ b \cup fv \ e$ 
|  $fv(\text{throw } e) = fv \ e$ 
|  $fv(\text{try } e_1 \ \text{catch}(C \ V) \ e_2) = fv \ e_1 \cup (fv \ e_2 - \{V\})$ 
|  $fvs(\[]) = \{\}$ 
|  $fvs(e\#es) = fv \ e \cup fvs \ es$ 

```

lemma $[\text{simp}]$: $fvs(es_1 \ @ \ es_2) = fvs \ es_1 \cup fvs \ es_2$ $\langle \text{proof} \rangle$

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

end

2.9 Program State

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

type-synonym

$locals = \text{vname} \rightarrow \text{val}$ — local vars, incl. params and “this”

type-synonym

$state = \text{heap} \times \text{locals}$

definition $hp :: \text{state} \Rightarrow \text{heap}$

where

$hp \equiv \text{fst}$

definition $lcl :: \text{state} \Rightarrow \text{locals}$

where

$lcl \equiv \text{snd}$

end

2.10 Big Step Semantics

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

inductive

$eval :: J\text{-prog} \Rightarrow \text{expr} \Rightarrow \text{state} \Rightarrow \text{expr} \Rightarrow \text{state} \Rightarrow \text{bool}$
 $(- \vdash ((1\langle -,/- \rangle) \Rightarrow / (1\langle -,/- \rangle))) [51,0,0,0,0] \ 81$

and $evals :: J\text{-prog} \Rightarrow \text{expr list} \Rightarrow \text{state} \Rightarrow \text{state} \Rightarrow \text{expr list} \Rightarrow \text{bool}$
 $(- \vdash ((1\langle -,/- \rangle) [\Rightarrow] / (1\langle -,/- \rangle))) [51,0,0,0,0] \ 81$

for $P :: J\text{-prog}$

where

New:

$$\begin{aligned} & \llbracket \text{new-Addr } h = \text{Some } a; P \vdash C \text{ has-fields FDTs}; h' = h(a \mapsto (C, \text{init-fields FDTs})) \rrbracket \\ & \implies P \vdash \langle \text{new } C, (h, l) \rangle \Rightarrow \langle \text{addr } a, (h', l) \rangle \end{aligned}$$

| *NewFail:*

$$\begin{aligned} & \text{new-Addr } h = \text{None} \implies \\ & P \vdash \langle \text{new } C, (h, l) \rangle \Rightarrow \langle \text{THROW OutOfMemory}, (h, l) \rangle \end{aligned}$$

| *Cast:*

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

| *CastNull:*

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

| *CastFail:*

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

| *CastThrow:*

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

| *Val:*

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

| *BinOp:*

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

| *BinOpThrow1:*

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

| *BinOpThrow2:*

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

| *Var:*

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

| *LAss:*

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

| *LAssThrow:*

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

| *FAcc:*

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

$$\Longrightarrow P \vdash \langle e \cdot F\{D\}, s_0 \rangle \Rightarrow \langle \text{Val } v, (h, l) \rangle$$

| *FAccNull*:

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

| *FAccThrow*:

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

| *FAss*:

$$\begin{aligned} \llbracket P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2) \rangle; \\ h_2 \ a = \text{Some}(C, fs); fs' = fs((F, D) \mapsto v); h_2' = h_2(a \mapsto (C, fs')) \rrbracket \\ \Longrightarrow P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{unit}, (h_2', l_2) \rangle \end{aligned}$$

| *FAssNull*:

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

| *FAssThrow1*:

$$\begin{aligned} P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle &\Longrightarrow \\ P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \end{aligned}$$

| *FAssThrow2*:

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

| *CallObjThrow*:

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

| *CallParamsThrow*:

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

| *CallNull*:

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

| *Call*:

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

| *Block*:

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

| *Seq*:

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

| *SeqThrow*:

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

| *CondT*:

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

| *CondF*:

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

| *CondThrow*:

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

| *WhileF*:

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

| *WhileT*:

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

| *WhileCondThrow*:

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

| *WhileBodyThrow*:

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

| *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*:

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

| *TryThrow*:

$$\llbracket P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{Throw } a, (h_1, l_1) \rangle; h_1 a = \text{Some}(D, fs); \neg P \vdash D \preceq^* C \rrbracket \\ \Longrightarrow P \vdash \langle \text{try } e_1 \text{ catch}(C V) e_2, s_0 \rangle \Rightarrow \langle \text{Throw } a, (h_1, l_1) \rangle$$
| *Nil*:
$$P \vdash \langle [], s \rangle [\Rightarrow] \langle [], s \rangle$$
| *Cons*:
$$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash \langle es, s_1 \rangle [\Rightarrow] \langle es', s_2 \rangle \rrbracket \\ \Longrightarrow P \vdash \langle e \# es, s_0 \rangle [\Rightarrow] \langle \text{Val } v \# es', s_2 \rangle$$
| *ConsThrow*:
$$P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow \\ P \vdash \langle e \# es, s_0 \rangle [\Rightarrow] \langle \text{throw } e' \# es, s_1 \rangle$$

2.10.1 Final expressions

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

where

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

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

where

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

lemma [*simp*]: *final*(Val v)<proof>

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

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

lemma [*iff*]: *finals* []<proof>

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

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

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

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

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

lemma *eval-final*: $P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \Longrightarrow \text{final } e'$

and *evals-final*: $P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \Longrightarrow \text{finals } es'$ <proof>

lemma *eval-lcl-incr*: $P \vdash \langle e, (h_0, l_0) \rangle \Rightarrow \langle e', (h_1, l_1) \rangle \Longrightarrow \text{dom } l_0 \subseteq \text{dom } l_1$

and *evals-lcl-incr*: $P \vdash \langle es, (h_0, l_0) \rangle [\Rightarrow] \langle es', (h_1, l_1) \rangle \Longrightarrow \text{dom } l_0 \subseteq \text{dom } l_1$ <proof>

Only used later, in the small to big translation, but is already a good sanity check:

lemma *eval-finalId*: $\text{final } e \Longrightarrow P \vdash \langle e, s \rangle \Rightarrow \langle e, s \rangle$ <proof>

lemma *eval-finalsId*:

assumes *finals*: *finals* es **shows** $P \vdash \langle es, s \rangle [\Rightarrow] \langle es, s \rangle$ <proof>

theorem *eval-hext*: $P \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle \Longrightarrow h \preceq h'$

and *evals-hext*: $P \vdash \langle es, (h, l) \rangle [\Rightarrow] \langle es', (h', l') \rangle \Longrightarrow h \preceq h'$ <proof>

end

2.11 Small Step Semantics

```

theory SmallStep
imports Expr State
begin

fun blocks :: vname list * ty list * val list * expr  $\Rightarrow$  expr
where
  blocks(V#Vs, T#Ts, v#vs, e) = { V:T := Val v; blocks(Vs, Ts, vs, e) }
  | blocks([], [], [], e) = e

lemmas blocks-induct = blocks.induct[split-format (complete)]

lemma [simp]:
  [| size vs = size Vs; size Ts = size Vs |]  $\Longrightarrow$  fv(blocks(Vs, Ts, vs, e)) = fv e - set Vs⟨proof⟩

definition assigned :: vname  $\Rightarrow$  expr  $\Rightarrow$  bool
where
  assigned V e  $\equiv$   $\exists v e'. e = (V := Val v;; e')$ 

inductive-set
  red :: J-prog  $\Rightarrow$  ((expr  $\times$  state)  $\times$  (expr  $\times$  state)) set
  and reds :: J-prog  $\Rightarrow$  ((expr list  $\times$  state)  $\times$  (expr list  $\times$  state)) set
  and red' :: J-prog  $\Rightarrow$  expr  $\Rightarrow$  state  $\Rightarrow$  expr  $\Rightarrow$  state  $\Rightarrow$  bool
    (-  $\vdash$  ((1⟨-, -⟩)  $\rightarrow$ / (1⟨-, -⟩)) [51, 0, 0, 0, 0] 81)
  and reds' :: J-prog  $\Rightarrow$  expr list  $\Rightarrow$  state  $\Rightarrow$  expr list  $\Rightarrow$  state  $\Rightarrow$  bool
    (-  $\vdash$  ((1⟨-, -⟩) [ $\rightarrow$ ]/ (1⟨-, -⟩)) [51, 0, 0, 0, 0] 81)
  for P :: J-prog
where

  P  $\vdash$  ⟨e, s⟩  $\rightarrow$  ⟨e', s'⟩  $\equiv$  ((e, s), e', s')  $\in$  red P
  | P  $\vdash$  ⟨es, s⟩ [ $\rightarrow$ ] ⟨es', s'⟩  $\equiv$  ((es, s), es', s')  $\in$  reds P

  | RedNew:
    [| new-Addr h = Some a; P  $\vdash$  C has-fields FDTs; h' = h(a $\mapsto$ (C, init-fields FDTs)) |]
     $\Longrightarrow$  P  $\vdash$  ⟨new C, (h, l)⟩  $\rightarrow$  ⟨addr a, (h', l)⟩

  | RedNewFail:
    new-Addr h = None  $\Longrightarrow$ 
    P  $\vdash$  ⟨new C, (h, l)⟩  $\rightarrow$  ⟨THROW OutOfMemory, (h, l)⟩

  | CastRed:
    P  $\vdash$  ⟨e, s⟩  $\rightarrow$  ⟨e', s'⟩  $\Longrightarrow$ 
    P  $\vdash$  ⟨Cast C e, s⟩  $\rightarrow$  ⟨Cast C e', s'⟩

  | RedCastNull:
    P  $\vdash$  ⟨Cast C null, s⟩  $\rightarrow$  ⟨null, s⟩

  | RedCast:
    [| hp s a = Some(D, fs); P  $\vdash$  D  $\preceq^*$  C |]
     $\Longrightarrow$  P  $\vdash$  ⟨Cast C (addr a), s⟩  $\rightarrow$  ⟨addr a, s⟩

  | RedCastFail:
    [| hp s a = Some(D, fs);  $\neg$  P  $\vdash$  D  $\preceq^*$  C |]

```

$$\Longrightarrow P \vdash \langle \text{Cast } C \text{ (addr } a), s \rangle \rightarrow \langle \text{THROW ClassCast}, s \rangle$$

| *BinOpRed1*:

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \Longrightarrow \\ P \vdash \langle e \text{ «bop» } e_2, s \rangle \rightarrow \langle e' \text{ «bop» } e_2, s' \rangle$$

| *BinOpRed2*:

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \Longrightarrow \\ P \vdash \langle (\text{Val } v_1) \text{ «bop» } e, s \rangle \rightarrow \langle (\text{Val } v_1) \text{ «bop» } e', s' \rangle$$

| *RedBinOp*:

$$\text{binop}(\text{bop}, v_1, v_2) = \text{Some } v \Longrightarrow \\ P \vdash \langle (\text{Val } v_1) \text{ «bop» } (\text{Val } v_2), s \rangle \rightarrow \langle \text{Val } v, s \rangle$$

| *RedVar*:

$$\text{lcl } s \text{ } V = \text{Some } v \Longrightarrow \\ P \vdash \langle \text{Var } V, s \rangle \rightarrow \langle \text{Val } v, s \rangle$$

| *LAssRed*:

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \Longrightarrow \\ P \vdash \langle V := e, s \rangle \rightarrow \langle V := e', s' \rangle$$

| *RedLAss*:

$$P \vdash \langle V := (\text{Val } v), (h, l) \rangle \rightarrow \langle \text{unit}, (h, l(V \mapsto v)) \rangle$$

| *FAccRed*:

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \Longrightarrow \\ P \vdash \langle e \cdot F\{D\}, s \rangle \rightarrow \langle e' \cdot F\{D\}, s' \rangle$$

| *RedFAcc*:

$$\llbracket \text{hp } s \text{ } a = \text{Some}(C, fs); fs(F, D) = \text{Some } v \rrbracket \\ \Longrightarrow P \vdash \langle (\text{addr } a) \cdot F\{D\}, s \rangle \rightarrow \langle \text{Val } v, s \rangle$$

| *RedFAccNull*:

$$P \vdash \langle \text{null} \cdot F\{D\}, s \rangle \rightarrow \langle \text{THROW NullPointer}, s \rangle$$

| *FAssRed1*:

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \Longrightarrow \\ P \vdash \langle e \cdot F\{D\} := e_2, s \rangle \rightarrow \langle e' \cdot F\{D\} := e_2, s' \rangle$$

| *FAssRed2*:

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \Longrightarrow \\ P \vdash \langle \text{Val } v \cdot F\{D\} := e, s \rangle \rightarrow \langle \text{Val } v \cdot F\{D\} := e', s' \rangle$$

| *RedFAss*:

$$h \text{ } a = \text{Some}(C, fs) \Longrightarrow \\ P \vdash \langle (\text{addr } a) \cdot F\{D\} := (\text{Val } v), (h, l) \rangle \rightarrow \langle \text{unit}, (h(a \mapsto (C, fs((F, D) \mapsto v))), l) \rangle$$

| *RedFAssNull*:

$$P \vdash \langle \text{null} \cdot F\{D\} := \text{Val } v, s \rangle \rightarrow \langle \text{THROW NullPointer}, s \rangle$$

| *CallObj*:

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \Longrightarrow \\ P \vdash \langle e \cdot M(es), s \rangle \rightarrow \langle e' \cdot M(es), s' \rangle$$

- | *CallParams*:
 $P \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle \Longrightarrow$
 $P \vdash \langle (Val\ v) \cdot M(es), s \rangle \rightarrow \langle (Val\ v) \cdot M(es'), s' \rangle$
- | *RedCall*:
 $\llbracket hp\ s\ a = Some(C, fs); P \vdash C\ sees\ M:Ts \rightarrow T = (pns, body)\ in\ D; size\ vs = size\ pns; size\ Ts = size\ pns \rrbracket$
 $\Longrightarrow P \vdash \langle (addr\ a) \cdot M(map\ Val\ vs), s \rangle \rightarrow \langle blocks(this\#pns, Class\ D\#Ts, Addr\ a\#vs, body), s \rangle$
- | *RedCallNull*:
 $P \vdash \langle null \cdot M(map\ Val\ vs), s \rangle \rightarrow \langle THROW\ NullPointer, s \rangle$
- | *BlockRedNone*:
 $\llbracket P \vdash \langle e, (h, l(V := None)) \rangle \rightarrow \langle e', (h', l') \rangle; l'\ V = None; \neg assigned\ V\ e \rrbracket$
 $\Longrightarrow P \vdash \langle \{V:T; e\}, (h, l) \rangle \rightarrow \langle \{V:T; e'\}, (h', l'(V := l\ V)) \rangle$
- | *BlockRedSome*:
 $\llbracket P \vdash \langle e, (h, l(V := None)) \rangle \rightarrow \langle e', (h', l') \rangle; l'\ V = Some\ v; \neg assigned\ V\ e \rrbracket$
 $\Longrightarrow P \vdash \langle \{V:T; e\}, (h, l) \rangle \rightarrow \langle \{V:T := Val\ v; e'\}, (h', l'(V := l\ V)) \rangle$
- | *InitBlockRed*:
 $\llbracket P \vdash \langle e, (h, l(V \mapsto v)) \rangle \rightarrow \langle e', (h', l') \rangle; l'\ V = Some\ v' \rrbracket$
 $\Longrightarrow P \vdash \langle \{V:T := Val\ v; e\}, (h, l) \rangle \rightarrow \langle \{V:T := Val\ v'; e'\}, (h', l'(V := l\ V)) \rangle$
- | *RedBlock*:
 $P \vdash \langle \{V:T; Val\ u\}, s \rangle \rightarrow \langle Val\ u, s \rangle$
- | *RedInitBlock*:
 $P \vdash \langle \{V:T := Val\ v; Val\ u\}, s \rangle \rightarrow \langle Val\ u, s \rangle$
- | *SeqRed*:
 $P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \Longrightarrow$
 $P \vdash \langle e;;e_2, s \rangle \rightarrow \langle e';;e_2, s' \rangle$
- | *RedSeq*:
 $P \vdash \langle (Val\ v);;e_2, s \rangle \rightarrow \langle e_2, s \rangle$
- | *CondRed*:
 $P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \Longrightarrow$
 $P \vdash \langle if\ (e)\ e_1\ else\ e_2, s \rangle \rightarrow \langle if\ (e')\ e_1\ else\ e_2, s' \rangle$
- | *RedCondT*:
 $P \vdash \langle if\ (true)\ e_1\ else\ e_2, s \rangle \rightarrow \langle e_1, s \rangle$
- | *RedCondF*:
 $P \vdash \langle if\ (false)\ e_1\ else\ e_2, s \rangle \rightarrow \langle e_2, s \rangle$
- | *RedWhile*:
 $P \vdash \langle while(b)\ c, s \rangle \rightarrow \langle if(b)\ (c;;while(b)\ c)\ else\ unit, s \rangle$
- | *ThrowRed*:
 $P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \Longrightarrow$
 $P \vdash \langle throw\ e, s \rangle \rightarrow \langle throw\ e', s' \rangle$

- | *RedThrowNull*:
 $P \vdash \langle \text{throw null}, s \rangle \rightarrow \langle \text{THROW NullPointer}, s \rangle$
- | *TryRed*:
 $P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$
 $P \vdash \langle \text{try } e \text{ catch}(C V) e_2, s \rangle \rightarrow \langle \text{try } e' \text{ catch}(C V) e_2, s' \rangle$
- | *RedTry*:
 $P \vdash \langle \text{try } (\text{Val } v) \text{ catch}(C V) e_2, s \rangle \rightarrow \langle \text{Val } v, s \rangle$
- | *RedTryCatch*:
 $\llbracket \text{hp } s \text{ a} = \text{Some}(D, fs); P \vdash D \preceq^* C \rrbracket$
 $\implies P \vdash \langle \text{try } (\text{Throw } a) \text{ catch}(C V) e_2, s \rangle \rightarrow \langle \{V: \text{Class } C := \text{addr } a; e_2\}, s \rangle$
- | *RedTryFail*:
 $\llbracket \text{hp } s \text{ a} = \text{Some}(D, fs); \neg P \vdash D \preceq^* C \rrbracket$
 $\implies P \vdash \langle \text{try } (\text{Throw } a) \text{ catch}(C V) e_2, s \rangle \rightarrow \langle \text{Throw } a, s \rangle$
- | *ListRed1*:
 $P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$
 $P \vdash \langle e \# es, s \rangle [\rightarrow] \langle e' \# es, s' \rangle$
- | *ListRed2*:
 $P \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle \implies$
 $P \vdash \langle \text{Val } v \# es, s \rangle [\rightarrow] \langle \text{Val } v \# es', s' \rangle$

— Exception propagation

- | *CastThrow*: $P \vdash \langle \text{Cast } C (\text{throw } e), s \rangle \rightarrow \langle \text{throw } e, s \rangle$
- | *BinOpThrow1*: $P \vdash \langle (\text{throw } e) \ll \text{bop} \gg e_2, s \rangle \rightarrow \langle \text{throw } e, s \rangle$
- | *BinOpThrow2*: $P \vdash \langle (\text{Val } v_1) \ll \text{bop} \gg (\text{throw } e), s \rangle \rightarrow \langle \text{throw } e, s \rangle$
- | *LAssThrow*: $P \vdash \langle V := (\text{throw } e), s \rangle \rightarrow \langle \text{throw } e, s \rangle$
- | *FAccThrow*: $P \vdash \langle (\text{throw } e) \cdot F\{D\}, s \rangle \rightarrow \langle \text{throw } e, s \rangle$
- | *FAssThrow1*: $P \vdash \langle (\text{throw } e) \cdot F\{D\} := e_2, s \rangle \rightarrow \langle \text{throw } e, s \rangle$
- | *FAssThrow2*: $P \vdash \langle \text{Val } v \cdot F\{D\} := (\text{throw } e), s \rangle \rightarrow \langle \text{throw } e, s \rangle$
- | *CallThrowObj*: $P \vdash \langle (\text{throw } e) \cdot M(es), s \rangle \rightarrow \langle \text{throw } e, s \rangle$
- | *CallThrowParams*: $\llbracket es = \text{map Val } vs @ \text{throw } e \# es' \rrbracket \implies P \vdash \langle (\text{Val } v) \cdot M(es), s \rangle \rightarrow \langle \text{throw } e, s \rangle$
- | *BlockThrow*: $P \vdash \langle \{V:T; \text{Throw } a\}, s \rangle \rightarrow \langle \text{Throw } a, s \rangle$
- | *InitBlockThrow*: $P \vdash \langle \{V:T := \text{Val } v; \text{Throw } a\}, s \rangle \rightarrow \langle \text{Throw } a, s \rangle$
- | *SeqThrow*: $P \vdash \langle (\text{throw } e); e_2, s \rangle \rightarrow \langle \text{throw } e, s \rangle$
- | *CondThrow*: $P \vdash \langle \text{if } (\text{throw } e) e_1 \text{ else } e_2, s \rangle \rightarrow \langle \text{throw } e, s \rangle$
- | *ThrowThrow*: $P \vdash \langle \text{throw}(\text{throw } e), s \rangle \rightarrow \langle \text{throw } e, s \rangle$

2.11.1 The reflexive transitive closure

abbreviation

Step :: $J\text{-prog} \Rightarrow \text{expr} \Rightarrow \text{state} \Rightarrow \text{expr} \Rightarrow \text{state} \Rightarrow \text{bool}$
 $(- \vdash ((1 \langle -, / - \rangle) \rightarrow^* / (1 \langle -, / - \rangle))) [51, 0, 0, 0, 0] \ 81$
where $P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \equiv ((e, s), e', s') \in (\text{red } P)^*$

abbreviation

Steps :: $J\text{-prog} \Rightarrow \text{expr list} \Rightarrow \text{state} \Rightarrow \text{expr list} \Rightarrow \text{state} \Rightarrow \text{bool}$
 $(- \vdash ((1 \langle -, / - \rangle) [\rightarrow]^* / (1 \langle -, / - \rangle))) [51, 0, 0, 0, 0] \ 81$

where $P \vdash \langle es, s \rangle [\rightarrow]^* \langle es', s' \rangle \equiv ((es, s), es', s') \in (reds\ P)^*$

lemma *converse-rtrancl-induct-red*[consumes 1]:

assumes $P \vdash \langle e, (h, l) \rangle \rightarrow^* \langle e', (h', l') \rangle$

and $\bigwedge e\ h\ l.\ R\ e\ h\ l\ e\ h\ l$

and $\bigwedge e_0\ h_0\ l_0\ e_1\ h_1\ l_1\ e'\ h'\ l'.$

$\llbracket P \vdash \langle e_0, (h_0, l_0) \rangle \rightarrow \langle e_1, (h_1, l_1) \rangle; R\ e_1\ h_1\ l_1\ e'\ h'\ l' \rrbracket \Longrightarrow R\ e_0\ h_0\ l_0\ e'\ h'\ l'$
shows $R\ e\ h\ l\ e'\ h'\ l'$ *(proof)*

2.11.2 Some easy lemmas

lemma [*iff*]: $\neg P \vdash \langle [], s \rangle [\rightarrow] \langle es', s' \rangle$ *(proof)*

lemma [*iff*]: $\neg P \vdash \langle Val\ v, s \rangle \rightarrow \langle e', s' \rangle$ *(proof)*

lemma [*iff*]: $\neg P \vdash \langle Throw\ a, s \rangle \rightarrow \langle e', s' \rangle$ *(proof)*

lemma *red-heat-incr*: $P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \Longrightarrow h \trianglelefteq h'$

and *reds-heat-incr*: $P \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \Longrightarrow h \trianglelefteq h'$ *(proof)*

lemma *red-lcl-incr*: $P \vdash \langle e, (h_0, l_0) \rangle \rightarrow \langle e', (h_1, l_1) \rangle \Longrightarrow dom\ l_0 \subseteq dom\ l_1$

and $P \vdash \langle es, (h_0, l_0) \rangle [\rightarrow] \langle es', (h_1, l_1) \rangle \Longrightarrow dom\ l_0 \subseteq dom\ l_1$ *(proof)*

lemma *red-lcl-add*: $P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \Longrightarrow (\bigwedge l_0.\ P \vdash \langle e, (h, l_0++l) \rangle \rightarrow \langle e', (h', l_0++l') \rangle)$

and $P \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \Longrightarrow (\bigwedge l_0.\ P \vdash \langle es, (h, l_0++l) \rangle [\rightarrow] \langle es', (h', l_0++l') \rangle)$ *(proof)*

lemma *Red-lcl-add*:

assumes $P \vdash \langle e, (h, l) \rangle \rightarrow^* \langle e', (h', l') \rangle$ **shows** $P \vdash \langle e, (h, l_0++l) \rangle \rightarrow^* \langle e', (h', l_0++l') \rangle$ *(proof)*

end

2.12 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 *SystemClasses* :: 'm cdecl list

where

$SystemClasses \equiv [ObjectC, NullPointerC, ClassCastC, OutOfMemoryC]$

end

2.13 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 $'m \text{ wf-mdecl-test} = 'm \text{ prog} \Rightarrow \text{cname} \Rightarrow 'm \text{ mdecl} \Rightarrow \text{bool}$

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

where

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

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

where

$\text{wf-mdecl } \text{wf-md } P \ C \equiv \lambda(M, Ts, T, mb).$

$(\forall T \in \text{set } Ts. \text{is-type } P \ T) \wedge \text{is-type } P \ T \wedge \text{wf-md } P \ C \ (M, Ts, T, mb)$

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

where

$\text{wf-cdecl } \text{wf-md } P \equiv \lambda(C, (D, fs, ms)).$

$(\forall f \in \text{set } fs. \text{wf-fdecl } P \ f) \wedge \text{distinct-fst } fs \wedge$

$(\forall m \in \text{set } ms. \text{wf-mdecl } \text{wf-md } P \ C \ m) \wedge \text{distinct-fst } ms \wedge$

$(C \neq \text{Object} \longrightarrow$

$\text{is-class } P \ D \wedge \neg P \vdash D \preceq^* C \wedge$

$(\forall (M, Ts, T, m) \in \text{set } ms.$

$\forall D' \ Ts' \ T' \ m'. P \vdash D \text{ sees } M:Ts' \rightarrow T' = m' \text{ in } D' \longrightarrow$
 $P \vdash Ts' [\leq] Ts \wedge P \vdash T \leq T')$

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

where

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

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

where

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

2.13.1 Well-formedness lemmas

lemma *class-wf*:

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

lemma *class-Object [simp]*:

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

lemma *is-class-Object* [*simp*]:

$wf\text{-prog } wf\text{-md } P \implies is\text{-class } P \text{ Object}\langle proof \rangle$

lemma *is-class-xcpt*:

$\llbracket C \in sys\text{-xcpts}; wf\text{-prog } wf\text{-md } P \rrbracket \implies is\text{-class } P \ C \langle proof \rangle$

lemma *subcls1-wfD*:

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

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

lemma *wf-cdecl-supD*:

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

lemma *subcls-asym*:

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

lemma *subcls-irrefl*:

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

lemma *acyclic-subcls1*:

$wf\text{-prog } wf\text{-md } P \implies acyclic \ (subcls1 \ P) \langle proof \rangle$

lemma *wf-subcls1*:

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

lemma *single-valued-subcls1*:

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

lemma *subcls-induct*:

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

lemma *subcls1-induct-aux*:

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

shows

$\llbracket \bigwedge C \ D \ fs \ ms.$

$\llbracket C \neq \text{Object}; is\text{-class } P \ C; class \ P \ C = \text{Some} \ (D,fs,ms) \wedge$

$wf\text{-cdecl } wf\text{-md } P \ (C,D,fs,ms) \wedge P \vdash C \prec^1 D \wedge is\text{-class } P \ D \wedge Q \ D \rrbracket \implies Q \ C \rrbracket$

$\implies Q \ C \langle proof \rangle$

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

$\llbracket wf\text{-prog } wf\text{-md } P; is\text{-class } P \ C; Q \ \text{Object};$

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

$\implies Q \ C \langle proof \rangle$

lemma *subcls-C-Object*:

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

shows $P \vdash C \preceq^* \ \text{Object} \langle proof \rangle$

lemma *is-type-pTs*:

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

shows $set \ Ts \subseteq types \ P \langle proof \rangle$

2.13.2 Well-formedness and method lookup

lemma sees-wf-mdecl:

assumes $wf: wf\text{-prog } wf\text{-md } P$ **and** $sees: P \vdash C \text{ sees } M:Ts \rightarrow T = m \text{ in } D$
shows $wf\text{-mdecl } wf\text{-md } P \ D \ (M, Ts, T, m) \langle proof \rangle$

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 $\forall D \ Ts \ T \ m. P \vdash C \text{ sees } M:Ts \rightarrow T = m \text{ in } D \longrightarrow$
 $(\exists D' \ Ts' \ T' \ m'. P \vdash C' \text{ sees } M:Ts' \rightarrow T' = m' \text{ in } D' \wedge P \vdash Ts \ [\leq] \ Ts' \wedge P \vdash T' \leq T) \langle proof \rangle$

lemma sees-method-mono2:

$\llbracket P \vdash C' \preceq^* C; wf\text{-prog } wf\text{-md } P;$
 $P \vdash C \text{ sees } M:Ts \rightarrow T = m \text{ in } D; P \vdash C' \text{ sees } M:Ts' \rightarrow T' = m' \text{ in } D' \rrbracket$
 $\Longrightarrow P \vdash Ts \ [\leq] \ Ts' \wedge P \vdash T' \leq T \langle proof \rangle$

lemma mdecls-visible:

assumes $wf: wf\text{-prog } wf\text{-md } P$ **and** $class: is\text{-class } P \ C$
shows $\bigwedge D \ fs \ ms. class \ P \ C = Some(D, fs, ms)$
 $\Longrightarrow \exists Mm. P \vdash C \text{ sees-methods } Mm \wedge (\forall (M, Ts, T, m) \in set \ ms. Mm \ M = Some((Ts, T, m), C)) \langle proof \rangle$

lemma mdecl-visible:

assumes $wf: wf\text{-prog } wf\text{-md } P$ **and** $C: (C, S, fs, ms) \in set \ P$ **and** $m: (M, Ts, T, m) \in set \ ms$
shows $P \vdash C \text{ sees } M:Ts \rightarrow T = m \text{ in } C \langle proof \rangle$

lemma Call-lemma:

assumes $sees: P \vdash C \text{ sees } M:Ts \rightarrow T = m \text{ in } D$ **and** $sub: P \vdash C' \preceq^* C$ **and** $wf: wf\text{-prog } wf\text{-md } P$
shows $\exists D' \ Ts' \ T' \ m'.$
 $P \vdash C' \text{ sees } M:Ts' \rightarrow T' = m' \text{ in } D' \wedge P \vdash Ts \ [\leq] \ Ts' \wedge P \vdash T' \leq T \wedge P \vdash C' \preceq^* D'$
 $\wedge is\text{-type } P \ T' \wedge (\forall T \in set \ Ts'. is\text{-type } P \ T) \wedge wf\text{-md } P \ D' \ (M, Ts', T', m') \langle proof \rangle$

lemma wf-prog-lift:

assumes $wf: wf\text{-prog } (\lambda P \ C \ bd. A \ P \ C \ bd) \ P$
and **rule:**
 $\bigwedge wf\text{-md } C \ M \ Ts \ C \ T \ m \ bd.$
 $wf\text{-prog } wf\text{-md } P \Longrightarrow$
 $P \vdash C \text{ sees } M:Ts \rightarrow T = m \text{ in } C \Longrightarrow$
 $set \ Ts \subseteq types \ P \Longrightarrow$
 $bd = (M, Ts, T, m) \Longrightarrow$
 $A \ P \ C \ bd \Longrightarrow$
 $B \ P \ C \ bd$
shows $wf\text{-prog } (\lambda P \ C \ bd. B \ P \ C \ bd) \ P \langle proof \rangle$

2.13.3 Well-formedness and field lookup

lemma wf-Fields-Ex:

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

lemma has-fields-types:

$\llbracket P \vdash C \text{ has-fields } FDTs; (FD, T) \in set \ FDTs; wf\text{-prog } wf\text{-md } P \rrbracket \Longrightarrow is\text{-type } P \ T \langle proof \rangle$

lemma sees-field-is-type:

$\llbracket P \vdash C \text{ sees } F:T \text{ in } D; wf\text{-prog } wf\text{-md } P \rrbracket \Longrightarrow is\text{-type } P \ T \langle proof \rangle$

lemma wf-syscls:

set SystemClasses \subseteq *set P* \implies *wf-syscls P*⟨*proof*⟩
end

2.14 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 $\lambda(M, Ts, T, (pns, body)).$
length Ts = *length pns* \wedge *distinct pns* \wedge *this* \notin *set pns* \wedge *fv body* \subseteq {*this*} \cup *set pns*

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

wwf-J-mdecl P C (M, Ts, T, pns, body) =
(*length Ts* = *length pns* \wedge *distinct pns* \wedge *this* \notin *set pns* \wedge *fv body* \subseteq {*this*} \cup *set pns*)⟨*proof*⟩

abbreviation

wwf-J-prog :: *J-prog* \Rightarrow *bool* **where**
wwf-J-prog == *wf-prog wwf-J-mdecl*

end

2.15 Equivalence of Big Step and Small Step Semantics

theory *Equivalence* **imports** *BigStep SmallStep WWellForm* **begin**

2.15.1 Small steps simulate big step

Cast

lemma *CastReds*:

$P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P \vdash \langle \text{Cast } C \ e, s \rangle \rightarrow^* \langle \text{Cast } C \ e', s' \rangle$ ⟨*proof*⟩

lemma *CastRedsNull*:

$P \vdash \langle e, s \rangle \rightarrow^* \langle \text{null}, s' \rangle \implies P \vdash \langle \text{Cast } C \ e, s \rangle \rightarrow^* \langle \text{null}, s' \rangle$ ⟨*proof*⟩

lemma *CastRedsAddr*:

$\llbracket P \vdash \langle e, s \rangle \rightarrow^* \langle \text{addr } a, s' \rangle; \text{hp } s' \ a = \text{Some}(D, \text{fs}); P \vdash D \preceq^* C \rrbracket \implies$
 $P \vdash \langle \text{Cast } C \ e, s \rangle \rightarrow^* \langle \text{addr } a, s' \rangle$ ⟨*proof*⟩

lemma *CastRedsFail*:

$\llbracket P \vdash \langle e, s \rangle \rightarrow^* \langle \text{addr } a, s' \rangle; \text{hp } s' \ a = \text{Some}(D, \text{fs}); \neg P \vdash D \preceq^* C \rrbracket \implies$
 $P \vdash \langle \text{Cast } C \ e, s \rangle \rightarrow^* \langle \text{THROW } \text{ClassCast}, s' \rangle$ ⟨*proof*⟩

lemma *CastRedsThrow*:

$\llbracket P \vdash \langle e, s \rangle \rightarrow^* \langle \text{throw } a, s' \rangle \rrbracket \implies P \vdash \langle \text{Cast } C \ e, s \rangle \rightarrow^* \langle \text{throw } a, s' \rangle$ ⟨*proof*⟩

LAss

lemma *LAssReds*:

$P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P \vdash \langle V := e, s \rangle \rightarrow^* \langle V := e', s' \rangle$ ⟨*proof*⟩

lemma *LAssRedsVal*:

$\llbracket P \vdash \langle e, s \rangle \rightarrow^* \langle \text{Val } v, (h', l') \rangle \rrbracket \implies P \vdash \langle V := e, s \rangle \rightarrow^* \langle \text{unit}, (h', l' (V \mapsto v)) \rangle$ ⟨*proof*⟩

lemma *LAssRedsThrow*:

$\llbracket P \vdash \langle e, s \rangle \rightarrow^* \langle \text{throw } a, s' \rangle \rrbracket \implies P \vdash \langle V := e, s \rangle \rightarrow^* \langle \text{throw } a, s' \rangle$ ⟨*proof*⟩

BinOp

lemma *BinOp1Reds*:

$$P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P \vdash \langle e \text{ «bop» } e_2, s \rangle \rightarrow^* \langle e' \text{ «bop» } e_2, s' \rangle \langle \text{proof} \rangle$$

lemma BinOp2Reds:

$$P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P \vdash \langle (\text{Val } v) \text{ «bop» } e, s \rangle \rightarrow^* \langle (\text{Val } v) \text{ «bop» } e', s' \rangle \langle \text{proof} \rangle$$

lemma BinOpRedsVal:

assumes $e_1\text{-steps}$: $P \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle \text{Val } v_1, s_1 \rangle$

and $e_2\text{-steps}$: $P \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle \text{Val } v_2, s_2 \rangle$

and op : $\text{binop}(\text{bop}, v_1, v_2) = \text{Some } v$

shows $P \vdash \langle e_1 \text{ «bop» } e_2, s_0 \rangle \rightarrow^* \langle \text{Val } v, s_2 \rangle \langle \text{proof} \rangle$

lemma BinOpRedsThrow1:

$$P \vdash \langle e, s \rangle \rightarrow^* \langle \text{throw } e', s' \rangle \implies P \vdash \langle e \text{ «bop» } e_2, s \rangle \rightarrow^* \langle \text{throw } e', s' \rangle \langle \text{proof} \rangle$$

lemma BinOpRedsThrow2:

assumes $e_1\text{-steps}$: $P \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle \text{Val } v_1, s_1 \rangle$

and $e_2\text{-steps}$: $P \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle \text{throw } e, s_2 \rangle$

shows $P \vdash \langle e_1 \text{ «bop» } e_2, s_0 \rangle \rightarrow^* \langle \text{throw } e, s_2 \rangle \langle \text{proof} \rangle$

F_{Acc}

lemma F_{Acc}Reds:

$$P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P \vdash \langle e \cdot F\{D\}, s \rangle \rightarrow^* \langle e' \cdot F\{D\}, s' \rangle \langle \text{proof} \rangle$$

lemma F_{Acc}RedsVal:

$$\llbracket P \vdash \langle e, s \rangle \rightarrow^* \langle \text{addr } a, s' \rangle; \text{hp } s' \ a = \text{Some}(C, fs); fs(F, D) = \text{Some } v \rrbracket$$

$$\implies P \vdash \langle e \cdot F\{D\}, s \rangle \rightarrow^* \langle \text{Val } v, s' \rangle \langle \text{proof} \rangle$$

lemma F_{Acc}RedsNull:

$$P \vdash \langle e, s \rangle \rightarrow^* \langle \text{null}, s' \rangle \implies P \vdash \langle e \cdot F\{D\}, s \rangle \rightarrow^* \langle \text{THROW NullPointer}, s' \rangle \langle \text{proof} \rangle$$

lemma F_{Acc}RedsThrow:

$$P \vdash \langle e, s \rangle \rightarrow^* \langle \text{throw } a, s' \rangle \implies P \vdash \langle e \cdot F\{D\}, s \rangle \rightarrow^* \langle \text{throw } a, s' \rangle \langle \text{proof} \rangle$$

F_{Ass}

lemma F_{Ass}Reds1:

$$P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P \vdash \langle e \cdot F\{D\} := e_2, s \rangle \rightarrow^* \langle e' \cdot F\{D\} := e_2, s' \rangle \langle \text{proof} \rangle$$

lemma F_{Ass}Reds2:

$$P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P \vdash \langle \text{Val } v \cdot F\{D\} := e, s \rangle \rightarrow^* \langle \text{Val } v \cdot F\{D\} := e', s' \rangle \langle \text{proof} \rangle$$

lemma F_{Ass}RedsVal:

assumes $e_1\text{-steps}$: $P \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle \text{addr } a, s_1 \rangle$

and $e_2\text{-steps}$: $P \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle \text{Val } v, (h_2, l_2) \rangle$

and hC : $\text{Some}(C, fs) = h_2 \ a$

shows $P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \rightarrow^* \langle \text{unit}, (h_2(a \mapsto (C, fs((F, D) \mapsto v))), l_2) \rangle \langle \text{proof} \rangle$

lemma F_{Ass}RedsNull:

assumes $e_1\text{-steps}$: $P \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle \text{null}, s_1 \rangle$

and $e_2\text{-steps}$: $P \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle \text{Val } v, s_2 \rangle$

shows $P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \rightarrow^* \langle \text{THROW NullPointer}, s_2 \rangle \langle \text{proof} \rangle$

lemma F_{Ass}RedsThrow1:

$$P \vdash \langle e, s \rangle \rightarrow^* \langle \text{throw } e', s' \rangle \implies P \vdash \langle e \cdot F\{D\} := e_2, s \rangle \rightarrow^* \langle \text{throw } e', s' \rangle \langle \text{proof} \rangle$$

lemma F_{Ass}RedsThrow2:

assumes $e_1\text{-steps}$: $P \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle \text{Val } v, s_1 \rangle$

and $e_2\text{-steps}$: $P \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle \text{throw } e, s_2 \rangle$

shows $P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \rightarrow^* \langle \text{throw } e, s_2 \rangle \langle \text{proof} \rangle$

;;

lemma SeqReds:

$$P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P \vdash \langle e;; e_2, s \rangle \rightarrow^* \langle e';; e_2, s' \rangle \langle \text{proof} \rangle$$

lemma SeqRedsThrow:

$$P \vdash \langle e, s \rangle \rightarrow^* \langle \text{throw } e', s' \rangle \implies P \vdash \langle e;; e_2, s \rangle \rightarrow^* \langle \text{throw } e', s' \rangle \langle \text{proof} \rangle$$

lemma *SeqReds2*:

assumes e_1 -steps: $P \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle \text{Val } v_1, s_1 \rangle$
and e_2 -steps: $P \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle e_2', s_2 \rangle$
shows $P \vdash \langle e_1;;e_2, s_0 \rangle \rightarrow^* \langle e_2', s_2 \rangle \langle \text{proof} \rangle$

If

lemma *CondReds*:

$P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s \rangle \rightarrow^* \langle \text{if } (e') e_1 \text{ else } e_2, s' \rangle \langle \text{proof} \rangle$

lemma *CondRedsThrow*:

$P \vdash \langle e, s \rangle \rightarrow^* \langle \text{throw } a, s' \rangle \implies P \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s \rangle \rightarrow^* \langle \text{throw } a, s' \rangle \langle \text{proof} \rangle$

lemma *CondReds2T*:

assumes e -steps: $P \vdash \langle e, s_0 \rangle \rightarrow^* \langle \text{true}, s_1 \rangle$
and e_1 -steps: $P \vdash \langle e_1, s_1 \rangle \rightarrow^* \langle e', s_2 \rangle$
shows $P \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s_0 \rangle \rightarrow^* \langle e', s_2 \rangle \langle \text{proof} \rangle$

lemma *CondReds2F*:

assumes e -steps: $P \vdash \langle e, s_0 \rangle \rightarrow^* \langle \text{false}, s_1 \rangle$
and e_2 -steps: $P \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle e', s_2 \rangle$
shows $P \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s_0 \rangle \rightarrow^* \langle e', s_2 \rangle \langle \text{proof} \rangle$

While

lemma *WhileFReds*:

assumes b -steps: $P \vdash \langle b, s \rangle \rightarrow^* \langle \text{false}, s' \rangle$
shows $P \vdash \langle \text{while } (b) c, s \rangle \rightarrow^* \langle \text{unit}, s' \rangle \langle \text{proof} \rangle$

lemma *WhileRedsThrow*:

assumes b -steps: $P \vdash \langle b, s \rangle \rightarrow^* \langle \text{throw } e, s' \rangle$
shows $P \vdash \langle \text{while } (b) c, s \rangle \rightarrow^* \langle \text{throw } e, s' \rangle \langle \text{proof} \rangle$

lemma *WhileTReds*:

assumes b -steps: $P \vdash \langle b, s_0 \rangle \rightarrow^* \langle \text{true}, s_1 \rangle$
and c -steps: $P \vdash \langle c, s_1 \rangle \rightarrow^* \langle \text{Val } v_1, s_2 \rangle$
and while -steps: $P \vdash \langle \text{while } (b) c, s_2 \rangle \rightarrow^* \langle e, s_3 \rangle$
shows $P \vdash \langle \text{while } (b) c, s_0 \rangle \rightarrow^* \langle e, s_3 \rangle \langle \text{proof} \rangle$

lemma *WhileTRedsThrow*:

assumes b -steps: $P \vdash \langle b, s_0 \rangle \rightarrow^* \langle \text{true}, s_1 \rangle$
and c -steps: $P \vdash \langle c, s_1 \rangle \rightarrow^* \langle \text{throw } e, s_2 \rangle$
shows $P \vdash \langle \text{while } (b) c, s_0 \rangle \rightarrow^* \langle \text{throw } e, s_2 \rangle \langle \text{proof} \rangle$

Throw

lemma *ThrowReds*:

$P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P \vdash \langle \text{throw } e, s \rangle \rightarrow^* \langle \text{throw } e', s' \rangle \langle \text{proof} \rangle$

lemma *ThrowRedsNull*:

$P \vdash \langle e, s \rangle \rightarrow^* \langle \text{null}, s' \rangle \implies P \vdash \langle \text{throw } e, s \rangle \rightarrow^* \langle \text{THROW NullPointer}, s' \rangle \langle \text{proof} \rangle$

lemma *ThrowRedsThrow*:

$P \vdash \langle e, s \rangle \rightarrow^* \langle \text{throw } a, s' \rangle \implies P \vdash \langle \text{throw } e, s \rangle \rightarrow^* \langle \text{throw } a, s' \rangle \langle \text{proof} \rangle$

InitBlock

lemma *InitBlockReds-aux*:

$P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies$
 $\forall h \ l \ h' \ l' \ v. s = (h, l(V \mapsto v)) \longrightarrow s' = (h', l') \longrightarrow$
 $P \vdash \langle \{ V:T := \text{Val } v; e \}, (h, l) \rangle \rightarrow^* \langle \{ V:T := \text{Val}(\text{the}(l' \ V)); e' \}, (h', l'(V := (l \ V))) \rangle \langle \text{proof} \rangle$

lemma *InitBlockReds*:

$P \vdash \langle e, (h, l(V \mapsto v)) \rangle \rightarrow^* \langle e', (h', l') \rangle \implies$

$$P \vdash \langle \{V:T := Val v; e\}, (h,l) \rangle \rightarrow^* \langle \{V:T := Val(the(l' V)); e'\}, (h',l'(V:=l V)) \rangle \langle proof \rangle$$

lemma *InitBlockRedsFinal*:

$$\llbracket P \vdash \langle e, (h, l(V \mapsto v)) \rangle \rightarrow^* \langle e', (h', l') \rangle; final e' \rrbracket \implies \\ P \vdash \langle \{V:T := Val v; e\}, (h,l) \rangle \rightarrow^* \langle e', (h', l'(V := l V)) \rangle \langle proof \rangle$$

Block

lemma *BlockRedsFinal*:

assumes *reds*: $P \vdash \langle e_0, s_0 \rangle \rightarrow^* \langle e_2, (h_2, l_2) \rangle$ **and** *fin*: *final* e_2
shows $\bigwedge h_0 l_0. s_0 = (h_0, l_0(V := None)) \implies P \vdash \langle \{V:T; e_0\}, (h_0, l_0) \rangle \rightarrow^* \langle e_2, (h_2, l_2(V := l_0 V)) \rangle \langle proof \rangle$

try-catch

lemma *TryReds*:

$$P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P \vdash \langle try e catch(C V) e_2, s \rangle \rightarrow^* \langle try e' catch(C V) e_2, s' \rangle \langle proof \rangle$$

lemma *TryRedsVal*:

$$P \vdash \langle e, s \rangle \rightarrow^* \langle Val v, s' \rangle \implies P \vdash \langle try e catch(C V) e_2, s \rangle \rightarrow^* \langle Val v, s' \rangle \langle proof \rangle$$

lemma *TryCatchRedsFinal*:

assumes *e1-steps*: $P \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle Throw a, (h_1, l_1) \rangle$
and $h_1 a = Some(D, fs)$ **and** *sub*: $P \vdash D \preceq^* C$
and *e2-steps*: $P \vdash \langle e_2, (h_1, l_1(V \mapsto Addr a)) \rangle \rightarrow^* \langle e_2', (h_2, l_2) \rangle$
and *final*: *final* e_2'
shows $P \vdash \langle try e_1 catch(C V) e_2, s_0 \rangle \rightarrow^* \langle e_2', (h_2, (l_2::locals)(V := l_1 V)) \rangle \langle proof \rangle$

lemma *TryRedsFail*:

$$\llbracket P \vdash \langle e_1, s \rangle \rightarrow^* \langle Throw a, (h, l) \rangle; h a = Some(D, fs); \neg P \vdash D \preceq^* C \rrbracket \\ \implies P \vdash \langle try e_1 catch(C V) e_2, s \rangle \rightarrow^* \langle Throw a, (h, l) \rangle \langle proof \rangle$$

List

lemma *ListReds1*:

$$P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P \vdash \langle e \# es, s \rangle [\rightarrow]^* \langle e' \# es, s' \rangle \langle proof \rangle$$

lemma *ListReds2*:

$$P \vdash \langle es, s \rangle [\rightarrow]^* \langle es', s' \rangle \implies P \vdash \langle Val v \# es, s \rangle [\rightarrow]^* \langle Val v \# es', s' \rangle \langle proof \rangle$$

lemma *ListRedsVal*:

$$\llbracket P \vdash \langle e, s_0 \rangle \rightarrow^* \langle Val v, s_1 \rangle; P \vdash \langle es, s_1 \rangle [\rightarrow]^* \langle es', s_2 \rangle \rrbracket \\ \implies P \vdash \langle e \# es, s_0 \rangle [\rightarrow]^* \langle Val v \# es', s_2 \rangle \langle proof \rangle$$

Call

First a few lemmas on what happens to free variables during redction.

lemma **assumes** *wf*: *wuf-J-prog* P

shows *Red-fv*: $P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies fv e' \subseteq fv e$
and $P \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \implies fvs es' \subseteq fvs es \langle proof \rangle$

lemma *Red-dom-lcl*:

$$P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies dom l' \subseteq dom l \cup fv e \text{ **and** } \\ P \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \implies dom l' \subseteq dom l \cup fvs es \langle proof \rangle$$

lemma *Reds-dom-lcl*:

assumes *wf*: *wuf-J-prog* P

shows $P \vdash \langle e, (h, l) \rangle \rightarrow^* \langle e', (h', l') \rangle \implies dom l' \subseteq dom l \cup fv e \langle proof \rangle$

Now a few lemmas on the behaviour of blocks during reduction.

lemma *override-on-upd-lemma*:

$$(override-on f (g(a \mapsto b)) A)(a := g a) = override-on f g (insert a A) \langle proof \rangle$$

lemma blocksReds:

$$\begin{aligned} \wedge l. \llbracket \text{length } Vs = \text{length } Ts; \text{length } vs = \text{length } Ts; \text{distinct } Vs; \\ P \vdash \langle e, (h, l(Vs \mapsto vs)) \rangle \rightarrow^* \langle e', (h', l') \rangle \rrbracket \\ \implies P \vdash \langle \text{blocks}(Vs, Ts, vs, e), (h, l) \rangle \rightarrow^* \langle \text{blocks}(Vs, Ts, \text{map } (the \circ l') \text{ } Vs, e'), (h', \text{override-on } l' \text{ } l \\ (\text{set } Vs)) \rangle \langle \text{proof} \rangle \end{aligned}$$

lemma blocksFinal:

$$\begin{aligned} \wedge l. \llbracket \text{length } Vs = \text{length } Ts; \text{length } vs = \text{length } Ts; \text{final } e \rrbracket \implies \\ P \vdash \langle \text{blocks}(Vs, Ts, vs, e), (h, l) \rangle \rightarrow^* \langle e, (h, l) \rangle \langle \text{proof} \rangle \end{aligned}$$

lemma blocksRedsFinal:

assumes *wf*: $\text{length } Vs = \text{length } Ts$ $\text{length } vs = \text{length } Ts$ *distinct* Vs
and *reds*: $P \vdash \langle e, (h, l(Vs \mapsto vs)) \rangle \rightarrow^* \langle e', (h', l') \rangle$
and *fin*: *final* e' **and** l'' : $l'' = \text{override-on } l' \text{ } l (\text{set } Vs)$
shows $P \vdash \langle \text{blocks}(Vs, Ts, vs, e), (h, l) \rangle \rightarrow^* \langle e', (h', l'') \rangle \langle \text{proof} \rangle$

An now the actual method call reduction lemmas.

lemma CallRedsObj:

$$P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P \vdash \langle e \cdot M(es), s \rangle \rightarrow^* \langle e' \cdot M(es), s' \rangle \langle \text{proof} \rangle$$

lemma CallRedsParams:

$$P \vdash \langle es, s \rangle \mapsto^* \langle es', s' \rangle \implies P \vdash \langle (Val \ v) \cdot M(es), s \rangle \rightarrow^* \langle (Val \ v) \cdot M(es'), s' \rangle \langle \text{proof} \rangle$$

lemma CallRedsFinal:

assumes *wwf*: *wwf-J-prog* P
and $P \vdash \langle e, s_0 \rangle \rightarrow^* \langle \text{addr } a, s_1 \rangle$
 $P \vdash \langle es, s_1 \rangle \mapsto^* \langle \text{map } Val \ vs, (h_2, l_2) \rangle$
 $h_2 \ a = \text{Some}(C, fs)$ $P \vdash C$ *sees* $M: Ts \rightarrow T = (pns, body)$ *in* D
 $\text{size } vs = \text{size } pns$
and l_2' : $l_2' = [this \mapsto \text{Addr } a, pns \mapsto vs]$
and *body*: $P \vdash \langle \text{body}, (h_2, l_2') \rangle \rightarrow^* \langle ef, (h_3, l_3) \rangle$
and *final* ef
shows $P \vdash \langle e \cdot M(es), s_0 \rangle \rightarrow^* \langle ef, (h_3, l_2) \rangle \langle \text{proof} \rangle$

lemma CallRedsThrowParams:

assumes *e-steps*: $P \vdash \langle e, s_0 \rangle \rightarrow^* \langle Val \ v, s_1 \rangle$
and *es-steps*: $P \vdash \langle es, s_1 \rangle \mapsto^* \langle \text{map } Val \ vs_1 \ @ \ \text{throw } a \ \# \ es_2, s_2 \rangle$
shows $P \vdash \langle e \cdot M(es), s_0 \rangle \rightarrow^* \langle \text{throw } a, s_2 \rangle \langle \text{proof} \rangle$

lemma CallRedsThrowObj:

$$P \vdash \langle e, s_0 \rangle \rightarrow^* \langle \text{throw } a, s_1 \rangle \implies P \vdash \langle e \cdot M(es), s_0 \rangle \rightarrow^* \langle \text{throw } a, s_1 \rangle \langle \text{proof} \rangle$$

lemma CallRedsNull:

assumes *e-steps*: $P \vdash \langle e, s_0 \rangle \rightarrow^* \langle \text{null}, s_1 \rangle$
and *es-steps*: $P \vdash \langle es, s_1 \rangle \mapsto^* \langle \text{map } Val \ vs, s_2 \rangle$
shows $P \vdash \langle e \cdot M(es), s_0 \rangle \rightarrow^* \langle \text{THROW } \text{NullPointer}, s_2 \rangle \langle \text{proof} \rangle$

The main Theorem

lemma *assumes* *wwf*: *wwf-J-prog* P

shows *big-by-small*: $P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \implies P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle$
and *big-by-small*s: $P \vdash \langle es, s \rangle \mapsto^* \langle es', s' \rangle \implies P \vdash \langle es, s \rangle \mapsto^* \langle es', s' \rangle \langle \text{proof} \rangle$

2.15.2 Big steps simulates small step

This direction was carried out by Norbert Schirmer and Daniel Wasserrab.

The big step equivalent of *RedWhile*:

lemma *unfold-while*:

$$P \vdash \langle \text{while}(b) \ c, s \rangle \Rightarrow \langle e', s' \rangle = P \vdash \langle \text{if}(b) \ (c; \text{while}(b) \ c) \ \text{else} \ (\text{unit}), s \rangle \Rightarrow \langle e', s' \rangle \langle \text{proof} \rangle$$

lemma *blocksEval*:

$$\begin{aligned} & \bigwedge Ts \ \text{vs} \ l \ l'. \llbracket \text{size } ps = \text{size } Ts; \ \text{size } ps = \text{size } vs; \ P \vdash \langle \text{blocks}(ps, Ts, vs, e), (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle \rrbracket \\ & \implies \exists l''. P \vdash \langle e, (h, l(ps[\mapsto]vs)) \rangle \Rightarrow \langle e', (h', l'') \rangle \langle \text{proof} \rangle \end{aligned}$$

lemma

assumes *wf*: *wwf-J-prog* *P*

shows *eval-restrict-lcl*:

$$\begin{aligned} & P \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle \implies (\bigwedge W. \text{fv } e \subseteq W \implies P \vdash \langle e, (h, l) \ 'W \rangle \Rightarrow \langle e', (h', l') \ 'W \rangle) \\ \text{and } & P \vdash \langle es, (h, l) \rangle [\Rightarrow] \langle es', (h', l') \rangle \implies (\bigwedge W. \text{fvs } es \subseteq W \implies P \vdash \langle es, (h, l) \ 'W \rangle [\Rightarrow] \langle es', (h', l') \ 'W \rangle) \langle \text{proof} \rangle \end{aligned}$$

lemma *eval-notfree-unchanged*:

$$\begin{aligned} & P \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle \implies (\bigwedge V. V \notin \text{fv } e \implies l' V = l V) \\ \text{and } & P \vdash \langle es, (h, l) \rangle [\Rightarrow] \langle es', (h', l') \rangle \implies (\bigwedge V. V \notin \text{fvs } es \implies l' V = l V) \langle \text{proof} \rangle \end{aligned}$$

lemma *eval-closed-lcl-unchanged*:

$$\llbracket P \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle; \ \text{fv } e = \{ \} \rrbracket \implies l' = l \langle \text{proof} \rangle$$

lemma *list-eval-Throw*:

assumes *eval-e*: $P \vdash \langle \text{throw } x, s \rangle \Rightarrow \langle e', s' \rangle$

shows $P \vdash \langle \text{map } \text{Val } \text{vs} \ @ \ \text{throw } x \ \# \ es', s \rangle [\Rightarrow] \langle \text{map } \text{Val } \text{vs} \ @ \ e' \ \# \ es', s' \rangle \langle \text{proof} \rangle$

The key lemma:

lemma

assumes *wf*: *wwf-J-prog* *P*

shows *extend-1-eval*:

$$\begin{aligned} & P \vdash \langle e, s \rangle \rightarrow \langle e'', s'' \rangle \implies (\bigwedge s' \ e'. P \vdash \langle e'', s'' \rangle \Rightarrow \langle e', s' \rangle \implies P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle) \\ \text{and } & \text{extend-1-evals:} \\ & P \vdash \langle es, t \rangle [\rightarrow] \langle es'', t'' \rangle \implies (\bigwedge t' \ es'. P \vdash \langle es'', t'' \rangle [\Rightarrow] \langle es', t' \rangle \implies P \vdash \langle es, t \rangle [\Rightarrow] \langle es', t' \rangle) \langle \text{proof} \rangle \end{aligned}$$

Its extension to \rightarrow^* :

lemma *extend-eval*:

assumes *wf*: *wwf-J-prog* *P*

and *reds*: $P \vdash \langle e, s \rangle \rightarrow^* \langle e'', s'' \rangle$ **and** *eval-rest*: $P \vdash \langle e'', s'' \rangle \Rightarrow \langle e', s' \rangle$

shows $P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \langle \text{proof} \rangle$

lemma *extend-evals*:

assumes *wf*: *wwf-J-prog* *P*

and *reds*: $P \vdash \langle es, s \rangle [\rightarrow]^* \langle es'', s'' \rangle$ **and** *eval-rest*: $P \vdash \langle es'', s'' \rangle [\Rightarrow] \langle es', s' \rangle$

shows $P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \langle \text{proof} \rangle$

Finally, small step semantics can be simulated by big step semantics:

theorem

assumes *wf*: *wwf-J-prog* *P*

shows *small-by-big*: $\llbracket P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle; \ \text{final } e \rrbracket \implies P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$

and $\llbracket P \vdash \langle es, s \rangle [\rightarrow]^* \langle es', s' \rangle; \ \text{finals } es \rrbracket \implies P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \langle \text{proof} \rangle$

2.15.3 Equivalence

And now, the crowning achievement:

corollary *big-iff-small*:

$$\begin{aligned} & \text{wf-J-prog } P \implies \\ & P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle = (P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \wedge \text{final } e') \langle \text{proof} \rangle \end{aligned}$$

end

2.16 Well-typedness of Jinja expressions

theory *WellType*

imports *../Common/Objects Expr*

begin

type-synonym

$$\text{env} = \text{vname} \rightarrow \text{ty}$$

inductive

$$\text{WT} :: [J\text{-prog}, \text{env}, \text{expr}, \text{ty}] \Rightarrow \text{bool}$$

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

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

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

for *P* :: *J-prog*

where

WTNew:

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

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

| *WTCast*:

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

| *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*:

$$\begin{aligned} & \llbracket P, E \vdash e_1 :: T_1; P, E \vdash e_2 :: T_2; P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1 \rrbracket \\ & \implies P, E \vdash e_1 \ll \text{Eq} \gg e_2 :: \text{Boolean} \end{aligned}$$

| *WTBinOpAdd*:

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

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

| *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*:

$$\llbracket P, E \vdash e :: \text{Class } C; P \vdash C \text{ sees } F:T \text{ in } D \rrbracket \\ \implies P, E \vdash e \cdot F\{D\} :: T$$

| *WTFAss*:

$$\llbracket P, E \vdash e_1 :: \text{Class } C; P \vdash C \text{ sees } F:T \text{ in } D; P, E \vdash e_2 :: T'; P \vdash T' \leq T \rrbracket \\ \implies P, E \vdash e_1 \cdot F\{D\} := e_2 :: \text{Void}$$

| *WTCall*:

$$\llbracket P, E \vdash e :: \text{Class } C; P \vdash C \text{ sees } M:Ts \rightarrow T = (\text{pns}, \text{body}) \text{ in } D; \\ P, E \vdash es \llbracket :: \rrbracket Ts'; P \vdash Ts' \llbracket \leq \rrbracket Ts \rrbracket \\ \implies P, E \vdash e \cdot M(es) :: T$$

| *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*:

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

| *WTWhile*:

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

| *WTThrow*:

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

| *WTTry*:

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

— well-typed expression lists

| *WTNil*:

$$P, E \vdash \llbracket \llbracket :: \rrbracket \rrbracket$$

| *WTCons*:

$$\llbracket P, E \vdash e :: T; P, E \vdash es \llbracket :: \rrbracket Ts \rrbracket \\ \implies P, E \vdash e \# es \llbracket :: \rrbracket T \# Ts$$

lemma [iff]: $(P, E \vdash \llbracket \llbracket :: \rrbracket \rrbracket Ts) = (Ts = \llbracket \rrbracket) \langle \text{proof} \rangle$

lemma [iff]: $(P, E \vdash e \# es \llbracket :: \rrbracket T \# Ts) = (P, E \vdash e :: T \wedge P, E \vdash es \llbracket :: \rrbracket Ts) \langle \text{proof} \rangle$

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

$$(\exists U Us. Ts = U \# Us \wedge P, E \vdash e :: U \wedge P, E \vdash es \llbracket :: \rrbracket Us) \langle \text{proof} \rangle$$

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

$$(\exists Ts_1 Ts_2. Ts = Ts_1 @ Ts_2 \wedge P, E \vdash es_1 \llbracket :: \rrbracket Ts_1 \wedge P, E \vdash es_2 \llbracket :: \rrbracket Ts_2) \langle \text{proof} \rangle$$

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

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

lemma [iff]: $P, E \vdash e_1; e_2 :: T_2 = (\exists T_1. P, E \vdash e_1 :: T_1 \wedge P, E \vdash e_2 :: T_2) \langle \text{proof} \rangle$

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

lemma *wt-env-mono*:

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

$P, E \vdash \text{es} [::] Ts \implies (\bigwedge E'. E \subseteq_m E' \implies P, E' \vdash \text{es} [::] Ts) \langle \text{proof} \rangle$

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

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

2.17 Runtime Well-typedness

theory *WellTypeRT*

imports *WellType*

begin

inductive

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

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

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

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

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

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

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

where

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

| $P, E, h \vdash \text{es} [::] Ts \equiv WTrts \ P \ h \ E \ \text{es} \ Ts$

| $WTrtNew$:

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

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

| $WTrtCast$:

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

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

| $WTrtVal$:

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

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

| $WTrtVar$:

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

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

| $WTrtBinOpEq$:

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

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

| $WTrtBinOpAdd$:

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

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

- | *WTrtLAss*:

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

$$\implies P, E, h \vdash V := e : \text{Void}$$
- | *WTrtFAcc*:

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

$$P, E, h \vdash e \cdot F\{D\} : T$$
- | *WTrtFAccNT*:

$$P, E, h \vdash e : NT \implies$$

$$P, E, h \vdash e \cdot F\{D\} : T$$
- | *WTrtFAss*:

$$\llbracket P, E, h \vdash e_1 : \text{Class } C; \ P \vdash C \text{ has } F:T \text{ in } D; \ P, E, h \vdash e_2 : T_2; \ P \vdash T_2 \leq T \rrbracket$$

$$\implies P, E, h \vdash e_1 \cdot F\{D\} := e_2 : \text{Void}$$
- | *WTrtFAssNT*:

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

$$\implies P, E, h \vdash e_1 \cdot F\{D\} := e_2 : \text{Void}$$
- | *WTrtCall*:

$$\llbracket P, E, h \vdash e : \text{Class } C; \ P \vdash C \text{ sees } M:Ts \rightarrow T = (pns, \text{body}) \text{ in } D;$$

$$P, E, h \vdash es \ [:] \ Ts'; \ P \vdash Ts' \leq Ts \rrbracket$$

$$\implies P, E, h \vdash e \cdot M(es) : T$$
- | *WTrtCallNT*:

$$\llbracket P, E, h \vdash e : NT; \ P, E, h \vdash es \ [:] \ Ts \rrbracket$$

$$\implies P, E, h \vdash e \cdot M(es) : T$$
- | *WTrtBlock*:

$$P, E(V \mapsto T), h \vdash e : T' \implies$$

$$P, E, h \vdash \{V:T; e\} : T'$$
- | *WTrtSeq*:

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

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

$$\llbracket P, E, h \vdash e : \text{Boolean}; \ P, E, h \vdash e_1 : T_1; \ P, E, h \vdash e_2 : T_2;$$

$$P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1; \ P \vdash T_1 \leq T_2 \longrightarrow T = T_2; \ P \vdash T_2 \leq T_1 \longrightarrow T = T_1 \rrbracket$$

$$\implies P, E, h \vdash \text{if } (e) \ e_1 \ \text{else } e_2 : T$$
- | *WTrtWhile*:

$$\llbracket P, E, h \vdash e : \text{Boolean}; \ P, E, h \vdash c : T \rrbracket$$

$$\implies P, E, h \vdash \text{while}(e) \ c : \text{Void}$$
- | *WTrtThrow*:

$$\llbracket P, E, h \vdash e : T_r; \ \text{is-refT } T_r \rrbracket \implies$$

$$P, E, h \vdash \text{throw } e : T$$
- | *WTrtTry*:

$$\llbracket P, E, h \vdash e_1 : T_1; \ P, E(V \mapsto \text{Class } C), h \vdash e_2 : T_2; \ P \vdash T_1 \leq T_2 \rrbracket$$

$$\implies P, E, h \vdash \text{try } e_1 \ \text{catch}(C \ V) \ e_2 : T_2$$

— well-typed expression lists

| *WTrtNil*:

$P, E, h \vdash [] \text{ [:] } []$

| *WTrtCons*:

$[P, E, h \vdash e : T; P, E, h \vdash es \text{ [:] } Ts]$
 $\implies P, E, h \vdash e \# es \text{ [:] } T \# Ts$

2.17.1 Easy consequences

lemma [*iff*]: $(P, E, h \vdash [] \text{ [:] } Ts) = (Ts = []) \langle proof \rangle$

lemma [*iff*]: $(P, E, h \vdash e \# es \text{ [:] } T \# Ts) = (P, E, h \vdash e : T \wedge P, E, h \vdash es \text{ [:] } Ts) \langle proof \rangle$

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

$(\exists U Us. Ts = U \# Us \wedge P, E, h \vdash e : U \wedge P, E, h \vdash es \text{ [:] } Us) \langle proof \rangle$

lemma [*simp*]: $\forall Ts. (P, E, h \vdash es_1 @ es_2 \text{ [:] } Ts) =$

$(\exists Ts_1 Ts_2. Ts = Ts_1 @ Ts_2 \wedge P, E, h \vdash es_1 \text{ [:] } Ts_1 \ \& \ P, E, h \vdash es_2 \text{ [:] } Ts_2) \langle proof \rangle$

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

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

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

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

2.17.2 Some interesting lemmas

lemma *WTrts-Val[simp]*:

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

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

lemma *WTrt-env-mono*:

$P, E, h \vdash e : T \implies (\bigwedge E'. E \subseteq_m E' \implies P, E', h \vdash e : T)$ **and**

$P, E, h \vdash es \text{ [:] } Ts \implies (\bigwedge E'. E \subseteq_m E' \implies P, E', h \vdash es \text{ [:] } Ts) \langle proof \rangle$

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

and *WTrts-hext-mono*: $P, E, h \vdash es \text{ [:] } Ts \implies h \leq h' \implies P, E, h' \vdash es \text{ [:] } Ts \langle proof \rangle$

lemma *WT-implies-WTrt*: $P, E \vdash e :: T \implies P, E, h \vdash e : T$

and *WTs-implies-WTrts*: $P, E \vdash es \text{ [::] } Ts \implies P, E, h \vdash es \text{ [:] } Ts \langle proof \rangle$

end

2.18 Definite assignment

theory *DefAss* **imports** *BigStep* **begin**

2.18.1 Hypersets

type-synonym *'a hyperset* = *'a set option*

definition *hyperUn* :: *'a hyperset* \Rightarrow *'a hyperset* \Rightarrow *'a hyperset* (**infixl** \sqcup 65)

where

$A \sqcup B \equiv \text{case } A \text{ of } \text{None} \Rightarrow \text{None}$

$$| [A] \Rightarrow (\text{case } B \text{ of } \text{None} \Rightarrow \text{None} \mid [B] \Rightarrow [A \cup B])$$

definition *hyperInt* :: 'a hyperset \Rightarrow 'a hyperset \Rightarrow 'a hyperset (infixl \sqcap 70)

where

$$\begin{aligned} A \sqcap B &\equiv \text{case } A \text{ of } \text{None} \Rightarrow B \\ &\mid [A] \Rightarrow (\text{case } B \text{ of } \text{None} \Rightarrow [A] \mid [B] \Rightarrow [A \cap B]) \end{aligned}$$

definition *hyperDiff1* :: 'a hyperset \Rightarrow 'a \Rightarrow 'a hyperset (infixl \ominus 65)

where

$$A \ominus a \equiv \text{case } A \text{ of } \text{None} \Rightarrow \text{None} \mid [A] \Rightarrow [A - \{a\}]$$

definition *hyper-isin* :: 'a \Rightarrow 'a hyperset \Rightarrow bool (infixl $\in\in$ 50)

where

$$a \in\in A \equiv \text{case } A \text{ of } \text{None} \Rightarrow \text{True} \mid [A] \Rightarrow a \in A$$

definition *hyper-subset* :: 'a hyperset \Rightarrow 'a hyperset \Rightarrow bool (infixl \sqsubseteq 50)

where

$$\begin{aligned} A \sqsubseteq B &\equiv \text{case } B \text{ of } \text{None} \Rightarrow \text{True} \\ &\mid [B] \Rightarrow (\text{case } A \text{ of } \text{None} \Rightarrow \text{False} \mid [A] \Rightarrow A \sqsubseteq B) \end{aligned}$$

lemmas *hyperset-defs* =

hyperUn-def hyperInt-def hyperDiff1-def hyper-isin-def hyper-subset-def

lemma [*simp*]: $[\{\}] \sqcup A = A \wedge A \sqcup [\{\}] = A$ (proof)

lemma [*simp*]: $[A] \sqcup [B] = [A \cup B] \wedge [A] \ominus a = [A - \{a\}]$ (proof)

lemma [*simp*]: $\text{None} \sqcup A = \text{None} \wedge A \sqcup \text{None} = \text{None}$ (proof)

lemma [*simp*]: $a \in\in \text{None} \wedge \text{None} \ominus a = \text{None}$ (proof)

lemma *hyperUn-assoc*: $(A \sqcup B) \sqcup C = A \sqcup (B \sqcup C)$ (proof)

lemma *hyper-insert-comm*: $A \sqcup [\{a\}] = [\{a\}] \sqcup A \wedge A \sqcup ([\{a\}] \sqcup B) = [\{a\}] \sqcup (A \sqcup B)$ (proof)

2.18.2 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

$$\begin{aligned} \mathcal{A} (\text{new } C) &= [\{\}] \\ \mathcal{A} (\text{Cast } C \ e) &= \mathcal{A} \ e \\ \mathcal{A} (\text{Val } v) &= [\{\}] \\ \mathcal{A} (e_1 \ll\text{bop}\gg e_2) &= \mathcal{A} \ e_1 \sqcup \mathcal{A} \ e_2 \\ \mathcal{A} (\text{Var } V) &= [\{\}] \\ \mathcal{A} (\text{LAss } V \ e) &= [\{V\}] \sqcup \mathcal{A} \ e \\ \mathcal{A} (e \cdot F\{D\}) &= \mathcal{A} \ e \\ \mathcal{A} (e_1 \cdot F\{D\};=e_2) &= \mathcal{A} \ e_1 \sqcup \mathcal{A} \ e_2 \\ \mathcal{A} (e \cdot M(es)) &= \mathcal{A} \ e \sqcup \mathcal{A}s \ es \\ \mathcal{A} (\{V:T; e\}) &= \mathcal{A} \ e \ominus V \\ \mathcal{A} (e_1;;e_2) &= \mathcal{A} \ e_1 \sqcup \mathcal{A} \ e_2 \\ \mathcal{A} (\text{if } (e) \ e_1 \ \text{else } e_2) &= \mathcal{A} \ e \sqcup (\mathcal{A} \ e_1 \sqcap \mathcal{A} \ e_2) \\ \mathcal{A} (\text{while } (b) \ e) &= \mathcal{A} \ b \\ \mathcal{A} (\text{throw } e) &= \text{None} \\ \mathcal{A} (\text{try } e_1 \ \text{catch}(C \ V) \ e_2) &= \mathcal{A} \ e_1 \sqcap (\mathcal{A} \ e_2 \ominus V) \end{aligned}$$

$\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

$\mathcal{D} (\text{new } C) A = \text{True}$
 $\mathcal{D} (\text{Cast } C \ e) A = \mathcal{D} \ e \ A$
 $\mathcal{D} (\text{Val } v) A = \text{True}$
 $\mathcal{D} (e_1 \ll \text{bop} \gg e_2) A = (\mathcal{D} \ e_1 \ A \wedge \mathcal{D} \ e_2 \ (A \sqcup \mathcal{A} \ e_1))$
 $\mathcal{D} (\text{Var } V) A = (V \in \in A)$
 $\mathcal{D} (\text{LAss } V \ e) A = \mathcal{D} \ e \ A$
 $\mathcal{D} (e \cdot F\{D\}) A = \mathcal{D} \ e \ A$
 $\mathcal{D} (e_1 \cdot F\{D\} := e_2) A = (\mathcal{D} \ e_1 \ A \wedge \mathcal{D} \ e_2 \ (A \sqcup \mathcal{A} \ e_1))$
 $\mathcal{D} (e \cdot M(es)) A = (\mathcal{D} \ e \ A \wedge \mathcal{D}s \ es \ (A \sqcup \mathcal{A} \ e))$
 $\mathcal{D} (\{V:T; e\}) A = \mathcal{D} \ e \ (A \ominus V)$
 $\mathcal{D} (e_1 ;; e_2) A = (\mathcal{D} \ e_1 \ A \wedge \mathcal{D} \ e_2 \ (A \sqcup \mathcal{A} \ e_1))$
 $\mathcal{D} (\text{if } (e) \ e_1 \ \text{else } e_2) A =$
 $(\mathcal{D} \ e \ A \wedge \mathcal{D} \ e_1 \ (A \sqcup \mathcal{A} \ e) \wedge \mathcal{D} \ e_2 \ (A \sqcup \mathcal{A} \ e))$
 $\mathcal{D} (\text{while } (e) \ c) A = (\mathcal{D} \ e \ A \wedge \mathcal{D} \ c \ (A \sqcup \mathcal{A} \ e))$
 $\mathcal{D} (\text{throw } e) A = \mathcal{D} \ e \ A$
 $\mathcal{D} (\text{try } e_1 \ \text{catch}(C \ V) \ e_2) A = (\mathcal{D} \ e_1 \ A \wedge \mathcal{D} \ e_2 \ (A \sqcup [\{V\}]))$

$\mathcal{D}s \ (\[]) 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]*: $\mathcal{A}s \ (\text{map } \text{Val } vs) = [\{\}] \langle \text{proof} \rangle$

lemma *D-append[iff]*: $\bigwedge A. \mathcal{D}s \ (es \ @ \ es') A = (\mathcal{D}s \ es \ A \wedge \mathcal{D}s \ es' \ (A \sqcup \mathcal{A}s \ es)) \langle \text{proof} \rangle$

lemma *A-fv*: $\bigwedge A. \mathcal{A} \ e = [A] \implies A \subseteq \text{fv } e$

and $\bigwedge A. \mathcal{A}s \ es = [A] \implies A \subseteq \text{fvs } es \langle \text{proof} \rangle$

lemma *sqUn-lem*: $A \sqsubseteq A' \implies A \sqcup B \sqsubseteq A' \sqcup B \langle \text{proof} \rangle$

lemma *diff-lem*: $A \sqsubseteq A' \implies A \ominus b \sqsubseteq A' \ominus b \langle \text{proof} \rangle$

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*: $\bigwedge A \ A'. A \sqsubseteq A' \implies \mathcal{D}s \ es \ A \implies \mathcal{D}s \ (es :: 'a \ \text{exp list}) \ A' \langle \text{proof} \rangle$

lemma *D-mono'*: $\mathcal{D} \ e \ A \implies A \sqsubseteq A' \implies \mathcal{D} \ e \ A'$

and *Ds-mono'*: $\mathcal{D}s \ es \ A \implies A \sqsubseteq A' \implies \mathcal{D}s \ es \ A' \langle \text{proof} \rangle$

end

2.19 Conformance Relations for Type Soundness Proofs

theory *Conform*

imports *Exceptions*

begin

definition *conf* :: $'m \ \text{prog} \Rightarrow \text{heap} \Rightarrow \text{val} \Rightarrow \text{ty} \Rightarrow \text{bool}$ $(-, \vdash -, \leq - \ [51,51,51,51] \ 50)$

where

$P, h \vdash v \leq T \equiv$
 $\exists T'. \text{typeof}_h \ v = \text{Some } T' \wedge P \vdash T' \leq T$

definition $oconf :: 'm prog \Rightarrow heap \Rightarrow obj \Rightarrow bool$ $(-, - \vdash - \surd [51, 51, 51] 50)$

where

$$P, h \vdash obj \surd \equiv$$

$$let (C, fs) = obj \text{ in } \forall F D T. P \vdash C \text{ has } F:T \text{ in } D \longrightarrow$$

$$(\exists v. fs(F, D) = Some v \wedge P, h \vdash v : \leq T)$$

definition $hconf :: 'm prog \Rightarrow heap \Rightarrow bool$ $(- \vdash - \surd [51, 51] 50)$

where

$$P \vdash h \surd \equiv$$

$$(\forall a \text{ obj}. h a = Some \text{ obj} \longrightarrow P, h \vdash obj \surd) \wedge \text{preallocated } h$$

definition $lconf :: 'm prog \Rightarrow heap \Rightarrow (vname \rightarrow val) \Rightarrow (vname \rightarrow ty) \Rightarrow bool$ $(-, - \vdash - '(:\leq)' - [51, 51, 51, 51] 50)$

where

$$P, h \vdash l (: \leq) E \equiv$$

$$\forall V v. l V = Some v \longrightarrow (\exists T. E V = Some T \wedge P, h \vdash v : \leq T)$$

abbreviation

$$confs :: 'm prog \Rightarrow heap \Rightarrow val \text{ list} \Rightarrow ty \text{ list} \Rightarrow bool$$

$$(-, - \vdash - [:\leq] - [51, 51, 51, 51] 50) \text{ where}$$

$$P, h \vdash vs [:\leq] Ts \equiv \text{list-all2 } (conf P h) \text{ vs } Ts$$

2.19.1 Value conformance $:\leq$

lemma $conf\text{-Null}$ [simp]: $P, h \vdash Null : \leq T = P \vdash NT \leq T \langle \text{proof} \rangle$

lemma $typeof\text{-conf}$ [simp]: $typeof_h v = Some T \Longrightarrow P, h \vdash v : \leq T \langle \text{proof} \rangle$

lemma $typeof\text{-lit}\text{-conf}$ [simp]: $typeof v = Some T \Longrightarrow P, h \vdash v : \leq T \langle \text{proof} \rangle$

lemma $defval\text{-conf}$ [simp]: $P, h \vdash \text{default-val } T : \leq T \langle \text{proof} \rangle$

lemma $conf\text{-upd}\text{-obj}$: $h a = Some(C, fs) \Longrightarrow (P, h(a \rightarrow (C, fs'))) \vdash x : \leq T = (P, h \vdash x : \leq T) \langle \text{proof} \rangle$

lemma $conf\text{-widen}$: $P, h \vdash v : \leq T \Longrightarrow P \vdash T \leq T' \Longrightarrow P, h \vdash v : \leq T' \langle \text{proof} \rangle$

lemma $conf\text{-hext}$: $h \leq h' \Longrightarrow P, h \vdash v : \leq T \Longrightarrow P, h' \vdash v : \leq T \langle \text{proof} \rangle$

lemma $conf\text{-ClassD}$: $P, h \vdash v : \leq \text{Class } C \Longrightarrow$

$$v = Null \vee (\exists a \text{ obj } T. v = \text{Addr } a \wedge h a = Some \text{ obj} \wedge \text{obj-ty } \text{obj} = T \wedge P \vdash T \leq \text{Class } C) \langle \text{proof} \rangle$$

lemma $conf\text{-NT}$ [iff]: $P, h \vdash v : \leq NT = (v = Null) \langle \text{proof} \rangle$

lemma $non\text{-npD}$: $\llbracket v \neq Null; P, h \vdash v : \leq \text{Class } C \rrbracket$

$$\Longrightarrow \exists a C' fs. v = \text{Addr } a \wedge h a = Some(C', fs) \wedge P \vdash C' \preceq^* C \langle \text{proof} \rangle$$

2.19.2 Value list conformance $[:\leq]$

lemma $confs\text{-widens}$ [trans]: $\llbracket P, h \vdash vs [:\leq] Ts; P \vdash Ts [:\leq] Ts' \rrbracket \Longrightarrow P, h \vdash vs [:\leq] Ts' \langle \text{proof} \rangle$

lemma $confs\text{-rev}$: $P, h \vdash \text{rev } s [:\leq] t = (P, h \vdash s [:\leq] \text{rev } t) \langle \text{proof} \rangle$

lemma $confs\text{-conv}\text{-map}$:

$$\bigwedge Ts'. P, h \vdash vs [:\leq] Ts' = (\exists Ts. \text{map } typeof_h vs = \text{map } Some Ts \wedge P \vdash Ts [:\leq] Ts') \langle \text{proof} \rangle$$

lemma $confs\text{-hext}$: $P, h \vdash vs [:\leq] Ts \Longrightarrow h \leq h' \Longrightarrow P, h' \vdash vs [:\leq] Ts \langle \text{proof} \rangle$

lemma $confs\text{-Cons2}$: $P, h \vdash xs [:\leq] y \# ys = (\exists z zs. xs = z \# zs \wedge P, h \vdash z : \leq y \wedge P, h \vdash zs [:\leq] ys) \langle \text{proof} \rangle$

2.19.3 Object conformance

lemma $oconf\text{-hext}$: $P, h \vdash obj \surd \Longrightarrow h \leq h' \Longrightarrow P, h' \vdash obj \surd \langle \text{proof} \rangle$

lemma $oconf\text{-init}\text{-fields}$:

$$P \vdash C \text{ has-fields } FDTs \Longrightarrow P, h \vdash (C, \text{init-fields } FDTs) \surd$$

$$\langle \text{proof} \rangle$$

lemma $oconf\text{-fupd}$ [intro?]:

$$\begin{aligned} & \llbracket P \vdash C \text{ has } F:T \text{ in } D; P, h \vdash v : \leq T; P, h \vdash (C, fs) \checkmark \rrbracket \\ & \implies P, h \vdash (C, fs((F, D) \mapsto v)) \checkmark \langle \text{proof} \rangle \end{aligned}$$

2.19.4 Heap conformance

lemma *hconfD*: $\llbracket P \vdash h \checkmark; h a = \text{Some } obj \rrbracket \implies P, h \vdash obj \checkmark \langle \text{proof} \rangle$

lemma *hconf-new*: $\llbracket P \vdash h \checkmark; h a = \text{None}; P, h \vdash obj \checkmark \rrbracket \implies P \vdash h(a \mapsto obj) \checkmark \langle \text{proof} \rangle$

lemma *hconf-upd-obj*: $\llbracket P \vdash h \checkmark; h a = \text{Some}(C, fs); P, h \vdash (C, fs') \checkmark \rrbracket \implies P \vdash h(a \mapsto (C, fs')) \checkmark \langle \text{proof} \rangle$

2.19.5 Local variable conformance

lemma *lconf-heap*: $\llbracket P, h \vdash l (: \leq) E; h \leq h' \rrbracket \implies P, h' \vdash l (: \leq) E \langle \text{proof} \rangle$

lemma *lconf-upd*:

$$\llbracket P, h \vdash l (: \leq) E; P, h \vdash v : \leq T; E V = \text{Some } T \rrbracket \implies P, h \vdash l(V \mapsto v) (: \leq) E \langle \text{proof} \rangle$$

lemma *lconf-empty[iff]*: $P, h \vdash \text{Map.empty} (: \leq) E \langle \text{proof} \rangle$

lemma *lconf-upd2*: $\llbracket P, h \vdash l (: \leq) E; P, h \vdash v : \leq T \rrbracket \implies P, h \vdash l(V \mapsto v) (: \leq) E(V \mapsto T) \langle \text{proof} \rangle$

end

2.20 Progress of Small Step Semantics

theory *Progress*

imports *Equivalence WellTypeRT DefAss ../Common/Conform*

begin

lemma *final-addrE*:

$$\begin{aligned} & \llbracket P, E, h \vdash e : \text{Class } C; \text{final } e; \\ & \quad \bigwedge a. e = \text{addr } a \implies R; \\ & \quad \bigwedge a. e = \text{Throw } a \implies R \rrbracket \implies R \langle \text{proof} \rangle \end{aligned}$$

lemma *finalRefE*:

$$\begin{aligned} & \llbracket P, E, h \vdash e : T; \text{is-refT } T; \text{final } e; \\ & \quad e = \text{null} \implies R; \\ & \quad \bigwedge a C. \llbracket e = \text{addr } a; T = \text{Class } C \rrbracket \implies R; \\ & \quad \bigwedge a. e = \text{Throw } a \implies R \rrbracket \implies R \langle \text{proof} \rangle \end{aligned}$$

Derivation of new induction scheme for well typing:

inductive

$$\begin{aligned} & \text{WTrt}' :: [J\text{-prog}, \text{heap}, \text{env}, \text{expr}, \text{ty}] \Rightarrow \text{bool} \\ & \text{and } \text{WTrts}' :: [J\text{-prog}, \text{heap}, \text{env}, \text{expr list}, \text{ty list}] \Rightarrow \text{bool} \\ & \text{and } \text{WTrt2}' :: [J\text{-prog}, \text{env}, \text{heap}, \text{expr}, \text{ty}] \Rightarrow \text{bool} \\ & \quad (-, -, - \vdash - : '' - [51, 51, 51] 50) \\ & \text{and } \text{WTrts2}' :: [J\text{-prog}, \text{env}, \text{heap}, \text{expr list}, \text{ty list}] \Rightarrow \text{bool} \\ & \quad (-, -, - \vdash - : [] - [51, 51, 51] 50) \\ & \text{for } P :: J\text{-prog} \text{ and } h :: \text{heap} \end{aligned}$$

where

$$\begin{aligned} & P, E, h \vdash e : ' T \equiv \text{WTrt}' P h E e T \\ & | P, E, h \vdash es [] Ts \equiv \text{WTrts}' P h E es Ts \\ \\ & | \text{is-class } P C \implies P, E, h \vdash \text{new } C : ' \text{Class } C \\ & | \llbracket P, E, h \vdash e : ' T; \text{is-refT } T; \text{is-class } P C \rrbracket \\ & \quad \implies P, E, h \vdash \text{Cast } C e : ' \text{Class } C \\ & | \text{typeof}_h v = \text{Some } T \implies P, E, h \vdash \text{Val } v : ' T \\ & | E v = \text{Some } T \implies P, E, h \vdash \text{Var } v : ' T \end{aligned}$$

$\llbracket P, E, h \vdash e_1 : ' T_1; P, E, h \vdash e_2 : ' T_2 \rrbracket$
 $\implies P, E, h \vdash e_1 \llbracket Eq \rrbracket e_2 : ' Boolean$
 $\llbracket P, E, h \vdash e_1 : ' Integer; P, E, h \vdash e_2 : ' Integer \rrbracket$
 $\implies P, E, h \vdash e_1 \llbracket Add \rrbracket e_2 : ' Integer$
 $\llbracket P, E, h \vdash Var V : ' T; P, E, h \vdash e : ' T'; P \vdash T' \leq T \rrbracket$
 $\implies P, E, h \vdash V := e : ' Void$
 $\llbracket P, E, h \vdash e : ' Class C; P \vdash C \text{ has } F:T \text{ in } D \rrbracket \implies P, E, h \vdash e \cdot F\{D\} : ' T$
 $P, E, h \vdash e : ' NT \implies P, E, h \vdash e \cdot F\{D\} : ' T$
 $\llbracket P, E, h \vdash e_1 : ' Class C; P \vdash C \text{ has } F:T \text{ in } D;$
 $P, E, h \vdash e_2 : ' T_2; P \vdash T_2 \leq T \rrbracket$
 $\implies P, E, h \vdash e_1 \cdot F\{D\} := e_2 : ' Void$
 $\llbracket P, E, h \vdash e_1 : ' NT; P, E, h \vdash e_2 : ' T_2 \rrbracket \implies P, E, h \vdash e_1 \cdot F\{D\} := e_2 : ' Void$
 $\llbracket P, E, h \vdash e : ' Class C; P \vdash C \text{ sees } M:Ts \rightarrow T = (pns, body) \text{ in } D;$
 $P, E, h \vdash es \llbracket \cdot \rrbracket Ts'; P \vdash Ts' \llbracket \leq \rrbracket Ts \rrbracket$
 $\implies P, E, h \vdash e \cdot M(es) : ' T$
 $\llbracket P, E, h \vdash e : ' NT; P, E, h \vdash es \llbracket \cdot \rrbracket Ts \rrbracket \implies P, E, h \vdash e \cdot M(es) : ' T$
 $P, E, h \vdash \llbracket \cdot \rrbracket \llbracket \cdot \rrbracket$
 $\llbracket P, E, h \vdash e : ' T; P, E, h \vdash es \llbracket \cdot \rrbracket Ts \rrbracket \implies P, E, h \vdash e \# es \llbracket \cdot \rrbracket T \# Ts$
 $\llbracket typeof_h v = Some T_1; P \vdash T_1 \leq T; P, E(V \mapsto T), h \vdash e_2 : ' T_2 \rrbracket$
 $\implies P, E, h \vdash \{V:T := Val v; e_2\} : ' T_2$
 $\llbracket P, E(V \mapsto T), h \vdash e : ' T'; \neg assigned V e \rrbracket \implies P, E, h \vdash \{V:T; e\} : ' T'$
 $\llbracket P, E, h \vdash e_1 : ' T_1; P, E, h \vdash e_2 : ' T_2 \rrbracket \implies P, E, h \vdash e_1 ;; e_2 : ' T_2$
 $\llbracket P, E, h \vdash e : ' Boolean; P, E, h \vdash e_1 : ' T_1; P, E, h \vdash e_2 : ' T_2;$
 $P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1;$
 $P \vdash T_1 \leq T_2 \longrightarrow T = T_2; P \vdash T_2 \leq T_1 \longrightarrow T = T_1 \rrbracket$
 $\implies P, E, h \vdash if (e) e_1 else e_2 : ' T$

$\llbracket P, E, h \vdash e : ' Boolean; P, E, h \vdash c : ' T \rrbracket$
 $\implies P, E, h \vdash while(e) c : ' Void$
 $\llbracket P, E, h \vdash e : ' T_r; is-refT T_r \rrbracket \implies P, E, h \vdash throw e : ' T$
 $\llbracket P, E, h \vdash e_1 : ' T_1; P, E(V \mapsto Class C), h \vdash e_2 : ' T_2; P \vdash T_1 \leq T_2 \rrbracket$
 $\implies P, E, h \vdash try e_1 catch(C V) e_2 : ' T_2$

lemma $\llbracket iff \rrbracket: P, E, h \vdash e_1 ;; e_2 : ' T_2 = (\exists T_1. P, E, h \vdash e_1 : ' T_1 \wedge P, E, h \vdash e_2 : ' T_2) \langle proof \rangle$

lemma $\llbracket iff \rrbracket: P, E, h \vdash Val v : ' T = (typeof_h v = Some T) \langle proof \rangle$

lemma $\llbracket iff \rrbracket: P, E, h \vdash Var v : ' T = (E v = Some T) \langle proof \rangle$

lemma $wt-wt'$: $P, E, h \vdash e : T \implies P, E, h \vdash e : ' T$

and $wts-wts'$: $P, E, h \vdash es \llbracket \cdot \rrbracket Ts \implies P, E, h \vdash es \llbracket \cdot \rrbracket Ts \langle proof \rangle$

lemma $wt'-wt$: $P, E, h \vdash e : ' T \implies P, E, h \vdash e : T$

and $wts'-wts$: $P, E, h \vdash es \llbracket \cdot \rrbracket Ts \implies P, E, h \vdash es \llbracket \cdot \rrbracket Ts \langle proof \rangle$

corollary $wt'-iff-wt$: $(P, E, h \vdash e : ' T) = (P, E, h \vdash e : T) \langle proof \rangle$

corollary $wts'-iff-wts$: $(P, E, h \vdash es \llbracket \cdot \rrbracket Ts) = (P, E, h \vdash es \llbracket \cdot \rrbracket Ts) \langle proof \rangle$

theorem assumes wf : $wuf\text{-}J\text{-}prog P$ **and** $hconf$: $P \vdash h \checkmark$

shows $progress$: $P, E, h \vdash e : T \implies$

$(\bigwedge l. \llbracket D e \llbracket dom l \rrbracket; \neg final e \rrbracket \implies \exists e' s'. P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', s' \rangle)$

and $P, E, h \vdash es \llbracket \cdot \rrbracket Ts \implies$

$(\bigwedge l. \llbracket Ds es \llbracket dom l \rrbracket; \neg finals es \rrbracket \implies \exists es' s'. P \vdash \langle es, (h, l) \rangle \llbracket \rightarrow \rrbracket \langle es', s' \rangle) \langle proof \rangle$

end

2.21 Well-formedness Constraints

theory *JWellForm*
imports *../Common/WellForm WWellForm WellType DefAss*
begin

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

wf-J-mdecl *P C* \equiv $\lambda(M, Ts, T, (pns, body)).$
 $length\ Ts = length\ pns \wedge$
 $distinct\ pns \wedge$
 $this \notin set\ pns \wedge$
 $(\exists T'. P, [this \mapsto Class\ C, pns \mapsto Ts] \vdash body :: T' \wedge P \vdash T' \leq T) \wedge$
 $\mathcal{D}\ body\ [\{this\} \cup set\ pns]$

lemma *wf-J-mdecl[simp]*:
 $wf-J-mdecl\ P\ C\ (M, Ts, T, pns, body) \equiv$
 $(length\ Ts = length\ pns \wedge$
 $distinct\ pns \wedge$
 $this \notin set\ pns \wedge$
 $(\exists T'. P, [this \mapsto Class\ C, pns \mapsto Ts] \vdash body :: T' \wedge P \vdash T' \leq T) \wedge$
 $\mathcal{D}\ body\ [\{this\} \cup set\ pns]) \langle proof \rangle$

abbreviation

wf-J-prog :: *J-prog* \Rightarrow *bool* **where**
 $wf-J-prog == wf-prog\ wf-J-mdecl$

lemma *wf-J-prog-wf-J-mdecl*:
 $\llbracket wf-J-prog\ P; (C, D, fds, mths) \in set\ P; jmdcl \in set\ mths \rrbracket$
 $\implies wf-J-mdecl\ P\ C\ jmdcl \langle proof \rangle$

lemma *wf-mdecl-wwf-mdecl*: $wf-J-mdecl\ P\ C\ Md \implies wwf-J-mdecl\ P\ C\ Md \langle proof \rangle$

lemma *wf-prog-wwf-prog*: $wf-J-prog\ P \implies wwf-J-prog\ P \langle proof \rangle$

end

2.22 Type Safety Proof

theory *TypeSafe*
imports *Progress JWellForm*
begin

2.22.1 Basic preservation lemmas

First two easy preservation lemmas.

theorem *red-preserves-hconf*:

$P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies (\bigwedge T E. \llbracket P, E, h \vdash e : T; P \vdash h \sqrt{\ } \rrbracket \implies P \vdash h' \sqrt{\ })$

and *reds-preserves-hconf*:

$P \vdash \langle es, (h, l) \rangle \rightarrow \langle es', (h', l') \rangle \implies (\bigwedge Ts E. \llbracket P, E, h \vdash es [:] Ts; P \vdash h \sqrt{\ } \rrbracket \implies P \vdash h' \sqrt{\ }) \langle proof \rangle$

theorem *red-preserves-lconf*:

$P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies$

$$(\wedge T E. \llbracket P, E, h \vdash e : T; P, h \vdash l (: \leq) E \rrbracket \Longrightarrow P, h' \vdash l' (: \leq) E)$$

and *reds-preserves-lconf*:

$$P \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \Longrightarrow$$

$$(\wedge Ts E. \llbracket P, E, h \vdash es [:] Ts; P, h \vdash l (: \leq) E \rrbracket \Longrightarrow P, h' \vdash l' (: \leq) E) \langle proof \rangle$$

Preservation of definite assignment more complex and requires a few lemmas first.

lemma [*iff*]: $\wedge A. \llbracket \text{length } Vs = \text{length } Ts; \text{length } vs = \text{length } Ts \rrbracket \Longrightarrow$

$$\mathcal{D} (\text{blocks } (Vs, Ts, vs, e)) A = \mathcal{D} e (A \sqcup [\text{set } Vs]) \langle proof \rangle$$

lemma *red-lA-incr*: $P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \Longrightarrow \llbracket \text{dom } l \rrbracket \sqcup \mathcal{A} e \sqsubseteq \llbracket \text{dom } l' \rrbracket \sqcup \mathcal{A} e'$

and *reds-lA-incr*: $P \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \Longrightarrow \llbracket \text{dom } l \rrbracket \sqcup \mathcal{A} s \text{ es} \sqsubseteq \llbracket \text{dom } l' \rrbracket \sqcup \mathcal{A} s \text{ es}' \langle proof \rangle$

Now preservation of definite assignment.

lemma *assumes wf*: *wf-J-prog* P

shows *red-preserves-defass*:

$$P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \Longrightarrow \mathcal{D} e \llbracket \text{dom } l \rrbracket \Longrightarrow \mathcal{D} e' \llbracket \text{dom } l' \rrbracket$$

and $P \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \Longrightarrow \mathcal{D} s \text{ es} \llbracket \text{dom } l \rrbracket \Longrightarrow \mathcal{D} s \text{ es}' \llbracket \text{dom } l' \rrbracket \langle proof \rangle$

Combining conformance of heap and local variables:

definition *sconf* :: $J\text{-prog} \Rightarrow \text{env} \Rightarrow \text{state} \Rightarrow \text{bool}$ (\neg, \vdash, \checkmark [51,51,51]50)

where

$$P, E \vdash s \checkmark \equiv \text{let } (h, l) = s \text{ in } P \vdash h \checkmark \wedge P, h \vdash l (: \leq) E$$

lemma *red-preserves-sconf*:

$$\llbracket P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle; P, E, hp \text{ s } \vdash e : T; P, E \vdash s \checkmark \rrbracket \Longrightarrow P, E \vdash s' \checkmark \langle proof \rangle$$

lemma *reds-preserves-sconf*:

$$\llbracket P \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle; P, E, hp \text{ s } \vdash es [:] Ts; P, E \vdash s \checkmark \rrbracket \Longrightarrow P, E \vdash s' \checkmark \langle proof \rangle$$

2.22.2 Subject reduction

lemma *wt-blocks*:

$$\wedge E. \llbracket \text{length } Vs = \text{length } Ts; \text{length } vs = \text{length } Ts \rrbracket \Longrightarrow$$

$$(P, E, h \vdash \text{blocks}(Vs, Ts, vs, e) : T) =$$

$$(P, E (Vs [\mapsto] Ts), h \vdash e : T \wedge (\exists Ts'. \text{map } (\text{typeof}_h) \text{ vs} = \text{map } \text{Some } Ts' \wedge P \vdash Ts' [\leq] Ts)) \langle proof \rangle$$

theorem *assumes wf*: *wf-J-prog* P

shows *subject-reduction2*: $P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \Longrightarrow$

$$(\wedge E T. \llbracket P, E \vdash (h, l) \checkmark; P, E, h \vdash e : T \rrbracket$$

$$\Longrightarrow \exists T'. P, E, h' \vdash e' : T' \wedge P \vdash T' \leq T)$$

and *subjects-reduction2*: $P \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \Longrightarrow$

$$(\wedge E Ts. \llbracket P, E \vdash (h, l) \checkmark; P, E, h \vdash es [:] Ts \rrbracket$$

$$\Longrightarrow \exists Ts'. P, E, h' \vdash es' [:] Ts' \wedge P \vdash Ts' [\leq] Ts) \langle proof \rangle$$

corollary *subject-reduction*:

$$\llbracket \text{wf-J-prog } P; P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle; P, E \vdash s \checkmark; P, E, hp \text{ s } \vdash e : T \rrbracket$$

$$\Longrightarrow \exists T'. P, E, hp \text{ s}' \vdash e' : T' \wedge P \vdash T' \leq T \langle proof \rangle$$

corollary *subjects-reduction*:

$$\llbracket \text{wf-J-prog } P; P \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle; P, E \vdash s \checkmark; P, E, hp \text{ s } \vdash es [:] Ts \rrbracket$$

$$\Longrightarrow \exists Ts'. P, E, hp \text{ s}' \vdash es' [:] Ts' \wedge P \vdash Ts' [\leq] Ts \langle proof \rangle$$

2.22.3 Lifting to \rightarrow^*

Now all these preservation lemmas are first lifted to the transitive closure ...

lemma *Red-preserves-sconf*:

assumes $wf: wf\text{-}J\text{-}prog\ P$ **and** $Red: P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle$
shows $\bigwedge T. \llbracket P, E, hp\ s \vdash e : T; P, E \vdash s \checkmark \rrbracket \Longrightarrow P, E \vdash s' \checkmark \langle proof \rangle$

lemma *Red-preserves-defass:*

assumes $wf: wf\text{-}J\text{-}prog\ P$ **and** $reds: P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle$
shows $\mathcal{D}\ e \ [dom(lcl\ s)] \Longrightarrow \mathcal{D}\ e' \ [dom(lcl\ s')]$
 $\langle proof \rangle$

lemma *Red-preserves-type:*

assumes $wf: wf\text{-}J\text{-}prog\ P$ **and** $Red: P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle$
shows $!!T. \llbracket P, E \vdash s \checkmark; P, E, hp\ s \vdash e : T \rrbracket$
 $\Longrightarrow \exists T'. P \vdash T' \leq T \wedge P, E, hp\ s' \vdash e' : T' \langle proof \rangle$

2.22.4 Lifting to \Rightarrow

... and now to the big step semantics, just for fun.

lemma *eval-preserves-sconf:*

$\llbracket wf\text{-}J\text{-}prog\ P; P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle; P, E \vdash e :: T; P, E \vdash s \checkmark \rrbracket \Longrightarrow P, E \vdash s' \checkmark \langle proof \rangle$

lemma *eval-preserves-type:* **assumes** $wf: wf\text{-}J\text{-}prog\ P$

shows $\llbracket P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle; P, E \vdash s \checkmark; P, E \vdash e :: T \rrbracket$
 $\Longrightarrow \exists T'. P \vdash T' \leq T \wedge P, E, hp\ s' \vdash e' : T' \langle proof \rangle$

2.22.5 The final polish

The above preservation lemmas are now combined and packed nicely.

definition $wf\text{-}config :: J\text{-}prog \Rightarrow env \Rightarrow state \Rightarrow expr \Rightarrow ty \Rightarrow bool$ $(-, -, \vdash -, - \checkmark \ [51, 0, 0, 0, 0] 50)$
where

$P, E, s \vdash e : T \checkmark \equiv P, E \vdash s \checkmark \wedge P, E, hp\ s \vdash e : T$

theorem *Subject-reduction:* **assumes** $wf: wf\text{-}J\text{-}prog\ P$

shows $P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \Longrightarrow P, E, s \vdash e : T \checkmark$
 $\Longrightarrow \exists T'. P, E, s' \vdash e' : T' \checkmark \wedge P \vdash T' \leq T \langle proof \rangle$

theorem *Subject-reductions:*

assumes $wf: wf\text{-}J\text{-}prog\ P$ **and** $reds: P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle$
shows $\bigwedge T. P, E, s \vdash e : T \checkmark \Longrightarrow \exists T'. P, E, s' \vdash e' : T' \checkmark \wedge P \vdash T' \leq T \langle proof \rangle$

corollary *Progress:* **assumes** $wf: wf\text{-}J\text{-}prog\ P$

shows $\llbracket P, E, s \vdash e : T \checkmark; \mathcal{D}\ e \ [dom(lcl\ s)]; \neg\ final\ e \rrbracket \Longrightarrow \exists e' s'. P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \langle proof \rangle$

corollary *TypeSafety:*

fixes $s :: state$

assumes $wf: wf\text{-}J\text{-}prog\ P$ **and** $sconf: P, E \vdash s \checkmark$ **and** $wt: P, E \vdash e :: T$

and $\mathcal{D}: \mathcal{D}\ e \ [dom(lcl\ s)]$

and $steps: P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle$

and $nstep: \neg(\exists e'' s''. P \vdash \langle e', s' \rangle \rightarrow \langle e'', s'' \rangle)$

shows $(\exists v. e' = Val\ v \wedge P, hp\ s' \vdash v : \leq T) \vee$
 $(\exists a. e' = Throw\ a \wedge a \in dom(hp\ s')) \langle proof \rangle$

end

2.23 Program annotation

theory *Annotate* **imports** *WellType* **begin**

inductive

Anno :: [*J-prog*, *env*, *expr* → *expr*] ⇒ *bool*
 (·, · ⊢ · ∼ · - [51,0,0,51]50)

and *Annos* :: [*J-prog*, *env*, *expr list*, *expr list*] ⇒ *bool*
 (·, · ⊢ · [↗] - [51,0,0,51]50)

for *P* :: *J-prog*

where

AnnoNew: $P, E \vdash \text{new } C \rightsquigarrow \text{new } C$

| *AnnoCast*: $P, E \vdash e \rightsquigarrow e' \implies P, E \vdash \text{Cast } C \ e \rightsquigarrow \text{Cast } C \ e'$

| *AnnoVal*: $P, E \vdash \text{Val } v \rightsquigarrow \text{Val } v$

| *AnnoVarVar*: $E \ V = [T] \implies P, E \vdash \text{Var } V \rightsquigarrow \text{Var } V$

| *AnnoVarField*: $\llbracket E \ V = \text{None}; E \ \text{this} = [\text{Class } C]; P \vdash C \ \text{sees } V:T \ \text{in } D \rrbracket$
 $\implies P, E \vdash \text{Var } V \rightsquigarrow \text{Var } \text{this} \cdot V\{D\}$

| *AnnoBinOp*:

$\llbracket P, E \vdash e1 \rightsquigarrow e1'; P, E \vdash e2 \rightsquigarrow e2' \rrbracket$
 $\implies P, E \vdash e1 \ \langle\langle \text{bop} \rangle\rangle \ e2 \rightsquigarrow e1' \ \langle\langle \text{bop} \rangle\rangle \ e2'$

| *AnnoLAssVar*:

$\llbracket E \ V = [T]; P, E \vdash e \rightsquigarrow e' \rrbracket \implies P, E \vdash V := e \rightsquigarrow V := e'$

| *AnnoLAssField*:

$\llbracket E \ V = \text{None}; E \ \text{this} = [\text{Class } C]; P \vdash C \ \text{sees } V:T \ \text{in } D; P, E \vdash e \rightsquigarrow e' \rrbracket$
 $\implies P, E \vdash V := e \rightsquigarrow \text{Var } \text{this} \cdot V\{D\} := e'$

| *AnnoFAcc*:

$\llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash e' :: \text{Class } C; P \vdash C \ \text{sees } F:T \ \text{in } D \rrbracket$
 $\implies P, E \vdash e \cdot F\{\} \rightsquigarrow e' \cdot F\{D\}$

| *AnnoFAss*: $\llbracket P, E \vdash e1 \rightsquigarrow e1'; P, E \vdash e2 \rightsquigarrow e2';$

$P, E \vdash e1' :: \text{Class } C; P \vdash C \ \text{sees } F:T \ \text{in } D \rrbracket$

$\implies P, E \vdash e1 \cdot F\{\} := e2 \rightsquigarrow e1' \cdot F\{D\} := e2'$

| *AnnoCall*:

$\llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash es \ [\rightsquigarrow] \ es' \rrbracket$
 $\implies P, E \vdash \text{Call } e \ M \ es \rightsquigarrow \text{Call } e' \ M \ es'$

| *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*: $P, E \vdash [] \ [\rightsquigarrow] \ []$

| *AnnoCons*: $\llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash es \ [\rightsquigarrow] \ es' \rrbracket$

$\implies P, E \vdash e \# es \ [\rightsquigarrow] \ e' \# es'$

end

2.24 Example Expressions

theory *Examples* **imports** *Expr* **begin**

definition *classObject* :: *J-mb cdecl*

where

```
classObject == ("Object", "", [], [])
```

definition *classI* :: *J-mb cdecl*

where

```
classI ==
("I", Object,
 [],
 [{"mult", [Integer, Integer], Integer, ["i", "j"]},
  if (Var "i" «Eq» Val(Intg 0)) (Val(Intg 0))
  else Var "j" «Add»
   Var this · "mult"([Var "i" «Add» Val(Intg (- 1)), Var "j"])]
])
```

definition *classL* :: *J-mb cdecl*

where

```
classL ==
("L", Object,
 [{"F", Integer}, ("N", Class "L")],
 [{"app", [Class "L"], Void, ["l"]},
  if (Var this · "N"{"L"} «Eq» null)
   (Var this · "N"{"L"} := Var "l")
  else (Var this · "N"{"L"} · "app"([Var "l"])]
])
```

definition *testExpr-BuildList* :: *expr*

where

```
testExpr-BuildList ==
{"l1": Class "L" := new "L";
 Var "l1" · "F"{"L"} := Val(Intg 1)};
{"l2": Class "L" := new "L";
 Var "l2" · "F"{"L"} := Val(Intg 2)};
{"l3": Class "L" := new "L";
 Var "l3" · "F"{"L"} := Val(Intg 3)};
{"l4": Class "L" := new "L";
 Var "l4" · "F"{"L"} := Val(Intg 4)};
Var "l1" · "app"([Var "l2"]);
Var "l1" · "app"([Var "l3"]);
Var "l1" · "app"([Var "l4"])}}}
```

definition *testExpr1* :: *expr*

where

```
testExpr1 == Val(Intg 5)
```

definition *testExpr2* :: *expr*

where

```
testExpr2 == BinOp (Val(Intg 5)) Add (Val(Intg 6))
```

definition *testExpr3* :: *expr*

where

testExpr3 == *BinOp* (*Var* "V") *Add* (*Val*(*Intg* 6))

definition *testExpr4* :: *expr*

where

testExpr4 == "V" := *Val*(*Intg* 6)

definition *testExpr5* :: *expr*

where

testExpr5 == *new* "Object"; { "V":(*Class* "C") := *new* "C"; *Var* "V"."F"{"C"} := *Val*(*Intg* 42)}

definition *testExpr6* :: *expr*

where

testExpr6 == { "V":(*Class* "I") := *new* "I"; *Var* "V"."mult"([*Val*(*Intg* 40),*Val*(*Intg* 4)])}

definition *mb-isNull*:: *expr*

where

mb-isNull == *Var* *this* · "test"{"A"} «Eq» *null*

definition *mb-add*:: *expr*

where

mb-add == (*Var* *this* · "int"{"A"} := (*Var* *this* · "int"{"A"} «Add» *Var* "i")); (*Var* *this* · "int"{"A"})

definition *mb-mult-cond*:: *expr*

where

mb-mult-cond == (*Var* "j" «Eq» *Val* (*Intg* 0)) «Eq» *Val* (*Bool* *False*)

definition *mb-mult-block*:: *expr*

where

mb-mult-block == "temp":=(*Var* "temp" «Add» *Var* "i"); "j":=(*Var* "j" «Add» *Val* (*Intg* (- 1)))

definition *mb-mult*:: *expr*

where

mb-mult == { "temp":*Integer*:=*Val* (*Intg* 0); *While* (*mb-mult-cond*) *mb-mult-block*; (*Var* *this* · "int"{"A"} := *Var* "temp"; *Var* "temp")}

definition *classA*:: *J-mb cdecl*

where

classA ==
 ("A", *Object*,
 [("int",*Integer*),
 ("test",*Class* "A")],
 [("isNull",[],*Boolean*,[], *mb-isNull*),
 ("add",[*Integer*],*Integer*,["i"], *mb-add*),
 ("mult",[*Integer*,*Integer*],*Integer*,["i","j"], *mb-mult*)])

definition *testExpr-ClassA*:: *expr*

where

testExpr-ClassA ==
 {"A1":*Class* "A":= *new* "A";
 {"A2":*Class* "A":= *new* "A";
 {"testint":*Integer*:= *Val* (*Intg* 5);
 (*Var* "A2" · "int"{"A"} := (*Var* "A1" · "add"([*Var* "testint"]));

```
(Var "A2". "int"{"A"}) := (Var "A1". "add"([Var "testint"]));
Var "A2". "mult"([Var "A2". "int"{"A"}, Var "testint"] )}}
```

end

2.25 Code Generation For BigStep

theory *execute-Bigstep*

imports

BigStep Examples

HOL-Library.Code-Target-Numeral

begin

inductive *map-val* :: *expr list* \Rightarrow *val list* \Rightarrow *bool*

where

Nil: *map-val* [] []

| *Cons*: *map-val* *xs ys* \Longrightarrow *map-val* (Val *y* # *xs*) (*y* # *ys*)

inductive *map-val2* :: *expr list* \Rightarrow *val list* \Rightarrow *expr list* \Rightarrow *bool*

where

Nil: *map-val2* [] [] []

| *Cons*: *map-val2* *xs ys zs* \Longrightarrow *map-val2* (Val *y* # *xs*) (*y* # *ys*) *zs*

| *Throw*: *map-val2* (*throw e* # *xs*) [] (*throw e* # *xs*)

theorem *map-val-conv*: (*xs* = *map Val ys*) = *map-val* *xs ys* <proof>

theorem *map-val2-conv*:

(*xs* = *map Val ys* @ *throw e* # *zs*) = *map-val2* *xs ys* (*throw e* # *zs*) <proof>

lemma *CallNull2*:

$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle; P \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle evs, s_2 \rangle; \text{map-val } evs \ vs \rrbracket$

$\Longrightarrow P \vdash \langle e \cdot M(ps), s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_2 \rangle$

<proof>

lemma *CallParamsThrow2*:

$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash \langle es, s_1 \rangle [\Rightarrow] \langle evs, s_2 \rangle;$

$\text{map-val2 } evs \ vs \ (\text{throw } ex \ \# \ es'') \rrbracket$

$\Longrightarrow P \vdash \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle \text{throw } ex, s_2 \rangle$

<proof>

lemma *Call2*:

$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle evs, (h_2, l_2) \rangle;$

$\text{map-val } evs \ vs;$

$h_2 \ a = \text{Some}(C, fs); P \vdash C \ \text{sees } M: Ts \rightarrow T = (pns, body) \ \text{in } D;$

$\text{length } vs = \text{length } pns; l_2' = [\text{this} \mapsto \text{Addr } a, pns[\mapsto] vs];$

$P \vdash \langle body, (h_2, l_2') \rangle \Rightarrow \langle e', (h_3, l_3) \rangle \rrbracket$

$\Longrightarrow P \vdash \langle e \cdot M(ps), s_0 \rangle \Rightarrow \langle e', (h_3, l_2) \rangle$

<proof>

code-pred

(*modes*: *i* \Rightarrow *o* \Rightarrow *bool*)

map-val

<proof>

code-pred

(modes: $i \Rightarrow o \Rightarrow o \Rightarrow \text{bool}$)
 map-val2
 <proof>

lemmas [code-pred-intro] =

eval-vals.NewFail
 eval-vals.Cast eval-vals.CastNull eval-vals.CastFail eval-vals.CastThrow
 eval-vals.Val eval-vals.Var
 eval-vals.BinOp eval-vals.BinOpThrow1 eval-vals.BinOpThrow2
 eval-vals.LAss eval-vals.LAssThrow
 eval-vals.FAcc eval-vals.FAccNull eval-vals.FAccThrow
 eval-vals.FAss eval-vals.FAssNull
 eval-vals.FAssThrow1 eval-vals.FAssThrow2
 eval-vals.CallObjThrow

declare CallNull2 [code-pred-intro CallNull2]**declare** CallParamsThrow2 [code-pred-intro CallParamsThrow2]**declare** Call2 [code-pred-intro Call2]**lemmas** [code-pred-intro] =

eval-vals.Block
 eval-vals.Seq eval-vals.SeqThrow
 eval-vals.CondT eval-vals.CondF eval-vals.CondThrow
 eval-vals.WhileF eval-vals.WhileT
 eval-vals.WhileCondThrow

declare eval-vals.WhileBodyThrow [code-pred-intro WhileBodyThrow2]**lemmas** [code-pred-intro] =

eval-vals.Throw eval-vals.ThrowNull
 eval-vals.ThrowThrow
 eval-vals.Try eval-vals.TryCatch eval-vals.TryThrow
 eval-vals.Nil eval-vals.Cons eval-vals.ConsThrow

code-pred

(modes: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow \text{bool}$ as execute)
 eval
 <proof>

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

definition test1 = $\square \vdash \langle \text{testExpr1}, (\text{Map.empty}, \text{Map.empty}) \rangle \Rightarrow \langle -, - \rangle$

definition test2 = $\square \vdash \langle \text{testExpr2}, (\text{Map.empty}, \text{Map.empty}) \rangle \Rightarrow \langle -, - \rangle$

definition test3 = $\square \vdash \langle \text{testExpr3}, (\text{Map.empty}, \text{Map.empty}(\text{"V"} \mapsto \text{Intg } 77)) \rangle \Rightarrow \langle -, - \rangle$

definition test4 = $\square \vdash \langle \text{testExpr4}, (\text{Map.empty}, \text{Map.empty}) \rangle \Rightarrow \langle -, - \rangle$

definition test5 = $[(\text{"Object"}, (\text{"", []}), (\text{"C"}, (\text{"Object"}, (\text{"F"}, \text{Integer}), [])))] \vdash \langle \text{testExpr5}, (\text{Map.empty}, \text{Map.empty}) \rangle \Rightarrow \langle -, - \rangle$

definition test6 = $[(\text{"Object"}, (\text{"", []}), \text{classI} \vdash \langle \text{testExpr6}, (\text{Map.empty}, \text{Map.empty}) \rangle \Rightarrow \langle -, - \rangle$

definition V = "V"

definition C = "C"

definition F = "F"

⟨ML⟩

definition $test7 = [classObject, classL] \vdash \langle testExpr-BuildList, (Map.empty, Map.empty) \rangle \Rightarrow \langle -, - \rangle$

definition $L = "L"$

definition $N = "N"$

⟨ML⟩

definition $test8 = [classObject, classA] \vdash \langle testExpr-ClassA, (Map.empty, Map.empty) \rangle \Rightarrow \langle -, - \rangle$

definition $i = "int"$

definition $t = "test"$

definition $A = "A"$

⟨ML⟩

end

2.26 Code Generation For WellType

theory *execute-WellType*

imports

WellType Examples

begin

lemma *WTCond1*:

$$\llbracket P, E \vdash e :: Boolean; P, E \vdash e_1 :: T_1; P, E \vdash e_2 :: T_2; P \vdash T_1 \leq T_2; \\ P \vdash T_2 \leq T_1 \longrightarrow T_2 = T_1 \rrbracket \Longrightarrow P, E \vdash \text{if } (e) \ e_1 \ \text{else } e_2 :: T_2$$

⟨proof⟩

lemma *WTCond2*:

$$\llbracket P, E \vdash e :: Boolean; P, E \vdash e_1 :: T_1; P, E \vdash e_2 :: T_2; P \vdash T_2 \leq T_1; \\ P \vdash T_1 \leq T_2 \longrightarrow T_1 = T_2 \rrbracket \Longrightarrow P, E \vdash \text{if } (e) \ e_1 \ \text{else } e_2 :: T_1$$

⟨proof⟩

lemmas [*code-pred-intro*] =

WT-WTs.WTNew

WT-WTs.WTCast

WT-WTs.WTVal

WT-WTs.WTVar

WT-WTs.WTBinOpEq

WT-WTs.WTBinOpAdd

WT-WTs.WTLAss

WT-WTs.WTFAcc

WT-WTs.WTFAss

WT-WTs.WTCall

WT-WTs.WTBlock

WT-WTs.WTSeq

declare

WTCond1 [*code-pred-intro WTCond1*]

WTCond2 [*code-pred-intro* *WTCond2*]

lemmas [*code-pred-intro*] =

WT-WTs.WTWhile
WT-WTs.WTThrow
WT-WTs.WTTry
WT-WTs.WTNil
WT-WTs.WTCons

code-pred

(*modes: i ⇒ i ⇒ i ⇒ i ⇒ bool as type-check, i ⇒ i ⇒ i ⇒ o ⇒ bool as infer-type*)

WT

⟨*proof*⟩

notation *infer-type* (-, - ⊢ - :: ' - [51,51,51]100)

definition *test1* **where** *test1* = [], *Map.empty* ⊢ *testExpr1* :: -

definition *test2* **where** *test2* = [], *Map.empty* ⊢ *testExpr2* :: -

definition *test3* **where** *test3* = [], *Map.empty*("V" ↦ *Integer*) ⊢ *testExpr3* :: -

definition *test4* **where** *test4* = [], *Map.empty*("V" ↦ *Integer*) ⊢ *testExpr4* :: -

definition *test5* **where** *test5* = [*classObject*, ("C", ("Object", [("F", *Integer*), []])), *Map.empty* ⊢ *testExpr5* :: -

definition *test6* **where** *test6* = [*classObject*, *classI*], *Map.empty* ⊢ *testExpr6* :: -

⟨*ML*⟩

definition *testmb-isNull* **where** *testmb-isNull* = [*classObject*, *classA*], *Map.empty*([*this*] [↦] [*Class* "A"]) ⊢ *mb-isNull* :: -

definition *testmb-add* **where** *testmb-add* = [*classObject*, *classA*], *Map.empty*([*this*, "i"] [↦] [*Class* "A", *Integer*]) ⊢ *mb-add* :: -

definition *testmb-mult-cond* **where** *testmb-mult-cond* = [*classObject*, *classA*], *Map.empty*([*this*, "j"] [↦] [*Class* "A", *Integer*]) ⊢ *mb-mult-cond* :: -

definition *testmb-mult-block* **where** *testmb-mult-block* = [*classObject*, *classA*], *Map.empty*([*this*, "i", "j", "temp"] [↦] [*Class* "A", *Integer*, *Integer*, *Integer*]) ⊢ *mb-mult-block* :: -

definition *testmb-mult* **where** *testmb-mult* = [*classObject*, *classA*], *Map.empty*([*this*, "i", "j"] [↦] [*Class* "A", *Integer*, *Integer*]) ⊢ *mb-mult* :: -

⟨*ML*⟩

definition *test* **where** *test* = [*classObject*, *classA*], *Map.empty* ⊢ *testExpr-ClassA* :: -

⟨*ML*⟩

end

Chapter 3

Jinja Virtual Machine

3.1 State of the JVM

```
theory JVMState imports ../Common/Objects begin
```

3.1.1 Frame Stack

```
type-synonym
```

```
pc = nat
```

```
type-synonym
```

```
frame = val list × val list × cname × mname × pc
```

— operand stack

— registers (including this pointer, method parameters, and local variables)

— name of class where current method is defined

— parameter types

— program counter within frame

3.1.2 Runtime State

```
type-synonym
```

```
jvm-state = addr option × heap × frame list
```

— exception flag, heap, frames

```
end
```

3.2 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
```

```
| Putfield vname cname — Set field in object
```

```
| Checkcast cname — Check whether object is of given type
```

```
| Invoke mname nat — inv. instance meth of an object
```

<i>Return</i>	— return from method
<i>Pop</i>	— 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

3.3 JVM Instruction Semantics

theory *JVMExecInstr*

imports *JVMInstructions JVMState ../Common/Exceptions*

begin

primrec

exec-instr :: [*instr*, *jvm-prog*, *heap*, *val list*, *val list*,
cname, *mname*, *pc*, *frame list*] => *jvm-state*

where

exec-instr-Load:

exec-instr (Load n) P h stk loc C₀ M₀ pc frs =
(None, h, ((loc ! n) # stk, loc, C₀, M₀, pc+1)#frs)

| *exec-instr (Store n) P h stk loc C₀ M₀ pc frs* =
(None, h, (tl stk, loc[n:=hd stk], C₀, M₀, pc+1)#frs)

| *exec-instr-Push*:

exec-instr (Push v) P h stk loc C₀ M₀ pc frs =
(None, h, (v # stk, loc, C₀, M₀, pc+1)#frs)

| *exec-instr-New*:

exec-instr (New C) P h stk loc C₀ M₀ pc frs =

(case new-Addr h of
 None \Rightarrow (Some (addr-of-sys-xcpt OutOfMemory), h, (stk, loc, C₀, M₀, pc)#frs)
 | Some a \Rightarrow (None, h(a \mapsto blank P C), (Addr a#stk, loc, C₀, M₀, pc+1)#frs))

| exec-instr (Getfield F C) P h stk loc C₀ M₀ pc frs =
 (let v = hd stk;
 xp' = if v=Null then [addr-of-sys-xcpt NullPointer] else None;
 (D,fs) = the(h(the-Addr v))
 in (xp', h, (the(fs(F,C))#(tl stk), loc, C₀, M₀, pc+1)#frs))

| exec-instr (Putfield F C) P h stk loc C₀ M₀ pc frs =
 (let v = hd stk;
 r = hd (tl stk);
 xp' = if r=Null then [addr-of-sys-xcpt NullPointer] else None;
 a = the-Addr r;
 (D,fs) = the (h a);
 h' = h(a \mapsto (D, fs((F,C) \mapsto v)))
 in (xp', h', (tl (tl stk), loc, C₀, M₀, pc+1)#frs))

| exec-instr (Checkcast C) P h stk loc C₀ M₀ pc frs =
 (let v = hd stk;
 xp' = if \neg cast-ok P C h v then [addr-of-sys-xcpt ClassCast] else None
 in (xp', h, (stk, loc, C₀, M₀, pc+1)#frs))

| exec-instr-Invoke:
 exec-instr (Invoke M n) P h stk loc C₀ M₀ pc frs =
 (let ps = take n stk;
 r = stk!n;
 xp' = if r=Null then [addr-of-sys-xcpt NullPointer] else None;
 C = fst(the(h(the-Addr r)));
 (D,M',Ts,ms,mxl₀,ins,xt) = method P C M;
 f' = ([, [r]@ (rev ps)@ (replicate mxl₀ undefined), D, M, 0])
 in (xp', h, f'#(stk, loc, C₀, M₀, pc)#frs))

| exec-instr Return P h stk loc₀ C₀ M₀ pc frs =
 (if frs=[] then (None, h, []) else
 let v = hd stk₀;
 (stk,loc,C,m,pc) = hd frs;
 n = length (fst (snd (method P C₀ M₀)))
 in (None, h, (v#(drop (n+1) stk),loc,C,m,pc+1)#tl frs))

| exec-instr Pop P h stk loc C₀ M₀ pc frs =
 (None, h, (tl stk, loc, C₀, M₀, pc+1)#frs)

| exec-instr IAdd P h stk loc C₀ M₀ pc frs =
 (let i₂ = the-Intg (hd stk);
 i₁ = the-Intg (hd (tl stk))
 in (None, h, (Intg (i₁+i₂)#(tl (tl stk)), loc, C₀, M₀, pc+1)#frs))

| exec-instr (IfFalse i) P h stk loc C₀ M₀ pc frs =
 (let pc' = if hd stk = Bool False then nat(int pc+i) else pc+1
 in (None, h, (tl stk, loc, C₀, M₀, pc')#frs))

| exec-instr CmpEq P h stk loc C₀ M₀ pc frs =

(let $v_2 = \text{hd } \text{stk}$;
 $v_1 = \text{hd } (\text{tl } \text{stk})$
in $(\text{None}, h, (\text{Bool } (v_1=v_2) \# \text{tl } (\text{tl } \text{stk}), \text{loc}, C_0, M_0, \text{pc}+1)\#\text{frs}))$)

| *exec-instr-Goto*:

exec-instr $(\text{Goto } i) P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} =$
 $(\text{None}, h, (\text{stk}, \text{loc}, C_0, M_0, \text{nat}(\text{int } \text{pc}+i))\#\text{frs})$

| *exec-instr Throw* $P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} =$

(let $xp' = \text{if } \text{hd } \text{stk} = \text{Null} \text{ then } \lfloor \text{addr-of-sys-xcpt } \text{NullPointer} \rfloor \text{ else } \lfloor \text{the-Addr}(\text{hd } \text{stk}) \rfloor$
in $(xp', h, (\text{stk}, \text{loc}, C_0, M_0, \text{pc})\#\text{frs}))$)

lemma *exec-instr-Store*:

exec-instr $(\text{Store } n) P h (v\#\text{stk}) \text{loc } C_0 M_0 \text{ pc } \text{frs} =$
 $(\text{None}, h, (\text{stk}, \text{loc}[n:=v], C_0, M_0, \text{pc}+1)\#\text{frs})$
 $\langle \text{proof} \rangle$

lemma *exec-instr-Getfield*:

exec-instr $(\text{Getfield } F C) P h (v\#\text{stk}) \text{loc } C_0 M_0 \text{ pc } \text{frs} =$
(let $xp' = \text{if } v = \text{Null} \text{ then } \lfloor \text{addr-of-sys-xcpt } \text{NullPointer} \rfloor \text{ else } \text{None}$;
 $(D, \text{fs}) = \text{the}(h(\text{the-Addr } v))$
in $(xp', h, (\text{the}(\text{fs}(F, C))\#\text{stk}, \text{loc}, C_0, M_0, \text{pc}+1)\#\text{frs}))$
 $\langle \text{proof} \rangle$

lemma *exec-instr-Putfield*:

exec-instr $(\text{Putfield } F C) P h (v\#r\#\text{stk}) \text{loc } C_0 M_0 \text{ pc } \text{frs} =$
(let $xp' = \text{if } r = \text{Null} \text{ then } \lfloor \text{addr-of-sys-xcpt } \text{NullPointer} \rfloor \text{ else } \text{None}$;
 $a = \text{the-Addr } r$;
 $(D, \text{fs}) = \text{the } (h a)$;
 $h' = h(a \mapsto (D, \text{fs}((F, C) \mapsto v)))$
in $(xp', h', (\text{stk}, \text{loc}, C_0, M_0, \text{pc}+1)\#\text{frs}))$
 $\langle \text{proof} \rangle$

lemma *exec-instr-Checkcast*:

exec-instr $(\text{Checkcast } C) P h (v\#\text{stk}) \text{loc } C_0 M_0 \text{ pc } \text{frs} =$
(let $xp' = \text{if } \neg \text{cast-ok } P C h v \text{ then } \lfloor \text{addr-of-sys-xcpt } \text{ClassCast} \rfloor \text{ else } \text{None}$
in $(xp', h, (v\#\text{stk}, \text{loc}, C_0, M_0, \text{pc}+1)\#\text{frs}))$
 $\langle \text{proof} \rangle$

lemma *exec-instr-Return*:

exec-instr $\text{Return } P h (v\#\text{stk}_0) \text{loc}_0 C_0 M_0 \text{ pc } \text{frs} =$
(if $\text{frs} = []$ then $(\text{None}, h, [])$ else
let $(\text{stk}, \text{loc}, C, m, \text{pc}) = \text{hd } \text{frs}$;
 $n = \text{length } (\text{fst } (\text{snd } (\text{method } P C_0 M_0)))$
in $(\text{None}, h, (v\#(\text{drop } (n+1) \text{stk}), \text{loc}, C, m, \text{pc}+1)\#\text{tl } \text{frs}))$
 $\langle \text{proof} \rangle$

lemma *exec-instr-IPop*:

exec-instr $\text{Pop } P h (v\#\text{stk}) \text{loc } C_0 M_0 \text{ pc } \text{frs} =$
 $(\text{None}, h, (\text{stk}, \text{loc}, C_0, M_0, \text{pc}+1)\#\text{frs})$
 $\langle \text{proof} \rangle$

lemma *exec-instr-IAdd*:

exec-instr IAdd $P h (Intg\ i_2 \# Intg\ i_1 \# stk) loc\ C_0\ M_0\ pc\ frs =$
 $(None, h, (Intg\ (i_1+i_2)\#stk, loc, C_0, M_0, pc+1)\#frs)$
 ⟨proof⟩

lemma *exec-instr-IfFalse*:

exec-instr IfFalse $i P h (v\#stk) loc\ C_0\ M_0\ pc\ frs =$
 $(let\ pc' = if\ v = Bool\ False\ then\ nat(int\ pc+i)\ else\ pc+1$
 $in\ (None, h, (stk, loc, C_0, M_0, pc')\#frs))$
 ⟨proof⟩

lemma *exec-instr-CmpEq*:

exec-instr CmpEq $P h (v_2\#v_1\#stk) loc\ C_0\ M_0\ pc\ frs =$
 $(None, h, (Bool\ (v_1=v_2) \# stk, loc, C_0, M_0, pc+1)\#frs)$
 ⟨proof⟩

lemma *exec-instr-Throw*:

exec-instr Throw $P h (v\#stk) loc\ C_0\ M_0\ pc\ frs =$
 $(let\ xp' = if\ v = Null\ then\ [addr-of-sys-xcpt\ NullPointer]\ else\ [the-Addr\ v]$
 $in\ (xp', h, (v\#stk, loc, C_0, M_0, pc)\#frs))$
 ⟨proof⟩

end

3.4 Exception handling in the JVM

theory *JVMExceptions* **imports** *JVMInstructions* *../Common/Exceptions* **begin**

definition *matches-ex-entry* $:: 'm\ prog \Rightarrow cname \Rightarrow pc \Rightarrow ex-entry \Rightarrow bool$

where

matches-ex-entry $P\ C\ pc\ xcp \equiv$
 $let\ (s, e, C', h, d) = xcp\ in$
 $s \leq pc \wedge pc < e \wedge P \vdash C \preceq^* C'$

primrec *match-ex-table* $:: 'm\ prog \Rightarrow cname \Rightarrow pc \Rightarrow ex-table \Rightarrow (pc \times nat)\ option$

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 $:: jvm-prog \Rightarrow cname \Rightarrow mname \Rightarrow ex-table$ **where**
 $ex-table-of\ P\ C\ M == snd\ (snd\ (snd\ (snd\ (snd\ (snd\ (method\ P\ C\ M))))))$

primrec *find-handler* $:: jvm-prog \Rightarrow addr \Rightarrow heap \Rightarrow frame\ list \Rightarrow jvm-state$

where

find-handler $P\ a\ h\ [] = (Some\ a, h, [])$
 $| find-handler\ P\ a\ h\ (fr\#frs) =$
 $(let\ (stk, loc, C, M, pc) = fr\ in$
 $case\ match-ex-table\ P\ (cname-of\ h\ a)\ pc\ (ex-table-of\ P\ C\ M)\ of$
 $None \Rightarrow find-handler\ P\ a\ h\ frs$

| *Some pc-d* \Rightarrow (*None*, *h*, (*Addr a # drop (size stk - snd pc-d) stk*, *loc*, *C*, *M*, *fst pc-d*)#*frs*)

end

3.5 Program Execution in the JVM

theory *JVMExec*

imports *JVMExecInstr JVMExceptions*

begin

abbreviation

instrs-of :: *jvm-prog* \Rightarrow *cname* \Rightarrow *mname* \Rightarrow *instr list* **where**
instrs-of *P C M* == *fst(snd(snd(snd(snd(snd(method P C M))))))*)

fun *exec* :: *jvm-prog* \times *jvm-state* \Rightarrow *jvm-state option* **where** — single step execution

exec (*P*, *xp*, *h*, []) = *None*

| *exec* (*P*, *None*, *h*, (*stk,loc,C,M,pc*)#*frs*) =
 (let
 i = *instrs-of P C M ! pc*;
 (*xcpt'*, *h'*, *frs'*) = *exec-instr i P h stk loc C M pc frs*
 in *Some(case xcpt' of*
 None \Rightarrow (*None,h',frs'*)
 | *Some a* \Rightarrow *find-handler P a h ((stk,loc,C,M,pc)#frs)*)

| *exec* (*P*, *Some xa*, *h*, *frs*) = *None*

— relational view

inductive-set

exec-1 :: *jvm-prog* \Rightarrow (*jvm-state* \times *jvm-state*) *set*
and *exec-1'* :: *jvm-prog* \Rightarrow *jvm-state* \Rightarrow *jvm-state* \Rightarrow *bool*
 (- \vdash / - - *jvm* \rightarrow_1 / - [61,61,61] 60)
for *P* :: *jvm-prog*

where

P \vdash σ -*jvm* \rightarrow_1 σ' \equiv (σ, σ') \in *exec-1 P*

| *exec-1I*: *exec* (*P*, σ) = *Some* σ' \Longrightarrow *P* \vdash σ -*jvm* \rightarrow_1 σ'

— reflexive transitive closure:

definition *exec-all* :: *jvm-prog* \Rightarrow *jvm-state* \Rightarrow *jvm-state* \Rightarrow *bool*

((- \vdash / - - *jvm* \rightarrow / -) [61,61,61] 60) **where**

exec-all-def1: *P* \vdash σ -*jvm* \rightarrow σ' \iff (σ, σ') \in (*exec-1 P*)^{*}

notation (*ASCII*)

exec-all (- \vdash / - - *jvm* \rightarrow / - [61,61,61] 60)

lemma *exec-1-eg*:

exec-1 P = {(σ, σ'). *exec* (*P*, σ) = *Some* σ' } \langle proof \rangle

lemma *exec-1-iff*:

P \vdash σ -*jvm* \rightarrow_1 σ' = (*exec* (*P*, σ) = *Some* σ') \langle proof \rangle

lemma *exec-all-def*:

P \vdash σ -*jvm* \rightarrow σ' = ((σ, σ') \in {(σ, σ'). *exec* (*P*, σ) = *Some* σ' }^{*}) \langle proof \rangle

lemma *jvm-refl*[*iff*]: $P \vdash \sigma \text{-jvm} \rightarrow \sigma \langle \text{proof} \rangle$
lemma *jvm-trans*[*trans*]:
 $\llbracket P \vdash \sigma \text{-jvm} \rightarrow \sigma'; P \vdash \sigma' \text{-jvm} \rightarrow \sigma'' \rrbracket \Longrightarrow P \vdash \sigma \text{-jvm} \rightarrow \sigma'' \langle \text{proof} \rangle$
lemma *jvm-one-step1*[*trans*]:
 $\llbracket P \vdash \sigma \text{-jvm} \rightarrow_1 \sigma'; P \vdash \sigma' \text{-jvm} \rightarrow \sigma'' \rrbracket \Longrightarrow P \vdash \sigma \text{-jvm} \rightarrow \sigma'' \langle \text{proof} \rangle$
lemma *jvm-one-step2*[*trans*]:
 $\llbracket P \vdash \sigma \text{-jvm} \rightarrow \sigma'; P \vdash \sigma' \text{-jvm} \rightarrow_1 \sigma'' \rrbracket \Longrightarrow P \vdash \sigma \text{-jvm} \rightarrow \sigma'' \langle \text{proof} \rangle$
lemma *exec-all-conf*:
 $\llbracket P \vdash \sigma \text{-jvm} \rightarrow \sigma'; P \vdash \sigma \text{-jvm} \rightarrow \sigma'' \rrbracket$
 $\Longrightarrow P \vdash \sigma' \text{-jvm} \rightarrow \sigma'' \vee P \vdash \sigma'' \text{-jvm} \rightarrow \sigma' \langle \text{proof} \rangle$
lemma *exec-all-finalD*: $P \vdash (x, h, []) \text{-jvm} \rightarrow \sigma \Longrightarrow \sigma = (x, h, []) \langle \text{proof} \rangle$
lemma *exec-all-deterministic*:
 $\llbracket P \vdash \sigma \text{-jvm} \rightarrow (x, h, []); P \vdash \sigma \text{-jvm} \rightarrow \sigma' \rrbracket \Longrightarrow P \vdash \sigma' \text{-jvm} \rightarrow (x, h, []) \langle \text{proof} \rangle$

The start configuration of the JVM: in the start heap, we call a method m of class C in program P . The *this* pointer of the frame is set to *Null* to simulate a static method invocation.

definition *start-state* :: *jvm-prog* \Rightarrow *cname* \Rightarrow *mname* \Rightarrow *jvm-state* **where**
start-state $P C M =$
 $(\text{let } (D, Ts, T, mxs, mxl_0, b) = \text{method } P C M \text{ in}$
 $(\text{None}, \text{start-heap } P, [([], \text{Null} \# \text{replicate } mxl_0 \text{ undefined}, C, M, 0)]))$

end

3.6 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 $v \longleftrightarrow v = \text{Null} \vee \text{is-Addr } v$

primrec *check-instr* :: [*instr*, *jvm-prog*, *heap*, *val list*, *val list*,
cname, *mname*, *pc*, *frame list*] \Rightarrow *bool* **where**
check-instr-Load:
check-instr (*Load* n) $P h stk loc C M_0 pc frs =$

- $(n < \text{length } \text{loc})$
- | *check-instr-Store*:
 $\text{check-instr } (\text{Store } n) P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} =$
 $(0 < \text{length } \text{stk} \wedge n < \text{length } \text{loc})$
- | *check-instr-Push*:
 $\text{check-instr } (\text{Push } v) P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} =$
 $(\neg \text{is-Addr } v)$
- | *check-instr-New*:
 $\text{check-instr } (\text{New } C) P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} =$
 $\text{is-class } P C$
- | *check-instr-Getfield*:
 $\text{check-instr } (\text{Getfield } F C) P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} =$
 $(0 < \text{length } \text{stk} \wedge (\exists C' T. P \vdash C \text{ sees } F:T \text{ in } C') \wedge$
 $(\text{let } (C', T) = \text{field } P C F; \text{ref} = \text{hd } \text{stk} \text{ in}$
 $C' = C \wedge \text{is-Ref } \text{ref} \wedge (\text{ref} \neq \text{Null} \longrightarrow$
 $h (\text{the-Addr } \text{ref}) \neq \text{None} \wedge$
 $(\text{let } (D, vs) = \text{the } (h (\text{the-Addr } \text{ref}))) \text{ in}$
 $P \vdash D \preceq^* C \wedge vs (F, C) \neq \text{None} \wedge P, h \vdash \text{the } (vs (F, C)) : \preceq T))))$
- | *check-instr-Putfield*:
 $\text{check-instr } (\text{Putfield } F C) P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} =$
 $(1 < \text{length } \text{stk} \wedge (\exists C' T. P \vdash C \text{ sees } F:T \text{ in } C') \wedge$
 $(\text{let } (C', T) = \text{field } P C F; v = \text{hd } \text{stk}; \text{ref} = \text{hd } (\text{tl } \text{stk}) \text{ in}$
 $C' = C \wedge \text{is-Ref } \text{ref} \wedge (\text{ref} \neq \text{Null} \longrightarrow$
 $h (\text{the-Addr } \text{ref}) \neq \text{None} \wedge$
 $(\text{let } D = \text{fst } (\text{the } (h (\text{the-Addr } \text{ref})))) \text{ in}$
 $P \vdash D \preceq^* C \wedge P, h \vdash v : \preceq T))))$
- | *check-instr-Checkcast*:
 $\text{check-instr } (\text{Checkcast } C) P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} =$
 $(0 < \text{length } \text{stk} \wedge \text{is-class } P C \wedge \text{is-Ref } (\text{hd } \text{stk}))$
- | *check-instr-Invoke*:
 $\text{check-instr } (\text{Invoke } M n) P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} =$
 $(n < \text{length } \text{stk} \wedge \text{is-Ref } (\text{stk}!n) \wedge$
 $(\text{stk}!n \neq \text{Null} \longrightarrow$
 $(\text{let } a = \text{the-Addr } (\text{stk}!n);$
 $C = \text{cname-of } h a;$
 $Ts = \text{fst } (\text{snd } (\text{method } P C M))$
 $\text{in } h a \neq \text{None} \wedge P \vdash C \text{ has } M \wedge$
 $P, h \vdash \text{rev } (\text{take } n \text{ stk}) [:\preceq Ts]))$
- | *check-instr-Return*:
 $\text{check-instr } \text{Return } P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} =$
 $(0 < \text{length } \text{stk} \wedge ((0 < \text{length } \text{frs}) \longrightarrow$
 $(P \vdash C_0 \text{ has } M_0) \wedge$
 $(\text{let } v = \text{hd } \text{stk};$
 $T = \text{fst } (\text{snd } (\text{snd } (\text{method } P C_0 M_0)))$
 $\text{in } P, h \vdash v : \preceq T))))$

- | *check-instr-Pop*:
 $check_instr\ Pop\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs =$
 $(0 < length\ stk)$
- | *check-instr-IAdd*:
 $check_instr\ IAdd\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs =$
 $(1 < length\ stk \wedge is_Intg\ (hd\ stk) \wedge is_Intg\ (hd\ (tl\ stk)))$
- | *check-instr-IfFalse*:
 $check_instr\ (IfFalse\ b)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs =$
 $(0 < length\ stk \wedge is_Bool\ (hd\ stk) \wedge 0 \leq int\ pc+b)$
- | *check-instr-CmpEq*:
 $check_instr\ CmpEq\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs =$
 $(1 < length\ stk)$
- | *check-instr-Goto*:
 $check_instr\ (Goto\ b)\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs =$
 $(0 \leq int\ pc+b)$
- | *check-instr-Throw*:
 $check_instr\ Throw\ P\ h\ stk\ loc\ C_0\ M_0\ pc\ frs =$
 $(0 < length\ stk \wedge is_Ref\ (hd\ stk))$

definition *check* :: *jvm-prog* \Rightarrow *jvm-state* \Rightarrow *bool* **where**

$check\ P\ \sigma = (let\ (xcpt,\ h,\ frs) = \sigma\ in$
 $(case\ frs\ of\ [] \Rightarrow True\ | (stk,loc,C,M,pc)\#frs' \Rightarrow$
 $P \vdash C\ has\ M \wedge$
 $(let\ (C',Ts,T,mxs,mxl_0,ins,xt) = method\ P\ C\ M; i = ins!pc\ in$
 $pc < size\ ins \wedge size\ stk \leq mxs \wedge$
 $check_instr\ i\ P\ h\ stk\ loc\ C\ M\ pc\ frs'))$

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 $exec-1-d' :: jvm-prog \Rightarrow jvm-state\ type-error \Rightarrow jvm-state\ type-error \Rightarrow bool$
 $(- \vdash - \text{-jvmd}\rightarrow_1 - [61,61,61]60)$

for $P :: jvm-prog$

where

$P \vdash \sigma \text{-jvmd}\rightarrow_1 \sigma' \equiv (\sigma, \sigma') \in exec-1-d\ P$

| *exec-1-d-ErrorI*: $exec-d\ P\ \sigma = TypeError \implies P \vdash Normal\ \sigma \text{-jvmd}\rightarrow_1\ TypeError$

| *exec-1-d-NormalI*: $exec-d\ P\ \sigma = Normal\ (Some\ \sigma') \implies P \vdash Normal\ \sigma \text{-jvmd}\rightarrow_1\ Normal\ \sigma'$

— reflexive transitive closure:

definition *exec-all-d* :: *jvm-prog* \Rightarrow *jvm-state type-error* \Rightarrow *jvm-state type-error* \Rightarrow *bool*

$(- \vdash - \text{-jvmd}\rightarrow - [61,61,61]60)$ **where**

$exec_all_d_def1: P \vdash \sigma \text{-jvmd}\rightarrow \sigma' \iff (\sigma, \sigma') \in (exec-1-d\ P)^*$

notation (*ASCII*)

$exec_all_d\ (- \vdash - \text{-jvmd}\rightarrow - [61,61,61]60)$

lemma *exec-1-d-eq*:

$$\begin{aligned} \text{exec-1-d } P = & \{(s,t). \exists \sigma. s = \text{Normal } \sigma \wedge t = \text{TypeError} \wedge \text{exec-d } P \sigma = \text{TypeError}\} \cup \\ & \{(s,t). \exists \sigma \sigma'. s = \text{Normal } \sigma \wedge t = \text{Normal } \sigma' \wedge \text{exec-d } P \sigma = \text{Normal } (\text{Some } \sigma')\} \end{aligned}$$

<proof>

declare *split-paired-All* [*simp del*]

declare *split-paired-Ex* [*simp del*]

lemma *if-neq* [*dest!*]:

$$(if\ P\ then\ A\ else\ B) \neq B \implies P$$

<proof>

lemma *exec-d-no-errorI* [*intro*]:

$$check\ P\ \sigma \implies \text{exec-d } P\ \sigma \neq \text{TypeError}$$

<proof>

theorem *no-type-error-commutes*:

$$\text{exec-d } P\ \sigma \neq \text{TypeError} \implies \text{exec-d } P\ \sigma = \text{Normal } (\text{exec } (P, \sigma))$$

<proof>

lemma *defensive-imp-aggressive*:

$$P \vdash (\text{Normal } \sigma) -jvmd \rightarrow (\text{Normal } \sigma') \implies P \vdash \sigma -jvm \rightarrow \sigma' \text{<proof>}$$

end

3.7 Example for generating executable code from JVM semantics

theory *JVMListExample*

imports

../Common/SystemClasses

JVMExec

HOL-Library.Code-Target-Numeral

begin

definition *list-name* :: *string*

where

$$\text{list-name} == \text{"list"}$$

definition *test-name* :: *string*

where

$$\text{test-name} == \text{"test"}$$

definition *val-name* :: *string*

where

$$\text{val-name} == \text{"val"}$$

definition *next-name* :: *string*

where

$$\text{next-name} == \text{"next"}$$

definition *append-name* :: *string*

where

append-name == "append"

definition *makelist-name* :: *string*

where

makelist-name == "makelist"

definition *append-ins* :: *bytecode*

where

append-ins ==
 [Load 0,
 Getfield *next-name* *list-name*,
 Load 0,
 Getfield *next-name* *list-name*,
 Push Null,
 CmpEq,
 IfFalse 7,
 Pop,
 Load 0,
 Load 1,
 Putfield *next-name* *list-name*,
 Push Unit,
 Return,
 Load 1,
 Invoke *append-name* 1,
 Return]

definition *list-class* :: *jvm-method class*

where

list-class ==
 (Object,
 [(*val-name*, Integer), (*next-name*, Class *list-name*)],
 [(*append-name*, [Class *list-name*], Void,
 (3, 0, *append-ins*, [(1, 2, NullPointerException, 7, 0)])])])

definition *make-list-ins* :: *bytecode*

where

make-list-ins ==
 [New *list-name*,
 Store 0,
 Load 0,
 Push (Intg 1),
 Putfield *val-name* *list-name*,
 New *list-name*,
 Store 1,
 Load 1,
 Push (Intg 2),
 Putfield *val-name* *list-name*,
 New *list-name*,
 Store 2,
 Load 2,
 Push (Intg 3),
 Putfield *val-name* *list-name*,

```

Load 0,
Load 1,
Invoke append-name 1,
Pop,
Load 0,
Load 2,
Invoke append-name 1,
Return]

```

definition *test-class* :: *jvm-method class*

where

```

test-class ==
  (Object, [],
   [(makelist-name, [], Void, (3, 2, make-list-ins, []))])

```

definition *E* :: *jvm-prog*

where

```

E == SystemClasses @ [(list-name, list-class), (test-name, test-class)]

```

definition *undefined-cname* :: *cname*

where [code del]: *undefined-cname* = *undefined*

declare *undefined-cname-def*[*symmetric*, *code-unfold*]

code-printing constant *undefined-cname* \rightarrow (*SML*) *object*

definition *undefined-val* :: *val*

where [code del]: *undefined-val* = *undefined*

declare *undefined-val-def*[*symmetric*, *code-unfold*]

code-printing constant *undefined-val* \rightarrow (*SML*) *Unit*

lemmas [*code-unfold*] = *SystemClasses-def* [*unfolded ObjectC-def NullPointerC-def ClassCastC-def OutOfMemoryC-def*]

definition *test* = *exec* (*E*, *start-state E test-name makelist-name*)

$\langle ML \rangle$

end

Chapter 4

Bytecode Verifier

4.1 Semilattices

```
theory Semilat
imports Main HOL-Library.While-Combinator
begin

type-synonym 'a ord    = 'a ⇒ 'a ⇒ bool
type-synonym 'a binop  = 'a ⇒ 'a ⇒ 'a
type-synonym 'a sl     = 'a set × 'a ord × 'a binop

definition lesub :: 'a ⇒ 'a ord ⇒ 'a ⇒ bool
  where lesub x r y ⟷ r x y

definition lessub :: 'a ⇒ 'a ord ⇒ 'a ⇒ bool
  where lessub x r y ⟷ lesub x r y ∧ x ≠ y

definition plussub :: 'a ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'b ⇒ 'c
  where plussub x f y = f x y

notation (ASCII)
  lesub ((- /<='-- -) [50, 1000, 51] 50) and
  lessub ((- /<'-- -) [50, 1000, 51] 50) and
  plussub ((- /+'-- -) [65, 1000, 66] 65)

notation
  lesub ((- /⊑- -) [50, 0, 51] 50) and
  lessub ((- /⊑- -) [50, 0, 51] 50) and
  plussub ((- /⊔- -) [65, 0, 66] 65)

abbreviation (input)
  lesub1 :: 'a ⇒ 'a ord ⇒ 'a ⇒ bool ((- /⊑- -) [50, 1000, 51] 50)
  where x ⊑r y == x ⊑r y

abbreviation (input)
  lessub1 :: 'a ⇒ 'a ord ⇒ 'a ⇒ bool ((- /⊑- -) [50, 1000, 51] 50)
  where x ⊒r y == x ⊒r y

abbreviation (input)
```

plussub1 :: 'a ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'b ⇒ 'c ((- /⊔ -) [65, 1000, 66] 65)
where $x \sqcup_f y == x \sqcup_f y$

definition *ord* :: ('a × 'a) set ⇒ 'a ord

where

$ord\ r = (\lambda x\ y. (x,y) \in r)$

definition *order* :: 'a ord ⇒ 'a set ⇒ bool

where

$order\ r\ A \longleftrightarrow (\forall x \in A. x \sqsubseteq_r x) \wedge (\forall x \in A. \forall y \in A. x \sqsubseteq_r y \wedge y \sqsubseteq_r x \longrightarrow x=y) \wedge (\forall x \in A. \forall y \in A. \forall z \in A. x \sqsubseteq_r y \wedge y \sqsubseteq_r z \longrightarrow x \sqsubseteq_r z)$

definition *top* :: 'a ord ⇒ 'a ⇒ bool

where

$top\ r\ T \longleftrightarrow (\forall x. x \sqsubseteq_r T)$

definition *acc* :: 'a ord ⇒ bool

where

$acc\ r \longleftrightarrow wf\ \{(y,x). x \sqsubseteq_r y\}$

definition *closed* :: 'a set ⇒ 'a binop ⇒ bool

where

$closed\ A\ f \longleftrightarrow (\forall x \in A. \forall y \in A. x \sqcup_f y \in A)$

definition *semilat* :: 'a sl ⇒ bool

where

$semilat = (\lambda(A,r,f). order\ r\ A \wedge closed\ A\ f \wedge$
 $(\forall x \in A. \forall y \in A. x \sqsubseteq_r x \sqcup_f y) \wedge$
 $(\forall x \in A. \forall y \in A. y \sqsubseteq_r x \sqcup_f y) \wedge$
 $(\forall x \in A. \forall y \in A. \forall z \in A. x \sqsubseteq_r z \wedge y \sqsubseteq_r z \longrightarrow x \sqcup_f y \sqsubseteq_r z))$

definition *is-ub* :: ('a × 'a) set ⇒ 'a ⇒ 'a ⇒ 'a ⇒ bool

where

$is-ub\ r\ x\ y\ u \longleftrightarrow (x,u) \in r \wedge (y,u) \in r$

definition *is-lub* :: ('a × 'a) set ⇒ 'a ⇒ 'a ⇒ 'a ⇒ bool

where

$is-lub\ r\ x\ y\ u \longleftrightarrow is-ub\ r\ x\ y\ u \wedge (\forall z. is-ub\ r\ x\ y\ z \longrightarrow (u,z) \in r)$

definition *some-lub* :: ('a × 'a) set ⇒ 'a ⇒ 'a ⇒ 'a

where

$some-lub\ r\ x\ y = (SOME\ z. is-lub\ r\ x\ y\ z)$

locale *Semilat* =

fixes $A :: 'a\ set$

fixes $r :: 'a\ ord$

fixes $f :: 'a\ binop$

assumes *semilat*: $semilat\ (A, r, f)$

lemma *order-refl* [*simp, intro*]: $order\ r\ A \Longrightarrow x \in A \Longrightarrow x \sqsubseteq_r x$

(proof)

lemma *order-antisym*: $\llbracket order\ r\ A; x \sqsubseteq_r y; y \sqsubseteq_r x; x \in A; y \in A \rrbracket \Longrightarrow x = y$

(proof)

lemma *order-trans*: $\llbracket order\ r\ A; x \sqsubseteq_r y; y \sqsubseteq_r z; x \in A; y \in A; z \in A \rrbracket \Longrightarrow x \sqsubseteq_r z$

<proof>

lemma *order-less-irrefl* [*intro, simp*]: $\text{order } r \ A \implies x \in A \implies \neg x \sqsubseteq_r x$

<proof>

lemma *order-less-trans*: $\llbracket \text{order } r \ A; x \sqsubseteq_r y; y \sqsubseteq_r z; x \in A; y \in A; z \in A \rrbracket \implies x \sqsubseteq_r z$

<proof>

lemma *topD* [*simp, intro*]: $\text{top } r \ T \implies x \sqsubseteq_r T$

<proof>

lemma *top-le-conv* [*simp*]: $\llbracket \text{order } r \ A; \text{top } r \ T; x \in A; T \in A \rrbracket \implies (T \sqsubseteq_r x) = (x = T)$

<proof>

lemma *semilat-Def*:

$\text{semilat}(A, r, f) \iff \text{order } r \ A \wedge \text{closed } A \ f \wedge$
 $(\forall x \in A. \forall y \in A. x \sqsubseteq_r x \sqcup_f y) \wedge$
 $(\forall x \in A. \forall y \in A. y \sqsubseteq_r x \sqcup_f y) \wedge$
 $(\forall x \in A. \forall y \in A. \forall z \in A. x \sqsubseteq_r z \wedge y \sqsubseteq_r z \implies x \sqcup_f y \sqsubseteq_r z)$

<proof>

lemma (*in Semilat*) *orderI* [*simp, intro*]: $\text{order } r \ A$

<proof>

lemma (*in Semilat*) *closedI* [*simp, intro*]: $\text{closed } A \ f$

<proof>

lemma *closedD*: $\llbracket \text{closed } A \ f; x \in A; y \in A \rrbracket \implies x \sqcup_f y \in A$

<proof>

lemma *closed-UNIV* [*simp*]: $\text{closed } UNIV \ f$

<proof>

lemma (*in Semilat*) *closed-f* [*simp, intro*]: $\llbracket x \in A; y \in A \rrbracket \implies x \sqcup_f y \in A$

<proof>

lemma (*in Semilat*) *refl-r* [*intro, simp*]: $x \in A \implies x \sqsubseteq_r x$ *<proof>*

lemma (*in Semilat*) *antisym-r* [*intro?*]: $\llbracket x \sqsubseteq_r y; y \sqsubseteq_r x; x \in A; y \in A \rrbracket \implies x = y$

<proof>

lemma (*in Semilat*) *trans-r* [*trans, intro?*]: $\llbracket x \sqsubseteq_r y; y \sqsubseteq_r z; x \in A; y \in A; z \in A \rrbracket \implies x \sqsubseteq_r z$

<proof>

lemma (*in Semilat*) *ub1* [*simp, intro?*]: $\llbracket x \in A; y \in A \rrbracket \implies x \sqsubseteq_r x \sqcup_f y$

<proof>

lemma (*in Semilat*) *ub2* [*simp, intro?*]: $\llbracket x \in A; y \in A \rrbracket \implies y \sqsubseteq_r x \sqcup_f y$

<proof>

lemma (*in Semilat*) *lub* [*simp, intro?*]:

$\llbracket x \sqsubseteq_r z; y \sqsubseteq_r z; x \in A; y \in A; z \in A \rrbracket \implies x \sqcup_f y \sqsubseteq_r z$

<proof>

lemma (*in Semilat*) *plus-le-conv* [*simp*]:

$\llbracket x \in A; y \in A; z \in A \rrbracket \implies (x \sqcup_f y \sqsubseteq_r z) = (x \sqsubseteq_r z \wedge y \sqsubseteq_r z)$

<proof>

lemma (*in Semilat*) *le-iff-plus-unchanged*:

assumes $x \in A$ **and** $y \in A$

shows $x \sqsubseteq_r y \iff x \sqcup_f y = y$ (**is** $?P \iff ?Q$) *<proof>*

lemma (*in Semilat*) *le-iff-plus-unchanged2*:

assumes $x \in A$ **and** $y \in A$

shows $x \sqsubseteq_r y \iff y \sqcup_f x = y$ (**is** $?P \iff ?Q$) *<proof>*

lemma (*in Semilat*) *plus-assoc* [*simp*]:

assumes $a: a \in A$ **and** $b: b \in A$ **and** $c: c \in A$

shows $a \sqcup_f (b \sqcup_f c) = a \sqcup_f b \sqcup_f c$ *<proof>*

lemma (*in Semilat*) *plus-com-lemma*:

$\llbracket a \in A; b \in A \rrbracket \implies a \sqcup_f b \sqsubseteq_r b \sqcup_f a$ *<proof>*

lemma (*in Semilat*) *plus-commutative*:

$\llbracket a \in A; b \in A \rrbracket \implies a \sqcup_f b = b \sqcup_f a$

(proof)

lemma *is-lubD*:

$is-lub\ r\ x\ y\ u \implies is-ub\ r\ x\ y\ u \wedge (\forall z. is-ub\ r\ x\ y\ z \longrightarrow (u,z) \in r)$

(proof)

lemma *is-ubI*:

$\llbracket (x,u) \in r; (y,u) \in r \rrbracket \implies is-ub\ r\ x\ y\ u$

(proof)

lemma *is-ubD*:

$is-ub\ r\ x\ y\ u \implies (x,u) \in r \wedge (y,u) \in r$

(proof)

lemma *is-lub-bigger1* [*iff*]:

$is-lub\ (r\hat{*})\ x\ y\ y = ((x,y) \in r\hat{*})$ *(proof)*

lemma *is-lub-bigger2* [*iff*]:

$is-lub\ (r\hat{*})\ x\ y\ x = ((y,x) \in r\hat{*})$ *(proof)*

lemma *extend-lub*:

$\llbracket single-valued\ r; is-lub\ (r\hat{*})\ x\ y\ u; (x',x) \in r \rrbracket$

$\implies \exists v. is-lub\ (r\hat{*})\ x'\ y\ v$ *(proof)*

lemma *single-valued-has-lubs* [*rule-format*]:

$\llbracket single-valued\ r; (x,u) \in r\hat{*} \rrbracket \implies (\forall y. (y,u) \in r\hat{*} \longrightarrow$

$(\exists z. is-lub\ (r\hat{*})\ x\ y\ z))$ *(proof)*

lemma *some-lub-conv*:

$\llbracket acyclic\ r; is-lub\ (r\hat{*})\ x\ y\ u \rrbracket \implies some-lub\ (r\hat{*})\ x\ y = u$ *(proof)*

lemma *is-lub-some-lub*:

$\llbracket single-valued\ r; acyclic\ r; (x,u) \in r\hat{*}; (y,u) \in r\hat{*} \rrbracket$

$\implies is-lub\ (r\hat{*})\ x\ y\ (some-lub\ (r\hat{*})\ x\ y)$

(proof)

4.1.1 An executable lub-finder

definition *exec-lub* :: ('a * 'a) set \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a *binop*

where

$exec-lub\ r\ f\ x\ y = while\ (\lambda z. (x,z) \notin r^*)\ f\ y$

lemma *exec-lub-refl*: $exec-lub\ r\ f\ T\ T = T$

(proof)

lemma *acyclic-single-valued-finite*:

$\llbracket acyclic\ r; single-valued\ r; (x,y) \in r^* \rrbracket$

$\implies finite\ (r \cap \{a. (x, a) \in r^*\} \times \{b. (b, y) \in r^*\})$ *(proof)*

lemma *exec-lub-conv*:

$\llbracket acyclic\ r; \forall x\ y. (x,y) \in r \longrightarrow f\ x = y; is-lub\ (r^*)\ x\ y\ u \rrbracket \implies$

$exec-lub\ r\ f\ x\ y = u$ *(proof)*

lemma *is-lub-exec-lub*:

$\llbracket single-valued\ r; acyclic\ r; (x,u):r\hat{*}; (y,u):r\hat{*}; \forall x\ y. (x,y) \in r \longrightarrow f\ x = y \rrbracket$

$\implies is-lub\ (r\hat{*})\ x\ y\ (exec-lub\ r\ f\ x\ y)$

(proof)

end

4.2 The Error Type

theory *Err*

imports *Semilat*

begin

datatype 'a err = Err | OK 'a

type-synonym 'a ebinop = 'a \Rightarrow 'a \Rightarrow 'a err

type-synonym 'a esl = 'a set \times 'a ord \times 'a ebinop

primrec ok-val :: 'a err \Rightarrow 'a

where

ok-val (OK x) = x

definition lift :: ('a \Rightarrow 'b err) \Rightarrow ('a err \Rightarrow 'b err)

where

lift f e = (case e of Err \Rightarrow Err | OK x \Rightarrow f x)

definition lift2 :: ('a \Rightarrow 'b \Rightarrow 'c err) \Rightarrow 'a err \Rightarrow 'b err \Rightarrow 'c err

where

lift2 f e₁ e₂ =
(case e₁ of Err \Rightarrow Err | OK x \Rightarrow (case e₂ of Err \Rightarrow Err | OK y \Rightarrow f x y))

definition le :: 'a ord \Rightarrow 'a err ord

where

le r e₁ e₂ =
(case e₂ of Err \Rightarrow True | OK y \Rightarrow (case e₁ of Err \Rightarrow False | OK x \Rightarrow x \sqsubseteq_r y))

definition sup :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a err \Rightarrow 'b err \Rightarrow 'c err)

where

sup f = lift2 ($\lambda x y. OK (x \sqcup_f y)$)

definition err :: 'a set \Rightarrow 'a err set

where

err A = insert Err {OK x | x. x \in A}

definition esl :: 'a sl \Rightarrow 'a esl

where

esl = ($\lambda(A,r,f). (A, r, \lambda x y. OK(f x y))$)

definition sl :: 'a esl \Rightarrow 'a err sl

where

sl = ($\lambda(A,r,f). (err A, le r, lift2 f)$)

abbreviation

err-semilat :: 'a esl \Rightarrow bool **where**
err-semilat L == semilat(sl L)

primrec strict :: ('a \Rightarrow 'b err) \Rightarrow ('a err \Rightarrow 'b err)

where

strict f Err = Err
| strict f (OK x) = f x

lemma err-def':

err A = insert Err {x. $\exists y \in A. x = OK y$ } <proof>

lemma strict-Some [simp]:

(strict f x = OK y) = ($\exists z. x = OK z \wedge f z = OK y$) <proof>

lemma *not-Err-eq*: $(x \neq \text{Err}) = (\exists a. x = \text{OK } a)$ *<proof>*

lemma *not-OK-eq*: $(\forall y. x \neq \text{OK } y) = (x = \text{Err})$ *<proof>*

lemma *unfold-lesub-err*: $e1 \sqsubseteq_{le\ r} e2 = le\ r\ e1\ e2$ *<proof>*

lemma *le-err-refl*: $\forall x. x \sqsubseteq_r x \implies e \sqsubseteq_{le\ r} e$ *<proof>*

lemma *le-err-refl'*: $(\forall x \in A. x \sqsubseteq_r x) \implies e \in \text{err } A \implies e \sqsubseteq_{le\ r} e$ *<proof><proof><proof><proof><proof><proof><proof>*

4.3 More about Options

theory *Opt* **imports** *Err* **begin**

definition *le* :: $'a\ \text{ord} \Rightarrow 'a\ \text{option}\ \text{ord}$

where

$le\ r\ o_1\ o_2 =$

$(\text{case } o_2\ \text{of } \text{None} \Rightarrow o_1 = \text{None} \mid \text{Some } y \Rightarrow (\text{case } o_1\ \text{of } \text{None} \Rightarrow \text{True} \mid \text{Some } x \Rightarrow x \sqsubseteq_r y))$

definition *opt* :: $'a\ \text{set} \Rightarrow 'a\ \text{option}\ \text{set}$

where

$opt\ A = \text{insert } \text{None} \ \{\text{Some } y \mid y. y \in A\}$

definition *sup* :: $'a\ \text{ebinop} \Rightarrow 'a\ \text{option}\ \text{ebinop}$

where

$sup\ f\ o_1\ o_2 =$

$(\text{case } o_1\ \text{of } \text{None} \Rightarrow \text{OK } o_2$

$\mid \text{Some } x \Rightarrow (\text{case } o_2\ \text{of } \text{None} \Rightarrow \text{OK } o_1$

$\mid \text{Some } y \Rightarrow (\text{case } f\ x\ y\ \text{of } \text{Err} \Rightarrow \text{Err} \mid \text{OK } z \Rightarrow \text{OK } (\text{Some } z))))$

definition *esl* :: $'a\ \text{esl} \Rightarrow 'a\ \text{option}\ \text{esl}$

where

$esl = (\lambda(A,r,f). (opt\ A, le\ r, sup\ f))$

lemma *unfold-le-opt*:

$o_1 \sqsubseteq_{le\ r} o_2 =$

$(\text{case } o_2\ \text{of } \text{None} \Rightarrow o_1 = \text{None} \mid$

$\text{Some } y \Rightarrow (\text{case } o_1\ \text{of } \text{None} \Rightarrow \text{True} \mid \text{Some } x \Rightarrow x \sqsubseteq_r y))$ *<proof>*

lemma *le-opt-refl*: $\text{order } r\ A \implies x \in opt\ A \implies x \sqsubseteq_{le\ r} x$ *<proof><proof><proof><proof><proof><proof><proof>*

4.4 Products as Semilattices

theory *Product*

imports *Err*

begin

definition *le* :: $'a\ \text{ord} \Rightarrow 'b\ \text{ord} \Rightarrow ('a \times 'b)\ \text{ord}$

where

$le\ r_A\ r_B = (\lambda(a_1, b_1)\ (a_2, b_2). a_1 \sqsubseteq_{r_A} a_2 \wedge b_1 \sqsubseteq_{r_B} b_2)$

definition *sup* :: $'a\ \text{ebinop} \Rightarrow 'b\ \text{ebinop} \Rightarrow ('a \times 'b)\ \text{ebinop}$

where

$sup\ f\ g = (\lambda(a_1, b_1)\ (a_2, b_2). \text{Err}.sup\ \text{Pair}\ (a_1 \sqcup_f a_2)\ (b_1 \sqcup_g b_2))$

definition *esl* :: $'a\ \text{esl} \Rightarrow 'b\ \text{esl} \Rightarrow ('a \times 'b)\ \text{esl}$

where

$esl = (\lambda(A,r_A,f_A) (B,r_B,f_B). (A \times B, le\ r_A\ r_B, sup\ f_A\ f_B))$

abbreviation

$lesubprod :: 'a \times 'b \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \times 'b \Rightarrow bool$
 $((- / \sqsubseteq'(-, -) [50, 0, 0, 51] 50) \mathbf{where}$
 $p \sqsubseteq(r_A, r_B) q == p \sqsubseteq_{Product.le\ r_A\ r_B} q$

lemma *unfold-lesub-prod*: $x \sqsubseteq(r_A, r_B) y = le\ r_A\ r_B\ x\ y \langle proof \rangle$

lemma *le-prod-Pair-conv* [iff]: $((a_1, b_1) \sqsubseteq(r_A, r_B) (a_2, b_2)) = (a_1 \sqsubseteq_{r_A} a_2 \ \&\ b_1 \sqsubseteq_{r_B} b_2) \langle proof \rangle$

lemma *less-prod-Pair-conv*:

$((a_1, b_1) \sqsubseteq_{Product.le\ r_A\ r_B} (a_2, b_2)) =$
 $(a_1 \sqsubseteq_{r_A} a_2 \ \&\ b_1 \sqsubseteq_{r_B} b_2 \mid a_1 \sqsubseteq_{r_A} a_2 \ \&\ b_1 \sqsubseteq_{r_B} b_2) \langle proof \rangle$

lemma *order-le-prodI* [iff]: $(order\ r_A\ A \ \&\ order\ r_B\ B) \Longrightarrow order\ (Product.le\ r_A\ r_B)\ (A \times B)$
 $\langle proof \rangle$

lemma *order-le-prodE*: $A \neq \{\} \Longrightarrow B \neq \{\} \Longrightarrow order\ (Product.le\ r_A\ r_B)\ (A \times B) \Longrightarrow (order\ r_A\ A$
 $\ \&\ order\ r_B\ B)$
 $\langle proof \rangle$

lemma *order-le-prod* [iff]: $A \neq \{\} \Longrightarrow B \neq \{\} \Longrightarrow order\ (Product.le\ r_A\ r_B)\ (A \times B) = (order\ r_A\ A$
 $\ \&\ order\ r_B\ B) \langle proof \rangle$

lemma *acc-le-prodI* [intro!]:

$\llbracket acc\ r_A; acc\ r_B \rrbracket \Longrightarrow acc\ (Product.le\ r_A\ r_B) \langle proof \rangle$

lemma *closed-lift2-sup*:

$\llbracket closed\ (err\ A)\ (lift2\ f); closed\ (err\ B)\ (lift2\ g) \rrbracket \Longrightarrow$
 $closed\ (err\ (A \times B))\ (lift2\ (sup\ f\ g)) \langle proof \rangle$

lemma *unfold-plussub-lift2*: $e_1 \sqcup_{lift2\ f} e_2 = lift2\ f\ e_1\ e_2 \langle proof \rangle$

lemma *plus-eq-Err-conv* [simp]:

assumes $x \in A\ y \in A\ semilat(err\ A, Err.le\ r, lift2\ f)$
shows $(x \sqcup_f y = Err) = (\neg(\exists z \in A. x \sqsubseteq_r z \ \&\ y \sqsubseteq_r z)) \langle proof \rangle$

lemma *err-semilat-Product-esl*:

$\bigwedge L_1\ L_2. \llbracket err-semilat\ L_1; err-semilat\ L_2 \rrbracket \Longrightarrow err-semilat\ (Product.esl\ L_1\ L_2) \langle proof \rangle$
end

4.5 Fixed Length Lists

theory *Listn*

imports *Err HOL-Library.NList*

begin

definition *le* :: $'a\ ord \Rightarrow ('a\ list)\ ord$

where

$le\ r = list-all2\ (\lambda x\ y. x \sqsubseteq_r y)$

abbreviation

$lesublist :: 'a\ list \Rightarrow 'a\ ord \Rightarrow 'a\ list \Rightarrow bool\ ((- / \sqsubseteq) [50, 0, 51] 50) \mathbf{where}$
 $x \sqsubseteq_r y == x <=-(Listn.le\ r)\ y$

abbreviation

$lessublist :: 'a\ list \Rightarrow 'a\ ord \Rightarrow 'a\ list \Rightarrow bool\ ((- / \sqsubseteq) [50, 0, 51] 50) \mathbf{where}$
 $x \sqsubseteq_r y == x <-(Listn.le\ r)\ y$

abbreviation

plussublist :: 'a list \Rightarrow ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b list \Rightarrow 'c list
 ((- / [□.] -) [65, 0, 66] 65) **where**
x [□.] *y* == *x* □_{map2} *f* *y*

primrec *coalesce* :: 'a err list \Rightarrow 'a list err

where

coalesce [] = OK []
 | *coalesce* (ex#exs) = Err.sup (#) ex (*coalesce* exs)

definition *sl* :: nat \Rightarrow 'a sl \Rightarrow 'a list sl

where

sl *n* = ($\lambda(A,r,f)$). (*nlists* *n* *A*, *le* *r*, *map2* *f*)

definition *sup* :: ('a \Rightarrow 'b \Rightarrow 'c err) \Rightarrow 'a list \Rightarrow 'b list \Rightarrow 'c list err

where

sup *f* = (λ xs *ys*. if size xs = size *ys* then *coalesce*(xs [□.] *ys*) else Err)

definition *upto-esl* :: nat \Rightarrow 'a esl \Rightarrow 'a list esl

where

upto-esl *m* = ($\lambda(A,r,f)$). (*Union*{*nlists* *n* *A* | *n*. *n* \leq *m*}, *le* *r*, *sup* *f*)

lemmas [*simp*] = *set-update-subsetI*

lemma *unfold-lesub-list*: xs [□.] *ys* = *Listn.le* *r* xs *ys*⟨*proof*⟩

lemma *Nil-le-conv* [*iff*]: (□ [□.] *ys*) = (*ys* = □)⟨*proof*⟩

lemma *Cons-notle-Nil* [*iff*]: \neg *x*#xs [□.] []⟨*proof*⟩

lemma *Cons-le-Cons* [*iff*]: *x*#xs [□.] *y*#*ys* = (*x* □_r *y* \wedge xs [□.] *ys*)⟨*proof*⟩

lemma *list-update-le-cong*:

[*i* < size xs; xs [□.] *ys*; *x* □_r *y*] \Longrightarrow xs[*i*:=*x*] [□.] *ys*[*i*:=*y*]⟨*proof*⟩

lemma *le-listD*: [xs [□.] *ys*; *p* < size xs] \Longrightarrow xs!*p* □_r *ys*!*p*⟨*proof*⟩

lemma *le-list-refl*: \forall *x*. *x* □_r *x* \Longrightarrow xs [□.] xs⟨*proof*⟩

lemma *le-list-trans*:

assumes *ord*: *order* *r* *A*

and *xs*: *xs* \in *nlists* *n* *A* **and** *ys*: *ys* \in *nlists* *n* *A* **and** *zs*: *zs* \in *nlists* *n* *A*

and *xs* [□.] *ys* **and** *ys* [□.] *zs*

shows *xs* [□.] *zs*⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

4.6 Typing and Dataflow Analysis Framework

theory *Typing-Framework-1* **imports** *Semilattices* **begin**

The relationship between dataflow analysis and a welltyped-instruction predicate.

type-synonym

's *step-type* = nat \Rightarrow 's \Rightarrow (nat \times 's) list

definition *stable* :: 's ord \Rightarrow 's *step-type* \Rightarrow 's list \Rightarrow nat \Rightarrow bool

where

stable *r* *step* τ *s* *p* \longleftrightarrow (\forall (*q*, τ) \in *set* (*step* *p* (τ !*s*)). τ □_r τ !*q*)

definition *stables* :: 's ord \Rightarrow 's step-type \Rightarrow 's list \Rightarrow bool

where

stables r step τ s $\longleftrightarrow (\forall p < \text{size } \tau. \text{stable } r \text{ step } \tau s p)$

definition *wt-step* :: 's ord \Rightarrow 's \Rightarrow 's step-type \Rightarrow 's list \Rightarrow bool

where

wt-step r T step τ s $\longleftrightarrow (\forall p < \text{size } \tau. \tau s!p \neq T \wedge \text{stable } r \text{ step } \tau s p)$

end

4.7 More on Semilattices

theory *SemilatAlg*

imports *Typing-Framework-1*

begin

definition *lesubstep-type* :: (nat \times 's) set \Rightarrow 's ord \Rightarrow (nat \times 's) set \Rightarrow bool

((- / { \sqsubseteq -.} -) [50, 0, 51] 50)

where A { \sqsubseteq_r } B $\equiv \forall (p, \tau) \in A. \exists \tau'. (p, \tau') \in B \wedge \tau \sqsubseteq_r \tau'$

notation (*ASCII*)

lesubstep-type ((- / { \leq' --} -) [50, 0, 51] 50)

primrec *pluslssub* :: 'a list \Rightarrow ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a ((- / \sqcup -) [65, 0, 66] 65)

where

pluslssub [] f y = y

| *pluslssub* (x#xs) f y = *pluslssub* xs f (x \sqcup_f y)

definition *bounded* :: 's step-type \Rightarrow nat \Rightarrow bool

where

bounded step n $\longleftrightarrow (\forall p < n. \forall \tau. \forall (q, \tau') \in \text{set } (\text{step } p \tau). q < n)$

definition *pres-type* :: 's step-type \Rightarrow nat \Rightarrow 's set \Rightarrow bool

where

pres-type step n A $\longleftrightarrow (\forall \tau \in A. \forall p < n. \forall (q, \tau') \in \text{set } (\text{step } p \tau). \tau' \in A)$

definition *mono* :: 's ord \Rightarrow 's step-type \Rightarrow nat \Rightarrow 's set \Rightarrow bool

where

mono r step n A \longleftrightarrow

$(\forall \tau p \tau'. \tau \in A \wedge p < n \wedge \tau \sqsubseteq_r \tau' \longrightarrow \text{set } (\text{step } p \tau) \{ \sqsubseteq_r \} \text{set } (\text{step } p \tau'))$

lemma [*iff*]: {} { \sqsubseteq_r } B

<proof>

lemma [*iff*]: (A { \sqsubseteq_r } {}) = (A = {})

<proof>

lemma *lesubstep-union*:

$\llbracket A_1 \{ \sqsubseteq_r \} B_1; A_2 \{ \sqsubseteq_r \} B_2 \rrbracket \Longrightarrow A_1 \cup A_2 \{ \sqsubseteq_r \} B_1 \cup B_2$

<proof>

lemma *pres-typeD*:

$\llbracket \text{pres-type } \text{step } n A; s \in A; p < n; (q, s') \in \text{set } (\text{step } p s) \rrbracket \Longrightarrow s' \in A$ *<proof>*

lemma *monoD*:

$\llbracket \text{mono } r \text{ step } n A; p < n; s \in A; s \sqsubseteq_r t \rrbracket \Longrightarrow \text{set } (\text{step } p s) \{ \sqsubseteq_r \} \text{set } (\text{step } p t)$ *<proof>*

lemma *boundedD*:

$\llbracket \text{bounded step } n; p < n; (q,t) \in \text{set } (\text{step } p \text{ } xs) \rrbracket \implies q < n \langle \text{proof} \rangle$

lemma *lesubstep-type-refl* [*simp, intro*]:

$(\bigwedge x. x \sqsubseteq_r x) \implies A \{\sqsubseteq_r\} A \langle \text{proof} \rangle$

lemma *lesub-step-typeD*:

$A \{\sqsubseteq_r\} B \implies (x,y) \in A \implies \exists y'. (x, y') \in B \wedge y \sqsubseteq_r y' \langle \text{proof} \rangle$

lemma *list-update-le-listI* [*rule-format*]:

$\text{set } xs \subseteq A \longrightarrow \text{set } ys \subseteq A \longrightarrow xs \llbracket \sqsubseteq_r \rrbracket ys \longrightarrow p < \text{size } xs \longrightarrow$

$x \sqsubseteq_r ys!p \longrightarrow \text{semilat}(A,r,f) \longrightarrow x \in A \longrightarrow$

$xs[p := x \sqcup_f xs!p] \llbracket \sqsubseteq_r \rrbracket ys \langle \text{proof} \rangle$

lemma *plusplus-closed*: **assumes** *Semilat A r f* **shows**

$\bigwedge y. \llbracket \text{set } x \subseteq A; y \in A \rrbracket \implies x \sqcup_f y \in A \langle \text{proof} \rangle$

lemma (**in** *Semilat*) *pp-ub2*:

$\bigwedge y. \llbracket \text{set } x \subseteq A; y \in A \rrbracket \implies y \sqsubseteq_r x \sqcup_f y \langle \text{proof} \rangle$

lemma (**in** *Semilat*) *pp-ub1*:

shows $\bigwedge y. \llbracket \text{set } ls \subseteq A; y \in A; x \in \text{set } ls \rrbracket \implies x \sqsubseteq_r ls \sqcup_f y \langle \text{proof} \rangle$

lemma (**in** *Semilat*) *pp-lub*:

assumes $z: z \in A$

shows

$\bigwedge y. y \in A \implies \text{set } xs \subseteq A \implies \forall x \in \text{set } xs. x \sqsubseteq_r z \implies y \sqsubseteq_r z \implies xs \sqcup_f y \sqsubseteq_r z \langle \text{proof} \rangle$

lemma *ub1'*: **assumes** *Semilat A r f*

shows $\llbracket \forall (p,s) \in \text{set } S. s \in A; y \in A; (a,b) \in \text{set } S \rrbracket$

$\implies b \sqsubseteq_r \text{map snd } [(p', t') \leftarrow S. p' = a] \sqcup_f y \langle \text{proof} \rangle$

lemma *plusplus-empty*:

$\forall s'. (q, s') \in \text{set } S \longrightarrow s' \sqcup_f ss ! q = ss ! q \implies$

$(\text{map snd } [(p', t') \leftarrow S. p' = q] \sqcup_f ss ! q) = ss ! q \langle \text{proof} \rangle$

end

4.8 Lifting the Typing Framework to *err*, *app*, and *eff*

theory *Typing-Framework-err* **imports** *SemilatAlg* **begin**

definition *wt-err-step* :: *'s ord* \Rightarrow *'s err step-type* \Rightarrow *'s err list* \Rightarrow *bool*

where

$\text{wt-err-step } r \text{ step } \tau s \longleftrightarrow \text{wt-step } (\text{Err.le } r) \text{ Err step } \tau s$

definition *wt-app-eff* :: *'s ord* \Rightarrow (*nat* \Rightarrow *'s* \Rightarrow *bool*) \Rightarrow *'s step-type* \Rightarrow *'s list* \Rightarrow *bool*

where

$\text{wt-app-eff } r \text{ app step } \tau s \longleftrightarrow$

$(\forall p < \text{size } \tau s. \text{app } p (\tau s!p) \wedge (\forall (q,\tau) \in \text{set } (\text{step } p (\tau s!p)). \tau \leq\text{-}r \tau s!q))$

definition *map-snd* :: (*'b* \Rightarrow *'c*) \Rightarrow (*'a* \times *'b*) *list* \Rightarrow (*'a* \times *'c*) *list*

where

$\text{map-snd } f = \text{map } (\lambda(x,y). (x, f y))$

definition *error* :: *nat* \Rightarrow (*nat* \times *'a err*) *list*

where

$error\ n = map\ (\lambda x. (x, Err))\ [0..<n]$

definition $err\ step :: nat \Rightarrow (nat \Rightarrow 's \Rightarrow bool) \Rightarrow 's\ step\ type \Rightarrow 's\ err\ step\ type$

where

$err\ step\ n\ app\ step\ p\ t =$
 $(case\ t\ of$
 $Err\ \Rightarrow\ error\ n$
 $| OK\ \tau \Rightarrow\ if\ app\ p\ \tau\ then\ map\ snd\ OK\ (step\ p\ \tau)\ else\ error\ n)$

definition $app\ mono :: 's\ ord \Rightarrow (nat \Rightarrow 's \Rightarrow bool) \Rightarrow nat \Rightarrow 's\ set \Rightarrow bool$

where

$app\ mono\ r\ app\ n\ A \longleftrightarrow$
 $(\forall s\ p\ t. s \in A \wedge p < n \wedge s \sqsubseteq_r t \longrightarrow app\ p\ t \longrightarrow app\ p\ s)$

lemmas $err\ step\ defs = err\ step\ def\ map\ snd\ def\ error\ def$

lemma $bounded\ err\ stepD:$

$\llbracket bounded\ (err\ step\ n\ app\ step)\ n;$
 $p < n; app\ p\ a; (q, b) \in set\ (step\ p\ a) \rrbracket \Longrightarrow q < n \langle proof \rangle$

lemma $in\ map\ sndD: (a, b) \in set\ (map\ snd\ f\ xs) \Longrightarrow \exists b'. (a, b') \in set\ xs \langle proof \rangle$

lemma $bounded\ err\ stepI:$

$\forall p. p < n \longrightarrow (\forall s. app\ p\ s \longrightarrow (\forall (q, s') \in set\ (step\ p\ s). q < n))$
 $\Longrightarrow bounded\ (err\ step\ n\ app\ step)\ n \langle proof \rangle$

lemma $bounded\ lift:$

$bounded\ step\ n \Longrightarrow bounded\ (err\ step\ n\ app\ step)\ n \langle proof \rangle$

lemma $le\ list\ map\ OK\ [simp]:$

$\bigwedge b. (map\ OK\ a\ [\sqsubseteq_{Err.le\ r}]\ map\ OK\ b) = (a\ [\sqsubseteq_r]\ b) \langle proof \rangle$

lemma $map\ snd\ lessI:$

$set\ xs\ \{\sqsubseteq_r\}\ set\ ys \Longrightarrow set\ (map\ snd\ OK\ xs)\ \{\sqsubseteq_{Err.le\ r}\}\ set\ (map\ snd\ OK\ ys) \langle proof \rangle$

lemma $mono\ lift:$

$\llbracket order\ r\ A; app\ mono\ r\ app\ n\ A; bounded\ (err\ step\ n\ app\ step)\ n;$
 $\forall s\ p\ t. s \in A \wedge p < n \wedge s \sqsubseteq_r t \longrightarrow app\ p\ t \longrightarrow set\ (step\ p\ s)\ \{\sqsubseteq_r\}\ set\ (step\ p\ t) \rrbracket$
 $\Longrightarrow mono\ (Err.le\ r)\ (err\ step\ n\ app\ step)\ n\ (err\ A) \langle proof \rangle$

lemma $in\ errorD: (x, y) \in set\ (error\ n) \Longrightarrow y = Err \langle proof \rangle$

lemma $pres\ type\ lift:$

$\forall s \in A. \forall p. p < n \longrightarrow app\ p\ s \longrightarrow (\forall (q, s') \in set\ (step\ p\ s). s' \in A)$
 $\Longrightarrow pres\ type\ (err\ step\ n\ app\ step)\ n\ (err\ A) \langle proof \rangle$

lemma $wt\ err\ imp\ wt\ app\ eff:$

assumes $wt: wt\ err\ step\ r\ (err\ step\ (size\ ts)\ app\ step)\ ts$
assumes $b: bounded\ (err\ step\ (size\ ts)\ app\ step)\ (size\ ts)$
shows $wt\ app\ eff\ r\ app\ step\ (map\ ok\ val\ ts) \langle proof \rangle$

lemma $wt\ app\ eff\ imp\ wt\ err:$

assumes $app\ eff: wt\ app\ eff\ r\ app\ step\ ts$
assumes $bounded: bounded\ (err\ step\ (size\ ts)\ app\ step)\ (size\ ts)$

shows *wt-err-step* *r* (*err-step* (*size ts*) *app step*) (*map OK ts*)*<proof>*
end

4.9 Kildall's Algorithm

theory *Kildall-1*

imports *SemilatAlg*

begin

primrec *merges* :: '*s* *binop* \Rightarrow (*nat* \times '*s*) *list* \Rightarrow '*s* *list* \Rightarrow '*s* *list*

where

merges *f* [] $\tau s = \tau s$
| *merges* *f* (*p*'#*ps*) $\tau s = (\text{let } (p, \tau) = p' \text{ in } \text{merges } f \text{ ps } (\tau s[p := \tau \sqcup_f \tau s!p]))$

lemmas [*simp*] = *Let-def Semilat.le-iff-plus-unchanged* [*OF Semilat.intro, symmetric*]

lemma (**in** *Semilat*) *nth-merges*:

$\bigwedge ss. \llbracket p < \text{length } ss; ss \in \text{nlists } n \ A; \forall (p, t) \in \text{set } ps. p < n \wedge t \in A \rrbracket \Longrightarrow$
 $(\text{merges } f \ ps \ ss)!p = \text{map } \text{snd} \llbracket (p', t') \leftarrow ps. p' = p \rrbracket \sqcup_f ss!p$
(is $\bigwedge ss. \llbracket -; -; ?\text{steptype } ps \rrbracket \Longrightarrow ?P \ ss \ ps)$ *<proof>*

lemma *length-merges* [*simp*]:

$\bigwedge ss. \text{size}(\text{merges } f \ ps \ ss) = \text{size } ss$ *<proof>*

lemma (**in** *Semilat*) *merges-preserves-type-lemma*:

shows $\forall xs. xs \in \text{nlists } n \ A \longrightarrow (\forall (p, x) \in \text{set } ps. p < n \wedge x \in A)$
 $\longrightarrow \text{merges } f \ ps \ xs \in \text{nlists } n \ A$ *<proof>*

lemma (**in** *Semilat*) *merges-preserves-type* [*simp*]:

$\llbracket xs \in \text{nlists } n \ A; \forall (p, x) \in \text{set } ps. p < n \wedge x \in A \rrbracket$
 $\Longrightarrow \text{merges } f \ ps \ xs \in \text{nlists } n \ A$
<proof>

lemma (**in** *Semilat*) *list-update-le-listI* [*rule-format*]:

$\text{set } xs \subseteq A \longrightarrow \text{set } ys \subseteq A \longrightarrow xs \llbracket \sqsubseteq_r \rrbracket ys \longrightarrow p < \text{size } xs \longrightarrow$
 $x \sqsubseteq_r ys!p \longrightarrow x \in A \longrightarrow xs[p := x \sqcup_f xs!p] \llbracket \sqsubseteq_r \rrbracket ys$ *<proof>*

lemma (**in** *Semilat*) *merges-pres-le-ub*:

assumes $\text{set } ts \subseteq A \ \text{set } ss \subseteq A$
 $\forall (p, t) \in \text{set } ps. t \sqsubseteq_r ts!p \wedge t \in A \wedge p < \text{size } ts \ \text{ss} \llbracket \sqsubseteq_r \rrbracket ts$
shows $\text{merges } f \ ps \ ss \llbracket \sqsubseteq_r \rrbracket ts$ *<proof>*

end

4.10 Kildall's Algorithm

theory *Kildall-2*

imports *SemilatAlg Kildall-1*

begin

primrec *propa* :: 's binop \Rightarrow (nat \times 's) list \Rightarrow 's list \Rightarrow nat set \Rightarrow 's list * nat set

where

propa *f* [] τs *w* = ($\tau s, w$)
| *propa* *f* (*q*'#*qs*) τs *w* = (let (*q*, τ) = *q*';
 $u = \tau \sqcup_f \tau s!q$;
 $w' = (\text{if } u = \tau s!q \text{ then } w \text{ else insert } q \ w)$
in *propa* *f* *qs* ($\tau s[q := u]$) *w'*)

definition *iter* :: 's binop \Rightarrow 's step-type \Rightarrow
's list \Rightarrow nat set \Rightarrow 's list \times nat set

where

iter *f* *step* τs *w* =
while ($\lambda(\tau s, w). w \neq \{\}$)
($\lambda(\tau s, w). \text{let } p = \text{SOME } p. p \in w$
in *propa* *f* (*step* *p* ($\tau s!p$)) τs ($w - \{p\}$))
($\tau s, w$)

definition *unstabes* :: 's ord \Rightarrow 's step-type \Rightarrow 's list \Rightarrow nat set

where

unstabes *r* *step* τs = {*p*. *p* < size $\tau s \wedge \neg \text{stable } r \text{ step } \tau s$ *p*}

definition *kildall* :: 's ord \Rightarrow 's binop \Rightarrow 's step-type \Rightarrow 's list \Rightarrow 's list

where

kildall *r* *f* *step* τs = *fst*(*iter* *f* *step* τs (*unstabes* *r* *step* τs))

lemma (in *Semilat*) *merges-incr-lemma*:

$\forall xs. xs \in \text{nlists } n \ A \longrightarrow (\forall (p,x) \in \text{set } ps. p < \text{size } xs \wedge x \in A) \longrightarrow xs \sqsubseteq_r \text{merges } f \ ps \ xs$
<proof>

lemma (in *Semilat*) *merges-incr*:

$\llbracket xs \in \text{nlists } n \ A; \forall (p,x) \in \text{set } ps. p < \text{size } xs \wedge x \in A \rrbracket$
 $\implies xs \sqsubseteq_r \text{merges } f \ ps \ xs$
<proof>

lemma (in *Semilat*) *merges-same-conv* [rule-format]:

$(\forall xs. xs \in \text{nlists } n \ A \longrightarrow (\forall (p,x) \in \text{set } ps. p < \text{size } xs \wedge x \in A) \longrightarrow$
 $(\text{merges } f \ ps \ xs = xs) = (\forall (p,x) \in \text{set } ps. x \sqsubseteq_r xs!p))$ <proof>

lemma *decomp-propa*:

$\bigwedge ss \ w. (\forall (q,t) \in \text{set } qs. q < \text{size } ss) \implies$
propa *f* *qs* *ss* *w* =
 $(\text{merges } f \ qs \ ss, \{q. \exists t. (q,t) \in \text{set } qs \wedge t \sqcup_f ss!q \neq ss!q\} \cup w)$ <proof>

lemma (in *Semilat*) *stable-pres-lemma*:

shows $\llbracket \text{pres-type } \text{step } n \ A; \text{bounded } \text{step } n;$
 $ss \in \text{nlists } n \ A; p \in w; \forall q \in w. q < n;$
 $\forall q. q < n \longrightarrow q \notin w \longrightarrow \text{stable } r \ \text{step } ss \ q; q < n;$
 $\forall s'. (q, s') \in \text{set } (\text{step } p \ (ss!p)) \longrightarrow s' \sqcup_f ss!q = ss!q;$
 $q \notin w \vee q = p \rrbracket$

\implies *stable r step (merges f (step p (ss!p)) ss) q<proof>*

lemma (in *Semilat*) *merges-bounded-lemma*:

\llbracket *mono r step n A; bounded step n; pres-type step n A;*
 $\forall (p',s') \in \text{set } (\text{step } p \text{ (ss!p)}). s' \in A; ss \in \text{nlists } n \text{ A}; ts \in \text{nlists } n \text{ A}; p < n;$
 $ss \llbracket \sqsubseteq_r \rrbracket ts; \forall p. p < n \implies \text{stable r step } ts \text{ } p \rrbracket$
 $\implies \text{merges f (step p (ss!p)) ss } \llbracket \sqsubseteq_r \rrbracket ts \langle \text{proof} \rangle$

lemma *termination-lemma*: **assumes** *Semilat A r f*

shows $\llbracket ss \in \text{nlists } n \text{ A}; \forall (q,t) \in \text{set } qs. q < n \wedge t \in A; p \in w \rrbracket \implies$

$ss \llbracket \sqsubseteq_r \rrbracket \text{merges f } qs \text{ } ss \vee$

$\text{merges f } qs \text{ } ss = ss \wedge \{q. \exists t. (q,t) \in \text{set } qs \wedge t \sqcup_f ss!q \neq ss!q\} \cup (w - \{p\}) \subset w \langle \text{proof} \rangle$

end

4.11 The Lightweight Bytecode Verifier

theory *LBVSpec*

imports *SemilatAlg Opt*

begin

type-synonym

's certificate = 's list

primrec *merge* :: *'s certificate* \Rightarrow *'s binop* \Rightarrow *'s ord* \Rightarrow *'s* \Rightarrow *nat* \Rightarrow (*nat* \times *'s*) *list* \Rightarrow *'s* \Rightarrow *'s*

where

merge cert f r T pc [] $x = x$
 $| \text{merge cert f r T pc (s\#\ss)} x = \text{merge cert f r T pc ss (let (pc',s') = s in}$
 $\quad \text{if pc' = pc + 1 then } s' \sqcup_f x$
 $\quad \text{else if } s' \llbracket \sqsubseteq_r \rrbracket \text{cert!pc' then } x$
 $\quad \text{else } T)$

definition *wtl-inst* :: *'s certificate* \Rightarrow *'s binop* \Rightarrow *'s ord* \Rightarrow *'s* \Rightarrow

's step-type \Rightarrow *nat* \Rightarrow *'s* \Rightarrow *'s*

where

wtl-inst cert f r T step pc s = merge cert f r T pc (step pc s) (cert!(pc+1))

definition *wtl-cert* :: *'s certificate* \Rightarrow *'s binop* \Rightarrow *'s ord* \Rightarrow *'s* \Rightarrow *'s* \Rightarrow

's step-type \Rightarrow *nat* \Rightarrow *'s* \Rightarrow *'s*

where

wtl-cert cert f r T B step pc s =
 $(\text{if cert!pc} = B \text{ then}$
 $\quad \text{wtl-inst cert f r T step pc s}$
 else
 $\quad \text{if } s \llbracket \sqsubseteq_r \rrbracket \text{cert!pc then wtl-inst cert f r T step pc (cert!pc) else } T)$

primrec *wtl-inst-list* :: *'a list* \Rightarrow *'s certificate* \Rightarrow *'s binop* \Rightarrow *'s ord* \Rightarrow *'s* \Rightarrow *'s* \Rightarrow

's step-type \Rightarrow *nat* \Rightarrow *'s* \Rightarrow *'s*

where

wtl-inst-list [] $\text{cert f r T B step pc s} = s$
 $| \text{wtl-inst-list (i\#is) cert f r T B step pc s} =$
 $\quad (\text{let } s' = \text{wtl-cert cert f r T B step pc s in}$

if $s' = T \vee s = T$ then T else $wtl-inst-list$ is $cert\ f\ r\ T\ B\ step\ (pc+1)\ s'$

definition $cert-ok :: 's\ certificate \Rightarrow nat \Rightarrow 's \Rightarrow 's \Rightarrow 's\ set \Rightarrow bool$

where

$cert-ok\ cert\ n\ T\ B\ A \longleftrightarrow (\forall i < n. cert!i \in A \wedge cert!i \neq T) \wedge (cert!n = B)$

definition $bottom :: 'a\ ord \Rightarrow 'a \Rightarrow bool$

where

$bottom\ r\ B \longleftrightarrow (\forall x. B \sqsubseteq_r x)$

locale $lbv = Semilat +$

fixes $T :: 'a\ (\top)$

fixes $B :: 'a\ (\perp)$

fixes $step :: 'a\ step-type$

assumes $top: top\ r\ \top$

assumes $T-A: \top \in A$

assumes $bot: bottom\ r\ \perp$

assumes $B-A: \perp \in A$

fixes $merge :: 'a\ certificate \Rightarrow nat \Rightarrow (nat \times 'a)\ list \Rightarrow 'a \Rightarrow 'a$

defines $mrg-def: merge\ cert \equiv LBVSpec.merge\ cert\ f\ r\ \top$

fixes $wti :: 'a\ certificate \Rightarrow nat \Rightarrow 'a \Rightarrow 'a$

defines $wti-def: wti\ cert \equiv wtl-inst\ cert\ f\ r\ \top\ step$

fixes $wtc :: 'a\ certificate \Rightarrow nat \Rightarrow 'a \Rightarrow 'a$

defines $wtc-def: wtc\ cert \equiv wtl-cert\ cert\ f\ r\ \top\ \perp\ step$

fixes $wtl :: 'b\ list \Rightarrow 'a\ certificate \Rightarrow nat \Rightarrow 'a \Rightarrow 'a$

defines $wtl-def: wtl\ ins\ cert \equiv wtl-inst-list\ ins\ cert\ f\ r\ \top\ \perp\ step$

lemma (in lbv) wti :

$wti\ c\ pc\ s = merge\ c\ pc\ (step\ pc\ s)\ (c!(pc+1))$

$\langle proof \rangle$

lemma (in lbv) wtc :

$wtc\ c\ pc\ s = (if\ c!pc = \perp\ then\ wti\ c\ pc\ s\ else\ if\ s \sqsubseteq_r\ c!pc\ then\ wti\ c\ pc\ (c!pc)\ else\ \top)$

$\langle proof \rangle$

lemma $cert-okD1$ [intro?]:

$cert-ok\ c\ n\ T\ B\ A \Longrightarrow pc < n \Longrightarrow c!pc \in A$

$\langle proof \rangle$

lemma $cert-okD2$ [intro?]:

$cert-ok\ c\ n\ T\ B\ A \Longrightarrow c!n = B$

$\langle proof \rangle$

lemma $cert-okD3$ [intro?]:

$cert-ok\ c\ n\ T\ B\ A \Longrightarrow B \in A \Longrightarrow pc < n \Longrightarrow c!Suc\ pc \in A$

$\langle proof \rangle$

lemma $cert-okD4$ [intro?]:

$cert-ok\ c\ n\ T\ B\ A \Longrightarrow pc < n \Longrightarrow c!pc \neq T$

$\langle proof \rangle$

declare $Let-def$ [simp]

4.11.1 more semilattice lemmas

lemma (in *lbu*) *sup-top* [*simp*, *elim*]:

assumes $x: x \in A$

shows $x \sqcup_f \top = \top$ *<proof>*

lemma (in *lbu*) *plusplussup-top* [*simp*, *elim*]:

set $xs \subseteq A \implies xs \sqcup_f \top = \top$

<proof>

lemma (in *Semilat*) *pp-ub1'*:

assumes $S: \text{snd}'set\ S \subseteq A$

assumes $y: y \in A$ and $ab: (a, b) \in set\ S$

shows $b \sqsubseteq_r \text{map}\ \text{snd}\ [(p', t') \leftarrow S \cdot p' = a] \sqcup_f y$ *<proof>*

lemma (in *lbu*) *bottom-le* [*simp*, *intro!*]: $\perp \sqsubseteq_r x$

<proof>

lemma (in *lbu*) *le-bottom* [*simp*]: $x \in A \implies x \sqsubseteq_r \perp = (x = \perp)$

<proof>

4.11.2 merge

lemma (in *lbu*) *merge-Nil* [*simp*]:

$\text{merge}\ c\ pc\ []\ x = x$ *<proof>*

lemma (in *lbu*) *merge-Cons* [*simp*]:

$\text{merge}\ c\ pc\ (l\#\!ls)\ x = \text{merge}\ c\ pc\ ls$ (if $\text{fst}\ l = pc + 1$ then $\text{snd}\ l + -\!f\ x$
 else if $\text{snd}\ l \sqsubseteq_r\ c!\text{fst}\ l$ then x
 else \top)

<proof>

lemma (in *lbu*) *merge-Err* [*simp*]:

$\text{snd}'set\ ss \subseteq A \implies \text{merge}\ c\ pc\ ss\ \top = \top$

<proof>

lemma (in *lbu*) *merge-not-top*:

$\bigwedge x. \text{snd}'set\ ss \subseteq A \implies \text{merge}\ c\ pc\ ss\ x \neq \top \implies$

$\forall (pc', s') \in set\ ss. (pc' \neq pc + 1 \longrightarrow s' \sqsubseteq_r\ c!\text{pc}')$

(is $\bigwedge x. ?set\ ss \implies ?merge\ ss\ x \implies ?P\ ss$) *<proof>*

lemma (in *lbu*) *merge-def*:

shows

$\bigwedge x. x \in A \implies \text{snd}'set\ ss \subseteq A \implies$

$\text{merge}\ c\ pc\ ss\ x =$

(if $\forall (pc', s') \in set\ ss. pc' \neq pc + 1 \longrightarrow s' \sqsubseteq_r\ c!\text{pc}'$ then

$\text{map}\ \text{snd}\ [(p', t') \leftarrow ss. p' = pc + 1] \sqcup_f x$

else \top)

(is $\bigwedge x. - \implies - \implies ?merge\ ss\ x = ?if\ ss\ x$ is $\bigwedge x. - \implies - \implies ?P\ ss\ x$) *<proof>*

lemma (in *lbu*) *merge-not-top-s*:

assumes $x: x \in A$ and $ss: \text{snd}'set\ ss \subseteq A$

assumes $m: \text{merge}\ c\ pc\ ss\ x \neq \top$

shows $\text{merge}\ c\ pc\ ss\ x = (\text{map}\ \text{snd}\ [(p', t') \leftarrow ss. p' = pc + 1] \sqcup_f x)$ *<proof>*

4.11.3 wtl-inst-list

lemmas [iff] = not-Err-eq

lemma (in lbv) wtl-Nil [simp]: wtl [] c pc s = s
 ⟨proof⟩

lemma (in lbv) wtl-Cons [simp]:
 wtl (i#is) c pc s =
 (let s' = wtc c pc s in if s' = ⊤ ∨ s = ⊤ then ⊤ else wtl is c (pc+1) s')
 ⟨proof⟩

lemma (in lbv) wtl-Cons-not-top:
 wtl (i#is) c pc s ≠ ⊤ =
 (wtc c pc s ≠ ⊤ ∧ s ≠ ⊤ ∧ wtl is c (pc+1) (wtc c pc s) ≠ ⊤)
 ⟨proof⟩

lemma (in lbv) wtl-top [simp]: wtl ls c pc ⊤ = ⊤
 ⟨proof⟩

lemma (in lbv) wtl-not-top:
 wtl ls c pc s ≠ ⊤ ⇒ s ≠ ⊤
 ⟨proof⟩

lemma (in lbv) wtl-append [simp]:
 ∧ pc s. wtl (a@b) c pc s = wtl b c (pc+length a) (wtl a c pc s)
 ⟨proof⟩

lemma (in lbv) wtl-take:
 wtl is c pc s ≠ ⊤ ⇒ wtl (take pc' is) c pc s ≠ ⊤
 (is ?wtl is ≠ - ⇒ -)⟨proof⟩

lemma take-Suc:
 ∀ n. n < length l → take (Suc n) l = (take n l)@[!n] (is ?P l)⟨proof⟩

lemma (in lbv) wtl-Suc:
 assumes suc: pc+1 < length is
 assumes wtl: wtl (take pc is) c 0 s ≠ ⊤
 shows wtl (take (pc+1) is) c 0 s = wtc c pc (wtl (take pc is) c 0 s)⟨proof⟩

lemma (in lbv) wtl-all:
 assumes all: wtl is c 0 s ≠ ⊤ (is ?wtl is ≠ -)
 assumes pc: pc < length is
 shows wtc c pc (wtl (take pc is) c 0 s) ≠ ⊤⟨proof⟩

4.11.4 preserves-type

lemma (in lbv) merge-pres:
 assumes s0: snd'set ss ⊆ A and x: x ∈ A
 shows merge c pc ss x ∈ A⟨proof⟩

lemma pres-typeD2:
 pres-type step n A ⇒ s ∈ A ⇒ p < n ⇒ snd'set (step p s) ⊆ A
 ⟨proof⟩

lemma (in lbv) wti-pres [intro?]:
 assumes pres: pres-type step n A
 assumes cert: c!(pc+1) ∈ A
 assumes s-pc: s ∈ A pc < n

shows $wtl\ c\ pc\ s \in A$ ⟨proof⟩
lemma (in lbv) wtc -pres:
assumes $pres$: pres-type step $n\ A$
assumes $c!pc \in A$ and $c!(pc+1) \in A$
assumes $s \in A$ and $pc < n$
shows $wtc\ c\ pc\ s \in A$ ⟨proof⟩
lemma (in lbv) wtl -pres:
assumes $pres$: pres-type step (length is) A
assumes $cert$: cert-ok c (length is) $\top \perp A$
assumes s : $s \in A$
assumes all : $wtl\ is\ c\ 0\ s \neq \top$
shows $pc < length\ is \implies wtl\ (take\ pc\ is)\ c\ 0\ s \in A$
(is ?len $pc \implies ?wtl\ pc \in A$)⟨proof⟩
end

4.12 Correctness of the LBV

theory $LBVCorrect$
imports $LBVSpec\ Typing-Framework-1$
begin

locale $lbvs = lbv +$
fixes $s_0 :: 'a$
fixes $c :: 'a\ list$
fixes $ins :: 'b\ list$
fixes $\tau s :: 'a\ list$
defines phi -def:
 $\tau s \equiv map\ (\lambda pc.\ if\ c!pc = \perp\ then\ wtl\ (take\ pc\ ins)\ c\ 0\ s_0\ else\ c!pc)$
 $[0..<size\ ins]$

assumes $bounded$: bounded step (size ins)
assumes $cert$: cert-ok c (size ins) $\top \perp A$
assumes $pres$: pres-type step (size ins) A

lemma (in $lbvs$) phi -None [intro?]:
 $\llbracket pc < size\ ins; c!pc = \perp \rrbracket \implies \tau s!pc = wtl\ (take\ pc\ ins)\ c\ 0\ s_0$ ⟨proof⟩
lemma (in $lbvs$) phi -Some [intro?]:
 $\llbracket pc < size\ ins; c!pc \neq \perp \rrbracket \implies \tau s!pc = c!pc$ ⟨proof⟩
lemma (in $lbvs$) phi -len [simp]: $size\ \tau s = size\ ins$ ⟨proof⟩
lemma (in $lbvs$) wtl -suc- pc :
assumes all : $wtl\ ins\ c\ 0\ s_0 \neq \top$
assumes pc : $pc+1 < size\ ins$
assumes sA : $s_0 \in A$
shows $wtl\ (take\ (pc+1)\ ins)\ c\ 0\ s_0 \sqsubseteq_r \tau s!(pc+1)$ ⟨proof⟩
lemma (in $lbvs$) wtl -stable:
assumes wtl : $wtl\ ins\ c\ 0\ s_0 \neq \top$
assumes s_0 : $s_0 \in A$ and pc : $pc < size\ ins$
shows stable r step $\tau s\ pc$ ⟨proof⟩
lemma (in $lbvs$) phi -not-top:
assumes wtl : $wtl\ ins\ c\ 0\ s_0 \neq \top$ and pc : $pc < size\ ins$
shows $\tau s!pc \neq \top$ ⟨proof⟩
lemma (in $lbvs$) phi -in- A :
assumes wtl : $wtl\ ins\ c\ 0\ s_0 \neq \top$ and s_0 : $s_0 \in A$

shows $\tau s \in nlists (size\ ins) A \langle proof \rangle$
lemma (in *lbs*) *phi0*:
 assumes *wtl*: $wtl\ ins\ c\ 0\ s_0 \neq \top$ and *0*: $0 < size\ ins$ and *s0*: $s_0 \in A$
 shows $s_0 \sqsubseteq_r \tau s!0 \langle proof \rangle$
theorem (in *lbs*) *wtl-sound*:
 assumes *wtl*: $wtl\ ins\ c\ 0\ s_0 \neq \top$ and *s0*: $s_0 \in A$
 shows $\exists \tau s. wt\ step\ r\ \top\ step\ \tau s \langle proof \rangle$
theorem (in *lbs*) *wtl-sound-strong*:
 assumes *wtl*: $wtl\ ins\ c\ 0\ s_0 \neq \top$
 assumes *s0*: $s_0 \in A$ and *ins*: $0 < size\ ins$
 shows $\exists \tau s \in nlists (size\ ins) A. wt\ step\ r\ \top\ step\ \tau s \wedge s_0 \sqsubseteq_r \tau s!0 \langle proof \rangle$
 end

4.13 Completeness of the LBV

theory *LBVComplete*
imports *LBVSpec Typing-Framework-1*
begin

definition *is-target* :: $'s\ step\ type \Rightarrow 's\ list \Rightarrow nat \Rightarrow bool$ **where**
is-target $step\ \tau s\ pc' \longleftrightarrow (\exists pc\ s'. pc' \neq pc+1 \wedge pc < size\ \tau s \wedge (pc', s') \in set (step\ pc (\tau s!pc)))$

definition *make-cert* :: $'s\ step\ type \Rightarrow 's\ list \Rightarrow 's \Rightarrow 's\ certificate$ **where**
make-cert $step\ \tau s\ B = map (\lambda pc. if\ is\ target\ step\ \tau s\ pc\ then\ \tau s!pc\ else\ B) [0..<size\ \tau s] @ [B]$

lemma [*code*]:
is-target $step\ \tau s\ pc' =$
list-ex $(\lambda pc. pc' \neq pc+1 \wedge List.member (map\ fst (step\ pc (\tau s!pc))) pc') [0..<size\ \tau s] \langle proof \rangle$

locale *lbvc* = *lbv* +
fixes $\tau s :: 'a\ list$
fixes $c :: 'a\ list$
defines *cert-def*: $c \equiv make\ cert\ step\ \tau s \perp$

assumes *mono*: $mono\ r\ step (size\ \tau s) A$
assumes *pres*: $pres\ type\ step (size\ \tau s) A$
assumes τs : $\forall pc < size\ \tau s. \tau s!pc \in A \wedge \tau s!pc \neq \top$
assumes *bounded*: $bounded\ step (size\ \tau s)$

assumes *B-neq-T*: $\perp \neq \top$

lemma (in *lbvc*) *cert*: $cert\ ok\ c (size\ \tau s) \top \perp A \langle proof \rangle$

lemmas [*simp del*] = *split-paired-Ex*

lemma (in *lbvc*) *cert-target* [*intro?*]:
 $\llbracket (pc', s') \in set (step\ pc (\tau s!pc));$
 $pc' \neq pc+1; pc < size\ \tau s; pc' < size\ \tau s \rrbracket$
 $\implies c!pc' = \tau s!pc' \langle proof \rangle$

lemma (in *lbvc*) *cert-approx* [*intro?*]:
 $\llbracket pc < size\ \tau s; c!pc \neq \perp \rrbracket \implies c!pc = \tau s!pc \langle proof \rangle$

lemma (in *lbv*) *le-top* [*simp, intro*]: $x <=-r\ \top \langle proof \rangle$

lemma (in *lbv*) *merge-mono*:
assumes *less*: $set\ ss_2 \sqsubseteq_r set\ ss_1$
assumes *x*: $x \in A$
assumes *ss1*: $snd'set\ ss_1 \subseteq A$
assumes *ss2*: $snd'set\ ss_2 \subseteq A$
assumes *boun*: $\forall x \in (fst'set\ ss_1). x < size\ \tau s$
assumes *cert*: $cert-ok\ c\ (size\ \tau s)\ T\ B\ A$
shows $merge\ c\ pc\ ss_2\ x \sqsubseteq_r merge\ c\ pc\ ss_1\ x$ (is $?s_2 \sqsubseteq_r ?s_1$)*<proof>*

lemma (in *lbvc*) *wti-mono*:
assumes *less*: $s_2 \sqsubseteq_r s_1$
assumes *pc*: $pc < size\ \tau s$ **and** *s1*: $s_1 \in A$ **and** *s2*: $s_2 \in A$
shows $wti\ c\ pc\ s_2 \sqsubseteq_r wti\ c\ pc\ s_1$ (is $?s_2' \sqsubseteq_r ?s_1'$)*<proof>*

lemma (in *lbvc*) *wtc-mono*:
assumes *less*: $s_2 \sqsubseteq_r s_1$
assumes *pc*: $pc < size\ \tau s$ **and** *s1*: $s_1 \in A$ **and** *s2*: $s_2 \in A$
shows $wtc\ c\ pc\ s_2 \sqsubseteq_r wtc\ c\ pc\ s_1$ (is $?s_2' \sqsubseteq_r ?s_1'$)*<proof>*

lemma (in *lbv*) *top-le-conv* [*simp*]: $x \in A \implies \top \sqsubseteq_r x = (x = \top)$ *<proof>*

lemma (in *lbv*) *neq-top* [*simp*, *elim*]: $[x \sqsubseteq_r y; y \neq \top; y \in A] \implies x \neq \top$ *<proof>*

lemma (in *lbvc*) *stable-wti*:
assumes *stable*: $stable\ r\ step\ \tau s\ pc$ **and** *pc*: $pc < size\ \tau s$
shows $wti\ c\ pc\ (\tau s!pc) \neq \top$ *<proof>*

lemma (in *lbvc*) *wti-less*:
assumes *stable*: $stable\ r\ step\ \tau s\ pc$ **and** *suc-pc*: $Suc\ pc < size\ \tau s$
shows $wti\ c\ pc\ (\tau s!pc) \sqsubseteq_r \tau s!Suc\ pc$ (is $?wti \sqsubseteq_r -$)*<proof>*

lemma (in *lbvc*) *stable-wtc*:
assumes *stable*: $stable\ r\ step\ \tau s\ pc$ **and** *pc*: $pc < size\ \tau s$
shows $wtc\ c\ pc\ (\tau s!pc) \neq \top$ *<proof>*

lemma (in *lbvc*) *wtc-less*:
assumes *stable*: $stable\ r\ step\ \tau s\ pc$ **and** *suc-pc*: $Suc\ pc < size\ \tau s$
shows $wtc\ c\ pc\ (\tau s!pc) \sqsubseteq_r \tau s!Suc\ pc$ (is $?wtc \sqsubseteq_r -$)*<proof>*

lemma (in *lbvc*) *wt-step-wtl-lemma*:
assumes *wt-step*: $wt-step\ r\ \top\ step\ \tau s$
shows $\bigwedge pc\ s. pc + size\ ls = size\ \tau s \implies s \sqsubseteq_r \tau s!pc \implies s \in A \implies s \neq \top \implies$
 $wtl\ ls\ c\ pc\ s \neq \top$
(is $\bigwedge pc\ s. - \implies - \implies - \implies - \implies ?wtl\ ls\ pc\ s \neq -$)*<proof>*

theorem (in *lbvc*) *wtl-complete*:
assumes *wt*: $wt-step\ r\ \top\ step\ \tau s$
assumes *s*: $s \sqsubseteq_r \tau s!0$ $s \in A$ $s \neq \top$ **and** *eq*: $size\ ins = size\ \tau s$
shows $wtl\ ins\ c\ 0\ s \neq \top$ *<proof>*

end

4.14 The Jinja Type System as a Semilattice

theory *SemiType*
imports *../Common/WellForm ../DFA/Semilattices*
begin

definition *super* :: 'a prog \Rightarrow cname \Rightarrow cname
where $super\ P\ C \equiv fst\ (the\ (class\ P\ C))$

lemma *superI*:
 $(C, D) \in subcls1\ P \implies super\ P\ C = D$
<proof>

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))

$\langle proof \rangle$

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*:

$wf-prog m P \Longrightarrow is-class P C = P \vdash C \preceq^* Object \langle proof \rangle$

lemma *subcls-antisym*:

$\llbracket wf-prog m P; P \vdash C \preceq^* D; P \vdash D \preceq^* C \rrbracket \Longrightarrow C = D$

$\langle proof \rangle$

lemma *widen-antisym*:

$\llbracket wf-prog m P; P \vdash T \leq U; P \vdash U \leq T \rrbracket \Longrightarrow T = U \langle proof \rangle$

lemma *order-widen* [*intro,simp*]:

$wf-prog m P \Longrightarrow order (subtype P) (types P) \langle proof \rangle$

lemma *NT-widen*:

$P \vdash NT \leq T = (T = NT \vee (\exists C. T = Class C)) \langle proof \rangle$

lemma *Class-widen2*: $P \vdash Class C \leq T = (\exists D. T = Class D \wedge P \vdash C \preceq^* D) \langle proof \rangle$

lemma *wf-converse-subcls1-impl-acc-subtype*:

$wf ((subcls1 P) \hat{-} 1) \implies acc (subtype P) \langle proof \rangle$

lemma *wf-subtype-acc* [intro, simp]:

$wf\text{-prog } wf\text{-mb } P \implies acc (subtype P) \langle proof \rangle$

lemma *exec-lub-refl* [simp]: $exec\text{-lub } r f T T = T \langle proof \rangle$

lemma *closed-err-types*:

$wf\text{-prog } wf\text{-mb } P \implies closed (err (types P)) (lift2 (sup P)) \langle proof \rangle$

lemma *sup-subtype-greater*:

$\llbracket wf\text{-prog } wf\text{-mb } P; is\text{-type } P t1; is\text{-type } P t2; sup P t1 t2 = OK s \rrbracket$
 $\implies subtype P t1 s \wedge subtype P t2 s \langle proof \rangle$

lemma *sup-subtype-smallest*:

$\llbracket wf\text{-prog } wf\text{-mb } P; is\text{-type } P a; is\text{-type } P b; is\text{-type } P c;$
 $subtype P a c; subtype P b c; sup P a b = OK d \rrbracket$
 $\implies subtype P d c \langle proof \rangle$

lemma *sup-exists*:

$\llbracket subtype P a c; subtype P b c \rrbracket \implies \exists T. sup P a b = OK T \langle proof \rangle$

lemma *err-semilat-JType-esl*:

$wf\text{-prog } wf\text{-mb } P \implies err\text{-semilat } (esl P) \langle proof \rangle$

end

4.15 The JVM Type System as Semilattice

theory *JVM-SemiType* **imports** *SemiType* **begin**

type-synonym $ty_l = ty\ err\ list$

type-synonym $ty_s = ty\ list$

type-synonym $ty_i = ty_s \times ty_l$

type-synonym $ty_i' = ty_i\ option$

type-synonym $ty_m = ty_i'\ list$

type-synonym $ty_P = mname \Rightarrow cname \Rightarrow ty_m$

definition $stk\text{-esl} :: 'c\ prog \Rightarrow nat \Rightarrow ty_s\ esl$

where

$stk\text{-esl } P\ mxs \equiv upto\text{-esl } mxs (SemiType.esl P)$

definition $loc\text{-sl} :: 'c\ prog \Rightarrow nat \Rightarrow ty_l\ sl$

where

$loc\text{-sl } P\ mxl \equiv Listn.sl\ mxl (Err.sl (SemiType.esl P))$

definition $sl :: 'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ sl$

where

$sl P\ mxs\ mxl \equiv$
 $Err.sl (Opt.esl (Product.esl (stk\text{-esl } P\ mxs) (Err.esl (loc\text{-sl } P\ mxl))))$

definition $states :: 'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ set$

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 $sup-ty-opt :: ['c prog, ty err, ty err] \Rightarrow bool$

$$(- \vdash - \leq_{\top} - [71, 71, 71] 70)$$

where

$$sup-ty-opt P \equiv Err.le (subtype P)$$

definition $sup-state :: ['c prog, ty_i, ty_i] \Rightarrow bool$

$$(- \vdash - \leq_i - [71, 71, 71] 70)$$

where

$$sup-state P \equiv Product.le (Listn.le (subtype P)) (Listn.le (sup-ty-opt P))$$

definition $sup-state-opt :: ['c prog, ty_i', ty_i'] \Rightarrow bool$

$$(- \vdash - \leq'' - [71, 71, 71] 70)$$

where

$$sup-state-opt P \equiv Opt.le (sup-state P)$$

abbreviation

$$sup-loc :: ['c prog, ty_l, ty_l] \Rightarrow bool \quad (- \vdash - [\leq_{\top}] - [71, 71, 71] 70)$$

$$\text{where } P \vdash LT [\leq_{\top}] LT' \equiv list-all2 (sup-ty-opt P) LT LT'$$

notation (ASCII)

$$sup-ty-opt \quad (- | - - \leq = T - [71, 71, 71] 70) \text{ and}$$

$$sup-state \quad (- | - - \leq = i - [71, 71, 71] 70) \text{ and}$$

$$sup-state-opt \quad (- | - - \leq = ' - [71, 71, 71] 70) \text{ and}$$

$$sup-loc \quad (- | - - [\leq = T] - [71, 71, 71] 70)$$

4.15.1 Unfolding

lemma *JVM-states-unfold*:

$$states P mxs mxl \equiv err(opt((Union \{nlists n (types P) \mid n. n \leq mxs\}) \times nlists mxl (err(types P))))\langle proof \rangle$$

lemma *JVM-le-unfold*:

$$le P m n \equiv$$

$$Err.le(Opt.le(Product.le(Listn.le(subtype P))(Listn.le(Err.le(subtype P))))\langle proof \rangle$$

lemma *sl-def2*:

$$JVM-SemiType.sl P mxs mxl \equiv$$

$$(states P mxs mxl, JVM-SemiType.le P mxs mxl, JVM-SemiType.sup P mxs mxl)\langle proof \rangle$$

lemma *JVM-le-conv*:

$$le P m n (OK t1) (OK t2) = P \vdash t1 \leq' t2 \langle proof \rangle$$

lemma *JVM-le-Err-conv*:

$$le P m n = Err.le (sup-state-opt P) \langle proof \rangle$$

lemma *err-le-unfold [iff]*:

$$Err.le r (OK a) (OK b) = r a b \langle proof \rangle$$

4.15.2 Semilattice

lemma *order-sup-state-opt' [intro, simp]*:

$$wf-prog wf-mb P \implies$$

order (sup-state-opt P) (opt ((\bigcup {nlists n (types P) | n. n ≤ max}) × nlists (Suc (length Ts + mxl₀)) (err (types P))))<proof><proof><proof><proof><proof><proof><proof><proof><proof><proof><proof><proof><proof><proof><proof>

4.16 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 max :: 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: mxl ≡ 1 + size Ts + mxl₀

 Program counter of successor instructions:

primrec succs :: instr ⇒ ty_i ⇒ pc ⇒ 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 (Putfield F C) 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-Throw:

 succs Throw τ pc = []

 Effect of instruction on the state type:

fun the-class :: ty ⇒ cname **where**

 the-class (Class C) = C

fun eff_i :: instr × 'm prog × ty_i ⇒ ty_i **where**

 eff_i-Load:

 eff_i (Load n, P, (ST, LT)) = (ok-val (LT ! n) # ST, LT)

| *eff_i-Store*:
 $\text{eff}_i (\text{Store } n, P, (T\#ST, LT)) = (ST, LT[n:= OK T])$

| *eff_i-Push*:
 $\text{eff}_i (\text{Push } v, P, (ST, LT)) = (\text{the } (\text{typeof } v) \# ST, LT)$

| *eff_i-Getfield*:
 $\text{eff}_i (\text{Getfield } F C, P, (T\#ST, LT)) = (\text{snd } (\text{field } P C F) \# ST, LT)$

| *eff_i-Putfield*:
 $\text{eff}_i (\text{Putfield } F C, P, (T_1\#T_2\#ST, LT)) = (ST, LT)$

| *eff_i-New*:
 $\text{eff}_i (\text{New } C, P, (ST, LT)) = (\text{Class } C \# ST, LT)$

| *eff_i-Checkcast*:
 $\text{eff}_i (\text{Checkcast } C, P, (T\#ST, LT)) = (\text{Class } C \# ST, LT)$

| *eff_i-Pop*:
 $\text{eff}_i (\text{Pop}, P, (T\#ST, LT)) = (ST, LT)$

| *eff_i-IAdd*:
 $\text{eff}_i (\text{IAdd}, P, (T_1\#T_2\#ST, LT)) = (\text{Integer}\#ST, LT)$

| *eff_i-CmpEq*:
 $\text{eff}_i (\text{CmpEq}, P, (T_1\#T_2\#ST, LT)) = (\text{Boolean}\#ST, LT)$

| *eff_i-IfFalse*:
 $\text{eff}_i (\text{IfFalse } b, P, (T_1\#ST, LT)) = (ST, LT)$

| *eff_i-Invoke*:
 $\text{eff}_i (\text{Invoke } M n, P, (ST, LT)) =$
 $(\text{let } C = \text{the-class } (ST!n); (D, Ts, Tr, b) = \text{method } P C M$
 $\text{in } (Tr \# \text{drop } (n+1) ST, LT))$

| *eff_i-Goto*:
 $\text{eff}_i (\text{Goto } n, P, s) = s$

fun *is-relevant-class* :: *instr* ⇒ *'m prog* ⇒ *cname* ⇒ *bool* **where**

rel-Getfield:
 $\text{is-relevant-class } (\text{Getfield } F D) = (\lambda P C. P \vdash \text{NullPointer} \preceq^* C)$

| *rel-Putfield*:
 $\text{is-relevant-class } (\text{Putfield } F D) = (\lambda P C. P \vdash \text{NullPointer} \preceq^* C)$

| *rel-Checkcast*:
 $\text{is-relevant-class } (\text{Checkcast } D) = (\lambda P C. P \vdash \text{ClassCast} \preceq^* C)$

| *rel-New*:
 $\text{is-relevant-class } (\text{New } D) = (\lambda P C. P \vdash \text{OutOfMemory} \preceq^* C)$

| *rel-Throw*:
 $\text{is-relevant-class } \text{Throw} = (\lambda P C. \text{True})$

| *rel-Invoke*:
 $\text{is-relevant-class } (\text{Invoke } M n) = (\lambda P C. \text{True})$

| *rel-default*:
 $\text{is-relevant-class } i = (\lambda P C. \text{False})$

definition *is-relevant-entry* :: *'m prog* ⇒ *instr* ⇒ *pc* ⇒ *ex-entry* ⇒ *bool* **where**

$\text{is-relevant-entry } P i pc e \iff (\text{let } (f, t, C, h, d) = e \text{ in } \text{is-relevant-class } i P C \wedge pc \in \{f..<t\})$

definition *relevant-entries* :: *'m prog* ⇒ *instr* ⇒ *pc* ⇒ *ex-table* ⇒ *ex-table* **where**

$\text{relevant-entries } P i pc = \text{filter } (\text{is-relevant-entry } P i pc)$

definition *xcpt-eff* :: *instr* ⇒ *'m prog* ⇒ *pc* ⇒ *ty_i*

⇒ *ex-table* ⇒ (*pc* × *ty_i*) *list* **where**

$\text{xcpt-eff } i P pc \tau et = (\text{let } (ST, LT) = \tau \text{ in}$

$\text{map } (\lambda (f, t, C, h, d). (h, \text{Some } (\text{Class } C \# \text{drop } (\text{size } ST - d) ST, LT))) (\text{relevant-entries } P i pc et))$

definition $norm\text{-}eff :: instr \Rightarrow 'm\ prog \Rightarrow nat \Rightarrow ty_i \Rightarrow (pc \times ty_i')$ list where
 $norm\text{-}eff\ i\ P\ pc\ \tau = map\ (\lambda pc'. (pc', Some\ (eff_i\ (i, P, \tau))))\ (succs\ i\ \tau\ pc)$

definition $eff :: instr \Rightarrow 'm\ prog \Rightarrow pc \Rightarrow ex\text{-}table \Rightarrow ty_i' \Rightarrow (pc \times ty_i')$ list where
 $eff\ i\ P\ pc\ et\ t = (case\ t\ of$
 $\quad None \Rightarrow []$
 $\quad | Some\ \tau \Rightarrow (norm\text{-}eff\ i\ P\ pc\ \tau) @ (xcpt\text{-}eff\ i\ P\ pc\ \tau\ et))$

lemma $eff\text{-}None$:

$eff\ i\ P\ pc\ xt\ None = []$
 $\langle proof \rangle$

lemma $eff\text{-}Some$:

$eff\ i\ P\ pc\ xt\ (Some\ \tau) = norm\text{-}eff\ i\ P\ pc\ \tau @ xcpt\text{-}eff\ i\ P\ pc\ \tau\ xt$
 $\langle proof \rangle$

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

$app_i\text{-}Load$:
 $app_i\ (Load\ n, P, pc, mxs, T_r, (ST, LT)) =$
 $(n < length\ LT \wedge LT\ !\ n \neq Err \wedge length\ ST < mxs)$
 $| app_i\text{-}Store$:
 $app_i\ (Store\ n, P, pc, mxs, T_r, (T\#ST, LT)) =$
 $(n < length\ LT)$
 $| app_i\text{-}Push$:
 $app_i\ (Push\ v, P, pc, mxs, T_r, (ST, LT)) =$
 $(length\ ST < mxs \wedge typeof\ v \neq None)$
 $| app_i\text{-}Getfield$:
 $app_i\ (Getfield\ F\ C, P, pc, mxs, T_r, (T\#ST, LT)) =$
 $(\exists T_f. P \vdash C\ sees\ F:T_f\ in\ C \wedge P \vdash T \leq Class\ C)$
 $| app_i\text{-}Putfield$:
 $app_i\ (Putfield\ F\ C, P, pc, mxs, T_r, (T_1\#T_2\#ST, LT)) =$
 $(\exists T_f. P \vdash C\ sees\ F:T_f\ in\ C \wedge P \vdash T_2 \leq (Class\ C) \wedge P \vdash T_1 \leq T_f)$
 $| app_i\text{-}New$:
 $app_i\ (New\ C, P, pc, mxs, T_r, (ST, LT)) =$
 $(is\text{-}class\ P\ C \wedge length\ ST < mxs)$
 $| app_i\text{-}Checkcast$:
 $app_i\ (Checkcast\ C, P, pc, mxs, T_r, (T\#ST, LT)) =$
 $(is\text{-}class\ P\ C \wedge is\text{-}refT\ T)$
 $| app_i\text{-}Pop$:
 $app_i\ (Pop, P, pc, mxs, T_r, (T\#ST, LT)) =$
 $True$
 $| app_i\text{-}IAdd$:
 $app_i\ (IAdd, P, pc, mxs, T_r, (T_1\#T_2\#ST, LT)) = (T_1 = T_2 \wedge T_1 = Integer)$
 $| app_i\text{-}CmpEq$:
 $app_i\ (CmpEq, P, pc, mxs, T_r, (T_1\#T_2\#ST, LT)) =$
 $(T_1 = T_2 \vee is\text{-}refT\ T_1 \wedge is\text{-}refT\ T_2)$
 $| app_i\text{-}IfFalse$:
 $app_i\ (IfFalse\ b, P, pc, mxs, T_r, (Boolean\#ST, LT)) =$
 $(0 \leq int\ pc + b)$
 $| app_i\text{-}Goto$:
 $app_i\ (Goto\ b, P, pc, mxs, T_r, s) =$
 $(0 \leq int\ pc + b)$

| *app_i-Return*:
 $app_i (Return, P, pc, mxs, T_r, (T\#ST,LT)) = (P \vdash T \leq T_r)$

| *app_i-Throw*:
 $app_i (Throw, P, pc, mxs, T_r, (T\#ST,LT)) = is-refT T$

| *app_i-Invoke*:
 $app_i (Invoke M n, P, pc, mxs, T_r, (ST,LT)) = (n < length ST \wedge (ST!n \neq NT \rightarrow (\exists C D Ts T m. ST!n = Class C \wedge P \vdash C sees M:Ts \rightarrow T = m in D \wedge P \vdash rev (take n ST) [\leq] Ts)))$

| *app_i-default*:
 $app_i (i,P, pc,mxs,T_r,s) = False$

definition *xcpt-app* :: *instr* \Rightarrow *'m prog* \Rightarrow *pc* \Rightarrow *nat* \Rightarrow *ex-table* \Rightarrow *ty_i* \Rightarrow *bool* **where**
 $xcpt-app i P pc mxs xt \tau \longleftrightarrow (\forall (f,t,C,h,d) \in set (relevant-entries P i pc xt). is-class P C \wedge d \leq size (fst \tau) \wedge d < mxs)$

definition *app* :: *instr* \Rightarrow *'m prog* \Rightarrow *nat* \Rightarrow *ty* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *ex-table* \Rightarrow *ty_i'* \Rightarrow *bool* **where**
 $app i P mxs T_r pc mpc xt t = (case t of None \Rightarrow True | Some \tau \Rightarrow app_i (i,P,pc,mxs,T_r,\tau) \wedge xcpt-app i P pc mxs xt \tau \wedge (\forall (pc',\tau') \in set (eff i P pc xt t). pc' < mpc))$

lemma *app-Some*:
 $app i P mxs T_r pc mpc xt (Some \tau) = (app_i (i,P,pc,mxs,T_r,\tau) \wedge xcpt-app i P pc mxs xt \tau \wedge (\forall (pc',s') \in set (eff i P pc xt (Some \tau)). pc' < mpc))$
 <proof>

locale *eff* = *jvm-method* +
fixes *eff_i* **and** *app_i* **and** *eff* **and** *app*
fixes *norm-eff* **and** *xcpt-app* **and** *xcpt-eff*

fixes *mpc*
defines *mpc* \equiv *size is*

defines *eff_i* *i* $\tau \equiv Effect.eff_i (i,P,\tau)$
notes *eff_i-simps* [*simp*] = *Effect.eff_i.simps* [**where** *P* = *P*, *folded eff_i-def*]

defines *app_i* *i* *pc* $\tau \equiv Effect.app_i (i, P, pc, mxs, T_r, \tau)$
notes *app_i-simps* [*simp*] = *Effect.app_i.simps* [**where** *P*=*P* **and** *mxs*=*mxs* **and** *T_r*=*T_r*, *folded app_i-def*]

defines *xcpt-eff* *i* *pc* $\tau \equiv Effect.xcpt-eff i P pc \tau xt$
notes *xcpt-eff* = *Effect.xcpt-eff-def* [*of - P - - xt*, *folded xcpt-eff-def*]

defines *norm-eff* *i* *pc* $\tau \equiv Effect.norm-eff i P pc \tau$
notes *norm-eff* = *Effect.norm-eff-def* [*of - P*, *folded norm-eff-def eff_i-def*]

defines *eff* *i* *pc* $\equiv Effect.eff i P pc xt$

notes $eff = Effect.eff-def$ [of - P - xt , folded $eff-def$ $norm-eff-def$ $xcpt-eff-def$]

defines $xcpt-app$ i pc $\tau \equiv Effect.xcpt-app$ i P pc mxs xt τ

notes $xcpt-app = Effect.xcpt-app-def$ [of - P - mxs xt , folded $xcpt-app-def$]

defines app i $pc \equiv Effect.app$ i P mxs T_r pc mpc xt

notes $app = Effect.app-def$ [of - P mxs T_r - mpc xt , folded $app-def$ $xcpt-app-def$ app_i-def $eff-def$]

lemma $length-cases2$:

assumes $\bigwedge LT. P (\[], LT)$

assumes $\bigwedge l ST LT. P (l\#ST, LT)$

shows $P s$

$\langle proof \rangle$

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 $P s \langle proof \rangle$

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 $P s \langle proof \rangle$

simp rules for app

lemma $appNone[simp]$: app i P mxs T_r pc mpc et $None = True$

$\langle proof \rangle$

lemma $appLoad[simp]$:

app_i ($Load$ idx, P, T_r, mxs, pc, s) = $(\exists ST LT. s = (ST, LT) \wedge idx < length LT \wedge LT!idx \neq Err \wedge length ST < mxs)$

$\langle proof \rangle$

lemma $appStore[simp]$:

app_i ($Store$ idx, P, pc, mxs, T_r, s) = $(\exists ts ST LT. s = (ts\#ST, LT) \wedge idx < length LT)$

$\langle proof \rangle$

lemma $appPush[simp]$:

app_i ($Push$ v, P, pc, mxs, T_r, s) =

$(\exists ST LT. s = (ST, LT) \wedge length ST < mxs \wedge typeof v \neq None)$

$\langle proof \rangle$

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 $F:vT$ in $C \wedge P \vdash oT \leq (Class C))$

$\langle proof \rangle$

lemma $appPutField[simp]$:

app_i (Putfield $F C, P, pc, mxs, T_r, s$) =
 $(\exists vT vT' oT ST LT. s = (vT\#oT\#ST, LT) \wedge$
 $P \vdash C \text{ sees } F:vT' \text{ in } C \wedge P \vdash oT \leq (\text{Class } C) \wedge P \vdash vT \leq vT')$
 ⟨proof⟩

lemma $appNew[simp]$:
 app_i (New C, P, pc, mxs, T_r, s) =
 $(\exists ST LT. s = (ST, LT) \wedge \text{is-class } P C \wedge \text{length } ST < mxs)$
 ⟨proof⟩

lemma $appCheckcast[simp]$:
 app_i (Checkcast C, P, pc, mxs, T_r, s) =
 $(\exists T ST LT. s = (T\#ST, LT) \wedge \text{is-class } P C \wedge \text{is-refT } T)$
 ⟨proof⟩

lemma $app_iPop[simp]$:
 app_i (Pop, P, pc, mxs, T_r, s) = $(\exists ts ST LT. s = (ts\#ST, LT))$
 ⟨proof⟩

lemma $appIAdd[simp]$:
 app_i (IAdd, P, pc, mxs, T_r, s) = $(\exists ST LT. s = (\text{Integer}\#\text{Integer}\#ST, LT))$ ⟨proof⟩

lemma $appIfFalse [simp]$:
 app_i (IfFalse b, P, pc, mxs, T_r, s) =
 $(\exists ST LT. s = (\text{Boolean}\#ST, LT) \wedge 0 \leq \text{int } pc + b)$ ⟨proof⟩

lemma $appCmpEq[simp]$:
 app_i (CmpEq, P, pc, mxs, T_r, s) =
 $(\exists T_1 T_2 ST LT. s = (T_1\#T_2\#ST, LT) \wedge (\neg \text{is-refT } T_1 \wedge T_2 = T_1 \vee \text{is-refT } T_1 \wedge \text{is-refT } T_2))$
 ⟨proof⟩

lemma $appReturn[simp]$:
 app_i (Return, P, pc, mxs, T_r, s) = $(\exists T ST LT. s = (T\#ST, LT) \wedge P \vdash T \leq T_r)$
 ⟨proof⟩

lemma $appThrow[simp]$:
 app_i (Throw, P, pc, mxs, T_r, s) = $(\exists T ST LT. s = (T\#ST, LT) \wedge \text{is-refT } T)$
 ⟨proof⟩

lemma $effNone$:
 $(pc', s') \in \text{set } (\text{eff } i P pc \text{ et } None) \implies s' = None$
 ⟨proof⟩

some helpers to make the specification directly executable:

lemma $relevant-entries-append [simp]$:
 $\text{relevant-entries } P i pc (xt @ xt') = \text{relevant-entries } P i pc xt @ \text{relevant-entries } P i pc xt'$
 ⟨proof⟩

lemma $xcpt-app-append [iff]$:
 $xcpt\text{-app } i P pc mxs (xt@xt') \tau = (xcpt\text{-app } i P pc mxs xt \tau \wedge xcpt\text{-app } i P pc mxs xt' \tau)$
 ⟨proof⟩

lemma $xcpt-eff-append [simp]$:
 $xcpt\text{-eff } i P pc \tau (xt@xt') = xcpt\text{-eff } i P pc \tau xt @ xcpt\text{-eff } i P pc \tau xt'$
 ⟨proof⟩

lemma *app-append* [*simp*]:

$app\ i\ P\ pc\ T\ mxs\ mpc\ (xt@xt')\ \tau = (app\ i\ P\ pc\ T\ mxs\ mpc\ xt\ \tau \wedge app\ i\ P\ pc\ T\ mxs\ mpc\ xt'\ \tau)$
<proof>

end

4.17 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 $app_i\ (i,P,mxs,mpc,rT,\tau') \implies app_i\ (i,P,mxs,mpc,rT,\tau)$ *<proof>*

lemma *succs-mono*:

assumes *wf*: *wf-prog* *p* *P* **and** *app_i*: $app_i\ (i,P,mxs,mpc,rT,\tau')$

shows $P \vdash \tau \leq_i \tau' \implies set\ (succs\ i\ \tau\ pc) \subseteq set\ (succs\ i\ \tau'\ pc)$ *<proof>*

lemma *app-mono*:

assumes *wf*: *wf-prog* *p* *P*

assumes *less'*: $P \vdash \tau \leq' \tau'$

shows $app\ i\ P\ m\ rT\ pc\ mpc\ xt\ \tau' \implies app\ i\ P\ m\ rT\ pc\ mpc\ xt\ \tau$ *<proof>*

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*: $succs\ i\ \tau\ pc \neq [] \wedge succs\ i\ \tau'\ pc \neq []$

shows $P \vdash eff_i\ (i,P,\tau) \leq_i eff_i\ (i,P,\tau')$ *<proof>*

end

4.18 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

$wt_instr :: ['m\ prog, ty, nat, pc, ex_table, instr, pc, ty_m] \Rightarrow bool$

$(\tau, \tau', \tau', \tau' \vdash \tau, \tau' :: - [60, 0, 0, 0, 0, 0, 0, 61] 60)$

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 $wt-start :: [m\ prog, cname, ty\ list, nat, ty_m] \Rightarrow bool$

where

$$wt-start\ P\ C\ Ts\ mxl_0\ \tau s \equiv$$

$$P \vdash Some\ ([], OK\ (Class\ C)\#map\ OK\ Ts@replicate\ mxl_0\ Err) \leq' \tau s!0$$

- 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 $wt-method :: [m\ prog, cname, ty\ list, ty, nat, nat, instr\ list,$
 $ex-table, ty_m] \Rightarrow bool$

where

$$wt-method\ P\ C\ Ts\ T_r\ mxs\ mxl_0\ is\ xt\ \tau s \equiv$$

$$0 < size\ is \wedge size\ \tau s = size\ is \wedge$$

$$check-types\ P\ mxs\ (1+size\ Ts+mxl_0)\ (map\ OK\ \tau s) \wedge$$

$$wt-start\ P\ C\ Ts\ mxl_0\ \tau s \wedge$$

$$(\forall pc < size\ is.\ P, T_r, mxs, size\ is, xt \vdash is!pc, pc :: \tau s)$$

— A program is welltyped if it is wellformed and all methods are welltyped

definition $wf-jvm-prog-phi :: ty_P \Rightarrow jvm-prog \Rightarrow bool\ (wf'-jvm'-prog-)$

where

$$wf-jvm-prog_{\Phi} \equiv$$

$$wf-prog\ (\lambda P\ C\ (M, Ts, T_r, (mxs, mxl_0, is, xt)).$$

$$wt-method\ P\ C\ Ts\ T_r\ mxs\ mxl_0\ is\ xt\ (\Phi\ C\ M))$$

definition $wf-jvm-prog :: jvm-prog \Rightarrow bool$

where

$$wf-jvm-prog\ P \equiv \exists \Phi.\ wf-jvm-prog_{\Phi}\ P$$

lemma $wt-jvm-progD$:

$$wf-jvm-prog_{\Phi}\ P \Longrightarrow \exists wt.\ wf-prog\ wt\ P \langle proof \rangle$$

lemma $wt-jvm-prog-impl-wt-instr$:

assumes wf : $wf-jvm-prog_{\Phi}\ P$ **and**

$$sees$$
: $P \vdash C\ sees\ M:Ts \rightarrow T = (mxs, mxl_0, ins, xt)$ in C **and**

$$pc$$
: $pc < size\ ins$

shows $P, T, mxs, size\ ins, xt \vdash ins!pc, pc :: \Phi\ C\ M \langle proof \rangle$

lemma $wt-jvm-prog-impl-wt-start$:

assumes wf : $wf-jvm-prog_{\Phi}\ P$ **and**

$$sees$$
: $P \vdash C\ sees\ M:Ts \rightarrow T = (mxs, mxl_0, ins, xt)$ in C

shows $0 < size\ ins \wedge wt-start\ P\ C\ Ts\ mxl_0\ (\Phi\ C\ M) \langle proof \rangle$

4.19 The Typing Framework for the JVM

theory $TF-JVM$

imports $../DFA/Typing-Framework-err\ EffectMono\ BVSpec$

begin

definition $exec :: jvm-prog \Rightarrow nat \Rightarrow ty \Rightarrow ex-table \Rightarrow instr\ list \Rightarrow ty_i' \text{ err step-type}$
where

$exec\ G\ mxs\ rT\ et\ bs \equiv$
 $err\text{-step}\ (size\ bs)\ (\lambda pc.\ app\ (bs!pc)\ G\ mxs\ rT\ pc\ (size\ bs)\ et)$
 $(\lambda pc.\ eff\ (bs!pc)\ G\ pc\ et)$

locale $JVM\text{-}sl =$

fixes $P :: jvm-prog$ **and** mxs **and** mxl_0 **and** n
fixes $Ts :: ty\ list$ **and** is **and** xt **and** T_r

fixes mxl **and** A **and** r **and** f **and** app **and** eff **and** $step$

defines $[simp]: mxl \equiv 1 + size\ Ts + mxl_0$

defines $[simp]: A \equiv states\ P\ mxs\ mxl$

defines $[simp]: r \equiv JVM\text{-}SemiType.le\ P\ mxs\ mxl$

defines $[simp]: f \equiv JVM\text{-}SemiType.sup\ P\ mxs\ mxl$

defines $[simp]: app \equiv \lambda pc.\ Effect.app\ (is!pc)\ P\ mxs\ T_r\ pc\ (size\ is)\ xt$

defines $[simp]: eff \equiv \lambda pc.\ Effect.eff\ (is!pc)\ P\ pc\ xt$

defines $[simp]: step \equiv err\text{-step}\ (size\ is)\ app\ eff$

defines $[simp]: n \equiv size\ is$

locale $start\text{-}context = JVM\text{-}sl +$

fixes p **and** C

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

assumes $C: is\text{-}class\ P\ C$

assumes $Ts: set\ Ts \subseteq types\ P$

fixes $first :: ty_i'$ **and** $start$

defines $[simp]:$

$first \equiv Some\ ([], OK\ (Class\ C)\ \# \ map\ OK\ Ts\ @ \ replicate\ mxl_0\ Err)$

defines $[simp]:$

$start \equiv OK\ first\ \# \ replicate\ (size\ is - 1)\ (OK\ None)$

4.19.1 Connecting JVM and Framework

lemma (**in** $start\text{-}context$) $semi: semilat\ (A, r, f)$

$\langle proof \rangle$

lemma (**in** $JVM\text{-}sl$) $step\text{-}def\text{-}exec: step \equiv exec\ P\ mxs\ T_r\ xt\ is$

$\langle proof \rangle$

lemma $special\text{-}ex\text{-}swap\text{-}lemma\ [iff]:$

$(? X. (? n. X = A\ n \ \& \ P\ n) \ \& \ Q\ X) = (? n. Q\ (A\ n) \ \& \ P\ n)$

$\langle proof \rangle$

lemma $ex\text{-}in\text{-}nlists\ [iff]:$

$(\exists n. ST \in nlists\ n\ A \wedge n \leq mxs) = (set\ ST \subseteq A \wedge size\ ST \leq mxs)$

$\langle proof \rangle$

lemma $singleton\text{-}nlists:$

$(\exists n. [Class\ C] \in nlists\ n\ (types\ P) \wedge n \leq mxs) = (is\text{-}class\ P\ C \wedge 0 < mxs)$

$\langle proof \rangle$

lemma *set-drop-subset*:

$set\ xs \subseteq A \implies set\ (drop\ n\ xs) \subseteq A$
 ⟨proof⟩

lemma *Suc-minus-minus-le*:

$n < mxs \implies Suc\ (n - (n - b)) \leq mxs$
 ⟨proof⟩

lemma *in-nlistsE*:

$\llbracket xs \in nlists\ n\ A; \llbracket size\ xs = n; set\ xs \subseteq A \rrbracket \implies P \rrbracket \implies P$
 ⟨proof⟩

declare *is-relevant-entry-def* [simp]

declare *set-drop-subset* [simp]

theorem (in *start-context*) *exec-pres-type*:

pres-type step (size is) A⟨proof⟩

declare *is-relevant-entry-def* [simp del]

declare *set-drop-subset* [simp del]

lemma *lesubstep-type-simple*:

$xs \sqsubseteq_{Product.le\ (=)\ r} ys \implies set\ xs \{\sqsubseteq_r\} set\ ys$ ⟨proof⟩

declare *is-relevant-entry-def* [simp del]

lemma *conjI2*: $\llbracket A; A \implies B \rrbracket \implies A \wedge B$ ⟨proof⟩

lemma (in *JVM-sl*) *eff-mono*:

$\llbracket wf\ prog\ p\ P; pc < length\ is; s \sqsubseteq_{sup\ state\ opt\ P} t; app\ pc\ t \rrbracket$
 $\implies set\ (eff\ pc\ s) \{\sqsubseteq_{sup\ state\ opt\ P}\} set\ (eff\ pc\ t)$ ⟨proof⟩

lemma (in *JVM-sl*) *bounded-step*: *bounded step (size is)*⟨proof⟩

theorem (in *JVM-sl*) *step-mono*:

$wf\ prog\ wf\ mb\ P \implies mono\ r\ step\ (size\ is)\ A$ ⟨proof⟩

lemma (in *start-context*) *first-in-A* [iff]: *OK first* $\in A$

⟨proof⟩

lemma (in *JVM-sl*) *wt-method-def2*:

wt-method $P\ C'\ Ts\ T_r\ mxs\ mxl_0\ is\ xt\ \tau s =$

$(is \neq [] \wedge$

$size\ \tau s = size\ is \wedge$

$OK\ 'set\ \tau s \subseteq states\ P\ mxs\ mxl \wedge$

$wt\ start\ P\ C'\ Ts\ mxl_0\ \tau s \wedge$

$wt\ app\ eff\ (sup\ state\ opt\ P)\ app\ eff\ \tau s)$ ⟨proof⟩

end

4.20 Typing and Dataflow Analysis Framework

theory *Typing-Framework-2* **imports** *Typing-Framework-1* **begin**

The relationship between dataflow analysis and a welltyped-instruction predicate.

definition $is-bcv :: 's\ ord \Rightarrow 's \Rightarrow 's\ step\ type \Rightarrow nat \Rightarrow 's\ set \Rightarrow ('s\ list \Rightarrow 's\ list) \Rightarrow bool$
where

$is-bcv\ r\ T\ step\ n\ A\ bcv \longleftrightarrow (\forall \tau s_0 \in nlists\ n\ A.$
 $(\forall p < n. (bcv\ \tau s_0)!p \neq T) = (\exists \tau s \in nlists\ n\ A. \tau s_0\ [\sqsubseteq_r]\ \tau s \wedge wt\ step\ r\ T\ step\ \tau s))$

end

4.21 Kildall for the JVM

theory $BVExec$

imports $../DFA/Abstract-BV\ TF-JVM\ ../DFA/Typing-Framework-2$

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 $wt-kildall :: jvm\ prog \Rightarrow cname \Rightarrow ty\ list \Rightarrow ty \Rightarrow nat \Rightarrow nat \Rightarrow$
 $instr\ list \Rightarrow ex\ table \Rightarrow bool$

where

$wt-kildall\ P\ C'\ Ts\ T_r\ mxs\ mxl_0\ is\ xt \equiv$
 $0 < size\ is \wedge$
 $(let\ first = Some\ ([],[OK\ (Class\ C')]\@(\map\ OK\ Ts)\@(replicate\ mxl_0\ Err));$
 $start = OK\ first\#(replicate\ (size\ is - 1)\ (OK\ None));$
 $result = kiljvm\ P\ mxs\ (1+size\ Ts+mxl_0)\ T_r\ is\ xt\ start$
 $in\ \forall n < size\ is. result!n \neq Err)$

definition $wf-jvm-prog_k :: jvm\ prog \Rightarrow bool$

where

$wf-jvm-prog_k\ P \equiv$
 $wf-prog\ (\lambda P\ C'\ (M,Ts,T_r,(mxs,mxl_0,is,xt)). wt-kildall\ P\ C'\ Ts\ T_r\ mxs\ mxl_0\ is\ xt)\ P$

context $start-context$

begin

lemma $Cons-less-Conss3$ $[simp]:$

$x\#xs\ [\sqsubseteq_r]\ y\#ys = (x\ \sqsubseteq_r\ y \wedge xs\ [\sqsubseteq_r]\ ys \vee x = y \wedge xs\ [\sqsubseteq_r]\ ys)$
 $\langle proof \rangle$

lemma $acc-le-listI3$ $[intro!]:$

$acc\ r \Longrightarrow acc\ (Listn.le\ r)$
 $\langle proof \rangle$

lemma $wf-jvm: wf\ \{(ss', ss). ss\ [\sqsubseteq_r]\ ss'\}$

$\langle proof \rangle$

lemma $iter-properties-bv$ $[rule-format]:$

shows $\llbracket \forall p \in w0. p < n; ss0 \in nlists\ n\ A; \forall p < n. p \notin w0 \longrightarrow stable\ r\ step\ ss0\ p \rrbracket \Longrightarrow$
 $iter\ f\ step\ ss0\ w0 = (ss', w') \longrightarrow$

$$ss' \in nlists\ n\ A \wedge stables\ r\ step\ ss' \wedge ss0\ [\sqsubseteq_r]\ ss' \wedge$$

$$(\forall ts \in nlists\ n\ A. ss0\ [\sqsubseteq_r]\ ts \wedge stables\ r\ step\ ts \longrightarrow ss' [\sqsubseteq_r]\ ts) \langle proof \rangle$$

lemma *kildall-properties-bv*:

shows $\llbracket ss0 \in nlists\ n\ A \rrbracket \implies$

$kildall\ r\ f\ step\ ss0 \in nlists\ n\ A \wedge$

$stables\ r\ step\ (kildall\ r\ f\ step\ ss0) \wedge$

$ss0\ [\sqsubseteq_r]\ kildall\ r\ f\ step\ ss0 \wedge$

$(\forall ts \in nlists\ n\ A. ss0\ [\sqsubseteq_r]\ ts \wedge stables\ r\ step\ ts \longrightarrow$

$kildall\ r\ f\ step\ ss0\ [\sqsubseteq_r]\ ts) \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle$

4.22 LBV for the JVM

theory *LBVJVM*

imports *../DFA/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

$check-cert\ P\ mxs\ mxl\ n\ cert \equiv check-types\ P\ mxs\ mxl\ cert \wedge size\ cert = n+1 \wedge$
 $(\forall i < n. cert!i \neq Err) \wedge cert!n = OK\ None$

definition *lbvjvm* :: *jvm-prog* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *ty* \Rightarrow *ex-table* \Rightarrow

ty_i' *err list* \Rightarrow *instr list* \Rightarrow *ty_i'* *err* \Rightarrow *ty_i'* *err*

where

$lbvjvm\ P\ mxs\ maxr\ T_r\ et\ cert\ bs \equiv$

$wtl-inst-list\ bs\ cert\ (JVM-SemiType.sup\ P\ mxs\ maxr)\ (JVM-SemiType.le\ P\ mxs\ maxr)\ Err\ (OK\ None)\ (exec\ P\ mxs\ T_r\ et\ bs)\ 0$

definition *wt-lbv* :: *jvm-prog* \Rightarrow *cname* \Rightarrow *ty list* \Rightarrow *ty* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow

ex-table \Rightarrow *ty_i'* *err list* \Rightarrow *instr list* \Rightarrow *bool*

where

$wt-lbv\ P\ C\ Ts\ T_r\ mxs\ mxl_0\ et\ cert\ ins \equiv$

$check-cert\ P\ mxs\ (1+size\ Ts+mxl_0)\ (size\ ins)\ cert \wedge$

$0 < size\ ins \wedge$

$(let\ start = Some\ ([],(OK\ (Class\ C))\#((map\ OK\ Ts))\@(replicate\ mxl_0\ Err));$

$result = lbvjvm\ P\ mxs\ (1+size\ Ts+mxl_0)\ T_r\ et\ cert\ ins\ (OK\ start)$

$in\ result \neq Err)$

definition *wt-jvm-prog-lbv* :: *jvm-prog* \Rightarrow *prog-cert* \Rightarrow *bool*

where

$wt-jvm-prog-lbv\ P\ cert \equiv$

$wf-prog\ (\lambda P\ C\ (mn, Ts, T_r, (mxs, mxl_0, b, et)). wt-lbv\ P\ C\ Ts\ T_r\ mxs\ mxl_0\ et\ (cert\ C\ mn)\ b)\ P$

definition *mk-cert* :: *jvm-prog* \Rightarrow *nat* \Rightarrow *ty* \Rightarrow *ex-table* \Rightarrow *instr list*

\Rightarrow *ty_m* \Rightarrow *ty_i'* *err list*

where

$mk-cert\ P\ mxs\ T_r\ et\ bs\ phi \equiv make-cert\ (exec\ P\ mxs\ T_r\ et\ bs)\ (map\ OK\ phi)\ (OK\ None)$

definition *prg-cert* :: *jvm-prog* \Rightarrow *ty_P* \Rightarrow *prog-cert*

where

$prg\text{-cert } P \text{ phi } C \text{ mn} \equiv \text{let } (C, Ts, T_r, (m\acute{x}s, m\acute{x}l_0, ins, et)) = \text{method } P \text{ } C \text{ mn}$
 $\text{in } mk\text{-cert } P \text{ } m\acute{x}s \text{ } T_r \text{ } et \text{ } ins \text{ } (phi \text{ } C \text{ mn})$

lemma *check-certD* [intro?]:

$check\text{-cert } P \text{ } m\acute{x}s \text{ } m\acute{x}l \text{ } n \text{ } cert \implies cert\text{-ok } cert \text{ } n \text{ } Err \text{ } (OK \text{ } None) \text{ } (states \text{ } P \text{ } m\acute{x}s \text{ } m\acute{x}l)$
 ⟨proof⟩

lemma (in *start-context*) *wt-lbv-wt-step*:

assumes $lbv: wt\text{-lbv } P \text{ } C \text{ } Ts \text{ } T_r \text{ } m\acute{x}s \text{ } m\acute{x}l_0 \text{ } xt \text{ } cert \text{ } is$
shows $\exists \tau s \in nlists \text{ } (size \text{ } is) \text{ } A. wt\text{-step } r \text{ } Err \text{ } step \text{ } \tau s \wedge OK \text{ } first \sqsubseteq_r \tau s!0$ ⟨proof⟩

lemma (in *start-context*) *wt-lbv-wt-method*:

assumes $lbv: wt\text{-lbv } P \text{ } C \text{ } Ts \text{ } T_r \text{ } m\acute{x}s \text{ } m\acute{x}l_0 \text{ } xt \text{ } cert \text{ } is$
shows $\exists \tau s. wt\text{-method } P \text{ } C \text{ } Ts \text{ } T_r \text{ } m\acute{x}s \text{ } m\acute{x}l_0 \text{ } is \text{ } xt \text{ } \tau s$ ⟨proof⟩

lemma (in *start-context*) *wt-method-wt-lbv*:

assumes $wt: wt\text{-method } P \text{ } C \text{ } Ts \text{ } T_r \text{ } m\acute{x}s \text{ } m\acute{x}l_0 \text{ } is \text{ } xt \text{ } \tau s$
defines [simp]: $cert \equiv mk\text{-cert } P \text{ } m\acute{x}s \text{ } T_r \text{ } xt \text{ } is \text{ } \tau s$

shows $wt\text{-lbv } P \text{ } C \text{ } Ts \text{ } T_r \text{ } m\acute{x}s \text{ } m\acute{x}l_0 \text{ } xt \text{ } cert \text{ } is$ ⟨proof⟩

theorem *jvm-lbv-correct*:

$wt\text{-jvm-prog-lbv } P \text{ } Cert \implies wf\text{-jvm-prog } P$ ⟨proof⟩

theorem *jvm-lbv-complete*:

assumes $wt: wf\text{-jvm-prog}_{\Phi} \text{ } P$
shows $wt\text{-jvm-prog-lbv } P \text{ } (prg\text{-cert } P \text{ } \Phi)$ ⟨proof⟩

end

4.23 BV Type Safety Invariant

theory *BVConform*

imports *BVSpec ../JVM/JVMExec ../Common/Conform*

begin

definition *confT* :: $'c \text{ prog} \Rightarrow heap \Rightarrow val \Rightarrow ty \text{ err} \Rightarrow bool$

$(-, - \vdash - : \leq_{\top} - [51, 51, 51, 51] \text{ } 50)$

where

$P, h \vdash v : \leq_{\top} E \equiv \text{case } E \text{ of } Err \Rightarrow True \mid OK \text{ } T \Rightarrow P, h \vdash v : \leq T$

notation (*ASCII*)

$confT \text{ } (-, - \mid - : \leq T - [51, 51, 51, 51] \text{ } 50)$

abbreviation

$confTs :: 'c \text{ prog} \Rightarrow heap \Rightarrow val \text{ list} \Rightarrow ty_l \Rightarrow bool$

$(-, - \vdash - [:\leq_{\top}] - [51, 51, 51, 51] \text{ } 50)$ **where**

$P, h \vdash vs [:\leq_{\top}] Ts \equiv list\text{-all2 } (confT \text{ } P \text{ } h) \text{ } vs \text{ } Ts$

notation (*ASCII*)

$confTs \text{ } (-, - \mid - [:\leq T] - [51, 51, 51, 51] \text{ } 50)$

definition *conf-f* :: $jvm\text{-prog} \Rightarrow heap \Rightarrow ty_i \Rightarrow bytecode \Rightarrow frame \Rightarrow bool$

where

$conf\text{-}f\ P\ h \equiv \lambda(ST,LT)\ is\ (stk,loc,C,M,pc).$
 $P, h \vdash stk\ [:\leq] ST \wedge P, h \vdash loc\ [:\leq_{\top}] LT \wedge pc < size\ is$

lemma $conf\text{-}f\text{-}def2$:

$conf\text{-}f\ P\ h\ (ST,LT)\ is\ (stk,loc,C,M,pc) \equiv$
 $P, h \vdash stk\ [:\leq] ST \wedge P, h \vdash loc\ [:\leq_{\top}] LT \wedge pc < size\ is$
 ⟨proof⟩

primrec $conf\text{-}fs :: [jvm\text{-}prog, heap, ty_P, mname, nat, ty, frame\ list] \Rightarrow bool$

where

$conf\text{-}fs\ P\ h\ \Phi\ M_0\ n_0\ T_0\ [] = True$
 $| conf\text{-}fs\ P\ h\ \Phi\ M_0\ n_0\ T_0\ (f\#\text{frs}) =$
 (let $(stk,loc,C,M,pc) = f$ in
 $(\exists ST\ LT\ Ts\ T\ mxs\ mxl_0\ is\ xt.$
 $\Phi\ C\ M\ !\ pc = Some\ (ST,LT) \wedge$
 $(P \vdash C\ sees\ M:Ts \rightarrow T = (mxs, mxl_0, is, xt)\ in\ C) \wedge$
 $(\exists D\ Ts'\ T'\ m\ D'.$
 $is!pc = (Invoke\ M_0\ n_0) \wedge ST!n_0 = Class\ D \wedge$
 $P \vdash D\ sees\ M_0:Ts' \rightarrow T' = m\ in\ D' \wedge P \vdash T_0 \leq T') \wedge$
 $conf\text{-}f\ P\ h\ (ST, LT)\ is\ f \wedge conf\text{-}fs\ P\ h\ \Phi\ M\ (size\ Ts)\ T\ frs))$

definition $correct\text{-}state :: [jvm\text{-}prog, ty_P, jvm\text{-}state] \Rightarrow bool\ (-, - \vdash - \sqrt{[61,0,0]}\ 61)$

where

$correct\text{-}state\ P\ \Phi \equiv \lambda(xp, h, frs).$
 $case\ xp\ of$
 $None \Rightarrow (case\ frs\ of$
 $\quad [] \Rightarrow True$
 $\quad | (f\#\text{fs}) \Rightarrow P \vdash h\sqrt{\wedge}$
 $\quad (let\ (stk,loc,C,M,pc) = f$
 $\quad in\ \exists\ Ts\ T\ mxs\ mxl_0\ is\ xt\ \tau.$
 $\quad (P \vdash C\ sees\ M:Ts \rightarrow T = (mxs, mxl_0, is, xt)\ in\ C) \wedge$
 $\quad \Phi\ C\ M\ !\ pc = Some\ \tau \wedge$
 $\quad conf\text{-}f\ P\ h\ \tau\ is\ f \wedge conf\text{-}fs\ P\ h\ \Phi\ M\ (size\ Ts)\ T\ fs))$
 $| Some\ x \Rightarrow frs = []$

notation

$correct\text{-}state\ (-, - \vdash - [ok]\ [61,0,0]\ 61)$

4.23.1 Values and \top

lemma $confT\text{-}Err$ [iff]: $P, h \vdash x : \leq_{\top} Err$
 ⟨proof⟩

lemma $confT\text{-}OK$ [iff]: $P, h \vdash x : \leq_{\top} OK\ T = (P, h \vdash x : \leq T)$
 ⟨proof⟩

lemma $confT\text{-}cases$:

$P, h \vdash x : \leq_{\top} X = (X = Err \vee (\exists T. X = OK\ T \wedge P, h \vdash x : \leq T))$
 ⟨proof⟩

lemma $confT\text{-}hext$ [intro?, trans]:

$\llbracket P, h \vdash x : \leq_{\top} T; h \sqsubseteq h' \rrbracket \Longrightarrow P, h' \vdash x : \leq_{\top} T$
<proof>

lemma *confT-widen* [*intro?*, *trans*]:
 $\llbracket P, h \vdash x : \leq_{\top} T; P \vdash T \leq_{\top} T' \rrbracket \Longrightarrow P, h \vdash x : \leq_{\top} T'$
<proof>

4.23.2 Stack and Registers

lemmas *confTs-Cons1* [*iff*] = *list-all2-Cons1* [*of confT P h*] **for** *P h*

lemma *confTs-confT-sup*:
 $\llbracket P, h \vdash \text{loc} [\leq_{\top}] LT; n < \text{size } LT; LT!n = \text{OK } T; P \vdash T \leq T' \rrbracket$
 $\Longrightarrow P, h \vdash (\text{loc}!n) : \leq T'$ *<proof>*

lemma *confTs-heat* [*intro?*]:
 $P, h \vdash \text{loc} [\leq_{\top}] LT \Longrightarrow h \sqsubseteq h' \Longrightarrow P, h' \vdash \text{loc} [\leq_{\top}] LT$
<proof>

lemma *confTs-widen* [*intro?*, *trans*]:
 $P, h \vdash \text{loc} [\leq_{\top}] LT \Longrightarrow P \vdash LT [\leq_{\top}] LT' \Longrightarrow P, h \vdash \text{loc} [\leq_{\top}] LT'$
<proof>

lemma *confTs-map* [*iff*]:
 $\bigwedge \text{vs}. (P, h \vdash \text{vs} [\leq_{\top}] \text{map } \text{OK } Ts) = (P, h \vdash \text{vs} [\leq] Ts)$
<proof>

lemma *reg-widen-Err* [*iff*]:
 $\bigwedge LT. (P \vdash \text{replicate } n \text{ Err} [\leq_{\top}] LT) = (LT = \text{replicate } n \text{ Err})$
<proof>

lemma *confTs-Err* [*iff*]:
 $P, h \vdash \text{replicate } n v [\leq_{\top}] \text{replicate } n \text{ Err}$
<proof>

4.23.3 correct-frames

lemmas [*simp del*] = *fun-upd-apply*

lemma *conf-fs-heat*:
 $\bigwedge M n T_r.$
 $\llbracket \text{conf-fs } P h \Phi M n T_r \text{ frs}; h \sqsubseteq h' \rrbracket \Longrightarrow \text{conf-fs } P h' \Phi M n T_r \text{ frs}$ *<proof>*
end

4.24 BV Type Safety Proof

theory *BVSpecTypeSafe*
imports *BVConform*
begin

This theory contains proof that the specification of the bytecode verifier only admits type safe programs.

4.24.1 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*

4.24.2 Exception Handling

For the *Invoke* instruction the BV has checked all handlers that guard the current *pc*.

lemma *Invoke-handlers*:

match-ex-table $P C pc xt = \text{Some } (pc', d') \implies$
 $\exists (f, t, D, h, d) \in \text{set } (\text{relevant-entries } P (\text{Invoke } n M) pc xt).$
 $P \vdash C \preceq^* D \wedge pc \in \{f..<t\} \wedge pc' = h \wedge d' = d$
 ⟨*proof*⟩

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.

term *find-handler*

lemma *uncaught-xcpt-correct*:

assumes *wt*: *wf-jvm-prog* Φ P
assumes *h*: $h xcp = \text{Some } obj$
shows $\bigwedge f. P, \Phi \vdash (\text{None}, h, f \# \text{frs}) \checkmark \implies P, \Phi \vdash (\text{find-handler } P xcp h \text{frs}) \checkmark$
(is $\bigwedge f. ?\text{correct } (\text{None}, h, f \# \text{frs}) \implies ?\text{correct } (?find \text{frs})$ **)**⟨*proof*⟩

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*:

$\llbracket \text{fst } (\text{exec-instr } (ins!pc) P h \text{stk vars } Cl M pc \text{frs}) = \text{Some } xcp;$
 $P, T, mxs, size \text{ ins}, xt \vdash ins!pc, pc :: \Phi C M;$
 $P, \Phi \vdash (\text{None}, h, (\text{stk}, loc, C, M, pc) \# \text{frs}) \checkmark \rrbracket$
 $\implies \exists obj. h xcp = \text{Some } obj$
(is $\llbracket ?xcpt; ?wt; ?correct \rrbracket \implies ?thesis$ **)**⟨*proof*⟩

lemma *conf-sys-xcpt*:

$\llbracket \text{preallocated } h; C \in \text{sys-xcpts} \rrbracket \implies P, h \vdash \text{Addr } (\text{addr-of-sys-xcpt } C) : \leq \text{Class } C$
 ⟨*proof*⟩

lemma *match-ex-table-SomeD*:

match-ex-table $P C pc xt = \text{Some } (pc', d') \implies$
 $\exists (f, t, D, h, d) \in \text{set } xt. \text{matches-ex-entry } P C pc (f, t, D, h, d) \wedge h = pc' \wedge d = d'$
 ⟨*proof*⟩

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* Φ P
assumes *meth*: $P \vdash C \text{ sees } M: Ts \rightarrow T = (mxs, mxl_0, ins, xt) \text{ in } C$
assumes *wt*: $P, T, mxs, size \text{ ins}, xt \vdash ins!pc, pc :: \Phi C M$
assumes *xp*: $\text{fst } (\text{exec-instr } (ins!pc) P h \text{stk loc } C M pc \text{frs}) = \text{Some } xcp$
assumes *s'*: $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (\text{stk}, loc, C, M, pc) \# \text{frs})$
assumes *correct*: $P, \Phi \vdash (\text{None}, h, (\text{stk}, loc, C, M, pc) \# \text{frs}) \checkmark$
shows $P, \Phi \vdash \sigma' \checkmark$ ⟨*proof*⟩

4.24.3 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.

declare *defs1* [*simp*]

lemma *Invoke-correct*:

fixes $\sigma' :: \text{jvm-state}$

assumes *wtprog*: $\text{wf-jvm-prog}_{\Phi} P$

assumes *meth-C*: $P \vdash C \text{ sees } M:Ts \rightarrow T=(\text{m}\lambda s, \text{m}\lambda l_0, \text{ins}, \text{xt}) \text{ in } C$

assumes *ins*: $\text{ins} ! \text{pc} = \text{Invoke } M' n$

assumes *wti*: $P, T, \text{m}\lambda s, \text{size } \text{ins}, \text{xt} \vdash \text{ins} ! \text{pc}, \text{pc} :: \Phi C M$

assumes σ' : $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs})$

assumes *approx*: $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark$

assumes *no-xcp*: $\text{fst } (\text{exec-instr } (\text{ins} ! \text{pc}) P h \text{ stk loc } C M \text{ pc frs}) = \text{None}$

shows $P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle$

declare *list-all2-Cons2* [*iff*]

lemma *Return-correct*:

fixes $\sigma' :: \text{jvm-state}$

assumes *wt-prog*: $\text{wf-jvm-prog}_{\Phi} P$

assumes *meth*: $P \vdash C \text{ sees } M:Ts \rightarrow T=(\text{m}\lambda s, \text{m}\lambda l_0, \text{ins}, \text{xt}) \text{ in } C$

assumes *ins*: $\text{ins} ! \text{pc} = \text{Return}$

assumes *wt*: $P, T, \text{m}\lambda s, \text{size } \text{ins}, \text{xt} \vdash \text{ins} ! \text{pc}, \text{pc} :: \Phi C M$

assumes s' : $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs})$

assumes *correct*: $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark$

shows $P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle$

declare *sup-state-opt-any-Some* [*iff*]

declare *not-Err-eq* [*iff*]

lemma *Load-correct*:

$\llbracket \text{wf-prog } \text{wt } P;$

$P \vdash C \text{ sees } M:Ts \rightarrow T=(\text{m}\lambda s, \text{m}\lambda l_0, \text{ins}, \text{xt}) \text{ in } C;$

$\text{ins} ! \text{pc} = \text{Load } \text{idx};$

$P, T, \text{m}\lambda s, \text{size } \text{ins}, \text{xt} \vdash \text{ins} ! \text{pc}, \text{pc} :: \Phi C M;$

$\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs});$

$P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark \rrbracket$

$\implies P, \Phi \vdash \sigma' \checkmark$

$\langle \text{proof} \rangle$

declare $\llbracket \text{simproc del: list-to-set-comprehension} \rrbracket$

lemma *Store-correct*:

$\llbracket \text{wf-prog } \text{wt } P;$

$P \vdash C \text{ sees } M:Ts \rightarrow T=(\text{m}\lambda s, \text{m}\lambda l_0, \text{ins}, \text{xt}) \text{ in } C;$

$\text{ins} ! \text{pc} = \text{Store } \text{idx};$

$P, T, \text{m}\lambda s, \text{size } \text{ins}, \text{xt} \vdash \text{ins} ! \text{pc}, \text{pc} :: \Phi C M;$

$\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs});$

$P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark \rrbracket$

$\implies P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle$

lemma *Push-correct*:

\llbracket *wf-prog wt P*;
 $P \vdash C \text{ sees } M:Ts \rightarrow T=(m\!x\!s, m\!x\!l_0, ins, xt) \text{ in } C$;
 $ins!pc = \text{Push } v$;
 $P, T, m\!x\!s, size \ ins, xt \vdash ins!pc, pc :: \Phi \ C \ M$;
 $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc)\#frs)$;
 $P, \Phi \vdash (\text{None}, h, (stk, loc, C, M, pc)\#frs)\checkmark$ \rrbracket
 $\implies P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle$

lemma *Cast-conf2*:

\llbracket *wf-prog ok P*; $P, h \vdash v : \leq T$; *is-refT T*; *cast-ok P C h v*;
 $P \vdash \text{Class } C \leq T'$; *is-class P C* \rrbracket
 $\implies P, h \vdash v : \leq T' \langle \text{proof} \rangle$

lemma *Checkcast-correct*:

\llbracket *wf-jvm-prog Φ P*;
 $P \vdash C \text{ sees } M:Ts \rightarrow T=(m\!x\!s, m\!x\!l_0, ins, xt) \text{ in } C$;
 $ins!pc = \text{Checkcast } D$;
 $P, T, m\!x\!s, size \ ins, xt \vdash ins!pc, pc :: \Phi \ C \ M$;
 $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc)\#frs)$;
 $P, \Phi \vdash (\text{None}, h, (stk, loc, C, M, pc)\#frs)\checkmark$;
 $\text{fst } (\text{exec-instr } (ins!pc) \ P \ h \ stk \ loc \ C \ M \ pc \ frs) = \text{None}$ \rrbracket
 $\implies P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle$

declare *split-paired-All* [*simp del*]

lemmas *widens-Cons* [*iff*] = *list-all2-Cons1* [*of widen P*] **for** *P*

lemma *Getfield-correct*:

fixes $\sigma' :: \text{jvm-state}$
assumes *wf*: *wf-prog wt P*
assumes *mC*: $P \vdash C \text{ sees } M:Ts \rightarrow T=(m\!x\!s, m\!x\!l_0, ins, xt) \text{ in } C$
assumes *i*: $ins!pc = \text{Getfield } F \ D$
assumes *wt*: $P, T, m\!x\!s, size \ ins, xt \vdash ins!pc, pc :: \Phi \ C \ M$
assumes *s'*: $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc)\#frs)$
assumes *cf*: $P, \Phi \vdash (\text{None}, h, (stk, loc, C, M, pc)\#frs)\checkmark$
assumes *xc*: $\text{fst } (\text{exec-instr } (ins!pc) \ P \ h \ stk \ loc \ C \ M \ pc \ frs) = \text{None}$

shows $P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle$

lemma *Putfield-correct*:

fixes $\sigma' :: \text{jvm-state}$
assumes *wf*: *wf-prog wt P*
assumes *mC*: $P \vdash C \text{ sees } M:Ts \rightarrow T=(m\!x\!s, m\!x\!l_0, ins, xt) \text{ in } C$
assumes *i*: $ins!pc = \text{Putfield } F \ D$
assumes *wt*: $P, T, m\!x\!s, size \ ins, xt \vdash ins!pc, pc :: \Phi \ C \ M$
assumes *s'*: $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc)\#frs)$
assumes *cf*: $P, \Phi \vdash (\text{None}, h, (stk, loc, C, M, pc)\#frs)\checkmark$
assumes *xc*: $\text{fst } (\text{exec-instr } (ins!pc) \ P \ h \ stk \ loc \ C \ M \ pc \ frs) = \text{None}$

shows $P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle$

lemma *has-fields-b-fields*:

$P \vdash C \text{ has-fields } FDTs \implies \text{fields } P \ C = FDTs \langle \text{proof} \rangle$

lemma *oconf-blank* [*intro, simp*]:

$\llbracket is-class \ P \ C; wf-prog \ wt \ P \rrbracket \implies P, h \vdash \text{blank } P \ C \checkmark \langle \text{proof} \rangle$

lemma *obj-ty-blank* [iff]: *obj-ty* (*blank P C*) = *Class C*
 ⟨*proof*⟩

lemma *New-correct*:

fixes $\sigma' :: \text{jvm-state}$

assumes *wf*: *wf-prog wt P*

assumes *meth*: $P \vdash C \text{ sees } M:Ts \rightarrow T=(m\text{xs},m\text{x}l_0,ins,xt) \text{ in } C$

assumes *ins*: $ins!pc = \text{New } X$

assumes *wt*: $P, T, m\text{xs}, \text{size } ins, xt \vdash ins!pc, pc :: \Phi C M$

assumes *exec*: $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc) \# frs)$

assumes *conf*: $P, \Phi \vdash (\text{None}, h, (stk, loc, C, M, pc) \# frs) \checkmark$

assumes *no-x*: $\text{fst } (\text{exec-instr } (ins!pc) P h stk loc C M pc frs) = \text{None}$

shows $P, \Phi \vdash \sigma' \checkmark$ ⟨*proof*⟩

lemma *Goto-correct*:

[[*wf-prog wt P*;

$P \vdash C \text{ sees } M:Ts \rightarrow T=(m\text{xs},m\text{x}l_0,ins,xt) \text{ in } C$;

$ins!pc = \text{Goto branch}$;

$P, T, m\text{xs}, \text{size } ins, xt \vdash ins!pc, pc :: \Phi C M$;

$\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc) \# frs)$;

$P, \Phi \vdash (\text{None}, h, (stk, loc, C, M, pc) \# frs) \checkmark$]]

$\Rightarrow P, \Phi \vdash \sigma' \checkmark$ ⟨*proof*⟩

lemma *IfFalse-correct*:

[[*wf-prog wt P*;

$P \vdash C \text{ sees } M:Ts \rightarrow T=(m\text{xs},m\text{x}l_0,ins,xt) \text{ in } C$;

$ins!pc = \text{IfFalse branch}$;

$P, T, m\text{xs}, \text{size } ins, xt \vdash ins!pc, pc :: \Phi C M$;

$\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc) \# frs)$;

$P, \Phi \vdash (\text{None}, h, (stk, loc, C, M, pc) \# frs) \checkmark$]]

$\Rightarrow P, \Phi \vdash \sigma' \checkmark$ ⟨*proof*⟩

lemma *CmpEq-correct*:

[[*wf-prog wt P*;

$P \vdash C \text{ sees } M:Ts \rightarrow T=(m\text{xs},m\text{x}l_0,ins,xt) \text{ in } C$;

$ins!pc = \text{CmpEq}$;

$P, T, m\text{xs}, \text{size } ins, xt \vdash ins!pc, pc :: \Phi C M$;

$\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc) \# frs)$;

$P, \Phi \vdash (\text{None}, h, (stk, loc, C, M, pc) \# frs) \checkmark$]]

$\Rightarrow P, \Phi \vdash \sigma' \checkmark$ ⟨*proof*⟩

lemma *Pop-correct*:

[[*wf-prog wt P*;

$P \vdash C \text{ sees } M:Ts \rightarrow T=(m\text{xs},m\text{x}l_0,ins,xt) \text{ in } C$;

$ins!pc = \text{Pop}$;

$P, T, m\text{xs}, \text{size } ins, xt \vdash ins!pc, pc :: \Phi C M$;

$\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc) \# frs)$;

$P, \Phi \vdash (\text{None}, h, (stk, loc, C, M, pc) \# frs) \checkmark$]]

$\Rightarrow P, \Phi \vdash \sigma' \checkmark$ ⟨*proof*⟩

lemma *IAdd-correct*:

[[*wf-prog wt P*;

$P \vdash C \text{ sees } M:Ts \rightarrow T=(m\text{xs},m\text{x}l_0,ins,xt) \text{ in } C$;

$ins!pc = \text{IAdd}$;

$P, T, m\text{xs}, \text{size } ins, xt \vdash ins!pc, pc :: \Phi C M$;

$\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc) \# frs)$;

$$P, \Phi \vdash (None, h, (stk, loc, C, M, pc) \# frs) \checkmark] \\ \implies P, \Phi \vdash \sigma' \checkmark \langle proof \rangle$$

lemma *Throw-correct:*

$$\llbracket wf\text{-prog } wt \ P; \\ P \vdash C \text{ sees } M: Ts \rightarrow T = (m\ xs, m\ xl_0, ins, xt) \text{ in } C; \\ ins \ ! \ pc = Throw; \\ \text{Some } \sigma' = exec \ (P, None, h, (stk, loc, C, M, pc) \# frs) \ ; \\ P, \Phi \vdash (None, h, (stk, loc, C, M, pc) \# frs) \checkmark; \\ fst \ (exec\text{-instr} \ (ins!pc) \ P \ h \ stk \ loc \ C \ M \ pc \ frs) = None \] \\ \implies P, \Phi \vdash \sigma' \checkmark \\ \langle proof \rangle$$

The next theorem collects the results of the sections above, i.e. exception handling and the execution step for each instruction. It states type safety for single step execution: in welltyped programs, a conforming state is transformed into another conforming state when one instruction is executed.

theorem *instr-correct:*

$$\llbracket wf\text{-jvm-prog}_{\Phi} \ P; \\ P \vdash C \text{ sees } M: Ts \rightarrow T = (m\ xs, m\ xl_0, ins, xt) \text{ in } C; \\ \text{Some } \sigma' = exec \ (P, None, h, (stk, loc, C, M, pc) \# frs); \\ P, \Phi \vdash (None, h, (stk, loc, C, M, pc) \# frs) \checkmark \] \\ \implies P, \Phi \vdash \sigma' \checkmark \langle proof \rangle$$

4.24.4 Main

lemma *correct-state-impl-Some-method:*

$$P, \Phi \vdash (None, h, (stk, loc, C, M, pc) \# frs) \checkmark \\ \implies \exists m \ Ts \ T. P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } C \\ \langle proof \rangle$$

lemma *BV-correct-1 [rule-format]:*

$$\Lambda \sigma. \llbracket wf\text{-jvm-prog}_{\Phi} \ P; P, \Phi \vdash \sigma \checkmark \] \implies P \vdash \sigma \text{-jvm} \rightarrow_1 \sigma' \longrightarrow P, \Phi \vdash \sigma' \checkmark \langle proof \rangle$$

theorem *progress:*

$$\llbracket xp = None; frs \neq [] \] \implies \exists \sigma'. P \vdash (xp, h, frs) \text{-jvm} \rightarrow_1 \sigma' \\ \langle proof \rangle$$

lemma *progress-conform:*

$$\llbracket wf\text{-jvm-prog}_{\Phi} \ P; P, \Phi \vdash (xp, h, frs) \checkmark; xp = None; frs \neq [] \] \\ \implies \exists \sigma'. P \vdash (xp, h, frs) \text{-jvm} \rightarrow_1 \sigma' \wedge P, \Phi \vdash \sigma' \checkmark \langle proof \rangle$$

theorem *BV-correct [rule-format]:*

$$\llbracket wf\text{-jvm-prog}_{\Phi} \ P; P \vdash \sigma \text{-jvm} \rightarrow \sigma' \] \implies P, \Phi \vdash \sigma \checkmark \longrightarrow P, \Phi \vdash \sigma' \checkmark \langle proof \rangle$$

lemma *hconf-start:*

assumes *wf:* $wf\text{-prog } wf\text{-mb } P$
shows $P \vdash (start\text{-heap } P) \checkmark \langle proof \rangle$

lemma *BV-correct-initial:*

shows $\llbracket wf\text{-jvm-prog}_{\Phi} \ P; P \vdash C \text{ sees } M: [] \rightarrow T = m \text{ in } C \] \\ \implies P, \Phi \vdash start\text{-state } P \ C \ M \checkmark \langle proof \rangle$

theorem *typesafe:*

assumes *welltyped:* $wf\text{-jvm-prog}_{\Phi} \ P$
assumes *main-method:* $P \vdash C \text{ sees } M: [] \rightarrow T = m \text{ in } C$
shows $P \vdash start\text{-state } P \ C \ M \text{-jvm} \rightarrow \sigma \implies P, \Phi \vdash \sigma \checkmark \langle proof \rangle$

end

4.25 Welltyped Programs produce no Type Errors

```

theory BVNoTypeError
imports ../JVM/JVMDefensive BVSpecTypeSafe
begin

```

```

lemma has-methodI:
   $P \vdash C \text{ sees } M:Ts \rightarrow T = m \text{ in } D \implies P \vdash C \text{ has } M$ 
   $\langle \text{proof} \rangle$ 

```

Some simple lemmas about the type testing functions of the defensive JVM:

```

lemma typeof-NoneD [simp,dest]:  $\text{typeof } v = \text{Some } x \implies \neg \text{is-Addr } v$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma is-Ref-def2:
   $\text{is-Ref } v = (v = \text{Null} \vee (\exists a. v = \text{Addr } a))$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma [iff]:  $\text{is-Ref } \text{Null} \langle \text{proof} \rangle$ 

```

```

lemma is-RefI [intro, simp]:  $P, h \vdash v : \leq T \implies \text{is-refT } T \implies \text{is-Ref } v \langle \text{proof} \rangle$ 

```

```

lemma is-IntgI [intro, simp]:  $P, h \vdash v : \leq \text{Integer} \implies \text{is-Intg } v \langle \text{proof} \rangle$ 

```

```

lemma is-BoolI [intro, simp]:  $P, h \vdash v : \leq \text{Boolean} \implies \text{is-Bool } v \langle \text{proof} \rangle$ 

```

```

declare defs1 [simp del]

```

```

lemma wf-jvm-prog-states:
   $\llbracket \text{wf-jvm-prog}_{\Phi} P; P \vdash C \text{ sees } M: Ts \rightarrow T = (m\text{xs}, m\text{x}l, \text{ins}, \text{et}) \text{ in } C;$ 
   $\Phi C M ! pc = \tau; pc < \text{size ins} \rrbracket$ 
   $\implies \text{OK } \tau \in \text{states } P \text{ m\text{x}s } (1 + \text{size } Ts + m\text{x}l) \langle \text{proof} \rangle$ 

```

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  $\text{welltyped}: \text{wf-jvm-prog}_{\Phi} P$  and  $\text{conforms}: P, \Phi \vdash \sigma \checkmark$ 
  shows  $\text{exec-d } P \sigma \neq \text{TypeError} \langle \text{proof} \rangle$ 

```

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 welltyped-aggressive-imp-defensive:
   $\text{wf-jvm-prog}_{\Phi} P \implies P, \Phi \vdash \sigma \checkmark \implies P \vdash \sigma \text{-jvm} \rightarrow \sigma'$ 
   $\implies P \vdash (\text{Normal } \sigma) \text{-jvmd} \rightarrow (\text{Normal } \sigma') \langle \text{proof} \rangle$ 

```

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 welltyped-commutes:
  fixes  $\sigma :: \text{jvm-state}$ 
  assumes  $\text{wf}: \text{wf-jvm-prog}_{\Phi} P$  and  $\text{conforms}: P, \Phi \vdash \sigma \checkmark$ 
  shows  $P \vdash (\text{Normal } \sigma) \text{-jvmd} \rightarrow (\text{Normal } \sigma') = P \vdash \sigma \text{-jvm} \rightarrow \sigma'$ 
   $\langle \text{proof} \rangle$ 

```

```

corollary welltyped-initial-commutes:
  assumes  $\text{wf}: \text{wf-jvm-prog } P$ 
  assumes  $\text{meth}: P \vdash C \text{ sees } M: [] \rightarrow T = b \text{ in } C$ 

```

```

defines start:  $\sigma \equiv \text{start-state } P \ C \ M$ 
shows  $P \vdash (\text{Normal } \sigma) \text{ -jvmd} \rightarrow (\text{Normal } \sigma') = P \vdash \sigma \text{ -jvm} \rightarrow \sigma'$ 
<proof>

```

```

lemma not-TypeError-eq [iff]:
 $x \neq \text{TypeError} = (\exists t. x = \text{Normal } t)$ 
<proof>

```

```

locale cnf =
fixes  $P$  and  $\Phi$  and  $\sigma$ 
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>

```

```

locale start =
fixes  $P$  and  $C$  and  $M$  and  $\sigma$  and  $T$  and  $b$ 
assumes wf: wf-jvm-prog  $P$ 
assumes sees:  $P \vdash C \text{ sees } M:[] \rightarrow T = b \text{ in } C$ 
defines  $\sigma \equiv \text{Normal } (\text{start-state } P \ C \ M)$ 

```

```

corollary (in start) bv-no-type-error:
shows  $P \vdash \sigma \text{ -jvmd} \rightarrow \sigma' \implies \sigma' \neq \text{TypeError}$ 
<proof>

```

```

end

```

4.26 Example Welltypings

```

theory BVExample
imports ../JVM/JVMListExample BVSpecTypeSafe BVExec
  HOL-Library.Code-Target-Numeral
begin

```

This theory shows type correctness of the example program in section 3.7 (p. 70) by explicitly providing a welltyping. It also shows that the start state of the program conforms to the welltyping; hence type safe execution is guaranteed.

4.26.1 Setup

```

lemma distinct-classes':
 $\text{list-name} \neq \text{test-name}$ 
 $\text{list-name} \neq \text{Object}$ 
 $\text{list-name} \neq \text{ClassCast}$ 
 $\text{list-name} \neq \text{OutOfMemory}$ 
 $\text{list-name} \neq \text{NullPointer}$ 
 $\text{test-name} \neq \text{Object}$ 
 $\text{test-name} \neq \text{OutOfMemory}$ 
 $\text{test-name} \neq \text{ClassCast}$ 

```

test-name \neq *NullPointer*
ClassCast \neq *NullPointer*
ClassCast \neq *Object*
NullPointer \neq *Object*
OutOfMemory \neq *ClassCast*
OutOfMemory \neq *NullPointer*
OutOfMemory \neq *Object*
 ⟨*proof*⟩

lemmas *distinct-classes* = *distinct-classes'* *distinct-classes'* [*symmetric*]

lemma *distinct-fields*:
val-name \neq *next-name*
next-name \neq *val-name*
 ⟨*proof*⟩

Abbreviations for definitions we will have to use often in the proofs below:

lemmas *system-defs* = *SystemClasses-def* *ObjectC-def* *NullPointerC-def*
OutOfMemoryC-def *ClassCastC-def*

lemmas *class-defs* = *list-class-def* *test-class-def*

These auxiliary proofs are for efficiency: class lookup, subclass relation, method and field lookup are computed only once:

lemma *class-Object* [*simp*]:
class E Object = *Some* (*undefined*, [], [])
 ⟨*proof*⟩

lemma *class-NullPointer* [*simp*]:
class E NullPointer = *Some* (*Object*, [], [])
 ⟨*proof*⟩

lemma *class-OutOfMemory* [*simp*]:
class E OutOfMemory = *Some* (*Object*, [], [])
 ⟨*proof*⟩

lemma *class-ClassCast* [*simp*]:
class E ClassCast = *Some* (*Object*, [], [])
 ⟨*proof*⟩

lemma *class-list* [*simp*]:
class E list-name = *Some list-class*
 ⟨*proof*⟩

lemma *class-test* [*simp*]:
class E test-name = *Some test-class*
 ⟨*proof*⟩

lemma *E-classes* [*simp*]:
 {*C. is-class E C*} = {*list-name*, *test-name*, *NullPointer*,
ClassCast, *OutOfMemory*, *Object*}
 ⟨*proof*⟩

The subclass relation spelled out:

lemma *subcls1*:

$subcls1\ E = \{(list-name, Object), (test-name, Object), (NullPointer, Object), (ClassCast, Object), (OutOfMemory, Object)\}$ $\langle proof \rangle$

The subclass relation is acyclic; hence its converse is well founded:

lemma *notin-rtrancl*:

$(a,b) \in r^* \implies a \neq b \implies (\bigwedge y. (a,y) \notin r) \implies False$
 $\langle proof \rangle$

lemma *acyclic-subcls1-E*: *acyclic* ($subcls1\ E$) $\langle proof \rangle$

lemma *wf-subcls1-E*: *wf* ($(subcls1\ E)^{-1}$) $\langle proof \rangle$

Method and field lookup:

lemma *method-append* [*simp*]:

method $E\ list-name\ append-name = (list-name, [Class\ list-name], Void, 3, 0, append-ins, [(1, 2, NullPointer, 7, 0)])$ $\langle proof \rangle$

lemma *method-makelist* [*simp*]:

method $E\ test-name\ makelist-name = (test-name, [], Void, 3, 2, make-list-ins, [])$ $\langle proof \rangle$

lemma *field-val* [*simp*]:

field $E\ list-name\ val-name = (list-name, Integer)$ $\langle proof \rangle$

lemma *field-next* [*simp*]:

field $E\ list-name\ next-name = (list-name, Class\ list-name)$ $\langle proof \rangle$

lemma [*simp*]: *fields* $E\ Object = []$

$\langle proof \rangle$

lemma [*simp*]: *fields* $E\ NullPointer = []$

$\langle proof \rangle$

lemma [*simp*]: *fields* $E\ ClassCast = []$

$\langle proof \rangle$

lemma [*simp*]: *fields* $E\ OutOfMemory = []$

$\langle proof \rangle$

lemma [*simp*]: *fields* $E\ test-name = []$ $\langle proof \rangle$

lemmas [*simp*] = *is-class-def*

4.26.2 Program structure

The program is structurally wellformed:

lemma *wf-struct*:

wf-prog $(\lambda G\ C\ mb.\ True)\ E\ (is\ wf-prog\ ?mb\ E)$ $\langle proof \rangle$

4.26.3 Welltypings

We show welltypings of the methods *append-name* in class *list-name*, and *makelist-name* in class *test-name*:

lemmas *eff-simps* [*simp*] = *eff-def norm-eff-def xcpt-eff-def*

definition *phi-append* :: $ty_m\ (\varphi_a)$

where

$\varphi_a \equiv map\ (\lambda(x,y).\ Some\ (x,\ map\ OK\ y))\ [$

```

(
  (
    [], [Class list-name, Class list-name]),
  (
    [Class list-name], [Class list-name, Class list-name]),
  (
    [Class list-name], [Class list-name, Class list-name]),
  (
    [Class list-name, Class list-name], [Class list-name, Class list-name]),
  (
    [Class list-name, Class list-name], [Class list-name, Class list-name]),
  ([NT, Class list-name, Class list-name], [Class list-name, Class list-name]),
  (
    [Boolean, Class list-name], [Class list-name, Class list-name]),

  (
    [Class Object], [Class list-name, Class list-name]),
  (
    [], [Class list-name, Class list-name]),
  (
    [Class list-name], [Class list-name, Class list-name]),
  (
    [Class list-name, Class list-name], [Class list-name, Class list-name]),
  (
    [], [Class list-name, Class list-name]),
  (
    [Void], [Class list-name, Class list-name]),

  (
    [Class list-name], [Class list-name, Class list-name]),
  (
    [Class list-name, Class list-name], [Class list-name, Class list-name]),
  (
    [Void], [Class list-name, Class list-name])

```

The next definition and three proof rules implement an algorithm to enumerate natural numbers. The command *apply* (*elim pc-end pc-next pc-0* transforms a goal of the form

$$pc < n \implies P \text{ pc}$$

into a series of goals

$$P (0::'a)$$

$$P (\text{Suc } 0)$$

...

$$P n$$

definition *intervall* :: *nat* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *bool* ($- \in [-, -']$)

where

$$x \in [a, b] \equiv a \leq x \wedge x < b$$

lemma *pc-0*: $x < n \implies (x \in [0, n] \implies P x) \implies P x$

<proof>

lemma *pc-next*: $x \in [n0, n] \implies P n0 \implies (x \in [\text{Suc } n0, n] \implies P x) \implies P x$ *<proof>*

lemma *pc-end*: $x \in [n, n] \implies P x$

<proof>

lemma *types-append* [*simp*]: *check-types E 3 (Suc (Suc 0)) (map OK φ_a)**<proof>*

lemma *wt-append* [*simp*]:

wt-method E list-name [Class list-name] Void 3 0 append-ins

[(Suc 0, 2, NullPointer, 7, 0)] φ_a *<proof>*

Some abbreviations for readability

abbreviation *Clist* == *Class list-name*

abbreviation *Ctest* == *Class test-name*

definition *phi-makelist* :: $ty_m (\varphi_m)$

where

$$\varphi_m \equiv \text{map } (\lambda(x,y). \text{Some } (x, y)) [$$

$$\begin{aligned} & (\quad \quad \quad [], [OK\ Ctest, Err \quad, Err \quad]), \\ & (\quad \quad \quad [Clist], [OK\ Ctest, Err \quad, Err \quad]), \\ & (\quad \quad \quad [], [OK\ Clist, Err \quad, Err \quad]), \\ & (\quad \quad \quad [Clist], [OK\ Clist, Err \quad, Err \quad]), \\ & (\quad \quad \quad [Integer, Clist], [OK\ Clist, Err \quad, Err \quad]), \\ \\ & (\quad \quad \quad [], [OK\ Clist, Err \quad, Err \quad]), \\ & (\quad \quad \quad [Clist], [OK\ Clist, Err \quad, Err \quad]), \\ & (\quad \quad \quad [], [OK\ Clist, OK\ Clist, Err \quad]), \\ & (\quad \quad \quad [Clist], [OK\ Clist, OK\ Clist, Err \quad]), \\ & (\quad \quad \quad [Integer, Clist], [OK\ Clist, OK\ Clist, Err \quad]), \\ \\ & (\quad \quad \quad [], [OK\ Clist, OK\ Clist, Err \quad]), \\ & (\quad \quad \quad [Clist], [OK\ Clist, OK\ Clist, Err \quad]), \\ & (\quad \quad \quad [], [OK\ Clist, OK\ Clist, OK\ Clist]), \\ & (\quad \quad \quad [Clist], [OK\ Clist, OK\ Clist, OK\ Clist]), \\ & (\quad \quad \quad [Integer, Clist], [OK\ Clist, OK\ Clist, OK\ Clist]), \\ \\ & (\quad \quad \quad [], [OK\ Clist, OK\ Clist, OK\ Clist]), \\ & (\quad \quad \quad [Clist], [OK\ Clist, OK\ Clist, OK\ Clist]), \\ & (\quad \quad \quad [Clist, Clist], [OK\ Clist, OK\ Clist, OK\ Clist]), \\ & (\quad \quad \quad [Void], [OK\ Clist, OK\ Clist, OK\ Clist]), \\ & (\quad \quad \quad [], [OK\ Clist, OK\ Clist, OK\ Clist]), \\ & (\quad \quad \quad [Clist], [OK\ Clist, OK\ Clist, OK\ Clist]), \\ & (\quad \quad \quad [Clist, Clist], [OK\ Clist, OK\ Clist, OK\ Clist]), \\ & (\quad \quad \quad [Void], [OK\ Clist, OK\ Clist, OK\ Clist]) \end{aligned}$$

lemma *types-makelist* [simp]: *check-types* $E\ 3\ (Suc\ (Suc\ (Suc\ 0)))\ (map\ OK\ \varphi_m)$ ⟨*proof*⟩

lemma *wt-makelist* [simp]:

wt-method $E\ test\ name\ []\ Void\ 3\ 2\ make\ list\ ins\ []\ \varphi_m$ ⟨*proof*⟩

lemma *wf-md'E*:

$$\begin{aligned} & \llbracket wf\ prog\ wf\ md\ P; \\ & \quad \wedge\ C\ S\ fs\ ms\ m. \llbracket (C, S, fs, ms) \in set\ P; m \in set\ ms \rrbracket \implies wf\ md'\ P\ C\ m \rrbracket \\ & \implies wf\ prog\ wf\ md'\ P \langle proof \rangle \end{aligned}$$

The whole program is welltyped:

definition *Phi* :: $ty_P (\Phi)$

where

$$\begin{aligned} \Phi\ C\ mn \equiv & \text{if } C = test\ name \wedge mn = makelist\ name \text{ then } \varphi_m \text{ else} \\ & \text{if } C = list\ name \wedge mn = append\ name \text{ then } \varphi_a \text{ else } [] \end{aligned}$$

lemma *wf-prog*:

wf-jvm-prog $\Phi\ E$ ⟨*proof*⟩

4.26.4 Conformance

Execution of the program will be typesafe, because its start state conforms to the welltyping:

lemma $E, \Phi \vdash start\ state\ E\ test\ name\ makelist\ name\ \surd$ ⟨*proof*⟩

4.26.5 Example for code generation: inferring method types

definition *test-kil* :: *jvm-prog* ⇒ *cname* ⇒ *ty list* ⇒ *ty* ⇒ *nat* ⇒ *nat* ⇒
ex-table ⇒ *instr list* ⇒ *ty_i' err list*

where

```
test-kil G C pTs rT mxs mxl et instr ≡
  (let first = Some ([],(OK (Class C))#(map OK pTs)@(replicate mxl Err));
    start = OK first#(replicate (size instr - 1) (OK None))
  in kiljvm G mxs (1+size pTs+mxl) rT instr et start)
```

lemma [*code*]:

```
unstabiles r step ss =
  fold (λp A. if ¬stable r step ss p then insert p A else A) [0..size ss] {}
⟨proof⟩
```

definition *some-elem* :: '*a set* ⇒ '*a where* [*code del*]:

```
some-elem = (%S. SOME x. x : S)
```

code-printing

```
constant some-elem → (SML) (case/ - of/ Set/ xs/ =>/ hd/ xs)
```

This code setup is just a demonstration and *not* sound!

notepad begin

```
⟨proof⟩
```

end

lemma [*code*]:

```
iter f step ss w = while (λ(ss, w). ¬ Set.is-empty w)
  (λ(ss, w).
    let p = some-elem w in propa f (step p (ss ! p)) ss (w - {p}))
  (ss, w)
⟨proof⟩
```

lemma *JVM-sup-unfold* [*code*]:

```
JVM-SemiType.sup S m n = lift2 (Opt.sup
  (Product.sup (Listn.sup (SemiType.sup S))
    (λx y. OK (map2 (lift2 (SemiType.sup S)) x y))))
⟨proof⟩
```

lemmas [*code*] = *SemiType.sup-def* [*unfolded exec-lub-def*] *JVM-le-unfold*

lemmas [*code*] = *lesub-def* *plussub-def*

lemma [*code*]:

```
is-refT T = (case T of NT ⇒ True | Class C ⇒ True | - ⇒ False)
⟨proof⟩
```

declare *app_i.simps* [*code*]

lemma [*code*]:

```
appi (Getfield F C, P, pc, mxs, Tr, (T#ST, LT)) =
  Predicate.holds (Predicate.bind (sees-field-i-i-i-o-i P C F C) (λTf. if P ⊢ T ≤ Class C then
  Predicate.single () else bot))
⟨proof⟩
```

lemma [code]:

```

appi (Putfield F C, P, pc, mxs, Tr, (T1#T2#ST, LT)) =
  Predicate.holds (Predicate.bind (sees-field-i-i-i-o-i P C F C) (λTf. if P ⊢ T2 ≤ (Class C) ∧ P ⊢
T1 ≤ Tf then Predicate.single () else bot))
⟨proof⟩

```

lemma [code]:

```

appi (Invoke M n, P, pc, mxs, Tr, (ST,LT)) =
  (n < length ST ∧
  (ST!n ≠ NT →
  (case ST!n of
  Class C ⇒ Predicate.holds (Predicate.bind (Method-i-i-i-o-o-o-o P C M) (λ(Ts, T, m, D). if
P ⊢ rev (take n ST) [≤] Ts then Predicate.single () else bot))
  | - ⇒ False)))
⟨proof⟩

```

lemmas [code] =

```

SemiType.sup-def [unfolded exec-lub-def]
widen.equation
is-relevant-class.simps

```

definition test1 **where**

```

test1 = test-kill E list-name [Class list-name] Void 3 0
[(Suc 0, 2, NullPointer, 7, 0)] append-ins

```

definition test2 **where**

```

test2 = test-kill E test-name [] Void 3 2 [] make-list-ins

```

definition test3 **where** test3 = φ_a

definition test4 **where** test4 = φ_m

⟨ML⟩

end

Chapter 5

Compilation

5.1 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*)

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*(*FAss e*₁ *F D e*₂) = *max* (*max-vars e*₁) (*max-vars e*₂)
| *max-vars*(*e.M(es)*) = *max* (*max-vars e*) (*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-vars*(*throw e*) = *max-vars e*
| *max-vars*(*try e*₁ *catch*(*C V*) *e*₂) = *max* (*max-vars e*₁) (*max-vars e*₂ + 1)

| *max-varss* [] = 0
| *max-varss* (*e#es*) = *max* (*max-vars e*) (*max-varss es*)

inductive

*eval*₁ :: *J*₁-*prog* ⇒ *expr*₁ ⇒ *state*₁ ⇒ *expr*₁ ⇒ *state*₁ ⇒ *bool*
 (- ⊢₁ ((1⟨-,/-⟩) ⇒ / (1⟨-,/-⟩)) [51,0,0,0,0] 81)
and *evals*₁ :: *J*₁-*prog* ⇒ *expr*₁ *list* ⇒ *state*₁ ⇒ *expr*₁ *list* ⇒ *state*₁ ⇒ *bool*
 (- ⊢₁ ((1⟨-,/-⟩) [⇒] / (1⟨-,/-⟩)) [51,0,0,0,0] 81)
for *P* :: *J*₁-*prog*

where

*New*₁:

$\llbracket P \vdash_1 \langle \text{new } C, (h, l) \rangle \Rightarrow \langle \text{addr } a, (h', l) \rangle \rrbracket$
 $\implies P \vdash_1 \langle \text{new } C, (h, l) \rangle \Rightarrow \langle \text{addr } a, (h', l) \rangle$

| *NewFail*₁:

$\text{new-Addr } h = \text{None} \implies$
 $P \vdash_1 \langle \text{new } C, (h, l) \rangle \Rightarrow \langle \text{THROW OutOfMemory}, (h, l) \rangle$

| *Cast*₁:

$\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, l) \rangle; h a = \text{Some}(D, fs); P \vdash D \preceq^* C \rrbracket$
 $\implies P \vdash_1 \langle \text{Cast } C e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, l) \rangle$

| *CastNull*₁:

$P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \implies$
 $P \vdash_1 \langle \text{Cast } C e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle$

| *CastFail*₁:

$\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, l) \rangle; h a = \text{Some}(D, fs); \neg P \vdash D \preceq^* C \rrbracket$
 $\implies P \vdash_1 \langle \text{Cast } C e, s_0 \rangle \Rightarrow \langle \text{THROW ClassCast}, (h, l) \rangle$

| *CastThrow*₁:

$P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \implies$
 $P \vdash_1 \langle \text{Cast } C e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$

| *Val*₁:

$P \vdash_1 \langle \text{Val } v, s \rangle \Rightarrow \langle \text{Val } v, s \rangle$

| *BinOp*₁:

$\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v_2, s_2 \rangle; \text{binop}(bop, v_1, v_2) = \text{Some } v \rrbracket$
 $\implies P \vdash_1 \langle e_1 \ll bop \gg e_2, s_0 \rangle \Rightarrow \langle \text{Val } v, s_2 \rangle$

| *BinOpThrow*₁₁:

$P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle \implies$
 $P \vdash_1 \langle e_1 \ll bop \gg e_2, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle$

| *BinOpThrow*₂₁:

$\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{throw } e, s_2 \rangle \rrbracket$
 $\implies P \vdash_1 \langle e_1 \ll bop \gg e_2, s_0 \rangle \Rightarrow \langle \text{throw } e, s_2 \rangle$

| *Var*₁:

$\llbracket ls!i = v; i < \text{size } ls \rrbracket \implies$
 $P \vdash_1 \langle \text{Var } i, (h, ls) \rangle \Rightarrow \langle \text{Val } v, (h, ls) \rangle$

| *LAss*₁:

$\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, (h, ls) \rangle; i < \text{size } ls; ls' = ls[i := v] \rrbracket$
 $\implies P \vdash_1 \langle i := e, s_0 \rangle \Rightarrow \langle \text{unit}, (h, ls') \rangle$

| *LAssThrow*₁:

$P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \implies$
 $P \vdash_1 \langle i := e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$

| *FAcc*₁:

$\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, ls) \rangle; h a = \text{Some}(C, fs); fs(F, D) = \text{Some } v \rrbracket$
 $\implies P \vdash_1 \langle e \cdot F\{D\}, s_0 \rangle \Rightarrow \langle \text{Val } v, (h, ls) \rangle$

| *FAccNull*₁:

$P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \implies$
 $P \vdash_1 \langle e \cdot F\{D\}, s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_1 \rangle$

| *FAccThrow*₁:

$P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \implies$
 $P \vdash_1 \langle e \cdot F\{D\}, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$

| *FAss*₁:
 $\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2) \rangle; \quad h_2 \ a = \text{Some}(C, fs); fs' = fs((F, D) \mapsto v); h_2' = h_2(a \mapsto (C, fs')) \rrbracket$
 $\implies P \vdash_1 \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{unit}, (h_2', l_2) \rangle$

| *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:
 $\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle \rrbracket$
 $\implies P \vdash_1 \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle$

| *CallObjThrow*₁:
 $P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \implies$
 $P \vdash_1 \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$

| *CallNull*₁:
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle; P \vdash_1 \langle es, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, s_2 \rangle \rrbracket$
 $\implies P \vdash_1 \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_2 \rangle$

| *Call*₁:
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash_1 \langle es, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_2, ls_2) \rangle; \quad h_2 \ a = \text{Some}(C, fs); P \vdash C \text{ sees } M: Ts \rightarrow T = \text{body in } D; \quad \text{size } vs = \text{size } Ts; ls_2' = (\text{Addr } a) \# vs \ @ \ \text{replicate } (\text{max-vars body}) \ \text{undefined}; \quad P \vdash_1 \langle \text{body}, (h_2, ls_2') \rangle \Rightarrow \langle e', (h_3, ls_3) \rangle \rrbracket$
 $\implies P \vdash_1 \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle e', (h_3, ls_2) \rangle$

| *CallParamsThrow*₁:
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash_1 \langle es, s_1 \rangle [\Rightarrow] \langle es', s_2 \rangle; \quad es' = \text{map Val } vs \ @ \ \text{throw } ex \ \# \ es_2 \rrbracket$
 $\implies P \vdash_1 \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle \text{throw } ex, s_2 \rangle$

| *Block*₁:
 $P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle e', s_1 \rangle \implies P \vdash_1 \langle \text{Block } i \ T \ e, s_0 \rangle \Rightarrow \langle e', s_1 \rangle$

| *Seq*₁:
 $\llbracket P \vdash_1 \langle e_0, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash_1 \langle e_1, s_1 \rangle \Rightarrow \langle e_2, s_2 \rangle \rrbracket$
 $\implies P \vdash_1 \langle e_0;; e_1, s_0 \rangle \Rightarrow \langle e_2, s_2 \rangle$

| *SeqThrow*₁:
 $P \vdash_1 \langle e_0, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle \implies$
 $P \vdash_1 \langle e_0;; e_1, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle$

| *CondT*₁:
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle; P \vdash_1 \langle e_1, s_1 \rangle \Rightarrow \langle e', s_2 \rangle \rrbracket$
 $\implies P \vdash_1 \langle \text{if } (e) \ e_1 \ \text{else } e_2, s_0 \rangle \Rightarrow \langle e', s_2 \rangle$

| *CondF*₁:
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{false}, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle e', s_2 \rangle \rrbracket$
 $\implies P \vdash_1 \langle \text{if } (e) \ e_1 \ \text{else } e_2, s_0 \rangle \Rightarrow \langle e', s_2 \rangle$

| *CondThrow*₁:
 $P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \implies$
 $P \vdash_1 \langle \text{if } (e) \ e_1 \ \text{else } e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$

| *WhileF*₁:
 $P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{false}, s_1 \rangle \implies$

$P \vdash_1 \langle \text{while } (e) \ c, s_0 \rangle \Rightarrow \langle \text{unit}, s_1 \rangle$
| *WhileT₁*:
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle; P \vdash_1 \langle c, s_1 \rangle \Rightarrow \langle \text{Val } v_1, s_2 \rangle; P \vdash_1 \langle \text{while } (e) \ c, s_2 \rangle \Rightarrow \langle e_3, s_3 \rangle \rrbracket$
 $\implies P \vdash_1 \langle \text{while } (e) \ c, s_0 \rangle \Rightarrow \langle e_3, s_3 \rangle$
| *WhileCondThrow₁*:
 $P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \implies$
 $P \vdash_1 \langle \text{while } (e) \ c, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$
| *WhileBodyThrow₁*:
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle; P \vdash_1 \langle c, s_1 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle \rrbracket$
 $\implies P \vdash_1 \langle \text{while } (e) \ c, s_0 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle$

| *Throw₁*:
 $P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle \implies$
 $P \vdash_1 \langle \text{throw } e, s_0 \rangle \Rightarrow \langle \text{Throw } a, s_1 \rangle$
| *ThrowNull₁*:
 $P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \implies$
 $P \vdash_1 \langle \text{throw } e, s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_1 \rangle$
| *ThrowThrow₁*:
 $P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \implies$
 $P \vdash_1 \langle \text{throw } e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$

| *Try₁*:
 $P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle \implies$
 $P \vdash_1 \langle \text{try } e_1 \ \text{catch}(C \ i) \ e_2, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle$
| *TryCatch₁*:
 $\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Throw } a, (h_1, ls_1) \rangle;$
 $h_1 \ a = \text{Some}(D, fs); P \vdash D \preceq^* C; i < \text{length } ls_1;$
 $P \vdash_1 \langle e_2, (h_1, ls_1[i := \text{Addr } a]) \rangle \Rightarrow \langle e_2', (h_2, ls_2) \rangle \rrbracket$
 $\implies P \vdash_1 \langle \text{try } e_1 \ \text{catch}(C \ i) \ e_2, s_0 \rangle \Rightarrow \langle e_2', (h_2, ls_2) \rangle$
| *TryThrow₁*:
 $\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Throw } a, (h_1, ls_1) \rangle; h_1 \ a = \text{Some}(D, fs); \neg P \vdash D \preceq^* C \rrbracket$
 $\implies P \vdash_1 \langle \text{try } e_1 \ \text{catch}(C \ i) \ e_2, s_0 \rangle \Rightarrow \langle \text{Throw } a, (h_1, ls_1) \rangle$

| *Nil₁*:
 $P \vdash_1 \langle [], s \rangle [\Rightarrow] \langle [], s \rangle$

| *Cons₁*:
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash_1 \langle es, s_1 \rangle [\Rightarrow] \langle es', s_2 \rangle \rrbracket$
 $\implies P \vdash_1 \langle e \# es, s_0 \rangle [\Rightarrow] \langle \text{Val } v \# es', s_2 \rangle$
| *ConsThrow₁*:
 $P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \implies$
 $P \vdash_1 \langle e \# es, s_0 \rangle [\Rightarrow] \langle \text{throw } e' \# es, s_1 \rangle$

lemma *eval₁-preserves-len*:

$$P \vdash_1 \langle e_0, (h_0, ls_0) \rangle \Rightarrow \langle e_1, (h_1, ls_1) \rangle \implies \text{length } ls_0 = \text{length } ls_1$$

and *evals₁-preserves-len*:

$$P \vdash_1 \langle es_0, (h_0, ls_0) \rangle [\Rightarrow] \langle es_1, (h_1, ls_1) \rangle \implies \text{length } ls_0 = \text{length } ls_1 \langle \text{proof} \rangle$$

lemma *evals₁-preserves-elen*:

$$\bigwedge es' \ s \ s'. P \vdash_1 \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \implies \text{length } es = \text{length } es' \langle \text{proof} \rangle$$

lemma *eval₁-final*: $P \vdash_1 \langle e, s \rangle \Rightarrow \langle e', s' \rangle \implies \text{final } e'$

and *evals₁-final*: $P \vdash_1 \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \implies \text{finals } es' \langle \text{proof} \rangle$

end

5.2 Well-Formedness of Intermediate Language

```
theory J1WellForm
imports ../J/JWellForm J1
begin
```

5.2.1 Well-Typedness

type-synonym

$env_1 = ty\ list$ — type environment indexed by variable number

inductive

```
WT1 :: [J1-prog, env1, expr1, ty] ⇒ bool
  ((-, - ⊢1 / - :: -) [51, 51, 51] 50)
and WTs1 :: [J1-prog, env1, expr1 list, ty list] ⇒ bool
  ((-, - ⊢1 / - :::] -) [51, 51, 51] 50)
for P :: J1-prog
where

  WTNew1:
  is-class P C ⇒
  P, E ⊢1 new C :: Class C

  | WTCast1:
  [| P, E ⊢1 e :: Class D; is-class P C; P ⊢ C ≤* D ∨ P ⊢ D ≤* C |]
  ⇒ P, E ⊢1 Cast C e :: Class C

  | WTVal1:
  typeof v = Some T ⇒
  P, E ⊢1 Val v :: T

  | WTVar1:
  [| E!i = T; i < size E |]
  ⇒ P, E ⊢1 Var i :: T

  | WTBinOp1:
  [| P, E ⊢1 e1 :: T1; P, E ⊢1 e2 :: T2;
   case bop of Eq ⇒ (P ⊢ T1 ≤ T2 ∨ P ⊢ T2 ≤ T1) ∧ T = Boolean
   | Add ⇒ T1 = Integer ∧ T2 = Integer ∧ T = Integer |]
  ⇒ P, E ⊢1 e1 «bop» e2 :: T

  | WTLAss1:
  [| E!i = T; i < size E; P, E ⊢1 e :: T'; P ⊢ T' ≤ T |]
  ⇒ P, E ⊢1 i:=e :: Void

  | WTFAcc1:
  [| P, E ⊢1 e :: Class C; P ⊢ C sees F:T in D |]
  ⇒ P, E ⊢1 e.F{D} :: T

  | WTFAss1:
  [| P, E ⊢1 e1 :: Class C; P ⊢ C sees F:T in D; P, E ⊢1 e2 :: T'; P ⊢ T' ≤ T |]
```

$$\Longrightarrow P, E \vdash_1 e_1 \cdot F\{D\} := e_2 :: \text{Void}$$

| *WTCall*₁:

$$\llbracket P, E \vdash_1 e :: \text{Class } C; P \vdash C \text{ sees } M:Ts' \rightarrow T = m \text{ in } D;$$

$$P, E \vdash_1 es \llbracket :: \rrbracket Ts; P \vdash Ts \llbracket \leq \rrbracket Ts' \rrbracket$$

$$\Longrightarrow P, E \vdash_1 e \cdot M(es) :: T$$

| *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:

$$\llbracket P, E \vdash_1 e_1 :: T; P, E@[Class C] \vdash_1 e_2 :: T; \text{is-class } P C \rrbracket$$

$$\Longrightarrow P, E \vdash_1 \text{try } e_1 \text{ catch}(C i) e_2 :: T$$

| *WTNil*₁:

$$P, E \vdash_1 [] \llbracket :: \rrbracket []$$

| *WTCons*₁:

$$\llbracket P, E \vdash_1 e :: T; P, E \vdash_1 es \llbracket :: \rrbracket Ts \rrbracket$$

$$\Longrightarrow P, E \vdash_1 e \# es \llbracket :: \rrbracket T \# Ts$$

lemma *WTs*₁-same-size: $\bigwedge Ts. P, E \vdash_1 es \llbracket :: \rrbracket Ts \Longrightarrow \text{size } es = \text{size } Ts \langle \text{proof} \rangle$

lemma *WT*₁-unique:

$$P, E \vdash_1 e :: T_1 \Longrightarrow (\bigwedge T_2. P, E \vdash_1 e :: T_2 \Longrightarrow T_1 = T_2) \text{ and}$$

$$P, E \vdash_1 es \llbracket :: \rrbracket Ts_1 \Longrightarrow (\bigwedge Ts_2. P, E \vdash_1 es \llbracket :: \rrbracket Ts_2 \Longrightarrow Ts_1 = Ts_2) \langle \text{proof} \rangle$$

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 $P, E \vdash_1 es \llbracket :: \rrbracket Ts \Longrightarrow \text{True} \langle \text{proof} \rangle$

5.2.2 Well-formedness

primrec $\mathcal{B} :: \text{expr}_1 \Rightarrow \text{nat} \Rightarrow \text{bool}$

and $Bs :: \text{expr}_1 \text{ list} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**

$\mathcal{B} (\text{new } C) i = \text{True} \mid$

$\mathcal{B} (\text{Cast } C \ e) \ i = \mathcal{B} \ e \ i \mid$
 $\mathcal{B} (\text{Val } v) \ i = \text{True} \mid$
 $\mathcal{B} (e_1 \ll\text{bop}\gg e_2) \ i = (\mathcal{B} \ e_1 \ i \wedge \mathcal{B} \ e_2 \ i) \mid$
 $\mathcal{B} (\text{Var } j) \ i = \text{True} \mid$
 $\mathcal{B} (e \cdot F\{D\}) \ i = \mathcal{B} \ e \ i \mid$
 $\mathcal{B} (j := e) \ i = \mathcal{B} \ e \ i \mid$
 $\mathcal{B} (e_1 \cdot F\{D\} := e_2) \ i = (\mathcal{B} \ e_1 \ i \wedge \mathcal{B} \ e_2 \ i) \mid$
 $\mathcal{B} (e \cdot M(es)) \ i = (\mathcal{B} \ e \ i \wedge \mathcal{B} s \ es \ i) \mid$
 $\mathcal{B} (\{j:T ; e\}) \ i = (i = j \wedge \mathcal{B} \ e \ (i+1)) \mid$
 $\mathcal{B} (e_1 ;; e_2) \ i = (\mathcal{B} \ e_1 \ i \wedge \mathcal{B} \ e_2 \ i) \mid$
 $\mathcal{B} (\text{if } (e) \ e_1 \ \text{else } e_2) \ i = (\mathcal{B} \ e \ i \wedge \mathcal{B} \ e_1 \ i \wedge \mathcal{B} \ e_2 \ i) \mid$
 $\mathcal{B} (\text{throw } e) \ i = \mathcal{B} \ e \ i \mid$
 $\mathcal{B} (\text{while } (e) \ c) \ i = (\mathcal{B} \ e \ i \wedge \mathcal{B} \ c \ i) \mid$
 $\mathcal{B} (\text{try } e_1 \ \text{catch}(C \ j) \ e_2) \ i = (\mathcal{B} \ e_1 \ i \wedge i=j \wedge \mathcal{B} \ e_2 \ (i+1)) \mid$

$\mathcal{B} s \ [] \ i = \text{True} \mid$
 $\mathcal{B} s \ (e \# es) \ i = (\mathcal{B} \ e \ i \wedge \mathcal{B} s \ es \ i)$

definition $wf\text{-}J_1\text{-mdecl} :: J_1\text{-prog} \Rightarrow \text{cname} \Rightarrow \text{expr}_1 \ \text{mdecl} \Rightarrow \text{bool}$

where

$wf\text{-}J_1\text{-mdecl} \ P \ C \equiv \lambda(M, Ts, T, body).$
 $(\exists T'. P, \text{Class } C \# Ts \vdash_1 \text{body} :: T' \wedge P \vdash T' \leq T) \wedge$
 $D \ \text{body} \ [\{\dots \text{size } Ts\}] \wedge \mathcal{B} \ \text{body} \ (\text{size } Ts + 1)$

lemma $wf\text{-}J_1\text{-mdecl}[\text{simp}]$:

$wf\text{-}J_1\text{-mdecl} \ P \ C \ (M, Ts, T, body) \equiv$
 $((\exists T'. P, \text{Class } C \# Ts \vdash_1 \text{body} :: T' \wedge P \vdash T' \leq T) \wedge$
 $D \ \text{body} \ [\{\dots \text{size } Ts\}] \wedge \mathcal{B} \ \text{body} \ (\text{size } Ts + 1)) \langle \text{proof} \rangle$

abbreviation $wf\text{-}J_1\text{-prog} == wf\text{-prog} \ wf\text{-}J_1\text{-mdecl}$

end

5.3 Program Compilation

theory *PCompiler*

imports *../Common/WellForm*

begin

definition $\text{comp}M :: ('a \Rightarrow 'b) \Rightarrow 'a \ \text{mdecl} \Rightarrow 'b \ \text{mdecl}$

where

$\text{comp}M \ f \equiv \lambda(M, Ts, T, m). (M, Ts, T, f \ m)$

definition $\text{comp}C :: ('a \Rightarrow 'b) \Rightarrow 'a \ \text{cdecl} \Rightarrow 'b \ \text{cdecl}$

where

$\text{comp}C \ f \equiv \lambda(C, D, Fdecls, Mdecls). (C, D, Fdecls, \text{map} \ (\text{comp}M \ f) \ Mdecls)$

definition $\text{comp}P :: ('a \Rightarrow 'b) \Rightarrow 'a \ \text{prog} \Rightarrow 'b \ \text{prog}$

where

$\text{comp}P \ f \equiv \text{map} \ (\text{comp}C \ f)$

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 :

$map\text{-of } (map (\lambda(x,a,b,c).(x,a,b,f c)) ts) =$
 $map\text{-option } (\lambda(a,b,c).(a,b,f c)) \circ (map\text{-of } ts)\langle proof \rangle$

lemma *class-compP*:

$class P C = Some (D, fs, ms)$
 $\implies class (compP f P) C = Some (D, fs, map (compM f) ms)\langle proof \rangle$

lemma *class-compPD*:

$class (compP f P) C = Some (D, fs, cms)$
 $\implies \exists ms. class P C = Some(D,fs,ms) \wedge cms = map (compM f) ms\langle proof \rangle$

lemma [*simp*]: $is\text{-class } (compP f P) C = is\text{-class } P C\langle proof \rangle$

lemma [*simp*]: $class (compP f P) C = map\text{-option } (\lambda c. snd(compC f (C,c))) (class P C)\langle proof \rangle$

lemma *sees-methods-compP*:

$P \vdash C \text{ sees-methods } Mm \implies$
 $compP f P \vdash C \text{ sees-methods } (map\text{-option } (\lambda((Ts,T,m),D). ((Ts,T,f m),D)) \circ Mm)\langle proof \rangle$

lemma *sees-method-compP*:

$P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } D \implies$
 $compP f P \vdash C \text{ sees } M: Ts \rightarrow T = (f m) \text{ in } D\langle proof \rangle$

lemma [*simp*]:

$P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } D \implies$
 $method (compP f P) C M = (D, Ts, T, f m)\langle proof \rangle$

lemma *sees-methods-compPD*:

$\llbracket cP \vdash C \text{ sees-methods } Mm'; cP = compP f P \rrbracket \implies$
 $\exists Mm. P \vdash C \text{ sees-methods } Mm \wedge$
 $Mm' = (map\text{-option } (\lambda((Ts,T,m),D). ((Ts,T,f m),D)) \circ Mm)\langle proof \rangle$

lemma *sees-method-compPD*:

$compP f P \vdash C \text{ sees } M: Ts \rightarrow T = fm \text{ in } D \implies$
 $\exists m. P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } D \wedge f m = fm\langle proof \rangle$

lemma [*simp*]: $subcls1 (compP f P) = subcls1 P\langle proof \rangle$

lemma *compP-widen*[*simp*]: $(compP f P \vdash T \leq T') = (P \vdash T \leq T')\langle proof \rangle$

lemma [*simp*]: $(compP f P \vdash Ts [\leq] Ts') = (P \vdash Ts [\leq] Ts')\langle proof \rangle$

lemma [*simp*]: $is\text{-type } (compP f P) T = is\text{-type } P T\langle proof \rangle$

lemma [*simp*]: $(compP (f::'a \Rightarrow 'b) P \vdash C \text{ has-fields } FDTs) = (P \vdash C \text{ has-fields } FDTs)\langle proof \rangle$

lemma [*simp*]: $fields (compP f P) C = fields P C\langle proof \rangle$

lemma [*simp*]: $(compP f P \vdash C \text{ sees } F:T \text{ in } D) = (P \vdash C \text{ sees } F:T \text{ in } D)\langle proof \rangle$

lemma [*simp*]: $field (compP f P) F D = field P F D\langle proof \rangle$

5.3.1 Invariance of *wf-prog* under compilation

lemma [iff]: $\text{distinct-fst } (\text{compP } f \ P) = \text{distinct-fst } P \langle \text{proof} \rangle$

lemma [iff]: $\text{distinct-fst } (\text{map } (\text{compM } f) \ ms) = \text{distinct-fst } ms \langle \text{proof} \rangle$

lemma [iff]: $\text{wf-syscls } (\text{compP } f \ P) = \text{wf-syscls } P \langle \text{proof} \rangle$

lemma [iff]: $\text{wf-fdecl } (\text{compP } f \ P) = \text{wf-fdecl } P \langle \text{proof} \rangle$

lemma *set-compP*:

$((C, D, fs, ms') \in \text{set}(\text{compP } f \ P)) =$
 $(\exists ms. (C, D, fs, ms) \in \text{set } P \wedge ms' = \text{map } (\text{compM } f) \ ms) \langle \text{proof} \rangle$

lemma *wf-cdecl-compPI*:

$\llbracket \bigwedge C \ M \ Ts \ T \ m.$
 $\llbracket \text{wf-mdecl } wf_1 \ P \ C \ (M, Ts, T, m); P \vdash C \text{ sees } M:Ts \rightarrow T = m \text{ in } C \rrbracket$
 $\implies \text{wf-mdecl } wf_2 \ (\text{compP } f \ P) \ C \ (M, Ts, T, f \ m);$
 $\forall x \in \text{set } P. \text{wf-cdecl } wf_1 \ P \ x; x \in \text{set } (\text{compP } f \ P); \text{wf-prog } p \ P \rrbracket$
 $\implies \text{wf-cdecl } wf_2 \ (\text{compP } f \ P) \ x \langle \text{proof} \rangle$

lemma *wf-prog-compPI*:

assumes *lift*:

$\bigwedge C \ M \ Ts \ T \ m.$
 $\llbracket P \vdash C \text{ sees } M:Ts \rightarrow T = m \text{ in } C; \text{wf-mdecl } wf_1 \ P \ C \ (M, Ts, T, m) \rrbracket$
 $\implies \text{wf-mdecl } wf_2 \ (\text{compP } f \ P) \ C \ (M, Ts, T, f \ m)$

and *wf*: $\text{wf-prog } wf_1 \ P$

shows $\text{wf-prog } wf_2 \ (\text{compP } f \ P) \langle \text{proof} \rangle$

end

theory *Hidden*

imports *List-Index.List-Index*

begin

definition *hidden* :: 'a list \Rightarrow nat \Rightarrow bool **where**

hidden $xs \ i \equiv i < \text{size } xs \wedge xs!i \in \text{set}(\text{drop } (i+1) \ xs)$

lemma *hidden-last-index*: $x \in \text{set } xs \implies \text{hidden } (xs \ @ \ [x]) \ (\text{last-index } xs \ x)$
 $\langle \text{proof} \rangle$

lemma *hidden-inacc*: $\text{hidden } xs \ i \implies \text{last-index } xs \ x \neq i$
 $\langle \text{proof} \rangle$

lemma [simp]: $\text{hidden } xs \ i \implies \text{hidden } (xs@[x]) \ i$
 $\langle \text{proof} \rangle$

lemma *fun-upds-apply*:

$(m(xs[\mapsto]ys)) \ x =$
 $(\text{let } xs' = \text{take } (\text{size } ys) \ xs$
 $\text{in if } x \in \text{set } xs' \text{ then } \text{Some}(ys \ ! \ \text{last-index } xs' \ x) \text{ else } m \ x)$
 $\langle \text{proof} \rangle$

lemma *map-upds-apply-eq-Some*:

$((m(xs[\mapsto]ys)) x = \text{Some } y) =$
 $(\text{let } xs' = \text{take } (\text{size } ys) \text{ } xs$
 $\text{ in if } x \in \text{set } xs' \text{ then } ys \text{ ! last-index } xs' \text{ } x = y \text{ else } m \text{ } x = \text{Some } y)$
 $\langle \text{proof} \rangle$

lemma *map-upds-upd-conv-last-index*:

$\llbracket x \in \text{set } xs; \text{size } xs \leq \text{size } ys \rrbracket$
 $\implies m(xs[\mapsto]ys, x \mapsto y) = m(xs[\mapsto]ys[\text{last-index } xs \text{ } x := y])$
 $\langle \text{proof} \rangle$

end

5.4 Compilation Stage 1

theory *Compiler1* **imports** *PCompiler J1 Hidden* **begin**

Replacing variable names by indices.

primrec $\text{comp}E_1 :: \text{vname list} \Rightarrow \text{expr} \Rightarrow \text{expr}_1$
and $\text{comp}Es_1 :: \text{vname list} \Rightarrow \text{expr list} \Rightarrow \text{expr}_1 \text{ list}$ **where**
 $\text{comp}E_1 \text{ } Vs \text{ } (\text{new } C) = \text{new } C$
 $\text{comp}E_1 \text{ } Vs \text{ } (\text{Cast } C \text{ } e) = \text{Cast } C \text{ } (\text{comp}E_1 \text{ } Vs \text{ } e)$
 $\text{comp}E_1 \text{ } Vs \text{ } (\text{Val } v) = \text{Val } v$
 $\text{comp}E_1 \text{ } Vs \text{ } (e_1 \text{ } \llcorner \text{bop} \llcorner \text{ } e_2) = (\text{comp}E_1 \text{ } Vs \text{ } e_1) \text{ } \llcorner \text{bop} \llcorner \text{ } (\text{comp}E_1 \text{ } Vs \text{ } e_2)$
 $\text{comp}E_1 \text{ } Vs \text{ } (\text{Var } V) = \text{Var}(\text{last-index } Vs \text{ } V)$
 $\text{comp}E_1 \text{ } Vs \text{ } (V := e) = (\text{last-index } Vs \text{ } V) := (\text{comp}E_1 \text{ } Vs \text{ } e)$
 $\text{comp}E_1 \text{ } Vs \text{ } (e \cdot F\{D\}) = (\text{comp}E_1 \text{ } Vs \text{ } e) \cdot F\{D\}$
 $\text{comp}E_1 \text{ } Vs \text{ } (e_1 \cdot F\{D\} := e_2) = (\text{comp}E_1 \text{ } Vs \text{ } e_1) \cdot F\{D\} := (\text{comp}E_1 \text{ } Vs \text{ } e_2)$
 $\text{comp}E_1 \text{ } Vs \text{ } (e \cdot M(es)) = (\text{comp}E_1 \text{ } Vs \text{ } e) \cdot M(\text{comp}Es_1 \text{ } Vs \text{ } es)$
 $\text{comp}E_1 \text{ } Vs \text{ } \{V:T; e\} = \{(\text{size } Vs):T; \text{comp}E_1 \text{ } (Vs@[V]) \text{ } e\}$
 $\text{comp}E_1 \text{ } Vs \text{ } (e_1 ;; e_2) = (\text{comp}E_1 \text{ } Vs \text{ } e_1) ;; (\text{comp}E_1 \text{ } Vs \text{ } e_2)$
 $\text{comp}E_1 \text{ } Vs \text{ } (\text{if } (e) \text{ } e_1 \text{ } \text{else } e_2) = \text{if } (\text{comp}E_1 \text{ } Vs \text{ } e) \text{ } (\text{comp}E_1 \text{ } Vs \text{ } e_1) \text{ } \text{else } (\text{comp}E_1 \text{ } Vs \text{ } e_2)$
 $\text{comp}E_1 \text{ } Vs \text{ } (\text{while } (e) \text{ } c) = \text{while } (\text{comp}E_1 \text{ } Vs \text{ } e) \text{ } (\text{comp}E_1 \text{ } Vs \text{ } c)$
 $\text{comp}E_1 \text{ } Vs \text{ } (\text{throw } e) = \text{throw } (\text{comp}E_1 \text{ } Vs \text{ } e)$
 $\text{comp}E_1 \text{ } Vs \text{ } (\text{try } e_1 \text{ } \text{catch}(C \text{ } V) \text{ } e_2) =$
 $\text{try}(\text{comp}E_1 \text{ } Vs \text{ } e_1) \text{ } \text{catch}(C \text{ } (\text{size } Vs)) \text{ } (\text{comp}E_1 \text{ } (Vs@[V]) \text{ } e_2)$
 $\text{comp}Es_1 \text{ } Vs \text{ } [] = []$
 $\text{comp}Es_1 \text{ } Vs \text{ } (e \# es) = \text{comp}E_1 \text{ } Vs \text{ } e \text{ } \# \text{comp}Es_1 \text{ } Vs \text{ } es$

lemma $[\text{simp}]$: $\text{comp}Es_1 \text{ } Vs \text{ } es = \text{map } (\text{comp}E_1 \text{ } Vs) \text{ } es \langle \text{proof} \rangle$

primrec $\text{fin}_1 :: \text{expr} \Rightarrow \text{expr}_1$ **where**

$\text{fin}_1(\text{Val } v) = \text{Val } v$
 $\text{fin}_1(\text{throw } e) = \text{throw}(\text{fin}_1 \text{ } e)$

lemma *comp-final*: $\text{final } e \implies \text{comp}E_1 \text{ } Vs \text{ } e = \text{fin}_1 \text{ } e \langle \text{proof} \rangle$

lemma $[\text{simp}]$:

$\bigwedge Vs. \text{max-vars } (\text{comp}E_1 \text{ } Vs \text{ } e) = \text{max-vars } e$
and $\bigwedge Vs. \text{max-varss } (\text{comp}Es_1 \text{ } Vs \text{ } es) = \text{max-varss } es \langle \text{proof} \rangle$

Compiling programs:

definition $compP_1 :: J\text{-prog} \Rightarrow J_1\text{-prog}$

where

$compP_1 \equiv compP (\lambda(pns, body). compE_1 (this\#pns) body)$

end

5.5 Correctness of Stage 1

theory *Correctness1*

imports *J1WellForm Compiler1*

begin

5.5.1 Correctness of program compilation

primrec $unmod :: expr_1 \Rightarrow nat \Rightarrow bool$

and $unmods :: expr_1 list \Rightarrow nat \Rightarrow bool$ **where**

$unmod (new C) i = True$ |

$unmod (Cast C e) i = unmod e i$ |

$unmod (Val v) i = True$ |

$unmod (e_1 \ll bop \gg e_2) i = (unmod e_1 i \wedge unmod e_2 i)$ |

$unmod (Var i) j = True$ |

$unmod (i:=e) j = (i \neq j \wedge unmod e j)$ |

$unmod (e \cdot F\{D\}) i = unmod e i$ |

$unmod (e_1 \cdot F\{D\} := e_2) i = (unmod e_1 i \wedge unmod e_2 i)$ |

$unmod (e \cdot M(es)) i = (unmod e i \wedge unmods es i)$ |

$unmod \{j:T; e\} i = unmod e i$ |

$unmod (e_1;;e_2) i = (unmod e_1 i \wedge unmod e_2 i)$ |

$unmod (if (e) e_1 else e_2) i = (unmod e i \wedge unmod e_1 i \wedge unmod e_2 i)$ |

$unmod (while (e) c) i = (unmod e i \wedge unmod c i)$ |

$unmod (throw e) i = unmod e i$ |

$unmod (try e_1 catch (C i) e_2) j = (unmod e_1 j \wedge (if i=j then False else unmod e_2 j))$ |

$unmods ([]) i = True$ |

$unmods (e\#es) i = (unmod e i \wedge unmods es i)$

lemma *hidden-unmod*: $\bigwedge Vs. hidden Vs i \Longrightarrow unmod (compE_1 Vs e) i$ **and**

$\bigwedge Vs. hidden Vs i \Longrightarrow unmods (compEs_1 Vs es) i$ *<proof>*

lemma *eval₁-preserves-unmod*:

$\llbracket P \vdash_1 \langle e, (h, ls) \rangle \Rightarrow \langle e', (h', ls') \rangle; unmod e i; i < size ls \rrbracket$

$\Longrightarrow ls ! i = ls' ! i$

and $\llbracket P \vdash_1 \langle es, (h, ls) \rangle [\Rightarrow] \langle es', (h', ls') \rangle; unmods es i; i < size ls \rrbracket$

$\Longrightarrow ls ! i = ls' ! i$ *<proof>*

lemma *LAss-lem*:

$\llbracket x \in set xs; size xs \leq size ys \rrbracket$

$\Longrightarrow m_1 \subseteq_m m_2(xs[\mapsto]ys) \Longrightarrow m_1(x \mapsto y) \subseteq_m m_2(xs[\mapsto]ys[last-index xs x := y])$ *<proof>*

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*: $V \in set Vs \Longrightarrow ls ! last-index Vs V = ls' ! last-index Vs V$

and *size*: $size ls = size ls' \quad size Vs < size ls'$

shows $l'(V := l V) \subseteq_m [Vs \mapsto ls'] \langle proof \rangle$

The main theorem:

theorem assumes $wf: wuf\text{-}J\text{-}prog P$

shows $eval_1\text{-}eval: P \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle$

$\Rightarrow (\bigwedge Vs ls. \llbracket fv e \subseteq set Vs; l \subseteq_m [Vs \mapsto ls]; size Vs + max\text{-}vars e \leq size ls \rrbracket$

$\Rightarrow \exists ls'. compP_1 P \vdash_1 \langle compE_1 Vs e, (h, ls) \rangle \Rightarrow \langle fin_1 e', (h', ls') \rangle \wedge l' \subseteq_m [Vs \mapsto ls']$

and $evals_1\text{-}evals: P \vdash \langle es, (h, l) \rangle [\Rightarrow] \langle es', (h', l') \rangle$

$\Rightarrow (\bigwedge Vs ls. \llbracket fvs es \subseteq set Vs; l \subseteq_m [Vs \mapsto ls]; size Vs + max\text{-}varss es \leq size ls \rrbracket$

$\Rightarrow \exists ls'. compP_1 P \vdash_1 \langle compEs_1 Vs es, (h, ls) \rangle [\Rightarrow] \langle compEs_1 Vs es', (h', ls') \rangle \wedge l' \subseteq_m [Vs \mapsto ls'] \langle proof \rangle$

5.5.2 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 $compE_1\text{-}pres\text{-}wt: \bigwedge Vs Ts U.$

$\llbracket P, [Vs \mapsto Ts] \vdash e :: U; size Ts = size Vs \rrbracket$

$\Rightarrow compP f P, Ts \vdash_1 compE_1 Vs e :: U$

and $\bigwedge Vs Ts Us.$

$\llbracket P, [Vs \mapsto Ts] \vdash es [::] Us; size Ts = size Vs \rrbracket$

$\Rightarrow compP f P, Ts \vdash_1 compEs_1 Vs es [::] Us \langle proof \rangle$

and the correct block numbering:

lemma $\mathcal{B}: \bigwedge Vs n. size Vs = n \Rightarrow \mathcal{B} (compE_1 Vs e) n$

and $\mathcal{B}s: \bigwedge Vs n. size Vs = n \Rightarrow \mathcal{B}s (compEs_1 Vs es) n \langle proof \rangle$

The main complication is preservation of definite assignment \mathcal{D} .

lemma $image\text{-}last\text{-}index: A \subseteq set(xs@[x]) \Rightarrow last\text{-}index (xs @ [x]) ' A =$

$(if x \in A then insert (size xs) (last\text{-}index xs ' (A - \{x\})) else last\text{-}index xs ' A) \langle proof \rangle$

lemma $A\text{-}compE_1\text{-}None[simp]:$

$\bigwedge Vs. \mathcal{A} e = None \Rightarrow \mathcal{A} (compE_1 Vs e) = None$

and $\bigwedge Vs. \mathcal{A}s es = None \Rightarrow \mathcal{A}s (compEs_1 Vs es) = None \langle proof \rangle$

lemma $A\text{-}compE_1:$

$\bigwedge A Vs. \llbracket \mathcal{A} e = [A]; fv e \subseteq set Vs \rrbracket \Rightarrow \mathcal{A} (compE_1 Vs e) = [last\text{-}index Vs ' A]$

and $\bigwedge A Vs. \llbracket \mathcal{A}s es = [A]; fvs es \subseteq set Vs \rrbracket \Rightarrow \mathcal{A}s (compEs_1 Vs es) = [last\text{-}index Vs ' A] \langle proof \rangle$

lemma $D\text{-}None[iff]: \mathcal{D} (e::'a exp) None \text{ and } [iff]: \mathcal{D}s (es::'a exp list) None \langle proof \rangle$

lemma $D\text{-}last\text{-}index\text{-}compE_1:$

$\bigwedge A Vs. \llbracket A \subseteq set Vs; fv e \subseteq set Vs \rrbracket \Rightarrow$

$\mathcal{D} e [A] \Rightarrow \mathcal{D} (compE_1 Vs e) [last\text{-}index Vs ' A]$

and $\bigwedge A Vs. \llbracket A \subseteq set Vs; fvs es \subseteq set Vs \rrbracket \Rightarrow$

$\mathcal{D}s es [A] \Rightarrow \mathcal{D}s (compEs_1 Vs es) [last\text{-}index Vs ' A] \langle proof \rangle$

lemma $last\text{-}index\text{-}image\text{-}set: distinct xs \Rightarrow last\text{-}index xs ' set xs = \{..<size xs\} \langle proof \rangle$

lemma $D\text{-}compE_1:$

$\llbracket \mathcal{D} e [set Vs]; fv e \subseteq set Vs; distinct Vs \rrbracket \Rightarrow \mathcal{D} (compE_1 Vs e) [\{..<length Vs\}] \langle proof \rangle$

lemma $D\text{-}compE_1':$

assumes $\mathcal{D} e \in [set(V\#Vs)]$ **and** $fv e \subseteq set(V\#Vs)$ **and** $distinct(V\#Vs)$
shows $\mathcal{D} (compE_1 (V\#Vs) e) [\{..length Vs\}] \langle proof \rangle$

lemma $compP_1\text{-pres-wf}: wf\text{-}J\text{-prog } P \implies wf\text{-}J_1\text{-prog } (compP_1 P) \langle proof \rangle$

end

5.6 Compilation Stage 2

theory *Compiler2*

imports *PCompiler J1 ../JVM/JVMExec*

begin

primrec $compE_2 :: expr_1 \Rightarrow instr\ list$
and $compEs_2 :: expr_1\ list \Rightarrow instr\ list$ **where**
 $compE_2 (new\ C) = [New\ C]$
 $compE_2 (Cast\ C\ e) = compE_2\ e\ @\ [Checkcast\ C]$
 $compE_2 (Val\ v) = [Push\ v]$
 $compE_2 (e_1 \ll bop \gg e_2) = compE_2\ e_1\ @\ compE_2\ e_2\ @\ (case\ bop\ of\ Eq \Rightarrow [CmpEq] | Add \Rightarrow [IAdd])$
 $compE_2 (Var\ i) = [Load\ i]$
 $compE_2 (i:=e) = compE_2\ e\ @\ [Store\ i,\ Push\ Unit]$
 $compE_2 (e \cdot F\{D\}) = compE_2\ e\ @\ [Getfield\ F\ D]$
 $compE_2 (e_1 \cdot F\{D\} := e_2) = compE_2\ e_1\ @\ compE_2\ e_2\ @\ [Putfield\ F\ D,\ Push\ Unit]$
 $compE_2 (e \cdot M(es)) = compE_2\ e\ @\ compEs_2\ es\ @\ [Invoke\ M\ (size\ es)]$
 $compE_2 (\{i:T; e\}) = compE_2\ e$
 $compE_2 (e_1;;e_2) = compE_2\ e_1\ @\ [Pop]\ @\ compE_2\ e_2$
 $compE_2 (if\ (e)\ e_1\ else\ e_2) = (let\ cnd = compE_2\ e; thn = compE_2\ e_1; els = compE_2\ e_2; test = IfFalse\ (int(size\ thn + 2)); thnex = Goto\ (int(size\ els + 1)) in\ cnd\ @\ [test]\ @\ thn\ @\ [thnex]\ @\ els)$
 $compE_2 (while\ (e)\ c) = (let\ cnd = compE_2\ e; bdy = compE_2\ c; test = IfFalse\ (int(size\ bdy + 3)); loop = Goto\ (-int(size\ bdy + size\ cnd + 2)) in\ cnd\ @\ [test]\ @\ bdy\ @\ [Pop]\ @\ [loop]\ @\ [Push\ Unit])$
 $compE_2 (throw\ e) = compE_2\ e\ @\ [instr.Throw]$
 $compE_2 (try\ e_1\ catch(C\ i)\ e_2) = (let\ catch = compE_2\ e_2 in\ compE_2\ e_1\ @\ [Goto\ (int(size\ catch)+2),\ Store\ i]\ @\ catch)$
 $compEs_2 [] = []$
 $compEs_2 (e\#es) = compE_2\ e\ @\ compEs_2\ es$

Compilation of exception table. Is given start address of code to compute absolute addresses necessary in exception table.

primrec $compxE_2 :: expr_1 \Rightarrow pc \Rightarrow nat \Rightarrow ex\ table$

and $\text{compxE}_2 :: \text{expr}_1 \text{ list} \Rightarrow \text{pc} \Rightarrow \text{nat} \Rightarrow \text{ex-table}$ **where**
 $\text{compxE}_2 (\text{new } C) \text{ pc } d = []$
 $\text{compxE}_2 (\text{Cast } C \ e) \text{ pc } d = \text{compxE}_2 \ e \ \text{pc } d$
 $\text{compxE}_2 (\text{Val } v) \text{ pc } d = []$
 $\text{compxE}_2 (e_1 \ll \text{bop} \gg e_2) \text{ pc } d =$
 $\quad \text{compxE}_2 \ e_1 \ \text{pc } d @ \text{compxE}_2 \ e_2 \ (\text{pc} + \text{size}(\text{compE}_2 \ e_1)) \ (d+1)$
 $\text{compxE}_2 (\text{Var } i) \text{ pc } d = []$
 $\text{compxE}_2 (i := e) \text{ pc } d = \text{compxE}_2 \ e \ \text{pc } d$
 $\text{compxE}_2 (e \cdot F\{D\}) \text{ pc } d = \text{compxE}_2 \ e \ \text{pc } d$
 $\text{compxE}_2 (e_1 \cdot F\{D\} := e_2) \text{ pc } d =$
 $\quad \text{compxE}_2 \ e_1 \ \text{pc } d @ \text{compxE}_2 \ e_2 \ (\text{pc} + \text{size}(\text{compE}_2 \ e_1)) \ (d+1)$
 $\text{compxE}_2 (e \cdot M(es)) \text{ pc } d =$
 $\quad \text{compxE}_2 \ e \ \text{pc } d @ \text{compxE}_2 \ es \ (\text{pc} + \text{size}(\text{compE}_2 \ e)) \ (d+1)$
 $\text{compxE}_2 (\{i:T; e\}) \text{ pc } d = \text{compxE}_2 \ e \ \text{pc } d$
 $\text{compxE}_2 (e_1 ;; e_2) \text{ pc } d =$
 $\quad \text{compxE}_2 \ e_1 \ \text{pc } d @ \text{compxE}_2 \ e_2 \ (\text{pc} + \text{size}(\text{compE}_2 \ e_1) + 1) \ d$
 $\text{compxE}_2 (\text{if } (e) \ e_1 \ \text{else } e_2) \text{ pc } d =$
 $\quad (\text{let } \text{pc}_1 = \text{pc} + \text{size}(\text{compE}_2 \ e) + 1;$
 $\quad \quad \text{pc}_2 = \text{pc}_1 + \text{size}(\text{compE}_2 \ e_1) + 1$
 $\quad \text{in } \text{compxE}_2 \ e \ \text{pc } d @ \text{compxE}_2 \ e_1 \ \text{pc}_1 \ d @ \text{compxE}_2 \ e_2 \ \text{pc}_2 \ d)$
 $\text{compxE}_2 (\text{while } (b) \ e) \text{ pc } d =$
 $\quad \text{compxE}_2 \ b \ \text{pc } d @ \text{compxE}_2 \ e \ (\text{pc} + \text{size}(\text{compE}_2 \ b) + 1) \ d$
 $\text{compxE}_2 (\text{throw } e) \text{ pc } d = \text{compxE}_2 \ e \ \text{pc } d$
 $\text{compxE}_2 (\text{try } e_1 \ \text{catch}(C \ i) \ e_2) \text{ pc } d =$
 $\quad (\text{let } \text{pc}_1 = \text{pc} + \text{size}(\text{compE}_2 \ e_1)$
 $\quad \text{in } \text{compxE}_2 \ e_1 \ \text{pc } d @ \text{compxE}_2 \ e_2 \ (\text{pc}_1 + 2) \ d @ [(pc, \text{pc}_1, C, \text{pc}_1 + 1, d)])$
 $\text{compxE}_2 [] \ \text{pc } d = []$
 $\text{compxE}_2 (e \# es) \text{ pc } d = \text{compxE}_2 \ e \ \text{pc } d @ \text{compxE}_2 \ es \ (\text{pc} + \text{size}(\text{compE}_2 \ e)) \ (d+1)$

primrec $\text{max-stack} :: \text{expr}_1 \Rightarrow \text{nat}$

and $\text{max-stacks} :: \text{expr}_1 \text{ list} \Rightarrow \text{nat}$ **where**
 $\text{max-stack} (\text{new } C) = 1$
 $\text{max-stack} (\text{Cast } C \ e) = \text{max-stack } e$
 $\text{max-stack} (\text{Val } v) = 1$
 $\text{max-stack} (e_1 \ll \text{bop} \gg e_2) = \max (\text{max-stack } e_1) (\text{max-stack } e_2) + 1$
 $\text{max-stack} (\text{Var } i) = 1$
 $\text{max-stack} (i := e) = \text{max-stack } e$
 $\text{max-stack} (e \cdot F\{D\}) = \text{max-stack } e$
 $\text{max-stack} (e_1 \cdot F\{D\} := e_2) = \max (\text{max-stack } e_1) (\text{max-stack } e_2) + 1$
 $\text{max-stack} (e \cdot M(es)) = \max (\text{max-stack } e) (\text{max-stacks } es) + 1$
 $\text{max-stack} (\{i:T; e\}) = \text{max-stack } e$
 $\text{max-stack} (e_1 ;; e_2) = \max (\text{max-stack } e_1) (\text{max-stack } e_2)$
 $\text{max-stack} (\text{if } (e) \ e_1 \ \text{else } e_2) =$
 $\quad \max (\text{max-stack } e) (\max (\text{max-stack } e_1) (\text{max-stack } e_2))$
 $\text{max-stack} (\text{while } (e) \ c) = \max (\text{max-stack } e) (\text{max-stack } c)$
 $\text{max-stack} (\text{throw } e) = \text{max-stack } e$
 $\text{max-stack} (\text{try } e_1 \ \text{catch}(C \ i) \ e_2) = \max (\text{max-stack } e_1) (\text{max-stack } e_2)$
 $\text{max-stacks} [] = 0$
 $\text{max-stacks} (e \# es) = \max (\text{max-stack } e) (1 + \text{max-stacks } es)$

lemma $\text{max-stack1}: 1 \leq \text{max-stack } e \langle \text{proof} \rangle$

definition $compMb_2 :: expr_1 \Rightarrow jvm-method$

where

$compMb_2 \equiv \lambda body.$
 $let\ ins = compE_2\ body\ @\ [Return];$
 $xt = compxE_2\ body\ 0\ 0$
 $in\ (max-stack\ body,\ max-vars\ body,\ ins,\ xt)$

definition $compP_2 :: J_1-prog \Rightarrow jvm-prog$

where

$compP_2 \equiv compP\ compMb_2$

lemma $compMb_2\ [simp]:$

$compMb_2\ e = (max-stack\ e,\ max-vars\ e,\ compE_2\ e\ @\ [Return],\ compxE_2\ e\ 0\ 0)\langle proof \rangle$

end

5.7 Correctness of Stage 2

theory *Correctness2*

imports *HOL-Library.Sublist Compiler2*

begin

5.7.1 Instruction sequences

How to select individual instructions and subsequences of instructions from a program given the class, method and program counter.

definition $before :: jvm-prog \Rightarrow cname \Rightarrow mname \Rightarrow nat \Rightarrow instr\ list \Rightarrow bool$

$((-, -, -, - / \triangleright -) [51, 0, 0, 0, 51] 50)$ **where**
 $P, C, M, pc \triangleright is \longleftrightarrow prefix\ is\ (drop\ pc\ (instrs-of\ P\ C\ M))$

definition $at :: jvm-prog \Rightarrow cname \Rightarrow mname \Rightarrow nat \Rightarrow instr \Rightarrow bool$

$((-, -, -, - / \triangleright -) [51, 0, 0, 0, 51] 50)$ **where**
 $P, C, M, pc \triangleright i \longleftrightarrow (\exists is.\ drop\ pc\ (instrs-of\ P\ C\ M) = i\ \#\ is)$

lemma $[simp]: P, C, M, pc \triangleright [] \langle proof \rangle$

lemma $[simp]: P, C, M, pc \triangleright (i\ \#\ is) = (P, C, M, pc \triangleright i \wedge P, C, M, pc + 1 \triangleright is) \langle proof \rangle$

lemma $[simp]: P, C, M, pc \triangleright (is_1\ @\ is_2) = (P, C, M, pc \triangleright is_1 \wedge P, C, M, pc + size\ is_1 \triangleright is_2) \langle proof \rangle$

lemma $[simp]: P, C, M, pc \triangleright i \implies instrs-of\ P\ C\ M\ !\ pc = i \langle proof \rangle$

lemma $beforeM:$

$P \vdash C\ sees\ M: Ts \rightarrow T = body\ in\ D \implies$
 $compP_2\ P, D, M, 0 \triangleright compE_2\ body\ @\ [Return] \langle proof \rangle$

This lemma executes a single instruction by rewriting:

lemma $[simp]:$

$P, C, M, pc \triangleright instr \implies$
 $(P \vdash (None, h, (vs, ls, C, M, pc) \# frs) -jvm \rightarrow \sigma') =$
 $((None, h, (vs, ls, C, M, pc) \# frs) = \sigma' \vee$
 $(\exists \sigma.\ exec(P, (None, h, (vs, ls, C, M, pc) \# frs)) = Some\ \sigma \wedge P \vdash \sigma -jvm \rightarrow \sigma')) \langle proof \rangle$

5.7.2 Exception tables

definition $pcs :: ex\text{-}table \Rightarrow nat\ set$

where

$$pcs\ xt \equiv \bigcup (f,t,C,h,d) \in set\ xt. \{f ..< t\}$$

lemma $pcs\text{-}subset$:

shows $\bigwedge pc\ d. pcs(compxE_2\ e\ pc\ d) \subseteq \{pc..<pc+size(compE_2\ e)\}$

and $\bigwedge pc\ d. pcs(compxEs_2\ es\ pc\ d) \subseteq \{pc..<pc+size(compEs_2\ es)\}$ $\langle proof \rangle$

lemma $[simp]$: $pcs\ [] = \{\}$ $\langle proof \rangle$

lemma $[simp]$: $pcs\ (x\#\!xt) = \{fst\ x ..< fst(snd\ x)\} \cup pcs\ xt$ $\langle proof \rangle$

lemma $[simp]$: $pcs(xt_1\ @\ xt_2) = pcs\ xt_1 \cup pcs\ xt_2$ $\langle proof \rangle$

lemma $[simp]$: $pc < pc_0 \vee pc_0 + size(compE_2\ e) \leq pc \Longrightarrow pc \notin pcs(compxE_2\ e\ pc_0\ d)$ $\langle proof \rangle$

lemma $[simp]$: $pc < pc_0 \vee pc_0 + size(compEs_2\ es) \leq pc \Longrightarrow pc \notin pcs(compxEs_2\ es\ pc_0\ d)$ $\langle proof \rangle$

lemma $[simp]$: $pc_1 + size(compE_2\ e_1) \leq pc_2 \Longrightarrow pcs(compxE_2\ e_1\ pc_1\ d_1) \cap pcs(compxE_2\ e_2\ pc_2\ d_2) = \{\}$ $\langle proof \rangle$

lemma $[simp]$: $pc_1 + size(compE_2\ e) \leq pc_2 \Longrightarrow pcs(compxE_2\ e\ pc_1\ d_1) \cap pcs(compxEs_2\ es\ pc_2\ d_2) = \{\}$ $\langle proof \rangle$

lemma $[simp]$:

$pc \notin pcs\ xt_0 \Longrightarrow match\text{-}ex\text{-}table\ P\ C\ pc\ (xt_0\ @\ xt_1) = match\text{-}ex\text{-}table\ P\ C\ pc\ xt_1$ $\langle proof \rangle$

lemma $[simp]$: $\llbracket x \in set\ xt; pc \notin pcs\ xt \rrbracket \Longrightarrow \neg matches\text{-}ex\text{-}entry\ P\ D\ pc\ x$ $\langle proof \rangle$

lemma $[simp]$:

assumes $xe: xe \in set(compxE_2\ e\ pc\ d)$ **and** *outside*: $pc' < pc \vee pc + size(compE_2\ e) \leq pc'$

shows $\neg matches\text{-}ex\text{-}entry\ P\ C\ pc'\ xe$ $\langle proof \rangle$

lemma $[simp]$:

assumes $xe: xe \in set(compxEs_2\ es\ pc\ d)$ **and** *outside*: $pc' < pc \vee pc + size(compEs_2\ es) \leq pc'$

shows $\neg matches\text{-}ex\text{-}entry\ P\ C\ pc'\ xe$ $\langle proof \rangle$

lemma $match\text{-}ex\text{-}table\text{-}app$ $[simp]$:

$\forall xte \in set\ xt_1. \neg matches\text{-}ex\text{-}entry\ P\ D\ pc\ xte \Longrightarrow$

$match\text{-}ex\text{-}table\ P\ D\ pc\ (xt_1\ @\ xt) = match\text{-}ex\text{-}table\ P\ D\ pc\ xt$ $\langle proof \rangle$

lemma $[simp]$:

$\forall x \in set\ xtab. \neg matches\text{-}ex\text{-}entry\ P\ C\ pc\ x \Longrightarrow$

$match\text{-}ex\text{-}table\ P\ C\ pc\ xtab = None$ $\langle proof \rangle$

lemma $match\text{-}ex\text{-}entry$:

$matches\text{-}ex\text{-}entry\ P\ C\ pc\ (start, end, catch\text{-}type, handler) =$

$(start \leq pc \wedge pc < end \wedge P \vdash C \preceq^* catch\text{-}type)$ $\langle proof \rangle$

definition $caught :: jvm\text{-}prog \Rightarrow pc \Rightarrow heap \Rightarrow addr \Rightarrow ex\text{-}table \Rightarrow bool$ **where**

$caught\ P\ pc\ h\ a\ xt \longleftrightarrow$

$(\exists entry \in set\ xt. matches\text{-}ex\text{-}entry\ P\ (cname\text{-}of\ h\ a)\ pc\ entry)$

definition *beforex* :: *jvm-prog* \Rightarrow *cname* \Rightarrow *mname* \Rightarrow *ex-table* \Rightarrow *nat set* \Rightarrow *nat* \Rightarrow *bool*
 $((2,-,-,- \triangleright / - /' / -,-) [51,0,0,0,0,51] 50)$ **where**

$P, C, M \triangleright xt / I, d \iff$

$(\exists xt_0 xt_1. \text{ex-table-of } P C M = xt_0 @ xt @ xt_1 \wedge \text{pcs } xt_0 \cap I = \{\} \wedge \text{pcs } xt \subseteq I \wedge$
 $(\forall pc \in I. \forall C pc' d'. \text{match-ex-table } P C pc xt_1 = [(pc', d')] \longrightarrow d' \leq d))$

definition *dummyx* :: *jvm-prog* \Rightarrow *cname* \Rightarrow *mname* \Rightarrow *ex-table* \Rightarrow *nat set* \Rightarrow *nat* \Rightarrow *bool* $((2,-,-,- \triangleright / - /' / -,-) [51,0,0,0,0,51] 50)$ **where**

$P, C, M \triangleright xt / I, d \iff P, C, M \triangleright xt / I, d$

abbreviation

*beforex*₀ *P C M d I xt xt₀ xt₁*

$\equiv \text{ex-table-of } P C M = xt_0 @ xt @ xt_1 \wedge \text{pcs } xt_0 \cap I = \{\}$
 $\wedge \text{pcs } xt \subseteq I \wedge (\forall pc \in I. \forall C pc' d'. \text{match-ex-table } P C pc xt_1 = [(pc', d')] \longrightarrow d' \leq d)$

lemma *beforex-beforex₀-eq*:

$P, C, M \triangleright xt / I, d \equiv \exists xt_0 xt_1. \text{beforex}_0 P C M d I xt xt_0 xt_1$
 $\langle \text{proof} \rangle$

lemma *beforexD1*: $P, C, M \triangleright xt / I, d \implies \text{pcs } xt \subseteq I \langle \text{proof} \rangle$

lemma *beforex-mono*: $\llbracket P, C, M \triangleright xt / I, d'; d' \leq d \rrbracket \implies P, C, M \triangleright xt / I, d \langle \text{proof} \rangle$

lemma [*simp*]: $P, C, M \triangleright xt / I, d \implies P, C, M \triangleright xt / I, \text{Suc } d \langle \text{proof} \rangle$

lemma *beforex-append*[*simp*]:

$\text{pcs } xt_1 \cap \text{pcs } xt_2 = \{\} \implies$
 $P, C, M \triangleright xt_1 @ xt_2 / I, d =$
 $(P, C, M \triangleright xt_1 / I - \text{pcs } xt_2, d \wedge P, C, M \triangleright xt_2 / I - \text{pcs } xt_1, d \wedge P, C, M \triangleright xt_1 @ xt_2 / I, d) \langle \text{proof} \rangle$

lemma *beforex-appendD1*:

assumes *bx*: $P, C, M \triangleright xt_1 @ xt_2 @ [(f, t, D, h, d)] / I, d$
and *pcs*: $\text{pcs } xt_1 \subseteq J$ **and** *JI*: $J \subseteq I$ **and** *Jpcs*: $J \cap \text{pcs } xt_2 = \{\}$
shows $P, C, M \triangleright xt_1 / J, d \langle \text{proof} \rangle$

lemma *beforex-appendD2*:

assumes *bx*: $P, C, M \triangleright xt_1 @ xt_2 @ [(f, t, D, h, d)] / I, d$
and *pcs*: $\text{pcs } xt_2 \subseteq J$ **and** *JI*: $J \subseteq I$ **and** *Jpcs*: $J \cap \text{pcs } xt_1 = \{\}$
shows $P, C, M \triangleright xt_2 / J, d \langle \text{proof} \rangle$

lemma *beforexM*:

$P \vdash C \text{ sees } M: Ts \rightarrow T = \text{body in } D \implies$
 $\text{compP}_2 P, D, M \triangleright \text{compxE}_2 \text{ body } 0 0 / \{.. < \text{size}(\text{compE}_2 \text{ body})\}, 0 \langle \text{proof} \rangle$

lemma *match-ex-table-SomeD2*:

assumes *met*: $\text{match-ex-table } P D pc (\text{ex-table-of } P C M) = [(pc', d')]$
and *bx*: $P, C, M \triangleright xt / I, d$
and *nmet*: $\forall x \in \text{set } xt. \neg \text{matches-ex-entry } P D pc x$ **and** *pcI*: $pc \in I$
shows $d' \leq d \langle \text{proof} \rangle$

lemma *match-ex-table-SomeD1*:

$\llbracket \text{match-ex-table } P D pc (\text{ex-table-of } P C M) = [(pc', d')] \rrbracket;$

$$P, C, M \triangleright xt / I, d; pc \in I; pc \notin pcs\ xt \] \Longrightarrow d' \leq d \langle proof \rangle$$

5.7.3 The correctness proof

definition

handle :: *jvm-prog* \Rightarrow *cname* \Rightarrow *mname* \Rightarrow *addr* \Rightarrow *heap* \Rightarrow *val list* \Rightarrow *val list* \Rightarrow *nat* \Rightarrow *frame list*
 \Rightarrow *jvm-state* **where**

handle *P C M a h vs ls pc frs* = *find-handler* *P a h ((vs, ls, C, M, pc) # frs)*

lemma handle-Cons:

$\llbracket P, C, M \triangleright xt / I, d; d \leq size\ vs; pc \in I;$
 $\forall x \in set\ xt. \neg matches\ ex\ entry\ P\ (cname\ of\ h\ xa)\ pc\ x \rrbracket \Longrightarrow$
handle *P C M xa h (v # vs) ls pc frs* = *handle* *P C M xa h vs ls pc frs* $\langle proof \rangle$

lemma handle-append:

assumes *bx*: *P, C, M* \triangleright *xt / I, d* **and** *d*: *d* \leq *size vs*

and *pcI*: *pc* \in *I* **and** *pc-not*: *pc* \notin *pcs xt*

shows *handle* *P C M xa h (ws @ vs) ls pc frs* = *handle* *P C M xa h vs ls pc frs* $\langle proof \rangle$

lemma *aux-isin[simp]*: $\llbracket B \subseteq A; a \in B \rrbracket \Longrightarrow a \in A \langle proof \rangle$

lemma *fixes* *P*₁ **defines** [*simp*]: *P* \equiv *comp**P*₂ *P*₁

shows *Jcc*:

*P*₁ $\vdash_1 \langle e, (h_0, ls_0) \rangle \Rightarrow \langle ef, (h_1, ls_1) \rangle \Longrightarrow$
 $(\bigwedge C\ M\ pc\ v\ xa\ vs\ frs\ I.$
 $\llbracket P, C, M, pc \triangleright compE_2\ e; P, C, M \triangleright compxE_2\ e\ pc\ (size\ vs) / I, size\ vs;$
 $\{pc.. < pc + size(compE_2\ e)\} \subseteq I \rrbracket \Longrightarrow$
 $(ef = Val\ v \longrightarrow$
 $P \vdash (None, h_0, (vs, ls_0, C, M, pc) \# frs) \text{ --jvm--}$
 $(None, h_1, (v \# vs, ls_1, C, M, pc + size(compE_2\ e)) \# frs))$
 \wedge
 $(ef = Throw\ xa \longrightarrow$
 $(\exists pc_1. pc \leq pc_1 \wedge pc_1 < pc + size(compE_2\ e) \wedge$
 $\neg caught\ P\ pc_1\ h_1\ xa\ (compxE_2\ e\ pc\ (size\ vs)) \wedge$
 $P \vdash (None, h_0, (vs, ls_0, C, M, pc) \# frs) \text{ --jvm--}$ *handle* *P C M xa h*₁ *vs* *ls*₁ *pc*₁ *frs*)))

and *P*₁ $\vdash_1 \langle es, (h_0, ls_0) \rangle [\Rightarrow] \langle fs, (h_1, ls_1) \rangle \Longrightarrow$

$(\bigwedge C\ M\ pc\ ws\ xa\ es'\ vs\ frs\ I.$
 $\llbracket P, C, M, pc \triangleright compEs_2\ es; P, C, M \triangleright compxEs_2\ es\ pc\ (size\ vs) / I, size\ vs;$
 $\{pc.. < pc + size(compEs_2\ es)\} \subseteq I \rrbracket \Longrightarrow$
 $(fs = map\ Val\ ws \longrightarrow$
 $P \vdash (None, h_0, (vs, ls_0, C, M, pc) \# frs) \text{ --jvm--}$
 $(None, h_1, (rev\ ws\ @\ vs, ls_1, C, M, pc + size(compEs_2\ es)) \# frs))$
 \wedge
 $(fs = map\ Val\ ws\ @\ Throw\ xa\ \# es' \longrightarrow$
 $(\exists pc_1. pc \leq pc_1 \wedge pc_1 < pc + size(compEs_2\ es) \wedge$
 $\neg caught\ P\ pc_1\ h_1\ xa\ (compxEs_2\ es\ pc\ (size\ vs)) \wedge$
 $P \vdash (None, h_0, (vs, ls_0, C, M, pc) \# frs) \text{ --jvm--}$ *handle* *P C M xa h*₁ *vs* *ls*₁ *pc*₁ *frs*))) $\langle proof \rangle$

lemma *atLeast0AtMost[simp]*: $\{0 :: nat..n\} = \{..n\}$
 $\langle proof \rangle$

lemma *atLeast0LessThan[simp]*: $\{0 :: nat..<n\} = \{..<n\}$

$\langle proof \rangle$

fun *exception* :: 'a exp \Rightarrow addr option **where**
exception (Throw a) = Some a
 | *exception* e = None

lemma *comp₂-correct*:

assumes *method*: $P_1 \vdash C$ sees $M:Ts \rightarrow T = \text{body}$ in C

and *eval*: $P_1 \vdash_1 \langle \text{body}, (h, ls) \rangle \Rightarrow \langle e', (h', ls') \rangle$

shows $\text{comp}P_2 P_1 \vdash (\text{None}, h, [([], ls, C, M, 0)]) \text{--jvm--} \rightarrow (\text{exception } e', h', []) \langle proof \rangle$
end

5.8 Combining Stages 1 and 2

theory *Compiler*

imports *Correctness1 Correctness2*

begin

definition *J2JVM* :: J-prog \Rightarrow jvm-prog

where

$J2JVM \equiv \text{comp}P_2 \circ \text{comp}P_1$

theorem *comp-correct*:

assumes *wwf*: wwf-J-prog P

and *method*: $P \vdash C$ sees $M:Ts \rightarrow T = (\text{pns}, \text{body})$ in C

and *eval*: $P \vdash \langle \text{body}, (h, [\text{this}\#\text{pns} \mapsto vs]) \rangle \Rightarrow \langle e', (h', l') \rangle$

and *sizes*: $\text{size } vs = \text{size } \text{pns} + 1$ $\text{size } \text{rest} = \text{max-vars } \text{body}$

shows $J2JVM P \vdash (\text{None}, h, [([], vs@rest, C, M, 0)]) \text{--jvm--} \rightarrow (\text{exception } e', h', []) \langle proof \rangle$

end

5.9 Preservation of Well-Typedness

theory *TypeComp*

imports *Compiler ../BV/BVSpec*

begin

locale *TC0* =

fixes $P :: J_1\text{-prog}$ **and** $\text{mxl} :: \text{nat}$

begin

definition $\text{ty } E e = (\text{THE } T. P, E \vdash_1 e :: T)$

definition $\text{ty}_i E A' = \text{map } (\lambda i. \text{if } i \in A' \wedge i < \text{size } E \text{ then } \text{OK}(E!i) \text{ else } \text{Err}) [0..<\text{mxl}]$

definition $\text{ty}_i' ST E A = (\text{case } A \text{ of } \text{None} \Rightarrow \text{None} \mid _ [A'] \Rightarrow \text{Some}(ST, \text{ty}_i E A))$

definition $\text{after } E A ST e = \text{ty}_i' (\text{ty } E e \# ST) E (A \sqcup A e)$

end

lemma (**in** *TC0*) *ty-def2* [*simp*]: $P, E \vdash_1 e :: T \Longrightarrow \text{ty } E e = T \langle proof \rangle$

lemma (in *TC0*) [*simp*]: $ty_i' ST E None = None\langle proof \rangle$

lemma (in *TC0*) *ty_l-app-diff*[*simp*]:

$$ty_l (E@[T]) (A - \{size E\}) = ty_l E A\langle proof \rangle$$

lemma (in *TC0*) *ty_i'-app-diff*[*simp*]:

$$ty_i' ST (E @ [T]) (A \ominus size E) = ty_i' ST E A\langle proof \rangle$$

lemma (in *TC0*) *ty_l-antimono*:

$$A \subseteq A' \implies P \vdash ty_l E A' [\leq_{\top}] ty_l E A\langle proof \rangle$$

lemma (in *TC0*) *ty_i'-antimono*:

$$A \subseteq A' \implies P \vdash ty_i' ST E [A'] \leq' ty_i' ST E [A]\langle proof \rangle$$

lemma (in *TC0*) *ty_l-env-antimono*:

$$P \vdash ty_l (E@[T]) A [\leq_{\top}] ty_l E A\langle proof \rangle$$

lemma (in *TC0*) *ty_i'-env-antimono*:

$$P \vdash ty_i' ST (E@[T]) A \leq' ty_i' ST E A\langle proof \rangle$$

lemma (in *TC0*) *ty_i'-incr*:

$$P \vdash ty_i' ST (E @ [T]) [insert (size E) A] \leq' ty_i' ST E [A]\langle proof \rangle$$

lemma (in *TC0*) *ty_l-incr*:

$$P \vdash ty_l (E @ [T]) (insert (size E) A) [\leq_{\top}] ty_l E A\langle proof \rangle$$

lemma (in *TC0*) *ty_l-in-types*:

$$set E \subseteq types P \implies ty_l E A \in nlists\ max (err (types P))\langle proof \rangle$$

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**

$$compT E A ST (new C) = []$$

$$| compT E A ST (Cast C e) =$$

$$compT E A ST e @ [after E A ST e]$$

$$| compT E A ST (Val v) = []$$

$$| compT E A ST (e_1 \ll bop \gg e_2) =$$

$$(let ST_1 = ty E e_1 \# ST; A_1 = A \sqcup \mathcal{A} e_1 \text{ in}$$

$$compT E A ST e_1 @ [after E A ST e_1] @$$

$$compT E A_1 ST_1 e_2 @ [after E A_1 ST_1 e_2])$$

$$| compT E A ST (Var i) = []$$

$$| compT E A ST (i := e) = compT E A ST e @$$

$$[after E A ST e, ty_i' ST E (A \sqcup \mathcal{A} e \sqcup [\{i\}])]$$

$$| compT E A ST (e \cdot F\{D\}) =$$

$$compT E A ST e @ [after E A ST e]$$

$$| compT E A ST (e_1 \cdot F\{D\} := e_2) =$$

$$(let ST_1 = ty E e_1 \# ST; A_1 = A \sqcup \mathcal{A} e_1; A_2 = A_1 \sqcup \mathcal{A} e_2 \text{ in}$$

$$compT E A ST e_1 @ [after E A ST e_1] @$$

$$compT E A_1 ST_1 e_2 @ [after E A_1 ST_1 e_2] @$$

$$[ty_i' ST E A_2])$$

$$| compT E A ST \{i:T; e\} = compT (E@[T]) (A \ominus i) ST e$$

$$| compT E A ST (e_1 ;; e_2) =$$

$$(let A_1 = A \sqcup \mathcal{A} e_1 \text{ in}$$

$$compT E A ST e_1 @ [after E A ST e_1, ty_i' ST E A_1] @$$

$compT E A_1 ST e_2)$
 $| compT E A ST (if (e) e_1 else e_2) =$
 $(let A_0 = A \sqcup \mathcal{A} e; \tau = ty_i' ST E A_0 in$
 $compT E A ST e @ [after E A ST e, \tau] @$
 $compT E A_0 ST e_1 @ [after E A_0 ST e_1, \tau] @$
 $compT E A_0 ST e_2)$
 $| compT E A ST (while (e) c) =$
 $(let A_0 = A \sqcup \mathcal{A} e; A_1 = A_0 \sqcup \mathcal{A} c; \tau = ty_i' ST E A_0 in$
 $compT E A ST e @ [after E A ST e, \tau] @$
 $compT E A_0 ST c @ [after E A_0 ST c, ty_i' ST E A_1, ty_i' ST E A_0])$
 $| compT E A ST (throw e) = compT E A ST e @ [after E A ST e]$
 $| compT E A ST (e.M(es)) =$
 $compT E A ST e @ [after E A ST e] @$
 $compTs E (A \sqcup \mathcal{A} e) (ty E e \# ST) es$
 $| compT E A ST (try e_1 catch (C i) e_2) =$
 $compT E A ST e_1 @ [after E A ST e_1] @$
 $[ty_i' (Class C \# ST) E A, ty_i' ST (E@[Class C]) (A \sqcup [\{i\}])] @$
 $compT (E@[Class C]) (A \sqcup [\{i\}]) ST e_2$
 $| compTs E A ST [] = []$
 $| compTs E A ST (e \# es) = compT E A ST e @ [after E A ST e] @$
 $compTs E (A \sqcup (\mathcal{A} e)) (ty E e \# ST) es$

definition $compT_a :: ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1 \Rightarrow ty_i'\ list$ **where**
 $compT_a E A ST e = compT E A ST e @ [after E A ST e]$

end

lemma $compE_2\text{-not-Nil}[simp]: compE_2 e \neq [] \langle proof \rangle$

lemma **(in TC1)** $compT\text{-sizes}[simp]:$

shows $\bigwedge E A ST. size(compT E A ST e) = size(compE_2 e) - 1$

and $\bigwedge E A ST. size(compTs E A ST es) = size(compEs_2 es) \langle proof \rangle$

lemma **(in TC1)** $[simp]: \bigwedge ST E. [\tau] \notin set (compT E None ST e)$

and $[simp]: \bigwedge ST E. [\tau] \notin set (compTs E None ST es) \langle proof \rangle$

lemma **(in TC0)** $pair\text{-eq-ty}_i'\text{-conv}:$

$([(ST, LT)] = ty_i' ST_0 E A) =$

$(case A of None \Rightarrow False \mid Some A \Rightarrow (ST = ST_0 \wedge LT = ty_l E A)) \langle proof \rangle$

lemma **(in TC0)** $pair\text{-conv-ty}_i':$

$[(ST, ty_l E A)] = ty_i' ST E [A] \langle proof \rangle$

lemma **(in TC1)** $compT\text{-LT-prefix}:$

$\bigwedge E A ST_0. \llbracket [(ST,LT)] \in set(compT E A ST_0 e); \mathcal{B} e (size E) \rrbracket$

$\implies P \vdash [(ST,LT)] \leq' ty_i' ST E A$

and

$\bigwedge E A ST_0. \llbracket [(ST,LT)] \in set(compTs E A ST_0 es); \mathcal{B} es (size E) \rrbracket$

$\implies P \vdash [(ST,LT)] \leq' ty_i' ST E A \langle proof \rangle$

lemma $[iff]: OK None \in states P\ mxs\ m\!x\!l \langle proof \rangle$

lemma **(in TC0)** $after\text{-in-states}:$

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 $stack: size ST + max\text{-stack } e \leq m\!x\!s$

shows OK (after $E A ST e$) \in states P mxs $m\lambda l$ (proof)

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 $m\lambda l$ (proof)

lemma is -class-type-aux: is -class $P C \implies is$ -type P (Class C) (proof)

theorem (in $TC1$) $compT$ -states:

assumes wf : wf -prog $p P$

shows $\bigwedge E T A ST$.

$\llbracket P, E \vdash_1 e :: T; set E \subseteq types P; set ST \subseteq types P;$
 $size ST + max-stack e \leq mxs; size E + max-vars e \leq m\lambda l \rrbracket$
 $\implies OK$ ' $set(compT E A ST e) \subseteq$ states P mxs $m\lambda l$

and $\bigwedge E Ts A ST$.

$\llbracket P, E \vdash_1 es[::]Ts; set E \subseteq types P; set ST \subseteq types P;$
 $size ST + max-stacks es \leq mxs; size E + max-varss es \leq m\lambda l \rrbracket$
 $\implies OK$ ' $set(compTs E A ST es) \subseteq$ states P mxs $m\lambda l$ (proof)

definition $shift :: nat \Rightarrow ex$ -table $\Rightarrow ex$ -table

where

$shift n xt \equiv map$ (λ (from,to, C ,handler,depth). (from+n,to+n, C ,handler+n,depth)) xt

lemma [simp]: $shift 0 xt = xt$ (proof)

lemma [simp]: $shift n [] = []$ (proof)

lemma [simp]: $shift n (xt_1 @ xt_2) = shift n xt_1 @ shift n xt_2$ (proof)

lemma [simp]: $shift m (shift n xt) = shift (m+n) xt$ (proof)

lemma [simp]: $pcs (shift n xt) = \{pc+n | pc. pc \in pcs xt\}$ (proof)

lemma $shift$ - $compxE_2$:

shows $\bigwedge pc pc' d. shift pc (compxE_2 e pc' d) = compxE_2 e (pc' + pc) d$

and $\bigwedge pc pc' d. shift pc (compxEs_2 es pc' d) = compxEs_2 es (pc' + pc) d$ (proof)

lemma $compxE_2$ -size-convs[simp]:

shows $n \neq 0 \implies compxE_2 e n d = shift n (compxE_2 e 0 d)$

and $n \neq 0 \implies compxEs_2 es n d = shift n (compxEs_2 es 0 d)$ (proof)

locale $TC2 = TC1$ +

fixes $T_r :: ty$ **and** $mxs :: pc$

begin

definition

wt -instrs $:: instr$ list $\Rightarrow ex$ -table $\Rightarrow ty_i'$ list $\Rightarrow bool$

((\vdash -, - /[::] / -) [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 ((-, -, \vdash / -, - /[::] / -) [50,50,50,50,50,51] 50)

lemma (in $TC2$) [simp]: $\tau s \neq [] \implies \vdash [], [] [::] \tau s$ (proof)

lemma [simp]: $eff i P pc et None = []$ (proof)

lemma wt -instr-appR:

$\llbracket P, T, m, mpc, xt \vdash is!pc, pc :: \tau s;$

$pc < \text{size } is; \text{size } is < \text{size } \tau s; mpc \leq \text{size } \tau s; mpc \leq mpc']$
 $\implies P, T, m, mpc', xt \vdash is!pc, pc :: \tau s @ \tau s' \langle \text{proof} \rangle$

lemma *relevant-entries-shift* [simp]:

relevant-entries $P \ i \ (pc+n) \ (\text{shift } n \ xt) = \text{shift } n \ (\text{relevant-entries } P \ i \ pc \ xt) \langle \text{proof} \rangle$

lemma [simp]:

xcpt-eff $i \ P \ (pc+n) \ \tau \ (\text{shift } n \ xt) =$
 $\text{map } (\lambda(pc, \tau). (pc + n, \tau)) \ (\text{xcpt-eff } i \ P \ pc \ \tau \ xt) \langle \text{proof} \rangle$

lemma [simp]:

app_i $(i, P, pc, m, T, \tau) \implies$
 $\text{eff } i \ P \ (pc+n) \ (\text{shift } n \ xt) \ (\text{Some } \tau) =$
 $\text{map } (\lambda(pc, \tau). (pc+n, \tau)) \ (\text{eff } i \ P \ pc \ xt \ (\text{Some } \tau)) \langle \text{proof} \rangle$

lemma [simp]:

xcpt-app $i \ P \ (pc+n) \ \text{mxs} \ (\text{shift } n \ xt) \ \tau = \text{xcpt-app } i \ P \ pc \ \text{mxs} \ xt \ \tau \langle \text{proof} \rangle$

lemma *wt-instr-appL*:

assumes $P, T, m, mpc, xt \vdash i, pc :: \tau s$ **and** $pc < \text{size } \tau s$ **and** $mpc \leq \text{size } \tau s$
shows $P, T, m, mpc + \text{size } \tau s', \text{shift } (\text{size } \tau s') \ xt \vdash i, pc + \text{size } \tau s' :: \tau s' @ \tau s \langle \text{proof} \rangle$

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*: $pc < \text{size } \tau s + 1$ **and** *mpcu*: $mpc \leq \text{size } \tau s + 1$
shows $P, T, m, mpc, [] \vdash i, pc :: \tau \# \tau s \langle \text{proof} \rangle$

lemma *wt-instr-append*:

assumes *wti*: $P, T, m, mpc - \text{size } \tau s', [] \vdash i, pc - \text{size } \tau s' :: \tau s$
and *pcl*: $\text{size } \tau s' \leq pc$ **and** *mpcl*: $\text{size } \tau s' \leq mpc$
and *pcu*: $pc < \text{size } \tau s + \text{size } \tau s'$ **and** *mpcu*: $mpc \leq \text{size } \tau s + \text{size } \tau s'$
shows $P, T, m, mpc, [] \vdash i, pc :: \tau s' @ \tau s \langle \text{proof} \rangle$

lemma *xcpt-app-pcs*:

$pc \notin pcs \ xt \implies \text{xcpt-app } i \ P \ pc \ \text{mxs} \ xt \ \tau \langle \text{proof} \rangle$

lemma *xcpt-eff-pcs*:

$pc \notin pcs \ xt \implies \text{xcpt-eff } i \ P \ pc \ \tau \ xt = [] \langle \text{proof} \rangle$

lemma *pcs-shift*:

$pc < n \implies pc \notin pcs \ (\text{shift } n \ xt) \langle \text{proof} \rangle$

lemma *wt-instr-appRx*:

$[[P, T, m, mpc, xt \vdash is!pc, pc :: \tau s; pc < \text{size } is; \text{size } is < \text{size } \tau s; mpc \leq \text{size } \tau s]$
 $\implies P, T, m, mpc, xt @ \text{shift } (\text{size } is) \ xt' \vdash is!pc, pc :: \tau s \langle \text{proof} \rangle$

lemma *wt-instr-appLx*:

$[[P, T, m, mpc, xt \vdash i, pc :: \tau s; pc \notin pcs \ xt']$
 $\implies P, T, m, mpc, xt' @ xt \vdash i, pc :: \tau s \langle \text{proof} \rangle$

lemma (**in** *TC2*) *wt-instrs-extR*:

$\vdash is, xt [::] \tau s \implies \vdash is, xt [::] \tau s @ \tau s' \langle \text{proof} \rangle$

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 $\vdash is_1 @ is_2, xt_1 @ shift (size is_1) xt_2 [::] \tau_{s_1} @ \tau_{s_2} \langle proof \rangle$

corollary (in *TC2*) *wt-instrs-ext2*:

$\llbracket \vdash is_2, xt_2 [::] \tau_{s_2}; \vdash is_1, xt_1 [::] \tau_{s_1} @ \tau_{s_2}; size \tau_{s_1} = size is_1 \rrbracket$
 $\implies \vdash is_1 @ is_2, xt_1 @ shift (size is_1) xt_2 [::] \tau_{s_1} @ \tau_{s_2} \langle proof \rangle$

corollary (in *TC2*) *wt-instrs-ext-prefix [trans]*:

$\llbracket \vdash is_1, xt_1 [::] \tau_{s_1} @ \tau_{s_2}; \vdash is_2, xt_2 [::] \tau_{s_3};$
 $size \tau_{s_1} = size is_1; prefix \tau_{s_3} \tau_{s_2} \rrbracket$
 $\implies \vdash is_1 @ is_2, xt_1 @ shift (size is_1) xt_2 [::] \tau_{s_1} @ \tau_{s_2} \langle proof \rangle$

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 $\vdash is_1 @ is_2, xt_1 @ shift (size is_1) xt_2 [::] \tau_{s_1} @ \tau \# \tau_{s_2} \langle proof \rangle$

corollary (in *TC2*) *wt-instrs-app-last [trans]*:

assumes $\vdash is_2, xt_2 [::] \tau \# \tau_{s_2} \vdash is_1, xt_1 [::] \tau_{s_1}$
 $last \tau_{s_1} = \tau \quad size \tau_{s_1} = size is_1 + 1$
shows $\vdash is_1 @ is_2, xt_1 @ shift (size is_1) xt_2 [::] \tau_{s_1} @ \tau_{s_2} \langle proof \rangle$

corollary (in *TC2*) *wt-instrs-append-last [trans]*:

assumes $wtis: \vdash is, xt [::] \tau s$ **and** $wti: P, T_r, mxs, mpc, [] \vdash i, pc :: \tau s$
and $pc: pc = size is$ **and** $mpc: mpc = size \tau s$ **and** $is\text{-}\tau s: size is + 1 < size \tau s$
shows $\vdash is @ [i], xt [::] \tau s \langle proof \rangle$

corollary (in *TC2*) *wt-instrs-app2*:

$\llbracket \vdash is_2, xt_2 [::] \tau' \# \tau_{s_2}; \vdash is_1, xt_1 [::] \tau \# \tau_{s_1} @ [\tau'];$
 $xt' = xt_1 @ shift (size is_1) xt_2; size \tau_{s_1} + 1 = size is_1 \rrbracket$
 $\implies \vdash is_1 @ is_2, xt' [::] \tau \# \tau_{s_1} @ \tau' \# \tau_{s_2} \langle proof \rangle$

corollary (in *TC2*) *wt-instrs-app2-simp [trans, simp]*:

$\llbracket \vdash is_2, xt_2 [::] \tau' \# \tau_{s_2}; \vdash is_1, xt_1 [::] \tau \# \tau_{s_1} @ [\tau']; size \tau_{s_1} + 1 = size is_1 \rrbracket$
 $\implies \vdash is_1 @ is_2, xt_1 @ shift (size is_1) xt_2 [::] \tau \# \tau_{s_1} @ \tau' \# \tau_{s_2} \langle proof \rangle$

corollary (in *TC2*) *wt-instrs-Cons [simp]*:

$\llbracket \tau s \neq []; \vdash [i], [] [::] [\tau, \tau']; \vdash is, xt [::] \tau' \# \tau s \rrbracket$
 $\implies \vdash i \# is, shift 1 xt [::] \tau \# \tau' \# \tau s \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle \langle proof \rangle$

theory *Jinja*

imports

J/TypeSafe

J/Annotate

J/execute-Bigstep

J/execute-WellType

JVM/JVMDefensive

JVM/JVMListExample

BV/BVExec

BV/LBVJVM

BV/BVNoTypeError

BV/BVExample

Compiler/TypeComp

begin

end

Bibliography

- [1] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical report, National ICT Australia, Sydney, Mar. 2004.
- [2] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. Prog. Lang. Syst.*, 28(4):619–695, 2006.