

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

Gerwin Klein

Tobias Nipkow

May 14, 2024



# Contents

<b>1</b>	<b>Preface</b>	<b>5</b>
1.1	Theory Dependencies . . . . .	5
<b>2</b>	<b>Jinja Source Language</b>	<b>7</b>
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 . . . . .	19
2.8	Expressions . . . . .	20
2.9	Program State . . . . .	22
2.10	Big Step Semantics . . . . .	22
2.11	Small Step Semantics . . . . .	26
2.12	System Classes . . . . .	31
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 . . . . .	45
2.19	Conformance Relations for Type Soundness Proofs . . . . .	47
2.20	Progress of Small Step Semantics . . . . .	48
2.21	Well-formedness Constraints . . . . .	50
2.22	Type Safety Proof . . . . .	51
2.23	Program annotation . . . . .	53
2.24	Example Expressions . . . . .	54
2.25	Code Generation For BigStep . . . . .	56
2.26	Code Generation For WellType . . . . .	60
<b>3</b>	<b>Jinja Virtual Machine</b>	<b>63</b>
3.1	State of the JVM . . . . .	63
3.2	Instructions of the JVM . . . . .	63
3.3	JVM Instruction Semantics . . . . .	64
3.4	Exception handling in the JVM . . . . .	67
3.5	Program Execution in the JVM . . . . .	68

3.6	A Defensive JVM . . . . .	69
3.7	Example for generating executable code from JVM semantics . . . . .	72
<b>4</b>	<b>Bytecode Verifier</b>	<b>77</b>
4.1	Semilattices . . . . .	77
4.2	The Error Type . . . . .	80
4.3	More about Options . . . . .	82
4.4	Products as Semilattices . . . . .	82
4.5	Fixed Length Lists . . . . .	83
4.6	Typing and Dataflow Analysis Framework . . . . .	84
4.7	More on Semilattices . . . . .	85
4.8	Lifting the Typing Framework to <code>err</code> , <code>app</code> , and <code>eff</code> . . . . .	86
4.9	Kildall's Algorithm . . . . .	88
4.10	Kildall's Algorithm . . . . .	89
4.11	The Lightweight Bytecode Verifier . . . . .	90
4.12	Correctness of the LBV . . . . .	94
4.13	Completeness of the LBV . . . . .	95
4.14	The Jinja Type System as a Semilattice . . . . .	97
4.15	The JVM Type System as Semilattice . . . . .	98
4.16	Effect of Instructions on the State Type . . . . .	100
4.17	Monotonicity of <code>eff</code> and <code>app</code> . . . . .	106
4.18	The Bytecode Verifier . . . . .	106
4.19	The Typing Framework for the JVM . . . . .	108
4.20	Typing and Dataflow Analysis Framework . . . . .	110
4.21	Kildall for the JVM . . . . .	110
4.22	LBV for the JVM . . . . .	112
4.23	BV Type Safety Invariant . . . . .	113
4.24	BV Type Safety Proof . . . . .	115
4.25	Welltyped Programs produce no Type Errors . . . . .	121
4.26	Example Welltypings . . . . .	123
<b>5</b>	<b>Compilation</b>	<b>131</b>
5.1	An Intermediate Language . . . . .	131
5.2	Well-Formedness of Intermediate Language . . . . .	135
5.3	Program Compilation . . . . .	137
5.4	Compilation Stage 1 . . . . .	140
5.5	Correctness of Stage 1 . . . . .	141
5.6	Compilation Stage 2 . . . . .	143
5.7	Correctness of Stage 2 . . . . .	145
5.8	Combining Stages 1 and 2 . . . . .	149
5.9	Preservation of Well-Typedness . . . . .	149

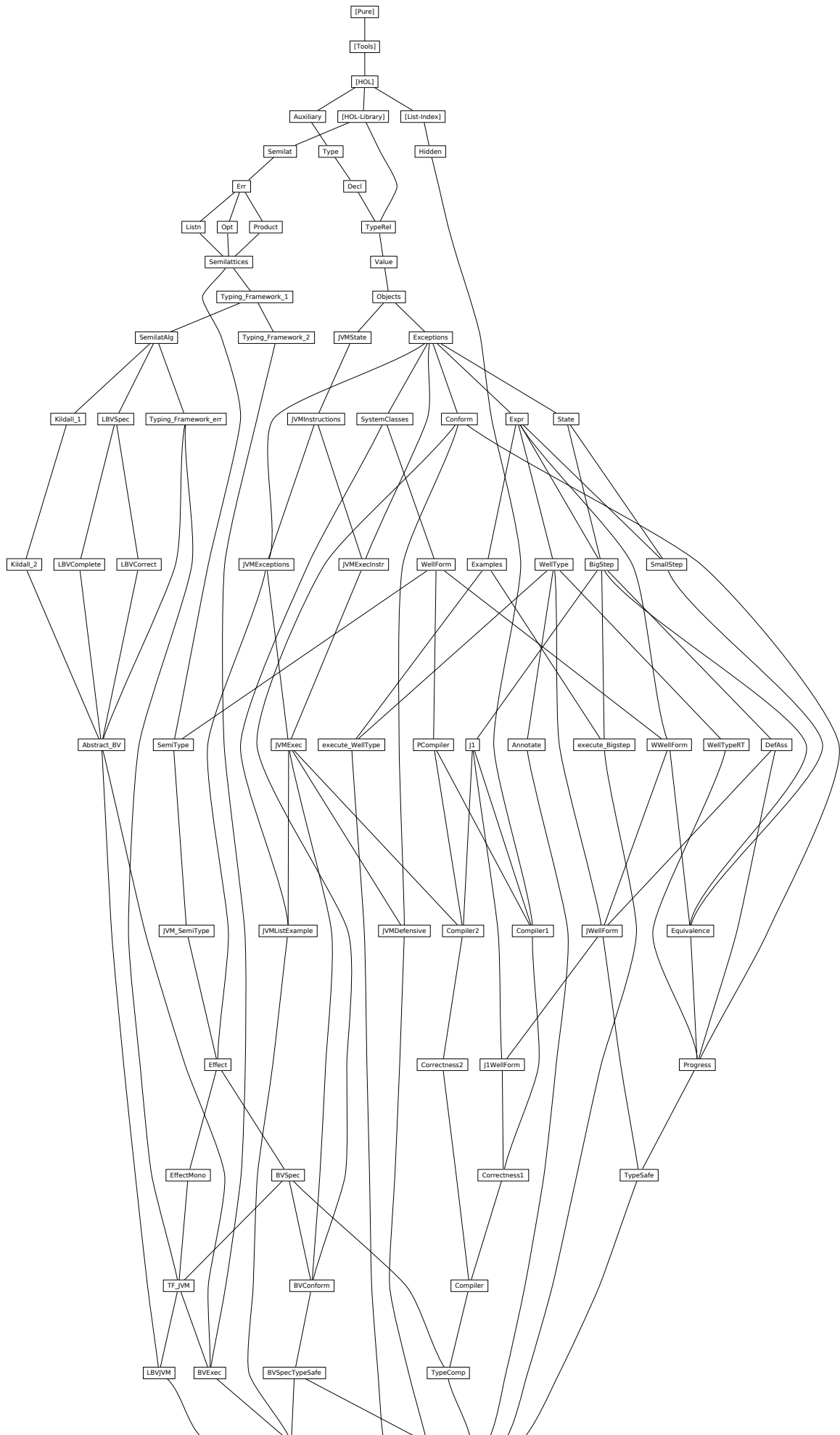
# Chapter 1

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

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

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

#### 2.1.1 *distinct-fst*

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

**where**

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

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

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

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

#### 2.1.2 Using *list-all2* for relations

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

**where**

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

Convenience lemmas

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

**lemma** *rel-list-all2-Cons1*:

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

**lemma** *rel-list-all2-Cons2*:

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

**lemma** *rel-list-all2-refl*:

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

**lemma** *rel-list-all2-antisym*:

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

**lemma** *rel-list-all2-trans*:

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

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

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

**lemma** *rel-list-all2-nthD*:

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

**lemma** *rel-list-all2I*:

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

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**

$$\text{Object} \equiv \text{"Object"}$$

**definition** *this* :: *vname*

**where**

$$\text{this} \equiv \text{"this"}$$

— 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*

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

**lemma** *refTE*:

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

**lemma** *not-refTE*:

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

**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}*

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

**where**

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

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

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

**abbreviation**

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

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

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

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

**lemma** *subcls1-def2*:

*subcls1* *P* =

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

**lemma** *finite-subcls1*: *finite* (*subcls1* *P*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**lemma** *visible-methods-exist*:

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

**lemma** *sees-methods-decl-above*:

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

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

**lemma** *sees-methods-idemp*:

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

**shows**  $\bigwedge m D. Mm M = \text{Some}(m, D) \Longrightarrow$   
 $\exists Mm'. (P \vdash D \text{ sees-methods } Mm') \wedge Mm' M = \text{Some}(m, D)$

**lemma** *sees-methods-decl-mono*:

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

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

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

**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:**

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

**lemma sees-method-decl-above:**

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

**lemma visible-method-exists:**

$P \vdash C \text{ sees } M:Ts \rightarrow T = m \text{ in } D \implies$   
 $\exists D' \ fs \ ms. \text{ class } P \ D = \text{Some}(D', fs, ms) \wedge \text{map-of } ms \ M = \text{Some}(Ts, T, m)$

**lemma sees-method-idemp:**

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

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

**lemma sees-method-is-class:**

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

## 2.4.4 Field lookup

**inductive**

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

**for**  $P :: 'm \ prog$

**where**

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

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

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

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

**lemma has-fields-fun:**

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

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

**lemma all-fields-in-has-fields:**

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

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

**lemma has-fields-decl-above:**

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

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

**lemma** *subcls-notin-has-fields*:

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

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

**lemma** *has-fields-mono-lem*:

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

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

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

**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**  $P \vdash C' \text{ has } F:T \text{ in } D$

**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*:

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

**lemma** *has-visible-field*:

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

**lemma** *sees-field-fun*:

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

**lemma** *sees-field-decl-above*:

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

**lemma** *sees-field-idemp*:

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

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

## 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*]:  $P \vdash C \text{ has-fields } FDTs \implies \text{fields } P \ C = FDTs$

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

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

## 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*

.

**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*

.

**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'*

.

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

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

**by** (*simp add: subcls'-def subcls1-def rtrancl-def*)

**code-pred**

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

*widen*

.

**code-pred**

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

*Fields*

.

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

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

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

**by**(*auto simp add: has-field-def*)

**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*

**by**(*auto simp add: has-field-def*)

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

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

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

**by**(*auto simp add: sees-field-def*)

**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

by(auto simp add: sees-field-def)

**code-pred**

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

.

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

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

by(auto simp add: Method-def)

**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

by(auto simp add: Method-def)

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

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

by(auto intro: Method-i-i-i-o-o-o-oI elim: Method-i-i-i-o-o-o-oE intro!: ext)

**lemma** method-code [code]:

method  $P \ C \ M =$

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

apply (rule sym, rule the-eqI)

apply (simp add: method-def eval-Method-i-i-i-o-o-o-o-conv)

apply (rule arg-cong [where f=The])

apply (auto simp add: Sup-fun-def Sup-bool-def fun-eq-iff)

done

**lemma** eval-Fields-conv:

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

by(auto intro: Fields-i-i-oI elim: Fields-i-i-oE intro!: ext)

**lemma** fields-code [code]:

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

by(simp add: fields-def Predicate.the-def eval-Fields-conv)

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

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

by(auto intro!: ext intro: sees-field-i-i-i-o-oI elim: sees-field-i-i-i-o-oE)

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

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

by(auto intro!: ext intro: sees-field-i-i-i-o-iI elim: sees-field-i-i-i-o-iE)

**lemma** field-code [code]:

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)))
apply (rule sym, rule the-eqI)
apply (simp add: field-def eval-sees-field-i-i-i-o-o-conv)
apply (rule arg-cong [where f=The])
apply (auto simp add: Sup-fun-def Sup-bool-def fun-eq-iff)
done

```

## 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 ⇒ int where
  the-Intg (Intg i) = i
```

```
primrec the-Addr :: val ⇒ addr where
  the-Addr (Addr a) = a
```

```
primrec default-val :: ty ⇒ 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 × cname ⇒ val — field name, defining class, value
```

```
type-synonym
  obj = cname × fields — class instance with class name and fields
```

```
definition obj-ty :: obj ⇒ ty
where
  obj-ty obj ≡ Class (fst obj)
```

```
definition init-fields :: ((vname × cname) × ty) list ⇒ fields
where
```



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

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

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

**where**

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

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

## 2.6.2 Heap

**type-synonym**  $heap = addr \rightarrow obj$

**abbreviation**

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

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

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

**where**

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

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

**where**

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

**definition**  $hext :: heap \Rightarrow heap \Rightarrow bool (- \sqsubseteq - [51, 51] 50)$

**where**

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

**primrec**  $typeof\text{-}h :: heap \Rightarrow val \Rightarrow ty option (typeof\text{-})$

**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**  $new\text{-}Addr\text{-}SomeD:$

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

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

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

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

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

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

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  $\implies$  typeofh v = Some T`

**lemma** `typeof-lit-is-type`:

`typeof v = Some T  $\implies$  is-type P T`

### 2.6.3 Heap extension $\triangleleft$

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

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

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

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

**lemma** `hext-trans`:  $\llbracket h \triangleleft h'; h' \triangleleft h'' \rrbracket \implies h \triangleleft h''$

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

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

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  $\text{Some}(\text{LEAST } a. a \geq n \wedge h a = \text{None})$  else  $\text{None}$`

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

`new-Addr h = gen-new-Addr h 0`

**by**(`simp add: new-Addr-def gen-new-Addr-def split del: if-split cong: if-cong`)

**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))`

**apply**(`simp add: gen-new-Addr-def`)

**apply**(`rule impI`)

**apply**(`rule conjI`)

**apply** `safe[1]`

**apply**(`fastforce intro: Least-equality`)

**apply**(`rule arg-cong[where f=Least]`)

**apply**(`rule ext`)

**apply**(`case-tac n = ac`)

**apply** `simp`

**apply**(`auto`)[1]

**apply** `clarify`

**apply**(`subgoal-tac a = n`)

**apply** `simp`

**apply**(`rule Least-equality`)

**apply** `auto`[2]

**apply**(`rule ccontr`)

**apply**(`erule-tac x=a in allE`)

**apply** `simp`

**done**

**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*  $\in$  *sys-xcpts*  $\wedge$  *OutOfMemory*  $\in$  *sys-xcpts*  $\wedge$  *ClassCast*  $\in$  *sys-xcpts*

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

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

### 2.7.2 preallocated

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

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

**lemma** *preallocatedD*:

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

**lemma** *preallocatedE* [*elim?*]:

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

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

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

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

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

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

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

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

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

**lemma** *preallocated-hext*:

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

**lemma** *preallocated-start*:

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

**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)* (- «-» - [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-mb* = *vname list* × *exp* — Jinja method body: parameter names and expression

**type-synonym**

*J-prog* = *J-mb prog* — Jinja program

The semantics of binary operators:

**fun** *binop* :: *bop* × *val* × *val* ⇒ *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]:

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

## 2.8.1 Syntactic sugar

**abbreviation** (*input*)

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

**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**

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

**abbreviation**

$THROW :: cname \Rightarrow 'a \text{ exp} \text{ where}$   
 $THROW xc == Throw(addr-of-sys-xcpt xc)$

## 2.8.2 Free Variables

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

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

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

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

**end**

## 2.9 Program State

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

**type-synonym**

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

**type-synonym**

$state = heap \times locals$

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

**where**

$hp \equiv fst$

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

**where**

$lcl \equiv snd$

**end**

## 2.10 Big Step Semantics

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

**inductive**

$eval :: J\text{-prog} \Rightarrow expr \Rightarrow state \Rightarrow expr \Rightarrow state \Rightarrow bool$

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

**and**  $evals :: J\text{-prog} \Rightarrow expr\ list \Rightarrow state \Rightarrow expr\ list \Rightarrow state \Rightarrow bool$

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

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

**where**

*New*:

$\llbracket new\text{-Addr } h = Some\ a; P \vdash C\ \text{has-fields}\ FDTs; h' = h(a \mapsto (C, init\text{-fields}\ FDTs)) \rrbracket$

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

| *NewFail*:

$new\text{-Addr } h = None \implies$

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

| *Cast*:

$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle addr\ a, (h,l) \rangle; h\ a = Some(D, fs); P \vdash D \preceq^* C \rrbracket$

$\implies P \vdash \langle Cast\ C\ e, s_0 \rangle \Rightarrow \langle addr\ a, (h,l) \rangle$

| *CastNull*:

$P \vdash \langle e, s_0 \rangle \Rightarrow \langle null, s_1 \rangle \implies$

$P \vdash \langle Cast\ C\ e, s_0 \rangle \Rightarrow \langle null, s_1 \rangle$

| *CastFail*:

$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle addr\ a, (h,l) \rangle; h\ a = Some(D, fs); \neg P \vdash D \preceq^* C \rrbracket$

$\implies P \vdash \langle Cast\ C\ e, s_0 \rangle \Rightarrow \langle THROW\ ClassCast, (h,l) \rangle$

| *CastThrow*:

$P \vdash \langle e, s_0 \rangle \Rightarrow \langle throw\ e', s_1 \rangle \implies$

$P \vdash \langle Cast\ C\ e, s_0 \rangle \Rightarrow \langle throw\ e', s_1 \rangle$

| *Val*:

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

| *BinOp*:

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

| *BinOpThrow1*:

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

| *BinOpThrow2*:

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

| *Var*:

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

| *LAss*:

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

| *LAssThrow*:

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

| *FAcc*:

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

| *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*:

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

| *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*:

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



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

| *WhileBodyThrow*:

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

| *Throw*:

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

| *ThrowNull*:

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

| *ThrowThrow*:

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

| *Try*:

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

| *TryCatch*:

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

| *TryThrow*:

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

| *Nil*:

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

| *Cons*:

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

| *ConsThrow*:

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

### 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**

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

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

**lemma** *[simp]*:  $\text{final}(\text{throw } e) = (\exists a. e = \text{addr } a)$   
**lemma** *finalE*:  $\llbracket \text{final } e; \bigwedge v. e = \text{Val } v \implies R; \bigwedge a. e = \text{Throw } a \implies R \rrbracket \implies R$   
**lemma** *[iff]*:  $\text{finals } []$   
**lemma** *[iff]*:  $\text{finals } (\text{Val } v \# es) = \text{finals } es$   
**lemma** *finals-app-map**[iff]*:  $\text{finals } (\text{map } \text{Val } vs @ es) = \text{finals } es$   
**lemma** *[iff]*:  $\text{finals } (\text{map } \text{Val } vs)$   
**lemma** *[iff]*:  $\text{finals } (\text{throw } e \# es) = (\exists a. e = \text{addr } a)$   
**lemma** *not-final-ConsI*:  $\neg \text{final } e \implies \neg \text{finals}(e \# es)$

**lemma** *eval-final*:  $P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle \implies \text{final } e'$   
**and** *evals-final*:  $P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \implies \text{finals } es'$

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

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

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

**lemma** *eval-finalId*:  
**assumes** *finals*:  $\text{finals } es$  **shows**  $P \vdash \langle es, s \rangle [\Rightarrow] \langle es, s \rangle$

**theorem** *eval-heat*:  $P \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle \implies h \trianglelefteq h'$   
**and** *evals-heat*:  $P \vdash \langle es, (h, l) \rangle [\Rightarrow] \langle es', (h', l') \rangle \implies h \trianglelefteq h'$

end

## 2.11 Small Step Semantics

**theory** *SmallStep*  
**imports** *Expr State*  
**begin**

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

**lemmas** *blocks-induct* =  $\text{blocks.induct}[\text{split-format } (\text{complete})]$

**lemma** *[simp]*:  
 $\llbracket \text{size } vs = \text{size } Vs; \text{size } Ts = \text{size } Vs \rrbracket \implies \text{fv}(\text{blocks}(Vs, Ts, vs, e)) = \text{fv } e - \text{set } Vs$

**definition** *assigned* ::  $vname \Rightarrow expr \Rightarrow bool$

**where**  
 $\text{assigned } V e \equiv \exists v e'. e = (V := \text{Val } v;; e')$

**inductive-set**

$\text{red} :: J\text{-prog} \Rightarrow ((expr \times state) \times (expr \times state)) \text{ set}$   
**and**  $\text{reds} :: J\text{-prog} \Rightarrow ((expr \text{ list} \times state) \times (expr \text{ list} \times state)) \text{ set}$   
**and**  $\text{red}' :: J\text{-prog} \Rightarrow expr \Rightarrow state \Rightarrow expr \Rightarrow state \Rightarrow bool$   
 $(- \vdash ((1 \langle -, - \rangle) \rightarrow / (1 \langle -, - \rangle))) [51, 0, 0, 0, 0] 81$   
**and**  $\text{reds}' :: J\text{-prog} \Rightarrow expr \text{ list} \Rightarrow state \Rightarrow expr \text{ list} \Rightarrow state \Rightarrow bool$   
 $(- \vdash ((1 \langle -, - \rangle) [\rightarrow] / (1 \langle -, - \rangle))) [51, 0, 0, 0, 0] 81$   
**for**  $P :: J\text{-prog}$

where

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \equiv ((e, s), e', s') \in \text{red } P$$

$$| P \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle \equiv ((es, s), es', s') \in \text{reds } P$$

$$| \text{RedNew:}$$

$$\llbracket \text{new-Addr } h = \text{Some } a; P \vdash C \text{ has-fields FDTs; } h' = h(a \mapsto (C, \text{init-fields FDTs})) \rrbracket$$

$$\implies P \vdash \langle \text{new } C, (h, l) \rangle \rightarrow \langle \text{addr } a, (h', l) \rangle$$

$$| \text{RedNewFail:}$$

$$\text{new-Addr } h = \text{None} \implies$$

$$P \vdash \langle \text{new } C, (h, l) \rangle \rightarrow \langle \text{THROW OutOfMemory}, (h, l) \rangle$$

$$| \text{CastRed:}$$

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$$

$$P \vdash \langle \text{Cast } C \ e, s \rangle \rightarrow \langle \text{Cast } C \ e', s' \rangle$$

$$| \text{RedCastNull:}$$

$$P \vdash \langle \text{Cast } C \ \text{null}, s \rangle \rightarrow \langle \text{null}, s \rangle$$

$$| \text{RedCast:}$$

$$\llbracket \text{hp } s \ a = \text{Some}(D, fs); P \vdash D \preceq^* C \rrbracket$$

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

$$| \text{RedCastFail:}$$

$$\llbracket \text{hp } s \ a = \text{Some}(D, fs); \neg P \vdash D \preceq^* C \rrbracket$$

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

$$| \text{BinOpRed1:}$$

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$$

$$P \vdash \langle e \ \llbracket \text{bop} \rrbracket \ e_2, s \rangle \rightarrow \langle e' \ \llbracket \text{bop} \rrbracket \ e_2, s' \rangle$$

$$| \text{BinOpRed2:}$$

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$$

$$P \vdash \langle (Val \ v_1) \ \llbracket \text{bop} \rrbracket \ e, s \rangle \rightarrow \langle (Val \ v_1) \ \llbracket \text{bop} \rrbracket \ e', s' \rangle$$

$$| \text{RedBinOp:}$$

$$\text{binop}(\text{bop}, v_1, v_2) = \text{Some } v \implies$$

$$P \vdash \langle (Val \ v_1) \ \llbracket \text{bop} \rrbracket \ (Val \ v_2), s \rangle \rightarrow \langle Val \ v, s \rangle$$

$$| \text{RedVar:}$$

$$\text{lcl } s \ V = \text{Some } v \implies$$

$$P \vdash \langle Var \ V, s \rangle \rightarrow \langle Val \ v, s \rangle$$

$$| \text{LAssRed:}$$

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$$

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

$$| \text{RedLAss:}$$

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

$$| \text{FAccRed:}$$

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$$

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

- | *RedFAcc*:  

$$\llbracket hp \ s \ a = \text{Some}(C,fs); fs(F,D) = \text{Some } v \rrbracket$$

$$\implies P \vdash \langle (\text{addr } a) \cdot F\{D\}, s \rangle \rightarrow \langle \text{Val } v, s \rangle$$
- | *RedFAccNull*:  

$$P \vdash \langle \text{null} \cdot F\{D\}, s \rangle \rightarrow \langle \text{THROW NullPointer}, s \rangle$$
- | *FAssRed1*:  

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$$

$$P \vdash \langle e \cdot F\{D\} := e_2, s \rangle \rightarrow \langle e' \cdot F\{D\} := e_2, s' \rangle$$
- | *FAssRed2*:  

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$$

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

$$h \ a = \text{Some}(C,fs) \implies$$

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

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

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$$

$$P \vdash \langle e \cdot M(es), s \rangle \rightarrow \langle e' \cdot M(es), s' \rangle$$
- | *CallParams*:  

$$P \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle \implies$$

$$P \vdash \langle (\text{Val } v) \cdot M(es), s \rangle \rightarrow \langle (\text{Val } v) \cdot M(es'), s' \rangle$$
- | *RedCall*:  

$$\llbracket hp \ s \ a = \text{Some}(C,fs); P \vdash C \text{ sees } M:Ts \rightarrow T = (pns, \text{body}) \text{ in } D; \text{ size } vs = \text{size } pns; \text{ size } Ts = \text{size } pns \rrbracket$$

$$\implies P \vdash \langle (\text{addr } a) \cdot M(\text{map Val } vs), s \rangle \rightarrow \langle \text{blocks}(\text{this}\#pns, \text{Class } D\#Ts, \text{Addr } a\#vs, \text{body}), s \rangle$$
- | *RedCallNull*:  

$$P \vdash \langle \text{null} \cdot M(\text{map Val } vs), s \rangle \rightarrow \langle \text{THROW NullPointer}, s \rangle$$
- | *BlockRedNone*:  

$$\llbracket P \vdash \langle e, (h, l(V := \text{None})) \rangle \rightarrow \langle e', (h', l') \rangle; l' \ V = \text{None}; \neg \text{assigned } V \ e \rrbracket$$

$$\implies P \vdash \langle \{V:T; e\}, (h,l) \rangle \rightarrow \langle \{V:T; e'\}, (h', l'(V := l \ V)) \rangle$$
- | *BlockRedSome*:  

$$\llbracket P \vdash \langle e, (h, l(V := \text{None})) \rangle \rightarrow \langle e', (h', l') \rangle; l' \ V = \text{Some } v; \neg \text{assigned } V \ e \rrbracket$$

$$\implies P \vdash \langle \{V:T; e\}, (h,l) \rangle \rightarrow \langle \{V:T := \text{Val } v; e'\}, (h', l'(V := l \ V)) \rangle$$
- | *InitBlockRed*:  

$$\llbracket P \vdash \langle e, (h, l(V \mapsto v)) \rangle \rightarrow \langle e', (h', l') \rangle; l' \ V = \text{Some } v' \rrbracket$$

$$\implies P \vdash \langle \{V:T := \text{Val } v; e\}, (h,l) \rangle \rightarrow \langle \{V:T := \text{Val } v'; e'\}, (h', l'(V := l \ V)) \rangle$$
- | *RedBlock*:  

$$P \vdash \langle \{V:T; \text{Val } u\}, s \rangle \rightarrow \langle \text{Val } u, s \rangle$$

| *RedInitBlock*:

$$P \vdash \langle \{V:T := Val v; Val u\}, s \rangle \rightarrow \langle Val u, s \rangle$$

| *SeqRed*:

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$$

$$P \vdash \langle e;;e_2, s \rangle \rightarrow \langle e';;e_2, s' \rangle$$

| *RedSeq*:

$$P \vdash \langle (Val v);;e_2, s \rangle \rightarrow \langle e_2, s \rangle$$

| *CondRed*:

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$$

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

| *RedCondT*:

$$P \vdash \langle \text{if } (\text{true}) \ e_1 \ \text{else } e_2, s \rangle \rightarrow \langle e_1, s \rangle$$

| *RedCondF*:

$$P \vdash \langle \text{if } (\text{false}) \ e_1 \ \text{else } e_2, s \rangle \rightarrow \langle e_2, s \rangle$$

| *RedWhile*:

$$P \vdash \langle \text{while}(b) \ c, s \rangle \rightarrow \langle \text{if}(b) \ (c;;\text{while}(b) \ c) \ \text{else } \text{unit}, s \rangle$$

| *ThrowRed*:

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies$$

$$P \vdash \langle \text{throw } e, s \rangle \rightarrow \langle \text{throw } e', s' \rangle$$

| *RedThrowNull*:

$$P \vdash \langle \text{throw } \text{null}, s \rangle \rightarrow \langle \text{THROW } \text{NullPointer}, s \rangle$$

| *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*:

$$P \vdash \langle \text{try } (Val v) \ \text{catch}(C \ V) \ e_2, s \rangle \rightarrow \langle Val v, s \rangle$$

| *RedTryCatch*:

$$\llbracket hp \ s \ a = \text{Some}(D, fs); P \vdash D \preceq^* C \rrbracket$$

$$\implies P \vdash \langle \text{try } (\text{Throw } a) \ \text{catch}(C \ V) \ e_2, s \rangle \rightarrow \langle \{V:\text{Class } C := \text{addr } a; e_2\}, s \rangle$$

| *RedTryFail*:

$$\llbracket hp \ s \ a = \text{Some}(D, fs); \neg P \vdash D \preceq^* C \rrbracket$$

$$\implies P \vdash \langle \text{try } (\text{Throw } a) \ \text{catch}(C \ V) \ e_2, s \rangle \rightarrow \langle \text{Throw } a, s \rangle$$

| *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*:

$$P \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle \implies$$

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

— Exception propagation

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

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

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

**shows**  $R \ e \ h \ l \ e' \ h' \ l'$

### 2.11.2 Some easy lemmas

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

**lemma** [*iff*]:  $\neg P \vdash \langle \text{Val } v, s \rangle \rightarrow \langle e', s' \rangle$

**lemma** [*iff*]:  $\neg P \vdash \langle \text{Throw } a, s \rangle \rightarrow \langle e', s' \rangle$

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

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

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

**and**  $P \vdash \langle es, (h_0, l_0) \rangle [\rightarrow] \langle es', (h_1, l_1) \rangle \implies \text{dom } l_0 \subseteq \text{dom } l_1$

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

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

**lemma** *Red-lcl-add*:

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

**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 ≡ (Object, (undefined,[],[]))
```

```
definition NullPointerC :: 'm cdecl
```

```
where
```

```
NullPointerC ≡ (NullPointer, (Object,[],[]))
```

```
definition ClassCastC :: 'm cdecl
```

```
where
```

```
ClassCastC ≡ (ClassCast, (Object,[],[]))
```

```
definition OutOfMemoryC :: 'm cdecl
```

```
where
```

```
OutOfMemoryC ≡ (OutOfMemory, (Object,[],[]))
```

```
definition SystemClasses :: 'm cdecl list
```

```
where
```

```
SystemClasses ≡ [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 wf-mdecl-test = 'm prog ⇒ cname ⇒ 'm mdecl ⇒ bool
```

```
definition wf-fdecl :: 'm prog ⇒ fdecl ⇒ bool
```

```
where
```

```
wf-fdecl P ≡ λ(F,T). is-type P T
```

```
definition wf-mdecl :: 'm wf-mdecl-test ⇒ 'm wf-mdecl-test
```

```
where
```

```
wf-mdecl wf-md P C ≡ λ(M,Ts,T,mb).
```

```
(∀ T ∈ set Ts. is-type P T) ∧ is-type P T ∧ wf-md P C (M,Ts,T,mb)
```

```
definition wf-cdecl :: 'm wf-mdecl-test ⇒ 'm prog ⇒ 'm cdecl ⇒ bool
```

```
where
```

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

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

**where**

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

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

**where**

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

### 2.13.1 Well-formedness lemmas

**lemma**  $class\text{-}wf$ :

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

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

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

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

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

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

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

**lemma**  $subcls1\text{-}wfD$ :

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

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

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

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

**lemma**  $subcls\text{-}asym$ :

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

**lemma**  $subcls\text{-}irrefl$ :

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

**lemma**  $acyclic\text{-}subcls1$ :

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

**lemma**  $wf\text{-}subcls1$ :

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

**lemma**  $single\text{-}valued\text{-}subcls1$ :

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

**lemma**  $subcls\text{-}induct$ :



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

**lemma** *subcls1-induct-aux*:

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

**shows**

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

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

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

**lemma** *subcls-C-Object*:

**assumes** *class*: *is-class*  $P C$  **and** *wf*: *wf-prog wf-md*  $P$

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

**lemma** *is-type-pTs*:

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

**shows**  $\text{set } Ts \subseteq \text{types } P$

## 2.13.2 Well-formedness and method lookup

**lemma** *sees-wf-mdecl*:

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

**shows** *wf-mdecl* *wf-md*  $P D (M, Ts, T, m)$

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

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

**shows**  $\forall D Ts T m. P \vdash C \text{ sees } M:Ts \rightarrow T = m \text{ in } D \longrightarrow$

$$(\exists D' Ts' T' m'. P \vdash C' \text{ sees } M:Ts' \rightarrow T' = m' \text{ in } D' \wedge P \vdash Ts \leq Ts' \wedge P \vdash T' \leq T)$$

**lemma** *sees-method-mono2*:

$$\begin{aligned} & \llbracket P \vdash C' \preceq^* C; \text{wf-prog wf-md } P; \\ & \quad P \vdash C \text{ sees } M:Ts \rightarrow T = m \text{ in } D; P \vdash C' \text{ sees } M:Ts' \rightarrow T' = m' \text{ in } D' \rrbracket \\ & \implies P \vdash Ts \leq Ts' \wedge P \vdash T' \leq T \end{aligned}$$

**lemma** *mdecls-visible*:

**assumes** *wf*: *wf-prog wf-md*  $P$  **and** *class*: *is-class*  $P C$

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

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

**lemma** *mdecl-visible*:

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

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

**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-prog wf-md*  $P$

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

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

**lemma** *wf-prog-lift*:  
**assumes** *wf*: *wf-prog* ( $\lambda P C$  *bd*. *A P C bd*) *P*  
**and rule**:  
 $\bigwedge$  *wf-md C M Ts C T m bd*.  
*wf-prog wf-md P*  $\implies$   
*P*  $\vdash$  *C sees M:Ts $\rightarrow$ T = m in C*  $\implies$   
*set Ts*  $\subseteq$  *types P*  $\implies$   
*bd = (M, Ts, T, m)*  $\implies$   
*A P C bd*  $\implies$   
*B P C bd*  
**shows** *wf-prog* ( $\lambda P C$  *bd*. *B P C bd*) *P*

### 2.13.3 Well-formedness and field lookup

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

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

**lemma** *sees-field-is-type*:  
 $\llbracket P \vdash C \text{ sees } F:T \text{ in } D; \text{wf-prog wf-md } P \rrbracket \implies \text{is-type } P T$

**lemma** *wf-syscls*:  
*set SystemClasses*  $\subseteq$  *set P*  $\implies$  *wf-syscls P*  
**end**

## 2.14 Weak well-formedness of Jinja programs

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

**definition** *wf-J-mdecl*  $:: J\text{-prog} \Rightarrow \text{cname} \Rightarrow J\text{-mb mdecl} \Rightarrow \text{bool}$   
**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*  $\wedge$  *fv body*  $\subseteq$   $\{this\} \cup \text{set } pns$

**lemma** *wf-J-mdecl[simp]*:  
*wf-J-mdecl P C (M, Ts, T, pns, body) =*  
*(length Ts = length pns*  $\wedge$  *distinct pns*  $\wedge$  *this*  $\notin$  *set pns*  $\wedge$  *fv body*  $\subseteq$   $\{this\} \cup \text{set } pns$ )

**abbreviation**

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

**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*:

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

**lemma** *CastRedsNull*:

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

**lemma** *CastRedsAddr*:

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

**lemma** *CastRedsFail*:

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

**lemma** *CastRedsThrow*:

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

## LAss

**lemma** *LAssReds*:

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

**lemma** *LAssRedsVal*:

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

**lemma** *LAssRedsThrow*:

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

## BinOp

**lemma** *BinOp1Reds*:

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

**lemma** *BinOp2Reds*:

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

**lemma** *BinOpRedsVal*:

**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:  $P \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle \text{Val } v_2, s_2 \rangle$   
**and**  $op$ :  $\text{binop}(\text{bop}, v_1, v_2) = \text{Some } v$

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

**lemma** *BinOpRedsThrow1*:

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

**lemma** *BinOpRedsThrow2*:

**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:  $P \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle \text{throw } e, s_2 \rangle$   
**shows**  $P \vdash \langle e_1 \ \langle \text{bop} \rangle \ e_2, s_0 \rangle \rightarrow^* \langle \text{throw } e, s_2 \rangle$

## FAcc

**lemma** *FAccReds*:

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

**lemma** *FAccRedsVal*:

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

**lemma** *FAccRedsNull*:

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

**lemma** *FAccRedsThrow*:

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

## FAss

**lemma** *FAssReds1*:

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

**lemma** *FAssReds2*:

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

**lemma** *FAssRedsVal*:

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

**lemma** *FAssRedsNull*:

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

**lemma** *FAssRedsThrow1*:

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

**lemma** *FAssRedsThrow2*:

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

;;

**lemma** *SeqReds*:

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

**lemma** *SeqRedsThrow*:

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

**lemma** *SeqReds2*:

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

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

**shows**  $P \vdash \langle e_1;; e_2, s_0 \rangle \rightarrow^* \langle e_2', s_2 \rangle$

**If**

**lemma** *CondReds*:

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

**lemma** *CondRedsThrow*:

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

**lemma** *CondReds2T*:

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

**and**  $e_1\text{-steps}$ :  $P \vdash \langle e_1, s_1 \rangle \rightarrow^* \langle e', s_2 \rangle$

**shows**  $P \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s_0 \rangle \rightarrow^* \langle e', s_2 \rangle$

**lemma** *CondReds2F*:

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

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

**shows**  $P \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s_0 \rangle \rightarrow^* \langle e', s_2 \rangle$

**While**

**lemma** *WhileFReds*:

**assumes**  $b\text{-steps}$ :  $P \vdash \langle b, s \rangle \rightarrow^* \langle \text{false}, s' \rangle$

**shows**  $P \vdash \langle \text{while } (b) c, s \rangle \rightarrow^* \langle \text{unit}, s' \rangle$

**lemma** *WhileRedsThrow*:

**assumes**  $b\text{-steps}$ :  $P \vdash \langle b, s \rangle \rightarrow^* \langle \text{throw } e, s' \rangle$

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

**lemma** *WhileTReds*:

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

**and**  $c\text{-steps}$ :  $P \vdash \langle c, s_1 \rangle \rightarrow^* \langle \text{Val } v_1, s_2 \rangle$

**and while-steps:**  $P \vdash \langle \text{while } (b) \ c, s_2 \rangle \rightarrow^* \langle e, s_3 \rangle$   
**shows**  $P \vdash \langle \text{while } (b) \ c, s_0 \rangle \rightarrow^* \langle e, s_3 \rangle$   
**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**  $P \vdash \langle \text{while } (b) \ c, s_0 \rangle \rightarrow^* \langle \text{throw } e, s_2 \rangle$

## Throw

**lemma ThrowReds:**  
 $P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P \vdash \langle \text{throw } e, s \rangle \rightarrow^* \langle \text{throw } e', s' \rangle$   
**lemma ThrowRedsNull:**  
 $P \vdash \langle e, s \rangle \rightarrow^* \langle \text{null}, s' \rangle \implies P \vdash \langle \text{throw } e, s \rangle \rightarrow^* \langle \text{THROW NullPointer}, s' \rangle$   
**lemma ThrowRedsThrow:**  
 $P \vdash \langle e, s \rangle \rightarrow^* \langle \text{throw } a, s' \rangle \implies P \vdash \langle \text{throw } e, s \rangle \rightarrow^* \langle \text{throw } a, s' \rangle$

## InitBlock

**lemma InitBlockReds-aux:**  
 $P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies$   
 $\forall h \ l \ h' \ l' \ v. \ s = (h, l(V \mapsto v)) \longrightarrow s' = (h', l') \longrightarrow$   
 $P \vdash \langle \{V:T := \text{Val } v; e\}, (h, l) \rangle \rightarrow^* \langle \{V:T := \text{Val}(\text{the}(l' \ V)); e'\}, (h', l'(V := (l \ V))) \rangle$   
**lemma InitBlockReds:**  
 $P \vdash \langle e, (h, l(V \mapsto v)) \rangle \rightarrow^* \langle e', (h', l') \rangle \implies$   
 $P \vdash \langle \{V:T := \text{Val } v; e\}, (h, l) \rangle \rightarrow^* \langle \{V:T := \text{Val}(\text{the}(l' \ V)); e'\}, (h', l'(V := (l \ V))) \rangle$   
**lemma InitBlockRedsFinal:**  
 $\llbracket P \vdash \langle e, (h, l(V \mapsto v)) \rangle \rightarrow^* \langle e', (h', l') \rangle; \text{final } e' \rrbracket \implies$   
 $P \vdash \langle \{V:T := \text{Val } v; e\}, (h, l) \rangle \rightarrow^* \langle e', (h', l'(V := l \ V)) \rangle$

## Block

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

## try-catch

**lemma TryReds:**  
 $P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P \vdash \langle \text{try } e \text{ catch}(C \ V) \ e_2, s \rangle \rightarrow^* \langle \text{try } e' \text{ catch}(C \ V) \ e_2, s' \rangle$   
**lemma TryRedsVal:**  
 $P \vdash \langle e, s \rangle \rightarrow^* \langle \text{Val } v, s' \rangle \implies P \vdash \langle \text{try } e \text{ catch}(C \ V) \ e_2, s \rangle \rightarrow^* \langle \text{Val } v, s' \rangle$   
**lemma TryCatchRedsFinal:**  
**assumes**  $e_1$ -steps:  $P \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle \text{Throw } a, (h_1, l_1) \rangle$   
**and**  $h_1 a$ :  $h_1 \ a = \text{Some}(D, fs)$  **and**  $\text{sub}$ :  $P \vdash D \preceq^* C$   
**and**  $e_2$ -steps:  $P \vdash \langle e_2, (h_1, l_1(V \mapsto \text{Addr } a)) \rangle \rightarrow^* \langle e_2', (h_2, l_2) \rangle$   
**and**  $\text{final}$ :  $\text{final } e_2'$   
**shows**  $P \vdash \langle \text{try } e_1 \text{ catch}(C \ V) \ e_2, s_0 \rangle \rightarrow^* \langle e_2', (h_2, (l_2::\text{locals})(V := l_1 \ V)) \rangle$   
**lemma TryRedsFail:**  
 $\llbracket P \vdash \langle e_1, s \rangle \rightarrow^* \langle \text{Throw } a, (h, l) \rangle; h \ a = \text{Some}(D, fs); \neg P \vdash D \preceq^* C \rrbracket$   
 $\implies P \vdash \langle \text{try } e_1 \text{ catch}(C \ V) \ e_2, s \rangle \rightarrow^* \langle \text{Throw } a, (h, l) \rangle$

## List

**lemma ListReds1:**  
 $P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P \vdash \langle e \# es, s \rangle [\rightarrow]^* \langle e' \# es, s' \rangle$

**lemma** *ListReds2*:

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

**lemma** *ListRedsVal*:

$$\begin{aligned} & \llbracket P \vdash \langle e, s_0 \rangle \rightarrow^* \langle Val v, s_1 \rangle; P \vdash \langle es, s_1 \rangle [\rightarrow]^* \langle es', s_2 \rangle \rrbracket \\ & \implies P \vdash \langle e \# es, s_0 \rangle [\rightarrow]^* \langle Val v \# es', s_2 \rangle \end{aligned}$$

## Call

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

**lemma** *assumes wf: wwf-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**  $P \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \implies fvs es' \subseteq fvs es$

**lemma** *Red-dom-lcl*:

$$P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies dom l' \subseteq dom l \cup fv e \text{ and}$$

$$P \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \implies dom l' \subseteq dom l \cup fvs es$$

**lemma** *Reds-dom-lcl*:

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

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

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

**lemma** *override-on-upd-lemma*:

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

**lemma** *blocksReds*:

$$\bigwedge l. \llbracket length Vs = length Ts; length vs = length Ts; distinct Vs;$$

$$P \vdash \langle e, (h, l(Vs [\mapsto] vs)) \rangle \rightarrow^* \langle e', (h', l') \rangle \rrbracket$$

$$\implies P \vdash \langle blocks(Vs, Ts, vs, e), (h, l) \rangle \rightarrow^* \langle blocks(Vs, Ts, map (the \circ l') Vs, e'), (h', override-on l' l (set Vs)) \rangle$$

**lemma** *blocksFinal*:

$$\bigwedge l. \llbracket length Vs = length Ts; length vs = length Ts; final e \rrbracket \implies$$

$$P \vdash \langle blocks(Vs, Ts, vs, e), (h, l) \rangle \rightarrow^* \langle e, (h, l) \rangle$$

**lemma** *blocksRedsFinal*:

**assumes** *wf: length Vs = length Ts length vs = 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''*:  $l'' = override-on l' l (set Vs)$

**shows**  $P \vdash \langle blocks(Vs, Ts, vs, e), (h, l) \rangle \rightarrow^* \langle e', (h', l'') \rangle$

An now the actual method call reduction lemmas.

**lemma** *CallRedsObj*:

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

**lemma** *CallRedsParams*:

$$P \vdash \langle es, s \rangle [\rightarrow]^* \langle es', s' \rangle \implies P \vdash \langle (Val v) \cdot M(es), s \rangle \rightarrow^* \langle (Val v) \cdot M(es'), s' \rangle$$

**lemma** *CallRedsFinal*:

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

**and**  $P \vdash \langle e, s_0 \rangle \rightarrow^* \langle addr a, s_1 \rangle$

$$P \vdash \langle es, s_1 \rangle [\rightarrow]^* \langle map Val vs, (h_2, l_2) \rangle$$

$$h_2 a = Some(C.fs) P \vdash C \text{ sees } M: Ts \rightarrow T = (pns, body) \text{ in } D$$

$$size vs = size pns$$

**and**  $l_2': l_2' = [this \mapsto Addr a, pns[\mapsto]vs]$

**and body:**  $P \vdash \langle \text{body}, (h_2, l_2^\wedge) \rangle \rightarrow^* \langle \text{ef}, (h_3, l_3) \rangle$   
**and final ef**  
**shows**  $P \vdash \langle e.M(\text{es}), s_0 \rangle \rightarrow^* \langle \text{ef}, (h_3, l_2) \rangle$

**lemma CallRedsThrowParams:**

**assumes**  $e\text{-steps}$ :  $P \vdash \langle e, s_0 \rangle \rightarrow^* \langle \text{Val } v, s_1 \rangle$   
**and**  $es\text{-steps}$ :  $P \vdash \langle \text{es}, s_1 \rangle [\rightarrow]^* \langle \text{map Val } vs_1 \ @ \ \text{throw } a \ # \ \text{es}_2, s_2 \rangle$   
**shows**  $P \vdash \langle e.M(\text{es}), s_0 \rangle \rightarrow^* \langle \text{throw } a, s_2 \rangle$

**lemma CallRedsThrowObj:**

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

**lemma CallRedsNull:**

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

## The main Theorem

**lemma assumes**  $wuf$ :  $wuf\text{-}J\text{-prog } P$

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

### 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:**

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

**lemma blocksEval:**

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

**lemma**

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

**shows**  $eval\text{-}restrict\text{-}lcl$ :

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

**lemma eval-notfree-unchanged:**

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

**lemma eval-closed-lcl-unchanged:**

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

**lemma list-eval-Throw:**

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

The key lemma:

**lemma**

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

**shows** *extend-1-eval*:

$$P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies (\bigwedge s' e'. P \vdash \langle e'', s'' \rangle \Rightarrow \langle e', s' \rangle \implies P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle)$$

**and** *extend-1-evals*:

$$P \vdash \langle es, t \rangle [\rightarrow] \langle es'', t'' \rangle \implies (\bigwedge t' es'. P \vdash \langle es'', t'' \rangle [\Rightarrow] \langle es', t' \rangle \implies P \vdash \langle es, t \rangle [\Rightarrow] \langle es', t' \rangle)$$

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**  $P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$

**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**  $P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle$

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

### 2.15.3 Equivalence

And now, the crowning achievement:

**corollary** *big-iff-small*:

$$\text{wwf-J-prog } P \implies$$

$$P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle = (P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \wedge \text{final } e')$$

**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**

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

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

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

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

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

**where**

*WTNew*:

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

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



| *WTCast*:

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

| *WTVal*:

$$\text{typeof } v = \text{Some } T \implies \\ P, E \vdash \text{Val } v :: T$$

| *WTVar*:

$$E \ V = \text{Some } T \implies \\ P, E \vdash \text{Var } V :: T$$

| *WTBinOpEq*:

$$\llbracket P, E \vdash e_1 :: T_1; P, E \vdash e_2 :: T_2; P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1 \rrbracket \\ \implies P, E \vdash e_1 \langle \text{Eq} \rangle e_2 :: \text{Boolean}$$

| *WTBinOpAdd*:

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

| *WTLAss*:

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

| *WTFAcc*:

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

| *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*:

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

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

| *WTCons*:  
 $\llbracket P, E \vdash e :: T; P, E \vdash es [::] Ts \rrbracket$   
 $\implies P, E \vdash e \# es [::] T \# Ts$

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

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

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

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

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

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

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

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

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

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

**lemma** *wt-env-mono*:

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

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

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

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

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

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

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

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

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

**where**

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

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

- | *WTrtNew*:  
 $is\text{-}class\ P\ C \implies$   
 $P, E, h \vdash new\ C : Class\ C$
- | *WTrtCast*:  
 $\llbracket P, E, h \vdash e : T; is\text{-}refT\ T; is\text{-}class\ P\ C \rrbracket$   
 $\implies P, E, h \vdash Cast\ C\ e : Class\ C$
- | *WTrtVal*:  
 $typeof_h\ v = Some\ T \implies$   
 $P, E, h \vdash Val\ v : T$
- | *WTrtVar*:  
 $E\ V = Some\ T \implies$   
 $P, E, h \vdash Var\ V : T$
- | *WTrtBinOpEq*:  
 $\llbracket P, E, h \vdash e_1 : T_1; P, E, h \vdash e_2 : T_2 \rrbracket$   
 $\implies P, E, h \vdash e_1 \llbracket Eq \rrbracket e_2 : Boolean$
- | *WTrtBinOpAdd*:  
 $\llbracket P, E, h \vdash e_1 : Integer; P, E, h \vdash e_2 : Integer \rrbracket$   
 $\implies P, E, h \vdash e_1 \llbracket Add \rrbracket e_2 : Integer$
- | *WTrtLAss*:  
 $\llbracket E\ V = Some\ T; P, E, h \vdash e : T'; P \vdash T' \leq T \rrbracket$   
 $\implies P, E, h \vdash V := e : Void$
- | *WTrtFAcc*:  
 $\llbracket P, E, h \vdash e : Class\ C; P \vdash C\ has\ F:T\ in\ D \rrbracket \implies$   
 $P, E, h \vdash e \cdot F\{D\} : T$
- | *WTrtFAccNT*:  
 $P, E, h \vdash e : NT \implies$   
 $P, E, h \vdash e \cdot F\{D\} : T$
- | *WTrtFAss*:  
 $\llbracket P, E, h \vdash e_1 : Class\ C; P \vdash C\ has\ F:T\ in\ D; P, E, h \vdash e_2 : T_2; P \vdash T_2 \leq T \rrbracket$   
 $\implies P, E, h \vdash e_1 \cdot F\{D\} := e_2 : Void$
- | *WTrtFAssNT*:  
 $\llbracket P, E, h \vdash e_1 : NT; P, E, h \vdash e_2 : T_2 \rrbracket$   
 $\implies P, E, h \vdash e_1 \cdot F\{D\} := e_2 : Void$
- | *WTrtCall*:  
 $\llbracket P, E, h \vdash e : Class\ C; P \vdash C\ sees\ M:Ts \rightarrow T = (pns, body)\ in\ D;$   
 $P, E, h \vdash es\ [:]\ Ts'; P \vdash Ts' \llbracket \leq \rrbracket Ts \rrbracket$   
 $\implies P, E, h \vdash e \cdot M(es) : T$
- | *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*:

$$\begin{array}{l} P, E(V \mapsto T), h \vdash e : T' \implies \\ P, E, h \vdash \{V:T; e\} : T' \end{array}$$

$$\begin{array}{l} | \text{WTrtSeq:} \\ \llbracket P, E, h \vdash e_1:T_1; P, E, h \vdash e_2:T_2 \rrbracket \\ \implies P, E, h \vdash e_1;;e_2 : T_2 \end{array}$$

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

$$\begin{array}{l} | \text{WTrtWhile:} \\ \llbracket P, E, h \vdash e : \text{Boolean}; P, E, h \vdash c:T \rrbracket \\ \implies P, E, h \vdash \text{while}(e) \ c : \text{Void} \end{array}$$

$$\begin{array}{l} | \text{WTrtThrow:} \\ \llbracket P, E, h \vdash e : T_r; \text{is-refT } T_r \rrbracket \implies \\ P, E, h \vdash \text{throw } e : T \end{array}$$

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

— well-typed expression lists

$$\begin{array}{l} | \text{WTrtNil:} \\ P, E, h \vdash [] [:] [] \end{array}$$

$$\begin{array}{l} | \text{WTrtCons:} \\ \llbracket P, E, h \vdash e : T; P, E, h \vdash es [:] Ts \rrbracket \\ \implies P, E, h \vdash e\#es [:] T\#Ts \end{array}$$

### 2.17.1 Easy consequences

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

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

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

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

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

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

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

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

### 2.17.2 Some interesting lemmas

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

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

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

**lemma** *WTrt-hext-mono*:  $P, E, h \vdash e : T \implies h \sqsubseteq h' \implies P, E, h' \vdash e : T$   
**and** *WTrts-hext-mono*:  $P, E, h \vdash es \ [:] \ Ts \implies h \sqsubseteq h' \implies P, E, h' \vdash es \ [:] \ Ts$

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

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**

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

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

$A \sqcap B \equiv \text{case } A \text{ of } \text{None} \Rightarrow B$   
 $\quad \mid \ [A] \Rightarrow (\text{case } B \text{ of } \text{None} \Rightarrow [A] \mid [B] \Rightarrow [A \cap B])$

**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 (infix  $\in\in$  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 (infix  $\sqsubseteq$  50)  
**where**

$A \sqsubseteq B \equiv \text{case } B \text{ of } \text{None} \Rightarrow \text{True}$   
 $\quad \mid \ [B] \Rightarrow (\text{case } A \text{ of } \text{None} \Rightarrow \text{False} \mid [A] \Rightarrow A \subseteq B)$

**lemmas** *hyperset-defs* =

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

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

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

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

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

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

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

## 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

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

$\mathcal{A}s ([\ ]) = [\{\}]$

$\mathcal{A}s (e\#es) = \mathcal{A} \ e \sqcup \mathcal{A}s \ es$

### 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**  $\text{As-map-Val[simp]}$ :  $\mathcal{A}s (\text{map Val } vs) = [\{\}]$

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

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

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

**lemma** *sqUn-lem*:  $A \sqsubseteq A' \implies A \sqcup B \sqsubseteq A' \sqcup B$

**lemma** *diff-lem*:  $A \sqsubseteq A' \implies A \ominus b \sqsubseteq A' \ominus b$

**lemma** *D-mono*:  $\bigwedge A A'. A \sqsubseteq A' \implies \mathcal{D} e A \implies \mathcal{D} (e::'a \text{ exp}) A'$

**and** *Ds-mono*:  $\bigwedge A A'. A \sqsubseteq A' \implies \mathcal{D}s \text{ es } A \implies \mathcal{D}s (es::'a \text{ exp list}) A'$

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

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

**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* ::  $'m \text{ prog} \Rightarrow \text{heap} \Rightarrow \text{obj} \Rightarrow \text{bool}$  ( $-, - \vdash - \surd$  [51,51,51] 50)

**where**

$P, h \vdash \text{obj } \surd \equiv$

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

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

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

**where**

$P \vdash h \surd \equiv$

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

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

**where**

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

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

**abbreviation**

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

( $-, - \vdash - [ : \leq ] -$  [51,51,51,51] 50) **where**

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

### 2.19.1 Value conformance $: \leq$

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

**lemma** *typeof-conf* [*simp*]:  $\text{typeof}_h v = \text{Some } T \implies P, h \vdash v : \leq T$

**lemma** *typeof-lit-conf* [*simp*]:  $\text{typeof } v = \text{Some } T \implies P, h \vdash v : \leq T$

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

**lemma** *conf-upd-obj*:  $h a = \text{Some}(C, fs) \implies (P, h(a \mapsto (C, fs')) \vdash x : \leq T) = (P, h \vdash x : \leq T)$

**lemma** *conf-widen*:  $P, h \vdash v : \leq T \implies P \vdash T \leq T' \implies P, h \vdash v : \leq T'$

**lemma** *conf-hext*:  $h \sqsubseteq h' \implies P, h \vdash v : \leq T \implies P, h' \vdash v : \leq T$

**lemma** *conf-ClassD*:  $P, h \vdash v : \leq \text{Class } C \implies$   
 $v = \text{Null} \vee (\exists a \text{ obj } T. v = \text{Addr } a \wedge h a = \text{Some obj} \wedge \text{obj-ty obj} = T \wedge P \vdash T \leq \text{Class } C)$   
**lemma** *conf-NT* [iff]:  $P, h \vdash v : \leq \text{NT} = (v = \text{Null})$   
**lemma** *non-npD*:  $\llbracket v \neq \text{Null}; P, h \vdash v : \leq \text{Class } C \rrbracket$   
 $\implies \exists a C' \text{ fs}. v = \text{Addr } a \wedge h a = \text{Some}(C', \text{fs}) \wedge P \vdash C' \preceq^* C$

### 2.19.2 Value list conformance $[:\leq]$

**lemma** *confs-widens* [trans]:  $\llbracket P, h \vdash vs [:\leq] Ts; P \vdash Ts [:\leq] Ts' \rrbracket \implies P, h \vdash vs [:\leq] Ts'$   
**lemma** *confs-rev*:  $P, h \vdash \text{rev } s [:\leq] t = (P, h \vdash s [:\leq] \text{rev } t)$   
**lemma** *confs-conv-map*:  
 $\bigwedge Ts'. P, h \vdash vs [:\leq] Ts' = (\exists Ts. \text{map typeof}_h vs = \text{map Some } Ts \wedge P \vdash Ts [:\leq] Ts')$   
**lemma** *confs-heat*:  $P, h \vdash vs [:\leq] Ts \implies h \trianglelefteq h' \implies P, h' \vdash vs [:\leq] Ts$   
**lemma** *confs-Cons2*:  $P, h \vdash xs [:\leq] y \# ys = (\exists z zs. xs = z \# zs \wedge P, h \vdash z : \leq y \wedge P, h \vdash zs [:\leq] ys)$

### 2.19.3 Object conformance

**lemma** *oconf-heat*:  $P, h \vdash \text{obj } \checkmark \implies h \trianglelefteq h' \implies P, h' \vdash \text{obj } \checkmark$   
**lemma** *oconf-init-fields*:  
 $P \vdash C \text{ has-fields FDTs} \implies P, h \vdash (C, \text{init-fields FDTs}) \checkmark$   
**by**(*fastforce simp add: has-field-def oconf-def init-fields-def map-of-map*  
*dest: has-fields-fun*)

**lemma** *oconf-fupd* [intro?]:  
 $\llbracket P \vdash C \text{ has } F:T \text{ in } D; P, h \vdash v : \leq T; P, h \vdash (C, \text{fs}) \checkmark \rrbracket$   
 $\implies P, h \vdash (C, \text{fs}((F, D) \mapsto v)) \checkmark$

### 2.19.4 Heap conformance

**lemma** *hconfD*:  $\llbracket P \vdash h \checkmark; h a = \text{Some obj} \rrbracket \implies P, h \vdash \text{obj } \checkmark$   
**lemma** *hconf-new*:  $\llbracket P \vdash h \checkmark; h a = \text{None}; P, h \vdash \text{obj } \checkmark \rrbracket \implies P \vdash h(a \mapsto \text{obj}) \checkmark$   
**lemma** *hconf-upd-obj*:  $\llbracket P \vdash h \checkmark; h a = \text{Some}(C, \text{fs}); P, h \vdash (C, \text{fs}') \checkmark \rrbracket \implies P \vdash h(a \mapsto (C, \text{fs}')) \checkmark$

### 2.19.5 Local variable conformance

**lemma** *lconf-heat*:  $\llbracket P, h \vdash l (:\leq) E; h \trianglelefteq h' \rrbracket \implies P, h' \vdash l (:\leq) E$   
**lemma** *lconf-upd*:  
 $\llbracket P, h \vdash l (:\leq) E; P, h \vdash v : \leq T; E V = \text{Some } T \rrbracket \implies P, h \vdash l(V \mapsto v) (:\leq) E$   
**lemma** *lconf-empty*[iff]:  $P, h \vdash \text{Map.empty} (:\leq) E$   
**lemma** *lconf-upd2*:  $\llbracket P, h \vdash l (:\leq) E; P, h \vdash v : \leq T \rrbracket \implies P, h \vdash l(V \mapsto v) (:\leq) E(V \mapsto T)$

end

## 2.20 Progress of Small Step Semantics

**theory** *Progress*  
**imports** *Equivalence WellTypeRT DefAss ../Common/Conform*  
**begin**

**lemma** *final-addrE*:  
 $\llbracket P, E, h \vdash e : \text{Class } C; \text{final } e;$   
 $\bigwedge a. e = \text{addr } a \implies R;$   
 $\bigwedge a. e = \text{Throw } a \implies R \rrbracket \implies R$



**lemma** *finalRefE*:

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

Derivation of new induction scheme for well typing:

**inductive**

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

**where**

$$\begin{aligned} & P, E, h \vdash e : ' T \equiv WTrt' P h E e T \\ & | P, E, h \vdash es [:\ ] Ts \equiv 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 \\ & \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 \\ & | \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 \text{Eq} \rrbracket e_2 : ' \text{Boolean} \\ & | \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 \llbracket \text{Add} \rrbracket e_2 : ' \text{Integer} \\ & | \llbracket P, E, h \vdash \text{Var } V : ' T; P, E, h \vdash e : ' T'; P \vdash T' \leq T \text{ ~~W/##/TT/TS~~ \rrbracket \\ & \implies P, E, h \vdash V := e : ' \text{Void} \\ & | \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.F\{D\} : ' T \\ & | P, E, h \vdash e : ' NT \implies P, E, h \vdash e.F\{D\} : ' T \\ & | \llbracket P, E, h \vdash e_1 : ' \text{Class } C; P \vdash C \text{ has } F:T \text{ in } D; \\ & \quad P, E, h \vdash e_2 : ' T_2; P \vdash T_2 \leq T \rrbracket \\ & \implies P, E, h \vdash e_1.F\{D\} := e_2 : ' \text{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.F\{D\} := e_2 : ' \text{Void} \\ & | \llbracket P, E, h \vdash e : ' \text{Class } C; P \vdash C \text{ sees } M:Ts \rightarrow T = (\text{pns}, \text{body}) \text{ in } D; \\ & \quad P, E, h \vdash es [:\ ] Ts'; P \vdash Ts' [\leq] Ts \rrbracket \\ & \implies P, E, h \vdash e.M(es) : ' T \\ & | \llbracket P, E, h \vdash e : ' NT; P, E, h \vdash es [:\ ] Ts \rrbracket \implies P, E, h \vdash e.M(es) : ' T \\ & | P, E, h \vdash [] [:\ ] \\ & | \llbracket P, E, h \vdash e : ' T; P, E, h \vdash es [:\ ] Ts \rrbracket \implies P, E, h \vdash e \# es [:\ ] T \# Ts \\ & | \llbracket \text{typeof}_h v = \text{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 := \text{Val } v; e_2\} : ' T_2 \\ & | \llbracket P, E(V \mapsto T), h \vdash e : ' T'; \neg \text{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 : ' \text{Boolean}; P, E, h \vdash e_1 : ' T_1; P, E, h \vdash e_2 : ' T_2; \\ & \quad P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1; \\ & \quad 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 \\ \\ & | \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} \\ & | \llbracket P, E, h \vdash e : ' T_r; \text{is-refT } T_r \rrbracket \implies P, E, h \vdash \text{throw } e : ' T \end{aligned}$$

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

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

**lemma** [iff]:  $P, E, h \vdash \text{Val } v : ' T = (\text{typeof}_h v = \text{Some } T)$

**lemma** [iff]:  $P, E, h \vdash \text{Var } v : ' T = (E v = \text{Some } T)$

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

**and** *wts-wts'*:  $P, E, h \vdash es [:] Ts \implies P, E, h \vdash es [:'] Ts$

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

**and** *wts'-wts*:  $P, E, h \vdash es [:'] Ts \implies P, E, h \vdash es [:] Ts$

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

**corollary** *wts'-iff-wts*:  $(P, E, h \vdash es [:'] Ts) = (P, E, h \vdash es [:] Ts)$

**theorem assumes** *wf*: *wwf-J-prog* *P* **and** *hconf*:  $P \vdash h \checkmark$

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

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

**and**  $P, E, h \vdash es [:] Ts \implies$

$(\bigwedge l. \llbracket \mathcal{D} s \text{ es [dom } l]; \neg \text{finals } es \rrbracket \implies \exists es' s'. P \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', s' \rangle)$

**end**

## 2.21 Well-formedness Constraints

**theory** *JWellForm*

**imports** *../Common/WellForm WWellForm WellType DefAss*

**begin**

**definition** *wf-J-mdecl* ::  $J\text{-prog} \Rightarrow \text{cname} \Rightarrow J\text{-mb mdecl} \Rightarrow \text{bool}$

**where**

$wf\text{-J-mdecl } P C \equiv \lambda(M, Ts, T, (pns, body)).$

$\text{length } Ts = \text{length } pns \wedge$

$\text{distinct } pns \wedge$

$\text{this} \notin \text{set } pns \wedge$

$(\exists T'. P, [\text{this} \mapsto \text{Class } C, pns \mapsto Ts] \vdash \text{body} :: T' \wedge P \vdash T' \leq T) \wedge$

$\mathcal{D} \text{ body } [\{\text{this}\} \cup \text{set } pns]$

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

$wf\text{-J-mdecl } P C (M, Ts, T, pns, body) \equiv$

$(\text{length } Ts = \text{length } pns \wedge$

$\text{distinct } pns \wedge$

$\text{this} \notin \text{set } pns \wedge$

$(\exists T'. P, [\text{this} \mapsto \text{Class } C, pns \mapsto Ts] \vdash \text{body} :: T' \wedge P \vdash T' \leq T) \wedge$

$\mathcal{D} \text{ body } [\{\text{this}\} \cup \text{set } pns])$

**abbreviation**

$wf\text{-J-prog} :: J\text{-prog} \Rightarrow \text{bool}$  **where**

$wf\text{-J-prog} == wf\text{-prog } wf\text{-J-mdecl}$

**lemma** *wf-J-prog-wf-J-mdecl*:

$\llbracket wf\text{-J-prog } P; (C, D, fds, mths) \in \text{set } P; jmdcl \in \text{set } mths \rrbracket$

$\implies wf\text{-}J\text{-}mdecl\ P\ C\ jmdcl$

**lemma**  $wf\text{-}mdecl\text{-}wvf\text{-}mdecl: wf\text{-}J\text{-}mdecl\ P\ C\ Md \implies wvf\text{-}J\text{-}mdecl\ P\ C\ Md$

**lemma**  $wf\text{-}prog\text{-}wvf\text{-}prog: wf\text{-}J\text{-}prog\ P \implies wvf\text{-}J\text{-}prog\ P$

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*:

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

**and** *reds-preserves-hconf*:

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

**theorem** *red-preserves-lconf*:

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

**and** *reds-preserves-lconf*:

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

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

**lemma** [*iff*]:  $\bigwedge A. \llbracket \text{length } Vs = \text{length } Ts; \text{length } vs = \text{length } Ts \rrbracket \implies$   
 $\mathcal{D} (\text{blocks } (Vs, Ts, vs, e))\ A = \mathcal{D}\ e\ (A \sqcup [\text{set } Vs])$

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

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

Now preservation of definite assignment.

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

**shows** *red-preserves-defass*:

$P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies \mathcal{D}\ e\ \lfloor \text{dom } l \rfloor \implies \mathcal{D}\ e'\ \lfloor \text{dom } l' \rfloor$

**and**  $P \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \implies \mathcal{D}s\ es\ \lfloor \text{dom } l \rfloor \implies \mathcal{D}s\ es'\ \lfloor \text{dom } l' \rfloor$

Combining conformance of heap and local variables:

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

$\llbracket P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle; P, E, hp\ s \vdash e : T; P, E \vdash s \checkmark \rrbracket \implies P, E \vdash s' \checkmark$

**lemma** *reds-preserves-sconf*:

$\llbracket P \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle; P, E, hp\ s \vdash es [:] Ts; P, E \vdash s \checkmark \rrbracket \implies P, E \vdash s' \checkmark$

### 2.22.2 Subject reduction

**lemma** *wf-blocks*:

$$\begin{aligned} \wedge E. \llbracket \text{length } Vs = \text{length } Ts; \text{length } vs = \text{length } Ts \rrbracket &\Longrightarrow \\ (P, E, h \vdash \text{blocks}(Vs, Ts, vs, e) : T) = & \\ (P, E(Vs[\mapsto]Ts), h \vdash e : T \wedge (\exists Ts'. \text{map } (\text{typeof}_h) \text{ vs} = \text{map } \text{Some } Ts' \wedge P \vdash Ts' [\leq] Ts)) & \end{aligned}$$

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

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

$$\begin{aligned} (\wedge E \ T. \llbracket P, E \vdash (h, l) \checkmark; P, E, h \vdash e : T \rrbracket & \\ \Longrightarrow \exists T'. P, E, h' \vdash e' : T' \wedge P \vdash T' \leq T & \end{aligned}$$

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

$$\begin{aligned} (\wedge E \ Ts. \llbracket P, E \vdash (h, l) \checkmark; P, E, h \vdash es [:] Ts \rrbracket & \\ \Longrightarrow \exists Ts'. P, E, h' \vdash es' [:] Ts' \wedge P \vdash Ts' [\leq] Ts & \end{aligned}$$

**corollary** *subject-reduction*:

$$\begin{aligned} \llbracket \text{wf-J-prog } P; P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle; P, E \vdash s \checkmark; P, E, hp \ s \vdash e : T \rrbracket & \\ \Longrightarrow \exists T'. P, E, hp \ s' \vdash e' : T' \wedge P \vdash T' \leq T & \end{aligned}$$

**corollary** *subjects-reduction*:

$$\begin{aligned} \llbracket \text{wf-J-prog } P; P \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle; P, E \vdash s \checkmark; P, E, hp \ s \vdash es [:] Ts \rrbracket & \\ \Longrightarrow \exists Ts'. P, E, hp \ s' \vdash es' [:] Ts' \wedge P \vdash Ts' [\leq] Ts & \end{aligned}$$

### 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-J-prog* *P* **and** *Red*:  $P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle$

**shows**  $\wedge T. \llbracket P, E, hp \ s \vdash e : T; P, E \vdash s \checkmark \rrbracket \Longrightarrow P, E \vdash s' \checkmark$

**lemma** *Red-preserves-defass*:

**assumes** *wf*: *wf-J-prog* *P* **and** *reds*:  $P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle$

**shows**  $\mathcal{D} \ e \ [ \text{dom}(\text{lcl } s) ] \Longrightarrow \mathcal{D} \ e' \ [ \text{dom}(\text{lcl } s') ]$

**using** *reds*

**proof** (*induct rule:converse-rtrancl-induct2*)

**case** *refl* **thus** ?*case* .

**next**

**case** (*step* *e s e' s'*) **thus** ?*case*

**by**(*cases s, cases s'*)(*auto dest:red-preserves-defass[OF wf]*)

**qed**

**lemma** *Red-preserves-type*:

**assumes** *wf*: *wf-J-prog* *P* **and** *Red*:  $P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle$

**shows**  $\llbracket P, E \vdash s \checkmark; P, E, hp \ s \vdash e : T \rrbracket$

$$\Longrightarrow \exists T'. P \vdash T' \leq T \wedge P, E, hp \ s' \vdash e' : T'$$

### 2.22.4 Lifting to $\Rightarrow$

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

**lemma** *eval-preserves-sconf*:

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

**lemma** *eval-preserves-type*: **assumes** *wf*: *wf-J-prog* *P*

**shows**  $\llbracket P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle; P, E \vdash s \checkmark; P, E \vdash e :: T \rrbracket$

$$\implies \exists T'. P \vdash T' \leq T \wedge P, E, hp \ s' \vdash e' : T'$$

### 2.22.5 The final polish

The above preservation lemmas are now combined and packed nicely.

**definition** *wf-config* :: *J-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-J-prog* *P*

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

**theorem** *Subject-reductions*:

**assumes** *wf*: *wf-J-prog* *P* **and** *reds*:  $P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle$

**shows**  $\bigwedge T. P, E, s \vdash e : T \checkmark \implies \exists T'. P, E, s' \vdash e' : T' \checkmark \wedge P \vdash T' \leq T$

**corollary** *Progress*: **assumes** *wf*: *wf-J-prog* *P*

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

**corollary** *TypeSafety*:

**fixes** *s*::*state*

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

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

**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**  $(\exists v. e' = Val \ v \wedge P, hp \ s' \vdash v : \leq T) \vee$   
 $(\exists a. e' = Throw \ a \wedge a \in dom(hp \ s'))$

**end**

## 2.23 Program annotation

**theory** *Annotate* **imports** *WellType* **begin**

**inductive**

*Anno* ::  $[J\text{-prog}, env, expr \quad , expr] \Rightarrow bool$

$(-, \vdash - \rightsquigarrow -$  [51,0,0,51]50)

**and** *Annos* ::  $[J\text{-prog}, env, expr \ list, expr \ list] \Rightarrow bool$

$(-, \vdash - [\rightsquigarrow] -$  [51,0,0,51]50)

**for** *P* :: *J-prog*

**where**

*AnnoNew*:  $P, E \vdash new \ C \rightsquigarrow new \ C$

| *AnnoCast*:  $P, E \vdash e \rightsquigarrow e' \implies P, E \vdash Cast \ C \ e \rightsquigarrow Cast \ C \ e'$

| *AnnoVal*:  $P, E \vdash Val \ v \rightsquigarrow Val \ v$

| *AnnoVarVar*:  $E \ V = [T] \implies P, E \vdash Var \ V \rightsquigarrow Var \ V$

| *AnnoVarField*:  $\llbracket E \ V = None; E \ this = [Class \ C]; P \vdash C \ sees \ V : T \ in \ D \rrbracket$   
 $\implies P, E \vdash Var \ V \rightsquigarrow Var \ this \cdot V \{D\}$

| *AnnoBinOp*:

$\llbracket P, E \vdash e1 \rightsquigarrow e1'; P, E \vdash e2 \rightsquigarrow e2' \rrbracket$

$\implies P, E \vdash e1 \ll\text{bop}\gg e2 \rightsquigarrow e1' \ll\text{bop}\gg e2'$

| *AnnoLAssVar*:

$\llbracket E \ V = \lfloor T \rfloor; P, E \vdash e \rightsquigarrow e' \rrbracket \implies P, E \vdash V := e \rightsquigarrow V := e'$   
| *AnnoLAssField*:  
 $\llbracket E \ V = \text{None}; E \ \text{this} = \lfloor \text{Class } C \rfloor; P \vdash C \ \text{sees } V:T \ \text{in } D; P, E \vdash e \rightsquigarrow e' \rrbracket$   
 $\implies P, E \vdash V := e \rightsquigarrow \text{Var } \text{this} \cdot V \{D\} := e'$   
| *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*:  $P, E \vdash \lfloor \rightsquigarrow \rfloor$   
| *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* ⇒ *val list* ⇒ *bool*

**where**

*Nil*: *map-val* [] []

| *Cons*: *map-val xs ys* ⇒ *map-val (Val y # xs) (y # ys)*

**inductive** *map-val2* :: *expr list* ⇒ *val list* ⇒ *expr list* ⇒ *bool*

**where**

*Nil*: *map-val2* [] [] []



| *Cons*:  $\text{map-val2 } xs \ ys \ zs \implies \text{map-val2 } (\text{Val } y \ \# \ xs) \ (y \ \# \ ys) \ zs$   
 | *Throw*:  $\text{map-val2 } (\text{throw } e \ \# \ xs) \ [] \ (\text{throw } e \ \# \ xs)$

**theorem** *map-val-conv*:  $(xs = \text{map Val } ys) = \text{map-val } xs \ ys$

**theorem** *map-val2-conv*:

$(xs = \text{map Val } ys \ @ \ \text{throw } e \ \# \ zs) = \text{map-val2 } xs \ ys \ (\text{throw } e \ \# \ zs)$

**lemma** *CallNull2*:

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

**apply**(*rule CallNull, assumption+*)

**apply**(*simp add: map-val-conv[symmetric]*)

**done**

**lemma** *CallParamsThrow2*:

$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash \langle es, s_1 \rangle [\Rightarrow] \langle evs, s_2 \rangle;$   
 $\text{map-val2 } evs \ vs \ (\text{throw } ex \ \# \ es') \rrbracket$   
 $\implies P \vdash \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle \text{throw } ex, s_2 \rangle$

**apply**(*rule eval-vals.CallParamsThrow, assumption+*)

**apply**(*simp add: map-val2-conv[symmetric]*)

**done**

**lemma** *Call2*:

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

**apply**(*rule Call, assumption+*)

**apply**(*simp add: map-val-conv[symmetric]*)

**apply** *assumption+*

**done**

**code-pred**

(*modes*:  $i \Rightarrow o \Rightarrow \text{bool}$ )  
*map-val*

.

**code-pred**

(*modes*:  $i \Rightarrow o \Rightarrow o \Rightarrow \text{bool}$ )  
*map-val2*

.

**lemmas** [*code-pred-intro*] =

*eval-vals.New 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 ⇒ i ⇒ i ⇒ o ⇒ o ⇒ bool as execute)
  eval
proof –
  case eval
  from eval.premis show thesis
  proof(cases (no-simp))
    case CallNull thus ?thesis
      by(rule eval.CallNull2[OF refl])(simp add: map-val-conv[symmetric])
    next
      case CallParamsThrow thus ?thesis
        by(rule eval.CallParamsThrow2[OF refl])(simp add: map-val2-conv[symmetric])
      next
        case Call thus ?thesis
          by –(rule eval.Call2[OF refl], simp-all add: map-val-conv[symmetric])
        next
          case WhileBodyThrow thus ?thesis by(rule eval.WhileBodyThrow2[OF refl])
          qed(assumption|erule (4) eval.that[OF refl]|erule (3) eval.that[OF refl])+
    next
      case evals
      from evals.premis show thesis
      by(cases (no-simp))(assumption|erule (3) evals.that[OF refl])+
  qed

```

```

notation execute (- ⊢ ((1⟨-,/-⟩) ⇒ / ⟨'-, '⟩) [51,0,0] 81)

```

```

definition test1 = [] ⊢ ⟨testExpr1,(Map.empty,Map.empty)⟩ ⇒ ⟨-, -⟩

```

```

definition test2 = [] ⊢ ⟨testExpr2,(Map.empty,Map.empty)⟩ ⇒ ⟨-, -⟩

```

```

definition test3 = [] ⊢ ⟨testExpr3,(Map.empty,Map.empty("V"↦Intg 77))⟩ ⇒ ⟨-, -⟩

```

```

definition test4 = [] ⊢ ⟨testExpr4,(Map.empty,Map.empty)⟩ ⇒ ⟨-, -⟩

```

```

definition test5 = [(("Object",('',[],[])),("C",("Object",[("F",Integer)],[]))] ⊢ ⟨testExpr5,(Map.empty,Map.empty)⟩ ⇒ ⟨-, -⟩

```

```

definition test6 = [(("Object",('',[],[])), classI)] ⊢ ⟨testExpr6,(Map.empty,Map.empty)⟩ ⇒ ⟨-, -⟩

```

**definition**  $V = "V"$

**definition**  $C = "C"$

**definition**  $F = "F"$

**ML-val**  $\langle$

$val SOME ((@{code Val} (@{code Intg} (@{code int-of-integer} 5)), -, -) = Predicate.yield @\{code test1\};$

$val SOME ((@{code Val} (@{code Intg} (@{code int-of-integer} 11)), -, -) = Predicate.yield @\{code test2\};$

$val SOME ((@{code Val} (@{code Intg} (@{code int-of-integer} 83)), -, -) = Predicate.yield @\{code test3\};$

$val SOME ((-, (-, l)), -) = Predicate.yield @\{code test4\};$

$val SOME (@\{code Intg} (@{code int-of-integer} 6)) = l @\{code V\};$

$val SOME ((-, (h, -)), -) = Predicate.yield @\{code test5\};$

$val SOME (c, fs) = h (@\{code nat-of-integer} 1);$

$val SOME (obj, -) = h (@\{code nat-of-integer} 0);$

$val SOME (@\{code Intg} i) = fs (@\{code F}, @\{code C});$

$@\{assert\} (c = @\{code C\} \text{ andalso } obj = @\{code Object\} \text{ andalso } i = @\{code int-of-integer} 42);$

$val SOME ((@{code Val} (@{code Intg} (@{code int-of-integer} 160)), -, -) = Predicate.yield @\{code test6\};$

$\rangle$

**definition**  $test7 = [classObject, classL] \vdash \langle testExpr-BuildList, (Map.empty, Map.empty) \rangle \Rightarrow \langle -, - \rangle$

**definition**  $L = "L"$

**definition**  $N = "N"$

**ML-val**  $\langle$

$val SOME ((-, (h, -)), -) = Predicate.yield @\{code test7\};$

$val SOME (-, fs1) = h (@\{code nat-of-integer} 0);$

$val SOME (-, fs2) = h (@\{code nat-of-integer} 1);$

$val SOME (-, fs3) = h (@\{code nat-of-integer} 2);$

$val SOME (-, fs4) = h (@\{code nat-of-integer} 3);$

$val F = @\{code F\};$

$val L = @\{code L\};$

$val N = @\{code N\};$

$@\{assert\} (fs1 (F, L) = SOME (@\{code Intg} (@{code int-of-integer} 1)) \text{ andalso}$

$fs1 (N, L) = SOME (@\{code Addr} (@{code nat-of-integer} 1)) \text{ andalso}$

$fs2 (F, L) = SOME (@\{code Intg} (@{code int-of-integer} 2)) \text{ andalso}$

$fs2 (N, L) = SOME (@\{code Addr} (@{code nat-of-integer} 2)) \text{ andalso}$

$fs3 (F, L) = SOME (@\{code Intg} (@{code int-of-integer} 3)) \text{ andalso}$

$fs3 (N, L) = SOME (@\{code Addr} (@{code nat-of-integer} 3)) \text{ andalso}$

$fs4 (F, L) = SOME (@\{code Intg} (@{code int-of-integer} 4)) \text{ andalso}$

$fs4 (N, L) = SOME @\{code Null\});$

$\rangle$

**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-val**  $\langle$

$val\ SOME\ ((-, (h, l)), -) = Predicate.yield\ @\{code\ test8\};$

$val\ SOME\ (-, fs1) = h\ (@\{code\ nat-of-integer\}\ 0);$

$val\ SOME\ (-, fs2) = h\ (@\{code\ nat-of-integer\}\ 1);$

$val\ i = @\{code\ i\};$

$val\ t = @\{code\ t\};$

$val\ A = @\{code\ A\};$

$@\{assert\}\ (fs1\ (i, A) = SOME\ (@\{code\ Intg\}\ (@\{code\ int-of-integer\}\ 10))\ andalso$

$fs1\ (t, A) = SOME\ @\{code\ Null\}\ andalso$

$fs2\ (i, A) = SOME\ (@\{code\ Intg\}\ (@\{code\ int-of-integer\}\ 50))\ andalso$

$fs2\ (t, A) = SOME\ @\{code\ Null\});$

$\rangle$

**end**

## 2.26 Code Generation For WellType

**theory** *execute-WellType*

**imports**

*WellType Examples*

**begin**

**lemma** *WTCond1*:

$\llbracket P, E \vdash e :: Boolean; P, E \vdash e_1 :: T_1; P, E \vdash e_2 :: T_2; P \vdash T_1 \leq T_2;$

$P \vdash T_2 \leq T_1 \longrightarrow T_2 = T_1 \rrbracket \Longrightarrow P, E \vdash if\ (e)\ e_1\ else\ e_2 :: T_2$

**by** (*fastforce*)

**lemma** *WTCond2*:

$\llbracket P, E \vdash e :: Boolean; P, E \vdash e_1 :: T_1; P, E \vdash e_2 :: T_2; P \vdash T_2 \leq T_1;$

$P \vdash T_1 \leq T_2 \longrightarrow T_1 = T_2 \rrbracket \Longrightarrow P, E \vdash if\ (e)\ e_1\ else\ e_2 :: T_1$

**by** (*fastforce*)

**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:  $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow \text{bool}$  as type-check,  $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$  as infer-type)  
 WT

proof -

case WT

from WT.premis show thesis

proof(cases (no-simp))

case (WTCond E e e1 T1 e2 T2 T)

from  $\langle x \vdash T1 \leq T2 \vee x \vdash T2 \leq T1 \rangle$  show thesis

proof

assume  $x \vdash T1 \leq T2$

with  $\langle x \vdash T1 \leq T2 \longrightarrow T = T2 \rangle$  have  $T = T2$  ..

from  $\langle xa = E \rangle \langle xb = \text{if } (e) \text{ e1 else e2} \rangle \langle xc = T \rangle \langle x, E \vdash e :: \text{Boolean} \rangle$

$\langle x, E \vdash e1 :: T1 \rangle \langle x, E \vdash e2 :: T2 \rangle \langle x \vdash T1 \leq T2 \rangle \langle x \vdash T2 \leq T1 \longrightarrow T = T1 \rangle$

show ?thesis unfolding  $\langle T = T2 \rangle$  by(rule WT.WTCond1[OF refl])

next

assume  $x \vdash T2 \leq T1$

with  $\langle x \vdash T2 \leq T1 \longrightarrow T = T1 \rangle$  have  $T = T1$  ..

from  $\langle xa = E \rangle \langle xb = \text{if } (e) \text{ e1 else e2} \rangle \langle xc = T \rangle \langle x, E \vdash e :: \text{Boolean} \rangle$

$\langle x, E \vdash e1 :: T1 \rangle \langle x, E \vdash e2 :: T2 \rangle \langle x \vdash T2 \leq T1 \rangle \langle x \vdash T1 \leq T2 \longrightarrow T = T2 \rangle$

show ?thesis unfolding  $\langle T = T1 \rangle$  by(rule WT.WTCond2[OF refl])

qed

qed(assumption|erule (2) WT.that[OF refl])+

next

case WTs

from WTs.premis show thesis

by(cases (no-simp))(assumption|erule (2) WTs.that[OF refl])+

qed

notation infer-type ( $-, - \vdash - :: 'l$  - [51,51,51]100)

definition test1 where test1 = [], Map.empty  $\vdash$  testExpr1 :: -

definition test2 where test2 = [], Map.empty  $\vdash$  testExpr2 :: -

definition test3 where test3 = [], Map.empty("V"  $\mapsto$  Integer)  $\vdash$  testExpr3 :: -

definition test4 where test4 = [], Map.empty("V"  $\mapsto$  Integer)  $\vdash$  testExpr4 :: -

definition test5 where test5 = [classObject, ("C", ("Object", [("F", Integer)], [])), Map.empty  $\vdash$  testExpr5 :: -

definition test6 where test6 = [classObject, classI], Map.empty  $\vdash$  testExpr6 :: -

ML-val  $\langle$

val SOME(@{code Integer}, -) = Predicate.yield @{code test1};

val SOME(@{code Integer}, -) = Predicate.yield @{code test2};

val SOME(@{code Integer}, -) = Predicate.yield @{code test3};

val SOME(@{code Void}, -) = Predicate.yield @{code test4};

```

    val SOME(@{code Void}, -) = Predicate.yield @>{code test5};
    val SOME(@{code Integer}, -) = Predicate.yield @>{code test6};
  >

```

**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-val** <

```

    val SOME(@{code Boolean}, -) = Predicate.yield @>{code testmb-isNull};
    val SOME(@{code Integer}, -) = Predicate.yield @>{code testmb-add};
    val SOME(@{code Boolean}, -) = Predicate.yield @>{code testmb-mult-cond};
    val SOME(@{code Void}, -) = Predicate.yield @>{code testmb-mult-block};
    val SOME(@{code Integer}, -) = Predicate.yield @>{code testmb-mult};
  >

```

**definition** *test* **where** *test* = [classObject, classA], Map.empty ⊢ *testExpr-ClassA* :: -

**ML-val** <

```

    val SOME(@{code Integer}, -) = Predicate.yield @>{code test};
  >

```

**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**

<i>instr</i> = <i>Load nat</i>	— load from local variable
<i>Store nat</i>	— store into local variable
<i>Push val</i>	— push a value (constant)
<i>New cname</i>	— create object
<i>Getfield vname cname</i>	— Fetch field from object
<i>Putfield vname cname</i>	— Set field in object
<i>Checkcast cname</i>	— Check whether object is of given type
<i>Invoke mname nat</i>	— 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 C<sub>0</sub> M<sub>0</sub> pc frs* =  
  (*None*, *h*, ((*loc ! n*) # *stk*, *loc*, *C<sub>0</sub>*, *M<sub>0</sub>*, *pc+1*)#*frs*)

| *exec-instr (Store n) P h stk loc C<sub>0</sub> M<sub>0</sub> pc frs* =  
  (*None*, *h*, (*tl stk*, *loc[n:=hd stk]*, *C<sub>0</sub>*, *M<sub>0</sub>*, *pc+1*)#*frs*)

| *exec-instr-Push*:

*exec-instr (Push v) P h stk loc C<sub>0</sub> M<sub>0</sub> pc frs* =  
  (*None*, *h*, (*v* # *stk*, *loc*, *C<sub>0</sub>*, *M<sub>0</sub>*, *pc+1*)#*frs*)

| *exec-instr-New*:

*exec-instr (New C) P h stk loc C<sub>0</sub> M<sub>0</sub> pc frs* =



(*case new-Addr h of*  
*None*  $\Rightarrow$  (*Some (addr-of-sys-xcpt OutOfMemory)*, *h*, (*stk*, *loc*, *C*<sub>0</sub>, *M*<sub>0</sub>, *pc*)#*frs*)  
| *Some a*  $\Rightarrow$  (*None*, *h(a* $\mapsto$ *blank P C)*, (*Addr a*#*stk*, *loc*, *C*<sub>0</sub>, *M*<sub>0</sub>, *pc+1*)#*frs*)

| *exec-instr (Getfield F C) P h stk loc C*<sub>0</sub> *M*<sub>0</sub> *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*, *C*<sub>0</sub>, *M*<sub>0</sub>, *pc+1*)#*frs*)

| *exec-instr (Putfield F C) P h stk loc C*<sub>0</sub> *M*<sub>0</sub> *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*, *C*<sub>0</sub>, *M*<sub>0</sub>, *pc+1*)#*frs*)

| *exec-instr (Checkcast C) P h stk loc C*<sub>0</sub> *M*<sub>0</sub> *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*, *C*<sub>0</sub>, *M*<sub>0</sub>, *pc+1*)#*frs*)

| *exec-instr-Invoke*:  
*exec-instr (Invoke M n) P h stk loc C*<sub>0</sub> *M*<sub>0</sub> *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,mxl<sub>0</sub>,ins,xt*) = *method P C M*;  
*f'* = ( $\square$ , [*r*]@(*rev ps*)@(*replicate mxl<sub>0</sub> undefined*), *D,M,0*)  
*in (xp'*, *h*, *f'*#(*stk*, *loc*, *C*<sub>0</sub>, *M*<sub>0</sub>, *pc*)#*frs*)

| *exec-instr Return P h stk loc<sub>0</sub> C*<sub>0</sub> *M*<sub>0</sub> *pc frs* =  
(*if frs*= $\square$  *then* (*None*, *h*,  $\square$ ) *else*  
*let v* = *hd stk<sub>0</sub>*;  
(*stk,loc,C,m,pc*) = *hd frs*;  
*n* = *length (fst (snd (method P C<sub>0</sub> M<sub>0</sub>)))*  
*in (None*, *h*, (*v*)#(*drop (n+1) stk*), *loc,C,m,pc+1*)#*tl frs*)

| *exec-instr Pop P h stk loc C*<sub>0</sub> *M*<sub>0</sub> *pc frs* =  
(*None*, *h*, (*tl stk*, *loc*, *C*<sub>0</sub>, *M*<sub>0</sub>, *pc+1*)#*frs*)

| *exec-instr IAdd P h stk loc C*<sub>0</sub> *M*<sub>0</sub> *pc frs* =  
(*let i*<sub>2</sub> = *the-Intg (hd stk)*;  
*i*<sub>1</sub> = *the-Intg (hd (tl stk))*  
*in (None*, *h*, (*Intg (i*<sub>1</sub>+*i*<sub>2</sub>)#(*tl (tl stk)*), *loc*, *C*<sub>0</sub>, *M*<sub>0</sub>, *pc+1*)#*frs*)

| *exec-instr (IfFalse i) P h stk loc C*<sub>0</sub> *M*<sub>0</sub> *pc frs* =  
(*let pc'* = *if hd stk = Bool False then nat(int pc+i)* *else pc+1*  
*in (None*, *h*, (*tl stk*, *loc*, *C*<sub>0</sub>, *M*<sub>0</sub>, *pc'*)#*frs*)

| *exec-instr CmpEq P h stk loc C*<sub>0</sub> *M*<sub>0</sub> *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*  $(\text{Store } n) P h (v\#\text{stk}) \text{loc } C_0 M_0 \text{ pc } \text{frs} =$   
 $(\text{None}, h, (\text{stk}, \text{loc}[n:=v], C_0, M_0, \text{pc}+1)\#\text{frs})$   
**by** *simp*

**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  $(xp', h, (\text{the}(\text{fs}(F, C))\#\text{stk}, \text{loc}, C_0, M_0, \text{pc}+1)\#\text{frs}))$   
**by** *simp*

**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  $(xp', h', (\text{stk}, \text{loc}, C_0, M_0, \text{pc}+1)\#\text{frs}))$   
**by** *simp*

**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  $(xp', h, (v\#\text{stk}, \text{loc}, C_0, M_0, \text{pc}+1)\#\text{frs}))$   
**by** *simp*

**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  $(\text{None}, h, (v\#(\text{drop } (n+1) \text{stk}), \text{loc}, C, m, \text{pc}+1)\#\text{tl } \text{frs}))$   
**by** *simp*

**lemma** *exec-instr-IPop*:

*exec-instr*  $\text{Pop } P h (v\#\text{stk}) \text{loc } C_0 M_0 \text{ pc } \text{frs} =$   
 $(\text{None}, h, (\text{stk}, \text{loc}, C_0, M_0, \text{pc}+1)\#\text{frs})$   
**by** *simp*

**lemma** *exec-instr-IAdd*:

*exec-instr IAdd*  $P h (Intg\ i_2 \# Intg\ i_1 \# stk) loc\ C_0\ M_0\ pc\ frs =$   
 $(None, h, (Intg\ (i_1+i_2)\#stk, loc, C_0, M_0, pc+1)\#frs)$   
**by** *simp*

**lemma** *exec-instr-IfFalse*:

*exec-instr IfFalse*  $i P h (v\#stk) loc\ C_0\ M_0\ pc\ frs =$   
 $(let\ pc' = if\ v = Bool\ False\ then\ nat(int\ pc+i)\ else\ pc+1$   
 $in\ (None, h, (stk, loc, C_0, M_0, pc')\#frs))$   
**by** *simp*

**lemma** *exec-instr-CmpEq*:

*exec-instr CmpEq*  $P h (v_2\#v_1\#stk) loc\ C_0\ M_0\ pc\ frs =$   
 $(None, h, (Bool\ (v_1=v_2) \# stk, loc, C_0, M_0, pc+1)\#frs)$   
**by** *simp*

**lemma** *exec-instr-Throw*:

*exec-instr Throw*  $P h (v\#stk) loc\ C_0\ M_0\ pc\ frs =$   
 $(let\ xp' = if\ v = Null\ then\ [addr-of-sys-xcpt\ NullPointer] \ else\ [the-Addr\ v]$   
 $in\ (xp', h, (v\#stk, loc, C_0, M_0, pc)\#frs))$   
**by** *simp*

**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*  $\vdash$   $\sigma$  -*jvm*  $\rightarrow_1$   $\sigma'$   $\equiv$  ( $\sigma, \sigma'$ )  $\in$  *exec-1 P*

| *exec-1I*: *exec* (*P*,  $\sigma$ ) = *Some*  $\sigma'$   $\Longrightarrow$  *P*  $\vdash$   $\sigma$  -*jvm*  $\rightarrow_1$   $\sigma'$

— 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*  $\vdash$   $\sigma$  -*jvm*  $\rightarrow$   $\sigma'$   $\iff$  ( $\sigma, \sigma'$ )  $\in$  (*exec-1 P*)<sup>\*</sup>

**notation** (*ASCII*)

*exec-all* (-  $\vdash$  / - - *jvm*  $\rightarrow$  / - [61,61,61] 60)

**lemma** *exec-1-eq*:

*exec-1 P* = {( $\sigma, \sigma'$ ). *exec* (*P*,  $\sigma$ ) = *Some*  $\sigma'$ }

**lemma** *exec-1-iff*:

*P*  $\vdash$   $\sigma$  -*jvm*  $\rightarrow_1$   $\sigma'$  = (*exec* (*P*,  $\sigma$ ) = *Some*  $\sigma'$ )

**lemma** *exec-all-def*:

*P*  $\vdash$   $\sigma$  -*jvm*  $\rightarrow$   $\sigma'$  = (( $\sigma, \sigma'$ )  $\in$  {( $\sigma, \sigma'$ ). *exec* (*P*,  $\sigma$ ) = *Some*  $\sigma'$ }<sup>\*</sup>)

**lemma** *jvm-refl*[*iff*]:  $P \vdash \sigma \text{-jvm} \rightarrow \sigma$

**lemma** *jvm-trans*[*trans*]:

$\llbracket P \vdash \sigma \text{-jvm} \rightarrow \sigma'; P \vdash \sigma' \text{-jvm} \rightarrow \sigma'' \rrbracket \Longrightarrow P \vdash \sigma \text{-jvm} \rightarrow \sigma''$

**lemma** *jvm-one-step1*[*trans*]:

$\llbracket P \vdash \sigma \text{-jvm} \rightarrow_1 \sigma'; P \vdash \sigma' \text{-jvm} \rightarrow \sigma'' \rrbracket \Longrightarrow P \vdash \sigma \text{-jvm} \rightarrow \sigma''$

**lemma** *jvm-one-step2*[*trans*]:

$\llbracket P \vdash \sigma \text{-jvm} \rightarrow \sigma'; P \vdash \sigma' \text{-jvm} \rightarrow_1 \sigma'' \rrbracket \Longrightarrow P \vdash \sigma \text{-jvm} \rightarrow \sigma''$

**lemma** *exec-all-conf*:

$\llbracket P \vdash \sigma \text{-jvm} \rightarrow \sigma'; P \vdash \sigma \text{-jvm} \rightarrow \sigma'' \rrbracket$   
 $\Longrightarrow P \vdash \sigma' \text{-jvm} \rightarrow \sigma'' \vee P \vdash \sigma'' \text{-jvm} \rightarrow \sigma'$

**lemma** *exec-all-finalD*:  $P \vdash (x, h, []) \text{-jvm} \rightarrow \sigma \Longrightarrow \sigma = (x, h, [])$

**lemma** *exec-all-deterministic*:

$\llbracket P \vdash \sigma \text{-jvm} \rightarrow (x, h, []); P \vdash \sigma \text{-jvm} \rightarrow \sigma' \rrbracket \Longrightarrow P \vdash \sigma' \text{-jvm} \rightarrow (x, h, [])$

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*  $P$   $C$   $M$  =

(*let* ( $D, Ts, T, mxs, mxl_0, b$ ) = *method*  $P$   $C$   $M$  *in*

(*None*, *start-heap*  $P$ ,  $\llbracket [], \text{Null} \# \text{replicate } mxl_0 \text{ undefined}, C, M, 0 \rrbracket$ ))

**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*  $\sigma = (let\ (xcpt,\ h,\ frs) = \sigma\ in$   
 $(case\ frs\ of\ [] \Rightarrow True \mid (stk,loc,C,M,pc)\#frs' \Rightarrow$   
 $P \vdash C\ has\ M \wedge$   
 $(let\ (C',Ts,T,mxs,mxl_0,ins,xt) = method\ P\ C\ M; i = ins!pc\ in$   
 $pc < size\ ins \wedge size\ stk \leq mxs \wedge$   
 $check\_instr\ i\ P\ h\ stk\ loc\ C\ M\ pc\ frs'))$

**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*:

$$\text{exec-1-d } P = \{(s,t). \exists \sigma. s = \text{Normal } \sigma \wedge t = \text{TypeError} \wedge \text{exec-d } P \sigma = \text{TypeError}\} \cup \\ \{(s,t). \exists \sigma \sigma'. s = \text{Normal } \sigma \wedge t = \text{Normal } \sigma' \wedge \text{exec-d } P \sigma = \text{Normal } (\text{Some } \sigma')\}$$

**by** (*auto elim!*: *exec-1-d.cases intro!*: *exec-1-d.intros*)

**declare** *split-paired-All* [*simp del*]

**declare** *split-paired-Ex* [*simp del*]

**lemma** *if-neq* [*dest!*]:

$$(if\ P\ then\ A\ else\ B) \neq B \implies P$$

**by** (*cases P, auto*)

**lemma** *exec-d-no-errorI* [*intro*]:

$$check\ P\ \sigma \implies \text{exec-d } P\ \sigma \neq \text{TypeError}$$

**by** (*unfold exec-d-def simp*)

**theorem** *no-type-error-commutes*:

$$\text{exec-d } P\ \sigma \neq \text{TypeError} \implies \text{exec-d } P\ \sigma = \text{Normal } (\text{exec } (P, \sigma))$$

**by** (*unfold exec-d-def, auto*)

**lemma** *defensive-imp-aggressive*:

$$P \vdash (\text{Normal } \sigma) -jvmd \rightarrow (\text{Normal } \sigma') \implies P \vdash \sigma -jvm \rightarrow \sigma'$$

**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**

$$list-name == "list"$$

**definition** *test-name* :: *string*

**where**

$$test-name == "test"$$

**definition** *val-name* :: *string*

**where**

$$val-name == "val"$$

**definition** *next-name* :: *string*

**where**

$$next-name == "next"$$



**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*,





76

>

**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*]:  $order\ r\ A \Longrightarrow x \in A \Longrightarrow x \sqsubseteq_r x$

**lemma** *order-antisym*:  $\llbracket order\ r\ A; x \sqsubseteq_r y; y \sqsubseteq_r x; x \in A; y \in A \rrbracket \Longrightarrow x = y$

**lemma** *order-trans*:  $\llbracket order\ r\ A; x \sqsubseteq_r y; y \sqsubseteq_r z; x \in A; y \in A; z \in A \rrbracket \Longrightarrow x \sqsubseteq_r z$

**lemma** *order-less-irrefl* [*intro*, *simp*]:  $\text{order } r \ A \implies x \in A \implies \neg x \sqsubseteq_r x$

**lemma** *order-less-trans*:  $\llbracket \text{order } r \ A; x \sqsubseteq_r y; y \sqsubseteq_r z; x \in A; y \in A; z \in A \rrbracket \implies x \sqsubseteq_r z$

**lemma** *topD* [*simp*, *intro*]:  $\text{top } r \ T \implies x \sqsubseteq_r T$

**lemma** *top-le-conv* [*simp*]:  $\llbracket \text{order } r \ A; \text{top } r \ T; x \in A; T \in A \rrbracket \implies (T \sqsubseteq_r x) = (x = T)$

**lemma** *semilat-Def*:

$\text{semilat}(A, r, f) \iff \text{order } r \ A \wedge \text{closed } A \ f \wedge$   
 $(\forall x \in A. \forall y \in A. x \sqsubseteq_r x \sqcup_f y) \wedge$   
 $(\forall x \in A. \forall y \in A. y \sqsubseteq_r x \sqcup_f y) \wedge$   
 $(\forall x \in A. \forall y \in A. \forall z \in A. x \sqsubseteq_r z \wedge y \sqsubseteq_r z \longrightarrow x \sqcup_f y \sqsubseteq_r z)$

**lemma** (**in** *Semilat*) *orderI* [*simp*, *intro*]:  $\text{order } r \ A$

**lemma** (**in** *Semilat*) *closedI* [*simp*, *intro*]:  $\text{closed } A \ f$

**lemma** *closedD*:  $\llbracket \text{closed } A \ f; x \in A; y \in A \rrbracket \implies x \sqcup_f y \in A$

**lemma** *closed-UNIV* [*simp*]:  $\text{closed } UNIV \ f$

**lemma** (**in** *Semilat*) *closed-f* [*simp*, *intro*]:  $\llbracket x \in A; y \in A \rrbracket \implies x \sqcup_f y \in A$

**lemma** (**in** *Semilat*) *refl-r* [*intro*, *simp*]:  $x \in A \implies x \sqsubseteq_r x$  **by** *auto*

**lemma** (**in** *Semilat*) *antisym-r* [*intro?*]:  $\llbracket x \sqsubseteq_r y; y \sqsubseteq_r x; x \in A; y \in A \rrbracket \implies x = y$

**lemma** (**in** *Semilat*) *trans-r* [*trans*, *intro?*]:  $\llbracket x \sqsubseteq_r y; y \sqsubseteq_r z; x \in A; y \in A; z \in A \rrbracket \implies x \sqsubseteq_r z$

**lemma** (**in** *Semilat*) *ub1* [*simp*, *intro?*]:  $\llbracket x \in A; y \in A \rrbracket \implies x \sqsubseteq_r x \sqcup_f y$

**lemma** (**in** *Semilat*) *ub2* [*simp*, *intro?*]:  $\llbracket x \in A; y \in A \rrbracket \implies y \sqsubseteq_r x \sqcup_f y$

**lemma** (**in** *Semilat*) *lub* [*simp*, *intro?*]:

$\llbracket x \sqsubseteq_r z; y \sqsubseteq_r z; x \in A; y \in A; z \in A \rrbracket \implies x \sqcup_f y \sqsubseteq_r z$

**lemma** (**in** *Semilat*) *plus-le-conv* [*simp*]:

$\llbracket x \in A; y \in A; z \in A \rrbracket \implies (x \sqcup_f y \sqsubseteq_r z) = (x \sqsubseteq_r z \wedge y \sqsubseteq_r z)$

**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**  $?P \iff ?Q$ )

**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**  $?P \iff ?Q$ )

**lemma** (**in** *Semilat*) *plus-assoc* [*simp*]:

**assumes**  $a: a \in A$  **and**  $b: b \in A$  **and**  $c: c \in A$

**shows**  $a \sqcup_f (b \sqcup_f c) = a \sqcup_f b \sqcup_f c$

**lemma** (**in** *Semilat*) *plus-com-lemma*:

$\llbracket a \in A; b \in A \rrbracket \implies a \sqcup_f b \sqsubseteq_r b \sqcup_f a$

**lemma** (**in** *Semilat*) *plus-commutative*:

$\llbracket a \in A; b \in A \rrbracket \implies a \sqcup_f b = b \sqcup_f a$

**lemma** *is-lubD*:

$$is-lub\ r\ x\ y\ u \implies is-ub\ r\ x\ y\ u \wedge (\forall z. is-ub\ r\ x\ y\ z \longrightarrow (u,z) \in r)$$

**lemma** *is-ubI*:

$$\llbracket (x,u) \in r; (y,u) \in r \rrbracket \implies is-ub\ r\ x\ y\ u$$

**lemma** *is-ubD*:

$$is-ub\ r\ x\ y\ u \implies (x,u) \in r \wedge (y,u) \in r$$

**lemma** *is-lub-bigger1* [iff]:

$$is-lub\ (r\hat{*})\ x\ y\ y = ((x,y) \in r\hat{*})$$

**lemma** *is-lub-bigger2* [iff]:

$$is-lub\ (r\hat{*})\ x\ y\ x = ((y,x) \in r\hat{*})$$

**lemma** *extend-lub*:

$$\llbracket single-valued\ r; is-lub\ (r\hat{*})\ x\ y\ u; (x',x) \in r \rrbracket \\ \implies \exists v. is-lub\ (r\hat{*})\ x'\ y\ v$$

**lemma** *single-valued-has-lubs* [rule-format]:

$$\llbracket single-valued\ r; (x,u) \in r\hat{*} \rrbracket \implies (\forall y. (y,u) \in r\hat{*} \longrightarrow \\ (\exists z. is-lub\ (r\hat{*})\ x\ y\ z))$$

**lemma** *some-lub-conv*:

$$\llbracket acyclic\ r; is-lub\ (r\hat{*})\ x\ y\ u \rrbracket \implies some-lub\ (r\hat{*})\ x\ y = u$$

**lemma** *is-lub-some-lub*:

$$\llbracket single-valued\ r; acyclic\ r; (x,u) \in r\hat{*}; (y,u) \in r\hat{*} \rrbracket \\ \implies is-lub\ (r\hat{*})\ x\ y\ (some-lub\ (r\hat{*})\ x\ y)$$

### 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*

by (*simp* add: *exec-lub-def* *while-unfold*)

**lemma** *acyclic-single-valued-finite*:

$$\llbracket acyclic\ r; single-valued\ r; (x,y) \in r^* \rrbracket \\ \implies finite\ (r \cap \{a. (x, a) \in r^*\} \times \{b. (b, y) \in r^*\})$$

**lemma** *exec-lub-conv*:

$$\llbracket acyclic\ r; \forall x\ y. (x,y) \in r \longrightarrow f\ x = y; is-lub\ (r^*)\ x\ y\ u \rrbracket \implies \\ exec-lub\ r\ f\ x\ y = u$$

**lemma** *is-lub-exec-lub*:

$$\llbracket single-valued\ r; acyclic\ r; (x,u):r\hat{*}; (y,u):r\hat{*}; \forall x\ y. (x,y) \in r \longrightarrow f\ x = y \rrbracket \\ \implies is-lub\ (r\hat{*})\ x\ y\ (exec-lub\ r\ f\ x\ y)$$

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 e<sub>1</sub> e<sub>2</sub> =  
(case e<sub>1</sub> of Err  $\Rightarrow$  Err | OK x  $\Rightarrow$  (case e<sub>2</sub> of Err  $\Rightarrow$  Err | OK y  $\Rightarrow$  f x y))

**definition** le :: 'a ord  $\Rightarrow$  'a err ord

**where**

le r e<sub>1</sub> e<sub>2</sub> =  
(case e<sub>2</sub> of Err  $\Rightarrow$  True | OK y  $\Rightarrow$  (case e<sub>1</sub> 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.  $\exists y \in A. x = OK y$ }

**lemma** strict-Some [simp]:

(strict f x = OK y) = ( $\exists z. x = OK z \wedge f z = OK y$ )

**lemma** *not-Err-eq*:  $(x \neq \text{Err}) = (\exists a. x = \text{OK } a)$   
**lemma** *not-OK-eq*:  $(\forall y. x \neq \text{OK } y) = (x = \text{Err})$   
**lemma** *unfold-lesub-err*:  $e1 \sqsubseteq_{le\ r} e2 = le\ r\ e1\ e2$   
**lemma** *le-err-refl*:  $\forall x. x \sqsubseteq_r x \implies e \sqsubseteq_{le\ r} e$   
**lemma** *le-err-refl'*:  $(\forall x \in A. x \sqsubseteq_r x) \implies e \in \text{err } A \implies e \sqsubseteq_{le\ r} e$

### 4.3 More about Options

**theory** *Opt* **imports** *Err* **begin**

**definition** *le* :: 'a ord  $\Rightarrow$  'a option 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 set  $\Rightarrow$  'a option set

**where**

$opt\ A = \text{insert } \text{None } \{\text{Some } y \mid y. y \in A\}$

**definition** *sup* :: 'a ebinop  $\Rightarrow$  'a option 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 esl  $\Rightarrow$  'a option esl

**where**

$esl = (\lambda(A,r,f). (opt\ A, le\ r, sup\ f))$

**lemma** *unfold-le-opt*:

$o_1 \sqsubseteq_{le\ r} o_2 =$   
 $(\text{case } o_2 \text{ of } \text{None} \Rightarrow o_1 = \text{None} \mid$   
 $\text{Some } y \Rightarrow (\text{case } o_1 \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } x \Rightarrow x \sqsubseteq_r y))$

**lemma** *le-opt-refl*:  $\text{order } r\ A \implies x \in \text{opt } A \implies x \sqsubseteq_{le\ r} x$

### 4.4 Products as Semilattices

**theory** *Product*

**imports** *Err*

**begin**

**definition** *le* :: 'a ord  $\Rightarrow$  'b ord  $\Rightarrow$  ('a  $\times$  'b) 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 ebinop  $\Rightarrow$  'b ebinop  $\Rightarrow$  ('a  $\times$  'b) 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 esl  $\Rightarrow$  'b esl  $\Rightarrow$  ('a  $\times$  'b) 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*:  $x \sqsubseteq(r_A, r_B) y = le\ r_A\ r_B\ x\ y$

**lemma** *le-prod-Pair-conv* [iff]:  $((a_1, b_1) \sqsubseteq(r_A, r_B) (a_2, b_2)) = (a_1 \sqsubseteq_{r_A} a_2 \ \&\ b_1 \sqsubseteq_{r_B} b_2)$

**lemma** *less-prod-Pair-conv*:

$((a_1, b_1) \sqsubseteq_{Product.le\ r_A\ r_B} (a_2, b_2)) =$   
 $(a_1 \sqsubseteq_{r_A} a_2 \ \&\ b_1 \sqsubseteq_{r_B} b_2 \mid a_1 \sqsubseteq_{r_A} a_2 \ \&\ b_1 \sqsubseteq_{r_B} b_2)$

**lemma** *order-le-prodI* [iff]:  $(order\ r_A\ A \ \&\ order\ r_B\ B) \Longrightarrow order\ (Product.le\ r_A\ r_B)\ (A \times B)$

**apply** *(unfold order-def)*

**apply** *safe*

**apply** *blast+*

**done**

**lemma** *order-le-prodE*:  $A \neq \{\} \Longrightarrow B \neq \{\} \Longrightarrow order\ (Product.le\ r_A\ r_B)\ (A \times B) \Longrightarrow (order\ r_A\ A$   
 $\ \&\ order\ r_B\ B)$

**apply** *(unfold order-def)*

**apply** *simp*

**apply** *safe*

**apply** *blast+*

**done**

**lemma** *order-le-prod* [iff]:  $A \neq \{\} \Longrightarrow B \neq \{\} \Longrightarrow order\ (Product.le\ r_A\ r_B)\ (A \times B) = (order\ r_A\ A$   
 $\ \&\ order\ r_B\ B)$

**lemma** *acc-le-prodI* [intro!]:

$\llbracket acc\ r_A; acc\ r_B \rrbracket \Longrightarrow acc\ (Product.le\ r_A\ r_B)$

**lemma** *closed-lift2-sup*:

$\llbracket closed\ (err\ A)\ (lift2\ f); closed\ (err\ B)\ (lift2\ g) \rrbracket \Longrightarrow$   
 $closed\ (err\ (A \times B))\ (lift2\ (sup\ f\ g))$

**lemma** *unfold-plussub-lift2*:  $e_1 \sqcup_{lift2\ f} e_2 = lift2\ f\ e_1\ e_2$

**lemma** *plus-eq-Err-conv* [simp]:

**assumes**  $x \in A\ y \in A\ semilat\ (err\ A, Err.le\ r, lift2\ f)$

**shows**  $(x \sqcup_f y = Err) = (\neg(\exists z \in A. x \sqsubseteq_r z \ \&\ y \sqsubseteq_r z))$

**lemma** *err-semilat-Product-esl*:

$\bigwedge L_1\ L_2. \llbracket err-semilat\ L_1; err-semilat\ L_2 \rrbracket \Longrightarrow err-semilat\ (Product.esl\ L_1\ L_2)$

**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) **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) **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  
 ((- / $\sqcup$ -) [65, 0, 66] 65) **where**  
 $x \sqcup_f y == x \sqcup_{map2\ f}\ y$

**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 = ( $\lambda xs\ ys.$  if size xs = size ys then *coalesce*(xs  $\sqcup_f$  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  $\sqsubseteq_r$  ys = Listn.le r xs ys

**lemma** *Nil-le-conv* [*iff*]: ([]  $\sqsubseteq_r$  ys) = (ys = [])

**lemma** *Cons-notle-Nil* [*iff*]:  $\neg x\#xs \sqsubseteq_r []$

**lemma** *Cons-le-Cons* [*iff*]:  $x\#xs \sqsubseteq_r y\#ys = (x \sqsubseteq_r y \wedge xs \sqsubseteq_r ys)$

**lemma** *list-update-le-cong*:

$[i < \text{size } xs; xs \sqsubseteq_r ys; x \sqsubseteq_r y] \Longrightarrow xs[i:=x] \sqsubseteq_r ys[i:=y]$

**lemma** *le-listD*:  $[xs \sqsubseteq_r ys; p < \text{size } xs] \Longrightarrow xs!p \sqsubseteq_r ys!p$

**lemma** *le-list-refl*:  $\forall x. x \sqsubseteq_r x \Longrightarrow xs \sqsubseteq_r xs$

**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  $\sqsubseteq_r$  ys **and** ys  $\sqsubseteq_r$  zs

**shows** xs  $\sqsubseteq_r$  zs

## 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**

$'s \text{ step-type} = \text{nat} \Rightarrow 's \Rightarrow (\text{nat} \times 's) \text{ list}$

**definition**  $\text{stable} :: 's \text{ ord} \Rightarrow 's \text{ step-type} \Rightarrow 's \text{ list} \Rightarrow \text{nat} \Rightarrow \text{bool}$

**where**

$\text{stable } r \text{ step } \tau s \ p \longleftrightarrow (\forall (q, \tau) \in \text{set } (\text{step } p \ (\tau s!p)). \tau \sqsubseteq_r \tau s!q)$

**definition**  $\text{stables} :: 's \text{ ord} \Rightarrow 's \text{ step-type} \Rightarrow 's \text{ list} \Rightarrow \text{bool}$

**where**

$\text{stables } r \text{ step } \tau s \longleftrightarrow (\forall p < \text{size } \tau s. \text{stable } r \text{ step } \tau s \ p)$

**definition**  $\text{wt-step} :: 's \text{ ord} \Rightarrow 's \Rightarrow 's \text{ step-type} \Rightarrow 's \text{ list} \Rightarrow \text{bool}$

**where**

$\text{wt-step } r \ T \ \text{step } \tau s \longleftrightarrow (\forall p < \text{size } \tau s. \tau s!p \neq T \wedge \text{stable } r \ \text{step } \tau s \ p)$

**end**

## 4.7 More on Semilattices

**theory** *SemilatAlg*

**imports** *Typing-Framework-1*

**begin**

**definition**  $\text{lesubstep-type} :: (\text{nat} \times 's) \text{ set} \Rightarrow 's \text{ ord} \Rightarrow (\text{nat} \times 's) \text{ set} \Rightarrow \text{bool}$

$((- / \{\sqsubseteq_r\} -) [50, 0, 51] 50)$

**where**  $A \{\sqsubseteq_r\} B \equiv \forall (p, \tau) \in A. \exists \tau'. (p, \tau') \in B \wedge \tau \sqsubseteq_r \tau'$

**notation** (*ASCII*)

$\text{lesubstep-type } ((- / \{\leq_r\} -) [50, 0, 51] 50)$

**primrec**  $\text{pluslssub} :: 'a \text{ list} \Rightarrow ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \ ((- / \sqcup -) [65, 0, 66] 65)$

**where**

$\text{pluslssub } [] \ f \ y = y$

$| \text{pluslssub } (x \# xs) \ f \ y = \text{pluslssub } xs \ f \ (x \sqcup_f y)$

**definition**  $\text{bounded} :: 's \text{ step-type} \Rightarrow \text{nat} \Rightarrow \text{bool}$

**where**

$\text{bounded } \text{step } n \longleftrightarrow (\forall p < n. \forall \tau. \forall (q, \tau') \in \text{set } (\text{step } p \ \tau). q < n)$

**definition**  $\text{pres-type} :: 's \text{ step-type} \Rightarrow \text{nat} \Rightarrow 's \text{ set} \Rightarrow \text{bool}$

**where**

$\text{pres-type } \text{step } n \ A \longleftrightarrow (\forall \tau \in A. \forall p < n. \forall (q, \tau') \in \text{set } (\text{step } p \ \tau). \tau' \in A)$

**definition**  $\text{mono} :: 's \text{ ord} \Rightarrow 's \text{ step-type} \Rightarrow \text{nat} \Rightarrow 's \text{ set} \Rightarrow \text{bool}$

**where**

$\text{mono } r \ \text{step } n \ A \longleftrightarrow$

$(\forall \tau \ p \ \tau'. \tau \in A \wedge p < n \wedge \tau \sqsubseteq_r \tau' \longrightarrow \text{set } (\text{step } p \ \tau) \{\sqsubseteq_r\} \text{set } (\text{step } p \ \tau'))$

**lemma**  $[\text{iff}]: \{\} \{\sqsubseteq_r\} B$

**lemma**  $[\text{iff}]: (A \{\sqsubseteq_r\} \{\}) = (A = \{\})$

**lemma** *lesubstep-union*:

$$\llbracket A_1 \{\sqsubseteq_r\} B_1; A_2 \{\sqsubseteq_r\} B_2 \rrbracket \Longrightarrow A_1 \cup A_2 \{\sqsubseteq_r\} B_1 \cup B_2$$

**lemma** *pres-typeD*:

$$\llbracket \text{pres-type step } n \ A; s \in A; p < n; (q, s') \in \text{set } (\text{step } p \ s) \rrbracket \Longrightarrow s' \in A$$

**lemma** *monoD*:

$$\llbracket \text{mono } r \ \text{step } n \ A; p < n; s \in A; s \sqsubseteq_r t \rrbracket \Longrightarrow \text{set } (\text{step } p \ s) \{\sqsubseteq_r\} \text{set } (\text{step } p \ t)$$

**lemma** *boundedD*:

$$\llbracket \text{bounded step } n; p < n; (q, t) \in \text{set } (\text{step } p \ xs) \rrbracket \Longrightarrow q < n$$

**lemma** *lesubstep-type-refl* [*simp, intro*]:

$$(\bigwedge x. x \sqsubseteq_r x) \Longrightarrow A \{\sqsubseteq_r\} A$$

**lemma** *lesub-step-typeD*:

$$A \{\sqsubseteq_r\} B \Longrightarrow (x, y) \in A \Longrightarrow \exists y'. (x, y') \in B \wedge y \sqsubseteq_r y'$$

**lemma** *list-update-le-listI* [*rule-format*]:

$$\begin{aligned} \text{set } xs \subseteq A \longrightarrow \text{set } ys \subseteq A \longrightarrow xs \{\sqsubseteq_r\} ys \longrightarrow p < \text{size } xs \longrightarrow \\ x \sqsubseteq_r ys!p \longrightarrow \text{semilat}(A, r, f) \longrightarrow x \in A \longrightarrow \\ xs[p := x \sqcup_f xs!p] \{\sqsubseteq_r\} ys \end{aligned}$$

**lemma** *plusplus-closed*: **assumes** *Semilat A r f* **shows**

$$\bigwedge y. \llbracket \text{set } x \subseteq A; y \in A \rrbracket \Longrightarrow x \sqcup_f y \in A$$

**lemma** (**in** *Semilat*) *pp-ub2*:

$$\bigwedge y. \llbracket \text{set } x \subseteq A; y \in A \rrbracket \Longrightarrow y \sqsubseteq_r x \sqcup_f y$$

**lemma** (**in** *Semilat*) *pp-ub1*:

**shows**  $\bigwedge y. \llbracket \text{set } ls \subseteq A; y \in A; x \in \text{set } ls \rrbracket \Longrightarrow x \sqsubseteq_r ls \sqcup_f y$

**lemma** (**in** *Semilat*) *pp-lub*:

**assumes**  $z: z \in A$

**shows**

$$\bigwedge y. y \in A \Longrightarrow \text{set } xs \subseteq A \Longrightarrow \forall x \in \text{set } xs. x \sqsubseteq_r z \Longrightarrow y \sqsubseteq_r z \Longrightarrow xs \sqcup_f y \sqsubseteq_r z$$

**lemma** *ub1'*: **assumes** *Semilat A r f*

**shows**  $\llbracket \forall (p, s) \in \text{set } S. s \in A; y \in A; (a, b) \in \text{set } S \rrbracket$

$$\Longrightarrow b \sqsubseteq_r \text{map } \text{snd } [(p', t') \leftarrow S. p' = a] \sqcup_f y$$

**lemma** *plusplus-empty*:

$$\begin{aligned} \forall s'. (q, s') \in \text{set } S \longrightarrow s' \sqcup_f ss ! q = ss ! q \Longrightarrow \\ (\text{map } \text{snd } [(p', t') \leftarrow S. p' = q] \sqcup_f ss ! q) = ss ! q \end{aligned}$$

**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**

$$map-snd f = map (\lambda(x,y). (x, f y))$$

**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**

$$\begin{aligned} err-step n app step p t = \\ (case t of \\ \quad Err \Rightarrow error n \\ \quad | OK \tau \Rightarrow if app p \tau then map-snd OK (step p \tau) else error n) \end{aligned}$$

**definition**  $app-mono :: 's ord \Rightarrow (nat \Rightarrow 's \Rightarrow bool) \Rightarrow nat \Rightarrow 's set \Rightarrow bool$

**where**

$$\begin{aligned} app-mono r app n A \iff \\ (\forall s p t. s \in A \wedge p < n \wedge s \sqsubseteq_r t \longrightarrow app p t \longrightarrow app p s) \end{aligned}$$

**lemmas**  $err-step-defs = err-step-def map-snd-def error-def$

**lemma**  $bounded-err-stepD$ :

$$\llbracket bounded (err-step n app step) n; \\ p < n; app p a; (q,b) \in set (step p a) \rrbracket \implies q < n$$

**lemma**  $in-map-sndD$ :  $(a,b) \in set (map-snd f xs) \implies \exists b'. (a,b') \in set xs$

**lemma**  $bounded-err-stepI$ :

$$\begin{aligned} \forall p. p < n \longrightarrow (\forall s. app p s \longrightarrow (\forall (q,s') \in set (step p s). q < n)) \\ \implies bounded (err-step n app step) n \end{aligned}$$

**lemma**  $bounded-lift$ :

$$bounded step n \implies bounded (err-step n app step) n$$

**lemma**  $le-list-map-OK [simp]$ :

$$\bigwedge b. (map OK a [\sqsubseteq_{Err.le} r] map OK b) = (a [\sqsubseteq_r] b)$$

**lemma**  $map-snd-lessI$ :

$$set xs \{\sqsubseteq_r\} set ys \implies set (map-snd OK xs) \{\sqsubseteq_{Err.le} r\} set (map-snd OK ys)$$

**lemma**  $mono-lift$ :

$$\begin{aligned} \llbracket order r A; app-mono r app n A; bounded (err-step n app step) n; \\ \forall s p t. s \in A \wedge p < n \wedge s \sqsubseteq_r t \longrightarrow app p t \longrightarrow set (step p s) \{\sqsubseteq_r\} set (step p t) \rrbracket \\ \implies mono (Err.le r) (err-step n app step) n (err A) \end{aligned}$$

**lemma**  $in-errorD$ :  $(x,y) \in set (error n) \implies y = Err$

**lemma**  $pres-type-lift$ :

$$\begin{aligned} \forall s \in A. \forall p. p < n \longrightarrow app p s \longrightarrow (\forall (q, s') \in set (step p s). s' \in A) \\ \implies pres-type (err-step n app step) n (err A) \end{aligned}$$

**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*)

**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*)

**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*:

$\bigwedge ss. [p < \text{length } ss; ss \in \text{nlists } n \ A; \forall (p, t) \in \text{set } ps. p < n \wedge t \in A] \implies$   
 $(\text{merges } f \ ps \ ss)!p = \text{map } \text{snd} \ [(p', t') \leftarrow ps. p' = p] \ \sqcup_f \ ss!p$   
**(is**  $\bigwedge ss. [ -; -; ?\text{steptype } ps] \implies ?P \ ss \ ps)$

**lemma** *length-merges* [*simp*]:

$\bigwedge ss. \text{size}(\text{merges } f \ ps \ ss) = \text{size } ss$

**lemma** (**in** *Semilat*) *merges-preserves-type-lemma*:

**shows**  $\forall xs. xs \in \text{nlists } n \ A \longrightarrow (\forall (p, x) \in \text{set } ps. p < n \wedge x \in A)$   
 $\longrightarrow \text{merges } f \ ps \ xs \in \text{nlists } n \ A$

**lemma** (**in** *Semilat*) *merges-preserves-type* [*simp*]:

$[xs \in \text{nlists } n \ A; \forall (p, x) \in \text{set } ps. p < n \wedge x \in A]$   
 $\implies \text{merges } f \ ps \ xs \in \text{nlists } n \ A$

**by** (*simp add: merges-preserves-type-lemma*)

**lemma** (**in** *Semilat*) *list-update-le-listI* [*rule-format*]:

$\text{set } xs \subseteq A \longrightarrow \text{set } ys \subseteq A \longrightarrow xs \ [\sqsubseteq_r] \ ys \longrightarrow p < \text{size } xs \longrightarrow$   
 $x \ \sqsubseteq_r \ ys!p \longrightarrow x \in A \longrightarrow xs[p := x \sqcup_f xs!p] \ [\sqsubseteq_r] \ ys$

**lemma** (**in** *Semilat*) *merges-pres-le-ub*:

**assumes**  $\text{set } ts \subseteq A \ \text{set } ss \subseteq A$   
 $\forall (p, t) \in \text{set } ps. t \ \sqsubseteq_r \ ts!p \wedge t \in A \wedge p < \text{size } ts \ \text{ss} \ [\sqsubseteq_r] \ ts$   
**shows**  $\text{merges } f \ ps \ ss \ [\sqsubseteq_r] \ ts$

**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* []  $\tau s$  *w* = ( $\tau s, w$ )  
| *propa* *f* (*q*'#*qs*)  $\tau s$  *w* = (let (*q*, $\tau$ ) = *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*  $\tau s$  *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* ( $\tau s!p$ ))  $\tau s$  ( $w - \{p\}$ ))  
( $\tau s, w$ )

**definition** *unstables* :: 's ord  $\Rightarrow$  's step-type  $\Rightarrow$  's list  $\Rightarrow$  nat set

**where**

*unstables* *r* *step*  $\tau s$  = {*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*  $\tau s$  = *fst*(*iter* *f* *step*  $\tau s$  (*unstables* *r* *step*  $\tau s$ ))

**lemma** (in *Semilat*) *merges-incr-lemma*:

$\forall xs. xs \in \text{nlists } n \ A \longrightarrow (\forall (p,x) \in \text{set } ps. p < \text{size } xs \wedge x \in A) \longrightarrow xs \ [\sqsubseteq_r] \ \text{merges } f \ ps \ xs$

**apply** (*induct* *ps*)

**apply** *simp*

**apply**(*insert orderI*)

**apply** (*fastforce intro:le-list-refl'*)

**apply** *simp*

**apply** *clarify*

**apply** (*rule order-trans*)

**defer**

**apply** (*erule list-update-incr*)

**apply** *simp+*

**done**

**lemma** (in *Semilat*) *merges-incr*:

$\llbracket xs \in \text{nlists } n \ A; \forall (p,x) \in \text{set } ps. p < \text{size } xs \wedge x \in A \rrbracket$   
 $\implies xs \ [\sqsubseteq_r] \ \text{merges } f \ ps \ xs$

by (*simp add: merges-incr-lemma*)

**lemma** (in *Semilat*) *merges-same-conv* [*rule-format*]:  
 $(\forall xs. xs \in nlists\ n\ A \longrightarrow (\forall (p,x) \in set\ ps. p < size\ xs \wedge x \in A) \longrightarrow$   
 $(merges\ f\ ps\ xs = xs) = (\forall (p,x) \in set\ ps. x \sqsubseteq_r\ xs!p))$

**lemma** *decomp-propa*:

$\bigwedge ss\ w. (\forall (q,t) \in set\ qs. q < size\ ss) \implies$   
 $propa\ f\ qs\ ss\ w =$   
 $(merges\ f\ qs\ ss, \{q. \exists t. (q,t) \in set\ qs \wedge t \sqcup_f\ ss!q \neq ss!q\} \cup w)$

**lemma** (in *Semilat*) *stable-pres-lemma*:

**shows**  $\llbracket pres\text{-}type\ step\ n\ A; bounded\ step\ n;$   
 $ss \in nlists\ n\ A; p \in w; \forall q \in w. q < n;$   
 $\forall q. q < n \longrightarrow q \notin w \longrightarrow stable\ r\ step\ ss\ q; q < n;$   
 $\forall s'. (q,s') \in set\ (step\ p\ (ss!p)) \longrightarrow s' \sqcup_f\ ss!q = ss!q;$   
 $q \notin w \vee q = p \rrbracket$   
 $\implies stable\ r\ step\ (merges\ f\ (step\ p\ (ss!p))\ ss)\ q$

**lemma** (in *Semilat*) *merges-bounded-lemma*:

$\llbracket mono\ r\ step\ n\ A; bounded\ step\ n; pres\text{-}type\ step\ n\ A;$   
 $\forall (p',s') \in set\ (step\ p\ (ss!p)). s' \in A; ss \in nlists\ n\ A; ts \in nlists\ n\ A; p < n;$   
 $ss \sqsubseteq_r\ ts; \forall p. p < n \longrightarrow stable\ r\ step\ ts\ p \rrbracket$   
 $\implies merges\ f\ (step\ p\ (ss!p))\ ss \sqsubseteq_r\ ts$

**lemma** *termination-lemma*: **assumes** *Semilat* *A* *r* *f*

**shows**  $\llbracket ss \in nlists\ n\ A; \forall (q,t) \in set\ qs. q < n \wedge t \in A; p \in w \rrbracket \implies$   
 $ss \sqsubseteq_r\ merges\ f\ qs\ ss \vee$   
 $merges\ f\ qs\ ss = ss \wedge \{q. \exists t. (q,t) \in set\ qs \wedge t \sqcup_f\ ss!q \neq ss!q\} \cup (w - \{p\}) \subset w$   
**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**

$merge\ cert\ f\ r\ T\ pc\ []\ x = x$   
 $| merge\ cert\ f\ r\ T\ pc\ (s\#\#ss)\ x = merge\ cert\ f\ r\ T\ pc\ ss\ (let\ (pc',s') = s\ in$   
 $\quad if\ pc' = pc + 1\ then\ s' \sqcup_f\ x$   
 $\quad else\ if\ s' \sqsubseteq_r\ cert!pc'\ then\ x$   
 $\quad else\ T)$

**definition** *wtl-inst* :: *'s* *certificate*  $\Rightarrow$  *'s* *binop*  $\Rightarrow$  *'s* *ord*  $\Rightarrow$  *'s*  $\Rightarrow$

*'s* *step-type*  $\Rightarrow$  *nat*  $\Rightarrow$  *'s*  $\Rightarrow$  *'s*

**where**

$wtl\text{-}inst\ cert\ f\ r\ T\ step\ pc\ s = merge\ cert\ f\ r\ T\ pc\ (step\ pc\ s)\ (cert!(pc+1))$

**definition**  $wtl\text{-cert} :: 's \text{ certificate} \Rightarrow 's \text{ binop} \Rightarrow 's \text{ ord} \Rightarrow 's \Rightarrow 's \Rightarrow 's \text{ step-type} \Rightarrow \text{nat} \Rightarrow 's \Rightarrow 's$

**where**

$wtl\text{-cert} \text{ cert } f \ r \ T \ B \ \text{step} \ pc \ s =$   
 (if  $\text{cert!pc} = B$  then  
 $wtl\text{-inst} \text{ cert } f \ r \ T \ \text{step} \ pc \ s$   
 else  
 if  $s \sqsubseteq_r \text{cert!pc}$  then  $wtl\text{-inst} \text{ cert } f \ r \ T \ \text{step} \ pc \ (\text{cert!pc})$  else  $T$ )

**primrec**  $wtl\text{-inst-list} :: 'a \text{ list} \Rightarrow 's \text{ certificate} \Rightarrow 's \text{ binop} \Rightarrow 's \text{ ord} \Rightarrow 's \Rightarrow 's \Rightarrow 's \text{ step-type} \Rightarrow \text{nat} \Rightarrow 's \Rightarrow 's$

**where**

$wtl\text{-inst-list} [] \ \text{cert } f \ r \ T \ B \ \text{step} \ pc \ s = s$   
 $| wtl\text{-inst-list} (i\#\text{is}) \ \text{cert } f \ r \ T \ B \ \text{step} \ pc \ s =$   
 (let  $s' = wtl\text{-cert} \ \text{cert } f \ r \ T \ B \ \text{step} \ pc \ s$  in  
 if  $s' = T \vee s = T$  then  $T$  else  $wtl\text{-inst-list} \ \text{is} \ \text{cert } f \ r \ T \ B \ \text{step} \ (pc+1) \ s'$ )

**definition**  $\text{cert-ok} :: 's \text{ certificate} \Rightarrow \text{nat} \Rightarrow 's \Rightarrow 's \Rightarrow 's \text{ set} \Rightarrow \text{bool}$

**where**

$\text{cert-ok} \ \text{cert} \ n \ T \ B \ A \longleftrightarrow (\forall i < n. \text{cert!}i \in A \wedge \text{cert!}i \neq T) \wedge (\text{cert!}n = B)$

**definition**  $\text{bottom} :: 'a \text{ ord} \Rightarrow 'a \Rightarrow \text{bool}$

**where**

$\text{bottom} \ r \ B \longleftrightarrow (\forall x. B \sqsubseteq_r x)$

**locale**  $lbv = \text{Semilat} +$

**fixes**  $T :: 'a \ (\top)$

**fixes**  $B :: 'a \ (\perp)$

**fixes**  $\text{step} :: 'a \ \text{step-type}$

**assumes**  $\text{top}: \text{top} \ r \ \top$

**assumes**  $T\text{-}A: \top \in A$

**assumes**  $\text{bot}: \text{bottom} \ r \ \perp$

**assumes**  $B\text{-}A: \perp \in A$

**fixes**  $\text{merge} :: 'a \text{ certificate} \Rightarrow \text{nat} \Rightarrow (\text{nat} \times 'a) \text{ list} \Rightarrow 'a \Rightarrow 'a$

**defines**  $\text{mrg-def}: \text{merge} \ \text{cert} \equiv \text{LBVSpec.merge} \ \text{cert} \ f \ r \ \top$

**fixes**  $\text{wti} :: 'a \text{ certificate} \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a$

**defines**  $\text{wti-def}: \text{wti} \ \text{cert} \equiv wtl\text{-inst} \ \text{cert} \ f \ r \ \top \ \text{step}$

**fixes**  $\text{wtc} :: 'a \text{ certificate} \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a$

**defines**  $\text{wtc-def}: \text{wtc} \ \text{cert} \equiv wtl\text{-cert} \ \text{cert} \ f \ r \ \top \ \perp \ \text{step}$

**fixes**  $\text{wtl} :: 'b \text{ list} \Rightarrow 'a \text{ certificate} \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a$

**defines**  $\text{wtl-def}: \text{wtl} \ \text{ins} \ \text{cert} \equiv wtl\text{-inst-list} \ \text{ins} \ \text{cert} \ f \ r \ \top \ \perp \ \text{step}$

**lemma** (in  $lbv$ )  $\text{wti}$ :

$\text{wti} \ c \ pc \ s = \text{merge} \ c \ pc \ (\text{step} \ pc \ s) \ (c!(pc+1))$

**lemma** (in  $lbv$ )  $\text{wtc}$ :

$\text{wtc} \ c \ pc \ s = (\text{if } c!pc = \perp \text{ then } \text{wti} \ c \ pc \ s \text{ else if } s \sqsubseteq_r c!pc \text{ then } \text{wti} \ c \ pc \ (c!pc) \text{ else } \top)$

**lemma** *cert-okD1* [intro?]:

*cert-ok c n T B A*  $\implies pc < n \implies c!pc \in A$

**lemma** *cert-okD2* [intro?]:

*cert-ok c n T B A*  $\implies c!n = B$

**lemma** *cert-okD3* [intro?]:

*cert-ok c n T B A*  $\implies B \in A \implies pc < n \implies c!Suc\ pc \in A$

**lemma** *cert-okD4* [intro?]:

*cert-ok c n T B A*  $\implies pc < n \implies c!pc \neq T$

**declare** *Let-def* [simp]

#### 4.11.1 more semilattice lemmas

**lemma** (in *lbu*) *sup-top* [simp, elim]:

**assumes** *x*:  $x \in A$

**shows**  $x \sqcup_f \top = \top$

**lemma** (in *lbu*) *plusplusup-top* [simp, elim]:

*set xs*  $\subseteq A \implies xs \sqcup_f \top = \top$

**by** (*induct xs*) *auto*

**lemma** (in *Semilat*) *pp-ub1'*:

**assumes** *S*:  $snd'set\ S \subseteq A$

**assumes** *y*:  $y \in A$  **and** *ab*:  $(a, b) \in set\ S$

**shows**  $b \sqsubseteq_r\ map\ snd\ [(p', t') \leftarrow S . p' = a] \sqcup_f y$

**lemma** (in *lbu*) *bottom-le* [simp, intro!]:  $\perp \sqsubseteq_r x$

**by** (*insert bot*) (*simp add: bottom-def*)

**lemma** (in *lbu*) *le-bottom* [simp]:  $x \in A \implies x \sqsubseteq_r \perp = (x = \perp)$

**using** *B-A* **by** (*blast intro: antisym-r*)

#### 4.11.2 merge

**lemma** (in *lbu*) *merge-Nil* [simp]:

*merge c pc [] x = x* **by** (*simp add: mrg-def*)

**lemma** (in *lbu*) *merge-Cons* [simp]:

*merge c pc (l#ls) x = merge c pc ls* (if  $fst\ l = pc + 1$  then  $snd\ l + -f\ x$   
 else if  $snd\ l \sqsubseteq_r\ c!fst\ l$  then  $x$   
 else  $\top$ )

**by** (*simp add: mrg-def split-beta*)

**lemma** (in *lbu*) *merge-Err* [simp]:

*snd'set ss*  $\subseteq A \implies merge\ c\ pc\ ss\ \top = \top$

**by** (*induct ss*) *auto*

**lemma** (in *lbu*) *merge-not-top*:

$\bigwedge x. snd'set\ ss \subseteq A \implies merge\ c\ pc\ ss\ x \neq \top \implies$

$\forall (pc', s') \in set\ ss. (pc' \neq pc + 1 \longrightarrow s' \sqsubseteq_r\ c!pc')$

(**is**  $\bigwedge x. ?set\ ss \implies ?merge\ ss\ x \implies ?P\ ss$ )

**lemma** (in *lbv*) *merge-def*:

**shows**

$\bigwedge x. x \in A \implies \text{snd}'\text{set } ss \subseteq A \implies$

$\text{merge } c \text{ pc } ss \ x =$

(if  $\forall (pc', s') \in \text{set } ss. pc' \neq pc+1 \implies s' \sqsubseteq_r c!pc'$  then

$\text{map snd } [(p', t') \leftarrow ss. p' = pc+1] \sqcup_f x$

else  $\top$ )

(is  $\bigwedge x. - \implies - \implies ?\text{merge } ss \ x = ?\text{if } ss \ x \text{ is } \bigwedge x. - \implies - \implies ?P \ ss \ x$ )

**lemma** (in *lbv*) *merge-not-top-s*:

**assumes** *x*:  $x \in A$  **and** *ss*:  $\text{snd}'\text{set } ss \subseteq A$

**assumes** *m*:  $\text{merge } c \text{ pc } ss \ x \neq \top$

**shows**  $\text{merge } c \text{ pc } ss \ x = (\text{map snd } [(p', t') \leftarrow ss. p' = pc+1] \sqcup_f x)$

### 4.11.3 wtl-inst-list

**lemmas** [*iff*] = *not-Err-eq*

**lemma** (in *lbv*) *wtl-Nil* [*simp*]:  $\text{wtl } [] \ c \ \text{pc} \ s = s$

**by** (*simp add: wtl-def*)

**lemma** (in *lbv*) *wtl-Cons* [*simp*]:

$\text{wtl } (i\#is) \ c \ \text{pc} \ s =$

(let  $s' = \text{wtc } c \ \text{pc} \ s$  in if  $s' = \top \vee s = \top$  then  $\top$  else  $\text{wtl } is \ c \ (\text{pc}+1) \ s'$ )

**by** (*simp add: wtl-def wtc-def*)

**lemma** (in *lbv*) *wtl-Cons-not-top*:

$\text{wtl } (i\#is) \ c \ \text{pc} \ s \neq \top =$

$(\text{wtc } c \ \text{pc} \ s \neq \top \wedge s \neq \top \wedge \text{wtl } is \ c \ (\text{pc}+1) \ (\text{wtc } c \ \text{pc} \ s) \neq \top)$

**by** (*auto simp del: split-paired-Ex*)

**lemma** (in *lbv*) *wtl-top* [*simp*]:  $\text{wtl } ls \ c \ \text{pc} \ \top = \top$

**by** (*cases ls*) *auto*

**lemma** (in *lbv*) *wtl-not-top*:

$\text{wtl } ls \ c \ \text{pc} \ s \neq \top \implies s \neq \top$

**by** (*cases s = \top*) *auto*

**lemma** (in *lbv*) *wtl-append* [*simp*]:

$\bigwedge \text{pc } s. \text{wtl } (a@b) \ c \ \text{pc} \ s = \text{wtl } b \ c \ (\text{pc} + \text{length } a) \ (\text{wtl } a \ c \ \text{pc} \ s)$

**by** (*induct a*) *auto*

**lemma** (in *lbv*) *wtl-take*:

$\text{wtl } is \ c \ \text{pc} \ s \neq \top \implies \text{wtl } (\text{take } pc' \ is) \ c \ \text{pc} \ s \neq \top$

(is  $?wtl \ is \ \neq \ - \implies \ -$ )

**lemma** *take-Suc*:

$\forall n. n < \text{length } l \implies \text{take } (\text{Suc } n) \ l = (\text{take } n \ l)@[l!n]$  (is  $?P \ l$ )

**lemma** (in *lbv*) *wtl-Suc*:

**assumes** *suc*:  $pc+1 < \text{length } is$

**assumes** *wtl*:  $\text{wtl } (\text{take } pc \ is) \ c \ 0 \ s \neq \top$

**shows**  $\text{wtl } (\text{take } (\text{pc}+1) \ is) \ c \ 0 \ s = \text{wtc } c \ \text{pc} \ (\text{wtl } (\text{take } pc \ is) \ c \ 0 \ s)$

**lemma** (in *lbv*) *wtl-all*:

**assumes** *all*:  $\text{wtl } is \ c \ 0 \ s \neq \top$  (is  $?wtl \ is \ \neq \ -$ )

**assumes** *pc*:  $pc < \text{length } is$

**shows**  $wtc\ c\ pc\ (wtl\ (take\ pc\ is)\ c\ 0\ s) \neq \top$

#### 4.11.4 preserves-type

**lemma** (in *lbv*) *merge-pres*:

**assumes**  $s0: snd'set\ ss \subseteq A$  **and**  $x: x \in A$

**shows**  $merge\ c\ pc\ ss\ x \in A$

**lemma** *pres-typeD2*:

*pres-type step n A*  $\implies s \in A \implies p < n \implies snd'set\ (step\ p\ s) \subseteq A$

**by** *auto* (*drule pres-typeD*)

**lemma** (in *lbv*) *wti-pres* [*intro?*]:

**assumes** *pres*: *pres-type step n A*

**assumes** *cert*:  $c!(pc+1) \in A$

**assumes** *s-pc*:  $s \in A\ pc < n$

**shows**  $wti\ c\ pc\ s \in A$

**lemma** (in *lbv*) *wtc-pres*:

**assumes** *pres-type step n A*

**assumes**  $c!pc \in A$  **and**  $c!(pc+1) \in A$

**assumes**  $s \in A$  **and**  $pc < n$

**shows**  $wtc\ c\ pc\ s \in A$

**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**  $?len\ pc \implies ?wtl\ pc \in A$ )

**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?*]:

$\llbracket pc < size\ ins; c!pc = \perp \rrbracket \implies \tau s!pc = wtl\ (take\ pc\ ins)\ c\ 0\ s_0$

**lemma** (in *lbvs*) *phi-Some* [*intro?*]:

$\llbracket pc < size\ ins; c!pc \neq \perp \rrbracket \implies \tau s!pc = c!pc$

**lemma** (in *lbvs*) *phi-len* [*simp*]:  $size\ \tau s = size\ ins$

**lemma** (in *lbs*) *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  $wtl\ (take\ (pc+1)\ ins)\ c\ 0\ s_0 \sqsubseteq_r \tau s!(pc+1)$

**lemma** (in *lbs*) *wtl-stable*:  
 assumes *wtl*:  $wtl\ ins\ c\ 0\ s_0 \neq \top$   
 assumes *s0*:  $s_0 \in A$  and *pc*:  $pc < size\ ins$   
 shows  $stable\ r\ step\ \tau s\ pc$

**lemma** (in *lbs*) *phi-not-top*:  
 assumes *wtl*:  $wtl\ ins\ c\ 0\ s_0 \neq \top$  and *pc*:  $pc < size\ ins$   
 shows  $\tau s!pc \neq \top$

**lemma** (in *lbs*) *phi-in-A*:  
 assumes *wtl*:  $wtl\ ins\ c\ 0\ s_0 \neq \top$  and *s0*:  $s_0 \in A$   
 shows  $\tau s \in nlists\ (size\ ins)\ A$

**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  $s_0 \sqsubseteq_r \tau s!0$

**theorem** (in *lbs*) *wtl-sound*:  
 assumes *wtl*:  $wtl\ ins\ c\ 0\ s_0 \neq \top$  and *s0*:  $s_0 \in A$   
 shows  $\exists \tau s. wt\text{-}step\ r\ \top\ step\ \tau s$

**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  $\exists \tau s \in nlists\ (size\ ins)\ A. wt\text{-}step\ r\ \top\ step\ \tau s \wedge s_0 \sqsubseteq_r \tau s!0$

end

## 4.13 Completeness of the LBV

**theory** *LBVComplete*  
**imports** *LBVSpec Typing-Framework-1*  
**begin**

**definition** *is-target* ::  $'s\ step\text{-}type \Rightarrow 's\ list \Rightarrow nat \Rightarrow bool$  **where**  
*is-target*  $step\ \tau s\ pc' \iff (\exists pc\ s'. pc' \neq pc+1 \wedge pc < size\ \tau s \wedge (pc', s') \in set\ (step\ pc\ (\tau s!pc)))$

**definition** *make-cert* ::  $'s\ step\text{-}type \Rightarrow 's\ list \Rightarrow 's \Rightarrow 's\ certificate$  **where**  
*make-cert*  $step\ \tau s\ B = map\ (\lambda pc. if\ is\text{-}target\ step\ \tau s\ pc\ then\ \tau s!pc\ else\ B)\ [0..<size\ \tau s]\ @\ [B]$

**lemma** [*code*]:  
*is-target*  $step\ \tau s\ pc' =$   
*list-ex*  $(\lambda pc. pc' \neq pc+1 \wedge List.member\ (map\ fst\ (step\ pc\ (\tau s!pc)))\ pc')\ [0..<size\ \tau s]$

**locale** *lbv* = *lbv* +  
**fixes**  $\tau s :: 'a\ list$   
**fixes**  $c :: 'a\ list$   
**defines** *cert-def*:  $c \equiv make\text{-}cert\ step\ \tau s\ \perp$

**assumes** *mono*:  $mono\ r\ step\ (size\ \tau s)\ A$   
**assumes** *pres*:  $pres\text{-}type\ step\ (size\ \tau s)\ A$   
**assumes**  $\tau s$ :  $\forall pc < size\ \tau s. \tau s!pc \in A \wedge \tau s!pc \neq \top$   
**assumes** *bounded*:  $bounded\ step\ (size\ \tau s)$

assumes  $B\text{-neg-}T: \perp \neq \top$

lemma (in *lbvc*) *cert*: *cert-ok*  $c$  (*size*  $\tau s$ )  $\top \perp A$

lemmas [*simp del*] = *split-paired-Ex*

lemma (in *lbvc*) *cert-target* [*intro?*]:

[[  $(pc', s') \in \text{set } (\text{step } pc \ (\tau s!pc))$ ;  
 $pc' \neq pc+1$ ;  $pc < \text{size } \tau s$ ;  $pc' < \text{size } \tau s$  ]]  
 $\implies c!pc' = \tau s!pc'$

lemma (in *lbvc*) *cert-approx* [*intro?*]:

[[  $pc < \text{size } \tau s$ ;  $c!pc \neq \perp$  ]]  $\implies c!pc = \tau s!pc$

lemma (in *lbv*) *le-top* [*simp, intro*]:  $x \leq_r \top$

lemma (in *lbv*) *merge-mono*:

assumes *less*:  $\text{set } ss_2 \sqsubseteq_r \text{set } ss_1$

assumes *x*:  $x \in A$

assumes *ss1*:  $\text{snd}'\text{set } ss_1 \subseteq A$

assumes *ss2*:  $\text{snd}'\text{set } ss_2 \subseteq A$

assumes *boun*:  $\forall x \in (\text{fst}'\text{set } ss_1). x < \text{size } \tau s$

assumes *cert*: *cert-ok*  $c$  (*size*  $\tau s$ )  $T B A$

shows *merge*  $c$   $pc$   $ss_2$   $x \sqsubseteq_r \text{merge } c$   $pc$   $ss_1$   $x$  (is  $?s_2 \sqsubseteq_r ?s_1$ )

lemma (in *lbvc*) *wti-mono*:

assumes *less*:  $s_2 \sqsubseteq_r s_1$

assumes *pc*:  $pc < \text{size } \tau s$  and *s1*:  $s_1 \in A$  and *s2*:  $s_2 \in A$

shows *wti*  $c$   $pc$   $s_2 \sqsubseteq_r \text{wti } c$   $pc$   $s_1$  (is  $?s_2' \sqsubseteq_r ?s_1'$ )

lemma (in *lbvc*) *wtc-mono*:

assumes *less*:  $s_2 \sqsubseteq_r s_1$

assumes *pc*:  $pc < \text{size } \tau s$  and *s1*:  $s_1 \in A$  and *s2*:  $s_2 \in A$

shows *wtc*  $c$   $pc$   $s_2 \sqsubseteq_r \text{wtc } c$   $pc$   $s_1$  (is  $?s_2' \sqsubseteq_r ?s_1'$ )

lemma (in *lbv*) *top-le-conv* [*simp*]:  $x \in A \implies \top \sqsubseteq_r x = (x = \top)$

lemma (in *lbv*) *neg-top* [*simp, elim*]: [[  $x \sqsubseteq_r y$ ;  $y \neq \top$ ;  $y \in A$  ]]  $\implies x \neq \top$

lemma (in *lbvc*) *stable-wti*:

assumes *stable*: *stable*  $r$  *step*  $\tau s$   $pc$  and *pc*:  $pc < \text{size } \tau s$

shows *wti*  $c$   $pc$   $(\tau s!pc) \neq \top$

lemma (in *lbvc*) *wti-less*:

assumes *stable*: *stable*  $r$  *step*  $\tau s$   $pc$  and *suc-pc*: *Suc*  $pc < \text{size } \tau s$

shows *wti*  $c$   $pc$   $(\tau s!pc) \sqsubseteq_r \tau s! \text{Suc } pc$  (is  $?wti \sqsubseteq_r -$ )

lemma (in *lbvc*) *stable-wtc*:

assumes *stable*: *stable*  $r$  *step*  $\tau s$   $pc$  and *pc*:  $pc < \text{size } \tau s$

shows *wtc*  $c$   $pc$   $(\tau s!pc) \neq \top$

lemma (in *lbvc*) *wtc-less*:

assumes *stable*: *stable*  $r$  *step*  $\tau s$   $pc$  and *suc-pc*: *Suc*  $pc < \text{size } \tau s$

shows *wtc*  $c$   $pc$   $(\tau s!pc) \sqsubseteq_r \tau s! \text{Suc } pc$  (is  $?wtc \sqsubseteq_r -$ )

lemma (in *lbvc*) *wt-step-wtl-lemma*:

assumes *wt-step*: *wt-step*  $r$   $\top$  *step*  $\tau s$

shows  $\bigwedge pc \ s. pc + \text{size } ls = \text{size } \tau s \implies s \sqsubseteq_r \tau s!pc \implies s \in A \implies s \neq \top \implies$

$\text{wtl } ls \ c \ pc \ s \neq \top$

(is  $\bigwedge pc \ s. - \implies - \implies - \implies - \implies ?wtl \ ls \ pc \ s \neq -$ )

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*  $ins = \text{size } \tau s$

shows *wtl*  $ins \ c \ 0 \ s \neq \top$

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*:  
 $(C,D) \in \text{subcls1 } P \implies \text{super } P C = D$   
**by** (unfold *super-def*) (auto dest: *subcls1D*)

**primrec** *the-Class* :: ty  $\Rightarrow$  cname  
**where**  
*the-Class* (Class C) = C

**definition** *sup* :: 'c prog  $\Rightarrow$  ty  $\Rightarrow$  ty  $\Rightarrow$  ty err  
**where**  
 $\text{sup } P T_1 T_2 \equiv$   
 if *is-refT* T<sub>1</sub>  $\wedge$  *is-refT* T<sub>2</sub> then  
 OK (if T<sub>1</sub> = NT then T<sub>2</sub> else  
   if T<sub>2</sub> = NT then T<sub>1</sub> else  
   (Class (exec-lub (subcls1 P) (super P) (the-Class T<sub>1</sub>) (the-Class T<sub>2</sub>))))  
 else  
 (if T<sub>1</sub> = T<sub>2</sub> then OK T<sub>1</sub> else Err)

**lemma** *sup-def'*:  
 $\text{sup } P = (\lambda T_1 T_2.$   
 if *is-refT* T<sub>1</sub>  $\wedge$  *is-refT* T<sub>2</sub> then  
 OK (if T<sub>1</sub> = NT then T<sub>2</sub> else  
   if T<sub>2</sub> = NT then T<sub>1</sub> else  
   (Class (exec-lub (subcls1 P) (super P) (the-Class T<sub>1</sub>) (the-Class T<sub>2</sub>))))  
 else  
 (if T<sub>1</sub> = T<sub>2</sub> then OK T<sub>1</sub> else Err))  
**by** (simp add: *sup-def fun-eq-iff*)

**abbreviation**  
*subtype* :: 'c prog  $\Rightarrow$  ty  $\Rightarrow$  ty  $\Rightarrow$  bool  
**where** *subtype* P  $\equiv$  widen P

**definition** *esl* :: 'c prog  $\Rightarrow$  ty esl  
**where**  
*esl* P  $\equiv$  (types P, subtype P, sup P)

**lemma** *is-class-is-subcls*:  
 $\text{wf-prog } m P \implies \text{is-class } P C = P \vdash C \preceq^* \text{Object}$

**lemma** *subcls-antisym*:  
 $\llbracket \text{wf-prog } m P; P \vdash C \preceq^* D; P \vdash D \preceq^* C \rrbracket \implies C = D$

**lemma** *widen-antisym*:

$\llbracket \text{wf-prog } m \ P; P \vdash T \leq U; P \vdash U \leq T \rrbracket \implies T = U$

**lemma** *order-widen* [*intro, simp*]:

$\text{wf-prog } m \ P \implies \text{order } (\text{subtype } P) \ (\text{types } P)$

**lemma** *NT-widen*:

$P \vdash NT \leq T = (T = NT \vee (\exists C. T = \text{Class } C))$

**lemma** *Class-widen2*:  $P \vdash \text{Class } C \leq T = (\exists D. T = \text{Class } D \wedge P \vdash C \preceq^* D)$

**lemma** *wf-converse-subcls1-impl-acc-subtype*:

$\text{wf } ((\text{subcls1 } P)^\wedge - 1) \implies \text{acc } (\text{subtype } P)$

**lemma** *wf-subtype-acc* [*intro, simp*]:

$\text{wf-prog } \text{wf-mb } P \implies \text{acc } (\text{subtype } P)$

**lemma** *exec-lub-refl* [*simp*]:  $\text{exec-lub } r \ f \ T \ T = T$

**lemma** *closed-err-types*:

$\text{wf-prog } \text{wf-mb } P \implies \text{closed } (\text{err } (\text{types } P)) \ (\text{lift2 } (\text{sup } P))$

**lemma** *sup-subtype-greater*:

$\llbracket \text{wf-prog } \text{wf-mb } P; \text{is-type } P \ t1; \text{is-type } P \ t2; \text{sup } P \ t1 \ t2 = \text{OK } s \rrbracket$   
 $\implies \text{subtype } P \ t1 \ s \wedge \text{subtype } P \ t2 \ s$

**lemma** *sup-subtype-smallest*:

$\llbracket \text{wf-prog } \text{wf-mb } P; \text{is-type } P \ a; \text{is-type } P \ b; \text{is-type } P \ c;$   
 $\text{subtype } P \ a \ c; \text{subtype } P \ b \ c; \text{sup } P \ a \ b = \text{OK } d \rrbracket$   
 $\implies \text{subtype } P \ d \ c$

**lemma** *sup-exists*:

$\llbracket \text{subtype } P \ a \ c; \text{subtype } P \ b \ c \rrbracket \implies \exists T. \text{sup } P \ a \ b = \text{OK } T$

**lemma** *err-semilat-JType-esl*:

$\text{wf-prog } \text{wf-mb } P \implies \text{err-semilat } (\text{esl } P)$

**end**

## 4.15 The JVM Type System as Semilattice

**theory** *JVM-SemiType* **imports** *SemiType* **begin**

**type-synonym**  $ty_l = ty \ \text{err} \ \text{list}$

**type-synonym**  $ty_s = ty \ \text{list}$

**type-synonym**  $ty_i = ty_s \times ty_l$

**type-synonym**  $ty_i' = ty_i \ \text{option}$

**type-synonym**  $ty_m = ty_i' \ \text{list}$

**type-synonym**  $ty_P = mname \Rightarrow cname \Rightarrow ty_m$

**definition**  $stk\text{-esl} :: 'c \ \text{prog} \Rightarrow nat \Rightarrow ty_s \ \text{esl}$

**where**

$stk\text{-esl } P \ mxs \equiv upto\text{-esl } mxs \ (\text{SemiType.esl } P)$

**definition**  $loc\text{-sl} :: 'c \ \text{prog} \Rightarrow nat \Rightarrow ty_l \ \text{sl}$

**where**

$loc\text{-sl } P \ mxl \equiv Listn.sl \ mxl \ (\text{Err.sl } (\text{SemiType.esl } P))$

**definition**  $sl :: 'c \text{ prog} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{ty}_i' \text{ err } sl$

**where**

$sl \ P \ mxs \ mxl \equiv$   
 $Err.sl(Opt.esl(Product.esl(stk-esl \ P \ mxs) (Err.esl(loc-sl \ P \ mxl))))$

**definition**  $states :: 'c \text{ prog} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{ty}_i' \text{ err } set$

**where**  $states \ P \ mxs \ mxl \equiv fst(sl \ P \ mxs \ mxl)$

**definition**  $le :: 'c \text{ prog} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{ty}_i' \text{ err } ord$

**where**

$le \ P \ mxs \ mxl \equiv fst(snd(sl \ P \ mxs \ mxl))$

**definition**  $sup :: 'c \text{ prog} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{ty}_i' \text{ err } binop$

**where**

$sup \ P \ mxs \ mxl \equiv snd(snd(sl \ P \ mxs \ mxl))$

**definition**  $sup\text{-}ty\text{-}opt :: ['c \text{ prog}, \text{ty } err, \text{ty } err] \Rightarrow bool$

$(- \vdash - \leq_{\top} - [71, 71, 71] \ 70)$

**where**

$sup\text{-}ty\text{-}opt \ P \equiv Err.le \ (subtype \ P)$

**definition**  $sup\text{-}state :: ['c \text{ prog}, \text{ty}_i, \text{ty}_i] \Rightarrow bool$

$(- \vdash - \leq_i - [71, 71, 71] \ 70)$

**where**

$sup\text{-}state \ P \equiv Product.le \ (Listn.le \ (subtype \ P)) \ (Listn.le \ (sup\text{-}ty\text{-}opt \ P))$

**definition**  $sup\text{-}state\text{-}opt :: ['c \text{ prog}, \text{ty}_i', \text{ty}_i'] \Rightarrow bool$

$(- \vdash - \leq'' - [71, 71, 71] \ 70)$

**where**

$sup\text{-}state\text{-}opt \ P \equiv Opt.le \ (sup\text{-}state \ P)$

**abbreviation**

$sup\text{-}loc :: ['c \text{ prog}, \text{ty}_l, \text{ty}_l] \Rightarrow bool \ (- \vdash - [\leq_{\top}] - [71, 71, 71] \ 70)$

**where**  $P \vdash LT [\leq_{\top}] LT' \equiv list\text{-}all2 \ (sup\text{-}ty\text{-}opt \ P) \ LT \ LT'$

**notation** (*ASCII*)

$sup\text{-}ty\text{-}opt \ (- \mid - \leq = T - [71, 71, 71] \ 70)$  **and**

$sup\text{-}state \ (- \mid - \leq = i - [71, 71, 71] \ 70)$  **and**

$sup\text{-}state\text{-}opt \ (- \mid - \leq = ' - [71, 71, 71] \ 70)$  **and**

$sup\text{-}loc \ (- \mid - [\leq = T] - [71, 71, 71] \ 70)$

#### 4.15.1 Unfolding

**lemma** *JVM-states-unfold*:

$states \ P \ mxs \ mxl \equiv err(opt((Union \ \{nlists \ n \ (types \ P) \ \mid n. \ n \ \leq \ mxs\}) \times$   
 $nlists \ mxl \ (err(types \ P))))$

**lemma** *JVM-le-unfold*:

$le \ P \ m \ n \equiv$

$Err.le(Opt.le(Product.le(Listn.le(subtype \ P))(Listn.le(Err.le(subtype \ P))))))$

**lemma** *sl-def2*:

$JVM\text{-}SemiType.sl \ P \ mxs \ mxl \equiv$

$(states \ P \ mxs \ mxl, \ JVM\text{-}SemiType.le \ P \ mxs \ mxl, \ JVM\text{-}SemiType.sup \ P \ mxs \ mxl)$

**lemma** *JVM-le-conv*:

$$le\ P\ m\ n\ (OK\ t1)\ (OK\ t2) = P \vdash t1 \leq' t2$$

**lemma** *JVM-le-Err-conv*:

$$le\ P\ m\ n = Err.le\ (sup\ state\ opt\ P)$$

**lemma** *err-le-unfold* [*iff*]:

$$Err.le\ r\ (OK\ a)\ (OK\ b) = r\ a\ b$$

### 4.15.2 Semilattice

**lemma** *order-sup-state-opt'* [*intro, simp*]:

$$wf\ prog\ wf\ mb\ P \implies \\ order\ (sup\ state\ opt\ P)\ (opt\ ((\bigcup\ \{nlists\ n\ (types\ P)\ \mid\ n.\ n \leq\ mxs\}\ ) \times\ nlists\ (Suc\ (length\ Ts + mxl_0)))\ (err\ (types\ P))))$$

## 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** *mxs* :: nat

**fixes** *mxl<sub>0</sub>* :: nat

**fixes** *Ts* :: ty list

**fixes** *T<sub>τ</sub>* :: ty

**fixes** *is* :: instr list

**fixes** *xt* :: ex-table

**fixes** *mxl* :: nat

**defines** *mxl-def*:  $mxl \equiv 1 + size\ Ts + mxl_0$

Program counter of successor instructions:

**primrec** *succs* :: instr  $\Rightarrow$  ty<sub>*i*</sub>  $\Rightarrow$  pc  $\Rightarrow$  pc list **where**

$$\begin{aligned} & succs\ (Load\ idx)\ \tau\ pc &= [pc+1] \\ | succs\ (Store\ idx)\ \tau\ pc &= [pc+1] \\ | succs\ (Push\ v)\ \tau\ pc &= [pc+1] \\ | succs\ (Getfield\ F\ C)\ \tau\ pc &= [pc+1] \\ | succs\ (Putfield\ F\ C)\ \tau\ pc &= [pc+1] \\ | succs\ (New\ C)\ \tau\ pc &= [pc+1] \\ | succs\ (Checkcast\ C)\ \tau\ pc &= [pc+1] \\ | succs\ Pop\ \tau\ pc &= [pc+1] \\ | succs\ IAdd\ \tau\ pc &= [pc+1] \\ | succs\ CmpEq\ \tau\ pc &= [pc+1] \\ | succs-IfFalse: & \\ & succs\ (IfFalse\ b)\ \tau\ pc &= [pc+1, nat\ (int\ pc + b)] \\ | succs-Goto: & \\ & succs\ (Goto\ b)\ \tau\ pc &= [nat\ (int\ pc + b)] \\ | succs-Return: & \\ & succs\ Return\ \tau\ pc &= [] \end{aligned}$$

| *succs-Invoke*:  
 $\text{succs } (\text{Invoke } M \ n) \ \tau \ pc = (\text{if } (\text{fst } \tau)!n = NT \ \text{then } [] \ \text{else } [pc+1])$

| *succs-Throw*:  
 $\text{succs } \text{Throw } \tau \ pc = []$

Effect of instruction on the state type:

**fun** *the-class* ::  $ty \Rightarrow cname$  **where**  
*the-class* (*Class* *C*) = *C*

**fun** *eff<sub>i</sub>* ::  $instr \times 'm \ \text{prog} \times ty_i \Rightarrow ty_i$  **where**

*eff<sub>i</sub>-Load*:  
 $\text{eff}_i (\text{Load } n, \ P, \ (ST, \ LT)) = (\text{ok-val } (LT \ ! \ n) \ \# \ ST, \ LT)$

| *eff<sub>i</sub>-Store*:  
 $\text{eff}_i (\text{Store } n, \ P, \ (T\#ST, \ LT)) = (ST, \ LT[n:= \ OK \ T])$

| *eff<sub>i</sub>-Push*:  
 $\text{eff}_i (\text{Push } v, \ P, \ (ST, \ LT)) = (\text{the } (\text{typeof } v) \ \# \ ST, \ LT)$

| *eff<sub>i</sub>-Getfield*:  
 $\text{eff}_i (\text{Getfield } F \ C, \ P, \ (T\#ST, \ LT)) = (\text{snd } (\text{field } P \ C \ F) \ \# \ ST, \ LT)$

| *eff<sub>i</sub>-Putfield*:  
 $\text{eff}_i (\text{Putfield } F \ C, \ P, \ (T_1\#T_2\#ST, \ LT)) = (ST, \ LT)$

| *eff<sub>i</sub>-New*:  
 $\text{eff}_i (\text{New } C, \ P, \ (ST, \ LT)) = (\text{Class } C \ \# \ ST, \ LT)$

| *eff<sub>i</sub>-Checkcast*:  
 $\text{eff}_i (\text{Checkcast } C, \ P, \ (T\#ST, \ LT)) = (\text{Class } C \ \# \ ST, \ LT)$

| *eff<sub>i</sub>-Pop*:  
 $\text{eff}_i (\text{Pop}, \ P, \ (T\#ST, \ LT)) = (ST, \ LT)$

| *eff<sub>i</sub>-IAdd*:  
 $\text{eff}_i (\text{IAdd}, \ P, \ (T_1\#T_2\#ST, \ LT)) = (\text{Integer}\#ST, \ LT)$

| *eff<sub>i</sub>-CmpEq*:  
 $\text{eff}_i (\text{CmpEq}, \ P, \ (T_1\#T_2\#ST, \ LT)) = (\text{Boolean}\#ST, \ LT)$

| *eff<sub>i</sub>-IfFalse*:  
 $\text{eff}_i (\text{IfFalse } b, \ P, \ (T_1\#ST, \ LT)) = (ST, \ LT)$

| *eff<sub>i</sub>-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<sub>i</sub>-Goto*:  
 $\text{eff}_i (\text{Goto } n, \ P, \ s) = s$

**fun** *is-relevant-class* ::  $instr \Rightarrow 'm \ \text{prog} \Rightarrow cname \Rightarrow bool$  **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-Checcast*:  
 $\text{is-relevant-class } (\text{Checkcast } D) = (\lambda P \ C. \ P \vdash \ \text{ClassCast} \ \preceq^* \ C)$

| *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**  
*is-relevant-entry* P i pc e ⇔ (let (f,t,C,h,d) = e in *is-relevant-class* i P C ∧ pc ∈ {f..<t})

**definition** *relevant-entries* :: 'm prog ⇒ instr ⇒ pc ⇒ ex-table ⇒ ex-table **where**  
*relevant-entries* P i pc = filter (*is-relevant-entry* P i pc)

**definition** *xcpt-eff* :: instr ⇒ 'm prog ⇒ pc ⇒ ty<sub>i</sub>  
⇒ ex-table ⇒ (pc × ty<sub>i</sub>') list **where**  
*xcpt-eff* i P pc τ et = (let (ST,LT) = τ in  
map (λ(f,t,C,h,d). (h, Some (Class C#drop (size ST - d) ST, LT))) (*relevant-entries* P i pc et))

**definition** *norm-eff* :: instr ⇒ 'm prog ⇒ nat ⇒ ty<sub>i</sub> ⇒ (pc × ty<sub>i</sub>') list **where**  
*norm-eff* i P pc τ = map (λpc'. (pc',Some (eff<sub>i</sub> (i,P,τ)))) (succs i τ pc)

**definition** *eff* :: instr ⇒ 'm prog ⇒ pc ⇒ ex-table ⇒ ty<sub>i</sub>' ⇒ (pc × ty<sub>i</sub>') list **where**  
*eff* i P pc et t = (case t of  
None ⇒ []  
| Some τ ⇒ (*norm-eff* i P pc τ) @ (*xcpt-eff* i P pc τ et))

**lemma** *eff-None*:

*eff* i P pc xt None = []

**by** (*simp* add: *eff-def*)

**lemma** *eff-Some*:

*eff* i P pc xt (Some τ) = *norm-eff* i P pc τ @ *xcpt-eff* i P pc τ xt

**by** (*simp* add: *eff-def*)

Conditions under which *eff* is applicable:

**fun** *app<sub>i</sub>* :: instr × 'm prog × pc × nat × ty × ty<sub>i</sub> ⇒ bool **where**

*app<sub>i</sub>*-Load:

*app<sub>i</sub>* (Load n, P, pc, mxs, T<sub>r</sub>, (ST,LT)) =  
(n < length LT ∧ LT ! n ≠ Err ∧ length ST < mxs)

| *app<sub>i</sub>*-Store:

*app<sub>i</sub>* (Store n, P, pc, mxs, T<sub>r</sub>, (T#ST, LT)) =  
(n < length LT)

| *app<sub>i</sub>*-Push:

*app<sub>i</sub>* (Push v, P, pc, mxs, T<sub>r</sub>, (ST,LT)) =  
(length ST < mxs ∧ typeof v ≠ None)

| *app<sub>i</sub>*-Getfield:

*app<sub>i</sub>* (Getfield F C, P, pc, mxs, T<sub>r</sub>, (T#ST, LT)) =  
(∃ T<sub>f</sub>. P ⊢ C sees F:T<sub>f</sub> in C ∧ P ⊢ T ≤ Class C)

| *app<sub>i</sub>*-Putfield:

*app<sub>i</sub>* (Putfield F C, P, pc, mxs, T<sub>r</sub>, (T<sub>1</sub>#T<sub>2</sub>#ST, LT)) =  
(∃ T<sub>f</sub>. P ⊢ C sees F:T<sub>f</sub> in C ∧ P ⊢ T<sub>2</sub> ≤ (Class C) ∧ P ⊢ T<sub>1</sub> ≤ T<sub>f</sub>)

| *app<sub>i</sub>*-New:

*app<sub>i</sub>* (New C, P, pc, mxs, T<sub>r</sub>, (ST,LT)) =  
(*is-class* P C ∧ length ST < mxs)

| *app<sub>i</sub>*-Checkcast:

*app<sub>i</sub>* (Checkcast C, P, pc, mxs, T<sub>r</sub>, (T#ST,LT)) =  
(*is-class* P C ∧ *is-refT* T)

| *app<sub>i</sub>*-Pop:

*app<sub>i</sub>* (Pop, P, pc, mxs, T<sub>r</sub>, (T#ST,LT)) =

$True$   
|  $app_i$ -IAdd:  
 $app_i (IAdd, P, pc, mxs, T_r, (T_1\#T_2\#ST,LT)) = (T_1 = T_2 \wedge T_1 = Integer)$   
|  $app_i$ -CmpEq:  
 $app_i (CmpEq, P, pc, mxs, T_r, (T_1\#T_2\#ST,LT)) =$   
 $(T_1 = T_2 \vee is-refT T_1 \wedge is-refT T_2)$   
|  $app_i$ -IfFalse:  
 $app_i (IfFalse b, P, pc, mxs, T_r, (Boolean\#ST,LT)) =$   
 $(0 \leq int pc + b)$   
|  $app_i$ -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:  
 $app_i (Invoke M n, P, pc, mxs, T_r, (ST,LT)) =$   
 $(n < length ST \wedge$   
 $(ST!n \neq NT \rightarrow$   
 $(\exists C D Ts T m. ST!n = Class C \wedge P \vdash C \text{ sees } M:Ts \rightarrow T = m \text{ in } D \wedge$   
 $P \vdash rev (take n ST) [\leq] Ts)))$   
|  $app_i$ -default:  
 $app_i (i,P, pc,mxs,T_r,s) = False$

**definition**  $xcpt$ -app ::  $instr \Rightarrow 'm \text{ prog} \Rightarrow pc \Rightarrow nat \Rightarrow ex\text{-table} \Rightarrow ty_i \Rightarrow bool$  **where**  
 $xcpt$ -app  $i P pc mxs xt \tau \longleftrightarrow (\forall (f,t,C,h,d) \in set (relevant\text{-entries } P i pc xt). is\text{-class } P C \wedge d \leq$   
 $size (fst \tau) \wedge d < mxs)$

**definition**  $app$  ::  $instr \Rightarrow 'm \text{ prog} \Rightarrow nat \Rightarrow ty \Rightarrow nat \Rightarrow nat \Rightarrow ex\text{-table} \Rightarrow ty_i' \Rightarrow bool$  **where**  
 $app i P mxs T_r pc mpc xt t = (case t of None \Rightarrow True | Some \tau \Rightarrow$   
 $app_i (i,P,pc,mxs,T_r,\tau) \wedge xcpt$ -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  $i P pc mxs xt \tau \wedge$   
 $(\forall (pc',s') \in set (eff i P pc xt (Some \tau)). pc' < mpc))$   
**by** ( $simp$  add:  $app$ -def)

**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**  $xcpt$ - $app$   $i$   $pc$   $\tau \equiv Effect.xcpt$ - $app$   $i$   $P$   $pc$   $mxs$   $xt$   $\tau$   
**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**  $P s$   
**by** ( $cases$   $s$ ,  $cases$   $fst$   $s$ ) (*auto* *intro!*: *assms*)

**lemma**  $length$ - $cases3$ :

**assumes**  $\bigwedge LT. P (\[], LT)$   
**assumes**  $\bigwedge l LT. P ([l], LT)$   
**assumes**  $\bigwedge l ST LT. P (l\#ST, LT)$   
**shows**  $P s$

**lemma**  $length$ - $cases4$ :

**assumes**  $\bigwedge LT. P (\[], LT)$   
**assumes**  $\bigwedge l LT. P ([l], LT)$   
**assumes**  $\bigwedge l l' LT. P ([l, l'], LT)$   
**assumes**  $\bigwedge l l' ST LT. P (l\#l'\#ST, LT)$   
**shows**  $P s$

simp rules for  $app$

**lemma**  $appNone$ [ $simp$ ]:  $app$   $i$   $P$   $mxs$   $T_r$   $pc$   $mpc$   $et$   $None = True$   
**by** ( $simp$  *add*:  $app$ - $def$ )

**lemma**  $appLoad$ [ $simp$ ]:

$app_i (Load\ idx, P, T_r, mxs, pc, s) = (\exists ST LT. s = (ST, LT) \wedge idx < length\ LT \wedge LT!idx \neq Err \wedge length\ ST < mxs)$   
**by** ( $cases$   $s$ ,  $simp$ )

**lemma**  $appStore$ [ $simp$ ]:

$app_i (Store\ idx, P, pc, mxs, T_r, s) = (\exists ts ST LT. s = (ts\#ST, LT) \wedge idx < length\ LT)$   
**by** (*rule*  $length$ - $cases2$ , *auto*)



**lemma** *appPush[simp]*:

$app_i (Push\ v,\ P,\ pc,\ mxs,\ T_r,\ s) =$   
 $(\exists\ ST\ LT.\ s = (ST,LT) \wedge length\ ST < mxs \wedge typeof\ v \neq None)$   
**by** (*cases s, simp*)

**lemma** *appGetField[simp]*:

$app_i (Getfield\ F\ C,\ P,\ pc,\ mxs,\ T_r,\ s) =$   
 $(\exists\ oT\ vT\ ST\ LT.\ s = (oT\#\!ST,\ LT) \wedge$   
 $P \vdash C\ sees\ F:\!vT\ in\ C \wedge P \vdash oT \leq (Class\ C))$   
**by** (*rule length-cases2 [of - s] auto*)

**lemma** *appPutField[simp]*:

$app_i (Putfield\ F\ C,\ P,\ pc,\ mxs,\ T_r,\ s) =$   
 $(\exists\ vT\ vT'\ oT\ ST\ LT.\ s = (vT\#\!oT\#\!ST,\ LT) \wedge$   
 $P \vdash C\ sees\ F:\!vT'\ in\ C \wedge P \vdash oT \leq (Class\ C) \wedge P \vdash vT \leq vT')$   
**by** (*rule length-cases4 [of - s], auto*)

**lemma** *appNew[simp]*:

$app_i (New\ C,\ P,\ pc,\ mxs,\ T_r,\ s) =$   
 $(\exists\ ST\ LT.\ s=(ST,LT) \wedge is-class\ P\ C \wedge length\ ST < mxs)$   
**by** (*cases s, simp*)

**lemma** *appCheckcast[simp]*:

$app_i (Checkcast\ C,\ P,\ pc,\ mxs,\ T_r,\ s) =$   
 $(\exists\ T\ ST\ LT.\ s = (T\#\!ST,LT) \wedge is-class\ P\ C \wedge is-refT\ T)$   
**by** (*cases s, cases fst s, simp add: app-def*) (*cases hd (fst s), auto*)

**lemma** *app\_iPop[simp]*:

$app_i (Pop,\ P,\ pc,\ mxs,\ T_r,\ s) = (\exists\ ts\ ST\ LT.\ s = (ts\#\!ST,LT))$   
**by** (*rule length-cases2, auto*)

**lemma** *appIAdd[simp]*:

$app_i (IAdd,\ P,\ pc,\ mxs,\ T_r,\ s) = (\exists\ ST\ LT.\ s = (Integer\#\!Integer\#\!ST,LT))$

**lemma** *appIfFalse [simp]*:

$app_i (IfFalse\ b,\ P,\ pc,\ mxs,\ T_r,\ s) =$   
 $(\exists\ ST\ LT.\ s = (Boolean\#\!ST,LT) \wedge 0 \leq int\ pc + b)$

**lemma** *appCmpEq[simp]*:

$app_i (CmpEq,\ P,\ pc,\ mxs,\ T_r,\ s) =$   
 $(\exists\ T_1\ T_2\ ST\ LT.\ s = (T_1\#\!T_2\#\!ST,LT) \wedge (\neg is-refT\ T_1 \wedge T_2 = T_1 \vee is-refT\ T_1 \wedge is-refT\ T_2))$   
**by** (*rule length-cases4, auto*)

**lemma** *appReturn[simp]*:

$app_i (Return,\ P,\ pc,\ mxs,\ T_r,\ s) = (\exists\ T\ ST\ LT.\ s = (T\#\!ST,LT) \wedge P \vdash T \leq T_r)$   
**by** (*rule length-cases2, auto*)

**lemma** *appThrow[simp]*:

$app_i (Throw,\ P,\ pc,\ mxs,\ T_r,\ s) = (\exists\ T\ ST\ LT.\ s=(T\#\!ST,LT) \wedge is-refT\ T)$   
**by** (*rule length-cases2, auto*)

**lemma** *effNone*:

$(pc', s') \in set\ (eff\ i\ P\ pc\ et\ None) \implies s' = None$   
**by** (*auto simp add: eff-def xcpt-eff-def norm-eff-def*)

some helpers to make the specification directly executable:

**lemma** *relevant-entries-append* [simp]:

*relevant-entries*  $P$   $i$   $pc$   $(xt @ xt')$  = *relevant-entries*  $P$   $i$   $pc$   $xt$  @ *relevant-entries*  $P$   $i$   $pc$   $xt'$   
**by** (*unfold relevant-entries-def*) *simp*

**lemma** *xcpt-app-append* [iff]:

*xcpt-app*  $i$   $P$   $pc$   $maxs$   $(xt@xt')$   $\tau$  = (*xcpt-app*  $i$   $P$   $pc$   $maxs$   $xt$   $\tau$   $\wedge$  *xcpt-app*  $i$   $P$   $pc$   $maxs$   $xt'$   $\tau$ )  
**by** (*unfold xcpt-app-def*) *fastforce*

**lemma** *xcpt-eff-append* [simp]:

*xcpt-eff*  $i$   $P$   $pc$   $\tau$   $(xt@xt')$  = *xcpt-eff*  $i$   $P$   $pc$   $\tau$   $xt$  @ *xcpt-eff*  $i$   $P$   $pc$   $\tau$   $xt'$   
**by** (*unfold xcpt-eff-def, cases*  $\tau$ ) *simp*

**lemma** *app-append* [simp]:

*app*  $i$   $P$   $pc$   $T$   $maxs$   $mpc$   $(xt@xt')$   $\tau$  = (*app*  $i$   $P$   $pc$   $T$   $maxs$   $mpc$   $xt$   $\tau$   $\wedge$  *app*  $i$   $P$   $pc$   $T$   $maxs$   $mpc$   $xt'$   $\tau$ )  
**by** (*unfold app-def eff-def*) *auto*

**end**

## 4.17 Monotonicity of eff and app

**theory** *EffectMono* **imports** *Effect* **begin**

**declare** *not-Err-eq* [iff]

**lemma** *app<sub>i</sub>-mono*:

**assumes** *wf*: *wf-prog*  $p$   $P$   
**assumes** *less*:  $P \vdash \tau \leq_i \tau'$   
**shows** *app<sub>i</sub>*  $(i, P, maxs, mpc, rT, \tau')$   $\implies$  *app<sub>i</sub>*  $(i, P, maxs, mpc, rT, \tau)$

**lemma** *succs-mono*:

**assumes** *wf*: *wf-prog*  $p$   $P$  **and** *app<sub>i</sub>*: *app<sub>i</sub>*  $(i, P, maxs, mpc, rT, \tau')$   
**shows**  $P \vdash \tau \leq_i \tau' \implies set (succs\ i\ \tau\ pc) \subseteq set (succs\ i\ \tau'\ pc)$

**lemma** *app-mono*:

**assumes** *wf*: *wf-prog*  $p$   $P$   
**assumes** *less'*:  $P \vdash \tau \leq' \tau'$   
**shows** *app*  $i$   $P$   $m$   $rT$   $pc$   $mpc$   $xt$   $\tau' \implies$  *app*  $i$   $P$   $m$   $rT$   $pc$   $mpc$   $xt$   $\tau$

**lemma** *eff<sub>i</sub>-mono*:

**assumes** *wf*: *wf-prog*  $p$   $P$   
**assumes** *less*:  $P \vdash \tau \leq_i \tau'$   
**assumes** *app<sub>i</sub>*: *app*  $i$   $P$   $m$   $rT$   $pc$   $mpc$   $xt$  (*Some*  $\tau'$ )  
**assumes** *succs*: *succs*  $i$   $\tau$   $pc \neq []$  *succs*  $i$   $\tau'$   $pc \neq []$   
**shows**  $P \vdash eff_i (i, P, \tau) \leq_i eff_i (i, P, \tau')$

**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**

$wt-instr :: ['m prog, ty, nat, pc, ex-table, instr, pc, ty_m] \Rightarrow bool$

$(\tau, -, -, -, - \vdash -, - :: - [60, 0, 0, 0, 0, 0, 0, 61] 60)$

**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**  $wt-jvm-progD$ :

$wf-jvm-prog_{\Phi} P \implies \exists wt. wf-prog wt P$

**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**  $P, T, mxs, size ins, xt \vdash ins!pc, pc :: \Phi C M$

**lemma**  $wt-jvm-prog-impl-wt-start$ :

**assumes**  $wf: wf\text{-jvm-prog}_{\Phi} P$  **and**  
 $sees: P \vdash C \text{ sees } M:Ts \rightarrow T = (m\text{xs}, m\text{x}l_0, i\text{ns}, x\text{t}) \text{ in } C$   
**shows**  $0 < \text{size } i\text{ns} \wedge \text{wt-start } P C Ts m\text{x}l_0 (\Phi C M)$

## 4.19 The Typing Framework for the JVM

**theory** *TF-JVM*  
**imports** *../DFA/Typing-Framework-err EffectMono BVSpec*  
**begin**

**definition**  $exec :: \text{jvm-prog} \Rightarrow \text{nat} \Rightarrow \text{ty} \Rightarrow \text{ex-table} \Rightarrow \text{instr list} \Rightarrow \text{ty}_i'$  *err step-type*  
**where**

$exec G m\text{xs } rT \text{ et } bs \equiv$   
 $err\text{-step } (\text{size } bs) (\lambda pc. \text{app } (bs!pc) G m\text{xs } rT pc (\text{size } bs) \text{ et})$   
 $(\lambda pc. \text{eff } (bs!pc) G pc \text{ et})$

**locale** *JVM-sl* =

**fixes**  $P :: \text{jvm-prog}$  **and**  $m\text{xs}$  **and**  $m\text{x}l_0$  **and**  $n$   
**fixes**  $Ts :: \text{ty list}$  **and**  $i\text{s}$  **and**  $x\text{t}$  **and**  $T_r$

**fixes**  $m\text{x}l$  **and**  $A$  **and**  $r$  **and**  $f$  **and**  $\text{app}$  **and**  $\text{eff}$  **and**  $\text{step}$

**defines**  $[simp]: m\text{x}l \equiv 1 + \text{size } Ts + m\text{x}l_0$   
**defines**  $[simp]: A \equiv \text{states } P m\text{xs } m\text{x}l$   
**defines**  $[simp]: r \equiv \text{JVM-SemiType.le } P m\text{xs } m\text{x}l$   
**defines**  $[simp]: f \equiv \text{JVM-SemiType.sup } P m\text{xs } m\text{x}l$

**defines**  $[simp]: \text{app} \equiv \lambda pc. \text{Effect.app } (i\text{s}!pc) P m\text{xs } T_r pc (\text{size } i\text{s}) x\text{t}$   
**defines**  $[simp]: \text{eff} \equiv \lambda pc. \text{Effect.eff } (i\text{s}!pc) P pc x\text{t}$   
**defines**  $[simp]: \text{step} \equiv \text{err-step } (\text{size } i\text{s}) \text{app } \text{eff}$

**defines**  $[simp]: n \equiv \text{size } i\text{s}$

**locale** *start-context* = *JVM-sl* +

**fixes**  $p$  **and**  $C$   
**assumes**  $wf: wf\text{-prog } p P$   
**assumes**  $C: is\text{-class } P C$   
**assumes**  $Ts: \text{set } Ts \subseteq \text{types } P$

**fixes**  $first :: \text{ty}_i'$  **and**  $\text{start}$

**defines**  $[simp]:$   
 $first \equiv \text{Some } ([], \text{OK } (\text{Class } C) \# \text{map } \text{OK } Ts @ \text{replicate } m\text{x}l_0 \text{Err})$   
**defines**  $[simp]:$   
 $\text{start} \equiv \text{OK } first \# \text{replicate } (\text{size } i\text{s} - 1) (\text{OK } \text{None})$

### 4.19.1 Connecting JVM and Framework

**lemma** (**in** *start-context*) *semi: semilat* ( $A, r, f$ )  
**apply** (*insert semilat-JVM[OF wf]*)  
**apply** (*unfold A-def r-def f-def JVM-SemiType.le-def JVM-SemiType.sup-def states-def*)  
**apply** *auto*  
**done**

**lemma** (**in** *JVM-sl*) *step-def-exec: step*  $\equiv \text{exec } P m\text{xs } T_r x\text{t } i\text{s}$   
**by** (*simp add: exec-def*)

**lemma** *special-ex-swap-lemma* [iff]:

$(? X. (? n. X = A n \ \& \ P n) \ \& \ Q X) = (? n. Q(A n) \ \& \ P n)$

by *blast*

**lemma** *ex-in-nlists* [iff]:

$(\exists n. ST \in nlists \ n \ A \ \wedge \ n \leq \ mxs) = (set \ ST \subseteq \ A \ \wedge \ size \ ST \leq \ mxs)$

by (*unfold nlists-def*) *auto*

**lemma** *singleton-nlists*:

$(\exists n. [Class \ C] \in nlists \ n \ (types \ P) \ \wedge \ n \leq \ mxs) = (is-class \ P \ C \ \wedge \ 0 < \ mxs)$

by *auto*

**lemma** *set-drop-subset*:

$set \ xs \subseteq \ A \implies set \ (drop \ n \ xs) \subseteq \ A$

by (*auto dest: in-set-dropD*)

**lemma** *Suc-minus-minus-le*:

$n < \ mxs \implies Suc \ (n - (n - b)) \leq \ mxs$

by *arith*

**lemma** *in-nlistsE*:

$\llbracket xs \in nlists \ n \ A; \llbracket size \ xs = n; \ set \ xs \subseteq \ A \rrbracket \implies P \rrbracket \implies P$

by (*unfold nlists-def*) *blast*

**declare** *is-relevant-entry-def* [simp]

**declare** *set-drop-subset* [simp]

**theorem** (in *start-context*) *exec-pres-type*:

*pres-type step (size is) A*

**declare** *is-relevant-entry-def* [simp del]

**declare** *set-drop-subset* [simp del]

**lemma** *lesubstep-type-simple*:

$xs \sqsubseteq_{Product.le \ (=) \ r} \ ys \implies set \ xs \ \{\sqsubseteq_r\} \ set \ ys$

**declare** *is-relevant-entry-def* [simp del]

**lemma** *conjI2*:  $\llbracket A; A \implies B \rrbracket \implies A \ \wedge \ B$  by *blast*

**lemma** (in *JVM-sl*) *eff-mono*:

$\llbracket wf-prog \ p \ P; \ pc < \ length \ is; \ s \sqsubseteq_{sup-state-opt} \ P \ t; \ app \ pc \ t \rrbracket$

$\implies set \ (eff \ pc \ s) \ \{\sqsubseteq_{sup-state-opt} \ P\} \ set \ (eff \ pc \ t)$

**lemma** (in *JVM-sl*) *bounded-step*: *bounded step (size is)*

**theorem** (in *JVM-sl*) *step-mono*:

$wf-prog \ wf-mb \ P \implies mono \ r \ step \ (size \ is) \ A$

**lemma** (in *start-context*) *first-in-A* [iff]: *OK first*  $\in \ A$

using *Ts C* by (*force intro!*: *nlists-appendI simp add: JVM-states-unfold*)

**lemma** (in *JVM-sl*) *wt-method-def2*:

*wt-method P C' Ts T<sub>r</sub> mxs mxl<sub>0</sub> is xt τs =*

*(is*  $\neq \llbracket \wedge$

```

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  $\tau s$ )

```

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<sub>i</sub>' err list  $\Rightarrow$  ty<sub>i</sub>' err list

**where**

```

kiljvm P mxs mxl Tr is xt  $\equiv$ 
kildall (JVM-SemiType.le P mxs mxl) (JVM-SemiType.sup P mxs mxl)
(exec P mxs Tr 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 Tr mxs mxl0 is xt  $\equiv$ 
0 < size is  $\wedge$ 
(let first = Some ([],[OK (Class C')](map OK Ts)(replicate mxl0 Err));
start = OK first#(replicate (size is - 1) (OK None));
result = kiljvm P mxs (1+size Ts+mxl0) Tr is xt start
in  $\forall n < size\ is. result!n \neq Err$ )

```

**definition** *wf-jvm-prog<sub>k</sub>* :: jvm-prog  $\Rightarrow$  bool

**where**

```

wf-jvm-progk 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)
apply (unfold lesssub-def )

```

```

apply auto
apply (insert sup-state-opt-err)
apply (unfold lesssub-def lesub-def sup-state-opt-def sup-state-def sup-ty-opt-def)
apply (simp only: JVM-le-unfold)
apply fastforce
done

lemma acc-le-listI3 [intro!]:
  acc r  $\implies$  acc (Listn.le r)
apply (unfold acc-def)
apply (subgoal-tac)
  wf (UN n. {(ys,xs). size xs = n  $\wedge$  size ys = n  $\wedge$  xs <-(Listn.le r) ys}))
  apply (erule wf-subset)
apply (blast intro: lesssub-lengthD)
apply (rule wf-UN)
prefer 2
apply (rename-tac m n)
apply (case-tac m=n)
apply simp
apply (fast intro!: equalsOI dest: not-sym)
apply (rename-tac n)
apply (induct-tac n)
apply (simp add: lesssub-def cong: conj-cong)
apply (rename-tac k)
apply (simp add: wf-eq-minimal del: r-def f-def step-def A-def)
apply (simp (no-asm) add: length-Suc-conv cong: conj-cong del: r-def f-def step-def A-def)
apply clarify
apply (rename-tac M m)
apply (case-tac  $\exists x xs. size xs = k \wedge x \# xs \in M$ )
prefer 2
apply (erule thin-rl)
apply (erule thin-rl)
apply blast
apply (erule-tac  $x = \{a. \exists xs. size xs = k \wedge a \# xs : M\}$  in alle)
apply (erule impE)
apply blast
apply (thin-tac  $\exists x xs. P x xs$  for P)
apply clarify
apply (rename-tac maxA xs)
apply (erule-tac  $x = \{ys. size ys = size xs \wedge maxA \# ys \in M\}$  in alle)
apply (erule impE)
apply blast
apply clarify
apply (thin-tac  $m \in M$ )
apply (thin-tac  $maxA \# xs \in M$ )
apply (rule bexI)
prefer 2
apply assumption
apply clarify
apply (simp del: r-def f-def step-def A-def)
apply blast
done

```

**lemma** *wf-jvm*:  $wf \{(ss', ss). ss \sqsubseteq_r ss'\}$   
**apply** (*insert acc-le-listI3 acc-JVM [OF wf]*)  
**by** (*simp add: acc-def r-def*)

**lemma** *iter-properties-bv[rule-format]*:  
**shows**  $\llbracket \forall p \in w0. p < n; ss0 \in nlists\ n\ A; \forall p < n. p \notin w0 \longrightarrow stable\ r\ step\ ss0\ p \rrbracket \implies$   
 $iter\ f\ step\ ss0\ w0 = (ss', w') \longrightarrow$   
 $ss' \in nlists\ n\ A \wedge stables\ r\ step\ ss' \wedge ss0 \sqsubseteq_r ss' \wedge$   
 $(\forall ts \in nlists\ n\ A. ss0 \sqsubseteq_r ts \wedge stables\ r\ step\ ts \longrightarrow ss' \sqsubseteq_r ts)$

**lemma** *kildall-properties-bv*:  
**shows**  $\llbracket ss0 \in nlists\ n\ A \rrbracket \implies$   
 $kildall\ r\ f\ step\ ss0 \in nlists\ n\ A \wedge$   
 $stables\ r\ step\ (kildall\ r\ f\ step\ ss0) \wedge$   
 $ss0 \sqsubseteq_r kildall\ r\ f\ step\ ss0 \wedge$   
 $(\forall ts \in nlists\ n\ A. ss0 \sqsubseteq_r ts \wedge stables\ r\ step\ ts \longrightarrow$   
 $kildall\ r\ f\ step\ ss0 \sqsubseteq_r ts)$

## 4.22 LBV for the JVM

**theory** *LBVJVM*  
**imports** *../DFA/Abstract-BV TF-JVM*  
**begin**

**type-synonym** *prog-cert* = *cname*  $\Rightarrow$  *mname*  $\Rightarrow$  *ty<sub>i</sub>' err list*

**definition** *check-cert* :: *jvm-prog*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *ty<sub>i</sub>' 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<sub>i</sub>' err list*  $\Rightarrow$  *instr list*  $\Rightarrow$  *ty<sub>i</sub>' err*  $\Rightarrow$  *ty<sub>i</sub>' 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<sub>i</sub>' 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**  $mk\text{-cert} :: jvm\text{-prog} \Rightarrow nat \Rightarrow ty \Rightarrow ex\text{-table} \Rightarrow instr\ list$   
 $\Rightarrow ty_m \Rightarrow ty_i' err\ list$

**where**

$mk\text{-cert } P\ mxs\ T_r\ et\ bs\ phi \equiv make\text{-cert } (exec\ P\ mxs\ T_r\ et\ bs)\ (map\ OK\ phi)\ (OK\ None)$

**definition**  $prg\text{-cert} :: jvm\text{-prog} \Rightarrow ty_P \Rightarrow prog\text{-cert}$

**where**

$prg\text{-cert } P\ phi\ C\ mn \equiv let\ (C, Ts, T_r, (mxs, mxl_0, ins, et)) = method\ P\ C\ mn$   
 $in\ mk\text{-cert } P\ mxs\ T_r\ et\ ins\ (phi\ C\ mn)$

**lemma**  $check\text{-certD}$  [intro?]:

$check\text{-cert } P\ mxs\ mxl\ n\ cert \implies cert\text{-ok } cert\ n\ Err\ (OK\ None)\ (states\ P\ mxs\ mxl)$

**by**  $(unfold\ cert\text{-ok}\text{-def } check\text{-cert}\text{-def } check\text{-types}\text{-def})\ auto$

**lemma** (in  $start\text{-context}$ )  $wt\text{-lbv}\text{-wt}\text{-step}$ :

**assumes**  $lbv: wt\text{-lbv } P\ C\ Ts\ T_r\ mxs\ mxl_0\ xt\ cert\ is$

**shows**  $\exists \tau s \in nlists\ (size\ is)\ A.\ wt\text{-step } r\ Err\ step\ \tau s \wedge OK\ first \sqsubseteq_r \tau s!0$

**lemma** (in  $start\text{-context}$ )  $wt\text{-lbv}\text{-wt}\text{-method}$ :

**assumes**  $lbv: wt\text{-lbv } P\ C\ Ts\ T_r\ mxs\ mxl_0\ xt\ cert\ is$

**shows**  $\exists \tau s.\ wt\text{-method } P\ C\ Ts\ T_r\ mxs\ mxl_0\ is\ xt\ \tau s$

**lemma** (in  $start\text{-context}$ )  $wt\text{-method}\text{-wt}\text{-lbv}$ :

**assumes**  $wt: wt\text{-method } P\ C\ Ts\ T_r\ mxs\ mxl_0\ is\ xt\ \tau s$

**defines**  $[simp]: cert \equiv mk\text{-cert } P\ mxs\ T_r\ xt\ is\ \tau s$

**shows**  $wt\text{-lbv } P\ C\ Ts\ T_r\ mxs\ mxl_0\ xt\ cert\ is$

**theorem**  $jvm\text{-lbv}\text{-correct}$ :

$wt\text{-jvm}\text{-prog}\text{-lbv } P\ Cert \implies wf\text{-jvm}\text{-prog } P$

**theorem**  $jvm\text{-lbv}\text{-complete}$ :

**assumes**  $wt: wf\text{-jvm}\text{-prog}_{\Phi} P$

**shows**  $wt\text{-jvm}\text{-prog}\text{-lbv } P\ (prg\text{-cert } P\ \Phi)$

**end**

## 4.23 BV Type Safety Invariant

**theory**  $BVConform$

**imports**  $BVSpec\ ../JVM/JVMEExec\ ../Common/Conform$

**begin**

**definition**  $confT :: 'c\ prog \Rightarrow heap \Rightarrow val \Rightarrow ty\ err \Rightarrow bool$

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

**where**

$P, h \vdash v : \leq_T E \equiv case\ E\ of\ Err \Rightarrow True \mid OK\ T \Rightarrow P, h \vdash v : \leq T$

**notation** ( $ASCII$ )

$confT\ (-, - \mid - : \leq T - [51, 51, 51, 51] 50)$

**abbreviation**

$confTs :: 'c\ prog \Rightarrow heap \Rightarrow val\ list \Rightarrow ty_l \Rightarrow bool$

$(-, - \vdash - [:\leq_{\top}] - [51,51,51,51] 50)$  **where**  
 $P, h \vdash vs [:\leq_{\top}] Ts \equiv list-all2 (confT P h) vs Ts$

**notation** (*ASCII*)

$confTs (-, - | - - [:\leq T] - [51,51,51,51] 50)$

**definition**  $conf-f :: jvm-prog \Rightarrow heap \Rightarrow ty_i \Rightarrow bytecode \Rightarrow frame \Rightarrow bool$   
**where**

$conf-f P h \equiv \lambda(ST, LT) \text{ is } (stk, loc, C, M, pc).$   
 $P, h \vdash stk [:\leq] ST \wedge P, h \vdash loc [:\leq_{\top}] LT \wedge pc < size \text{ is}$

**lemma**  $conf-f-def2:$

$conf-f P h (ST, LT) \text{ is } (stk, loc, C, M, pc) \equiv$   
 $P, h \vdash stk [:\leq] ST \wedge P, h \vdash loc [:\leq_{\top}] LT \wedge pc < size \text{ is}$   
**by** (*simp add: conf-f-def*)

**primrec**  $conf-fs :: [jvm-prog, heap, ty_P, mname, nat, ty, frame list] \Rightarrow bool$

**where**

$conf-fs P h \Phi M_0 n_0 T_0 [] = True$   
 $| conf-fs P h \Phi M_0 n_0 T_0 (f\#frs) =$   
 $(let (stk, loc, C, M, pc) = f \text{ in}$   
 $(\exists ST LT Ts T mxs mxl_0 \text{ is } xt.$   
 $\Phi C M ! pc = Some (ST, LT) \wedge$   
 $(P \vdash C \text{ sees } M:Ts \rightarrow T = (mxs, mxl_0, is, xt) \text{ in } C) \wedge$   
 $(\exists D Ts' T' m D'.$   
 $is!pc = (Invoke M_0 n_0) \wedge ST!n_0 = Class D \wedge$   
 $P \vdash D \text{ sees } M_0:Ts' \rightarrow T' = m \text{ in } D' \wedge P \vdash T_0 \leq T') \wedge$   
 $conf-f P h (ST, LT) \text{ is } f \wedge conf-fs P h \Phi M (size Ts) T frs))$

**definition**  $correct-state :: [jvm-prog, ty_P, jvm-state] \Rightarrow bool (-, - \vdash - \surd [61,0,0] 61)$

**where**

$correct-state P \Phi \equiv \lambda(xp, h, frs).$   
 $case xp \text{ of}$   
 $None \Rightarrow (case frs \text{ of}$   
 $[] \Rightarrow True$   
 $| (f\#fs) \Rightarrow P \vdash h \surd \wedge$   
 $(let (stk, loc, C, M, pc) = f$   
 $\text{ in } \exists Ts T mxs mxl_0 \text{ is } xt \tau.$   
 $(P \vdash C \text{ sees } M:Ts \rightarrow T = (mxs, mxl_0, is, xt) \text{ in } C) \wedge$   
 $\Phi C M ! pc = Some \tau \wedge$   
 $conf-f P h \tau \text{ is } f \wedge conf-fs P h \Phi M (size Ts) T fs))$   
 $| Some x \Rightarrow frs = []$

**notation**

$correct-state (-, - | - - [ok] [61,0,0] 61)$

### 4.23.1 Values and $\top$

**lemma**  $confT-Err$  [*iff*]:  $P, h \vdash x : \leq_{\top} Err$   
**by** (*simp add: confT-def*)

**lemma**  $confT-OK$  [*iff*]:  $P, h \vdash x : \leq_{\top} OK T = (P, h \vdash x : \leq T)$

by (*simp add: confT-def*)

**lemma** *confT-cases*:

$P, h \vdash x : \leq_{\top} X = (X = \text{Err} \vee (\exists T. X = \text{OK } T \wedge P, h \vdash x : \leq T))$   
 by (*cases X*) *auto*

**lemma** *confT-heat* [*intro?*, *trans*]:

$\llbracket P, h \vdash x : \leq_{\top} T; h \trianglelefteq h' \rrbracket \Longrightarrow P, h' \vdash x : \leq_{\top} T$   
 by (*cases T*) (*blast intro: conf-heat*)<sup>+</sup>

**lemma** *confT-widen* [*intro?*, *trans*]:

$\llbracket P, h \vdash x : \leq_{\top} T; P \vdash T \leq_{\top} T' \rrbracket \Longrightarrow P, h \vdash x : \leq_{\top} T'$   
 by (*cases T'*, *auto intro: conf-widen*)

### 4.23.2 Stack and Registers

**lemmas** *confTs-Cons1* [*iff*] = *list-all2-Cons1* [*of confT P h*] **for**  $P h$

**lemma** *confTs-confT-sup*:

$\llbracket P, h \vdash \text{loc } [: \leq_{\top}] LT; n < \text{size } LT; LT!n = \text{OK } T; P \vdash T \leq T' \rrbracket$   
 $\Longrightarrow P, h \vdash (\text{loc}!n) : \leq T'$

**lemma** *confTs-heat* [*intro?*]:

$P, h \vdash \text{loc } [: \leq_{\top}] LT \Longrightarrow h \trianglelefteq h' \Longrightarrow P, h' \vdash \text{loc } [: \leq_{\top}] LT$   
 by (*fast elim: list-all2-mono confT-heat*)

**lemma** *confTs-widen* [*intro?*, *trans*]:

$P, h \vdash \text{loc } [: \leq_{\top}] LT \Longrightarrow P \vdash LT \llbracket \leq_{\top} \rrbracket LT' \Longrightarrow P, h \vdash \text{loc } [: \leq_{\top}] LT'$   
 by (*rule list-all2-trans*, *rule confT-widen*)

**lemma** *confTs-map* [*iff*]:

$\bigwedge \text{vs}. (P, h \vdash \text{vs } [: \leq_{\top}] \text{map } \text{OK } Ts) = (P, h \vdash \text{vs } [: \leq] Ts)$   
 by (*induct Ts*) (*auto simp add: list-all2-Cons2*)

**lemma** *reg-widen-Err* [*iff*]:

$\bigwedge LT. (P \vdash \text{replicate } n \text{ Err } \llbracket \leq_{\top} \rrbracket LT) = (LT = \text{replicate } n \text{ Err})$   
 by (*induct n*) (*auto simp add: list-all2-Cons1*)

**lemma** *confTs-Err* [*iff*]:

$P, h \vdash \text{replicate } n v \llbracket \leq_{\top} \rrbracket \text{replicate } n \text{ Err}$   
 by (*induct n*) *auto*

### 4.23.3 correct-frames

**lemmas** [*simp del*] = *fun-upd-apply*

**lemma** *conf-fs-heat*:

$\bigwedge M n T_r.$   
 $\llbracket \text{conf-fs } P h \Phi M n T_r \text{ frs}; h \trianglelefteq h' \rrbracket \Longrightarrow \text{conf-fs } P h' \Phi M n T_r \text{ frs}$   
 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*:

$$\begin{aligned} & \text{match-ex-table } P \ C \ pc \ xt = \text{Some } (pc', d') \implies \\ & \exists (f, t, D, h, d) \in \text{set } (\text{relevant-entries } P \ (\text{Invoke } n \ M) \ pc \ xt). \\ & P \vdash C \preceq^* D \wedge pc \in \{f..<t\} \wedge pc' = h \wedge d' = d \\ & \text{by } (\text{induct } xt) \ (\text{auto simp add: relevant-entries-def matches-ex-entry-def} \\ & \quad \text{is-relevant-entry-def split: if-split-asm}) \end{aligned}$$

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*:

$$\begin{aligned} & \text{assumes } wt: \text{wf-jvm-prog}_{\Phi} \ P \\ & \text{assumes } h: h \ xcp = \text{Some } obj \\ & \text{shows } \bigwedge f. P, \Phi \vdash (\text{None}, h, f \# \text{frs}) \checkmark \implies P, \Phi \vdash (\text{find-handler } P \ xcp \ h \ \text{frs}) \checkmark \\ & (\text{is } \bigwedge f. ?\text{correct } (\text{None}, h, f \# \text{frs}) \implies ?\text{correct } (?find \ \text{frs})) \end{aligned}$$

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*:

$$\begin{aligned} & \llbracket \text{fst } (\text{exec-instr } (\text{ins!pc}) \ P \ h \ \text{stk} \ \text{vars} \ Cl \ M \ pc \ \text{frs}) = \text{Some } xcp; \\ & \quad P, T, mxs, size \ \text{ins}, xt \vdash \text{ins!pc}, pc :: \Phi \ C \ M; \\ & \quad P, \Phi \vdash (\text{None}, h, (\text{stk}, loc, C, M, pc) \# \text{frs}) \checkmark \rrbracket \\ & \implies \exists obj. h \ xcp = \text{Some } obj \\ & (\text{is } \llbracket ?\text{xcpt}; ?wt; ?\text{correct} \rrbracket \implies ?\text{thesis}) \end{aligned}$$

**lemma** *conf-sys-xcpt*:

$$\begin{aligned} & \llbracket \text{preallocated } h; C \in \text{sys-xcpts} \rrbracket \implies P, h \vdash \text{Addr } (\text{addr-of-sys-xcpt } C) : \leq \text{Class } C \\ & \text{by } (\text{auto elim: preallocatedE}) \end{aligned}$$

**lemma** *match-ex-table-SomeD*:

$$\begin{aligned} & \text{match-ex-table } P \ C \ pc \ xt = \text{Some } (pc', d') \implies \\ & \exists (f, t, D, h, d) \in \text{set } xt. \text{matches-ex-entry } P \ C \ pc \ (f, t, D, h, d) \wedge h = pc' \wedge d = d' \\ & \text{by } (\text{induct } xt) \ (\text{auto split: if-split-asm}) \end{aligned}$$

Finally we can state that, whenever an exception occurs, the next state always conforms:

**lemma** *xcpt-correct*:

**fixes**  $\sigma' :: \text{jvm-state}$   
**assumes**  $\text{wtp}: \text{wf-jvm-prog}_{\Phi} P$   
**assumes**  $\text{meth}: P \vdash C \text{ sees } M:Ts \rightarrow T=(\text{mxs}, \text{mxl}_0, \text{ins}, \text{xt}) \text{ in } C$   
**assumes**  $\text{wt}: P, T, \text{mxs}, \text{size } \text{ins}, \text{xt} \vdash \text{ins!pc}, \text{pc} :: \Phi C M$   
**assumes**  $\text{xp}: \text{fst} (\text{exec-instr} (\text{ins!pc}) P h \text{ stk } \text{loc } C M \text{ pc } \text{ frs}) = \text{Some } \text{xcp}$   
**assumes**  $\text{s}': \text{Some } \sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs})$   
**assumes**  $\text{correct}: P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark$   
**shows**  $P, \Phi \vdash \sigma' \checkmark$

### 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**  $\text{defs1} [\text{simp}]$

**lemma** *Invoke-correct*:

**fixes**  $\sigma' :: \text{jvm-state}$   
**assumes**  $\text{wtprog}: \text{wf-jvm-prog}_{\Phi} P$   
**assumes**  $\text{meth-C}: P \vdash C \text{ sees } M:Ts \rightarrow T=(\text{mxs}, \text{mxl}_0, \text{ins}, \text{xt}) \text{ in } C$   
**assumes**  $\text{ins}: \text{ins} ! \text{pc} = \text{Invoke } M' n$   
**assumes**  $\text{wti}: P, T, \text{mxs}, \text{size } \text{ins}, \text{xt} \vdash \text{ins!pc}, \text{pc} :: \Phi C M$   
**assumes**  $\sigma': \text{Some } \sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs})$   
**assumes**  $\text{approx}: P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark$   
**assumes**  $\text{no-xcp}: \text{fst} (\text{exec-instr} (\text{ins!pc}) P h \text{ stk } \text{loc } C M \text{ pc } \text{ frs}) = \text{None}$   
**shows**  $P, \Phi \vdash \sigma' \checkmark$

**declare**  $\text{list-all2-Cons2} [\text{iff}]$

**lemma** *Return-correct*:

**fixes**  $\sigma' :: \text{jvm-state}$   
**assumes**  $\text{wt-prog}: \text{wf-jvm-prog}_{\Phi} P$   
**assumes**  $\text{meth}: P \vdash C \text{ sees } M:Ts \rightarrow T=(\text{mxs}, \text{mxl}_0, \text{ins}, \text{xt}) \text{ in } C$   
**assumes**  $\text{ins}: \text{ins} ! \text{pc} = \text{Return}$   
**assumes**  $\text{wt}: P, T, \text{mxs}, \text{size } \text{ins}, \text{xt} \vdash \text{ins!pc}, \text{pc} :: \Phi C M$   
**assumes**  $\text{s}': \text{Some } \sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs})$   
**assumes**  $\text{correct}: P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark$

**shows**  $P, \Phi \vdash \sigma' \checkmark$

**declare**  $\text{sup-state-opt-any-Some} [\text{iff}]$

**declare**  $\text{not-Err-eq} [\text{iff}]$

**lemma** *Load-correct*:

$\llbracket \text{wf-prog } \text{wt } P;$   
 $P \vdash C \text{ sees } M:Ts \rightarrow T=(\text{mxs}, \text{mxl}_0, \text{ins}, \text{xt}) \text{ in } C;$   
 $\text{ins!pc} = \text{Load } \text{idx};$   
 $P, T, \text{mxs}, \text{size } \text{ins}, \text{xt} \vdash \text{ins!pc}, \text{pc} :: \Phi C M;$   
 $\text{Some } \sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs});$   
 $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark \rrbracket$

$\implies P, \Phi \vdash \sigma' \checkmark$

**by** ( $\text{fastforce } \text{dest}: \text{sees-method-fun} [\text{of } - C] \text{ elim!}: \text{confTs-confT-sup}$ )

**declare**  $\llbracket \text{simproc del: list-to-set-comprehension} \rrbracket$

**lemma** *Store-correct*:

$\llbracket$  *wf-prog wt P*;  
 $P \vdash C \text{ sees } M:Ts \rightarrow T=(m\text{xs},m\text{x}l_0,ins,xt) \text{ in } C$ ;  
 $ins!pc = \text{Store } id\text{x}$ ;  
 $P, T, m\text{xs}, size \text{ ins}, xt \vdash ins!pc, pc :: \Phi \ C \ M$ ;  
 $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc) \# frs)$ ;  
 $P, \Phi \vdash (\text{None}, h, (stk, loc, C, M, pc) \# frs) \checkmark \rrbracket$   
 $\implies P, \Phi \vdash \sigma' \checkmark$

**lemma** *Push-correct*:

$\llbracket$  *wf-prog wt P*;  
 $P \vdash C \text{ sees } M:Ts \rightarrow T=(m\text{xs},m\text{x}l_0,ins,xt) \text{ in } C$ ;  
 $ins!pc = \text{Push } v$ ;  
 $P, T, m\text{xs}, size \text{ ins}, xt \vdash ins!pc, pc :: \Phi \ C \ M$ ;  
 $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc) \# frs)$ ;  
 $P, \Phi \vdash (\text{None}, h, (stk, loc, C, M, pc) \# frs) \checkmark \rrbracket$   
 $\implies P, \Phi \vdash \sigma' \checkmark$

**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* $\rrbracket$   
 $\implies P, h \vdash v : \leq T'$

**lemma** *Checkcast-correct*:

$\llbracket$  *wf-jvm-prog $_{\Phi}$  P*;  
 $P \vdash C \text{ sees } M:Ts \rightarrow T=(m\text{xs},m\text{x}l_0,ins,xt) \text{ in } C$ ;  
 $ins!pc = \text{Checkcast } D$ ;  
 $P, T, m\text{xs}, size \text{ ins}, xt \vdash ins!pc, pc :: \Phi \ C \ M$ ;  
 $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc) \# frs)$  ;  
 $P, \Phi \vdash (\text{None}, h, (stk, loc, C, M, pc) \# frs) \checkmark$ ;  
 $fst (\text{exec-instr } (ins!pc) \ P \ h \ stk \ loc \ C \ M \ pc \ frs) = \text{None}$   $\rrbracket$   
 $\implies P, \Phi \vdash \sigma' \checkmark$

**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*:  $P \vdash C \text{ sees } M:Ts \rightarrow T=(m\text{xs},m\text{x}l_0,ins,xt) \text{ in } C$   
**assumes** *i*:  $ins!pc = \text{Getfield } F \ D$   
**assumes** *wt*:  $P, T, m\text{xs}, size \text{ ins}, xt \vdash ins!pc, pc :: \Phi \ C \ M$   
**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*:  $fst (\text{exec-instr } (ins!pc) \ P \ h \ stk \ loc \ C \ M \ pc \ frs) = \text{None}$

**shows**  $P, \Phi \vdash \sigma' \checkmark$

**lemma** *Putfield-correct*:

**fixes**  $\sigma' :: \text{jvm-state}$   
**assumes** *wf*: *wf-prog wt P*  
**assumes** *mC*:  $P \vdash C \text{ sees } M:Ts \rightarrow T=(m\text{xs},m\text{x}l_0,ins,xt) \text{ in } C$   
**assumes** *i*:  $ins!pc = \text{Putfield } F \ D$   
**assumes** *wt*:  $P, T, m\text{xs}, size \text{ ins}, xt \vdash ins!pc, pc :: \Phi \ C \ M$   
**assumes** *s'*:  $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc) \# frs)$

**assumes** *cf*:  $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark$   
**assumes** *xc*:  $\text{fst} (\text{exec-instr} (\text{ins!pc}) P h \text{stk loc } C M \text{pc frs}) = \text{None}$

**shows**  $P, \Phi \vdash \sigma' \checkmark$

**lemma** *has-fields-b-fields*:

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

**lemma** *oconf-blank* [*intro, simp*]:

$\llbracket \text{is-class } P C; \text{wf-prog wt } P \rrbracket \implies P, h \vdash \text{blank } P C \checkmark$

**lemma** *obj-ty-blank* [*iff*]:  $\text{obj-ty} (\text{blank } P C) = \text{Class } C$

**by** (*simp add: blank-def*)

**lemma** *New-correct*:

**fixes**  $\sigma' :: \text{jvm-state}$

**assumes** *wf*:  $\text{wf-prog wt } P$

**assumes** *meth*:  $P \vdash C \text{ sees } M: Ts \rightarrow T = (\text{mxs}, \text{mxl}_0, \text{ins}, \text{xt}) \text{ in } C$

**assumes** *ins*:  $\text{ins!pc} = \text{New } X$

**assumes** *wt*:  $P, T, \text{mxs}, \text{size } \text{ins}, \text{xt} \vdash \text{ins!pc}, \text{pc} :: \Phi C M$

**assumes** *exec*:  $\text{Some } \sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs})$

**assumes** *conf*:  $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark$

**assumes** *no-x*:  $\text{fst} (\text{exec-instr} (\text{ins!pc}) P h \text{stk loc } C M \text{pc frs}) = \text{None}$

**shows**  $P, \Phi \vdash \sigma' \checkmark$

**lemma** *Goto-correct*:

$\llbracket \text{wf-prog wt } P;$   
 $P \vdash C \text{ sees } M: Ts \rightarrow T = (\text{mxs}, \text{mxl}_0, \text{ins}, \text{xt}) \text{ in } C;$   
 $\text{ins!pc} = \text{Goto branch};$   
 $P, T, \text{mxs}, \text{size } \text{ins}, \text{xt} \vdash \text{ins!pc}, \text{pc} :: \Phi C M;$   
 $\text{Some } \sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs});$   
 $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark \rrbracket$   
 $\implies P, \Phi \vdash \sigma' \checkmark$

**lemma** *IfFalse-correct*:

$\llbracket \text{wf-prog wt } P;$   
 $P \vdash C \text{ sees } M: Ts \rightarrow T = (\text{mxs}, \text{mxl}_0, \text{ins}, \text{xt}) \text{ in } C;$   
 $\text{ins!pc} = \text{IfFalse branch};$   
 $P, T, \text{mxs}, \text{size } \text{ins}, \text{xt} \vdash \text{ins!pc}, \text{pc} :: \Phi C M;$   
 $\text{Some } \sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs});$   
 $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark \rrbracket$   
 $\implies P, \Phi \vdash \sigma' \checkmark$

**lemma** *CmpEq-correct*:

$\llbracket \text{wf-prog wt } P;$   
 $P \vdash C \text{ sees } M: Ts \rightarrow T = (\text{mxs}, \text{mxl}_0, \text{ins}, \text{xt}) \text{ in } C;$   
 $\text{ins!pc} = \text{CmpEq};$   
 $P, T, \text{mxs}, \text{size } \text{ins}, \text{xt} \vdash \text{ins!pc}, \text{pc} :: \Phi C M;$   
 $\text{Some } \sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs});$   
 $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark \rrbracket$   
 $\implies P, \Phi \vdash \sigma' \checkmark$

**lemma** *Pop-correct*:

$\llbracket \text{wf-prog wt } P;$   
 $P \vdash C \text{ sees } M: Ts \rightarrow T = (\text{mxs}, \text{mxl}_0, \text{ins}, \text{xt}) \text{ in } C;$   
 $\text{ins!pc} = \text{Pop};$   
 $P, T, \text{mxs}, \text{size } \text{ins}, \text{xt} \vdash \text{ins!pc}, \text{pc} :: \Phi C M;$

$$\begin{aligned} & \text{Some } \sigma' = \text{exec } (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) ; \\ & P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark \ ] \\ \implies & P, \Phi \vdash \sigma' \checkmark \end{aligned}$$

**lemma** *IAdd-correct*:

$$\begin{aligned} & \llbracket \text{wf-prog wt } P ; \\ & P \vdash C \text{ sees } M : Ts \rightarrow T = (\text{mxs}, \text{mxl}_0, \text{ins}, \text{xt}) \text{ in } C ; \\ & \text{ins} ! \text{pc} = \text{IAdd} ; \\ & P, T, \text{mxs}, \text{size } \text{ins}, \text{xt} \vdash \text{ins} ! \text{pc}, \text{pc} :: \Phi \ C \ M ; \\ & \text{Some } \sigma' = \text{exec } (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) ; \\ & P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark \ ] \\ \implies & P, \Phi \vdash \sigma' \checkmark \end{aligned}$$

**lemma** *Throw-correct*:

$$\begin{aligned} & \llbracket \text{wf-prog wt } P ; \\ & P \vdash C \text{ sees } M : Ts \rightarrow T = (\text{mxs}, \text{mxl}_0, \text{ins}, \text{xt}) \text{ in } C ; \\ & \text{ins} ! \text{pc} = \text{Throw} ; \\ & \text{Some } \sigma' = \text{exec } (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) ; \\ & P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark ; \\ & \text{fst } (\text{exec-instr } (\text{ins} ! \text{pc}) \ P \ h \ \text{stk} \ \text{loc} \ C \ M \ \text{pc} \ \text{frs}) = \text{None} \ ] \\ \implies & P, \Phi \vdash \sigma' \checkmark \\ & \text{by } \text{simp} \end{aligned}$$

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*:

$$\begin{aligned} & \llbracket \text{wf-jvm-prog}_{\Phi} \ P ; \\ & P \vdash C \text{ sees } M : Ts \rightarrow T = (\text{mxs}, \text{mxl}_0, \text{ins}, \text{xt}) \text{ in } C ; \\ & \text{Some } \sigma' = \text{exec } (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) ; \\ & P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark \ ] \\ \implies & P, \Phi \vdash \sigma' \checkmark \end{aligned}$$

#### 4.24.4 Main

**lemma** *correct-state-impl-Some-method*:

$$\begin{aligned} & P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark \\ \implies & \exists m \ Ts \ T. P \vdash C \text{ sees } M : Ts \rightarrow T = m \text{ in } C \\ & \text{by } \text{fastforce} \end{aligned}$$

**lemma** *BV-correct-1* [rule-format]:

$$\bigwedge \sigma. \llbracket \text{wf-jvm-prog}_{\Phi} \ P ; P, \Phi \vdash \sigma \checkmark \ ] \implies P \vdash \sigma \text{ -jvm} \rightarrow_1 \sigma' \longrightarrow P, \Phi \vdash \sigma' \checkmark$$

**theorem** *progress*:

$$\begin{aligned} & \llbracket xp = \text{None} ; \text{frs} \neq [] \ ] \implies \exists \sigma'. P \vdash (xp, h, \text{frs}) \text{ -jvm} \rightarrow_1 \sigma' \\ & \text{by } (\text{clarsimp} \ \text{simp} \ \text{add} : \text{exec-1-iff} \ \text{neq-Nil-conv} \ \text{split-beta} \\ & \quad \text{simp} \ \text{del} : \text{split-paired-Ex}) \end{aligned}$$

**lemma** *progress-conform*:

$$\begin{aligned} & \llbracket \text{wf-jvm-prog}_{\Phi} \ P ; P, \Phi \vdash (xp, h, \text{frs}) \checkmark ; xp = \text{None} ; \text{frs} \neq [] \ ] \\ \implies & \exists \sigma'. P \vdash (xp, h, \text{frs}) \text{ -jvm} \rightarrow_1 \sigma' \wedge P, \Phi \vdash \sigma' \checkmark \end{aligned}$$

**theorem** *BV-correct* [rule-format]:

$$\llbracket \text{wf-jvm-prog}_{\Phi} \ P ; P \vdash \sigma \text{ -jvm} \rightarrow \sigma' \ ] \implies P, \Phi \vdash \sigma \checkmark \longrightarrow P, \Phi \vdash \sigma' \checkmark$$



**lemma** *hconf-start*:

**assumes** *wf*: *wf-prog wf-mb P*

**shows**  $P \vdash (\text{start-heap } P) \checkmark$

**lemma** *BV-correct-initial*:

**shows**  $\llbracket \text{wf-jvm-prog}_{\Phi} P; P \vdash C \text{ sees } M:[] \rightarrow T = m \text{ in } C \rrbracket$

$\implies P, \Phi \vdash \text{start-state } P C M \checkmark$

**theorem** *typesafe*:

**assumes** *welltyped*: *wf-jvm-prog* $_{\Phi}$  *P*

**assumes** *main-method*:  $P \vdash C \text{ sees } M:[] \rightarrow T = m \text{ in } C$

**shows**  $P \vdash \text{start-state } P C M \text{ -jvm} \rightarrow \sigma \implies P, \Phi \vdash \sigma \checkmark$

**end**

## 4.25 Welltyped Programs produce no Type Errors

**theory** *BVNoTypeError*

**imports** *../JVM/JVMDefensive BVSpecTypeSafe*

**begin**

**lemma** *has-methodI*:

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

**by** (*unfold has-method-def*) *blast*

Some simple lemmas about the type testing functions of the defensive JVM:

**lemma** *typeof-NoneD* [*simp, dest*]: *typeof v = Some x*  $\implies \neg \text{is-Addr } v$

**by** (*cases v*) *auto*

**lemma** *is-Ref-def2*:

*is-Ref v = (v = Null*  $\vee$   $(\exists a. v = \text{Addr } a)$ )

**by** (*cases v*) (*auto simp add: is-Ref-def*)

**lemma** [*iff*]: *is-Ref Null* **by** (*simp add: is-Ref-def2*)

**lemma** *is-RefI* [*intro, simp*]:  $P, h \vdash v : \leq T \implies \text{is-refT } T \implies \text{is-Ref } v$

**lemma** *is-IntgI* [*intro, simp*]:  $P, h \vdash v : \leq \text{Integer} \implies \text{is-Intg } v$

**lemma** *is-BoolI* [*intro, simp*]:  $P, h \vdash v : \leq \text{Boolean} \implies \text{is-Bool } v$

**declare** *defs1* [*simp del*]

**lemma** *wt-jvm-prog-states*:

$\llbracket \text{wf-jvm-prog}_{\Phi} P; P \vdash C \text{ sees } M: Ts \rightarrow T = (m_{xs}, m_{xl}, ins, et) \text{ in } C;$

$\Phi C M ! pc = \tau; pc < \text{size } ins \rrbracket$

$\implies \text{OK } \tau \in \text{states } P m_{xs} (1 + \text{size } Ts + m_{xl})$

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** *welltyped*: *wf-jvm-prog* $_{\Phi}$  *P* **and** *conforms*:  $P, \Phi \vdash \sigma \checkmark$

**shows** *exec-d P*  $\sigma \neq \text{TypeError}$

The theorem above tells us that, in welltyped programs, the defensive machine reaches the same result as the aggressive one (after arbitrarily many steps).

**theorem** *welltyped-aggressive-imp-defensive*:

*wf-jvm-prog* $_{\Phi}$  *P*  $\implies P, \Phi \vdash \sigma \checkmark \implies P \vdash \sigma \text{ -jvm} \rightarrow \sigma'$

$$\implies P \vdash (\text{Normal } \sigma) \text{ -jvmd} \rightarrow (\text{Normal } \sigma')$$

As corollary we get that the aggressive and the defensive machine are equivalent for well-typed programs (if started in a conformant state or in the canonical start state)

**corollary** *welltyped-commutes*:

**fixes**  $\sigma :: \text{jvm-state}$   
**assumes**  $wf: wf\text{-jvm-prog}_{\Phi} P$  **and**  $conforms: P, \Phi \vdash \sigma \checkmark$   
**shows**  $P \vdash (\text{Normal } \sigma) \text{ -jvmd} \rightarrow (\text{Normal } \sigma') = P \vdash \sigma \text{ -jvm} \rightarrow \sigma'$   
**apply rule**  
**apply** (*erule defensive-imp-aggressive*)  
**apply** (*erule welltyped-aggressive-imp-defensive [OF wf conforms]*)  
**done**

**corollary** *welltyped-initial-commutes*:

**assumes**  $wf: wf\text{-jvm-prog } P$   
**assumes**  $meth: P \vdash C \text{ sees } M:[] \rightarrow T = b \text{ in } C$   
**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** –

**from**  $wf$  **obtain**  $\Phi$  **where**  $wf': wf\text{-jvm-prog}_{\Phi} P$  **by** (*auto simp: wf-jvm-prog-def*)  
**from** *this meth* **have**  $P, \Phi \vdash \sigma \checkmark$  **unfolding**  $start$  **by** (*rule BV-correct-initial*)  
**with**  $wf'$  **show** *?thesis* **by** (*rule welltyped-commutes*)

**qed**

**lemma** *not-TypeError-eq [iff]*:

$x \neq \text{TypeError} = (\exists t. x = \text{Normal } t)$   
**by** (*cases x auto*)

**locale**  $cnf =$

**fixes**  $P$  **and**  $\Phi$  **and**  $\sigma$   
**assumes**  $wf: wf\text{-jvm-prog}_{\Phi} P$   
**assumes**  $cnf: \text{correct-state } P \Phi \sigma$

**theorem** (**in**  $cnf$ ) *no-type-errors*:

$P \vdash (\text{Normal } \sigma) \text{ -jvmd} \rightarrow \sigma' \implies \sigma' \neq \text{TypeError}$   
**apply** (*unfold exec-all-d-def1*)  
**apply** (*erule rtrancl-induct*)  
**apply** *simp*  
**apply** (*fold exec-all-d-def1*)  
**apply** (*insert cnf wf*)  
**apply** *clarsimp*  
**apply** (*erule defensive-imp-aggressive*)  
**apply** (*erule (2) BV-correct*)  
**apply** (*erule (1) no-type-error*) **back**  
**apply** (*auto simp add: exec-1-d-eq*)  
**done**

**locale**  $start =$

**fixes**  $P$  **and**  $C$  **and**  $M$  **and**  $\sigma$  **and**  $T$  **and**  $b$   
**assumes**  $wf: wf\text{-jvm-prog } P$   
**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 -
  from wf obtain  $\Phi$  where wf-jvm-prog $\Phi$  P by (auto simp: wf-jvm-prog-def)
  moreover
  with sees have correct-state P  $\Phi$  (start-state P C M)
  by - (rule BV-correct-initial)
  ultimately have cnf P  $\Phi$  (start-state P C M) by (rule cnf.intro)
  moreover assume  $P \vdash \sigma \text{ -jvmd} \rightarrow \sigma'$ 
  ultimately show ?thesis by (unfold  $\sigma$ -def) (rule cnf.no-type-errors)
qed

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. 72) 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':
  list-name  $\neq$  test-name
  list-name  $\neq$  Object
  list-name  $\neq$  ClassCast
  list-name  $\neq$  OutOfMemory
  list-name  $\neq$  NullPointer
  test-name  $\neq$  Object
  test-name  $\neq$  OutOfMemory
  test-name  $\neq$  ClassCast
  test-name  $\neq$  NullPointer
  ClassCast  $\neq$  NullPointer
  ClassCast  $\neq$  Object
  NullPointer  $\neq$  Object
  OutOfMemory  $\neq$  ClassCast
  OutOfMemory  $\neq$  NullPointer
  OutOfMemory  $\neq$  Object
  by (simp-all add: list-name-def test-name-def Object-def NullPointer-def
    OutOfMemory-def ClassCast-def)

```

```

lemmas distinct-classes = distinct-classes' distinct-classes' [symmetric]

```

```

lemma distinct-fields:
  val-name  $\neq$  next-name
  next-name  $\neq$  val-name
  by (simp-all add: val-name-def next-name-def)

```

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, [], [])*  
**by** (*simp add: class-def system-defs E-def*)

**lemma** *class-NullPointer* [*simp*]:  
   *class E NullPointer* = *Some (Object, [], [])*  
**by** (*simp add: class-def system-defs E-def distinct-classes*)

**lemma** *class-OutOfMemory* [*simp*]:  
   *class E OutOfMemory* = *Some (Object, [], [])*  
**by** (*simp add: class-def system-defs E-def distinct-classes*)

**lemma** *class-ClassCast* [*simp*]:  
   *class E ClassCast* = *Some (Object, [], [])*  
**by** (*simp add: class-def system-defs E-def distinct-classes*)

**lemma** *class-list* [*simp*]:  
   *class E list-name* = *Some list-class*  
**by** (*simp add: class-def system-defs E-def distinct-classes*)

**lemma** *class-test* [*simp*]:  
   *class E test-name* = *Some test-class*  
**by** (*simp add: class-def system-defs E-def distinct-classes*)

**lemma** *E-classes* [*simp*]:  
    $\{C. \text{is-class } E \ C\} = \{\text{list-name}, \text{test-name}, \text{NullPointer},$   
                            $\text{ClassCast}, \text{OutOfMemory}, \text{Object}\}$   
**by** (*auto simp add: is-class-def class-def system-defs E-def class-defs*)

The subclass relation spelled out:

**lemma** *subcls1*:  
    $\text{subcls1 } E = \{(\text{list-name}, \text{Object}), (\text{test-name}, \text{Object}), (\text{NullPointer}, \text{Object}),$   
                    $(\text{ClassCast}, \text{Object}), (\text{OutOfMemory}, \text{Object})\}$

The subclass relation is acyclic; hence its converse is well founded:

**lemma** *notin-rtrancl*:  
    $(a, b) \in r^* \implies a \neq b \implies (\bigwedge y. (a, y) \notin r) \implies \text{False}$   
**by** (*auto elim: converse-rtranclE*)

**lemma** *acyclic-subcls1-E*: *acyclic (subcls1 E)*

**lemma** *wf-subcls1-E*: *wf ((subcls1 E)<sup>-1</sup>)*

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)])

**lemma** *method-makelist* [simp]:  
*method E test-name makelist-name =*  
*(test-name, [], Void, 3, 2, make-list-ins, [])*

**lemma** *field-val* [simp]:  
*field E list-name val-name = (list-name, Integer)*

**lemma** *field-next* [simp]:  
*field E list-name next-name = (list-name, Class list-name)*

**lemma** [simp]: *fields E Object = []*  
**by** (*fastforce intro: fields-def2 Fields.intros*)

**lemma** [simp]: *fields E NullPointer = []*  
**by** (*fastforce simp add: distinct-classes intro: fields-def2 Fields.intros*)

**lemma** [simp]: *fields E ClassCast = []*  
**by** (*fastforce simp add: distinct-classes intro: fields-def2 Fields.intros*)

**lemma** [simp]: *fields E OutOfMemory = []*  
**by** (*fastforce simp add: distinct-classes intro: fields-def2 Fields.intros*)

**lemma** [simp]: *fields E test-name = []*

**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)*

## 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**

$$\varphi_a \equiv \text{map } (\lambda(x,y). \text{Some } (x, \text{map OK } y)) [$$

- ( $\quad \quad \quad []$ , [Class *list-name*, Class *list-name*]),
- ( $\quad \quad \quad$  [Class *list-name*], [Class *list-name*, Class *list-name*]),
- ( $\quad \quad \quad$  [Class *list-name*], [Class *list-name*, Class *list-name*]),
- ( $\quad$  [Class *list-name*, Class *list-name*], [Class *list-name*, Class *list-name*]),
- ( $\quad$  [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*]),
- ( $\quad$  [Boolean, Class *list-name*], [Class *list-name*, Class *list-name*]),
- ( $\quad \quad \quad$  [Class Object], [Class *list-name*, Class *list-name*]),
- ( $\quad \quad \quad$   $\quad \quad \quad$  [], [Class *list-name*, Class *list-name*]),
- ( $\quad \quad \quad$  [Class *list-name*], [Class *list-name*, Class *list-name*]),
- ( $\quad$  [Class *list-name*, Class *list-name*], [Class *list-name*, Class *list-name*]),
- ( $\quad \quad \quad$   $\quad \quad \quad$  [], [Class *list-name*, Class *list-name*]),
- ( $\quad \quad \quad$  [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

$$pc < n \implies P pc$$

into a series of goals

$$P (0::'a)$$

$$P (Suc 0)$$

...

$$P n$$

**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*:  $x < n \implies (x \in [0, n) \implies P x) \implies P x$

**by** (*simp add: intervall-def*)

**lemma** *pc-next*:  $x \in [n0, n) \implies P n0 \implies (x \in [Suc n0, n) \implies P x) \implies P x$

**lemma** *pc-end*:  $x \in [n, n) \implies P x$

**by** (*unfold intervall-def arith*)

**lemma** *types-append* [*simp*]: *check-types E 3 (Suc (Suc 0)) (map OK  $\varphi_a$ )*

**lemma** *wt-append* [*simp*]:

$$\text{wt-method } E \text{ list-name } [Class \text{ list-name}] \text{ Void } 3 \text{ 0 append-ins} \\ [(Suc 0, 2, \text{NullPointer}, 7, 0)] \varphi_a$$

Some abbreviations for readability

**abbreviation** *Clist* == *Class list-name*

**abbreviation** *Ctest* == *Class test-name*

**definition** *phi-makelist* :: *ty<sub>m</sub>* ( $\varphi_m$ )

**where**

$$\varphi_m \equiv \text{map } (\lambda(x,y). \text{Some } (x, y)) [ \\ ( [], [OK \text{ Ctest}, \text{Err}, \text{Err}] ), \\ ( [Clist], [OK \text{ Ctest}, \text{Err}, \text{Err}] ), \\ ( [], [OK \text{ Clist}, \text{Err}, \text{Err}] ), \\ ( [Clist], [OK \text{ Clist}, \text{Err}, \text{Err}] ), \\ ( [Integer, Clist], [OK \text{ Clist}, \text{Err}, \text{Err}] ), \\ \\ ( [], [OK \text{ Clist}, \text{Err}, \text{Err}] ), \\ ( [Clist], [OK \text{ Clist}, \text{Err}, \text{Err}] ), \\ ( [], [OK \text{ Clist}, OK \text{ Clist}, \text{Err}] ), \\ ( [Clist], [OK \text{ Clist}, OK \text{ Clist}, \text{Err}] ), \\ ( [Integer, Clist], [OK \text{ Clist}, OK \text{ Clist}, \text{Err}] ),$$

```

(
  [], [OK Clist, OK Clist, Err  ],
(
  [Clist], [OK Clist, OK Clist, Err  ]),
(
  [], [OK Clist, OK Clist, OK Clist]),
(
  [Clist], [OK Clist, OK Clist, OK Clist]),
(
  [Integer, Clist], [OK Clist, OK Clist, OK Clist]),

(
  [], [OK Clist, OK Clist, OK Clist]),
(
  [Clist], [OK Clist, OK Clist, OK Clist]),
(
  [Clist, Clist], [OK Clist, OK Clist, OK Clist]),
(
  [Void], [OK Clist, OK Clist, OK Clist]),
(
  [], [OK Clist, OK Clist, OK Clist]),
(
  [Clist], [OK Clist, OK Clist, OK Clist]),
(
  [Clist, Clist], [OK Clist, OK Clist, OK Clist]),
(
  [Void], [OK Clist, OK Clist, OK Clist])

```

**lemma** *types-makelist* [simp]: *check-types E 3 (Suc (Suc (Suc 0))) (map OK  $\varphi_m$ )*

**lemma** *wt-makelist* [simp]:

*wt-method E test-name [] Void 3 2 make-list-ins []  $\varphi_m$*

**lemma** *wf-md'E*:

$\llbracket$  *wf-prog wf-md P*;  
 $\bigwedge C S fs ms m. [(C, S, fs, ms) \in \text{set } P; m \in \text{set } ms] \implies \text{wf-md}' P C m \rrbracket$   
 $\implies \text{wf-prog wf-md}' P$

The whole program is welltyped:

**definition** *Phi* :: *ty<sub>P</sub>* ( $\Phi$ )

**where**

$\Phi C mn \equiv$  *if*  $C = \text{test-name} \wedge mn = \text{makelist-name}$  *then*  $\varphi_m$  *else*  
*if*  $C = \text{list-name} \wedge mn = \text{append-name}$  *then*  $\varphi_a$  *else*  $[]$

**lemma** *wf-prog*:

*wf-jvm-prog <sub>$\Phi$</sub>  E*

#### 4.26.4 Conformance

Execution of the program will be typesafe, because its start state conforms to the welltyping:

**lemma**  $E, \Phi \vdash \text{start-state } E \text{ test-name makelist-name } \checkmark$

#### 4.26.5 Example for code generation: inferring method types

**definition** *test-kil* :: *jvm-prog*  $\Rightarrow$  *cname*  $\Rightarrow$  *ty list*  $\Rightarrow$  *ty*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat*  $\Rightarrow$   
*ex-table*  $\Rightarrow$  *instr list*  $\Rightarrow$  *ty<sub>i</sub>' err list*

**where**

*test-kil G C pTs rT mxs mxl et instr*  $\equiv$   
*(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]:

*unstables r step ss =*  
*fold ( $\lambda p A. \text{if } \neg \text{stable } r \text{ step } ss \text{ } p \text{ then insert } p \text{ } A \text{ else } A$ ) [0..*size ss*] {}*

**proof** –

**have** *unstables r step ss = (UN p: {..<size ss}. if  $\neg \text{stable } r \text{ step } ss \text{ } p$  then {p} else {})*  
**apply** (*unfold unstables-def*)

```

apply (rule equalityI)
apply (rule subsetI)
apply (erule CollectE)
apply (erule conjE)
apply (rule UN-I)
apply simp
apply simp
apply (rule subsetI)
apply (erule UN-E)
apply (case-tac  $\neg$  stable r step ss p)
apply simp+
done
also have  $\bigwedge f. (UN\ p:\{..<size\ ss\}. f\ p) = Union\ (set\ (map\ f\ [0..<size\ ss]))$  by auto
also note Sup-set-fold also note fold-map
also have  $(\cup) \circ (\lambda p. if\ \neg\ stable\ r\ step\ ss\ p\ then\ \{p\}\ else\ \{\}) =$ 
 $(\lambda p\ A. if\ \neg\ stable\ r\ step\ ss\ p\ then\ insert\ p\ A\ else\ A)$ 
by(auto simp add: fun-eq-iff)
finally show ?thesis .
qed

```

**definition** some-elem :: 'a set  $\Rightarrow$  'a **where** [code del]:  
some-elem = (%S. SOME x. x : S)

**code-printing**

**constant** some-elem  $\rightarrow$  (SML) (case/ - of/ Set/ xs/ =>/ hd/ xs)

This code setup is just a demonstration and *not* sound!

**notepad begin**

```

have some-elem (set [False, True]) = False by eval
moreover have some-elem (set [True, False]) = True by eval
ultimately have False by (simp add: some-elem-def)

```

**end**

**lemma** [code]:

```

iter f step ss w = while ( $\lambda(ss, w). \neg Set.is-empty\ w$ )
  ( $\lambda(ss, w).$ 
    let p = some-elem w in propa f (step p (ss ! p)) ss (w - {p}))
  (ss, w)
unfolding iter-def Set.is-empty-def some-elem-def ..

```

**lemma** JVM-sup-unfold [code]:

```

JVM-SemiType.sup S m n = lift2 (Opt.sup
  (Product.sup (Listn.sup (SemiType.sup S))
    ( $\lambda x\ y. OK\ (map2\ (lift2\ (SemiType.sup\ S))\ x\ y)$ )))
apply (unfold JVM-SemiType.sup-def JVM-SemiType.sl-def Opt.esl-def Err.sl-def
  stk-esl-def loc-sl-def Product.esl-def
  Listn.sl-def upto-esl-def SemiType.esl-def Err.esl-def)
by simp

```

**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  $\Rightarrow$  True | Class C  $\Rightarrow$  True | -  $\Rightarrow$  False)



```

by (simp add: is-refT-def split: ty.split)

declare appi.simps [code]

lemma [code]:
  appi (Getfield F C, P, pc, mxs, Tr, (T1#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))
by(auto simp add: Predicate.holds-eq intro: sees-field-i-i-i-o-iI elim: sees-field-i-i-i-o-iE)

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))
by(auto simp add: Predicate.holds-eq simp del: eval-bind split: if-split-asm elim!: sees-field-i-i-i-o-iE
Predicate.bindE intro: Predicate.bindI sees-field-i-i-i-o-iI)

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)))
by (fastforce simp add: Predicate.holds-eq simp del: eval-bind split: ty.split-asm if-split-asm intro: bindI
Method-i-i-i-o-o-o-oI elim!: bindE Method-i-i-i-o-o-oE)

lemmas [code] =
  SemiType.sup-def [unfolded exec-lub-def]
  widen.equation
  is-relevant-class.simps

definition test1 where
  test1 = test-kil E list-name [Class list-name] Void 3 0
  [(Suc 0, 2, NullPointer, 7, 0)] append-ins
definition test2 where
  test2 = test-kil E test-name [] Void 3 2 [] make-list-ins
definition test3 where test3 = φa
definition test4 where test4 = φm

ML-val <
  if @{code test1} = @{code map} @{code OK} @{code test3} then () else error wrong result;
  if @{code test2} = @{code map} @{code OK} @{code test4} then () else error wrong result
  >

end

```



# Chapter 5

## Compilation

### 5.1 An Intermediate Language

theory *J1* imports *../J/BigStep* begin

**type-synonym** *expr*<sub>1</sub> = *nat exp*  
**type-synonym** *J*<sub>1</sub>-*prog* = *expr*<sub>1</sub> *prog*  
**type-synonym** *state*<sub>1</sub> = *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*<sub>1</sub> «*bop*» *e*<sub>2</sub>) = *max* (*max-vars e*<sub>1</sub>) (*max-vars e*<sub>2</sub>)  
| *max-vars*(*Var V*) = 0  
| *max-vars*(*V:=e*) = *max-vars e*  
| *max-vars*(*e.F{D}*) = *max-vars e*  
| *max-vars*(*FAss e*<sub>1</sub> *F D e*<sub>2</sub>) = *max* (*max-vars e*<sub>1</sub>) (*max-vars e*<sub>2</sub>)  
| *max-vars*(*e.M(es)*) = *max* (*max-vars e*) (*max-varss es*)  
| *max-vars*(*{V:T; e}*) = *max-vars e* + 1  
| *max-vars*(*e*<sub>1</sub>;;*e*<sub>2</sub>) = *max* (*max-vars e*<sub>1</sub>) (*max-vars e*<sub>2</sub>)  
| *max-vars*(*if* (*e*) *e*<sub>1</sub> *else e*<sub>2</sub>) =  
  *max* (*max-vars e*) (*max* (*max-vars e*<sub>1</sub>) (*max-vars e*<sub>2</sub>))  
| *max-vars*(*while* (*b*) *e*) = *max* (*max-vars b*) (*max-vars e*)  
| *max-vars*(*throw e*) = *max-vars e*  
| *max-vars*(*try e*<sub>1</sub> *catch*(*C V*) *e*<sub>2</sub>) = *max* (*max-vars e*<sub>1</sub>) (*max-vars e*<sub>2</sub> + 1)  
  
| *max-varss* [] = 0  
| *max-varss* (*e#es*) = *max* (*max-vars e*) (*max-varss es*)

**inductive**

*eval*<sub>1</sub> :: *J*<sub>1</sub>-*prog* ⇒ *expr*<sub>1</sub> ⇒ *state*<sub>1</sub> ⇒ *expr*<sub>1</sub> ⇒ *state*<sub>1</sub> ⇒ *bool*  
  (- ⊢<sub>1</sub> ((1⟨-,/-⟩) ⇒ / (1⟨-,/-⟩)) [51,0,0,0,0] 81)  
**and** *evals*<sub>1</sub> :: *J*<sub>1</sub>-*prog* ⇒ *expr*<sub>1</sub> *list* ⇒ *state*<sub>1</sub> ⇒ *expr*<sub>1</sub> *list* ⇒ *state*<sub>1</sub> ⇒ *bool*  
  (- ⊢<sub>1</sub> ((1⟨-,/-⟩) [⇒] / (1⟨-,/-⟩)) [51,0,0,0,0] 81)  
**for** *P* :: *J*<sub>1</sub>-*prog*

**where**

*New*<sub>1</sub>:

$\llbracket P \vdash_1 \langle \text{new } C, (h, l) \rangle \Rightarrow \langle \text{addr } a, (h', l) \rangle \rrbracket$   
 $\implies P \vdash_1 \langle \text{new } C, (h, l) \rangle \Rightarrow \langle \text{addr } a, (h', l) \rangle$

| *NewFail*<sub>1</sub>:

$\text{new-Addr } h = \text{None} \implies$   
 $P \vdash_1 \langle \text{new } C, (h, l) \rangle \Rightarrow \langle \text{THROW OutOfMemory}, (h, l) \rangle$

| *Cast*<sub>1</sub>:

$\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, l) \rangle; h a = \text{Some}(D, fs); P \vdash D \preceq^* C \rrbracket$   
 $\implies P \vdash_1 \langle \text{Cast } C e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, l) \rangle$

| *CastNull*<sub>1</sub>:

$P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \implies$   
 $P \vdash_1 \langle \text{Cast } C e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle$

| *CastFail*<sub>1</sub>:

$\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, l) \rangle; h a = \text{Some}(D, fs); \neg P \vdash D \preceq^* C \rrbracket$   
 $\implies P \vdash_1 \langle \text{Cast } C e, s_0 \rangle \Rightarrow \langle \text{THROW ClassCast}, (h, l) \rangle$

| *CastThrow*<sub>1</sub>:

$P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \implies$   
 $P \vdash_1 \langle \text{Cast } C e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$

| *Val*<sub>1</sub>:

$P \vdash_1 \langle \text{Val } v, s \rangle \Rightarrow \langle \text{Val } v, s \rangle$

| *BinOp*<sub>1</sub>:

$\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v_2, s_2 \rangle; \text{binop}(bop, v_1, v_2) = \text{Some } v \rrbracket$   
 $\implies P \vdash_1 \langle e_1 \ll bop \gg e_2, s_0 \rangle \Rightarrow \langle \text{Val } v, s_2 \rangle$

| *BinOpThrow*<sub>11</sub>:

$P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle \implies$   
 $P \vdash_1 \langle e_1 \ll bop \gg e_2, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle$

| *BinOpThrow*<sub>21</sub>:

$\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{throw } e, s_2 \rangle \rrbracket$   
 $\implies P \vdash_1 \langle e_1 \ll bop \gg e_2, s_0 \rangle \Rightarrow \langle \text{throw } e, s_2 \rangle$

| *Var*<sub>1</sub>:

$\llbracket ls!i = v; i < \text{size } ls \rrbracket \implies$   
 $P \vdash_1 \langle \text{Var } i, (h, ls) \rangle \Rightarrow \langle \text{Val } v, (h, ls) \rangle$

| *LAss*<sub>1</sub>:

$\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, (h, ls) \rangle; i < \text{size } ls; ls' = ls[i := v] \rrbracket$   
 $\implies P \vdash_1 \langle i := e, s_0 \rangle \Rightarrow \langle \text{unit}, (h, ls') \rangle$

| *LAssThrow*<sub>1</sub>:

$P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \implies$   
 $P \vdash_1 \langle i := e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$

| *FAcc*<sub>1</sub>:

$\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, ls) \rangle; h a = \text{Some}(C, fs); fs(F, D) = \text{Some } v \rrbracket$   
 $\implies P \vdash_1 \langle e \cdot F\{D\}, s_0 \rangle \Rightarrow \langle \text{Val } v, (h, ls) \rangle$

| *FAccNull*<sub>1</sub>:

$P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \implies$   
 $P \vdash_1 \langle e \cdot F\{D\}, s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_1 \rangle$

| *FAccThrow*<sub>1</sub>:

$P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \implies$   
 $P \vdash_1 \langle e \cdot F\{D\}, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$

| *FAss*<sub>1</sub>:  

$$\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2) \rangle; \quad h_2 a = \text{Some}(C, fs); fs' = fs((F, D) \mapsto v); h_2' = h_2(a \mapsto (C, fs')) \rrbracket$$

$$\Longrightarrow P \vdash_1 \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{unit}, (h_2', l_2) \rangle$$

| *FAssNull*<sub>1</sub>:  

$$\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, s_2 \rangle \rrbracket$$

$$\Longrightarrow P \vdash_1 \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_2 \rangle$$

| *FAssThrow*<sub>11</sub>:  

$$P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow$$

$$P \vdash_1 \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$$

| *FAssThrow*<sub>21</sub>:  

$$\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle \rrbracket$$

$$\Longrightarrow P \vdash_1 \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle$$

| *CallObjThrow*<sub>1</sub>:  

$$P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow$$

$$P \vdash_1 \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$$

| *CallNull*<sub>1</sub>:  

$$\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle; P \vdash_1 \langle es, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, s_2 \rangle \rrbracket$$

$$\Longrightarrow P \vdash_1 \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_2 \rangle$$

| *Call*<sub>1</sub>:  

$$\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash_1 \langle es, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_2, ls_2) \rangle; \quad h_2 a = \text{Some}(C, fs); P \vdash C \text{ sees } M: Ts \rightarrow T = \text{body in } D; \quad \text{size } vs = \text{size } Ts; ls_2' = (\text{Addr } a) \# vs @ \text{replicate } (\text{max-vars body}) \text{ undefined}; \quad P \vdash_1 \langle \text{body}, (h_2, ls_2') \rangle \Rightarrow \langle e', (h_3, ls_3) \rangle \rrbracket$$

$$\Longrightarrow P \vdash_1 \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle e', (h_3, ls_2) \rangle$$

| *CallParamsThrow*<sub>1</sub>:  

$$\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash_1 \langle es, s_1 \rangle [\Rightarrow] \langle es', s_2 \rangle; \quad es' = \text{map Val } vs @ \text{throw } ex \# es_2 \rrbracket$$

$$\Longrightarrow P \vdash_1 \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle \text{throw } ex, s_2 \rangle$$

| *Block*<sub>1</sub>:  

$$P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle e', s_1 \rangle \Longrightarrow P \vdash_1 \langle \text{Block } i \ T \ e, s_0 \rangle \Rightarrow \langle e', s_1 \rangle$$

| *Seq*<sub>1</sub>:  

$$\llbracket P \vdash_1 \langle e_0, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash_1 \langle e_1, s_1 \rangle \Rightarrow \langle e_2, s_2 \rangle \rrbracket$$

$$\Longrightarrow P \vdash_1 \langle e_0;; e_1, s_0 \rangle \Rightarrow \langle e_2, s_2 \rangle$$

| *SeqThrow*<sub>1</sub>:  

$$P \vdash_1 \langle e_0, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle \Longrightarrow$$

$$P \vdash_1 \langle e_0;; e_1, s_0 \rangle \Rightarrow \langle \text{throw } e, s_1 \rangle$$

| *CondT*<sub>1</sub>:  

$$\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle; P \vdash_1 \langle e_1, s_1 \rangle \Rightarrow \langle e', s_2 \rangle \rrbracket$$

$$\Longrightarrow P \vdash_1 \langle \text{if } (e) \ e_1 \ \text{else } e_2, s_0 \rangle \Rightarrow \langle e', s_2 \rangle$$

| *CondF*<sub>1</sub>:  

$$\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{false}, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle e', s_2 \rangle \rrbracket$$

$$\Longrightarrow P \vdash_1 \langle \text{if } (e) \ e_1 \ \text{else } e_2, s_0 \rangle \Rightarrow \langle e', s_2 \rangle$$

| *CondThrow*<sub>1</sub>:  

$$P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow$$

$$P \vdash_1 \langle \text{if } (e) \ e_1 \ \text{else } e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$$

| *WhileF*<sub>1</sub>:  

$$P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{false}, s_1 \rangle \Longrightarrow$$

$P \vdash_1 \langle \text{while } (e) \ c, s_0 \rangle \Rightarrow \langle \text{unit}, s_1 \rangle$   
 | *WhileT<sub>1</sub>*:  
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle; P \vdash_1 \langle c, s_1 \rangle \Rightarrow \langle \text{Val } v_1, s_2 \rangle; \\ P \vdash_1 \langle \text{while } (e) \ c, s_2 \rangle \Rightarrow \langle e_3, s_3 \rangle \rrbracket \\ \Longrightarrow P \vdash_1 \langle \text{while } (e) \ c, s_0 \rangle \Rightarrow \langle e_3, s_3 \rangle$   
 | *WhileCondThrow<sub>1</sub>*:  
 $P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow \\ P \vdash_1 \langle \text{while } (e) \ c, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$   
 | *WhileBodyThrow<sub>1</sub>*:  
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle; P \vdash_1 \langle c, s_1 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle \rrbracket \\ \Longrightarrow P \vdash_1 \langle \text{while } (e) \ c, s_0 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle$   
  
 | *Throw<sub>1</sub>*:  
 $P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle \Longrightarrow \\ P \vdash_1 \langle \text{throw } e, s_0 \rangle \Rightarrow \langle \text{Throw } a, s_1 \rangle$   
 | *ThrowNull<sub>1</sub>*:  
 $P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \Longrightarrow \\ P \vdash_1 \langle \text{throw } e, s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_1 \rangle$   
 | *ThrowThrow<sub>1</sub>*:  
 $P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow \\ P \vdash_1 \langle \text{throw } e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$   
  
 | *Try<sub>1</sub>*:  
 $P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle \Longrightarrow \\ P \vdash_1 \langle \text{try } e_1 \ \text{catch}(C \ i) \ e_2, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle$   
 | *TryCatch<sub>1</sub>*:  
 $\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Throw } a, (h_1, ls_1) \rangle; \\ h_1 \ a = \text{Some}(D, fs); P \vdash D \preceq^* C; i < \text{length } ls_1; \\ P \vdash_1 \langle e_2, (h_1, ls_1[i := \text{Addr } a]) \rangle \Rightarrow \langle e_2', (h_2, ls_2) \rangle \rrbracket \\ \Longrightarrow P \vdash_1 \langle \text{try } e_1 \ \text{catch}(C \ i) \ e_2, s_0 \rangle \Rightarrow \langle e_2', (h_2, ls_2) \rangle$   
 | *TryThrow<sub>1</sub>*:  
 $\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Throw } a, (h_1, ls_1) \rangle; h_1 \ a = \text{Some}(D, fs); \neg P \vdash D \preceq^* C \rrbracket \\ \Longrightarrow P \vdash_1 \langle \text{try } e_1 \ \text{catch}(C \ i) \ e_2, s_0 \rangle \Rightarrow \langle \text{Throw } a, (h_1, ls_1) \rangle$   
  
 | *Nil<sub>1</sub>*:  
 $P \vdash_1 \langle [], s \rangle [\Rightarrow] \langle [], s \rangle$   
  
 | *Cons<sub>1</sub>*:  
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash_1 \langle es, s_1 \rangle [\Rightarrow] \langle es', s_2 \rangle \rrbracket \\ \Longrightarrow P \vdash_1 \langle e \# es, s_0 \rangle [\Rightarrow] \langle \text{Val } v \# es', s_2 \rangle$   
 | *ConsThrow<sub>1</sub>*:  
 $P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow \\ P \vdash_1 \langle e \# es, s_0 \rangle [\Rightarrow] \langle \text{throw } e' \# es, s_1 \rangle$

**lemma** *eval<sub>1</sub>-preserves-len*:

$$P \vdash_1 \langle e_0, (h_0, ls_0) \rangle \Rightarrow \langle e_1, (h_1, ls_1) \rangle \Longrightarrow \text{length } ls_0 = \text{length } ls_1$$

**and** *evals<sub>1</sub>-preserves-len*:

$$P \vdash_1 \langle es_0, (h_0, ls_0) \rangle [\Rightarrow] \langle es_1, (h_1, ls_1) \rangle \Longrightarrow \text{length } ls_0 = \text{length } ls_1$$

**lemma** *evals<sub>1</sub>-preserves-elen*:

$$\bigwedge es' \ s \ s'. P \vdash_1 \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \Longrightarrow \text{length } es = \text{length } es'$$

**lemma** *eval<sub>1</sub>-final*:  $P \vdash_1 \langle e, s \rangle \Rightarrow \langle e', s' \rangle \Longrightarrow \text{final } e'$

**and** *evals<sub>1</sub>-final*:  $P \vdash_1 \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \Longrightarrow \text{finals } es'$

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*<sub>1</sub>:

$$\llbracket P, E \vdash_1 e :: \text{Class } C; P \vdash C \text{ sees } M:Ts' \rightarrow T = m \text{ in } D;$$

$$P, E \vdash_1 es \llbracket :: \rrbracket Ts; P \vdash Ts \llbracket \leq \rrbracket Ts' \rrbracket$$

$$\Longrightarrow P, E \vdash_1 e \cdot M(es) :: T$$

| *WTBlock*<sub>1</sub>:

$$\llbracket \text{is-type } P T; P, E@[T] \vdash_1 e :: T' \rrbracket$$

$$\Longrightarrow P, E \vdash_1 \{i:T; e\} :: T'$$

| *WTSeq*<sub>1</sub>:

$$\llbracket P, E \vdash_1 e_1 :: T_1; P, E \vdash_1 e_2 :: T_2 \rrbracket$$

$$\Longrightarrow P, E \vdash_1 e_1;;e_2 :: T_2$$

| *WTCond*<sub>1</sub>:

$$\llbracket P, E \vdash_1 e :: \text{Boolean}; P, E \vdash_1 e_1 :: T_1; P, E \vdash_1 e_2 :: T_2;$$

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

$$\Longrightarrow P, E \vdash_1 \text{if } (e) e_1 \text{ else } e_2 :: T$$

| *WTWhile*<sub>1</sub>:

$$\llbracket P, E \vdash_1 e :: \text{Boolean}; P, E \vdash_1 c :: T \rrbracket$$

$$\Longrightarrow P, E \vdash_1 \text{while } (e) c :: \text{Void}$$

| *WTThrow*<sub>1</sub>:

$$P, E \vdash_1 e :: \text{Class } C \Longrightarrow$$

$$P, E \vdash_1 \text{throw } e :: \text{Void}$$

| *WTTry*<sub>1</sub>:

$$\llbracket P, E \vdash_1 e_1 :: T; P, E@[Class C] \vdash_1 e_2 :: T; \text{is-class } P C \rrbracket$$

$$\Longrightarrow P, E \vdash_1 \text{try } e_1 \text{ catch}(C i) e_2 :: T$$

| *WTNil*<sub>1</sub>:

$$P, E \vdash_1 [] \llbracket :: \rrbracket []$$

| *WTCons*<sub>1</sub>:

$$\llbracket P, E \vdash_1 e :: T; P, E \vdash_1 es \llbracket :: \rrbracket Ts \rrbracket$$

$$\Longrightarrow P, E \vdash_1 e \# es \llbracket :: \rrbracket T \# Ts$$

**lemma** *WTs*<sub>1</sub>-same-size:  $\bigwedge Ts. P, E \vdash_1 es \llbracket :: \rrbracket Ts \Longrightarrow \text{size } es = \text{size } Ts$

**lemma** *WT*<sub>1</sub>-unique:

$$P, E \vdash_1 e :: T_1 \Longrightarrow (\bigwedge T_2. P, E \vdash_1 e :: T_2 \Longrightarrow T_1 = T_2) \text{ and}$$

$$P, E \vdash_1 es \llbracket :: \rrbracket Ts_1 \Longrightarrow (\bigwedge Ts_2. P, E \vdash_1 es \llbracket :: \rrbracket Ts_2 \Longrightarrow Ts_1 = Ts_2)$$

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

**shows** *WT*<sub>1</sub>-*is-type*:  $P, E \vdash_1 e :: T \Longrightarrow \text{set } E \subseteq \text{types } P \Longrightarrow \text{is-type } P T$

**and**  $P, E \vdash_1 es \llbracket :: \rrbracket Ts \Longrightarrow \text{True}$

## 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**

$\mathcal{B} (\text{new } C) i = \text{True}$  |



$$\begin{aligned}
& \mathcal{B} (\text{Cast } C \ e) \ i = \mathcal{B} \ e \ i \mid \\
& \mathcal{B} (\text{Val } v) \ i = \text{True} \mid \\
& \mathcal{B} (e_1 \ll \text{bop} \gg e_2) \ i = (\mathcal{B} \ e_1 \ i \wedge \mathcal{B} \ e_2 \ i) \mid \\
& \mathcal{B} (\text{Var } j) \ i = \text{True} \mid \\
& \mathcal{B} (e \cdot F\{D\}) \ i = \mathcal{B} \ e \ i \mid \\
& \mathcal{B} (j := e) \ i = \mathcal{B} \ e \ i \mid \\
& \mathcal{B} (e_1 \cdot F\{D\} := e_2) \ i = (\mathcal{B} \ e_1 \ i \wedge \mathcal{B} \ e_2 \ i) \mid \\
& \mathcal{B} (e \cdot M(es)) \ i = (\mathcal{B} \ e \ i \wedge \mathcal{B} s \ es \ i) \mid \\
& \mathcal{B} (\{j:T ; e\}) \ i = (i = j \wedge \mathcal{B} \ e \ (i+1)) \mid \\
& \mathcal{B} (e_1 ;; e_2) \ i = (\mathcal{B} \ e_1 \ i \wedge \mathcal{B} \ e_2 \ i) \mid \\
& \mathcal{B} (\text{if } (e) \ e_1 \ \text{else } e_2) \ i = (\mathcal{B} \ e \ i \wedge \mathcal{B} \ e_1 \ i \wedge \mathcal{B} \ e_2 \ i) \mid \\
& \mathcal{B} (\text{throw } e) \ i = \mathcal{B} \ e \ i \mid \\
& \mathcal{B} (\text{while } (e) \ c) \ i = (\mathcal{B} \ e \ i \wedge \mathcal{B} \ c \ i) \mid \\
& \mathcal{B} (\text{try } e_1 \ \text{catch}(C \ j) \ e_2) \ i = (\mathcal{B} \ e_1 \ i \wedge i=j \wedge \mathcal{B} \ e_2 \ (i+1)) \mid \\
& \mathcal{B} s \ [] \ i = \text{True} \mid \\
& \mathcal{B} s \ (e \# es) \ i = (\mathcal{B} \ e \ i \wedge \mathcal{B} s \ es \ i)
\end{aligned}$$

**definition**  $wf\text{-}J_1\text{-mdecl} :: J_1\text{-prog} \Rightarrow \text{cname} \Rightarrow \text{expr}_1 \text{ mdecl} \Rightarrow \text{bool}$

**where**

$$\begin{aligned}
& wf\text{-}J_1\text{-mdecl} \ P \ C \equiv \lambda(M, Ts, T, body). \\
& (\exists T'. P, \text{Class } C \# Ts \vdash_1 \text{body} :: T' \wedge P \vdash T' \leq T) \wedge \\
& \mathcal{D} \ \text{body} \ [ \{ ..size \ Ts \} ] \wedge \mathcal{B} \ \text{body} \ (size \ Ts + 1)
\end{aligned}$$

**lemma**  $wf\text{-}J_1\text{-mdecl}[simp]$ :

$$\begin{aligned}
& wf\text{-}J_1\text{-mdecl} \ P \ C \ (M, Ts, T, body) \equiv \\
& ((\exists T'. P, \text{Class } C \# Ts \vdash_1 \text{body} :: T' \wedge P \vdash T' \leq T) \wedge \\
& \mathcal{D} \ \text{body} \ [ \{ ..size \ Ts \} ] \wedge \mathcal{B} \ \text{body} \ (size \ Ts + 1))
\end{aligned}$$

**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**  $compM :: ('a \Rightarrow 'b) \Rightarrow 'a \ \text{mdecl} \Rightarrow 'b \ \text{mdecl}$

**where**

$$compM \ f \equiv \lambda(M, Ts, T, m). (M, Ts, T, f \ m)$$

**definition**  $compC :: ('a \Rightarrow 'b) \Rightarrow 'a \ \text{cdecl} \Rightarrow 'b \ \text{cdecl}$

**where**

$$compC \ f \equiv \lambda(C, D, Fdecls, Mdecls). (C, D, Fdecls, \text{map} \ (compM \ f) \ Mdecls)$$

**definition**  $compP :: ('a \Rightarrow 'b) \Rightarrow 'a \ \text{prog} \Rightarrow 'b \ \text{prog}$

**where**

$$compP \ f \equiv \text{map} \ (compC \ f)$$

Compilation preserves the program structure. Therefore lookup functions either commute with compilation (like method lookup) or are preserved by it (like the subclass relation).

**lemma**  $map\text{-of-map4}$ :

$map\text{-of } (map (\lambda(x,a,b,c).(x,a,b,f c)) ts) =$   
 $map\text{-option } (\lambda(a,b,c).(a,b,f c)) \circ (map\text{-of } ts)$

**lemma** *class-compP*:

$class P C = Some (D, fs, ms)$   
 $\implies class (compP f P) C = Some (D, fs, map (compM f) ms)$

**lemma** *class-compPD*:

$class (compP f P) C = Some (D, fs, cms)$   
 $\implies \exists ms. class P C = Some(D,fs,ms) \wedge cms = map (compM f) ms$

**lemma** [*simp*]:  $is\text{-class } (compP f P) C = is\text{-class } P C$

**lemma** [*simp*]:  $class (compP f P) C = map\text{-option } (\lambda c. snd(compC f (C,c))) (class P C)$

**lemma** *sees-methods-compP*:

$P \vdash C \text{ sees-methods } Mm \implies$   
 $compP f P \vdash C \text{ sees-methods } (map\text{-option } (\lambda((Ts,T,m),D). ((Ts,T,f m),D)) \circ Mm)$

**lemma** *sees-method-compP*:

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

**lemma** [*simp*]:

$P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } D \implies$   
 $method (compP f P) C M = (D, Ts, T, f m)$

**lemma** *sees-methods-compPD*:

$\llbracket cP \vdash C \text{ sees-methods } Mm'; cP = compP f P \rrbracket \implies$   
 $\exists Mm. P \vdash C \text{ sees-methods } Mm \wedge$   
 $Mm' = (map\text{-option } (\lambda((Ts,T,m),D). ((Ts,T,f m),D)) \circ Mm)$

**lemma** *sees-method-compPD*:

$compP f P \vdash C \text{ sees } M: Ts \rightarrow T = fm \text{ in } D \implies$   
 $\exists m. P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } D \wedge f m = fm$

**lemma** [*simp*]:  $subcls1 (compP f P) = subcls1 P$

**lemma** *compP-widen*[*simp*]:  $(compP f P \vdash T \leq T') = (P \vdash T \leq T')$

**lemma** [*simp*]:  $(compP f P \vdash Ts [\leq] Ts') = (P \vdash Ts [\leq] Ts')$

**lemma** [*simp*]:  $is\text{-type } (compP f P) T = is\text{-type } P T$

**lemma** [*simp*]:  $(compP (f::'a \Rightarrow 'b) P \vdash C \text{ has-fields } FDTs) = (P \vdash C \text{ has-fields } FDTs)$

**lemma** [*simp*]:  $fields (compP f P) C = fields P C$

**lemma** [*simp*]:  $(compP f P \vdash C \text{ sees } F:T \text{ in } D) = (P \vdash C \text{ sees } F:T \text{ in } D)$

**lemma** [*simp*]:  $field (compP f P) F D = field P F D$

### 5.3.1 Invariance of *wf-prog* under compilation

**lemma** [iff]:  $\text{distinct-fst } (\text{compP } f \ P) = \text{distinct-fst } P$

**lemma** [iff]:  $\text{distinct-fst } (\text{map } (\text{compM } f) \ ms) = \text{distinct-fst } ms$

**lemma** [iff]:  $\text{wf-syscls } (\text{compP } f \ P) = \text{wf-syscls } P$

**lemma** [iff]:  $\text{wf-fdecl } (\text{compP } f \ P) = \text{wf-fdecl } P$

**lemma** *set-compP*:

$((C, D, fs, ms') \in \text{set}(\text{compP } f \ P)) =$   
 $(\exists ms. (C, D, fs, ms) \in \text{set } P \wedge ms' = \text{map } (\text{compM } f) \ ms)$

**lemma** *wf-cdecl-compPI*:

$\llbracket \bigwedge C \ M \ Ts \ T \ m.$   
 $\llbracket \text{wf-mdecl } wf_1 \ P \ C \ (M, Ts, T, m); P \vdash C \ \text{sees } M:Ts \rightarrow T = m \ \text{in } C \rrbracket$   
 $\implies \text{wf-mdecl } wf_2 \ (\text{compP } f \ P) \ C \ (M, Ts, T, f \ m);$   
 $\forall x \in \text{set } P. \ \text{wf-cdecl } wf_1 \ P \ x; x \in \text{set } (\text{compP } f \ P); \text{wf-prog } p \ P \rrbracket$   
 $\implies \text{wf-cdecl } wf_2 \ (\text{compP } f \ P) \ x$

**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**  $\text{wf-prog } wf_2 \ (\text{compP } f \ P)$

**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*:  $x \in \text{set } xs \implies \text{hidden } (xs \ @ \ [x]) \ (\text{last-index } xs \ x)$

**by**(*auto simp add: hidden-def nth-append rev-nth[symmetric]*  
*dest: last-index-less[OF - le-refl]*)

**lemma** *hidden-inacc*:  $\text{hidden } xs \ i \implies \text{last-index } xs \ x \neq i$

**by**(*auto simp add: hidden-def last-index-drop last-index-less-size-conv*)

**lemma** [*simp*]:  $\text{hidden } xs \ i \implies \text{hidden } (xs@[x]) \ i$

**by**(*auto simp add: hidden-def nth-append*)

**lemma** *fun-upds-apply*:

$(m(xs[\mapsto]ys)) \ x =$   
 $(\text{let } xs' = \text{take } (\text{size } ys) \ xs$   
 $\text{in if } x \in \text{set } xs' \ \text{then } \text{Some}(ys \ ! \ \text{last-index } xs' \ x) \ \text{else } m \ x)$

```

proof(induct xs arbitrary: m ys)
  case Nil then show ?case by(simp add: Let-def)
next
  case Cons show ?case
  proof(cases ys)
    case Nil
    then show ?thesis by(simp add:Let-def)
  next
    case Cons': Cons
    then show ?thesis using Cons by(simp add: Let-def last-index-Cons)
  qed
qed

```

**lemma** *map-upds-apply-eq-Some*:  
 $((m(xs[\mapsto]ys)) x = \text{Some } y) =$   
 $(\text{let } xs' = \text{take } (\text{size } ys) \text{ } xs$   
*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)  
**by**(*simp add:fun-upds-apply Let-def*)*

**lemma** *map-upds-upd-conv-last-index*:  
 $\llbracket x \in \text{set } xs; \text{size } xs \leq \text{size } ys \rrbracket$   
 $\implies m(xs[\mapsto]ys, x \mapsto y) = m(xs[\mapsto]ys[\text{last-index } xs \text{ } x := y])$   
**by**(*rule ext*) (*simp add:fun-upds-apply eq-sym-conv Let-def*)

**end**

## 5.4 Compilation Stage 1

**theory** *Compiler1* **imports** *PCompiler J1 Hidden* **begin**

Replacing variable names by indices.

```

primrec compE1 :: vname list  $\Rightarrow$  expr  $\Rightarrow$  expr1
  and compEs1 :: vname list  $\Rightarrow$  expr list  $\Rightarrow$  expr1 list where
  compE1 Vs (new C) = new C
| compE1 Vs (Cast C e) = Cast C (compE1 Vs e)
| compE1 Vs (Val v) = Val v
| compE1 Vs (e1 «bop» e2) = (compE1 Vs e1) «bop» (compE1 Vs e2)
| compE1 Vs (Var V) = Var(last-index Vs V)
| compE1 Vs (V:=e) = (last-index Vs V):= (compE1 Vs e)
| compE1 Vs (e•F{D}) = (compE1 Vs e)•F{D}
| compE1 Vs (e1•F{D}):=e2) = (compE1 Vs e1)•F{D} := (compE1 Vs e2)
| compE1 Vs (e•M(es)) = (compE1 Vs e)•M(compEs1 Vs es)
| compE1 Vs {V:T; e} = {(size Vs):T; compE1 (Vs@[V]) e}
| compE1 Vs (e1;;e2) = (compE1 Vs e1);;(compE1 Vs e2)
| compE1 Vs (if (e) e1 else e2) = if (compE1 Vs e) (compE1 Vs e1) else (compE1 Vs e2)
| compE1 Vs (while (e) c) = while (compE1 Vs e) (compE1 Vs c)
| compE1 Vs (throw e) = throw (compE1 Vs e)
| compE1 Vs (try e1 catch(C V) e2) =
  try(compE1 Vs e1) catch(C (size Vs)) (compE1 (Vs@[V]) e2)

| compEs1 Vs [] = []
| compEs1 Vs (e#es) = compE1 Vs e # compEs1 Vs es

```

**lemma** [simp]:  $\text{compEs}_1 \text{ Vs } es = \text{map } (\text{compE}_1 \text{ Vs}) es$

**primrec**  $\text{fin}_1 :: \text{expr} \Rightarrow \text{expr}_1$  **where**

$\text{fin}_1(\text{Val } v) = \text{Val } v$   
 $|\ \text{fin}_1(\text{throw } e) = \text{throw}(\text{fin}_1 e)$

**lemma** *comp-final*:  $\text{final } e \Longrightarrow \text{compE}_1 \text{ Vs } e = \text{fin}_1 e$

**lemma** [simp]:

$\bigwedge \text{Vs. } \text{max-vars } (\text{compE}_1 \text{ Vs } e) = \text{max-vars } e$   
**and**  $\bigwedge \text{Vs. } \text{max-varss } (\text{compEs}_1 \text{ Vs } es) = \text{max-varss } es$

Compiling programs:

**definition**  $\text{compP}_1 :: J\text{-prog} \Rightarrow J_1\text{-prog}$

**where**

$\text{compP}_1 \equiv \text{compP } (\lambda(\text{pns}, \text{body}). \text{compE}_1 (\text{this}\#\text{pns}) \text{ body})$

**end**

## 5.5 Correctness of Stage 1

**theory** *Correctness1*

**imports** *J1WellForm Compiler1*

**begin**

### 5.5.1 Correctness of program compilation

**primrec**  $\text{unmod} :: \text{expr}_1 \Rightarrow \text{nat} \Rightarrow \text{bool}$

**and**  $\text{unmods} :: \text{expr}_1 \text{ list} \Rightarrow \text{nat} \Rightarrow \text{bool}$  **where**

$\text{unmod } (\text{new } C) i = \text{True} \mid$   
 $\text{unmod } (\text{Cast } C e) i = \text{unmod } e i \mid$   
 $\text{unmod } (\text{Val } v) i = \text{True} \mid$   
 $\text{unmod } (e_1 \ll\text{bop}\gg e_2) i = (\text{unmod } e_1 i \wedge \text{unmod } e_2 i) \mid$   
 $\text{unmod } (\text{Var } i) j = \text{True} \mid$   
 $\text{unmod } (i:=e) j = (i \neq j \wedge \text{unmod } e j) \mid$   
 $\text{unmod } (e \cdot F\{D\}) i = \text{unmod } e i \mid$   
 $\text{unmod } (e_1 \cdot F\{D\} := e_2) i = (\text{unmod } e_1 i \wedge \text{unmod } e_2 i) \mid$   
 $\text{unmod } (e \cdot M(es)) i = (\text{unmod } e i \wedge \text{unmods } es i) \mid$   
 $\text{unmod } \{j:T; e\} i = \text{unmod } e i \mid$   
 $\text{unmod } (e_1;;e_2) i = (\text{unmod } e_1 i \wedge \text{unmod } e_2 i) \mid$   
 $\text{unmod } (\text{if } (e) e_1 \text{ else } e_2) i = (\text{unmod } e i \wedge \text{unmod } e_1 i \wedge \text{unmod } e_2 i) \mid$   
 $\text{unmod } (\text{while } (e) c) i = (\text{unmod } e i \wedge \text{unmod } c i) \mid$   
 $\text{unmod } (\text{throw } e) i = \text{unmod } e i \mid$   
 $\text{unmod } (\text{try } e_1 \text{ catch}(C i) e_2) j = (\text{unmod } e_1 j \wedge (\text{if } i=j \text{ then } \text{False} \text{ else } \text{unmod } e_2 j)) \mid$

$\text{unmods } ([]) i = \text{True} \mid$

$\text{unmods } (e\#es) i = (\text{unmod } e i \wedge \text{unmods } es i)$

**lemma** *hidden-unmod*:  $\bigwedge \text{Vs. } \text{hidden } \text{Vs } i \Longrightarrow \text{unmod } (\text{compE}_1 \text{ Vs } e) i$  **and**

$\bigwedge \text{Vs. } \text{hidden } \text{Vs } i \Longrightarrow \text{unmods } (\text{compEs}_1 \text{ Vs } es) i$

**lemma** *eval<sub>1</sub>-preserves-unmod*:

$\llbracket P \vdash_1 \langle e, (h, ls) \rangle \Rightarrow \langle e', (h', ls') \rangle; \text{unmod } e i; i < \text{size } ls \rrbracket$

$\implies ls ! i = ls' ! i$   
**and**  $\llbracket P \vdash_1 \langle es, (h, ls) \rangle \rrbracket [\Rightarrow] \langle es', (h', ls') \rangle; \text{unmods } es \ i; i < \text{size } ls \rrbracket$   
 $\implies ls ! i = ls' ! i$

**lemma** *LAss-lem*:

$\llbracket x \in \text{set } xs; \text{size } xs \leq \text{size } ys \rrbracket$   
 $\implies m_1 \subseteq_m m_2(xs[\mapsto]ys) \implies m_1(x \mapsto y) \subseteq_m m_2(xs[\mapsto]ys[\text{last-index } xs \ x := y])$  **lemma** *Block-lem*:  
**fixes**  $l :: 'a \rightarrow 'b$

**assumes**  $0: l \subseteq_m [Vs [\mapsto] ls]$

**and**  $1: l' \subseteq_m [Vs [\mapsto] ls', V \mapsto v]$

**and** *hidden*:  $V \in \text{set } Vs \implies ls ! \text{last-index } Vs \ V = ls' ! \text{last-index } Vs \ V$

**and** *size*:  $\text{size } ls = \text{size } ls' \quad \text{size } Vs < \text{size } ls'$

**shows**  $l'(V := l \ V) \subseteq_m [Vs [\mapsto] ls']$

The main theorem:

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

**shows** *eval<sub>1</sub>-eval*:  $P \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle$

$\implies (\bigwedge Vs \ ls. \llbracket fv \ e \subseteq \text{set } Vs; l \subseteq_m [Vs[\mapsto]ls]; \text{size } Vs + \text{max-vars } e \leq \text{size } ls \rrbracket$

$\implies \exists ls'. \text{comp}P_1 \ P \vdash_1 \langle \text{comp}E_1 \ Vs \ e, (h, ls) \rangle \Rightarrow \langle \text{fin}_1 \ e', (h', ls') \rangle \wedge l' \subseteq_m [Vs[\mapsto]ls']$ )

**and** *evals<sub>1</sub>-evals*:  $P \vdash \langle es, (h, l) \rangle [\Rightarrow] \langle es', (h', l') \rangle$

$\implies (\bigwedge Vs \ ls. \llbracket fvs \ es \subseteq \text{set } Vs; l \subseteq_m [Vs[\mapsto]ls]; \text{size } Vs + \text{max-varss } es \leq \text{size } ls \rrbracket$

$\implies \exists ls'. \text{comp}P_1 \ P \vdash_1 \langle \text{comp}Es_1 \ Vs \ es, (h, ls) \rangle [\Rightarrow] \langle \text{comp}Es_1 \ Vs \ es', (h', ls') \rangle \wedge$   
 $l' \subseteq_m [Vs[\mapsto]ls']$ )

## 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<sub>1</sub>-pres-wt*:  $\bigwedge Vs \ Ts \ U.$

$\llbracket P, [Vs[\mapsto]Ts] \vdash e :: U; \text{size } Ts = \text{size } Vs \rrbracket$

$\implies \text{comp}P \ f \ P, Ts \vdash_1 \text{comp}E_1 \ Vs \ e :: U$

**and**  $\bigwedge Vs \ Ts \ Us.$

$\llbracket P, [Vs[\mapsto]Ts] \vdash es [::] Us; \text{size } Ts = \text{size } Vs \rrbracket$

$\implies \text{comp}P \ f \ P, Ts \vdash_1 \text{comp}Es_1 \ Vs \ es [::] Us$

and the correct block numbering:

**lemma**  $\mathcal{B}$ :  $\bigwedge Vs \ n. \text{size } Vs = n \implies \mathcal{B}(\text{comp}E_1 \ Vs \ e) \ n$

**and**  $\mathcal{B}s$ :  $\bigwedge Vs \ n. \text{size } Vs = n \implies \mathcal{B}s(\text{comp}Es_1 \ Vs \ es) \ n$

The main complication is preservation of definite assignment  $\mathcal{D}$ .

**lemma** *image-last-index*:  $A \subseteq \text{set}(xs@[x]) \implies \text{last-index}(xs \ @ \ [x]) \ 'A =$

$(\text{if } x \in A \text{ then insert } (\text{size } xs) \ (\text{last-index } xs \ ' (A - \{x\})) \text{ else last-index } xs \ ' A)$

**lemma** *A-compE<sub>1</sub>-None[simp]*:

$\bigwedge Vs. \mathcal{A} \ e = \text{None} \implies \mathcal{A}(\text{comp}E_1 \ Vs \ e) = \text{None}$

**and**  $\bigwedge Vs. \mathcal{A}s \ es = \text{None} \implies \mathcal{A}s(\text{comp}Es_1 \ Vs \ es) = \text{None}$

**lemma** *A-compE<sub>1</sub>*:

$\bigwedge A \ Vs. \llbracket \mathcal{A} \ e = [A]; fv \ e \subseteq \text{set } Vs \rrbracket \implies \mathcal{A}(\text{comp}E_1 \ Vs \ e) = [\text{last-index } Vs \ ' A]$

**and**  $\bigwedge A \ Vs. \llbracket \mathcal{A}s \ es = [A]; fvs \ es \subseteq \text{set } Vs \rrbracket \implies \mathcal{A}s(\text{comp}Es_1 \ Vs \ es) = [\text{last-index } Vs \ ' A]$

**lemma** *D-None[iff]*:  $\mathcal{D}(e::'a \ \text{exp}) \ \text{None}$  **and**  $[\text{iff}]: \mathcal{D}s(es::'a \ \text{exp list}) \ \text{None}$

**lemma** *D-last-index-compE<sub>1</sub>*:

$$\bigwedge A \text{ Vs. } \llbracket A \subseteq \text{set Vs}; \text{fv } e \subseteq \text{set Vs} \rrbracket \implies \\ \mathcal{D} \ e \ [A] \implies \mathcal{D} \ (\text{compE}_1 \text{ Vs } e) \ [\text{last-index Vs ' } A]$$

**and**  $\bigwedge A \text{ Vs. } \llbracket A \subseteq \text{set Vs}; \text{fvs } es \subseteq \text{set Vs} \rrbracket \implies$

$$\mathcal{D} \ s \ es \ [A] \implies \mathcal{D} \ s \ (\text{compEs}_1 \text{ Vs } es) \ [\text{last-index Vs ' } A]$$

**lemma** *last-index-image-set: distinct xs  $\implies$  last-index xs ' set xs = {..*

**lemma** *D-compE<sub>1</sub>*:

$$\llbracket \mathcal{D} \ e \ [\text{set Vs}]; \text{fv } e \subseteq \text{set Vs}; \text{distinct Vs} \rrbracket \implies \mathcal{D} \ (\text{compE}_1 \text{ Vs } e) \ [\{\text{..$$

**lemma** *D-compE<sub>1</sub>'*:

**assumes**  $\mathcal{D} \ e \ [\text{set}(V\#Vs)]$  **and**  $\text{fv } e \subseteq \text{set}(V\#Vs)$  **and**  $\text{distinct}(V\#Vs)$

**shows**  $\mathcal{D} \ (\text{compE}_1 (V\#Vs) \ e) \ [\{\text{..length Vs}\}]$

**lemma** *compP<sub>1</sub>-pres-wf: wf-J-prog P  $\implies$  wf-J<sub>1</sub>-prog (compP<sub>1</sub> P)*

**end**

## 5.6 Compilation Stage 2

**theory** *Compiler2*

**imports** *PCompiler J1 ../JVM/JVMExec*

**begin**

**primrec** *compE<sub>2</sub>* :: *expr<sub>1</sub>  $\Rightarrow$  instr list*

**and** *compEs<sub>2</sub>* :: *expr<sub>1</sub> list  $\Rightarrow$  instr list* **where**

$$\text{compE}_2 \ (\text{new } C) = [\text{New } C]$$

$$| \text{compE}_2 \ (\text{Cast } C \ e) = \text{compE}_2 \ e \ @ \ [\text{Checkcast } C]$$

$$| \text{compE}_2 \ (\text{Val } v) = [\text{Push } v]$$

$$| \text{compE}_2 \ (e_1 \ll \text{bop} \gg e_2) = \text{compE}_2 \ e_1 \ @ \ \text{compE}_2 \ e_2 \ @$$

$$(\text{case } \text{bop} \ \text{of } \text{Eq} \Rightarrow [\text{CmpEq}]$$

$$| \text{Add} \Rightarrow [\text{IAdd}])$$

$$| \text{compE}_2 \ (\text{Var } i) = [\text{Load } i]$$

$$| \text{compE}_2 \ (i := e) = \text{compE}_2 \ e \ @ \ [\text{Store } i, \text{Push Unit}]$$

$$| \text{compE}_2 \ (e \cdot F \{D\}) = \text{compE}_2 \ e \ @ \ [\text{Getfield } F \ D]$$

$$| \text{compE}_2 \ (e_1 \cdot F \{D\} := e_2) =$$

$$\text{compE}_2 \ e_1 \ @ \ \text{compE}_2 \ e_2 \ @ \ [\text{Putfield } F \ D, \text{Push Unit}]$$

$$| \text{compE}_2 \ (e \cdot M(es)) = \text{compE}_2 \ e \ @ \ \text{compEs}_2 \ es \ @ \ [\text{Invoke } M \ (\text{size } es)]$$

$$| \text{compE}_2 \ (\{i:T; e\}) = \text{compE}_2 \ e$$

$$| \text{compE}_2 \ (e_1 ;; e_2) = \text{compE}_2 \ e_1 \ @ \ [\text{Pop}] \ @ \ \text{compE}_2 \ e_2$$

$$| \text{compE}_2 \ (\text{if } (e) \ e_1 \ \text{else } e_2) =$$

$$(\text{let } \text{cnd} \ = \ \text{compE}_2 \ e;$$

$$\text{thn} \ = \ \text{compE}_2 \ e_1;$$

$$\text{els} \ = \ \text{compE}_2 \ e_2;$$

$$\text{test} \ = \ \text{IfFalse} \ (\text{int}(\text{size } \text{thn} \ + \ 2));$$

$$\text{thnex} \ = \ \text{Goto} \ (\text{int}(\text{size } \text{els} \ + \ 1))$$

$$\text{in } \text{cnd} \ @ \ [\text{test}] \ @ \ \text{thn} \ @ \ [\text{thnex}] \ @ \ \text{els})$$

$$| \text{compE}_2 \ (\text{while } (e) \ c) =$$

$$(\text{let } \text{cnd} \ = \ \text{compE}_2 \ e;$$

$$\text{bdy} \ = \ \text{compE}_2 \ c;$$

$$\text{test} \ = \ \text{IfFalse} \ (\text{int}(\text{size } \text{bdy} \ + \ 3));$$

$$\text{loop} \ = \ \text{Goto} \ (-\text{int}(\text{size } \text{bdy} \ + \ \text{size } \text{cnd} \ + \ 2))$$

$$\begin{aligned}
& \text{in } \text{cnd} \ @ \ [\text{test}] \ @ \ \text{bdy} \ @ \ [\text{Pop}] \ @ \ [\text{loop}] \ @ \ [\text{Push Unit}] \\
| \text{compE}_2 \ (\text{throw } e) &= \text{compE}_2 \ e \ @ \ [\text{instr. Throw}] \\
| \text{compE}_2 \ (\text{try } e_1 \ \text{catch}(C \ i) \ e_2) &= \\
& \ (\text{let } \text{catch} = \text{compE}_2 \ e_2 \\
& \ \text{in } \text{compE}_2 \ e_1 \ @ \ [\text{Goto } (\text{int}(\text{size } \text{catch})+2), \ \text{Store } i] \ @ \ \text{catch}) \\
| \text{compEs}_2 \ [] &= [] \\
| \text{compEs}_2 \ (e\#es) &= \text{compE}_2 \ e \ @ \ \text{compEs}_2 \ es
\end{aligned}$$

Compilation of exception table. Is given start address of code to compute absolute addresses necessary in exception table.

**primrec**  $\text{compxE}_2 :: \text{expr}_1 \Rightarrow \text{pc} \Rightarrow \text{nat} \Rightarrow \text{ex-table}$   
**and**  $\text{compxEs}_2 :: \text{expr}_1 \ \text{list} \Rightarrow \text{pc} \Rightarrow \text{nat} \Rightarrow \text{ex-table}$  **where**

$$\begin{aligned}
& \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 &= \\
& \ \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 &= \\
& \ \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 &= \\
& \ \text{compxE}_2 \ e \ \text{pc} \ d \ @ \ \text{compxEs}_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 &= \\
& \ \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 &= \\
& \ (\text{let } \text{pc}_1 = \text{pc} + \text{size}(\text{compE}_2 \ e) + 1; \\
& \ \text{pc}_2 = \text{pc}_1 + \text{size}(\text{compE}_2 \ e_1) + 1 \\
& \ \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 &= \\
& \ \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 &= \\
& \ (\text{let } \text{pc}_1 = \text{pc} + \text{size}(\text{compE}_2 \ e_1) \\
& \ \text{in } \text{compxE}_2 \ e_1 \ \text{pc} \ d \ @ \ \text{compxE}_2 \ e_2 \ (\text{pc}_1 + 2) \ d \ @ \ [(\text{pc}, \text{pc}_1, C, \text{pc}_1 + 1, d)]) \\
| \text{compxEs}_2 \ [] \ \text{pc} \ d &= [] \\
| \text{compxEs}_2 \ (e\#es) \ \text{pc} \ d &= \text{compxE}_2 \ e \ \text{pc} \ d \ @ \ \text{compxEs}_2 \ es \ (\text{pc} + \text{size}(\text{compE}_2 \ e)) \ (d+1)
\end{aligned}$$

**primrec**  $\text{max-stack} :: \text{expr}_1 \Rightarrow \text{nat}$   
**and**  $\text{max-stacks} :: \text{expr}_1 \ \text{list} \Rightarrow \text{nat}$  **where**

$$\begin{aligned}
& \text{max-stack} \ (\text{new } C) = 1 \\
| \text{max-stack} \ (\text{Cast } C \ e) &= \text{max-stack} \ e \\
| \text{max-stack} \ (\text{Val } v) &= 1 \\
| \text{max-stack} \ (e_1 \ \ll\text{bop}\gg \ e_2) &= \max(\text{max-stack} \ e_1) \ (\text{max-stack} \ e_2) + 1 \\
| \text{max-stack} \ (\text{Var } i) &= 1 \\
| \text{max-stack} \ (i:=e) &= \text{max-stack} \ e \\
| \text{max-stack} \ (e \cdot F\{D\}) &= \text{max-stack} \ e \\
| \text{max-stack} \ (e_1 \cdot F\{D\} := e_2) &= \max(\text{max-stack} \ e_1) \ (\text{max-stack} \ e_2) + 1 \\
| \text{max-stack} \ (e \cdot M(es)) &= \max(\text{max-stack} \ e) \ (\text{max-stacks} \ es) + 1 \\
| \text{max-stack} \ (\{i:T; e\}) &= \text{max-stack} \ e
\end{aligned}$$



```

| max-stack (e1;;e2) = max (max-stack e1) (max-stack e2)
| max-stack (if (e) e1 else e2) =
  max (max-stack e) (max (max-stack e1) (max-stack e2))
| max-stack (while (e) c) = max (max-stack e) (max-stack c)
| max-stack (throw e) = max-stack e
| max-stack (try e1 catch(C i) e2) = max (max-stack e1) (max-stack e2)

| max-stacks [] = 0
| max-stacks (e#es) = max (max-stack e) (1 + max-stacks es)

```

**lemma** *max-stack1*:  $1 \leq \text{max-stack } e$

**definition** *compMb<sub>2</sub>* :: *expr<sub>1</sub>* ⇒ *jvm-method*

**where**

```

compMb2 ≡ λbody.
  let ins = compE2 body @ [Return];
      xt = compxE2 body 0 0
  in (max-stack body, max-vars body, ins, xt)

```

**definition** *compP<sub>2</sub>* :: *J<sub>1</sub>-prog* ⇒ *jvm-prog*

**where**

```

compP2 ≡ compP compMb2

```

**lemma** *compMb<sub>2</sub>* [*simp*]:

```

compMb2 e = (max-stack e, max-vars e, compE2 e @ [Return], compxE2 e 0 0)

```

**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* ⇒ *cname* ⇒ *mname* ⇒ *nat* ⇒ *instr list* ⇒ *bool*

```

((-, -, -, / ▷ -) [51, 0, 0, 0, 51] 50) where
  P, C, M, pc ▷ is ⟷ prefix is (drop pc (instrs-of P C M))

```

**definition** *at* :: *jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *nat* ⇒ *instr* ⇒ *bool*

```

((-, -, -, / ▷ -) [51, 0, 0, 0, 51] 50) where
  P, C, M, pc ▷ i ⟷ (∃ is. drop pc (instrs-of P C M) = i#is)

```

**lemma** [*simp*]:  $P, C, M, pc \triangleright []$

**lemma** [*simp*]:  $P, C, M, pc \triangleright (i\#is) = (P, C, M, pc \triangleright i \wedge P, C, M, pc + 1 \triangleright is)$

**lemma** [*simp*]:  $P, C, M, pc \triangleright (is_1 @ is_2) = (P, C, M, pc \triangleright is_1 \wedge P, C, M, pc + \text{size } is_1 \triangleright is_2)$

**lemma** [simp]:  $P, C, M, pc \triangleright i \implies \text{instrs-of } P \ C \ M \ ! \ pc = i$

**lemma** beforeM:

$P \vdash C \text{ sees } M: Ts \rightarrow T = \text{body in } D \implies$   
 $\text{comp}P_2 \ P, D, M, 0 \triangleright \text{comp}E_2 \ \text{body} \ @ \ [\text{Return}]$

This lemma executes a single instruction by rewriting:

**lemma** [simp]:

$P, C, M, pc \triangleright \text{instr} \implies$   
 $(P \vdash (\text{None}, h, (vs, ls, C, M, pc) \# \text{frs}) \text{ --jvm--} \sigma') =$   
 $((\text{None}, h, (vs, ls, C, M, pc) \# \text{frs}) = \sigma' \vee$   
 $(\exists \sigma. \text{exec}(P, (\text{None}, h, (vs, ls, C, M, pc) \# \text{frs})) = \text{Some } \sigma \wedge P \vdash \sigma \text{ --jvm--} \sigma'))$

## 5.7.2 Exception tables

**definition** pcs :: ex-table  $\Rightarrow$  nat set

where

$\text{pcs } xt \equiv \bigcup (f, t, C, h, d) \in \text{set } xt. \{f \ ..< t\}$

**lemma** pcs-subset:

**shows**  $\bigwedge pc \ d. \text{pcs}(\text{comp}xE_2 \ e \ pc \ d) \subseteq \{pc..<pc+size(\text{comp}E_2 \ e)\}$   
**and**  $\bigwedge pc \ d. \text{pcs}(\text{comp}xEs_2 \ es \ pc \ d) \subseteq \{pc..<pc+size(\text{comp}Es_2 \ es)\}$

**lemma** [simp]:  $\text{pcs } [] = \{\}$

**lemma** [simp]:  $\text{pcs } (x\#\text{xt}) = \{\text{fst } x \ ..< \text{fst}(\text{snd } x)\} \cup \text{pcs } \text{xt}$

**lemma** [simp]:  $\text{pcs}(xt_1 \ @ \ xt_2) = \text{pcs } xt_1 \cup \text{pcs } xt_2$

**lemma** [simp]:  $pc < pc_0 \vee pc_0 + size(\text{comp}E_2 \ e) \leq pc \implies pc \notin \text{pcs}(\text{comp}xE_2 \ e \ pc_0 \ d)$

**lemma** [simp]:  $pc < pc_0 \vee pc_0 + size(\text{comp}Es_2 \ es) \leq pc \implies pc \notin \text{pcs}(\text{comp}xEs_2 \ es \ pc_0 \ d)$

**lemma** [simp]:  $pc_1 + size(\text{comp}E_2 \ e_1) \leq pc_2 \implies \text{pcs}(\text{comp}xE_2 \ e_1 \ pc_1 \ d_1) \cap \text{pcs}(\text{comp}xE_2 \ e_2 \ pc_2 \ d_2) = \{\}$

**lemma** [simp]:  $pc_1 + size(\text{comp}E_2 \ e) \leq pc_2 \implies \text{pcs}(\text{comp}xE_2 \ e \ pc_1 \ d_1) \cap \text{pcs}(\text{comp}xEs_2 \ es \ pc_2 \ d_2) = \{\}$

**lemma** [simp]:

$pc \notin \text{pcs } xt_0 \implies \text{match-ex-table } P \ C \ pc \ (xt_0 \ @ \ xt_1) = \text{match-ex-table } P \ C \ pc \ xt_1$

**lemma** [simp]:  $\llbracket x \in \text{set } xt; pc \notin \text{pcs } xt \rrbracket \implies \neg \text{matches-ex-entry } P \ D \ pc \ x$

**lemma** [simp]:

**assumes**  $xe: xe \in \text{set}(\text{comp}xE_2 \ e \ pc \ d)$  **and** *outside*:  $pc' < pc \vee pc + size(\text{comp}E_2 \ e) \leq pc'$   
**shows**  $\neg \text{matches-ex-entry } P \ C \ pc' \ xe$

**lemma** [simp]:

**assumes**  $xe: xe \in \text{set}(\text{comp}xEs_2 \ es \ pc \ d)$  **and** *outside*:  $pc' < pc \vee pc + size(\text{comp}Es_2 \ es) \leq pc'$   
**shows**  $\neg \text{matches-ex-entry } P \ C \ pc' \ xe$

**lemma** match-ex-table-app[simp]:

$\forall xte \in \text{set } xt_1. \neg \text{matches-ex-entry } P \ D \ pc \ xte \implies$

$match\text{-}ex\text{-}table\ P\ D\ pc\ (xt_1\ @\ xt) = match\text{-}ex\text{-}table\ P\ D\ pc\ xt$

**lemma** [simp]:

$\forall x \in set\ xtab. \neg matches\text{-}ex\text{-}entry\ P\ C\ pc\ x \implies$   
 $match\text{-}ex\text{-}table\ P\ C\ pc\ xtab = None$

**lemma** *match-ex-entry*:

$matches\text{-}ex\text{-}entry\ P\ C\ pc\ (start,\ end,\ catch\text{-}type,\ handler) =$   
 $(start \leq pc \wedge pc < end \wedge P \vdash C \preceq^* catch\text{-}type)$

**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* ::  $jvm\text{-}prog \Rightarrow cname \Rightarrow mname \Rightarrow ex\text{-}table \Rightarrow nat\ set \Rightarrow nat \Rightarrow bool$

$((\lambda -, / -, / - \triangleright / - /' / -, / -) [51, 0, 0, 0, 0, 51] 50)$  **where**

$P, C, M \triangleright xt / I, d \longleftrightarrow$

$(\exists xt_0\ xt_1. ex\text{-}table\text{-}of\ P\ C\ M = xt_0\ @\ xt\ @\ xt_1 \wedge pcs\ xt_0 \cap I = \{\} \wedge pcs\ xt \subseteq I \wedge$   
 $(\forall pc \in I. \forall C\ pc'\ d'. match\text{-}ex\text{-}table\ P\ C\ pc\ xt_1 = [(pc', d')] \longrightarrow d' \leq d))$

**definition** *dummyx* ::  $jvm\text{-}prog \Rightarrow cname \Rightarrow mname \Rightarrow ex\text{-}table \Rightarrow nat\ set \Rightarrow nat \Rightarrow bool$   $((\lambda -, / -, / - \triangleright / - /' / -, / -) [51, 0, 0, 0, 0, 51] 50)$  **where**

$P, C, M \triangleright xt / I, d \longleftrightarrow P, C, M \triangleright xt / I, d$

**abbreviation**

$beforex_0\ P\ C\ M\ d\ I\ xt\ xt_0\ xt_1$

$\equiv ex\text{-}table\text{-}of\ P\ C\ M = xt_0\ @\ xt\ @\ xt_1 \wedge pcs\ xt_0 \cap I = \{\}$

$\wedge pcs\ xt \subseteq I \wedge (\forall pc \in I. \forall C\ pc'\ d'. match\text{-}ex\text{-}table\ P\ C\ pc\ xt_1 = [(pc', d')] \longrightarrow d' \leq d)$

**lemma** *beforex-beforex\_0-eq*:

$P, C, M \triangleright xt / I, d \equiv \exists xt_0\ xt_1. beforex_0\ P\ C\ M\ d\ I\ xt\ xt_0\ xt_1$

**using** *beforex-def* **by** *auto*

**lemma** *beforexD1*:  $P, C, M \triangleright xt / I, d \implies pcs\ xt \subseteq I$

**lemma** *beforex-mono*:  $\llbracket P, C, M \triangleright xt / I, d'; d' \leq d \rrbracket \implies P, C, M \triangleright xt / I, d$

**lemma** [simp]:  $P, C, M \triangleright xt / I, d \implies P, C, M \triangleright xt / I, Suc\ d$

**lemma** *beforex-append*[simp]:

$pcs\ xt_1 \cap pcs\ xt_2 = \{\} \implies$

$P, C, M \triangleright xt_1\ @\ xt_2 / I, d =$

$(P, C, M \triangleright xt_1 / I - pcs\ xt_2, d \wedge P, C, M \triangleright xt_2 / I - pcs\ xt_1, d \wedge P, C, M \triangleright xt_1 @ xt_2 / I, d)$

**lemma** *beforex-appendD1*:

**assumes** *bx*:  $P, C, M \triangleright xt_1\ @\ xt_2\ @\ [(f, t, D, h, d)] / I, d$

**and** *pcs*:  $pcs\ xt_1 \subseteq J$  **and** *JI*:  $J \subseteq I$  **and** *Jpcs*:  $J \cap pcs\ xt_2 = \{\}$

**shows**  $P, C, M \triangleright xt_1 / J, d$

**lemma** *beforex-appendD2*:

**assumes** *bx*:  $P, C, M \triangleright xt_1\ @\ xt_2\ @\ [(f, t, D, h, d)] / I, d$

**and** *pcs*:  $pcs\ xt_2 \subseteq J$  **and** *JI*:  $J \subseteq I$  **and** *Jpcs*:  $J \cap pcs\ xt_1 = \{\}$

**shows**  $P, C, M \triangleright xt_2 / J, d$

**lemma** *beforexM*:

$P \vdash C \text{ sees } M: Ts \rightarrow T = \text{body in } D \implies$   
 $\text{comp}P_2 P, D, M \triangleright \text{comp}xE_2 \text{ body } 0 \ 0 / \{.. < \text{size}(\text{comp}E_2 \text{ body})\}, 0$

**lemma** *match-ex-table-SomeD2*:

**assumes** *met*:  $\text{match-ex-table } P \ D \ pc \ (\text{ex-table-of } P \ C \ M) = \lfloor (pc', d') \rfloor$   
**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**  $d' \leq d$

**lemma** *match-ex-table-SomeD1*:

$\llbracket \text{match-ex-table } P \ D \ pc \ (\text{ex-table-of } P \ C \ M) = \lfloor (pc', d') \rfloor;$   
 $P, C, M \triangleright xt / I, d; pc \in I; pc \notin \text{pcs } xt \rrbracket \implies d' \leq d$

### 5.7.3 The correctness proof

**definition**

$\text{handle} :: \text{jvm-prog} \Rightarrow \text{cname} \Rightarrow \text{mname} \Rightarrow \text{addr} \Rightarrow \text{heap} \Rightarrow \text{val list} \Rightarrow \text{val list} \Rightarrow \text{nat} \Rightarrow \text{frame list}$   
 $\Rightarrow \text{jvm-state}$  **where**  
 $\text{handle } P \ C \ M \ a \ h \ vs \ ls \ pc \ frs = \text{find-handler } P \ a \ h \ ((vs, ls, C, M, pc) \# frs)$

**lemma** *handle-Cons*:

$\llbracket P, C, M \triangleright xt / I, d; d \leq \text{size } vs; pc \in I;$   
 $\forall x \in \text{set } xt. \neg \text{matches-ex-entry } P \ (\text{cname-of } h \ xa) \ pc \ x \rrbracket \implies$   
 $\text{handle } P \ C \ M \ xa \ h \ (v \# vs) \ ls \ pc \ frs = \text{handle } P \ C \ M \ xa \ h \ vs \ ls \ pc \ frs$

**lemma** *handle-append*:

**assumes** *bx*:  $P, C, M \triangleright xt / I, d$  **and** *d*:  $d \leq \text{size } vs$   
**and** *pcI*:  $pc \in I$  **and** *pc-not*:  $pc \notin \text{pcs } xt$   
**shows**  $\text{handle } P \ C \ M \ xa \ h \ (ws \ @ \ vs) \ ls \ pc \ frs = \text{handle } P \ C \ M \ xa \ h \ vs \ ls \ pc \ frs$

**lemma** *aux-isin[simp]*:  $\llbracket B \subseteq A; a \in B \rrbracket \implies a \in A$

**lemma** *fixes*  $P_1$  **defines** *[simp]*:  $P \equiv \text{comp}P_2 \ P_1$

**shows** *Jcc*:

$P_1 \vdash_1 \langle e, (h_0, ls_0) \rangle \Rightarrow \langle ef, (h_1, ls_1) \rangle \implies$   
 $(\bigwedge C \ M \ pc \ v \ xa \ vs \ frs \ I.$   
 $\llbracket P, C, M, pc \triangleright \text{comp}E_2 \ e; P, C, M \triangleright \text{comp}xE_2 \ e \ pc \ (\text{size } vs) / I, \text{size } vs;$   
 $\{pc.. < pc + \text{size}(\text{comp}E_2 \ e)\} \subseteq I \rrbracket \implies$   
 $(ef = \text{Val } v \longrightarrow$   
 $P \vdash (\text{None}, h_0, (vs, ls_0), C, M, pc) \# frs \text{ --jvm--}$   
 $(\text{None}, h_1, (v \# vs, ls_1), C, M, pc + \text{size}(\text{comp}E_2 \ e)) \# frs))$   
 $\wedge$   
 $(ef = \text{Throw } xa \longrightarrow$   
 $(\exists pc_1. pc \leq pc_1 \wedge pc_1 < pc + \text{size}(\text{comp}E_2 \ e) \wedge$   
 $\neg \text{caught } P \ pc_1 \ h_1 \ xa \ (\text{comp}xE_2 \ e \ pc \ (\text{size } vs)) \wedge$   
 $P \vdash (\text{None}, h_0, (vs, ls_0), C, M, pc) \# frs \text{ --jvm--}$   
 $\text{handle } P \ C \ M \ xa \ h_1 \ vs \ ls_1 \ pc_1 \ frs)))$

**and**  $P_1 \vdash_1 \langle es, (h_0, ls_0) \rangle [\Rightarrow] \langle fs, (h_1, ls_1) \rangle \implies$

$(\bigwedge C \ M \ pc \ ws \ xa \ es' \ vs \ frs \ I.$   
 $\llbracket P, C, M, pc \triangleright \text{comp}Es_2 \ es; P, C, M \triangleright \text{comp}xE_2 \ es \ pc \ (\text{size } vs) / I, \text{size } vs;$   
 $\{pc.. < pc + \text{size}(\text{comp}Es_2 \ es)\} \subseteq I \rrbracket \implies$   
 $(fs = \text{map } \text{Val } ws \longrightarrow$

$$\begin{aligned}
& P \vdash (None, h_0, (vs, ls_0, C, M, pc) \# frs) \text{ --jvm--} \\
& \quad (None, h_1, (rev\ ws \ @ \ vs, ls_1, C, M, pc + size(compEs_2\ es)) \# frs)) \\
& \wedge \\
& (fs = map\ Val\ ws \ @ \ Throw\ xa \ \# \ es' \ \longrightarrow \\
& (\exists pc_1. pc \leq pc_1 \wedge pc_1 < pc + size(compEs_2\ es) \wedge \\
& \quad \neg caught\ P\ pc_1\ h_1\ xa\ (compEs_2\ es\ pc\ (size\ vs)) \wedge \\
& \quad P \vdash (None, h_0, (vs, ls_0, C, M, pc) \# frs) \text{ --jvm--} handle\ P\ C\ M\ xa\ h_1\ vs\ ls_1\ pc_1\ frs))
\end{aligned}$$

**lemma** *atLeast0AtMost*[simp]:  $\{0::nat..n\} = \{..n\}$

**by** *auto*

**lemma** *atLeast0LessThan*[simp]:  $\{0::nat..<n\} = \{..<n\}$

**by** *auto*

**fun** *exception* :: 'a exp  $\Rightarrow$  addr option **where**

*exception* (Throw a) = Some a

| *exception* e = None

**lemma** *comp2-correct*:

**assumes** *method*:  $P_1 \vdash C\ sees\ M:Ts \rightarrow T = body\ in\ C$

**and** *eval*:  $P_1 \vdash_1 \langle body, (h, ls) \rangle \Rightarrow \langle e', (h', ls') \rangle$

**shows** *compP2*  $P_1 \vdash (None, h, ([], ls, C, M, 0)) \text{ --jvm--} (exception\ e', h', [])$

**end**

## 5.8 Combining Stages 1 and 2

**theory** *Compiler*

**imports** *Correctness1 Correctness2*

**begin**

**definition** *J2JVM* :: J-prog  $\Rightarrow$  jvm-prog

**where**

$J2JVM \equiv compP_2 \circ compP_1$

**theorem** *comp-correct*:

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

**and** *method*:  $P \vdash C\ sees\ M:Ts \rightarrow T = (pns, body)\ in\ C$

**and** *eval*:  $P \vdash \langle body, (h, [this \# pns \ [\mapsto] \ vs]) \rangle \Rightarrow \langle e', (h', l') \rangle$

**and** *sizes*:  $size\ vs = size\ pns + 1 \quad size\ rest = max-vars\ body$

**shows** *J2JVM*  $P \vdash (None, h, ([], vs @ rest, C, M, 0)) \text{ --jvm--} (exception\ e', h', [])$

**end**

## 5.9 Preservation of Well-Typedness

**theory** *TypeComp*

**imports** *Compiler ../BV/BVSpec*

**begin**

**locale** *TC0* =

**fixes** P :: J<sub>1</sub>-prog **and** mxl :: nat

**begin**

**definition**  $ty\ E\ e = (THE\ T.\ P, E \vdash_1\ e :: T)$

**definition**  $ty_l\ E\ A' = map\ (\lambda i.\ if\ i \in A' \wedge i < size\ E\ then\ OK(E!i)\ else\ Err)\ [0..<max]$

**definition**  $ty_i'\ ST\ E\ A = (case\ A\ of\ None \Rightarrow None\ |\ [A'] \Rightarrow Some(ST,\ ty_l\ E\ A'))$

**definition**  $after\ E\ A\ ST\ e = ty_i'\ (ty\ E\ e\ \#\ ST)\ E\ (A\ \sqcup\ \mathcal{A}\ e)$

**end**

**lemma** (in *TC0*)  $ty\ def2\ [simp]:\ P, E \vdash_1\ e :: T \Longrightarrow ty\ E\ e = T$

**lemma** (in *TC0*)  $[simp]:\ ty_i'\ ST\ E\ None = None$

**lemma** (in *TC0*)  $ty_l\ app\ diff\ [simp]:$

$ty_l\ (E@[T])\ (A - \{size\ E\}) = ty_l\ E\ A$

**lemma** (in *TC0*)  $ty_i'\ app\ diff\ [simp]:$

$ty_i'\ ST\ (E\ @\ [T])\ (A\ \ominus\ size\ E) = ty_i'\ ST\ E\ A$

**lemma** (in *TC0*)  $ty_l\ antimono:$

$A \subseteq A' \Longrightarrow P \vdash ty_l\ E\ A' [\leq_{\top}] ty_l\ E\ A$

**lemma** (in *TC0*)  $ty_i'\ antimono:$

$A \subseteq A' \Longrightarrow P \vdash ty_i'\ ST\ E\ [A'] \leq' ty_i'\ ST\ E\ [A]$

**lemma** (in *TC0*)  $ty_l\ env\ antimono:$

$P \vdash ty_l\ (E@[T])\ A [\leq_{\top}] ty_l\ E\ A$

**lemma** (in *TC0*)  $ty_i'\ env\ antimono:$

$P \vdash ty_i'\ ST\ (E@[T])\ A \leq' ty_i'\ ST\ E\ A$

**lemma** (in *TC0*)  $ty_i'\ incr:$

$P \vdash ty_i'\ ST\ (E\ @\ [T])\ [insert\ (size\ E)\ A] \leq' ty_i'\ ST\ E\ [A]$

**lemma** (in *TC0*)  $ty_l\ incr:$

$P \vdash ty_l\ (E\ @\ [T])\ (insert\ (size\ E)\ A) [\leq_{\top}] ty_l\ E\ A$

**lemma** (in *TC0*)  $ty_l\ in\ types:$

$set\ E \subseteq types\ P \Longrightarrow ty_l\ E\ A \in nlists\ max\ (err\ (types\ P))$

**locale** *TC1* = *TC0*

**begin**

**primrec**  $compT :: ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1 \Rightarrow ty_i'\ list$  **and**

$compTs :: ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1\ list \Rightarrow ty_i'\ list$  **where**

$compT\ E\ A\ ST\ (new\ C) = []$

$| compT\ E\ A\ ST\ (Cast\ C\ e) =$

$compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e]$

$| compT\ E\ A\ ST\ (Val\ v) = []$

$| compT\ E\ A\ ST\ (e_1\ \ll bop \gg\ e_2) =$

$(let\ ST_1 = ty\ E\ e_1\ \# ST; A_1 = A\ \sqcup\ \mathcal{A}\ e_1\ in$

$compT\ E\ A\ ST\ e_1\ @\ [after\ E\ A\ ST\ e_1]\ @$

$compT\ E\ A_1\ ST_1\ e_2\ @\ [after\ E\ A_1\ ST_1\ e_2])$

$| compT\ E\ A\ ST\ (Var\ i) = []$

$\mid \text{compT } E \ A \ ST \ (i := e) = \text{compT } E \ A \ ST \ e \ @$   
 $\quad [\text{after } E \ A \ ST \ e, \text{ty}_i' \ ST \ E \ (A \sqcup \mathcal{A} \ e \sqcup \{\{i\}\})]$   
 $\mid \text{compT } E \ A \ ST \ (e \cdot F\{D\}) =$   
 $\quad \text{compT } E \ A \ ST \ e \ @ \ [\text{after } E \ A \ ST \ e]$   
 $\mid \text{compT } E \ A \ ST \ (e_1 \cdot F\{D\} := e_2) =$   
 $\quad (\text{let } ST_1 = \text{ty } E \ e_1 \# ST; A_1 = A \sqcup \mathcal{A} \ e_1; A_2 = A_1 \sqcup \mathcal{A} \ e_2 \text{ in}$   
 $\quad \text{compT } E \ A \ ST \ e_1 \ @ \ [\text{after } E \ A \ ST \ e_1] \ @$   
 $\quad \text{compT } E \ A_1 \ ST_1 \ e_2 \ @ \ [\text{after } E \ A_1 \ ST_1 \ e_2] \ @$   
 $\quad [\text{ty}_i' \ ST \ E \ A_2])$   
 $\mid \text{compT } E \ A \ ST \ \{i:T; e\} = \text{compT } (E@[T]) \ (A \ominus i) \ ST \ e$   
 $\mid \text{compT } E \ A \ ST \ (e_1;;e_2) =$   
 $\quad (\text{let } A_1 = A \sqcup \mathcal{A} \ e_1 \text{ in}$   
 $\quad \text{compT } E \ A \ ST \ e_1 \ @ \ [\text{after } E \ A \ ST \ e_1, \text{ty}_i' \ ST \ E \ A_1] \ @$   
 $\quad \text{compT } E \ A_1 \ ST \ e_2)$   
 $\mid \text{compT } E \ A \ ST \ (\text{if } (e) \ e_1 \ \text{else } e_2) =$   
 $\quad (\text{let } A_0 = A \sqcup \mathcal{A} \ e; \tau = \text{ty}_i' \ ST \ E \ A_0 \text{ in}$   
 $\quad \text{compT } E \ A \ ST \ e \ @ \ [\text{after } E \ A \ ST \ e, \tau] \ @$   
 $\quad \text{compT } E \ A_0 \ ST \ e_1 \ @ \ [\text{after } E \ A_0 \ ST \ e_1, \tau] \ @$   
 $\quad \text{compT } E \ A_0 \ ST \ e_2)$   
 $\mid \text{compT } E \ A \ ST \ (\text{while } (e) \ c) =$   
 $\quad (\text{let } A_0 = A \sqcup \mathcal{A} \ e; A_1 = A_0 \sqcup \mathcal{A} \ c; \tau = \text{ty}_i' \ ST \ E \ A_0 \text{ in}$   
 $\quad \text{compT } E \ A \ ST \ e \ @ \ [\text{after } E \ A \ ST \ e, \tau] \ @$   
 $\quad \text{compT } E \ A_0 \ ST \ c \ @ \ [\text{after } E \ A_0 \ ST \ c, \text{ty}_i' \ ST \ E \ A_1, \text{ty}_i' \ ST \ E \ A_0])$   
 $\mid \text{compT } E \ A \ ST \ (\text{throw } e) = \text{compT } E \ A \ ST \ e \ @ \ [\text{after } E \ A \ ST \ e]$   
 $\mid \text{compT } E \ A \ ST \ (e \cdot M(es)) =$   
 $\quad \text{compT } E \ A \ ST \ e \ @ \ [\text{after } E \ A \ ST \ e] \ @$   
 $\quad \text{compTs } E \ (A \sqcup \mathcal{A} \ e) \ (\text{ty } E \ e \ # \ ST) \ es$   
 $\mid \text{compT } E \ A \ ST \ (\text{try } e_1 \ \text{catch}(C \ i) \ e_2) =$   
 $\quad \text{compT } E \ A \ ST \ e_1 \ @ \ [\text{after } E \ A \ ST \ e_1] \ @$   
 $\quad [\text{ty}_i' \ (\text{Class } C \ # \ ST) \ E \ A, \text{ty}_i' \ ST \ (E@[Class \ C]) \ (A \sqcup \{\{i\}\})] \ @$   
 $\quad \text{compT } (E@[Class \ C]) \ (A \sqcup \{\{i\}\}) \ ST \ e_2$   
 $\mid \text{compTs } E \ A \ ST \ [] = []$   
 $\mid \text{compTs } E \ A \ ST \ (e \ # \ es) = \text{compT } E \ A \ ST \ e \ @ \ [\text{after } E \ A \ ST \ e] \ @$   
 $\quad \text{compTs } E \ (A \sqcup (\mathcal{A} \ e)) \ (\text{ty } E \ e \ # \ ST) \ es$

**definition**  $\text{compT}_a :: \text{ty list} \Rightarrow \text{nat hyperset} \Rightarrow \text{ty list} \Rightarrow \text{expr}_1 \Rightarrow \text{ty}_i' \ \text{list}$  **where**  
 $\text{compT}_a \ E \ A \ ST \ e = \text{compT } E \ A \ ST \ e \ @ \ [\text{after } E \ A \ ST \ e]$

**end**

**lemma**  $\text{compE}_2\text{-not-Nil}[simp]: \text{compE}_2 \ e \neq []$

**lemma** (**in**  $TC1$ )  $\text{compT-sizes}[simp]:$

**shows**  $\bigwedge E \ A \ ST. \text{size}(\text{compT } E \ A \ ST \ e) = \text{size}(\text{compE}_2 \ e) - 1$

**and**  $\bigwedge E \ A \ ST. \text{size}(\text{compTs } E \ A \ ST \ es) = \text{size}(\text{compEs}_2 \ es)$

**lemma** (**in**  $TC1$ )  $[simp]: \bigwedge ST \ E. [\tau] \notin \text{set}(\text{compT } E \ \text{None} \ ST \ e)$

**and**  $[simp]: \bigwedge ST \ E. [\tau] \notin \text{set}(\text{compTs } E \ \text{None} \ ST \ es)$

**lemma** (**in**  $TC0$ )  $\text{pair-eq-ty}_i'\text{-conv}:$

$([(ST, LT)] = \text{ty}_i' \ ST_0 \ E \ A) =$

$(\text{case } A \ \text{of } \text{None} \Rightarrow \text{False} \mid \text{Some } A \Rightarrow (ST = ST_0 \wedge LT = \text{ty}_l \ E \ A))$

**lemma** (**in**  $TC0$ )  $\text{pair-conv-ty}_i':$

$[(ST, \text{ty}_l \ E \ A)] = \text{ty}_i' \ ST \ E \ [A]$

**lemma** (in *TC1*) *compT-LT-prefix*:

$\bigwedge E A ST_0. \llbracket [(ST,LT)] \in \text{set}(\text{compT } E A ST_0 e); \mathcal{B} e \text{ (size } E) \rrbracket$   
 $\implies P \vdash \llbracket [(ST,LT)] \leq' ty_i' ST E A \rrbracket$

**and**

$\bigwedge E A ST_0. \llbracket [(ST,LT)] \in \text{set}(\text{compTs } E A ST_0 es); \mathcal{B} s \text{ es (size } E) \rrbracket$   
 $\implies P \vdash \llbracket [(ST,LT)] \leq' ty_i' ST E A \rrbracket$

**lemma** [*iff*]: *OK None*  $\in$  *states P mxs mxl*

**lemma** (in *TC0*) *after-in-states*:

**assumes** *wf*: *wf-prog p P* **and** *wt*:  $P, E \vdash_1 e :: T$

**and** *Etypes*:  $\text{set } E \subseteq \text{types } P$  **and** *STtypes*:  $\text{set } ST \subseteq \text{types } P$

**and** *stack*:  $\text{size } ST + \text{max-stack } e \leq \text{mxs}$

**shows** *OK* (after *E A ST e*)  $\in$  *states P mxs mxl*

**lemma** (in *TC0*) *OK-ty\_i'-in-statesI[simp]*:

$\llbracket \text{set } E \subseteq \text{types } P; \text{set } ST \subseteq \text{types } P; \text{size } ST \leq \text{mxs} \rrbracket$   
 $\implies \text{OK } (ty_i' ST E A) \in \text{states } P \text{ mxs mxl}$

**lemma** *is-class-type-aux*: *is-class P C*  $\implies$  *is-type P* (*Class C*)

**theorem** (in *TC1*) *compT-states*:

**assumes** *wf*: *wf-prog p P*

**shows**  $\bigwedge E T A ST.$

$\llbracket P, E \vdash_1 e :: T; \text{set } E \subseteq \text{types } P; \text{set } ST \subseteq \text{types } P;$   
 $\text{size } ST + \text{max-stack } e \leq \text{mxs}; \text{size } E + \text{max-vars } e \leq \text{mxl} \rrbracket$   
 $\implies \text{OK } \text{' set}(\text{compT } E A ST e) \subseteq \text{states } P \text{ mxs mxl}$

**and**  $\bigwedge E Ts A ST.$

$\llbracket P, E \vdash_1 es[::]Ts; \text{set } E \subseteq \text{types } P; \text{set } ST \subseteq \text{types } P;$   
 $\text{size } ST + \text{max-stacks } es \leq \text{mxs}; \text{size } E + \text{max-varss } es \leq \text{mxl} \rrbracket$   
 $\implies \text{OK } \text{' set}(\text{compTs } E A ST es) \subseteq \text{states } P \text{ mxs mxl}$

**definition** *shift* :: *nat*  $\Rightarrow$  *ex-table*  $\Rightarrow$  *ex-table*

**where**

*shift n xt*  $\equiv$  *map* ( $\lambda(\text{from}, \text{to}, C, \text{handler}, \text{depth}). (\text{from} + n, \text{to} + n, C, \text{handler} + n, \text{depth}))$  *xt*

**lemma** [*simp*]: *shift 0 xt* = *xt*

**lemma** [*simp*]: *shift n []* = []

**lemma** [*simp*]: *shift n (xt<sub>1</sub> @ xt<sub>2</sub>)* = *shift n xt<sub>1</sub> @ shift n xt<sub>2</sub>*

**lemma** [*simp*]: *shift m (shift n xt)* = *shift (m+n) xt*

**lemma** [*simp*]: *pcs (shift n xt)* = {*pc+n* | *pc. pc*  $\in$  *pcs xt*}

**lemma** *shift-compxE<sub>2</sub>*:

**shows**  $\bigwedge pc pc' d. \text{shift } pc (\text{compxE}_2 e pc' d) = \text{compxE}_2 e (pc' + pc) d$

**and**  $\bigwedge pc pc' d. \text{shift } pc (\text{compxEs}_2 es pc' d) = \text{compxEs}_2 es (pc' + pc) d$

**lemma** *compxE<sub>2</sub>-size-convs[simp]*:

**shows**  $n \neq 0 \implies \text{compxE}_2 e n d = \text{shift } n (\text{compxE}_2 e 0 d)$

**and**  $n \neq 0 \implies \text{compxEs}_2 es n d = \text{shift } n (\text{compxEs}_2 es 0 d)$

**locale** *TC2* = *TC1* +

**fixes** *T<sub>r</sub>* :: *ty* **and** *mxs* :: *pc*

**begin**

**definition**



$wt\text{-instrs} :: instr\ list \Rightarrow ex\text{-table} \Rightarrow ty_i' \text{ list} \Rightarrow bool$   
 $((\vdash -, - / [::] / -) [0,0,51] 50) \text{ where}$   
 $\vdash is, xt [::] \tau s \iff size\ is < size\ \tau s \wedge pcs\ xt \subseteq \{0..<size\ is\} \wedge$   
 $(\forall pc < size\ is. P, T, m, mpc, size\ \tau s, xt \vdash is!pc, pc :: \tau s)$

**end**

**notation**  $TC2.wt\text{-instrs} ((-, -, - \vdash / -, - / [::] / -) [50,50,50,50,50,51] 50)$

**lemma** (in  $TC2$ )  $[simp]: \tau s \neq [] \implies \vdash [], [] [::] \tau s$

**lemma**  $[simp]: eff\ i\ P\ pc\ et\ None = []$

**lemma**  $wt\text{-instr}\text{-appR}$ :

$\llbracket P, T, m, mpc, xt \vdash is!pc, pc :: \tau s;$   
 $pc < size\ is; size\ is < size\ \tau s; mpc \leq size\ \tau s; mpc \leq mpc' \rrbracket$   
 $\implies P, T, m, mpc', xt \vdash is!pc, pc :: \tau s @ \tau s'$

**lemma**  $relevant\text{-entries}\text{-shift} [simp]:$

$relevant\text{-entries}\ P\ i\ (pc+n)\ (shift\ n\ xt) = shift\ n\ (relevant\text{-entries}\ P\ i\ pc\ xt)$

**lemma**  $[simp]:$

$xcpt\text{-eff}\ i\ P\ (pc+n)\ \tau\ (shift\ n\ xt) =$   
 $map\ (\lambda(pc, \tau). (pc + n, \tau))\ (xcpt\text{-eff}\ i\ P\ pc\ \tau\ xt)$

**lemma**  $[simp]:$

$app_i\ (i, P, pc, m, T, \tau) \implies$   
 $eff\ i\ P\ (pc+n)\ (shift\ n\ xt)\ (Some\ \tau) =$   
 $map\ (\lambda(pc, \tau). (pc+n, \tau))\ (eff\ i\ P\ pc\ xt\ (Some\ \tau))$

**lemma**  $[simp]:$

$xcpt\text{-app}\ i\ P\ (pc+n)\ mpc\ (shift\ n\ xt)\ \tau = xcpt\text{-app}\ i\ P\ pc\ mpc\ xt\ \tau$

**lemma**  $wt\text{-instr}\text{-appL}$ :

**assumes**  $P, T, m, mpc, xt \vdash i, pc :: \tau s$  **and**  $pc < size\ \tau s$  **and**  $mpc \leq size\ \tau s$   
**shows**  $P, T, m, mpc + size\ \tau s', shift\ (size\ \tau s')\ xt \vdash i, pc + size\ \tau s' :: \tau s' @ \tau s$

**lemma**  $wt\text{-instr}\text{-Cons}$ :

**assumes**  $uti: P, T, m, mpc - 1, [] \vdash i, pc - 1 :: \tau s$   
**and**  $pcl: 0 < pc$  **and**  $mpcl: 0 < mpc$   
**and**  $pcu: pc < size\ \tau s + 1$  **and**  $mpcu: mpc \leq size\ \tau s + 1$   
**shows**  $P, T, m, mpc, [] \vdash i, pc :: \tau \# \tau s$

**lemma**  $wt\text{-instr}\text{-append}$ :

**assumes**  $uti: P, T, m, mpc - size\ \tau s', [] \vdash i, pc - size\ \tau s' :: \tau s$   
**and**  $pcl: size\ \tau s' \leq pc$  **and**  $mpcl: size\ \tau s' \leq mpc$   
**and**  $pcu: pc < size\ \tau s + size\ \tau s'$  **and**  $mpcu: mpc \leq size\ \tau s + size\ \tau s'$   
**shows**  $P, T, m, mpc, [] \vdash i, pc :: \tau s' @ \tau s$

**lemma**  $xcpt\text{-app}\text{-pcs}$ :

$pc \notin pcs\ xt \implies xcpt\text{-app}\ i\ P\ pc\ mpc\ xt\ \tau$

**lemma**  $xcpt\text{-eff}\text{-pcs}$ :

$pc \notin pcs\ xt \implies xcpt\text{-eff}\ i\ P\ pc\ \tau\ xt = []$

**lemma**  $pcs\text{-shift}$ :

$pc < n \implies pc \notin pcs \text{ (shift } n \text{ } xt)$

**lemma** *wt-instr-appRx*:

$\llbracket P, T, m, mpc, xt \vdash is!pc, pc :: \tau s; pc < size \ is; size \ is < size \ \tau s; mpc \leq size \ \tau s \rrbracket$   
 $\implies P, T, m, mpc, xt @ shift \ (size \ is) \ xt' \vdash is!pc, pc :: \tau s$

**lemma** *wt-instr-appLx*:

$\llbracket P, T, m, mpc, xt \vdash i, pc :: \tau s; pc \notin pcs \ xt' \rrbracket$   
 $\implies P, T, m, mpc, xt' @ xt \vdash i, pc :: \tau s$

**lemma** (in *TC2*) *wt-instrs-extR*:

$\vdash is, xt \ [::] \ \tau s \implies \vdash is, xt \ [::] \ \tau s @ \ \tau s'$

**lemma** (in *TC2*) *wt-instrs-ext*:

**assumes**  $wt_1: \vdash is_1, xt_1 \ [::] \ \tau s_1 @ \ \tau s_2$  **and**  $wt_2: \vdash is_2, xt_2 \ [::] \ \tau s_2$   
**and**  $\tau s\text{-size}: size \ \tau s_1 = size \ is_1$

**shows**  $\vdash is_1 @ is_2, xt_1 @ shift \ (size \ is_1) \ xt_2 \ [::] \ \tau s_1 @ \ \tau s_2$

**corollary** (in *TC2*) *wt-instrs-ext2*:

$\llbracket \vdash is_2, xt_2 \ [::] \ \tau s_2; \vdash is_1, xt_1 \ [::] \ \tau s_1 @ \ \tau s_2; size \ \tau s_1 = size \ is_1 \rrbracket$   
 $\implies \vdash is_1 @ is_2, xt_1 @ shift \ (size \ is_1) \ xt_2 \ [::] \ \tau s_1 @ \ \tau s_2$

**corollary** (in *TC2*) *wt-instrs-ext-prefix* [*trans*]:

$\llbracket \vdash is_1, xt_1 \ [::] \ \tau s_1 @ \ \tau s_2; \vdash is_2, xt_2 \ [::] \ \tau s_3;$   
 $size \ \tau s_1 = size \ is_1; prefix \ \tau s_3 \ \tau s_2 \rrbracket$   
 $\implies \vdash is_1 @ is_2, xt_1 @ shift \ (size \ is_1) \ xt_2 \ [::] \ \tau s_1 @ \ \tau s_2$

**corollary** (in *TC2*) *wt-instrs-app*:

**assumes**  $is_1: \vdash is_1, xt_1 \ [::] \ \tau s_1 @ [\tau]$   
**assumes**  $is_2: \vdash is_2, xt_2 \ [::] \ \tau \# \ \tau s_2$   
**assumes**  $s: size \ \tau s_1 = size \ is_1$   
**shows**  $\vdash is_1 @ is_2, xt_1 @ shift \ (size \ is_1) \ xt_2 \ [::] \ \tau s_1 @ \ \tau \# \ \tau s_2$

**corollary** (in *TC2*) *wt-instrs-app-last* [*trans*]:

**assumes**  $\vdash is_2, xt_2 \ [::] \ \tau \# \ \tau s_2 \vdash is_1, xt_1 \ [::] \ \tau s_1$   
 $last \ \tau s_1 = \tau \ size \ \tau s_1 = size \ is_1 + 1$   
**shows**  $\vdash is_1 @ is_2, xt_1 @ shift \ (size \ is_1) \ xt_2 \ [::] \ \tau s_1 @ \ \tau s_2$

**corollary** (in *TC2*) *wt-instrs-append-last* [*trans*]:

**assumes**  $wtis: \vdash is, xt \ [::] \ \tau s$  **and**  $wti: P, T_r, max, mpc, [] \vdash i, pc :: \tau s$   
**and**  $pc: pc = size \ is$  **and**  $mpc: mpc = size \ \tau s$  **and**  $is\text{-}\tau s: size \ is + 1 < size \ \tau s$   
**shows**  $\vdash is @ [i], xt \ [::] \ \tau s$

**corollary** (in *TC2*) *wt-instrs-app2*:

$\llbracket \vdash is_2, xt_2 \ [::] \ \tau' \# \ \tau s_2; \vdash is_1, xt_1 \ [::] \ \tau \# \ \tau s_1 @ [\tau'];$   
 $xt' = xt_1 @ shift \ (size \ is_1) \ xt_2; size \ \tau s_1 + 1 = size \ is_1 \rrbracket$   
 $\implies \vdash is_1 @ is_2, xt' \ [::] \ \tau \# \ \tau s_1 @ \ \tau' \# \ \tau s_2$

**corollary** (in *TC2*) *wt-instrs-app2-simp* [*trans, simp*]:

$\llbracket \vdash is_2, xt_2 \ [::] \ \tau' \# \ \tau s_2; \vdash is_1, xt_1 \ [::] \ \tau \# \ \tau s_1 @ [\tau']; size \ \tau s_1 + 1 = size \ is_1 \rrbracket$   
 $\implies \vdash is_1 @ is_2, xt_1 @ shift \ (size \ is_1) \ xt_2 \ [::] \ \tau \# \ \tau s_1 @ \ \tau' \# \ \tau s_2$

**corollary** (in *TC2*) *wt-instrs-Cons* [*simp*]:

$\llbracket \tau s \neq []; \vdash [i], [] \ [::] \ [\tau, \tau']; \vdash is, xt \ [::] \ \tau' \# \ \tau s \rrbracket$   
 $\implies \vdash i \# is, shift \ 1 \ xt \ [::] \ \tau \# \ \tau' \# \ \tau s$

```
theory 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.