

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

Gerwin Klein

Tobias Nipkow

March 12, 2013



# Contents

<b>1</b>	<b>Preface</b>	<b>5</b>
1.1	Theory Dependencies . . . . .	5
<b>2</b>	<b>Jinja Source Language</b>	<b>9</b>
2.1	Auxiliary Definitions . . . . .	10
2.2	Jinja types . . . . .	12
2.3	Class Declarations and Programs . . . . .	13
2.4	Relations between Jinja Types . . . . .	14
2.5	Jinja Values . . . . .	20
2.6	Objects and the Heap . . . . .	21
2.7	Exceptions . . . . .	23
2.8	Expressions . . . . .	25
2.9	Program State . . . . .	27
2.10	Big Step Semantics . . . . .	28
2.11	Small Step Semantics . . . . .	33
2.12	System Classes . . . . .	38
2.13	Generic Well-formedness of programs . . . . .	39
2.14	Weak well-formedness of Jinja programs . . . . .	42
2.15	Equivalence of Big Step and Small Step Semantics . . . . .	43
2.16	Well-typedness of Jinja expressions . . . . .	49
2.17	Runtime Well-typedness . . . . .	51
2.18	Definite assignment . . . . .	54
2.19	Conformance Relations for Type Soundness Proofs . . . . .	56
2.20	Progress of Small Step Semantics . . . . .	58
2.21	Well-formedness Constraints . . . . .	60
2.22	Type Safety Proof . . . . .	61
2.23	Program annotation . . . . .	64
2.24	Example Expressions . . . . .	65
2.25	Code Generation For BigStep . . . . .	68
2.26	Code Generation For WellType . . . . .	71
<b>3</b>	<b>Jinja Virtual Machine</b>	<b>73</b>
3.1	State of the JVM . . . . .	74
3.2	Instructions of the JVM . . . . .	75
3.3	JVM Instruction Semantics . . . . .	76
3.4	Exception handling in the JVM . . . . .	79
3.5	Program Execution in the JVM . . . . .	80

3.6	A Defensive JVM . . . . .	82
3.7	Example for generating executable code from JVM semantics . . . . .	86
<b>4</b>	<b>Bytecode Verifier</b>	<b>89</b>
4.1	Semilattices . . . . .	90
4.2	The Error Type . . . . .	94
4.3	More about Options . . . . .	97
4.4	Products as Semilattices . . . . .	98
4.5	Fixed Length Lists . . . . .	99
4.6	Typing and Dataflow Analysis Framework . . . . .	103
4.7	More on Semilattices . . . . .	104
4.8	Lifting the Typing Framework to <code>err</code> , <code>app</code> , and <code>eff</code> . . . . .	106
4.9	Kildall's Algorithm . . . . .	108
4.10	The Lightweight Bytecode Verifier . . . . .	111
4.11	Correctness of the LBV . . . . .	115
4.12	Completeness of the LBV . . . . .	116
4.13	The Jinja Type System as a Semilattice . . . . .	118
4.14	The JVM Type System as Semilattice . . . . .	120
4.15	Effect of Instructions on the State Type . . . . .	123
4.16	Monotonicity of <code>eff</code> and <code>app</code> . . . . .	130
4.17	The Bytecode Verifier . . . . .	131
4.18	The Typing Framework for the JVM . . . . .	133
4.19	Kildall for the JVM . . . . .	135
4.20	LBV for the JVM . . . . .	136
4.21	BV Type Safety Invariant . . . . .	138
4.22	BV Type Safety Proof . . . . .	141
4.23	Welltyped Programs produce no Type Errors . . . . .	147
4.24	Example Welltypings . . . . .	149
<b>5</b>	<b>Compilation</b>	<b>155</b>
5.1	An Intermediate Language . . . . .	156
5.2	Well-Formedness of Intermediate Language . . . . .	160
5.3	Program Compilation . . . . .	163
5.4	Compilation Stage 1 . . . . .	169
5.5	Correctness of Stage 1 . . . . .	170
5.6	Compilation Stage 2 . . . . .	172
5.7	Correctness of Stage 2 . . . . .	175
5.8	Combining Stages 1 and 2 . . . . .	179
5.9	Preservation of Well-Typedness . . . . .	180

# 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.

A tabled implementation of the reflexive transitive closure **theory** *Transitive-Closure-Table*

**imports** *Main*

**begin**

**inductive** *rtrancl-path* :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a  $\Rightarrow$  'a list  $\Rightarrow$  'a  $\Rightarrow$  bool  
**for** *r* :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool

**where**

*base*: *rtrancl-path* *r* *x* [] *x*

| *step*: *r* *x* *y*  $\Longrightarrow$  *rtrancl-path* *r* *y* *ys* *z*  $\Longrightarrow$  *rtrancl-path* *r* *x* (*y* # *ys*) *z*

**lemma** *rtranclp-eq-rtrancl-path*:  $r^{**}$  *x* *y* = ( $\exists$  *xs*. *rtrancl-path* *r* *x* *xs* *y*)  
 $\langle$ *proof* $\rangle$

**lemma** *rtrancl-path-trans*:

**assumes** *xy*: *rtrancl-path* *r* *x* *xs* *y*

**and** *yz*: *rtrancl-path* *r* *y* *ys* *z*

**shows** *rtrancl-path* *r* *x* (*xs* @ *ys*) *z*  $\langle$ *proof* $\rangle$

**lemma** *rtrancl-path-appendE*:

**assumes** *xz*: *rtrancl-path* *r* *x* (*xs* @ *y* # *ys*) *z*

**obtains** *rtrancl-path* *r* *x* (*xs* @ [*y*]) *y* **and** *rtrancl-path* *r* *y* *ys* *z*  $\langle$ *proof* $\rangle$

**lemma** *rtrancl-path-distinct*:

**assumes** *xy*: *rtrancl-path* *r* *x* *xs* *y*

**obtains** *xs'* **where** *rtrancl-path* *r* *x* *xs'* *y* **and** *distinct* (*x* # *xs'*)  $\langle$ *proof* $\rangle$

**inductive** *rtrancl-tab* :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  bool

**for** *r* :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool

**where**

*base*: *rtrancl-tab* *r* *xs* *x* *x*

| *step*: *x*  $\notin$  *set xs*  $\Longrightarrow$  *r* *x* *y*  $\Longrightarrow$  *rtrancl-tab* *r* (*x* # *xs*) *y* *z*  $\Longrightarrow$  *rtrancl-tab* *r* *xs* *x* *z*

**lemma** *rtrancl-path-imp-rtrancl-tab*:

**assumes** *path*: *rtrancl-path* *r* *x* *xs* *y*

**and** *x*: *distinct* (*x* # *xs*)

**and** *ys*: ( $\{x\} \cup \text{set } xs$ )  $\cap$  *set ys* =  $\{x\}$

**shows** *rtrancl-tab* *r* *ys* *x* *y*  $\langle$ *proof* $\rangle$

**lemma** *rtrancl-tab-imp-rtrancl-path*:

**assumes** *tab*: *rtrancl-tab* *r* *ys* *x* *y*

**obtains** *xs* **where** *rtrancl-path* *r* *x* *xs* *y*  $\langle$ *proof* $\rangle$

**lemma** *rtranclp-eq-rtrancl-tab-nil*:  $r^{**}$  *x* *y* = *rtrancl-tab* *r* [] *x* *y*  
 $\langle$ *proof* $\rangle$

**declare** *rtranclp-rtrancl-eq*[*code del*]

**declare** *rtranclp-eq-rtrancl-tab-nil*[*THEN iffD2, code-pred-intro*]

**code-pred** *rtranclp*  $\langle$ *proof* $\rangle$

### 1.1.1 A simple example

**datatype** *ty* = *A* | *B* | *C*

```
inductive test :: ty => ty => bool
where
  test A B
| test B A
| test B C
```

**Invoking with the predicate compiler and the generic code generator**

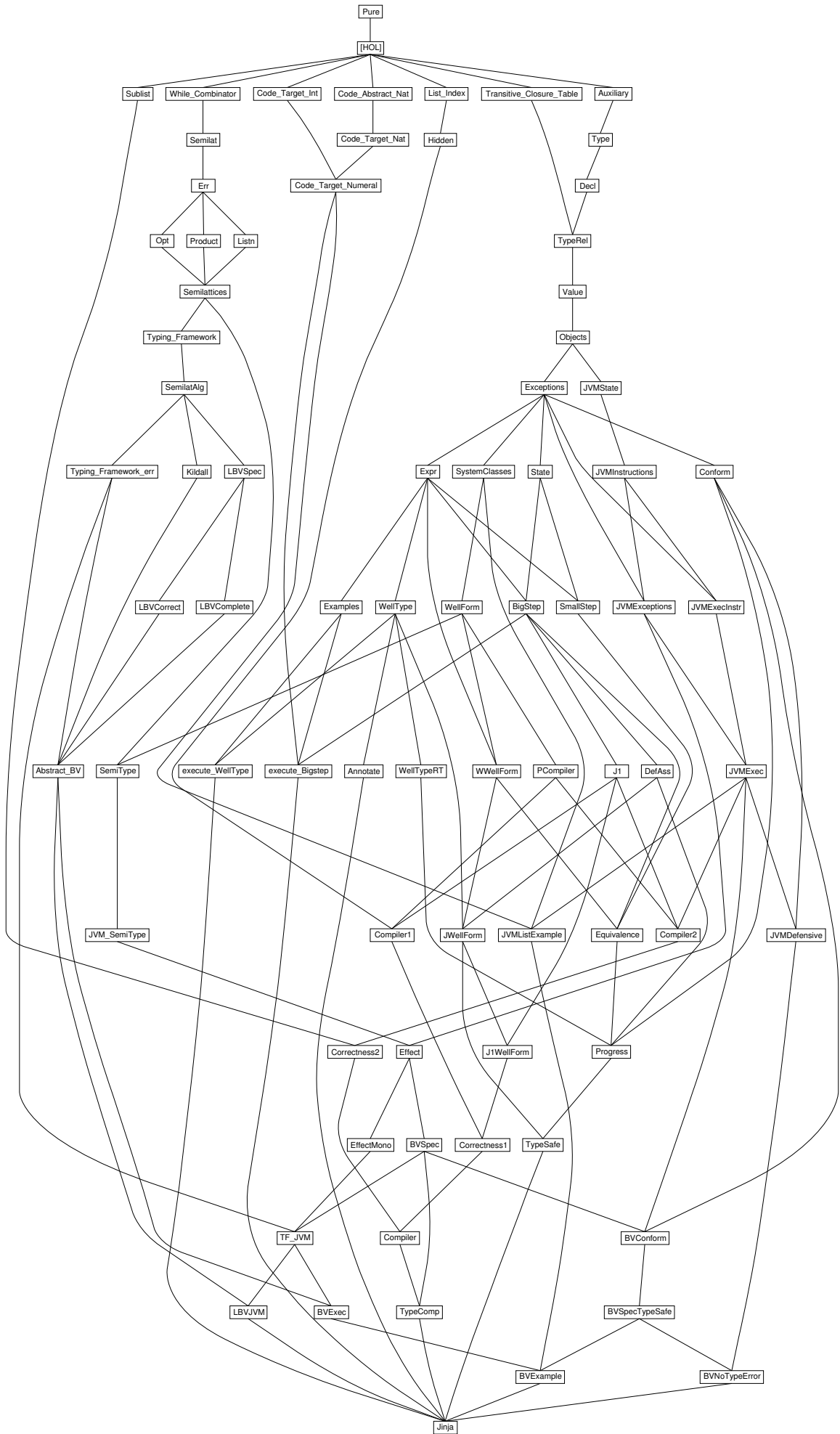
```
code-pred test <proof>
```

```
values {x. test** A C}
values {x. test** C A}
values {x. test** A x}
values {x. test** x C}
```

```
value test** A C
value test** C A
```

```
hide-type ty
hide-const test A B C
```

```
end
```





## Chapter 2

# Jinja Source Language

## 2.1 Auxiliary Definitions

**theory** *Auxiliary* **imports** *Main* **begin**

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

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

*<proof>*

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

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

*<proof>*

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

### 2.1.1 *distinct-fst*

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

**where**

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

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

$$\text{distinct-fst } []$$

*<proof>*

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

$$\text{distinct-fst } ((k,x)\#kxs) = (\text{distinct-fst } kxs \wedge (\forall y. (k,y) \notin \text{set } kxs))$$

*<proof>*

**lemma** *map-of-SomeI*:

$$[\text{distinct-fst } kxs; (k,x) \in \text{set } kxs] \Longrightarrow \text{map-of } kxs\ k = \text{Some } x$$

*<proof>*

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

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

**where**

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

Convenience lemmas

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

$$\text{list-all2 } (\text{fun-of } S) (x\#xs) (y\#ys) = \\ ((x,y) \in S \wedge \text{list-all2 } (\text{fun-of } S) xs ys)$$

*<proof>*

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

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

*<proof>*

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

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

*<proof>*

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

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

*<proof>*

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

$$[\bigwedge x y. [(x,y) \in S; (y,x) \in T] \Longrightarrow x = y]; \\ \text{list-all2 } (\text{fun-of } S) xs ys; \text{list-all2 } (\text{fun-of } T) ys xs] \Longrightarrow xs = ys$$

*<proof>*

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

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

*<proof>*

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

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

*<proof>*

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

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

*<proof>*

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

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

*<proof>*

**end**

## 2.2 Jinja types

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

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

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

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

**definition** *Object* :: *cname*

**where**

*Object*  $\equiv$  "Object"

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

**where**

*this*  $\equiv$  "this"

— types

**datatype** *ty*

= *Void* — type of statements

| *Boolean*

| *Integer*

| *NT* — null type

| *Class cname* — class type

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

**where**

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

**lemma** [*iff*]: *is-refT* *NT*  $\langle$ *proof* $\rangle$

**lemma** [*iff*]: *is-refT* (*Class* *C*)  $\langle$ *proof* $\rangle$

**lemma** *refTE*:

$\llbracket \text{is-refT } T; T = NT \implies P; \wedge C. T = \text{Class } C \implies P \rrbracket \implies P \langle \text{proof} \rangle$

**lemma** *not-refTE*:

$\llbracket \neg \text{is-refT } T; T = \text{Void} \vee T = \text{Boolean} \vee T = \text{Integer} \implies P \rrbracket \implies P \langle \text{proof} \rangle$

**end**

## 2.3 Class Declarations and Programs

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

**type-synonym**

$fdecl = vname \times ty$  — field declaration

**type-synonym**

$'m\ mdecl = mname \times ty\ list \times ty \times 'm$  — method = name, arg. types, return type, body

**type-synonym**

$'m\ class = cname \times fdecl\ list \times 'm\ mdecl\ list$  — class = superclass, fields, methods

**type-synonym**

$'m\ cdecl = cname \times 'm\ class$  — class declaration

**type-synonym**

$'m\ prog = 'm\ cdecl\ list$  — program

**definition**  $class :: 'm\ prog \Rightarrow cname \rightarrow 'm\ class$

**where**

$class \equiv map\ of$

**definition**  $is\ class :: 'm\ prog \Rightarrow cname \Rightarrow bool$

**where**

$is\ class\ P\ C \equiv class\ P\ C \neq None$

**lemma**  $finite\ is\ class: finite\ \{C. is\ class\ P\ C\}$

*<proof>*

**definition**  $is\ type :: 'm\ prog \Rightarrow ty \Rightarrow bool$

**where**

$is\ type\ P\ T \equiv$

$(case\ T\ of\ Void \Rightarrow True \mid Boolean \Rightarrow True \mid Integer \Rightarrow True \mid NT \Rightarrow True$   
 $\mid Class\ C \Rightarrow is\ class\ P\ C)$

**lemma**  $is\ type\ simps\ [simp]:$

$is\ type\ P\ Void \wedge is\ type\ P\ Boolean \wedge is\ type\ P\ Integer \wedge$

$is\ type\ P\ NT \wedge is\ type\ P\ (Class\ C) = is\ class\ P\ C$  *<proof>*

**abbreviation**

$types\ P == Collect\ (is\ type\ P)$

**end**

## 2.4 Relations between Jinja Types

**theory** *TypeRel* **imports**  
 $\sim$ /src/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**

$P \vdash C \prec^1 D \equiv (C, D) \in \text{subcls1 } P$   
 $\mid \text{subcls1I: } \llbracket \text{class } P \ C = \text{Some } (D, \text{rest}); C \neq \text{Object} \rrbracket \Longrightarrow P \vdash C \prec^1 D$

**abbreviation**

*subcls* :: 'm prog  $\Rightarrow$  [cname, cname]  $\Rightarrow$  bool (-  $\vdash$  -  $\preceq^*$  - [71,71,71] 70)  
**where**  $P \vdash C \preceq^* D \equiv (C, D) \in (\text{subcls1 } P)^*$

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

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

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

**lemma** *subcls1-def2*:

*subcls1* *P* =  
 $(\text{SIGMA } C: \{C. \text{is-class } P \ C\}. \{D. C \neq \text{Object} \wedge \text{fst } (\text{the } (\text{class } P \ C)) = D\}) \langle \text{proof} \rangle$

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

### 2.4.2 The subtype relations

**inductive**

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

**where**

*widen-refl*[iff]:  $P \vdash T \leq T$   
 $\mid \text{widen-subcls: } P \vdash C \preceq^* D \Longrightarrow P \vdash \text{Class } C \leq \text{Class } D$   
 $\mid \text{widen-null}$ [iff]:  $P \vdash \text{NT} \leq \text{Class } C$

**abbreviation** (*xsymbols*)

*widens* :: 'm prog  $\Rightarrow$  ty list  $\Rightarrow$  ty list  $\Rightarrow$  bool  
 (-  $\vdash$  - [ $\leq$ ] - [71,71,71] 70) **where**  
 $\text{widens } P \ Ts \ Ts' \equiv \text{list-all2 } (\text{widen } P) \ Ts \ Ts'$

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

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

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

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

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

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

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

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

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

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

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

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

### 2.4.3 Method lookup

**inductive**

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

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

**where**

*sees-methods-Object*:

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

| *sees-methods-rec*:

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

**lemma** *sees-methods-fun*:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**where**

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

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

**where**

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

**lemma** *sees-method-fun*:

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

**lemma** *visible-method-exists*:

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

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

**lemma** *sees-method-idemp*:

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

$\langle \text{proof} \rangle$

**lemma** *sees-method-decl-mono*:

$\llbracket P \vdash C' \preceq^* C; P \vdash C \text{ sees } M:Ts \rightarrow T = m \text{ in } D;$

$P \vdash C' \text{ sees } M:Ts' \rightarrow T' = m' \text{ in } D' \rrbracket \implies P \vdash D' \preceq^* D$

$\langle \text{proof} \rangle$

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

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

## 2.4.4 Field lookup

**inductive**

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

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

**where**

*has-fields-rec*:

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

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

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

| *has-fields-Object*:

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

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

**lemma** *has-fields-fun*:

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

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

$\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

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

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

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



**shows**  $P \vdash C$  has-fields FDTs

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

**definition** *has-field* :: 'm prog  $\Rightarrow$  cname  $\Rightarrow$  vname  $\Rightarrow$  ty  $\Rightarrow$  cname  $\Rightarrow$  bool  
 (-  $\vdash$  - has :- 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*:

$$\llbracket P \vdash C \text{ has } F:T \text{ in } D; P \vdash C' \preceq^* C \rrbracket \implies P \vdash C' \text{ has } F:T \text{ in } D \langle \text{proof} \rangle$$

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

**where**

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

**lemma** *map-of-remap-SomeD*:

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

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

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

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

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

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

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

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

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

## 2.4.5 Functional lookup

**definition** *method* :: 'm prog  $\Rightarrow$  cname  $\Rightarrow$  mname  $\Rightarrow$  cname  $\times$  ty list  $\times$  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 prog  $\Rightarrow$  cname  $\Rightarrow$  vname  $\Rightarrow$  cname  $\times$  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 prog  $\Rightarrow$  cname  $\Rightarrow$  ((vname  $\times$  cname)  $\times$  ty) list

**where**

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

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

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

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

## 2.4.6 Code generator setup

### code-pred

```
(modes: i ⇒ i ⇒ i ⇒ bool, i ⇒ i ⇒ o ⇒ bool)
subcls1p
⟨proof⟩
declare subcls1-def [code-pred-def]
```

### code-pred

```
(modes: i ⇒ i × o ⇒ bool, i ⇒ i × i ⇒ bool)
[inductify]
subcls1
⟨proof⟩
definition subcls' where subcls' G = (subcls1p G) ^**
```

### code-pred

```
(modes: i ⇒ i ⇒ i ⇒ bool, i ⇒ i ⇒ o ⇒ bool)
[inductify]
subcls'
⟨proof⟩
```

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

```
(subcls1 G) ^* = {(C, D). subcls' G C D}
⟨proof⟩
```

### code-pred

```
(modes: i ⇒ i ⇒ i ⇒ bool)
widen
⟨proof⟩
```

### code-pred

```
(modes: i ⇒ i ⇒ o ⇒ bool)
Fields
⟨proof⟩
```

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

```
[[ P ⊢ C has-fields FDTs; map-of FDTs (F, D) = [T] ]]
⇒ P ⊢ C has F:T in D
⟨proof⟩
```

### code-pred

```
(modes: i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ bool, i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool)
has-field
⟨proof⟩
```

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

```
[[ P ⊢ C has-fields FDTs; map-of (map (λ((F, D), T). (F, D, T)) FDTs) F = [(D, T)] ]]
⇒ P ⊢ C sees F:T in D
⟨proof⟩
```

### code-pred

```
(modes: i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ bool, i ⇒ i ⇒ i ⇒ o ⇒ i ⇒ bool,
i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool, i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ bool)
sees-field
⟨proof⟩
```

**code-pred**

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

Methods

$\langle \text{proof} \rangle$

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

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

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

$\langle \text{proof} \rangle$

**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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

$\text{method } P \ C \ M =$

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

$\langle \text{proof} \rangle$

**lemma** *eval-Fields-conv*:

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

## 2.5 Jinja Values

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

**type-synonym** *addr* = *nat*

**datatype** *val*

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

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

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

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

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

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

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

**end**

## 2.6 Objects and the Heap

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

### 2.6.1 Objects

**type-synonym**

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

**type-synonym**

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

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

**where**

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

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

**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*]:  $obj\text{-}ty\ (C, fs) = Class\ C\ \langle proof \rangle$

### 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**

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

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

**where**

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

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

**where**

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

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

**where**

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

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

```

    new-Addr h = Some a  $\implies$  h a = None
  <proof>
lemma [simp]: (typeofh v = Some Boolean) = ( $\exists$  b. v = Bool b)
  <proof>
lemma [simp]: (typeofh v = Some Integer) = ( $\exists$  i. v = Intg i) <proof>
lemma [simp]: (typeofh v = Some NT) = (v = Null)
  <proof>
lemma [simp]: (typeofh v = Some(Class C)) = ( $\exists$  a fs. v = Addr a  $\wedge$  h a = Some(C,fs))
  <proof>
lemma [simp]: h a = Some(C,fs)  $\implies$  typeof(h(a $\mapsto$ (C,fs'))) v = typeofh v
  <proof>

```

For literal values the first parameter of *typeof* can be set to *Map.empty* because they do not contain addresses:

#### abbreviation

```

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

```

```

lemma typeof-lit-typeof:
  typeof v = Some T  $\implies$  typeofh v = Some T
  <proof>
lemma typeof-lit-is-type:
  typeof v = Some T  $\implies$  is-type P T
  <proof>

```

### 2.6.3 Heap extension $\triangleleft$

```

lemma hextI:  $\forall$  a C fs. h a = Some(C,fs)  $\longrightarrow$  ( $\exists$  fs'. h' a = Some(C,fs'))  $\implies$  h  $\triangleleft$  h' <proof>
lemma hext-objD:  $\llbracket$  h  $\triangleleft$  h'; h a = Some(C,fs)  $\rrbracket \implies \exists$  fs'. h' a = Some(C,fs') <proof>
lemma hext-refl [iff]: h  $\triangleleft$  h <proof>
lemma hext-new [simp]: h a = None  $\implies$  h  $\triangleleft$  h(a $\mapsto$ x) <proof>
lemma hext-trans:  $\llbracket$  h  $\triangleleft$  h'; h'  $\triangleleft$  h''  $\rrbracket \implies$  h  $\triangleleft$  h'' <proof>
lemma hext-upd-obj: h a = Some (C,fs)  $\implies$  h  $\triangleleft$  h(a $\mapsto$ (C,fs')) <proof>
lemma hext-typeof-mono:  $\llbracket$  h  $\triangleleft$  h'; typeofh v = Some T  $\rrbracket \implies$  typeofh' v = Some T <proof>

```

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 = None then Some(LEAST a. a  $\geq$  n  $\wedge$  h a = None)
  else None

```

```

lemma new-Addr-code-code [code]:
  new-Addr h = gen-new-Addr h 0
  <proof>

```

```

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))
  <proof>

```

**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*  $\equiv$  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* (proof)

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

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

### 2.7.2 preallocated

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

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

**lemma** *preallocatedD*:

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

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

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

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

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

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

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

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

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

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

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

**lemma** *preallocated-hext*:

$\llbracket \text{preallocated } h; h \leq h' \rrbracket \implies \text{preallocated } h' \langle \text{proof} \rangle$

**lemma** *preallocated-start*:

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

**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)* ( $- \ll - \gg$  - [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*  $\times$  *expr* — Jinja method body: parameter names and expression

**type-synonym**

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

The semantics of binary operators:

**fun** *binop* :: *bop*  $\times$  *val*  $\times$  *val*  $\Rightarrow$  *val option* **where**

*binop*(*Eq*, *v*<sub>1</sub>, *v*<sub>2</sub>) = *Some*(*Bool* (*v*<sub>1</sub> = *v*<sub>2</sub>))

| *binop*(*Add*, *Intg* *i*<sub>1</sub>, *Intg* *i*<sub>2</sub>) = *Some*(*Intg*(*i*<sub>1</sub>+*i*<sub>2</sub>))

| *binop*(*bop*, *v*<sub>1</sub>, *v*<sub>2</sub>) = *None*

**lemma** [*simp*]:

(*binop*(*Add*, *v*<sub>1</sub>, *v*<sub>2</sub>) = *Some v*) = ( $\exists i_1 i_2. v_1 = \text{Intg } i_1 \wedge v_2 = \text{Intg } i_2 \wedge v = \text{Intg}(i_1+i_2)$ ) $\langle$ *proof* $\rangle$

### 2.8.1 Syntactic sugar

**abbreviation** (*input*)

*InitBlock*:: *'a*  $\Rightarrow$  *ty*  $\Rightarrow$  *'a exp*  $\Rightarrow$  *'a exp*  $\Rightarrow$  *'a exp* ( $((1'\{-:- := -;/ -\}))$ ) **where**

*InitBlock* *V T e1 e2* ==  $\{V:T; V := e1;; e2\}$

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

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

**abbreviation** *addr a* == *Val(Addr a)*

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

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

**abbreviation**

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

**abbreviation**

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

## 2.8.2 Free Variables

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

*fv*(*new* *C*) = {}  
| *fv*(*Cast* *C* *e*) = *fv* *e*  
| *fv*(*Val* *v*) = {}  
| *fv*(*e*<sub>1</sub>  $\ll$  *bop*  $\gg$  *e*<sub>2</sub>) = *fv* *e*<sub>1</sub>  $\cup$  *fv* *e*<sub>2</sub>  
| *fv*(*Var* *V*) = {*V*}  
| *fv*(*LAss* *V* *e*) = {*V*}  $\cup$  *fv* *e*  
| *fv*(*e*  $\cdot$  *F*{*D*}) = *fv* *e*  
| *fv*(*e*<sub>1</sub>  $\cdot$  *F*{*D*}:=*e*<sub>2</sub>) = *fv* *e*<sub>1</sub>  $\cup$  *fv* *e*<sub>2</sub>  
| *fv*(*e*  $\cdot$  *M*(*es*)) = *fv* *e*  $\cup$  *fvs* *es*  
| *fv*({*V*:*T*; *e*}) = *fv* *e* - {*V*}  
| *fv*(*e*<sub>1</sub>;;*e*<sub>2</sub>) = *fv* *e*<sub>1</sub>  $\cup$  *fv* *e*<sub>2</sub>  
| *fv*(*if* (*b*) *e*<sub>1</sub> *else* *e*<sub>2</sub>) = *fv* *b*  $\cup$  *fv* *e*<sub>1</sub>  $\cup$  *fv* *e*<sub>2</sub>  
| *fv*(*while* (*b*) *e*) = *fv* *b*  $\cup$  *fv* *e*  
| *fv*(*throw* *e*) = *fv* *e*  
| *fv*(*try* *e*<sub>1</sub> *catch*(*C* *V*) *e*<sub>2</sub>) = *fv* *e*<sub>1</sub>  $\cup$  (*fv* *e*<sub>2</sub> - {*V*})  
| *fvs*([]) = {}  
| *fvs*(*e*#*es*) = *fv* *e*  $\cup$  *fvs* *es*

**lemma** [*simp*]: *fvs*(*es*<sub>1</sub> @ *es*<sub>2</sub>) = *fvs* *es*<sub>1</sub>  $\cup$  *fvs* *es*<sub>2</sub> *<proof>*

**lemma** [*simp*]: *fvs*(*map* *Val* *vs*) = {} *<proof>*

**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-prog*  $\Rightarrow$  *expr*  $\Rightarrow$  *state*  $\Rightarrow$  *expr*  $\Rightarrow$  *state*  $\Rightarrow$  *bool*  
 ( $\vdash$  ((*I*  $\langle$ -,/- $\rangle$ )  $\Rightarrow$  / (*I*  $\langle$ -,/- $\rangle$ )) [*51,0,0,0,0*] *81*)  
**and** *evals* :: *J-prog*  $\Rightarrow$  *expr list*  $\Rightarrow$  *state*  $\Rightarrow$  *expr list*  $\Rightarrow$  *state*  $\Rightarrow$  *bool*  
 ( $\vdash$  ((*I*  $\langle$ -,/- $\rangle$ ) [ $\Rightarrow$ ] / (*I*  $\langle$ -,/- $\rangle$ )) [*51,0,0,0,0*] *81*)  
**for** *P* :: *J-prog*

**where**

*New*:

$\llbracket$  *new-Addr* *h* = *Some a*; *P*  $\vdash$  *C* *has-fields FDTs*; *h'* = *h*(*a* $\mapsto$ (*C,init-fields FDTs*))  $\rrbracket$   
 $\Longrightarrow$  *P*  $\vdash$   $\langle$ *new C,(h,l)* $\rangle$   $\Rightarrow$   $\langle$ *addr a,(h',l)* $\rangle$

| *NewFail*:

*new-Addr* *h* = *None*  $\Longrightarrow$   
*P*  $\vdash$   $\langle$ *new C, (h,l)* $\rangle$   $\Rightarrow$   $\langle$ *THROW OutOfMemory,(h,l)* $\rangle$

| *Cast*:

$\llbracket$  *P*  $\vdash$   $\langle$ *e,s<sub>0</sub>* $\rangle$   $\Rightarrow$   $\langle$ *addr a,(h,l)* $\rangle$ ; *h a* = *Some(D,fs)*; *P*  $\vdash$  *D*  $\preceq^*$  *C*  $\rrbracket$   
 $\Longrightarrow$  *P*  $\vdash$   $\langle$ *Cast C e,s<sub>0</sub>* $\rangle$   $\Rightarrow$   $\langle$ *addr a,(h,l)* $\rangle$

| *CastNull*:

*P*  $\vdash$   $\langle$ *e,s<sub>0</sub>* $\rangle$   $\Rightarrow$   $\langle$ *null,s<sub>1</sub>* $\rangle$   $\Longrightarrow$   
*P*  $\vdash$   $\langle$ *Cast C e,s<sub>0</sub>* $\rangle$   $\Rightarrow$   $\langle$ *null,s<sub>1</sub>* $\rangle$

| *CastFail*:

$\llbracket$  *P*  $\vdash$   $\langle$ *e,s<sub>0</sub>* $\rangle$   $\Rightarrow$   $\langle$ *addr a,(h,l)* $\rangle$ ; *h a* = *Some(D,fs)*;  $\neg$  *P*  $\vdash$  *D*  $\preceq^*$  *C*  $\rrbracket$   
 $\Longrightarrow$  *P*  $\vdash$   $\langle$ *Cast C e,s<sub>0</sub>* $\rangle$   $\Rightarrow$   $\langle$ *THROW ClassCast,(h,l)* $\rangle$

| *CastThrow*:

*P*  $\vdash$   $\langle$ *e,s<sub>0</sub>* $\rangle$   $\Rightarrow$   $\langle$ *throw e',s<sub>1</sub>* $\rangle$   $\Longrightarrow$   
*P*  $\vdash$   $\langle$ *Cast C e,s<sub>0</sub>* $\rangle$   $\Rightarrow$   $\langle$ *throw e',s<sub>1</sub>* $\rangle$

| *Val*:

*P*  $\vdash$   $\langle$ *Val v,s* $\rangle$   $\Rightarrow$   $\langle$ *Val v,s* $\rangle$

| *BinOp*:

$\llbracket$  *P*  $\vdash$   $\langle$ *e<sub>1</sub>,s<sub>0</sub>* $\rangle$   $\Rightarrow$   $\langle$ *Val v<sub>1</sub>,s<sub>1</sub>* $\rangle$ ; *P*  $\vdash$   $\langle$ *e<sub>2</sub>,s<sub>1</sub>* $\rangle$   $\Rightarrow$   $\langle$ *Val v<sub>2</sub>,s<sub>2</sub>* $\rangle$ ; *binop(bop,v<sub>1</sub>,v<sub>2</sub>)* = *Some v*  $\rrbracket$   
 $\Longrightarrow$  *P*  $\vdash$   $\langle$ *e<sub>1</sub> <<bop>> e<sub>2</sub>,s<sub>0</sub>* $\rangle$   $\Rightarrow$   $\langle$ *Val v,s<sub>2</sub>* $\rangle$

| *BinOpThrow1*:

*P*  $\vdash$   $\langle$ *e<sub>1</sub>,s<sub>0</sub>* $\rangle$   $\Rightarrow$   $\langle$ *throw e,s<sub>1</sub>* $\rangle$   $\Longrightarrow$   
*P*  $\vdash$   $\langle$ *e<sub>1</sub> <<bop>> e<sub>2</sub>, s<sub>0</sub>* $\rangle$   $\Rightarrow$   $\langle$ *throw e,s<sub>1</sub>* $\rangle$

| *BinOpThrow2*:

$\llbracket$  *P*  $\vdash$   $\langle$ *e<sub>1</sub>,s<sub>0</sub>* $\rangle$   $\Rightarrow$   $\langle$ *Val v<sub>1</sub>,s<sub>1</sub>* $\rangle$ ; *P*  $\vdash$   $\langle$ *e<sub>2</sub>,s<sub>1</sub>* $\rangle$   $\Rightarrow$   $\langle$ *throw e,s<sub>2</sub>* $\rangle$   $\rrbracket$   
 $\Longrightarrow$  *P*  $\vdash$   $\langle$ *e<sub>1</sub> <<bop>> e<sub>2</sub>,s<sub>0</sub>* $\rangle$   $\Rightarrow$   $\langle$ *throw e,s<sub>2</sub>* $\rangle$

| *Var*:

*l V* = *Some v*  $\Longrightarrow$   
*P*  $\vdash$   $\langle$ *Var V,(h,l)* $\rangle$   $\Rightarrow$   $\langle$ *Val v,(h,l)* $\rangle$

- | *LAss*:  

$$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, (h, l) \rangle; l' = l(V \mapsto v) \rrbracket$$

$$\Longrightarrow P \vdash \langle V := e, s_0 \rangle \Rightarrow \langle \text{unit}, (h, l') \rangle$$
- | *LAssThrow*:  

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

$$P \vdash \langle V := e, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$$
- | *FAcc*:  

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

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

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

$$P \vdash \langle e \cdot F \{D\}, s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_1 \rangle$$
- | *FAccThrow*:  

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

$$P \vdash \langle e \cdot F \{D\}, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$$
- | *FAss*:  

$$\llbracket P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2) \rangle;$$

$$h_2 \ a = \text{Some}(C, fs); fs' = fs((F, D) \mapsto v); h_2' = h_2(a \mapsto (C, fs')) \rrbracket$$

$$\Longrightarrow P \vdash \langle e_1 \cdot F \{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{unit}, (h_2', l_2) \rangle$$
- | *FAssNull*:  

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

$$P \vdash \langle e_1 \cdot F \{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_2 \rangle$$
- | *FAssThrow1*:  

$$P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \Longrightarrow$$

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

$$\llbracket P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle \rrbracket$$

$$\Longrightarrow P \vdash \langle e_1 \cdot F \{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle$$
- | *CallObjThrow*:  

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

$$P \vdash \langle e \cdot M(ps), s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$$
- | *CallParamsThrow*:  

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

$$\Longrightarrow P \vdash \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle \text{throw } ex, s_2 \rangle$$
- | *CallNull*:  

$$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle; P \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, s_2 \rangle \rrbracket$$

$$\Longrightarrow P \vdash \langle e \cdot M(ps), s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_2 \rangle$$
- | *Call*:  

$$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_2, l_2) \rangle;$$

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

$$\text{length } vs = \text{length } pns; l_2' = [\text{this} \mapsto \text{Addr } a, pns[\mapsto] vs];$$

$$\begin{aligned} & P \vdash \langle \text{body}, (h_2, l_2') \rangle \Rightarrow \langle e', (h_3, l_3) \rangle \llbracket \\ \Rightarrow & P \vdash \langle e \cdot M(ps), s_0 \rangle \Rightarrow \langle e', (h_3, l_2) \rangle \end{aligned}$$

| *Block*:

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

| *Seq*:

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

| *SeqThrow*:

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

| *CondT*:

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

| *CondF*:

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

| *CondThrow*:

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

| *WhileF*:

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

| *WhileT*:

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

| *WhileCondThrow*:

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

| *WhileBodyThrow*:

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

| *Throw*:

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

| *ThrowNull*:

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

| *ThrowThrow*:

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

| *Try*:

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

| *TryCatch*:

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

| *TryThrow*:

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

| *Nil*:

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

| *Cons*:

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

| *ConsThrow*:

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

## 2.10.1 Final expressions

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

where

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

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

where

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

**lemma** [*simp*]: *final*(Val v)⟨proof⟩

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

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

**lemma** [*iff*]: *finals* []⟨proof⟩

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

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

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

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

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

⟨proof⟩

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

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

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

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

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

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

**lemma** *eval-finalsId*:

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

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

**and** *evals-hext*:  $P \vdash \langle es, (h, l) \rangle [\Rightarrow] \langle es', (h', l') \rangle \implies h \trianglelefteq h' \langle proof \rangle$

**end**



## 2.11 Small Step Semantics

```
theory SmallStep
imports Expr State
begin
```

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

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

```
lemma [simp]:
  [| size vs = size Vs; size Ts = size Vs |] ⇒ fv(blocks(Vs,Ts,vs,e)) = fv e - set Vs<proof>
```

```
definition assigned :: vname ⇒ expr ⇒ bool
where
  assigned V e ≡ ∃ v e'. e = (V := Val v;; e')
```

```
inductive-set
  red :: J-prog ⇒ ((expr × state) × (expr × state)) set
  and reds :: J-prog ⇒ ((expr list × state) × (expr list × state)) set
  and red' :: J-prog ⇒ expr ⇒ state ⇒ expr ⇒ state ⇒ bool
    (- ⊢ ((1⟨-,/-⟩) → / (1⟨-,/-⟩)) [51,0,0,0,0] 81)
  and reds' :: J-prog ⇒ expr list ⇒ state ⇒ expr list ⇒ state ⇒ bool
    (- ⊢ ((1⟨-,/-⟩) [→] / (1⟨-,/-⟩)) [51,0,0,0,0] 81)
  for P :: J-prog
where
```

```
  P ⊢ ⟨e,s⟩ → ⟨e',s'⟩ ≡ ((e,s), e',s') ∈ red P
| P ⊢ ⟨es,s⟩ [→] ⟨es',s'⟩ ≡ ((es,s), es',s') ∈ reds P
```

```
| RedNew:
  [| new-Addr h = Some a; P ⊢ C has-fields FDTs; h' = h(a→(C,init-fields FDTs)) |]
  ⇒ P ⊢ ⟨new C, (h,l)⟩ → ⟨addr a, (h',l)⟩
```

```
| RedNewFail:
  new-Addr h = None ⇒
  P ⊢ ⟨new C, (h,l)⟩ → ⟨THROW OutOfMemory, (h,l)⟩
```

```
| CastRed:
  P ⊢ ⟨e,s⟩ → ⟨e',s'⟩ ⇒
  P ⊢ ⟨Cast C e, s⟩ → ⟨Cast C e', s'⟩
```

```
| RedCastNull:
  P ⊢ ⟨Cast C null, s⟩ → ⟨null,s⟩
```

```
| RedCast:
  [| hp s a = Some(D,fs); P ⊢ D ≼* C |]
  ⇒ P ⊢ ⟨Cast C (addr a), s⟩ → ⟨addr a, s⟩
```

```
| RedCastFail:
  [| hp s a = Some(D,fs); ¬ P ⊢ D ≼* C |]
```

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

| *BinOpRed1*:

$$\begin{aligned} P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle &\implies \\ P \vdash \langle e \ll bop \gg e_2, s \rangle &\rightarrow \langle e' \ll bop \gg e_2, s' \rangle \end{aligned}$$

| *BinOpRed2*:

$$\begin{aligned} P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle &\implies \\ P \vdash \langle (\text{Val } v_1) \ll bop \gg e, s \rangle &\rightarrow \langle (\text{Val } v_1) \ll bop \gg e', s' \rangle \end{aligned}$$

| *RedBinOp*:

$$\begin{aligned} \text{binop}(bop, v_1, v_2) = \text{Some } v &\implies \\ P \vdash \langle (\text{Val } v_1) \ll bop \gg (\text{Val } v_2), s \rangle &\rightarrow \langle \text{Val } v, s \rangle \end{aligned}$$

| *RedVar*:

$$\begin{aligned} \text{lcl } s \text{ } V = \text{Some } v &\implies \\ P \vdash \langle \text{Var } V, s \rangle &\rightarrow \langle \text{Val } v, s \rangle \end{aligned}$$

| *LAssRed*:

$$\begin{aligned} P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle &\implies \\ P \vdash \langle V := e, s \rangle &\rightarrow \langle V := e', s' \rangle \end{aligned}$$

| *RedLAss*:

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

| *FAccRed*:

$$\begin{aligned} P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle &\implies \\ P \vdash \langle e \cdot F\{D\}, s \rangle &\rightarrow \langle e' \cdot F\{D\}, s' \rangle \end{aligned}$$

| *RedFAcc*:

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

| *RedFAccNull*:

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

| *FAssRed1*:

$$\begin{aligned} P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle &\implies \\ P \vdash \langle e \cdot F\{D\} := e_2, s \rangle &\rightarrow \langle e' \cdot F\{D\} := e_2, s' \rangle \end{aligned}$$

| *FAssRed2*:

$$\begin{aligned} P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle &\implies \\ P \vdash \langle \text{Val } v \cdot F\{D\} := e, s \rangle &\rightarrow \langle \text{Val } v \cdot F\{D\} := e', s' \rangle \end{aligned}$$

| *RedFAss*:

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

| *RedFAssNull*:

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

| *CallObj*:

$$\begin{aligned} P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle &\implies \\ P \vdash \langle e \cdot M(es), s \rangle &\rightarrow \langle e' \cdot M(es), s' \rangle \end{aligned}$$

| *CallParams*:

$$\begin{aligned} P \vdash \langle es, s \rangle &\rightarrow \langle es', s' \rangle \implies \\ P \vdash \langle (Val\ v) \cdot M(es), s \rangle &\rightarrow \langle (Val\ v) \cdot M(es'), s' \rangle \end{aligned}$$

| *RedCall*:

$$\begin{aligned} \llbracket hp\ s\ a = Some(C, fs); P \vdash C\ sees\ M:Ts \rightarrow T = (pns, body)\ in\ D; size\ vs = size\ pns; size\ Ts = size\ pns \rrbracket \\ \implies P \vdash \langle (addr\ a) \cdot M(map\ Val\ vs), s \rangle \rightarrow \langle blocks(this\#pns, Class\ D\#Ts, Addr\ a\#vs, body), s \rangle \end{aligned}$$

| *RedCallNull*:

$$P \vdash \langle null \cdot M(map\ Val\ vs), s \rangle \rightarrow \langle THROW\ NullPointer, s \rangle$$

| *BlockRedNone*:

$$\begin{aligned} \llbracket P \vdash \langle e, (h, l(V := None)) \rangle \rightarrow \langle e', (h', l') \rangle; l'\ V = None; \neg assigned\ V\ e \rrbracket \\ \implies P \vdash \langle \{V:T; e\}, (h, l) \rangle \rightarrow \langle \{V:T; e'\}, (h', l'(V := l\ V)) \rangle \end{aligned}$$

| *BlockRedSome*:

$$\begin{aligned} \llbracket P \vdash \langle e, (h, l(V := None)) \rangle \rightarrow \langle e', (h', l') \rangle; l'\ V = Some\ v; \neg assigned\ V\ e \rrbracket \\ \implies P \vdash \langle \{V:T; e\}, (h, l) \rangle \rightarrow \langle \{V:T := Val\ v; e'\}, (h', l'(V := l\ V)) \rangle \end{aligned}$$

| *InitBlockRed*:

$$\begin{aligned} \llbracket P \vdash \langle e, (h, l(V \mapsto v)) \rangle \rightarrow \langle e', (h', l') \rangle; l'\ V = Some\ v' \rrbracket \\ \implies P \vdash \langle \{V:T := Val\ v; e\}, (h, l) \rangle \rightarrow \langle \{V:T := Val\ v'; e'\}, (h', l'(V := l\ V)) \rangle \end{aligned}$$

| *RedBlock*:

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

| *RedInitBlock*:

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

| *SeqRed*:

$$\begin{aligned} P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies \\ P \vdash \langle e;;e_2, s \rangle \rightarrow \langle e';;e_2, s' \rangle \end{aligned}$$

| *RedSeq*:

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

| *CondRed*:

$$\begin{aligned} P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies \\ P \vdash \langle if\ (e)\ e_1\ else\ e_2, s \rangle \rightarrow \langle if\ (e')\ e_1\ else\ e_2, s' \rangle \end{aligned}$$

| *RedCondT*:

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

| *RedCondF*:

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

| *RedWhile*:

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

| *ThrowRed*:

$$\begin{aligned} P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies \\ P \vdash \langle throw\ e, s \rangle \rightarrow \langle throw\ e', s' \rangle \end{aligned}$$

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

— Exception propagation

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

### 2.11.1 The reflexive transitive closure

#### abbreviation

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

#### abbreviation

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

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

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

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

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

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

**shows**  $R e h l e' h' l'$  (proof)

### 2.11.2 Some easy lemmas

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

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

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

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

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

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

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

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

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

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

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

end

## 2.12 System Classes

```

theory SystemClasses
imports Decl Exceptions
begin

```

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

```

definition ObjectC :: 'm cdecl
where
  ObjectC  $\equiv$  (Object, (undefined, [], []))

```

```

definition NullPointerC :: 'm cdecl
where
  NullPointerC  $\equiv$  (NullPointer, (Object, [], []))

```

```

definition ClassCastC :: 'm cdecl
where
  ClassCastC  $\equiv$  (ClassCast, (Object, [], []))

```

```

definition OutOfMemoryC :: 'm cdecl
where
  OutOfMemoryC  $\equiv$  (OutOfMemory, (Object, [], []))

```

```

definition SystemClasses :: 'm cdecl list
where
  SystemClasses  $\equiv$  [ObjectC, NullPointerC, ClassCastC, OutOfMemoryC]

```

```

end

```

## 2.13 Generic Well-formedness of programs

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

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

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

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

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

**where**

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

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

**where**

$wf-mdecl$  *wf-md P C*  $\equiv$   $\lambda(M,Ts,T,mb).$   
 $(\forall T \in set\ Ts. is-type\ P\ T) \wedge is-type\ P\ T \wedge wf-md\ P\ C\ (M,Ts,T,mb)$

**definition** *wf-cdecl* ::  $'m$  *wf-mdecl-test*  $\Rightarrow$   $'m$  *prog*  $\Rightarrow$   $'m$  *cdecl*  $\Rightarrow$  *bool*

**where**

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

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

**where**

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

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

**where**

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

### 2.13.1 Well-formedness lemmas

**lemma** *class-wf*:

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

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

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

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

$wf-prog\ wf-md\ P \Longrightarrow is-class\ P\ Object\langle proof \rangle$

**lemma** *is-class-xcpt*:

$\llbracket C \in sys-xcpts; wf-prog\ wf-md\ P \rrbracket \Longrightarrow is-class\ P\ C\langle proof \rangle$

**lemma** *subcls1-wfD*:

$$\llbracket P \vdash C \prec^1 D; \text{wf-prog wf-md } P \rrbracket \Longrightarrow D \neq C \wedge (D, C) \notin (\text{subcls1 } P)^+ \langle \text{proof} \rangle$$

**lemma** *wf-cdecl-supD*:

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

**lemma** *subcls-asym*:

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

**lemma** *subcls-irrefl*:

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

**lemma** *acyclic-subcls1*:

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

**lemma** *wf-subcls1*:

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

**lemma** *single-valued-subcls1*:

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

**lemma** *subcls-induct*:

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

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

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

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

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

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

$$\llbracket \text{is-class } P C; \text{wf-prog wf-md } P \rrbracket \Longrightarrow P \vdash C \preceq^* \text{Object} \langle \text{proof} \rangle$$

**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**  $\text{set } Ts \subseteq \text{types } P \langle \text{proof} \rangle$

## 2.13.2 Well-formedness and method lookup

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

$$\llbracket \text{wf-prog wf-md } P; P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } D \rrbracket \Longrightarrow \text{wf-mdecl wf-md } P D (M, Ts, T, m) \langle \text{proof} \rangle$$

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

$$\begin{aligned} & \llbracket P \vdash C' \preceq^* C; \text{wf-prog wf-md } P \rrbracket \Longrightarrow \\ & \forall D Ts T m. P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } D \longrightarrow \\ & \quad (\exists D' Ts' T' m'. P \vdash C' \text{ sees } M: Ts' \rightarrow T' = m' \text{ in } D' \wedge P \vdash Ts [\leq] Ts' \wedge P \vdash T' \leq T) \langle \text{proof} \rangle \end{aligned}$$



**lemma sees-method-mono2:**

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

**lemma mdecls-visible:**

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

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

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

**lemma mdecl-visible:**

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

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

**lemma Call-lemma:**

$\llbracket P \vdash C \text{ sees } M:Ts \rightarrow T = m \text{ in } D; P \vdash C' \preceq^* C; \text{wf-prog wf-md } P \rrbracket$

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

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

**lemma wf-prog-lift:**

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

**and rule:**

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

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

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

$\text{set } Ts \subseteq \text{types } P \implies$

$\text{bd} = (M, Ts, T, m) \implies$

$A P C \text{ bd} \implies$

$B P C \text{ bd}$

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

### 2.13.3 Well-formedness and field lookup

**lemma wf-Fields-Ex:**

$\llbracket \text{wf-prog wf-md } P; \text{is-class } P C \rrbracket \implies \exists FDTs. P \vdash C \text{ has-fields } FDTs \langle \text{proof} \rangle$

**lemma has-fields-types:**

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

**lemma sees-field-is-type:**

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

**lemma wf-syscls:**

$\text{set SystemClasses} \subseteq \text{set } P \implies \text{wf-syscls } P \langle \text{proof} \rangle$

**end**

## 2.14 Weak well-formedness of Jinja programs

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

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

**where**

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

$length\ Ts = length\ pns \wedge distinct\ pns \wedge this \notin set\ pns \wedge fv\ body \subseteq \{this\} \cup set\ pns$

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

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

$(length\ Ts = length\ pns \wedge distinct\ pns \wedge this \notin set\ pns \wedge fv\ body \subseteq \{this\} \cup set\ pns)$ *(proof)*

**abbreviation**

*wuf-J-prog* :: *J-prog*  $\Rightarrow$  *bool* **where**

*wuf-J-prog* == *wf-prog wuf-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*:

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

**lemma** *CastRedsNull*:

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

**lemma** *CastRedsAddr*:

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

**lemma** *CastRedsFail*:

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

**lemma** *CastRedsThrow*:

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

#### LAss

**lemma** *LAssReds*:

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

**lemma** *LAssRedsVal*:

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

**lemma** *LAssRedsThrow*:

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

#### BinOp

**lemma** *BinOp1Reds*:

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

**lemma** *BinOp2Reds*:

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

**lemma** *BinOpRedsVal*:

$$\llbracket P \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle \text{Val } v_1, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle \text{Val } v_2, s_2 \rangle; \text{binop}(\text{bop}, v_1, v_2) = \text{Some } v \rrbracket \implies P \vdash \langle e_1 \ll \text{bop} \gg e_2, s_0 \rangle \rightarrow^* \langle \text{Val } v, s_2 \rangle \langle \text{proof} \rangle$$

**lemma** *BinOpRedsThrow1*:

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

**lemma** *BinOpRedsThrow2*:

$$\llbracket P \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle \text{Val } v_1, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle \text{throw } e, s_2 \rangle \rrbracket \implies P \vdash \langle e_1 \ll \text{bop} \gg e_2, s_0 \rangle \rightarrow^* \langle \text{throw } e, s_2 \rangle \langle \text{proof} \rangle$$

#### FAcc

**lemma** *FAccReds*:

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

**lemma** *FAccRedsVal*:

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

**lemma** *FAccRedsNull*:

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

**lemma** *FAccRedsThrow*:

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

## FAss

**lemma** *FAssReds1*:

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

**lemma** *FAssReds2*:

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

**lemma** *FAssRedsVal*:

$$\llbracket P \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle \text{addr } a, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle \text{Val } v, (h_2, l_2) \rangle; \text{Some}(C, fs) = h_2 \ a \rrbracket \implies P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \rightarrow^* \langle \text{unit}, (h_2(a \mapsto (C, fs((F, D) \mapsto v))), l_2) \rangle \langle \text{proof} \rangle$$

**lemma** *FAssRedsNull*:

$$\llbracket P \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle \text{null}, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle \text{Val } v, s_2 \rangle \rrbracket \implies P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \rightarrow^* \langle \text{THROW } \text{NullPointer}, s_2 \rangle \langle \text{proof} \rangle$$

**lemma** *FAssRedsThrow1*:

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

**lemma** *FAssRedsThrow2*:

$$\llbracket P \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle \text{Val } v, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle \text{throw } e, s_2 \rangle \rrbracket \implies P \vdash \langle e_1 \cdot F\{D\} := e_2, s_0 \rangle \rightarrow^* \langle \text{throw } e, s_2 \rangle \langle \text{proof} \rangle$$

;;

**lemma** *SeqReds*:

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

**lemma** *SeqRedsThrow*:

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

**lemma** *SeqReds2*:

$$\llbracket P \vdash \langle e_1, s_0 \rangle \rightarrow^* \langle \text{Val } v_1, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle e_2', s_2 \rangle \rrbracket \implies P \vdash \langle e_1 ;; e_2, s_0 \rangle \rightarrow^* \langle e_2', s_2 \rangle \langle \text{proof} \rangle$$

## If

**lemma** *CondReds*:

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

**lemma** *CondRedsThrow*:

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

**lemma** *CondReds2T*:

$$\llbracket P \vdash \langle e, s_0 \rangle \rightarrow^* \langle \text{true}, s_1 \rangle; P \vdash \langle e_1, s_1 \rangle \rightarrow^* \langle e', s_2 \rangle \rrbracket \implies P \vdash \langle \text{if } (e) \ e_1 \ \text{else } e_2, s_0 \rangle \rightarrow^* \langle e', s_2 \rangle \langle \text{proof} \rangle$$

**lemma** *CondReds2F*:

$$\llbracket P \vdash \langle e, s_0 \rangle \rightarrow^* \langle \text{false}, s_1 \rangle; P \vdash \langle e_2, s_1 \rangle \rightarrow^* \langle e', s_2 \rangle \rrbracket \implies P \vdash \langle \text{if } (e) \ e_1 \ \text{else } e_2, s_0 \rangle \rightarrow^* \langle e', s_2 \rangle \langle \text{proof} \rangle$$

## While

**lemma** *WhileFReds*:

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

**lemma** *WhileRedsThrow*:

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

**lemma** *WhileTReds*:

$$\llbracket P \vdash \langle b, s_0 \rangle \rightarrow^* \langle \text{true}, s_1 \rangle; P \vdash \langle c, s_1 \rangle \rightarrow^* \langle \text{Val } v_1, s_2 \rangle; P \vdash \langle \text{while } (b) \ c, s_2 \rangle \rightarrow^* \langle e, s_3 \rangle \rrbracket \implies P \vdash \langle \text{while } (b) \ c, s_0 \rangle \rightarrow^* \langle e, s_3 \rangle \langle \text{proof} \rangle$$

**lemma** *WhileTRedsThrow*:

$$\llbracket P \vdash \langle b, s_0 \rangle \rightarrow^* \langle \text{true}, s_1 \rangle; P \vdash \langle c, s_1 \rangle \rightarrow^* \langle \text{throw } e, s_2 \rangle \rrbracket \implies P \vdash \langle \text{while } (b) \ c, s_0 \rangle \rightarrow^* \langle \text{throw } e, s_2 \rangle \langle \text{proof} \rangle$$

**Throw****lemma** *ThrowReds*:

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

**lemma** *ThrowRedsNull*:

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

**lemma** *ThrowRedsThrow*:

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

**InitBlock****lemma** *InitBlockReds-aux*:

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

**lemma** *InitBlockReds*:

$$\begin{aligned} & P \vdash \langle e, (h, l(V \mapsto v)) \rangle \rightarrow^* \langle e', (h', l') \rangle \implies \\ & P \vdash \langle \{V:T := \text{Val } v; e\}, (h, l) \rangle \rightarrow^* \langle \{V:T := \text{Val}(\text{the}(l' \, V)); e'\}, (h', l'(V := (l \, V))) \rangle \langle \text{proof} \rangle \end{aligned}$$

**lemma** *InitBlockRedsFinal*:

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

**Block****lemma** *BlockRedsFinal*:

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

**try-catch****lemma** *TryReds*:

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

**lemma** *TryRedsVal*:

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

**lemma** *TryCatchRedsFinal*:

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

**lemma** *TryRedsFail*:

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

**List****lemma** *ListReds1*:

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

**lemma** *ListReds2*:

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

**lemma** *ListRedsVal*:

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

**Call**

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

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

**shows**  $Red\text{-}fv: P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies fv\ e' \subseteq fv\ e$

**and**  $P \vdash \langle es, (h, l) \rangle [\rightarrow] \langle es', (h', l') \rangle \implies fvs\ es' \subseteq fvs\ es \langle proof \rangle$

**lemma**  $Red\text{-}dom\text{-}lcl:$

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

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

**lemma**  $Reds\text{-}dom\text{-}lcl:$

$\llbracket wwf\text{-}J\text{-prog } P; P \vdash \langle e, (h, l) \rangle \rightarrow^* \langle e', (h', l') \rangle \rrbracket \implies dom\ l' \subseteq dom\ l \cup fv\ e \langle proof \rangle$

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

**lemma**  $override\text{-}on\text{-}upd\text{-}lemma:$

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

**lemma**  $blocksReds:$

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

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

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

**lemma**  $blocksFinal:$

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

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

**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**  $l'': l'' = override\text{-}on\ l'\ l\ (set\ Vs)$

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

An now the actual method call reduction lemmas.

**lemma**  $CallRedsObj:$

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

**lemma**  $CallRedsParams:$

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

**lemma**  $CallRedsFinal:$

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

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

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

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

$size\ vs = size\ pns$

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

**and**  $body: P \vdash \langle body, (h_2, l_2') \rangle \rightarrow^* \langle ef, (h_3, l_3) \rangle$

**and**  $final\ ef$

**shows**  $P \vdash \langle e \cdot M(es), s_0 \rangle \rightarrow^* \langle ef, (h_3, l_2) \rangle \langle proof \rangle$

**lemma**  $CallRedsThrowParams:$

$\llbracket P \vdash \langle e, s_0 \rangle \rightarrow^* \langle Val\ v, s_1 \rangle; P \vdash \langle es, s_1 \rangle [\rightarrow]^* \langle map\ Val\ vs_1\ @\ throw\ a\ \#\ es_2, s_2 \rangle \rrbracket$

$\implies P \vdash \langle e \cdot M(es), s_0 \rangle \rightarrow^* \langle throw\ a, s_2 \rangle \langle proof \rangle$

**lemma**  $CallRedsThrowObj:$

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

**lemma** *CallRedsNull*:

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

## The main Theorem

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

**shows** *big-by-small*:  $P \vdash \langle e, s \rangle \Rightarrow \langle e', s^\wedge \rangle \implies P \vdash \langle e, s \rangle \rightarrow^* \langle e', s^\wedge \rangle$

**and** *bigs-by-smalls*:  $P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s^\wedge \rangle \implies P \vdash \langle es, s \rangle [\rightarrow]^* \langle es', s^\wedge \rangle \langle \text{proof} \rangle$

### 2.15.2 Big steps simulates small step

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

The big step equivalent of *RedWhile*:

**lemma** *unfold-while*:

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

**lemma** *blocksEval*:

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

**lemma**

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

**shows** *eval-restrict-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**  $P \vdash \langle es, (h, l) \rangle [\Rightarrow] \langle es', (h', l') \rangle \implies (\bigwedge W. \text{fvs } es \subseteq W \implies P \vdash \langle es, (h, l|'W) \rangle [\Rightarrow] \langle es', (h', l'|'W) \rangle) \langle \text{proof} \rangle$

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

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

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

**lemma** *list-eval-Throw*:

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

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

The key lemma:

**lemma**

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

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

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

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

$$P \vdash \langle es, t \rangle [\rightarrow] \langle es'', t'' \rangle \implies (\bigwedge t' es'. P \vdash \langle es'', t'' \rangle [\Rightarrow] \langle es', t^\wedge \rangle \implies P \vdash \langle es, t \rangle [\Rightarrow] \langle es', t^\wedge \rangle) \langle \text{proof} \rangle$$

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

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

**lemma** *extend-evals*:

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

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

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

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

**theorem**

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

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

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

### 2.15.3 Equivalence

And now, the crowning achievement:

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

$wf\text{-}J\text{-}prog\ P \Longrightarrow$

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

**end**



## 2.16 Well-typedness of Jinja expressions

```

theory WellType
imports ../Common/Objects Expr
begin

type-synonym
  env = vname  $\rightarrow$  ty

inductive
  WT :: [J-prog, env, expr, ty]  $\Rightarrow$  bool
  (-, -  $\vdash$  - :: - [51,51,51]50)
  and WTs :: [J-prog, env, expr list, ty list]  $\Rightarrow$  bool
  (-, -  $\vdash$  - [::] - [51,51,51]50)
  for P :: J-prog
where

  WTNew:
    is-class P C  $\Longrightarrow$ 
    P, E  $\vdash$  new C :: Class C

  | WTCast:
    [[ P, E  $\vdash$  e :: Class D; is-class P C; P  $\vdash$  C  $\leq^*$  D  $\vee$  P  $\vdash$  D  $\leq^*$  C ]
     $\Longrightarrow$  P, E  $\vdash$  Cast C e :: Class C

  | WTVal:
    typeof v = Some T  $\Longrightarrow$ 
    P, E  $\vdash$  Val v :: T

  | WTVar:
    E V = Some T  $\Longrightarrow$ 
    P, E  $\vdash$  Var V :: T

  | WTBinOpEq:
    [[ P, E  $\vdash$  e1 :: T1; P, E  $\vdash$  e2 :: T2; P  $\vdash$  T1  $\leq$  T2  $\vee$  P  $\vdash$  T2  $\leq$  T1 ]
     $\Longrightarrow$  P, E  $\vdash$  e1  $\ll$ Eq $\gg$  e2 :: Boolean

  | WTBinOpAdd:
    [[ P, E  $\vdash$  e1 :: Integer; P, E  $\vdash$  e2 :: Integer ]
     $\Longrightarrow$  P, E  $\vdash$  e1  $\ll$ Add $\gg$  e2 :: Integer

  | WTLAss:
    [[ E V = Some T; P, E  $\vdash$  e :: T'; P  $\vdash$  T'  $\leq$  T; V  $\neq$  this ]
     $\Longrightarrow$  P, E  $\vdash$  V := e :: Void

  | WTFAcc:
    [[ P, E  $\vdash$  e :: Class C; P  $\vdash$  C sees F:T in D ]
     $\Longrightarrow$  P, E  $\vdash$  e  $\bullet$  F {D} :: T

  | WTFAss:
    [[ P, E  $\vdash$  e1 :: Class C; P  $\vdash$  C sees F:T in D; P, E  $\vdash$  e2 :: T'; P  $\vdash$  T'  $\leq$  T ]
     $\Longrightarrow$  P, E  $\vdash$  e1  $\bullet$  F {D} := e2 :: Void

  | WTCall:

```

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

$$\begin{aligned} & | \text{WTBlock:} \\ & \llbracket \text{is-type } P T; P, E(V \mapsto T) \vdash e :: T' \rrbracket \\ & \implies P, E \vdash \{V:T; e\} :: T' \end{aligned}$$

$$\begin{aligned} & | \text{WTSeq:} \\ & \llbracket P, E \vdash e_1 :: T_1; P, E \vdash e_2 :: T_2 \rrbracket \\ & \implies P, E \vdash e_1; e_2 :: T_2 \end{aligned}$$

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

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

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

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

— well-typed expression lists

$$\begin{aligned} & | \text{WTNil:} \\ & P, E \vdash [] \llbracket :: \rrbracket \end{aligned}$$

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

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

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

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

**lemma** [iff]:  $\bigwedge Ts. (P, E \vdash es_1 @ es_2 \llbracket :: Ts) =$   
 $(\exists Ts_1 Ts_2. Ts = Ts_1 @ Ts_2 \wedge P, E \vdash es_1 \llbracket :: Ts_1 \wedge P, E \vdash es_2 \llbracket :: Ts_2) \langle \text{proof} \rangle$

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

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

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

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

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

$$\begin{aligned} & P, E \vdash e :: T \implies (\bigwedge E'. E \subseteq_m E' \implies P, E' \vdash e :: T) \text{ and} \\ & P, E \vdash es \llbracket :: Ts \implies (\bigwedge E'. E \subseteq_m E' \implies P, E' \vdash es \llbracket :: Ts) \langle \text{proof} \rangle \end{aligned}$$

**lemma** *WT-fv*:  $P, E \vdash e :: T \implies \text{fv } e \subseteq \text{dom } E$   
**and**  $P, E \vdash es \llbracket :: Ts \implies \text{fvs } es \subseteq \text{dom } E \langle \text{proof} \rangle$

## 2.17 Runtime Well-typedness

```

theory WellTypeRT
imports WellType
begin

inductive
  WTrt :: J-prog ⇒ heap ⇒ env ⇒ expr ⇒ ty ⇒ bool
  and WTrts :: J-prog ⇒ heap ⇒ env ⇒ expr list ⇒ ty list ⇒ bool
  and WTrt2 :: [J-prog,env,heap,expr,ty] ⇒ bool
    (·,·,- ⊢ - : - [51,51,51]50)
  and WTrts2 :: [J-prog,env,heap,expr list, ty list] ⇒ bool
    (·,·,- ⊢ - [:] - [51,51,51]50)
  for P :: J-prog and h :: heap
where

  P,E,h ⊢ e : T ≡ WTrt P h E e T
  | P,E,h ⊢ es[:]Ts ≡ WTrts P h E es Ts

  | WTrtNew:
    is-class P C ⇒
    P,E,h ⊢ new C : Class C

  | WTrtCast:
    [ [ P,E,h ⊢ e : T; is-refT T; is-class P C ]
    ⇒ P,E,h ⊢ Cast C e : Class C

  | WTrtVal:
    typeofh v = Some T ⇒
    P,E,h ⊢ Val v : T

  | WTrtVar:
    E V = Some T ⇒
    P,E,h ⊢ Var V : T

  | WTrtBinOpEq:
    [ [ P,E,h ⊢ e1 : T1; P,E,h ⊢ e2 : T2 ]
    ⇒ P,E,h ⊢ e1 «Eq» e2 : Boolean

  | WTrtBinOpAdd:
    [ [ P,E,h ⊢ e1 : Integer; P,E,h ⊢ e2 : Integer ]
    ⇒ P,E,h ⊢ e1 «Add» e2 : Integer

  | WTrtLAss:
    [ [ E V = Some T; P,E,h ⊢ e : T'; P ⊢ T' ≤ T ]
    ⇒ P,E,h ⊢ V:=e : Void

  | WTrtFAcc:
    [ [ P,E,h ⊢ e : Class C; P ⊢ C has F:T in D ] ⇒
    P,E,h ⊢ e·F{D} : T

  | WTrtFAccNT:
    P,E,h ⊢ e : NT ⇒
    P,E,h ⊢ e·F{D} : T

```

- | *WTrtFAss*:  
 $\llbracket P, E, h \vdash e_1 : \text{Class } C; P \vdash C \text{ has } F:T \text{ in } D; P, E, h \vdash e_2 : T_2; P \vdash T_2 \leq T \rrbracket$   
 $\implies P, E, h \vdash e_1 \cdot F\{D\} := e_2 : \text{Void}$
- | *WTrtFAssNT*:  
 $\llbracket P, E, h \vdash e_1 : NT; P, E, h \vdash e_2 : T_2 \rrbracket$   
 $\implies P, E, h \vdash e_1 \cdot F\{D\} := e_2 : \text{Void}$
- | *WTrtCall*:  
 $\llbracket P, E, h \vdash e : \text{Class } C; P \vdash C \text{ sees } M:Ts \rightarrow T = (\text{pns}, \text{body}) \text{ in } D;$   
 $P, E, h \vdash es \text{ } [:] \text{ } Ts'; P \vdash Ts' \leq Ts \rrbracket$   
 $\implies P, E, h \vdash e \cdot M(es) : T$
- | *WTrtCallNT*:  
 $\llbracket P, E, h \vdash e : NT; P, E, h \vdash es \text{ } [:] \text{ } Ts \rrbracket$   
 $\implies P, E, h \vdash e \cdot M(es) : T$
- | *WTrtBlock*:  
 $P, E(V \mapsto T), h \vdash e : T' \implies$   
 $P, E, h \vdash \{V:T; e\} : T'$
- | *WTrtSeq*:  
 $\llbracket P, E, h \vdash e_1 : T_1; P, E, h \vdash e_2 : T_2 \rrbracket$   
 $\implies P, E, h \vdash e_1 ; e_2 : T_2$
- | *WTrtCond*:  
 $\llbracket P, E, h \vdash e : \text{Boolean}; P, E, h \vdash e_1 : T_1; P, E, h \vdash e_2 : T_2;$   
 $P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1; P \vdash T_1 \leq T_2 \longrightarrow T = T_2; P \vdash T_2 \leq T_1 \longrightarrow T = T_1 \rrbracket$   
 $\implies P, E, h \vdash \text{if } (e) \ e_1 \ \text{else } e_2 : T$
- | *WTrtWhile*:  
 $\llbracket P, E, h \vdash e : \text{Boolean}; P, E, h \vdash c : T \rrbracket$   
 $\implies P, E, h \vdash \text{while}(e) \ c : \text{Void}$
- | *WTrtThrow*:  
 $\llbracket P, E, h \vdash e : T_r; \text{is-refT } T_r \rrbracket \implies$   
 $P, E, h \vdash \text{throw } e : T$
- | *WTrtTry*:  
 $\llbracket P, E, h \vdash e_1 : T_1; P, E(V \mapsto \text{Class } C), h \vdash e_2 : T_2; P \vdash T_1 \leq T_2 \rrbracket$   
 $\implies P, E, h \vdash \text{try } e_1 \ \text{catch}(C \ V) \ e_2 : T_2$
- well-typed expression lists
- | *WTrtNil*:  
 $P, E, h \vdash [] \text{ } [:] \text{ } []$
- | *WTrtCons*:  
 $\llbracket P, E, h \vdash e : T; P, E, h \vdash es \text{ } [:] \text{ } Ts \rrbracket$   
 $\implies P, E, h \vdash e \# es \text{ } [:] \text{ } T \# Ts$

### 2.17.1 Easy consequences

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

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

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

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

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

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

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

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

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

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

### 2.17.2 Some interesting lemmas

**lemma** *WTrts-Val*[simp]:

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

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

**lemma** *WTrt-env-mono*:

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

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

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

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

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

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

**end**

## 2.18 Definite assignment

theory *DefAss* imports *BigStep* begin

### 2.18.1 Hypersets

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

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

where

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

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

where

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

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

where

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

**definition** *hyper-isin* :: 'a  $\Rightarrow$  'a *hyperset*  $\Rightarrow$  *bool* (**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

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

**lemmas** *hyperset-defs* =

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

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

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

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

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

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

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

### 2.18.2 Definite assignment

**primrec**

*A* :: 'a *exp*  $\Rightarrow$  'a *hyperset*

**and** *As* :: 'a *exp list*  $\Rightarrow$  'a *hyperset*

where

$$\begin{aligned} \mathcal{A} (\text{new } C) &= [\{\}] \\ \mathcal{A} (\text{Cast } C \ e) &= \mathcal{A} \ e \\ \mathcal{A} (\text{Val } v) &= [\{\}] \\ \mathcal{A} (e_1 \ll\text{bop}\gg e_2) &= \mathcal{A} \ e_1 \sqcup \mathcal{A} \ e_2 \\ \mathcal{A} (\text{Var } V) &= [\{\}] \\ \mathcal{A} (\text{LAss } V \ e) &= [\{V\}] \sqcup \mathcal{A} \ e \\ \mathcal{A} (e \cdot F \{D\}) &= \mathcal{A} \ e \\ \mathcal{A} (e_1 \cdot F \{D\}; = e_2) &= \mathcal{A} \ e_1 \sqcup \mathcal{A} \ e_2 \end{aligned}$$

$\mathcal{A} (e \cdot M(es)) = \mathcal{A} e \sqcup \mathcal{A} s es$   
 $\mathcal{A} (\{V:T; e\}) = \mathcal{A} e \ominus V$   
 $\mathcal{A} (e_1;;e_2) = \mathcal{A} e_1 \sqcup \mathcal{A} e_2$   
 $\mathcal{A} (\text{if } (e) e_1 \text{ else } e_2) = \mathcal{A} e \sqcup (\mathcal{A} e_1 \sqcap \mathcal{A} e_2)$   
 $\mathcal{A} (\text{while } (b) e) = \mathcal{A} b$   
 $\mathcal{A} (\text{throw } e) = \text{None}$   
 $\mathcal{A} (\text{try } e_1 \text{ catch}(C V) e_2) = \mathcal{A} e_1 \sqcap (\mathcal{A} e_2 \ominus V)$

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

**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**

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

$\mathcal{D} s (\square) A = \text{True}$   
 $\mathcal{D} s (e\#es) A = (\mathcal{D} e A \wedge \mathcal{D} s es (A \sqcup \mathcal{A} e))$

**lemma** *As-map-Val[simp]*:  $\mathcal{A} s (\text{map Val } vs) = \{\square\}$  *<proof>*

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

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

**and**  $\bigwedge A. \mathcal{A} s es = \lfloor A \rfloor \implies A \subseteq \text{fvs } es$  *<proof>*

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

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

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

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

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

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

**end**

## 2.19 Conformance Relations for Type Soundness Proofs

**theory** *Conform*  
**imports** *Exceptions*  
**begin**

**definition** *conf* :: 'm prog ⇒ heap ⇒ val ⇒ ty ⇒ bool (·, · ⊢ · :≤ · [51,51,51,51] 50)

**where**

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

**definition** *oconf* :: 'm prog ⇒ heap ⇒ obj ⇒ bool (·, · ⊢ · √ [51,51,51] 50)

**where**

$P, h \vdash \text{obj } \checkmark \equiv$   
 $\text{let } (C, fs) = \text{obj in } \forall F D T. P \vdash C \text{ has } F:T \text{ in } D \longrightarrow$   
 $(\exists v. fs(F, D) = \text{Some } v \wedge P, h \vdash v : \leq T)$

**definition** *hconf* :: 'm prog ⇒ heap ⇒ bool (· ⊢ · √ [51,51] 50)

**where**

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

**definition** *lconf* :: 'm prog ⇒ heap ⇒ (vname → val) ⇒ (vname → ty) ⇒ bool (·, · ⊢ · '(≤) · [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 prog ⇒ heap ⇒ val list ⇒ ty list ⇒ bool  
 (·, · ⊢ · [≤] · [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 :≤

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

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

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

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

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

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

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

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

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

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

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

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

### 2.19.2 Value list conformance [≤]

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

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

**lemma** *confs-conv-map*:

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

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



**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) \langle proof \rangle$

### 2.19.3 Object conformance

**lemma** *oconf-hext*:  $P, h \vdash obj \checkmark \implies h \sqsubseteq h' \implies P, h' \vdash obj \checkmark \langle proof \rangle$

**lemma** *oconf-init-fields*:

$P \vdash C \text{ has-fields FDTs} \implies P, h \vdash (C, \text{init-fields FDTs}) \checkmark \langle proof \rangle$

**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, fs) \checkmark \rrbracket \implies P, h \vdash (C, fs((F, D) \mapsto v)) \checkmark \langle proof \rangle$

### 2.19.4 Heap conformance

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

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

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

### 2.19.5 Local variable conformance

**lemma** *lconf-hext*:  $\llbracket P, h \vdash l \[:\leq] E; h \sqsubseteq h' \rrbracket \implies P, h' \vdash l \[:\leq] E \langle proof \rangle$

**lemma** *lconf-upd*:

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

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

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

**end**

## 2.20 Progress of Small Step Semantics

**theory** *Progress*

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

**begin**

**lemma** *final-addrE*:

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

**lemma** *finalRefE*:

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

Derivation of new induction scheme for well typing:

**inductive**

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

**where**

$$\begin{aligned} & P, E, h \vdash e : ' T \equiv \text{WTrt}' P h E e T \\ & | P, E, h \vdash \text{es } [:\ ] Ts \equiv \text{WTrts}' P h E \text{es } Ts \\ \\ & | \text{is-class } P C \implies P, E, h \vdash \text{new } C : ' \text{Class } C \\ & | \llbracket P, E, h \vdash e : ' T; \text{is-refT } T; \text{is-class } P C \rrbracket \\ & \quad \implies P, E, h \vdash \text{Cast } C e : ' \text{Class } C \\ & | \text{typeof}_h v = \text{Some } T \implies P, E, h \vdash \text{Val } v : ' T \\ & | E v = \text{Some } T \implies P, E, h \vdash \text{Var } v : ' T \\ & | \llbracket P, E, h \vdash e_1 : ' T_1; P, E, h \vdash e_2 : ' T_2 \rrbracket \\ & \quad \implies P, E, h \vdash e_1 \ll \text{Eq} \gg e_2 : ' \text{Boolean} \\ & | \llbracket P, E, h \vdash e_1 : ' \text{Integer}; P, E, h \vdash e_2 : ' \text{Integer} \rrbracket \\ & \quad \implies P, E, h \vdash e_1 \ll \text{Add} \gg e_2 : ' \text{Integer} \\ & | \llbracket P, E, h \vdash \text{Var } V : ' T; P, E, h \vdash e : ' T'; P \vdash T' \leq T (* V \neq \text{This} *) \rrbracket \\ & \quad \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 \cdot F\{D\} : ' T \\ & | P, E, h \vdash e : ' NT \implies P, E, h \vdash e \cdot F\{D\} : ' T \\ & | \llbracket P, E, h \vdash e_1 : ' \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 \\ & \quad \implies P, E, h \vdash e_1 \cdot 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 \cdot 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 \text{es } [:\ ] Ts'; P \vdash Ts' [\leq] Ts \rrbracket \\ & \quad \implies P, E, h \vdash e \cdot M(\text{es}) : ' T \\ & | \llbracket P, E, h \vdash e : ' NT; P, E, h \vdash \text{es } [:\ ] Ts \rrbracket \implies P, E, h \vdash e \cdot M(\text{es}) : ' T \\ & | P, E, h \vdash \llbracket [:\ ] \rrbracket \\ & | \llbracket P, E, h \vdash e : ' T; P, E, h \vdash \text{es } [:\ ] Ts \rrbracket \implies P, E, h \vdash e \# \text{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 \end{aligned}$$

$$\begin{aligned} & \Longrightarrow P, E, h \vdash \{V:T := \text{Val } v; e_2\} : ' T_2 \\ | \llbracket P, E, h(V \mapsto T), h \vdash e : ' T'; \neg \text{assigned } V e \rrbracket & \Longrightarrow P, E, h \vdash \{V:T; e\} : ' T' \\ | \llbracket P, E, h \vdash e_1 : ' T_1; P, E, h \vdash e_2 : ' T_2 \rrbracket & \Longrightarrow P, E, h \vdash e_1; e_2 : ' T_2 \\ | \llbracket P, E, h \vdash e : ' \text{Boolean}; P, E, h \vdash e_1 : ' T_1; P, E, h \vdash e_2 : ' T_2; \\ & P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1; \\ & P \vdash T_1 \leq T_2 \longrightarrow T = T_2; P \vdash T_2 \leq T_1 \longrightarrow T = T_1 \rrbracket \\ & \Longrightarrow P, E, h \vdash \text{if } (e) e_1 \text{ else } e_2 : ' T \end{aligned}$$

$$\begin{aligned} | \llbracket P, E, h \vdash e : ' \text{Boolean}; P, E, h \vdash c : ' T \rrbracket \\ & \Longrightarrow P, E, h \vdash \text{while}(e) c : ' \text{Void} \\ | \llbracket P, E, h \vdash e : ' T_r; \text{is-ref } T T_r \rrbracket & \Longrightarrow P, E, h \vdash \text{throw } e : ' T \\ | \llbracket P, E, h \vdash e_1 : ' T_1; P, E, h(V \mapsto \text{Class } C), h \vdash e_2 : ' T_2; P \vdash T_1 \leq T_2 \rrbracket \\ & \Longrightarrow P, E, h \vdash \text{try } e_1 \text{ catch}(C V) e_2 : ' T_2 \end{aligned}$$

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

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

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

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

**and** *wts-wts'*:  $P, E, h \vdash \text{es } [:] Ts \Longrightarrow P, E, h \vdash \text{es } [:'] Ts \langle \text{proof} \rangle$

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

**and** *wts'-wts*:  $P, E, h \vdash \text{es } [:'] Ts \Longrightarrow P, E, h \vdash \text{es } [:] Ts \langle \text{proof} \rangle$

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

**corollary** *wts'-iff-wts*:  $(P, E, h \vdash \text{es } [:'] Ts) = (P, E, h \vdash \text{es } [:] Ts) \langle \text{proof} \rangle$

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

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

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

**and**  $P, E, h \vdash \text{es } [:] Ts \Longrightarrow$

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

**end**

## 2.21 Well-formedness Constraints

**theory** *JWellForm*

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

**begin**

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

**where**

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

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

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

**abbreviation**

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

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

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

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

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

**end**

## 2.22 Type Safety Proof

```
theory TypeSafe
imports Progress JWellForm
begin
```

### 2.22.1 Basic preservation lemmas

First two easy preservation lemmas.

**theorem** *red-preserves-hconf*:

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

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

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

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

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

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

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

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

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

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

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

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

Now preservation of definite assignment.

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

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

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

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

Combining conformance of heap and local variables:

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

**where**

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

**lemma** *red-preserves-sconf*:

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

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

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

### 2.22.2 Subject reduction

**lemma** *wt-blocks*:

$$\bigwedge E. \llbracket \text{length } Vs = \text{length } Ts; \text{length } vs = \text{length } Ts \rrbracket \Longrightarrow \\ (P, E, h \vdash \text{blocks } (Vs, Ts, vs, e) : T) = \\ (P, E (Vs [\rightarrow] Ts), h \vdash e : T \wedge (\exists Ts'. \text{map } (\text{typeof}_h) \ vs = \text{map } \text{Some } Ts' \wedge P \vdash Ts' [\leq] Ts)) \langle proof \rangle$$

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

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

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

$$\begin{aligned} &\implies \exists T'. P, E, h' \vdash e': T' \wedge P \vdash T' \leq T \\ \text{and } \textit{subjects-reduction2}: & P \vdash \langle es, (h, l) \rangle \rightarrow \langle es', (h', l') \rangle \implies \\ & (\wedge E \textit{ Ts}. \llbracket P, E \vdash (h, l) \checkmark; P, E, h \vdash es \text{ [:] } Ts \rrbracket \\ & \implies \exists Ts'. P, E, h' \vdash es' \text{ [:] } Ts' \wedge P \vdash Ts' \leq Ts \langle \textit{proof} \rangle \end{aligned}$$

**corollary** *subject-reduction*:

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

**corollary** *subjects-reduction*:

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

### 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**  $\wedge T. \llbracket P, E, hp \ s \vdash e : T; P, E \vdash s \checkmark \rrbracket \implies P, E \vdash s' \checkmark \langle \textit{proof} \rangle$

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

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

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

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

### 2.22.4 Lifting to $\Rightarrow$

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

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

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

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

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

### 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**  $P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies P, E, s \vdash e : T \checkmark$   
 $\implies \exists T'. P, E, s' \vdash e' : T' \checkmark \wedge P \vdash T' \leq T \langle \textit{proof} \rangle$

**theorem** *Subject-reductions*:

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

shows  $\bigwedge T. P, E, s \vdash e : T \checkmark \implies \exists T'. P, E, s' \vdash e' : T' \checkmark \wedge P \vdash T' \leq T \langle \text{proof} \rangle$

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

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

**corollary** *TypeSafety*:

$\llbracket \text{wf-J-prog } P; P, E \vdash s \checkmark; P, E \vdash e :: T; \mathcal{D} e \llbracket \text{dom}(lcl\ s) \rrbracket;$   
 $P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle; \neg(\exists e'' s''. P \vdash \langle e', s' \rangle \rightarrow \langle e'', s'' \rangle) \rrbracket$   
 $\implies (\exists v. e' = \text{Val } v \wedge P, hp\ s' \vdash v : \leq T) \vee$   
 $(\exists a. e' = \text{Throw } a \wedge a \in \text{dom}(hp\ s')) \langle \text{proof} \rangle$

**end**

## 2.23 Program annotation

theory *Annotate* imports *WellType* begin

inductive

*Anno* :: [*J-prog*, *env*, *expr* , *expr*]  $\Rightarrow$  *bool*  
 ( $\cdot, \cdot \vdash \cdot \rightsquigarrow \cdot$  - [51,0,0,51]50)

and *Annos* :: [*J-prog*, *env*, *expr list*, *expr list*]  $\Rightarrow$  *bool*  
 ( $\cdot, \cdot \vdash \cdot$  - [ $\rightsquigarrow$ ] - [51,0,0,51]50)

for *P* :: *J-prog*

where

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

| *AnnoCast*:  $P, E \vdash e \rightsquigarrow e' \Longrightarrow P, E \vdash \text{Cast } C \ e \rightsquigarrow \text{Cast } C \ e'$

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

| *AnnoVarVar*:  $E \ V = \lfloor T \rfloor \Longrightarrow P, E \vdash \text{Var } V \rightsquigarrow \text{Var } V$

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

| *AnnoBinOp*:

$\llbracket P, E \vdash e1 \rightsquigarrow e1'; P, E \vdash e2 \rightsquigarrow e2' \rrbracket$   
 $\Longrightarrow P, E \vdash e1 \ll bop \gg e2 \rightsquigarrow e1' \ll bop \gg e2'$

| *AnnoLAssVar*:

$\llbracket E \ V = \lfloor T \rfloor; P, E \vdash e \rightsquigarrow e' \rrbracket \Longrightarrow P, E \vdash V := e \rightsquigarrow V := e'$

| *AnnoLAssField*:

$\llbracket E \ V = \text{None}; E \ \text{this} = \lfloor \text{Class } C \rfloor; P \vdash C \ \text{sees } V:T \ \text{in } D; P, E \vdash e \rightsquigarrow e' \rrbracket$   
 $\Longrightarrow P, E \vdash V := e \rightsquigarrow \text{Var } \text{this} \cdot V \{D\} := e'$

| *AnnoFAcc*:

$\llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash e' :: \text{Class } C; P \vdash C \ \text{sees } F:T \ \text{in } D \rrbracket$   
 $\Longrightarrow P, E \vdash e \cdot F \{\} \rightsquigarrow e' \cdot F \{D\}$

| *AnnoFAss*:  $\llbracket P, E \vdash e1 \rightsquigarrow e1'; P, E \vdash e2 \rightsquigarrow e2';$   
 $P, E \vdash e1' :: \text{Class } C; P \vdash C \ \text{sees } F:T \ \text{in } D \rrbracket$   
 $\Longrightarrow P, E \vdash e1 \cdot F \{\} := e2 \rightsquigarrow e1' \cdot F \{D\} := e2'$

| *AnnoCall*:

$\llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash es \rightsquigarrow es' \rrbracket$   
 $\Longrightarrow P, E \vdash \text{Call } e \ M \ es \rightsquigarrow \text{Call } e' \ M \ es'$

| *AnnoBlock*:

$P, E(V \mapsto T) \vdash e \rightsquigarrow e' \Longrightarrow P, E \vdash \{V:T; e\} \rightsquigarrow \{V:T; e'\}$

| *AnnoComp*:  $\llbracket P, E \vdash e1 \rightsquigarrow e1'; P, E \vdash e2 \rightsquigarrow e2' \rrbracket$

$\Longrightarrow P, E \vdash e1 ;; e2 \rightsquigarrow e1' ;; e2'$

| *AnnoCond*:  $\llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash e1 \rightsquigarrow e1'; P, E \vdash e2 \rightsquigarrow e2' \rrbracket$

$\Longrightarrow P, E \vdash \text{if } (e) \ e1 \ \text{else } e2 \rightsquigarrow \text{if } (e') \ e1' \ \text{else } e2'$

| *AnnoLoop*:  $\llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash c \rightsquigarrow c' \rrbracket$

$\Longrightarrow P, E \vdash \text{while } (e) \ c \rightsquigarrow \text{while } (e') \ c'$

| *AnnoThrow*:  $P, E \vdash e \rightsquigarrow e' \Longrightarrow P, E \vdash \text{throw } e \rightsquigarrow \text{throw } e'$

| *AnnoTry*:  $\llbracket P, E \vdash e1 \rightsquigarrow e1'; P, E(V \mapsto \text{Class } C) \vdash e2 \rightsquigarrow e2' \rrbracket$

$\Longrightarrow P, E \vdash \text{try } e1 \ \text{catch}(C \ V) \ e2 \rightsquigarrow \text{try } e1' \ \text{catch}(C \ V) \ e2'$

| *AnnoNil*:  $P, E \vdash \llbracket \rightsquigarrow \rrbracket$

| *AnnoCons*:  $\llbracket P, E \vdash e \rightsquigarrow e'; P, E \vdash es \rightsquigarrow es' \rrbracket$

$\Longrightarrow P, E \vdash e \# es \rightsquigarrow e' \# es'$

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*

*~/src/HOL/Library/Code-Target-Numeral*

**begin**

**inductive** *map-val* :: *expr list*  $\Rightarrow$  *val list*  $\Rightarrow$  *bool*

**where**

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

| *Cons*: *map-val* *xs ys*  $\Longrightarrow$  *map-val* (*Val y # xs*) (*y # ys*)

**inductive** *map-val2* :: *expr list*  $\Rightarrow$  *val list*  $\Rightarrow$  *expr list*  $\Rightarrow$  *bool*

**where**

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

| *Cons*: *map-val2* *xs ys zs*  $\Longrightarrow$  *map-val2* (*Val y # xs*) (*y # ys*) *zs*

| *Throw*: *map-val2* (*throw e # xs*) [] (*throw e # xs*)

**theorem** *map-val-conv*: (*xs = map Val ys*) = *map-val* *xs ys*  $\langle$ *proof* $\rangle$

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

(*xs = map Val ys @ throw e # zs*) = *map-val2* *xs ys (throw e # zs)*  $\langle$ *proof* $\rangle$

**lemma** *CallNull2*:

$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle; P \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle evs, s_2 \rangle; \text{map-val } evs \text{ vs} \rrbracket$

$\Longrightarrow P \vdash \langle e \cdot M(ps), s_0 \rangle \Rightarrow \langle \text{THROW NullPointer}, s_2 \rangle$

$\langle$ *proof* $\rangle$

**lemma** *CallParamsThrow2*:

$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash \langle es, s_1 \rangle [\Rightarrow] \langle evs, s_2 \rangle;$

$\text{map-val2 } evs \text{ vs } (\text{throw } ex \# es'') \rrbracket$

$\Longrightarrow P \vdash \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle \text{throw } ex, s_2 \rangle$

$\langle$ *proof* $\rangle$

**lemma** *Call2*:

$\llbracket P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle evs, (h_2, l_2) \rangle;$

$\text{map-val } evs \text{ vs};$

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

$\text{length } vs = \text{length } pns; l_2' = [\text{this} \mapsto \text{Addr } a, pns \mapsto vs];$

$P \vdash \langle body, (h_2, l_2') \rangle \Rightarrow \langle e', (h_3, l_3) \rangle \rrbracket$

$\Longrightarrow P \vdash \langle e \cdot M(ps), s_0 \rangle \Rightarrow \langle e', (h_3, l_2) \rangle$

$\langle$ *proof* $\rangle$

**code-pred**

(*modes*: *i*  $\Rightarrow$  *o*  $\Rightarrow$  *bool*)

*map-val*

$\langle$ *proof* $\rangle$

**code-pred**

(*modes*: *i*  $\Rightarrow$  *o*  $\Rightarrow$  *o*  $\Rightarrow$  *bool*)

*map-val2*

$\langle$ *proof* $\rangle$

**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*⟩

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

**definition** *test1* =  $\square \vdash \langle \text{testExpr1}, (\text{empty}, \text{empty}) \rangle \Rightarrow \langle -, - \rangle$

**definition** *test2* =  $\square \vdash \langle \text{testExpr2}, (\text{empty}, \text{empty}) \rangle \Rightarrow \langle -, - \rangle$

**definition** *test3* =  $\square \vdash \langle \text{testExpr3}, (\text{empty}, \text{empty} ("V" \mapsto \text{Intg } 77)) \rangle \Rightarrow \langle -, - \rangle$

**definition** *test4* =  $\square \vdash \langle \text{testExpr4}, (\text{empty}, \text{empty}) \rangle \Rightarrow \langle -, - \rangle$

**definition** *test5* =  $[(\text{"Object"}, (\text{"", []}), (\text{"C"}, (\text{"Object"}, [(\text{"F"}, \text{Integer}), []]))] \vdash \langle \text{testExpr5}, (\text{empty}, \text{empty}) \rangle \Rightarrow \langle -, - \rangle$

**definition** *test6* =  $[(\text{"Object"}, (\text{"", []}), \text{classI}] \vdash \langle \text{testExpr6}, (\text{empty}, \text{empty}) \rangle \Rightarrow \langle -, - \rangle$

**definition** *V* = *"V"*

**definition** *C* = *"C"*

**definition** *F* = *"F"*

⟨*ML*⟩

**definition** *test7* =  $[\text{classObject}, \text{classL}] \vdash \langle \text{testExpr-BuildList}, (\text{empty}, \text{empty}) \rangle \Rightarrow \langle -, - \rangle$

**definition** *L* = *"L"*

**definition** *N* = *"N"*

$\langle ML \rangle$

**definition**  $test8 = [classObject, classA] \vdash \langle testExpr-ClassA, (empty, empty) \rangle \Rightarrow \langle -, - \rangle$

**definition**  $i = "int"$

**definition**  $t = "test"$

**definition**  $A = "A"$

$\langle ML \rangle$

**end**

## 2.26 Code Generation For WellType

**theory** *execute-WellType*

**imports**

*WellType Examples*

**begin**

**lemma** *WTCond1*:

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

**lemma** *WTCond2*:

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

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

**notation** *infer-type* ( $-, \vdash - :: 'l - [51, 51, 51] 100$ )

**definition** *test1* **where** *test1* =  $\llbracket, \text{empty} \vdash \text{testExpr1} :: -$

**definition** *test2* **where** *test2* =  $\llbracket, \text{empty} \vdash \text{testExpr2} :: -$

**definition** *test3* **where** *test3* =  $\llbracket, \text{empty} ("V" \mapsto \text{Integer}) \vdash \text{testExpr3} :: -$

**definition** *test4* **where** *test4* = [], *empty*("V"  $\mapsto$  Integer)  $\vdash$  *testExpr4* :: -

**definition** *test5* **where** *test5* = [classObject, ("C",("Object",("F",Integer),[]))], *empty*  $\vdash$  *testExpr5* :: -

**definition** *test6* **where** *test6* = [classObject, classI], *empty*  $\vdash$  *testExpr6* :: -

$\langle ML \rangle$

**definition** *testmb-isNull* **where** *testmb-isNull* = [classObject, classA], *empty*([this]  $\mapsto$  [Class "A"])  $\vdash$  *mb-isNull* :: -

**definition** *testmb-add* **where** *testmb-add* = [classObject, classA], *empty*([this,"i"]  $\mapsto$  [Class "A",Integer])  $\vdash$  *mb-add* :: -

**definition** *testmb-mult-cond* **where** *testmb-mult-cond* = [classObject, classA], *empty*([this,"j"]  $\mapsto$  [Class "A",Integer])  $\vdash$  *mb-mult-cond* :: -

**definition** *testmb-mult-block* **where** *testmb-mult-block* = [classObject, classA], *empty*([this,"i","j","temp"]  $\mapsto$  [Class "A",Integer,Integer,Integer])  $\vdash$  *mb-mult-block* :: -

**definition** *testmb-mult* **where** *testmb-mult* = [classObject, classA], *empty*([this,"i","j"]  $\mapsto$  [Class "A",Integer,Integer])  $\vdash$  *mb-mult* :: -

$\langle ML \rangle$

**definition** *test* **where** *test* = [classObject, classA], *empty*  $\vdash$  *testExpr-ClassA* :: -

$\langle ML \rangle$

**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* =  $pc \times pc \times cname \times pc \times nat$   
 — start-pc, end-pc, exception type, handler-pc, remaining stack depth

**type-synonym**

*ex-table* = *ex-entry list*

**type-synonym**

*jvm-method* =  $nat \times nat \times bytecode \times ex-table$   
 — 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* ⇒ (*Some* (*addr-of-sys-xcpt OutOfMemory*), *h*, (*stk*, *loc*, *C<sub>0</sub>*, *M<sub>0</sub>*, *pc*)#*frs*)  
   | *Some a* ⇒ (*None*, *h*(*a*→*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* ↦ (*D*, *fs*((*F,C*) ↦ *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* ¬*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,mxs,mxl<sub>0</sub>,ins,xt*) = *method P C M*;

$$f' = ([, [r]@\text{(rev ps)}@\text{(replicate mxl}_0 \text{ undefined)}, D, M, \theta)$$

$$\text{in } (xp', h, f'\#(\text{stk}, \text{loc}, C_0, M_0, \text{pc})\#frs))$$

$$\text{| exec-instr Return } P \text{ h stk}_0 \text{ loc}_0 \text{ C}_0 \text{ M}_0 \text{ pc frs} =$$

$$\text{(if frs}=[\text{ then (None, h, []) else}$$

$$\text{let } v = \text{hd stk}_0;$$

$$\text{(stk, loc, C, m, pc)} = \text{hd frs};$$

$$n = \text{length (fst (snd (method P C}_0 \text{ M}_0))$$

$$\text{in (None, h, (v\#(drop (n+1) stk), loc, C, m, pc+1)\#tl frs))$$

$$\text{| exec-instr Pop } P \text{ h stk loc C}_0 \text{ M}_0 \text{ pc frs} =$$

$$\text{(None, h, (tl stk, loc, C}_0 \text{, M}_0 \text{, pc+1)\#frs)}$$

$$\text{| exec-instr IAdd } P \text{ h stk loc C}_0 \text{ M}_0 \text{ pc frs} =$$

$$\text{(let } i_2 = \text{the-Intg (hd stk);}$$

$$i_1 = \text{the-Intg (hd (tl stk))}$$

$$\text{in (None, h, (Intg (i}_1+i_2)\#(\text{tl (tl stk)}), \text{loc, C}_0 \text{, M}_0 \text{, pc+1)\#frs))$$

$$\text{| exec-instr (IfFalse } i) P \text{ h stk loc C}_0 \text{ M}_0 \text{ pc frs} =$$

$$\text{(let pc' = if hd stk = Bool False then nat(int pc+i) else pc+1}$$

$$\text{in (None, h, (tl stk, loc, C}_0 \text{, M}_0 \text{, pc')\#frs))$$

$$\text{| exec-instr CmpEq } P \text{ h stk loc C}_0 \text{ M}_0 \text{ pc frs} =$$

$$\text{(let } v_2 = \text{hd stk};$$

$$v_1 = \text{hd (tl stk)}$$

$$\text{in (None, h, (Bool (v}_1=v_2) \# \text{tl (tl stk), loc, C}_0 \text{, M}_0 \text{, pc+1)\#frs))$$

$$\text{| exec-instr-Goto:}$$

$$\text{exec-instr (Goto } i) P \text{ h stk loc C}_0 \text{ M}_0 \text{ pc frs} =$$

$$\text{(None, h, (stk, loc, C}_0 \text{, M}_0 \text{, nat(int pc+i))\#frs)}$$

$$\text{| exec-instr Throw } P \text{ h stk loc C}_0 \text{ M}_0 \text{ pc frs} =$$

$$\text{(let xp' = if hd stk = Null then [addr-of-sys-xcpt NullPointer] else [the-Addr(hd stk)]}$$

$$\text{in (xp', h, (stk, loc, C}_0 \text{, M}_0 \text{, pc)\#frs))$$

**lemma** *exec-instr-Store:*

$$\text{exec-instr (Store } n) P \text{ h (v\#stk) loc C}_0 \text{ M}_0 \text{ pc frs} =$$

$$\text{(None, h, (stk, loc[n:=v], C}_0 \text{, M}_0 \text{, pc+1)\#frs)}$$

*<proof>*

**lemma** *exec-instr-Getfield:*

$$\text{exec-instr (Getfield } F \text{ C) } P \text{ h (v\#stk) loc C}_0 \text{ M}_0 \text{ pc frs} =$$

$$\text{(let xp' = if v=Null then [addr-of-sys-xcpt NullPointer] else None;$$

$$(D, fs) = \text{the(h(the-Addr v))}$$

$$\text{in (xp', h, (the(fs(F, C))\#stk, loc, C}_0 \text{, M}_0 \text{, pc+1)\#frs))}$$

*<proof>*

**lemma** *exec-instr-Putfield:*

$$\text{exec-instr (Putfield } F \text{ C) } P \text{ h (v\#r\#stk) loc C}_0 \text{ M}_0 \text{ pc frs} =$$

$$\text{(let xp' = if r=Null then [addr-of-sys-xcpt NullPointer] else None;$$

$$a = \text{the-Addr r};$$

$$(D, fs) = \text{the (h a)};$$

$$h' = \text{h(a} \mapsto \text{(D, fs((F, C) } \mapsto \text{v)))}$$

in ( $xp'$ ,  $h'$ , ( $stk$ ,  $loc$ ,  $C_0$ ,  $M_0$ ,  $pc+1$ )# $frs$ )  
 ⟨proof⟩

**lemma** *exec-instr-Checkcast*:

*exec-instr* (*Checkcast*  $C$ )  $P$   $h$  ( $v\#stk$ )  $loc$   $C_0$   $M_0$   $pc$   $frs$  =  
 (let  $xp' =$  if  $\neg$ cast-ok  $P$   $C$   $h$   $v$  then [addr-of-sys-xcpt *ClassCast*] else *None*  
 in ( $xp'$ ,  $h$ , ( $v\#stk$ ,  $loc$ ,  $C_0$ ,  $M_0$ ,  $pc+1$ )# $frs$ )  
 ⟨proof⟩

**lemma** *exec-instr-Return*:

*exec-instr* *Return*  $P$   $h$  ( $v\#stk_0$ )  $loc_0$   $C_0$   $M_0$   $pc$   $frs$  =  
 (if  $frs = []$  then (*None*,  $h$ , []) else  
 let ( $stk, loc, C, m, pc$ ) = *hd*  $frs$ ;  
      $n =$  length (*fst* (*snd* (*method*  $P$   $C_0$   $M_0$ )))  
 in (*None*,  $h$ , ( $v\#(drop$  ( $n+1$ )  $stk$ ),  $loc, C, m, pc+1$ )#*tl*  $frs$ )  
 ⟨proof⟩

**lemma** *exec-instr-IPop*:

*exec-instr* *Pop*  $P$   $h$  ( $v\#stk$ )  $loc$   $C_0$   $M_0$   $pc$   $frs$  =  
 (*None*,  $h$ , ( $stk$ ,  $loc$ ,  $C_0$ ,  $M_0$ ,  $pc+1$ )# $frs$ )  
 ⟨proof⟩

**lemma** *exec-instr-IAdd*:

*exec-instr* *IAdd*  $P$   $h$  (*Intg*  $i_2$  # *Intg*  $i_1$  #  $stk$ )  $loc$   $C_0$   $M_0$   $pc$   $frs$  =  
 (*None*,  $h$ , (*Intg* ( $i_1+i_2$ )# $stk$ ,  $loc$ ,  $C_0$ ,  $M_0$ ,  $pc+1$ )# $frs$ )  
 ⟨proof⟩

**lemma** *exec-instr-IfFalse*:

*exec-instr* (*IfFalse*  $i$ )  $P$   $h$  ( $v\#stk$ )  $loc$   $C_0$   $M_0$   $pc$   $frs$  =  
 (let  $pc' =$  if  $v =$  *Bool* *False* then *nat*(*int*  $pc+i$ ) else  $pc+1$   
 in (*None*,  $h$ , ( $stk$ ,  $loc$ ,  $C_0$ ,  $M_0$ ,  $pc'$ )# $frs$ )  
 ⟨proof⟩

**lemma** *exec-instr-CmpEq*:

*exec-instr* *CmpEq*  $P$   $h$  ( $v_2\#v_1\#stk$ )  $loc$   $C_0$   $M_0$   $pc$   $frs$  =  
 (*None*,  $h$ , (*Bool* ( $v_1=v_2$ ) #  $stk$ ,  $loc$ ,  $C_0$ ,  $M_0$ ,  $pc+1$ )# $frs$ )  
 ⟨proof⟩

**lemma** *exec-instr-Throw*:

*exec-instr* *Throw*  $P$   $h$  ( $v\#stk$ )  $loc$   $C_0$   $M_0$   $pc$   $frs$  =  
 (let  $xp' =$  if  $v =$  *Null* then [addr-of-sys-xcpt *NullPointer*] else [*the-Addr*  $v$ ]  
 in ( $xp'$ ,  $h$ , ( $v\#stk$ ,  $loc$ ,  $C_0$ ,  $M_0$ ,  $pc$ )# $frs$ )  
 ⟨proof⟩

**end**

### 3.4 Exception handling in the JVM

**theory** *JVMExceptions* **imports** *JVMInstructions* *../Common/Exceptions* **begin**

**definition** *matches-ex-entry* :: 'm prog ⇒ cname ⇒ pc ⇒ ex-entry ⇒ bool

**where**

*matches-ex-entry* P C pc xcp ≡  
 let (s, e, C', h, d) = xcp in  
 s ≤ pc ∧ pc < e ∧ P ⊢ C ≤\* C'

**primrec** *match-ex-table* :: 'm prog ⇒ cname ⇒ pc ⇒ ex-table ⇒ (pc × nat) option

**where**

*match-ex-table* P C pc [] = None  
 | *match-ex-table* P C pc (e#es) = (if *matches-ex-entry* P C pc e  
 then Some (snd(snd(snd e)))  
 else *match-ex-table* P C pc es)

**abbreviation**

*ex-table-of* :: jvm-prog ⇒ cname ⇒ mname ⇒ ex-table **where**  
*ex-table-of* P C M == snd (snd (snd (snd (snd (snd (method P C M))))))

**primrec** *find-handler* :: jvm-prog ⇒ addr ⇒ heap ⇒ frame list ⇒ 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 ⇒ *find-handler* P a h frs  
 | Some pc-d ⇒ (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
  (- | / - -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  $\sigma'$   $\implies$  P  $\vdash$   $\sigma$  -jvm $\rightarrow_1$   $\sigma'$ 

— reflexive transitive closure:
definition exec-all :: jvm-prog  $\Rightarrow$  jvm-state  $\Rightarrow$  jvm-state  $\Rightarrow$  bool
  (- | - / - -jvm $\rightarrow$ / - [61,61,61]60) where

  exec-all-def1: P | -  $\sigma$  -jvm $\rightarrow$   $\sigma'$   $\iff$  ( $\sigma$ , $\sigma'$ )  $\in$  (exec-1 P)*

notation (xsymbols)
  exec-all ((- | / - -jvm $\rightarrow$ / -) [61,61,61]60)

lemma exec-1-eq:
  exec-1 P = {( $\sigma$ , $\sigma'$ ). exec (P, $\sigma$ ) = Some  $\sigma'$ } $\langle$ proof $\rangle$ 
lemma exec-1-iff:
  P  $\vdash$   $\sigma$  -jvm $\rightarrow_1$   $\sigma'$  = (exec (P, $\sigma$ ) = Some  $\sigma'$ ) $\langle$ proof $\rangle$ 
lemma exec-all-def:
  P  $\vdash$   $\sigma$  -jvm $\rightarrow$   $\sigma'$  = (( $\sigma$ , $\sigma'$ )  $\in$  {( $\sigma$ , $\sigma'$ ). exec (P, $\sigma$ ) = Some  $\sigma'$ }*) $\langle$ proof $\rangle$ 
lemma jvm-refl[iff]: P  $\vdash$   $\sigma$  -jvm $\rightarrow$   $\sigma$  $\langle$ proof $\rangle$ 
lemma jvm-trans[trans]:
  [| P  $\vdash$   $\sigma$  -jvm $\rightarrow$   $\sigma'$ ; P  $\vdash$   $\sigma'$  -jvm $\rightarrow$   $\sigma''$  |]  $\implies$  P  $\vdash$   $\sigma$  -jvm $\rightarrow$   $\sigma''$  $\langle$ proof $\rangle$ 
lemma jvm-one-step1[trans]:
  [| P  $\vdash$   $\sigma$  -jvm $\rightarrow_1$   $\sigma'$ ; P  $\vdash$   $\sigma'$  -jvm $\rightarrow$   $\sigma''$  |]  $\implies$  P  $\vdash$   $\sigma$  -jvm $\rightarrow$   $\sigma''$  $\langle$ proof $\rangle$ 

```



**lemma** *jvm-one-step2[trans]*:

$$\llbracket P \vdash \sigma \text{-jvm} \rightarrow \sigma'; P \vdash \sigma' \text{-jvm} \rightarrow_1 \sigma'' \rrbracket \Longrightarrow P \vdash \sigma \text{-jvm} \rightarrow \sigma'' \langle \text{proof} \rangle$$

**lemma** *exec-all-conf*:

$$\llbracket P \vdash \sigma \text{-jvm} \rightarrow \sigma'; P \vdash \sigma \text{-jvm} \rightarrow \sigma'' \rrbracket \\ \Longrightarrow P \vdash \sigma' \text{-jvm} \rightarrow \sigma'' \vee P \vdash \sigma'' \text{-jvm} \rightarrow \sigma' \langle \text{proof} \rangle$$

**lemma** *exec-all-finalD*:  $P \vdash (x, h, []) \text{-jvm} \rightarrow \sigma \Longrightarrow \sigma = (x, h, []) \langle \text{proof} \rangle$

**lemma** *exec-all-deterministic*:

$$\llbracket P \vdash \sigma \text{-jvm} \rightarrow (x, h, []); P \vdash \sigma \text{-jvm} \rightarrow \sigma' \rrbracket \Longrightarrow P \vdash \sigma' \text{-jvm} \rightarrow (x, h, []) \langle \text{proof} \rangle$$

The start configuration of the JVM: in the start heap, we call a method  $m$  of class  $C$  in program  $P$ . The *this* pointer of the frame is set to *Null* to simulate a static method invocation.

**definition** *start-state* :: *jvm-prog*  $\Rightarrow$  *cname*  $\Rightarrow$  *mname*  $\Rightarrow$  *jvm-state* **where**

*start-state*  $P C M =$

(*let* ( $D, Ts, T, mxs, mxl_0, b$ ) = *method*  $P C M$  *in*

(*None*, *start-heap*  $P$ ,  $[([], \text{Null} \# \text{replicate } mxl_0 \text{ undefined}, C, M, 0)]$ ))

**end**

### 3.6 A Defensive JVM

```
theory JVMDefensive
imports JVMExec ../Common/Conform
begin
```

Extend the state space by one element indicating a type error (or other abnormal termination)

```
datatype 'a type-error = TypeError | Normal 'a
```

```
fun is-Addr :: val  $\Rightarrow$  bool where
  is-Addr (Addr a)  $\longleftrightarrow$  True
| is-Addr v  $\longleftrightarrow$  False
```

```
fun is-Intg :: val  $\Rightarrow$  bool where
  is-Intg (Intg i)  $\longleftrightarrow$  True
| is-Intg v  $\longleftrightarrow$  False
```

```
fun is-Bool :: val  $\Rightarrow$  bool where
  is-Bool (Bool b)  $\longleftrightarrow$  True
| is-Bool v  $\longleftrightarrow$  False
```

```
definition is-Ref :: val  $\Rightarrow$  bool where
  is-Ref v  $\longleftrightarrow$  v = Null  $\vee$  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 M0 pc frs =
    (n < length loc)
```

```
| check-instr-Store:
```

```
  check-instr (Store n) P h stk loc C0 M0 pc frs =
    (0 < length stk  $\wedge$  n < length loc)
```

```
| check-instr-Push:
```

```
  check-instr (Push v) P h stk loc C0 M0 pc frs =
    ( $\neg$ is-Addr v)
```

```
| check-instr-New:
```

```
  check-instr (New C) P h stk loc C0 M0 pc frs =
    is-class P C
```

```
| check-instr-Getfield:
```

```
  check-instr (Getfield F C) P h stk loc C0 M0 pc frs =
    (0 < length stk  $\wedge$  ( $\exists$  C' T. P  $\vdash$  C sees F:T in C')  $\wedge$ 
    (let (C', T) = field P C F; ref = hd stk in
     C' = C  $\wedge$  is-Ref ref  $\wedge$  (ref  $\neq$  Null  $\longrightarrow$ 
     h (the-Addr ref)  $\neq$  None  $\wedge$ 
     (let (D, vs) = the (h (the-Addr ref)) in
      P  $\vdash$  D  $\preceq^*$  C  $\wedge$  vs (F, C)  $\neq$  None  $\wedge$  P, h  $\vdash$  the (vs (F, C))  $:\leq$  T))))
```

```
| check-instr-Putfield:
```

```
  check-instr (Putfield F C) P h stk loc C0 M0 pc frs =
```

$$\begin{aligned}
& (1 < \text{length } \text{stk} \wedge (\exists C' T. P \vdash C \text{ sees } F:T \text{ in } C') \wedge \\
& (\text{let } (C', T) = \text{field } P C F; v = \text{hd } \text{stk}; \text{ref} = \text{hd } (\text{tl } \text{stk}) \text{ in} \\
& \quad C' = C \wedge \text{is-Ref } \text{ref} \wedge (\text{ref} \neq \text{Null} \longrightarrow \\
& \quad \quad h (\text{the-Addr } \text{ref}) \neq \text{None} \wedge \\
& \quad \quad (\text{let } D = \text{fst } (\text{the } (h (\text{the-Addr } \text{ref})))) \text{ in} \\
& \quad \quad P \vdash D \preceq^* C \wedge P, h \vdash v : \leq T)))
\end{aligned}$$

| *check-instr-Checkcast*:

$$\begin{aligned}
& \text{check-instr } (\text{Checkcast } C) P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} = \\
& (0 < \text{length } \text{stk} \wedge \text{is-class } P C \wedge \text{is-Ref } (\text{hd } \text{stk}))
\end{aligned}$$

| *check-instr-Invoke*:

$$\begin{aligned}
& \text{check-instr } (\text{Invoke } M n) P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} = \\
& (n < \text{length } \text{stk} \wedge \text{is-Ref } (\text{stk}!n) \wedge \\
& (\text{stk}!n \neq \text{Null} \longrightarrow \\
& \quad (\text{let } a = \text{the-Addr } (\text{stk}!n); \\
& \quad \quad C = \text{cname-of } h a; \\
& \quad \quad Ts = \text{fst } (\text{snd } (\text{method } P C M)) \\
& \quad \text{in } h a \neq \text{None} \wedge P \vdash C \text{ has } M \wedge \\
& \quad P, h \vdash \text{rev } (\text{take } n \text{ stk}) [:\leq] Ts)))
\end{aligned}$$

| *check-instr-Return*:

$$\begin{aligned}
& \text{check-instr } \text{Return } P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} = \\
& (0 < \text{length } \text{stk} \wedge ((0 < \text{length } \text{frs}) \longrightarrow \\
& \quad (P \vdash C_0 \text{ has } M_0) \wedge \\
& \quad (\text{let } v = \text{hd } \text{stk}; \\
& \quad \quad T = \text{fst } (\text{snd } (\text{snd } (\text{method } P C_0 M_0))) \\
& \quad \text{in } P, h \vdash v : \leq T)))
\end{aligned}$$

| *check-instr-Pop*:

$$\begin{aligned}
& \text{check-instr } \text{Pop } P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} = \\
& (0 < \text{length } \text{stk})
\end{aligned}$$

| *check-instr-IAdd*:

$$\begin{aligned}
& \text{check-instr } \text{IAdd } P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} = \\
& (1 < \text{length } \text{stk} \wedge \text{is-Intg } (\text{hd } \text{stk}) \wedge \text{is-Intg } (\text{hd } (\text{tl } \text{stk})))
\end{aligned}$$

| *check-instr-IfFalse*:

$$\begin{aligned}
& \text{check-instr } (\text{IfFalse } b) P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} = \\
& (0 < \text{length } \text{stk} \wedge \text{is-Bool } (\text{hd } \text{stk}) \wedge 0 \leq \text{int } \text{pc}+b)
\end{aligned}$$

| *check-instr-CmpEq*:

$$\begin{aligned}
& \text{check-instr } \text{CmpEq } P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} = \\
& (1 < \text{length } \text{stk})
\end{aligned}$$

| *check-instr-Goto*:

$$\begin{aligned}
& \text{check-instr } (\text{Goto } b) P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} = \\
& (0 \leq \text{int } \text{pc}+b)
\end{aligned}$$

| *check-instr-Throw*:

$$\begin{aligned}
& \text{check-instr } \text{Throw } P h \text{ stk } \text{loc } C_0 M_0 \text{ pc } \text{frs} = \\
& (0 < \text{length } \text{stk} \wedge \text{is-Ref } (\text{hd } \text{stk}))
\end{aligned}$$

**definition** *check* :: *jvm-prog*  $\Rightarrow$  *jvm-state*  $\Rightarrow$  *bool* **where**

$check\ P\ \sigma = (let\ (xcpt,\ h,\ frs) = \sigma\ in$   
 $(case\ frs\ of\ [] \Rightarrow True\ |\ (stk,\ loc,\ C,\ M,\ pc)\#frs' \Rightarrow$   
 $P \vdash C\ has\ M \wedge$   
 $(let\ (C',\ Ts,\ T,\ mxs,\ mxl_0,\ ins,\ xt) = method\ P\ C\ M;\ i = ins!pc\ in$   
 $pc < size\ ins \wedge size\ stk \leq mxs \wedge$   
 $check\ instr\ i\ P\ h\ stk\ loc\ C\ M\ pc\ frs'))$

**definition**  $exec\text{-}d :: jvm\text{-}prog \Rightarrow jvm\text{-}state \Rightarrow jvm\text{-}state\ option\ type\text{-}error$  **where**  
 $exec\text{-}d\ P\ \sigma = (if\ check\ P\ \sigma\ then\ Normal\ (exec\ (P,\ \sigma))\ else\ TypeError)$

**inductive-set**

$exec\text{-}1\text{-}d :: jvm\text{-}prog \Rightarrow (jvm\text{-}state\ type\text{-}error \times jvm\text{-}state\ type\text{-}error)\ set$   
**and**  $exec\text{-}1\text{-}d' :: jvm\text{-}prog \Rightarrow jvm\text{-}state\ type\text{-}error \Rightarrow jvm\text{-}state\ type\text{-}error \Rightarrow bool$   
 $(-\vdash-\ -jvmd\rightarrow_1 - [61,61,61]60)$

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

**where**

$P \vdash \sigma \text{-}jvmd\rightarrow_1 \sigma' \equiv (\sigma, \sigma') \in exec\text{-}1\text{-}d\ P$   
 $| exec\text{-}1\text{-}d\text{-}ErrorI: exec\text{-}d\ P\ \sigma = TypeError \implies P \vdash Normal\ \sigma \text{-}jvmd\rightarrow_1\ TypeError$   
 $| exec\text{-}1\text{-}d\text{-}NormalI: exec\text{-}d\ P\ \sigma = Normal\ (Some\ \sigma') \implies P \vdash Normal\ \sigma \text{-}jvmd\rightarrow_1\ Normal\ \sigma'$

— reflexive transitive closure:

**definition**  $exec\text{-}all\text{-}d :: jvm\text{-}prog \Rightarrow jvm\text{-}state\ type\text{-}error \Rightarrow jvm\text{-}state\ type\text{-}error \Rightarrow bool$   
 $(-\vdash-\ -jvmd\rightarrow - [61,61,61]60)$  **where**  
 $exec\text{-}all\text{-}d\text{-}def1: P \vdash \sigma \text{-}jvmd\rightarrow \sigma' \iff (\sigma, \sigma') \in (exec\text{-}1\text{-}d\ P)^*$

**notation** ( $xsymbols$ )

$exec\text{-}all\text{-}d\ (-\vdash-\ -jvmd\rightarrow - [61,61,61]60)$

**lemma**  $exec\text{-}1\text{-}d\text{-}eq$ :

$exec\text{-}1\text{-}d\ P = \{(s,t). \exists \sigma. s = Normal\ \sigma \wedge t = TypeError \wedge exec\text{-}d\ P\ \sigma = TypeError\} \cup$   
 $\{(s,t). \exists \sigma\ \sigma'. s = Normal\ \sigma \wedge t = Normal\ \sigma' \wedge exec\text{-}d\ P\ \sigma = Normal\ (Some\ \sigma')\}$

$\langle proof \rangle$

**declare**  $split\text{-}paired\text{-}All [simp\ del]$

**declare**  $split\text{-}paired\text{-}Ex [simp\ del]$

**lemma**  $if\text{-}neq [dest!]$ :

$(if\ P\ then\ A\ else\ B) \neq B \implies P$   
 $\langle proof \rangle$

**lemma**  $exec\text{-}d\text{-}no\text{-}errorI [intro]$ :

$check\ P\ \sigma \implies exec\text{-}d\ P\ \sigma \neq TypeError$   
 $\langle proof \rangle$

**theorem**  $no\text{-}type\text{-}error\text{-}commutes$ :

$exec\text{-}d\ P\ \sigma \neq TypeError \implies exec\text{-}d\ P\ \sigma = Normal\ (exec\ (P,\ \sigma))$   
 $\langle proof \rangle$

**lemma**  $defensive\text{-}imp\text{-}aggressive$ :

$P \vdash (Normal\ \sigma) \text{-}jvmd\rightarrow (Normal\ \sigma') \implies P \vdash \sigma \text{-}jvm\rightarrow \sigma' \langle proof \rangle$

**end**

### 3.7 Example for generating executable code from JVM semantics

```

theory JVMListExample
imports
  ../Common/SystemClasses
  JVMExec
  ~/src/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, NullPointer, 7, 0))])])
```

**definition** *make-list-ins* :: *bytecode*

**where**

```
make-list-ins ==
  [New list-name,
   Store 0,
   Load 0,
   Push (Intg 1),
   Putfield val-name list-name,
   New list-name,
   Store 1,
   Load 1,
   Push (Intg 2),
   Putfield val-name list-name,
   New list-name,
   Store 2,
   Load 2,
   Push (Intg 3),
   Putfield val-name list-name,
   Load 0,
   Load 1,
   Invoke append-name 1,
   Pop,
   Load 0,
   Load 2,
   Invoke append-name 1,
   Return]
```

**definition** *test-class* :: *jvm-method class*

**where**

```
test-class ==
  (Object, [],
   [(makelist-name, [], Void, (3, 2, make-list-ins, []))])
```

**definition** *E* :: *jvm-prog*

**where**

```
E == SystemClasses @ [(list-name, list-class), (test-name, test-class)]
```

**definition** *undefined-cname* :: *cname*

**where** [code del]: *undefined-cname* = *undefined*

**declare** *undefined-cname-def*[*symmetric, code-unfold*]

**code-const** *undefined-cname*

(*SML object*)

**definition** *undefined-val* :: *val*

**where** [code del]: *undefined-val* = *undefined*

**declare** *undefined-val-def*[*symmetric, code-unfold*]

**code-const** *undefined-val*  
(*SML Unit*)

**lemmas** [*code-unfold*] = *SystemClasses-def* [*unfolded ObjectC-def NullPointerC-def ClassCastC-def OutOfMemoryC-def*]

**definition** *test* = *exec* (*E*, *start-state E test-name makelist-name*)

⟨*ML*⟩

**end**



## Chapter 4

# Bytecode Verifier

## 4.1 Semilattices

**theory** *Semilat*

**imports** *Main*  $\sim\sim$  /src/HOL/Library/While-Combinator

**begin**

**type-synonym** *'a ord* = *'a*  $\Rightarrow$  *'a*  $\Rightarrow$  *bool*

**type-synonym** *'a binop* = *'a*  $\Rightarrow$  *'a*  $\Rightarrow$  *'a*

**type-synonym** *'a sl* = *'a set*  $\times$  *'a ord*  $\times$  *'a binop*

**consts**

*lesub* :: *'a*  $\Rightarrow$  *'a ord*  $\Rightarrow$  *'a*  $\Rightarrow$  *bool*

*lesssub* :: *'a*  $\Rightarrow$  *'a ord*  $\Rightarrow$  *'a*  $\Rightarrow$  *bool*

*plussub* :: *'a*  $\Rightarrow$  (*'a*  $\Rightarrow$  *'b*  $\Rightarrow$  *'c*)  $\Rightarrow$  *'b*  $\Rightarrow$  **'cnotation** (*xsymbols*)

*lesub* ((- / $\sqsubseteq$ -) [50, 0, 51] 50) **and**

*lesssub* ((- / $\sqsubset$ -) [50, 0, 51] 50) **and**

*plussub* ((- / $\sqcup$ -) [65, 0, 66] 65)

**defs**

*lesub-def*:  $x \sqsubseteq_r y \equiv r x y$

*lesssub-def*:  $x \sqsubset_r y \equiv x \sqsubseteq_r y \wedge x \neq y$

*plussub-def*:  $x \sqcup_f y \equiv f x y$

**definition** *ord* :: (*'a*  $\times$  *'a*) *set*  $\Rightarrow$  *'a ord*

**where**

*ord* *r* = ( $\lambda x y. (x,y) \in r$ )

**definition** *order* :: *'a ord*  $\Rightarrow$  *bool*

**where**

*order* *r*  $\longleftrightarrow (\forall x. x \sqsubseteq_r x) \wedge (\forall x y. x \sqsubseteq_r y \wedge y \sqsubseteq_r x \longrightarrow x=y) \wedge (\forall x y z. x \sqsubseteq_r y \wedge y \sqsubseteq_r z \longrightarrow x \sqsubseteq_r z)$

**definition** *top* :: *'a ord*  $\Rightarrow$  *'a*  $\Rightarrow$  *bool*

**where**

*top* *r* *T*  $\longleftrightarrow (\forall x. x \sqsubseteq_r T)$

**definition** *acc* :: *'a ord*  $\Rightarrow$  *bool*

**where**

*acc* *r*  $\longleftrightarrow wf \{(y,x). x \sqsubseteq_r y\}$

**definition** *closed* :: *'a set*  $\Rightarrow$  *'a binop*  $\Rightarrow$  *bool*

**where**

*closed* *A* *f*  $\longleftrightarrow (\forall x \in A. \forall y \in A. x \sqcup_f y \in A)$

**definition** *semilat* :: *'a sl*  $\Rightarrow$  *bool*

**where**

*semilat* = ( $\lambda(A,r,f). order r \wedge closed A f \wedge$   
 $(\forall x \in A. \forall y \in A. x \sqsubseteq_r x \sqcup_f y) \wedge$   
 $(\forall x \in A. \forall y \in A. y \sqsubseteq_r x \sqcup_f y) \wedge$   
 $(\forall x \in A. \forall y \in A. \forall z \in A. x \sqsubseteq_r z \wedge y \sqsubseteq_r z \longrightarrow x \sqcup_f y \sqsubseteq_r z)$ )

**definition** *is-ub* :: (*'a*  $\times$  *'a*) *set*  $\Rightarrow$  *'a*  $\Rightarrow$  *'a*  $\Rightarrow$  *'a*  $\Rightarrow$  *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* ⇒  $x \sqsubseteq_r x$

*<proof>*

**lemma** *order-antisym*:  $\llbracket order\ r; x \sqsubseteq_r\ y; y \sqsubseteq_r\ x \rrbracket \Longrightarrow x = y$

*<proof>*

**lemma** *order-trans*:  $\llbracket order\ r; x \sqsubseteq_r\ y; y \sqsubseteq_r\ z \rrbracket \Longrightarrow x \sqsubseteq_r\ z$

*<proof>*

**lemma** *order-less-irrefl* [*intro*, *simp*]: *order r* ⇒  $\neg x \sqsubseteq_r x$

*<proof>*

**lemma** *order-less-trans*:  $\llbracket order\ r; x \sqsubseteq_r\ y; y \sqsubseteq_r\ z \rrbracket \Longrightarrow x \sqsubseteq_r\ z$

*<proof>*

**lemma** *topD* [*simp*, *intro*]: *top r T* ⇒  $x \sqsubseteq_r T$

*<proof>*

**lemma** *top-le-conv* [*simp*]:  $\llbracket order\ r; top\ r\ T \rrbracket \Longrightarrow (T \sqsubseteq_r x) = (x = T)$

*<proof>*

**lemma** *semilat-Def*:

$semilat(A,r,f) \longleftrightarrow order\ r \wedge closed\ A\ f \wedge$

$(\forall x \in A. \forall y \in A. x \sqsubseteq_r x \sqcup_f y) \wedge$

$(\forall x \in A. \forall y \in A. y \sqsubseteq_r x \sqcup_f y) \wedge$

$(\forall x \in A. \forall y \in A. \forall z \in A. x \sqsubseteq_r z \wedge y \sqsubseteq_r z \longrightarrow x \sqcup_f y \sqsubseteq_r z)$

*<proof>*

**lemma** (**in** *Semilat*) *orderI* [*simp*, *intro*]: *order r*

*<proof>*

**lemma** (**in** *Semilat*) *closedI* [*simp*, *intro*]: *closed A f*

*<proof>*

**lemma** *closedD*:  $\llbracket closed\ A\ f; x \in A; y \in A \rrbracket \Longrightarrow x \sqcup_f y \in A$

*<proof>*

**lemma** *closed-UNIV* [*simp*]: *closed UNIV f*

*<proof>*

**lemma** (**in** *Semilat*) *closed-f* [*simp*, *intro*]:  $\llbracket x \in A; y \in A \rrbracket \Longrightarrow x \sqcup_f y \in A$

*<proof>*

**lemma** (**in** *Semilat*) *refl-r* [*intro*, *simp*]:  $x \sqsubseteq_r x$  *<proof>*

**lemma** (**in** *Semilat*) *antisym-r* [*intro?*]:  $\llbracket x \sqsubseteq_r\ y; y \sqsubseteq_r\ x \rrbracket \Longrightarrow x = y$

*<proof>*

**lemma** (**in** *Semilat*) *trans-r* [*trans*, *intro?*]:  $\llbracket x \sqsubseteq_r\ y; y \sqsubseteq_r\ z \rrbracket \Longrightarrow x \sqsubseteq_r\ z$

*<proof>*

**lemma** (**in** *Semilat*) *ub1* [*simp*, *intro?*]:  $\llbracket x \in A; y \in A \rrbracket \Longrightarrow x \sqsubseteq_r x \sqcup_f y$

*<proof>*

**lemma** (**in** *Semilat*) *ub2* [*simp*, *intro?*]:  $\llbracket x \in A; y \in A \rrbracket \Longrightarrow y \sqsubseteq_r x \sqcup_f y$

*<proof>*

**lemma** (in *Semilat*) *lub* [*simp*, *intro?*]:

$\llbracket x \sqsubseteq_r z; y \sqsubseteq_r z; x \in A; y \in A; z \in A \rrbracket \implies x \sqcup_f y \sqsubseteq_r z$   
 ⟨*proof*⟩

**lemma** (in *Semilat*) *plus-le-conv* [*simp*]:

$\llbracket x \in A; y \in A; z \in A \rrbracket \implies (x \sqcup_f y \sqsubseteq_r z) = (x \sqsubseteq_r z \wedge y \sqsubseteq_r z)$   
 ⟨*proof*⟩

**lemma** (in *Semilat*) *le-iff-plus-unchanged*:

assumes  $x \in A$  and  $y \in A$

shows  $x \sqsubseteq_r y \iff x \sqcup_f y = y$  (is  $?P \iff ?Q$ )⟨*proof*⟩

**lemma** (in *Semilat*) *le-iff-plus-unchanged2*:

assumes  $x \in A$  and  $y \in A$

shows  $x \sqsubseteq_r y \iff y \sqcup_f x = y$  (is  $?P \iff ?Q$ )⟨*proof*⟩

**lemma** (in *Semilat*) *plus-assoc* [*simp*]:

assumes  $a: a \in A$  and  $b: b \in A$  and  $c: c \in A$

shows  $a \sqcup_f (b \sqcup_f c) = a \sqcup_f b \sqcup_f c$ ⟨*proof*⟩

**lemma** (in *Semilat*) *plus-com-lemma*:

$\llbracket a \in A; b \in A \rrbracket \implies a \sqcup_f b \sqsubseteq_r b \sqcup_f a$ ⟨*proof*⟩

**lemma** (in *Semilat*) *plus-commutative*:

$\llbracket a \in A; b \in A \rrbracket \implies a \sqcup_f b = b \sqcup_f a$   
 ⟨*proof*⟩

**lemma** *is-lubD*:

$is-lub\ r\ x\ y\ u \implies is-ub\ r\ x\ y\ u \wedge (\forall z. is-ub\ r\ x\ y\ z \longrightarrow (u, z) \in r)$   
 ⟨*proof*⟩

**lemma** *is-ubI*:

$\llbracket (x, u) \in r; (y, u) \in r \rrbracket \implies is-ub\ r\ x\ y\ u$   
 ⟨*proof*⟩

**lemma** *is-ubD*:

$is-ub\ r\ x\ y\ u \implies (x, u) \in r \wedge (y, u) \in r$   
 ⟨*proof*⟩

**lemma** *is-lub-bigger1* [*iff*]:

$is-lub\ (r^{\hat{*}})\ x\ y\ y = ((x, y) \in r^{\hat{*}})$ ⟨*proof*⟩

**lemma** *is-lub-bigger2* [*iff*]:

$is-lub\ (r^{\hat{*}})\ x\ y\ x = ((y, x) \in r^{\hat{*}})$ ⟨*proof*⟩

**lemma** *extend-lub*:

$\llbracket single-valued\ r; is-lub\ (r^{\hat{*}})\ x\ y\ u; (x', x) \in r \rrbracket$   
 $\implies EX\ v. is-lub\ (r^{\hat{*}})\ x'\ y\ v$ ⟨*proof*⟩

**lemma** *single-valued-has-lubs* [*rule-format*]:

$\llbracket single-valued\ r; (x, u) \in r^{\hat{*}} \rrbracket \implies (\forall y. (y, u) \in r^{\hat{*}} \longrightarrow$   
 $(EX\ z. is-lub\ (r^{\hat{*}})\ x\ y\ z))$ ⟨*proof*⟩

**lemma** *some-lub-conv*:

$\llbracket acyclic\ r; is-lub\ (r^{\hat{*}})\ x\ y\ u \rrbracket \implies some-lub\ (r^{\hat{*}})\ x\ y = u$ ⟨*proof*⟩

**lemma** *is-lub-some-lub*:

$\llbracket single-valued\ r; acyclic\ r; (x, u) \in r^{\hat{*}}; (y, u) \in r^{\hat{*}} \rrbracket$   
 $\implies is-lub\ (r^{\hat{*}})\ x\ y\ (some-lub\ (r^{\hat{*}})\ x\ y)$   
 ⟨*proof*⟩

#### 4.1.1 An executable lub-finder

**definition** *exec-lub* :: ('a \* 'a) set  $\Rightarrow$  ('a  $\Rightarrow$  'a)  $\Rightarrow$  'a binop

where

$exec-lub\ r\ f\ x\ y = while\ (\lambda z. (x, z) \notin r^*)\ f\ y$

**lemma** *exec-lub-refl*:  $exec-lub\ r\ f\ T\ T = T$

$\langle proof \rangle$

**lemma** *acyclic-single-valued-finite*:

$\llbracket acyclic\ r; single\text{-}valued\ r; (x,y) \in r^* \rrbracket$   
 $\implies finite\ (r \cap \{a. (x, a) \in r^*\} \times \{b. (b, y) \in r^*\}) \langle proof \rangle$

**lemma** *exec-lub-conv*:

$\llbracket acyclic\ r; \forall x\ y. (x,y) \in r \longrightarrow f\ x = y; is\text{-}lub\ (r^*)\ x\ y\ u \rrbracket \implies$   
 $exec\text{-}lub\ r\ f\ x\ y = u \langle proof \rangle$

**lemma** *is-lub-exec-lub*:

$\llbracket single\text{-}valued\ r; acyclic\ r; (x,u):r^*; (y,u):r^*; \forall x\ y. (x,y) \in r \longrightarrow f\ x = y \rrbracket$   
 $\implies is\text{-}lub\ (r^*)\ x\ y\ (exec\text{-}lub\ r\ f\ x\ y)$

$\langle proof \rangle$

**end**

## 4.2 The Error Type

```

theory Err
imports Semilat
begin

datatype 'a err = Err | OK 'a

type-synonym 'a ebinop = 'a  $\Rightarrow$  'a  $\Rightarrow$  'a err
type-synonym 'a esl = 'a set  $\times$  'a ord  $\times$  'a ebinop

primrec ok-val :: 'a err  $\Rightarrow$  'a
where
  ok-val (OK x) = x

definition lift :: ('a  $\Rightarrow$  'b err)  $\Rightarrow$  ('a err  $\Rightarrow$  'b err)
where
  lift f e = (case e of Err  $\Rightarrow$  Err | OK x  $\Rightarrow$  f x)

definition lift2 :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'c err)  $\Rightarrow$  'a err  $\Rightarrow$  'b err  $\Rightarrow$  'c err
where
  lift2 f e1 e2 =
    (case e1 of Err  $\Rightarrow$  Err | OK x  $\Rightarrow$  (case e2 of Err  $\Rightarrow$  Err | OK y  $\Rightarrow$  f x y))

definition le :: 'a ord  $\Rightarrow$  'a err ord
where
  le r e1 e2 =
    (case e2 of Err  $\Rightarrow$  True | OK y  $\Rightarrow$  (case e1 of Err  $\Rightarrow$  False | OK x  $\Rightarrow$  x  $\sqsubseteq_r$  y))

definition sup :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'c)  $\Rightarrow$  ('a err  $\Rightarrow$  'b err  $\Rightarrow$  'c err)
where
  sup f = lift2 ( $\lambda$ x y. OK (x  $\sqcup_f$  y))

definition err :: 'a set  $\Rightarrow$  'a err set
where
  err A = insert Err {OK x | x. x  $\in$  A}

definition esl :: 'a sl  $\Rightarrow$  'a esl
where
  esl = ( $\lambda$ (A,r,f). (A, r,  $\lambda$ x y. OK(f x y)))

definition sl :: 'a esl  $\Rightarrow$  'a err sl
where
  sl = ( $\lambda$ (A,r,f). (err A, le r, lift2 f))

abbreviation
  err-semilat :: 'a esl  $\Rightarrow$  bool where
  err-semilat L == semilat(sl L)

primrec strict :: ('a  $\Rightarrow$  'b err)  $\Rightarrow$  ('a err  $\Rightarrow$  'b err)
where
  strict f Err = Err
  | strict f (OK x) = f x

```

**lemma** *err-def'*:

$err\ A = insert\ Err\ \{x.\ \exists y \in A.\ x = OK\ y\}\langle proof \rangle$

**lemma** *strict-Some* [*simp*]:

$(strict\ f\ x = OK\ y) = (\exists z.\ x = OK\ z \wedge f\ z = OK\ y)\langle proof \rangle$

**lemma** *not-Err-eq*:  $(x \neq Err) = (\exists a.\ x = OK\ a)\langle proof \rangle$

**lemma** *not-OK-eq*:  $(\forall y.\ x \neq OK\ y) = (x = Err)\langle proof \rangle$

**lemma** *unfold-lesub-err*:  $e1 \sqsubseteq_{le\ r} e2 = le\ r\ e1\ e2\langle proof \rangle$

**lemma** *le-err-reft*:  $\forall x.\ x \sqsubseteq_r x \implies e \sqsubseteq_{le\ r} e\langle proof \rangle$

**lemma** *le-err-trans* [*rule-format*]:

$order\ r \implies e1 \sqsubseteq_{le\ r} e2 \longrightarrow e2 \sqsubseteq_{le\ r} e3 \longrightarrow e1 \sqsubseteq_{le\ r} e3\langle proof \rangle$

**lemma** *le-err-antisym* [*rule-format*]:

$order\ r \implies e1 \sqsubseteq_{le\ r} e2 \longrightarrow e2 \sqsubseteq_{le\ r} e1 \longrightarrow e1 = e2\langle proof \rangle$

**lemma** *OK-le-err-OK*:  $(OK\ x \sqsubseteq_{le\ r} OK\ y) = (x \sqsubseteq_r y)\langle proof \rangle$

**lemma** *order-le-err* [*iff*]:  $order(le\ r) = order\ r\langle proof \rangle$

**lemma** *le-Err* [*iff*]:  $e \sqsubseteq_{le\ r} Err\langle proof \rangle$

**lemma** *Err-le-conv* [*iff*]:  $Err \sqsubseteq_{le\ r} e = (e = Err)\langle proof \rangle$

**lemma** *le-OK-conv* [*iff*]:  $e \sqsubseteq_{le\ r} OK\ x = (\exists y.\ e = OK\ y \wedge y \sqsubseteq_r x)\langle proof \rangle$

**lemma** *OK-le-conv*:  $OK\ x \sqsubseteq_{le\ r} e = (e = Err \vee (\exists y.\ e = OK\ y \wedge x \sqsubseteq_r y))\langle proof \rangle$

**lemma** *top-Err* [*iff*]:  $top\ (le\ r)\ Err\langle proof \rangle$

**lemma** *OK-less-conv* [*rule-format, iff*]:

$OK\ x \sqsubseteq_{le\ r} e = (e = Err \vee (\exists y.\ e = OK\ y \wedge x \sqsubseteq_r y))\langle proof \rangle$

**lemma** *not-Err-less* [*rule-format, iff*]:  $\neg(Err \sqsubseteq_{le\ r} x)\langle proof \rangle$

**lemma** *semilat-errI* [*intro*]: **assumes** *Semilat* *A* *r* *f*

**shows** *semilat*(*err* *A*, *le* *r*, *lift2*( $\lambda x\ y.\ OK(f\ x\ y)$ ))*proof*

**lemma** *err-semilat-eslI-aux*:

**assumes** *Semilat* *A* *r* *f* **shows** *err-semilat*(*esl*(*A*,*r*,*f*))*proof*

**lemma** *err-semilat-eslI* [*intro, simp*]:

$semilat\ L \implies err-semilat\ (esl\ L)\langle proof \rangle$

**lemma** *acc-err* [*simp, intro!*]:  $acc\ r \implies acc(le\ r)\langle proof \rangle$

**lemma** *Err-in-err* [*iff*]:  $Err : err\ A\langle proof \rangle$

**lemma** *Ok-in-err* [*iff*]:  $(OK\ x \in err\ A) = (x \in A)\langle proof \rangle$

### 4.2.1 lift

**lemma** *lift-in-errI*:

$\llbracket e \in err\ S; \forall x \in S.\ e = OK\ x \longrightarrow f\ x \in err\ S \rrbracket \implies lift\ f\ e \in err\ S\langle proof \rangle$

**lemma** *Err-lift2* [*simp*]:  $Err \sqcup_{lift2}\ f\ x = Err\langle proof \rangle$

**lemma** *lift2-Err* [*simp*]:  $x \sqcup_{lift2}\ f\ Err = Err\langle proof \rangle$

**lemma** *OK-lift2-OK* [*simp*]:  $OK\ x \sqcup_{lift2}\ f\ OK\ y = x \sqcup_f\ y\langle proof \rangle$

### 4.2.2 sup

**lemma** *Err-sup-Err* [*simp*]:  $Err \sqcup_{sup}\ f\ x = Err\langle proof \rangle$

**lemma** *Err-sup-Err2* [*simp*]:  $x \sqcup_{sup}\ f\ Err = Err\langle proof \rangle$

**lemma** *Err-sup-OK* [*simp*]:  $OK\ x \sqcup_{sup}\ f\ OK\ y = OK\ (x \sqcup_f\ y)\langle proof \rangle$

**lemma** *Err-sup-eq-OK-conv* [*iff*]:

$(sup\ f\ ex\ ey = OK\ z) = (\exists x\ y.\ ex = OK\ x \wedge ey = OK\ y \wedge f\ x\ y = z)\langle proof \rangle$

**lemma** *Err-sup-eq-Err* [*iff*]:  $(sup\ f\ ex\ ey = Err) = (ex = Err \vee ey = Err)\langle proof \rangle$

### 4.2.3 semilat (err A) (le r) f

**lemma** *semilat-le-err-Err-plus* [*simp*]:

$\llbracket x \in err\ A; semilat(err\ A, le\ r, f) \rrbracket \implies Err \sqcup_f\ x = Err\langle proof \rangle$

**lemma** *semilat-le-err-plus-Err* [*simp*]:

$\llbracket x \in \text{err } A; \text{semilat}(\text{err } A, \text{le } r, f) \rrbracket \implies x \sqcup_f \text{Err} = \text{Err}\langle \text{proof} \rangle$

**lemma** *semilat-le-err-OK1*:

$\llbracket x \in A; y \in A; \text{semilat}(\text{err } A, \text{le } r, f); \text{OK } x \sqcup_f \text{OK } y = \text{OK } z \rrbracket$   
 $\implies x \sqsubseteq_r z \langle \text{proof} \rangle$

**lemma** *semilat-le-err-OK2*:

$\llbracket x \in A; y \in A; \text{semilat}(\text{err } A, \text{le } r, f); \text{OK } x \sqcup_f \text{OK } y = \text{OK } z \rrbracket$   
 $\implies y \sqsubseteq_r z \langle \text{proof} \rangle$

**lemma** *eq-order-le*:

$\llbracket x=y; \text{order } r \rrbracket \implies x \sqsubseteq_r y \langle \text{proof} \rangle$

**lemma** *OK-plus-OK-eq-Err-conv* [simp]:

**assumes**  $x \in A \quad y \in A \quad \text{semilat}(\text{err } A, \text{le } r, fe)$

**shows**  $(\text{OK } x \sqcup_{fe} \text{OK } y = \text{Err}) = (\neg(\exists z \in A. x \sqsubseteq_r z \wedge y \sqsubseteq_r z)) \langle \text{proof} \rangle$

#### 4.2.4 semilat (err(Union AS))

**lemma** *all-bex-swap-lemma* [iff]:

$(\forall x. (\exists y \in A. x = f y) \longrightarrow P x) = (\forall y \in A. P(f y)) \langle \text{proof} \rangle$

**lemma** *closed-err-Union-lift2I*:

$\llbracket \forall A \in AS. \text{closed}(\text{err } A) (\text{lift2 } f); AS \neq \{\};$   
 $\forall A \in AS. \forall B \in AS. A \neq B \longrightarrow (\forall a \in A. \forall b \in B. a \sqcup_f b = \text{Err}) \rrbracket$   
 $\implies \text{closed}(\text{err}(\text{Union } AS)) (\text{lift2 } f) \langle \text{proof} \rangle$

If  $AS = \{\}$  the thm collapses to  $\text{order } r \wedge \text{closed} \{ \text{Err} \} f \wedge \text{Err} \sqcup_f \text{Err} = \text{Err}$  which may not hold

**lemma** *err-semilat-UnionI*:

$\llbracket \forall A \in AS. \text{err-semilat}(A, r, f); AS \neq \{\};$   
 $\forall A \in AS. \forall B \in AS. A \neq B \longrightarrow (\forall a \in A. \forall b \in B. \neg a \sqsubseteq_r b \wedge a \sqcup_f b = \text{Err}) \rrbracket$   
 $\implies \text{err-semilat}(\text{Union } AS, r, f) \langle \text{proof} \rangle$

**end**





## 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**

$$sup\ f\ g = (\lambda(a_1, b_1)(a_2, b_2). Err.sup\ Pair\ (a_1 \sqcup_f a_2)\ (b_1 \sqcup_g b_2))$$

**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** (*xsymbols*)

$$lesubprod :: 'a \times 'b \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \times 'b \Rightarrow bool$$

$$((- / \sqsubseteq'(-, -) -) [50, 0, 0, 51] 50) \textbf{ where}$$

$$p \sqsubseteq(rA, rB)\ q == p \sqsubseteq_{Product.le\ rA\ rB}\ q$$

**lemma** *unfold-lesub-prod*:  $x \sqsubseteq(r_A, r_B)\ y = le\ r_A\ r_B\ x\ y$ *<proof>*

**lemma** *le-prod-Pair-conv* [*iff*]:  $((a_1, b_1) \sqsubseteq(r_A, r_B)\ (a_2, b_2)) = (a_1 \sqsubseteq_{r_A} a_2 \ \&\ b_1 \sqsubseteq_{r_B} b_2)$ *<proof>*

**lemma** *less-prod-Pair-conv*:

$$((a_1, b_1) \sqsubseteq_{Product.le\ r_A\ r_B}\ (a_2, b_2)) =$$

$$(a_1 \sqsubseteq_{r_A} a_2 \ \&\ b_1 \sqsubseteq_{r_B} b_2 \mid a_1 \sqsubseteq_{r_A} a_2 \ \&\ b_1 \sqsubseteq_{r_B} b_2)$$
*<proof>*

**lemma** *order-le-prod* [*iff*]:  $order(Product.le\ r_A\ r_B) = (order\ r_A \ \&\ order\ r_B)$ *<proof>*

**lemma** *acc-le-prodI* [*intro!*]:

$$\llbracket acc\ r_A; acc\ r_B \rrbracket \Longrightarrow acc(Product.le\ r_A\ r_B)$$
*<proof>*

**lemma** *closed-lift2-sup*:

$$\llbracket closed\ (err\ A)\ (lift2\ f); closed\ (err\ B)\ (lift2\ g) \rrbracket \Longrightarrow$$

$$closed\ (err\ (A \times B))\ (lift2\ (sup\ f\ g))$$
*<proof>*

**lemma** *unfold-plussub-lift2*:  $e_1 \sqcup_{lift2\ f}\ e_2 = lift2\ f\ e_1\ e_2$ *<proof>*

**lemma** *plus-eq-Err-conv* [*simp*]:

**assumes**  $x \in A\ y \in A\ semilat(err\ A, Err.le\ r, lift2\ f)$

**shows**  $(x \sqcup_f y = Err) = (\neg(\exists z \in A. x \sqsubseteq_r z \wedge y \sqsubseteq_r z))$ *<proof>*

**lemma** *err-semilat-Product-esl*:

$$\bigwedge L_1\ L_2. \llbracket err-semilat\ L_1; err-semilat\ L_2 \rrbracket \Longrightarrow err-semilat(Product.esl\ L_1\ L_2)$$
*<proof>*

**end**

## 4.5 Fixed Length Lists

**theory** Listn  
**imports** Err  
**begin**

**definition** list :: nat  $\Rightarrow$  'a set  $\Rightarrow$  'a list set  
**where**  
 list n A = {xs. size xs = n  $\wedge$  set xs  $\subseteq$  A}

**definition** le :: 'a ord  $\Rightarrow$  ('a list)ord  
**where**  
 le r = list-all2 ( $\lambda$ x y. x  $\sqsubseteq_r$  y)

**abbreviation** (xsymbols)  
 lesublist :: 'a list  $\Rightarrow$  'a ord  $\Rightarrow$  'a list  $\Rightarrow$  bool ((- / $\sqsubseteq_r$ -) [50, 0, 51] 50) **where**  
 x  $\sqsubseteq_r$  y == x  $\leq$ -(Listn.le r) y

**abbreviation** (xsymbols)  
 lessublist :: 'a list  $\Rightarrow$  'a ord  $\Rightarrow$  'a list  $\Rightarrow$  bool ((- / $\sqsubseteq$ -) [50, 0, 51] 50) **where**  
 x  $\sqsubseteq$  y == x  $\leq$ -(Listn.le r) y

**definition** map2 :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'c)  $\Rightarrow$  'a list  $\Rightarrow$  'b list  $\Rightarrow$  'c list  
**where**  
 map2 f = ( $\lambda$ xs ys. map (split f) (zip xs ys))

**abbreviation** (xsymbols)  
 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  $\sqcup_{\text{map2 } f}$  y

**primrec** coalesce :: 'a err list  $\Rightarrow$  'a list err  
**where**  
 coalesce [] = OK []  
 | coalesce (ex#exs) = Err.sup (op #) ex (coalesce exs)

**definition** sl :: nat  $\Rightarrow$  'a sl  $\Rightarrow$  'a list sl  
**where**  
 sl n = ( $\lambda$ (A,r,f). (list 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{list 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<proof>

**lemma** Nil-le-conv [iff]: ([]  $\sqsubseteq_r$  ys) = (ys = [])<proof>

**lemma** *Cons-notle-Nil* [iff]:  $\neg x\#xs \sqsubseteq_r []$  (proof)

**lemma** *Cons-le-Cons* [iff]:  $x\#xs \sqsubseteq_r y\#ys = (x \sqsubseteq_r y \wedge xs \sqsubseteq_r ys)$  (proof)

**lemma** *Cons-less-Cons* [simp]:

$order\ r \implies x\#xs \sqsubseteq_r y\#ys = (x \sqsubseteq_r y \wedge xs \sqsubseteq_r ys \vee x = y \wedge xs \sqsubseteq_r ys)$  (proof)

**lemma** *list-update-le-cong*:

$[i < size\ xs; xs \sqsubseteq_r ys; x \sqsubseteq_r y] \implies xs[i:=x] \sqsubseteq_r ys[i:=y]$  (proof)

**lemma** *le-listD*:  $[xs \sqsubseteq_r ys; p < size\ xs] \implies xs!p \sqsubseteq_r ys!p$  (proof)

**lemma** *le-list-refl*:  $\forall x. x \sqsubseteq_r x \implies xs \sqsubseteq_r xs$  (proof)

**lemma** *le-list-trans*:  $[order\ r; xs \sqsubseteq_r ys; ys \sqsubseteq_r zs] \implies xs \sqsubseteq_r zs$  (proof)

**lemma** *le-list-antisym*:  $[order\ r; xs \sqsubseteq_r ys; ys \sqsubseteq_r xs] \implies xs = ys$  (proof)

**lemma** *order-listI* [simp, intro!]:  $order\ r \implies order(Listn.le\ r)$  (proof)

**lemma** *lesub-list-impl-same-size* [simp]:  $xs \sqsubseteq_r ys \implies size\ ys = size\ xs$  (proof)

**lemma** *lessub-lengthD*:  $xs \sqsubseteq_r ys \implies size\ ys = size\ xs$  (proof)

**lemma** *le-list-appendI*:  $a \sqsubseteq_r b \implies c \sqsubseteq_r d \implies a@c \sqsubseteq_r b@d$  (proof)

**lemma** *le-listI*:

assumes  $length\ a = length\ b$

assumes  $\bigwedge n. n < length\ a \implies a!n \sqsubseteq_r b!n$

shows  $a \sqsubseteq_r b$  (proof)

**lemma** *listI*:  $[size\ xs = n; set\ xs \subseteq A] \implies xs \in list\ n\ A$  (proof)

**lemma** *listE-length* [simp]:  $xs \in list\ n\ A \implies size\ xs = n$  (proof)

**lemma** *less-lengthI*:  $[xs \in list\ n\ A; p < n] \implies p < size\ xs$  (proof)

**lemma** *listE-set* [simp]:  $xs \in list\ n\ A \implies set\ xs \subseteq A$  (proof)

**lemma** *list-0* [simp]:  $list\ 0\ A = \{[]\}$  (proof)

**lemma** *in-list-Suc-iff*:

$(xs \in list\ (Suc\ n)\ A) = (\exists y \in A. \exists ys \in list\ n\ A. xs = y\#ys)$  (proof)

**lemma** *Cons-in-list-Suc* [iff]:

$(x\#xs \in list\ (Suc\ n)\ A) = (x \in A \wedge xs \in list\ n\ A)$  (proof)

**lemma** *list-not-empty*:

$\exists a. a \in A \implies \exists xs. xs \in list\ n\ A$  (proof)

**lemma** *nth-in* [rule-format, simp]:

$\forall i\ n. size\ xs = n \longrightarrow set\ xs \subseteq A \longrightarrow i < n \longrightarrow (xs!i) \in A$  (proof)

**lemma** *listE-nth-in*:  $[xs \in list\ n\ A; i < n] \implies xs!i \in A$  (proof)

**lemma** *listn-Cons-Suc* [elim!]:

$l\#xs \in list\ n\ A \implies (\bigwedge n'. n = Suc\ n' \implies l \in A \implies xs \in list\ n'\ A \implies P) \implies P$  (proof)

**lemma** *listn-appendE* [elim!]:

$a@c \in list\ n\ A \implies (\bigwedge n1\ n2. n = n1 + n2 \implies a \in list\ n1\ A \implies c \in list\ n2\ A \implies P) \implies P$  (proof)

**lemma** *listt-update-in-list* [simp, intro!]:

$[xs \in list\ n\ A; x \in A] \implies xs[i := x] \in list\ n\ A$  (proof)

**lemma** *list-appendI* [intro?]:

$[a \in list\ n\ A; b \in list\ m\ A] \implies a @ b \in list\ (n+m)\ A$  (proof)

**lemma** *list-map* [simp]:  $(map\ f\ xs \in list\ (size\ xs)\ A) = (f\ ' set\ xs \subseteq A)$  (proof)

**lemma** *list-replicateI* [intro]:  $x \in A \implies replicate\ n\ x \in list\ n\ A$  (proof)

**lemma** *plus-list-Nil* [simp]:  $[] \sqcup_f xs = []$  (proof)

**lemma** *plus-list-Cons* [simp]:

$(x\#xs) \sqcup_f ys = (case\ ys\ of\ [] \Rightarrow [] \mid y\#ys \Rightarrow (x \sqcup_f y)\#(xs \sqcup_f ys))$  (proof)

**lemma** *length-plus-list* [rule-format, simp]:

$\forall ys. size(xs \sqcup_f ys) = min(size\ xs)\ (size\ ys)$  (proof)

**lemma** *nth-plus-list* [rule-format, simp]:

$\forall xs\ ys\ i. size\ xs = n \longrightarrow size\ ys = n \longrightarrow i < n \longrightarrow (xs \sqcup_f ys)!i = (xs!i) \sqcup_f (ys!i)$  (proof)

**lemma** (in *Semilat*) *plus-list-ub1* [rule-format]:

$\llbracket \text{set } xs \subseteq A; \text{set } ys \subseteq A; \text{size } xs = \text{size } ys \rrbracket$   
 $\implies xs \llbracket \sqsubseteq_r \rrbracket xs \llbracket \sqcup_f \rrbracket ys \langle \text{proof} \rangle$

**lemma** (in *Semilat*) *plus-list-ub2*:

$\llbracket \text{set } xs \subseteq A; \text{set } ys \subseteq A; \text{size } xs = \text{size } ys \rrbracket \implies ys \llbracket \sqsubseteq_r \rrbracket xs \llbracket \sqcup_f \rrbracket ys \langle \text{proof} \rangle$

**lemma** (in *Semilat*) *plus-list-lub* [rule-format]:

**shows**  $\forall xs \ ys \ zs. \text{set } xs \subseteq A \longrightarrow \text{set } ys \subseteq A \longrightarrow \text{set } zs \subseteq A$   
 $\longrightarrow \text{size } xs = n \wedge \text{size } ys = n \longrightarrow$

$xs \llbracket \sqsubseteq_r \rrbracket zs \wedge ys \llbracket \sqsubseteq_r \rrbracket zs \longrightarrow xs \llbracket \sqcup_f \rrbracket ys \llbracket \sqsubseteq_r \rrbracket zs \langle \text{proof} \rangle$

**lemma** (in *Semilat*) *list-update-incr* [rule-format]:

$x \in A \implies \text{set } xs \subseteq A \longrightarrow$   
 $(\forall i. i < \text{size } xs \longrightarrow xs \llbracket \sqsubseteq_r \rrbracket xs[i := x \sqcup_f xs!i]) \langle \text{proof} \rangle$

**lemma** *acc-le-listI* [intro!]:

$\llbracket \text{order } r; \text{acc } r \rrbracket \implies \text{acc}(\text{Listn.le } r) \langle \text{proof} \rangle$

**lemma** *closed-listI*:

$\text{closed } S \ f \implies \text{closed } (\text{list } n \ S) \ (\text{map2 } f) \langle \text{proof} \rangle$

**lemma** *Listn-sl-aux*:

**assumes** *Semilat*  $A \ r \ f$  **shows** *semilat*  $(\text{Listn.sl } n \ (A, r, f)) \langle \text{proof} \rangle$

**lemma** *Listn-sl*: *semilat*  $L \implies \text{semilat } (\text{Listn.sl } n \ L) \langle \text{proof} \rangle$

**lemma** *coalesce-in-err-list* [rule-format]:

$\forall xes. xes \in \text{list } n \ (\text{err } A) \longrightarrow \text{coalesce } xes \in \text{err}(\text{list } n \ A) \langle \text{proof} \rangle$

**lemma** *lem*:  $\bigwedge x \ xs. x \sqcup_{op} \# \ xs = x \# \ xs \langle \text{proof} \rangle$

**lemma** *coalesce-eq-OK1-D* [rule-format]:

*semilat* $(\text{err } A, \text{Err.le } r, \text{lift2 } f) \implies$   
 $\forall xs. xs \in \text{list } n \ A \longrightarrow (\forall ys. ys \in \text{list } n \ A \longrightarrow$   
 $(\forall zs. \text{coalesce } (xs \llbracket \sqcup_f \rrbracket ys) = \text{OK } zs \longrightarrow xs \llbracket \sqsubseteq_r \rrbracket zs) \rangle \langle \text{proof} \rangle$

**lemma** *coalesce-eq-OK2-D* [rule-format]:

*semilat* $(\text{err } A, \text{Err.le } r, \text{lift2 } f) \implies$   
 $\forall xs. xs \in \text{list } n \ A \longrightarrow (\forall ys. ys \in \text{list } n \ A \longrightarrow$   
 $(\forall zs. \text{coalesce } (xs \llbracket \sqcup_f \rrbracket ys) = \text{OK } zs \longrightarrow ys \llbracket \sqsubseteq_r \rrbracket zs) \rangle \langle \text{proof} \rangle$

**lemma** *lift2-le-ub*:

$\llbracket \text{semilat}(\text{err } A, \text{Err.le } r, \text{lift2 } f); x \in A; y \in A; x \sqcup_f y = \text{OK } z;$   
 $u \in A; x \llbracket \sqsubseteq_r \rrbracket u; y \llbracket \sqsubseteq_r \rrbracket u \rrbracket \implies z \llbracket \sqsubseteq_r \rrbracket u \langle \text{proof} \rangle$

**lemma** *coalesce-eq-OK-ub-D* [rule-format]:

*semilat* $(\text{err } A, \text{Err.le } r, \text{lift2 } f) \implies$   
 $\forall xs. xs \in \text{list } n \ A \longrightarrow (\forall ys. ys \in \text{list } n \ A \longrightarrow$   
 $(\forall zs \ us. \text{coalesce } (xs \llbracket \sqcup_f \rrbracket ys) = \text{OK } zs \wedge xs \llbracket \sqsubseteq_r \rrbracket us \wedge ys \llbracket \sqsubseteq_r \rrbracket us$   
 $\wedge us \in \text{list } n \ A \longrightarrow zs \llbracket \sqsubseteq_r \rrbracket us) \rangle \langle \text{proof} \rangle$

**lemma** *lift2-eq-ErrD*:

$\llbracket x \sqcup_f y = \text{Err}; \text{semilat}(\text{err } A, \text{Err.le } r, \text{lift2 } f); x \in A; y \in A \rrbracket$   
 $\implies \neg(\exists u \in A. x \llbracket \sqsubseteq_r \rrbracket u \wedge y \llbracket \sqsubseteq_r \rrbracket u) \langle \text{proof} \rangle$

**lemma** *coalesce-eq-Err-D* [rule-format]:

$\llbracket \text{semilat}(\text{err } A, \text{Err.le } r, \text{lift2 } f) \rrbracket$   
 $\implies \forall xs. xs \in \text{list } n \ A \longrightarrow (\forall ys. ys \in \text{list } n \ A \longrightarrow$   
 $\text{coalesce } (xs \llbracket \sqcup_f \rrbracket ys) = \text{Err} \longrightarrow$   
 $\neg(\exists zs \in \text{list } n \ A. xs \llbracket \sqsubseteq_r \rrbracket zs \wedge ys \llbracket \sqsubseteq_r \rrbracket zs) \rangle \langle \text{proof} \rangle$

**lemma** *closed-err-lift2-conv*:

$\text{closed } (\text{err } A) \ (\text{lift2 } f) = (\forall x \in A. \forall y \in A. x \sqcup_f y \in \text{err } A) \langle \text{proof} \rangle$

**lemma** *closed-map2-list* [rule-format]:

$\text{closed } (\text{err } A) \ (\text{lift2 } f) \implies$   
 $\forall xs. xs \in \text{list } n \ A \longrightarrow (\forall ys. ys \in \text{list } n \ A \longrightarrow$

*map2 f xs ys ∈ list n (err A)**<proof>*

**lemma** *closed-lift2-sup:*

*closed (err A) (lift2 f) ⇒*

*closed (err (list n A)) (lift2 (sup f))**<proof>*

**lemma** *err-semilat-sup:*

*err-semilat (A,r,f) ⇒*

*err-semilat (list n A, Listn.le r, sup f)**<proof>*

**lemma** *err-semilat-upto-esl:*

*∧L. err-semilat L ⇒ err-semilat(upto-esl m L)**<proof>*

**end**

## 4.6 Typing and Dataflow Analysis Framework

**theory** *Typing-Framework* **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** *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** *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** *wt-step* ::  $'s \text{ ord} \Rightarrow 's \Rightarrow 's \text{ step-type} \Rightarrow 's \text{ list} \Rightarrow \text{bool}$

**where**

$\text{wt-step } r \ T \ \text{step } \tau s \longleftrightarrow (\forall p < \text{size } \tau s. \tau s!p \neq T \wedge \text{stable } r \ \text{step } \tau s \ p)$

**definition** *is-bcv* ::  $'s \text{ ord} \Rightarrow 's \Rightarrow 's \text{ step-type} \Rightarrow \text{nat} \Rightarrow 's \text{ set} \Rightarrow ('s \text{ list} \Rightarrow 's \text{ list}) \Rightarrow \text{bool}$

**where**

$\text{is-bcv } r \ T \ \text{step } n \ A \ \text{bcv} \longleftrightarrow (\forall \tau s_0 \in \text{list } n \ A.$

$(\forall p < n. (\text{bcv } \tau s_0)!p \neq T) = (\exists \tau s \in \text{list } n \ A. \tau s_0 \sqsubseteq_r \tau s \wedge \text{wt-step } r \ T \ \text{step } \tau s))$

**end**

## 4.7 More on Semilattices

```

theory SemilatAlg
imports Typing-Framework
begin

consts
  lesubstep-type :: (nat × 's) set ⇒ 's ord ⇒ (nat × 's) set ⇒ bool notation (xsymbols)
  lesubstep-type ((- / {⊑r}-) -) [50, 0, 51] 50)
defs lesubstep-type-def:
  A {⊑r} B ≡ ∀(p,τ) ∈ A. ∃τ'. (p,τ') ∈ B ∧ τ ⊑r τ'

primrec pluslssub :: 'a list ⇒ ('a ⇒ 'a ⇒ 'a) ⇒ 'a ⇒ 'a
where
  pluslssub [] f y = y
| pluslssub (x#xs) f y = pluslssub xs f (x ⊔f y) notation (xsymbols)
  pluslssub ((- / ⊔f-) -) [65, 0, 66] 65)

definition bounded :: 's step-type ⇒ nat ⇒ bool
where
  bounded step n ⟷ (∀ p < n. ∀ τ. ∀ (q,τ') ∈ set (step p τ). q < n)

definition pres-type :: 's step-type ⇒ nat ⇒ 's set ⇒ bool
where
  pres-type step n A ⟷ (∀ τ ∈ A. ∀ p < n. ∀ (q,τ') ∈ set (step p τ). τ' ∈ A)

definition mono :: 's ord ⇒ 's step-type ⇒ nat ⇒ 's set ⇒ bool
where
  mono r step n A ⟷
    (∀ τ p τ'. τ ∈ A ∧ p < n ∧ τ ⊑r τ' ⟶ set (step p τ) {⊑r} set (step p τ'))

lemma [iff]: {} {⊑r} B
  <proof>
lemma [iff]: (A {⊑r} {}) = (A = {})
  <proof>
lemma lesubstep-union:
  [[ A1 {⊑r} B1; A2 {⊑r} B2 ]] ⟹ A1 ∪ A2 {⊑r} B1 ∪ B2
  <proof>
lemma pres-typeD:
  [[ pres-type step n A; s ∈ A; p < n; (q,s') ∈ set (step p s) ]] ⟹ s' ∈ A <proof>
lemma monoD:
  [[ mono r step n A; p < n; s ∈ A; s ⊑r t ]] ⟹ set (step p s) {⊑r} set (step p t) <proof>
lemma boundedD:
  [[ bounded step n; p < n; (q,t) ∈ set (step p xs) ]] ⟹ q < n <proof>
lemma lesubstep-type-refl [simp, intro]:
  (∧ x. x ⊑r x) ⟹ A {⊑r} A <proof>
lemma lesub-step-typeD:
  A {⊑r} B ⟹ (x,y) ∈ A ⟹ ∃ y'. (x, y') ∈ B ∧ y ⊑r y' <proof>

lemma list-update-le-listI [rule-format]:
  set xs ⊆ A ⟶ set ys ⊆ A ⟶ xs [⊑r] ys ⟶ p < size xs ⟶
  x ⊑r ys!p ⟶ semilat(A,r,f) ⟶ x ∈ A ⟶
  xs[p := x ⊔f xs!p] [⊑r] ys <proof>
lemma plusplus-closed: assumes Semilat A r f shows

```



$\bigwedge y. \llbracket \text{set } x \subseteq A; y \in A \rrbracket \Longrightarrow x \sqcup_f y \in A \langle \text{proof} \rangle$

**lemma** (in *Semilat*) *pp-ub2*:

$\bigwedge y. \llbracket \text{set } x \subseteq A; y \in A \rrbracket \Longrightarrow y \sqsubseteq_r x \sqcup_f y \langle \text{proof} \rangle$

**lemma** (in *Semilat*) *pp-ub1*:

**shows**  $\bigwedge y. \llbracket \text{set } ls \subseteq A; y \in A; x \in \text{set } ls \rrbracket \Longrightarrow x \sqsubseteq_r ls \sqcup_f y \langle \text{proof} \rangle$

**lemma** (in *Semilat*) *pp-lub*:

**assumes**  $z: z \in A$

**shows**

$\bigwedge y. y \in A \Longrightarrow \text{set } xs \subseteq A \Longrightarrow \forall x \in \text{set } xs. x \sqsubseteq_r z \Longrightarrow y \sqsubseteq_r z \Longrightarrow xs \sqcup_f y \sqsubseteq_r z \langle \text{proof} \rangle$

**lemma** *ub1'*: **assumes** *Semilat A r f*

**shows**  $\llbracket \forall (p,s) \in \text{set } S. s \in A; y \in A; (a,b) \in \text{set } S \rrbracket$

$\Longrightarrow b \sqsubseteq_r \text{map snd } [(p', t') \leftarrow S. p' = a] \sqcup_f y \langle \text{proof} \rangle$

**lemma** *plusplus-empty*:

$\forall s'. (q, s') \in \text{set } S \longrightarrow s' \sqcup_f ss ! q = ss ! q \Longrightarrow$

$(\text{map snd } [(p', t') \leftarrow S. p' = q] \sqcup_f ss ! q) = ss ! q \langle \text{proof} \rangle$

**end**

## 4.8 Lifting the Typing Framework to `err`, `app`, and `eff`

**theory** *Typing-Framework-err* **imports** *Typing-Framework SemilatAlg* **begin**

**definition** *wt-err-step* :: 's ord  $\Rightarrow$  's err step-type  $\Rightarrow$  's err list  $\Rightarrow$  bool

**where**

*wt-err-step* r step  $\tau$  s  $\longleftrightarrow$  *wt-step* (Err.le r) 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**

*wt-app-eff* r app step  $\tau$  s  $\longleftrightarrow$

( $\forall p < \text{size } \tau$  s. 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**

*err-step* n app step p t =

(case t of

Err  $\Rightarrow$  error n

| OK  $\tau \Rightarrow$  if app p  $\tau$  then map-snd OK (step p  $\tau$ ) else error n)

**definition** *app-mono* :: 's ord  $\Rightarrow$  (nat  $\Rightarrow$  's  $\Rightarrow$  bool)  $\Rightarrow$  nat  $\Rightarrow$  's set  $\Rightarrow$  bool

**where**

*app-mono* r app n A  $\longleftrightarrow$

( $\forall s$  p t.  $s \in A \wedge p < n \wedge s \sqsubseteq_r t \longrightarrow$  app p t  $\longrightarrow$  app p s)

**lemmas** *err-step-defs* = *err-step-def* *map-snd-def* *error-def*

**lemma** *bounded-err-stepD*:

$\llbracket$  bounded (*err-step* n app step) n;

$p < n$ ; app p a; (q,b)  $\in$  set (step p a)  $\rrbracket \Longrightarrow q < n$ ⟨proof⟩

**lemma** *in-map-sndD*: (a,b)  $\in$  set (map-snd f xs)  $\Longrightarrow \exists b'. (a,b') \in$  set xs⟨proof⟩

**lemma** *bounded-err-stepI*:

$\forall p. p < n \longrightarrow (\forall s. \text{app } p \ s \longrightarrow (\forall (q,s') \in \text{set } (\text{step } p \ s). q < n))$

$\Longrightarrow$  bounded (*err-step* n app step) n⟨proof⟩

**lemma** *bounded-lift*:

bounded step n  $\Longrightarrow$  bounded (*err-step* n app step) n⟨proof⟩

**lemma** *le-list-map-OK* [simp]:

$\bigwedge b. (\text{map } \text{OK } a \ [\sqsubseteq_{\text{Err.le } r}] \ \text{map } \text{OK } b) = (a \ [\sqsubseteq_r] \ b)$ ⟨proof⟩

**lemma** *map-snd-lessI*:

$set\ xs\ \{\sqsubseteq_r\}\ set\ ys\ \Longrightarrow\ set\ (map\ snd\ OK\ xs)\ \{\sqsubseteq_{Err.le\ r}\}\ set\ (map\ snd\ OK\ ys)\langle proof \rangle$

**lemma** *mono-lift*:

$\llbracket\ order\ r;\ app\ mono\ r\ app\ n\ A;\ bounded\ (err\ step\ n\ app\ step)\ n;\ \forall\ s\ p\ t.\ s \in A \wedge p < n \wedge s \sqsubseteq_r t \longrightarrow app\ p\ t \longrightarrow set\ (step\ p\ s)\ \{\sqsubseteq_r\}\ set\ (step\ p\ t)\ \rrbracket$   
 $\Longrightarrow\ mono\ (Err.le\ r)\ (err\ step\ n\ app\ step)\ n\ (err\ A)\langle proof \rangle$

**lemma** *in-errorD*:  $(x,y) \in set\ (error\ n) \Longrightarrow y = Err\langle proof \rangle$

**lemma** *pres-type-lift*:

$\forall\ s \in A.\ \forall\ p.\ p < n \longrightarrow app\ p\ s \longrightarrow (\forall\ (q,\ s') \in set\ (step\ p\ s).\ s' \in A)$   
 $\Longrightarrow\ pres\ type\ (err\ step\ n\ app\ step)\ n\ (err\ A)\langle proof \rangle$

**lemma** *wt-err-imp-wt-app-eff*:

**assumes** *wt*:  $wt\ err\ step\ r\ (err\ step\ (size\ ts)\ app\ step)\ ts$   
**assumes** *b*:  $bounded\ (err\ step\ (size\ ts)\ app\ step)\ (size\ ts)$   
**shows**  $wt\ app\ eff\ r\ app\ step\ (map\ ok\ val\ ts)\langle proof \rangle$

**lemma** *wt-app-eff-imp-wt-err*:

**assumes** *app-eff*:  $wt\ app\ eff\ r\ app\ step\ ts$   
**assumes** *bounded*:  $bounded\ (err\ step\ (size\ ts)\ app\ step)\ (size\ ts)$   
**shows**  $wt\ err\ step\ r\ (err\ step\ (size\ ts)\ app\ step)\ (map\ OK\ ts)\langle proof \rangle$

**end**

## 4.9 Kildall's Algorithm

**theory** *Kildall*  
**imports** *SemilatAlg*  
**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*';  
 $u = \tau \sqcup_f \tau s!q$ ;  
 $w' = (\text{if } u = \tau s!q \text{ then } w \text{ else insert } q \text{ } w)$   
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** *unstabes* :: 's ord  $\Rightarrow$  's step-type  $\Rightarrow$  's list  $\Rightarrow$  nat set  
**where**

*unstabes* *r* step  $\tau s$  =  $\{p. p < \text{size } \tau s \wedge \neg \text{stable } r \text{ step } \tau s \text{ } p\}$

**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  $\tau s$  (*unstabes* *r* step  $\tau s$ ))

**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*)  $\tau s$  = (let (*p*, $\tau$ ) = *p*' in *merges* *f* *ps* ( $\tau s[p := \tau \sqcup_f \tau s!p]$ ))

**lemmas** [*simp*] = *Let-def Semilat.le-iff-plus-unchanged [OF Semilat.intro, symmetric]*

**lemma** (in *Semilat*) *nth-merges*:

$\bigwedge ss. \llbracket p < \text{length } ss; ss \in \text{list } n \text{ } A; \forall (p, t) \in \text{set } ps. p < n \wedge t \in A \rrbracket \Longrightarrow$   
(*merges* *f* *ps* *ss*)!*p* = *map snd*  $\llbracket (p', t') \leftarrow ps. p' = p \rrbracket \sqcup_f ss!p$   
(**is**  $\bigwedge ss. \llbracket -; -; ?\text{steptype } ps \rrbracket \Longrightarrow ?P \text{ } ss \text{ } ps$ )*(proof)*

**lemma** *length-merges* [*simp*]:

$\bigwedge ss. \text{size}(\text{merges } f \text{ } ps \text{ } ss) = \text{size } ss$ *(proof)*

**lemma** (in *Semilat*) *merges-preserves-type-lemma*:

**shows**  $\forall xs. xs \in \text{list } n \text{ } A \longrightarrow (\forall (p, x) \in \text{set } ps. p < n \wedge x \in A)$   
 $\longrightarrow \text{merges } f \text{ } ps \text{ } xs \in \text{list } n \text{ } A$ *(proof)*

**lemma** (in *Semilat*) *merges-preserves-type* [simp]:

$\llbracket xs \in \text{list } n \ A; \forall (p,x) \in \text{set } ps. p < n \wedge x \in A \rrbracket$   
 $\implies \text{merges } f \ ps \ xs \in \text{list } n \ A$

$\langle \text{proof} \rangle$

**lemma** (in *Semilat*) *merges-incr-lemma*:

$\forall xs. xs \in \text{list } n \ A \longrightarrow (\forall (p,x) \in \text{set } ps. p < \text{size } xs \wedge x \in A) \longrightarrow xs \ [\sqsubseteq_r] \ \text{merges } f \ ps \ xs \langle \text{proof} \rangle$

**lemma** (in *Semilat*) *merges-incr*:

$\llbracket xs \in \text{list } n \ A; \forall (p,x) \in \text{set } ps. p < \text{size } xs \wedge x \in A \rrbracket$   
 $\implies xs \ [\sqsubseteq_r] \ \text{merges } f \ ps \ xs$

$\langle \text{proof} \rangle$

**lemma** (in *Semilat*) *merges-same-conv* [rule-format]:

$(\forall xs. xs \in \text{list } n \ A \longrightarrow (\forall (p,x) \in \text{set } ps. p < \text{size } xs \wedge x \in A) \longrightarrow$   
 $(\text{merges } f \ ps \ xs = xs) = (\forall (p,x) \in \text{set } ps. x \ [\sqsubseteq_r] \ xs!p)) \langle \text{proof} \rangle$

**lemma** (in *Semilat*) *list-update-le-listI* [rule-format]:

$\text{set } xs \subseteq A \longrightarrow \text{set } ys \subseteq A \longrightarrow xs \ [\sqsubseteq_r] \ ys \longrightarrow p < \text{size } xs \longrightarrow$   
 $x \ [\sqsubseteq_r] \ ys!p \longrightarrow x \in A \longrightarrow xs[p := x \sqcup_f \ ys!p] \ [\sqsubseteq_r] \ ys \langle \text{proof} \rangle$

**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**  $\text{merges } f \ ps \ ss \ [\sqsubseteq_r] \ ts \langle \text{proof} \rangle$

**lemma** *decomp-propa*:

$\bigwedge ss \ w. (\forall (q,t) \in \text{set } qs. q < \text{size } ss) \implies$

$\text{propa } f \ qs \ ss \ w =$

$(\text{merges } f \ qs \ ss, \{q. \exists t. (q,t) \in \text{set } qs \wedge t \sqcup_f \ ss!q \neq ss!q\} \cup w) \langle \text{proof} \rangle$

**lemma** (in *Semilat*) *stable-pres-lemma*:

**shows**  $\llbracket \text{pres-type step } n \ A; \text{bounded step } n;$

$ss \in \text{list } n \ A; p \in w; \forall q \in w. q < n;$

$\forall q. q < n \longrightarrow q \notin w \longrightarrow \text{stable } r \ \text{step } ss \ q; q < n;$

$\forall s'. (q,s') \in \text{set } (\text{step } p \ (ss!p)) \longrightarrow s' \sqcup_f \ ss!q = ss!q;$

$q \notin w \vee q = p \rrbracket$

$\implies \text{stable } r \ \text{step } (\text{merges } f \ (\text{step } p \ (ss!p)) \ ss) \ q \langle \text{proof} \rangle$

**lemma** (in *Semilat*) *merges-bounded-lemma*:

$\llbracket \text{mono } r \ \text{step } n \ A; \text{bounded step } n;$

$\forall (p',s') \in \text{set } (\text{step } p \ (ss!p)). s' \in A; ss \in \text{list } n \ A; ts \in \text{list } n \ A; p < n;$

$ss \ [\sqsubseteq_r] \ ts; \forall p. p < n \longrightarrow \text{stable } r \ \text{step } ts \ p \rrbracket$

$\implies \text{merges } f \ (\text{step } p \ (ss!p)) \ ss \ [\sqsubseteq_r] \ ts \langle \text{proof} \rangle$

**lemma** *termination-lemma*: **assumes** *Semilat*  $A \ r \ f$

**shows**  $\llbracket ss \in \text{list } n \ A; \forall (q,t) \in \text{set } qs. q < n \wedge t \in A; p \in w \rrbracket \implies$

$ss \ [\sqsubseteq_r] \ \text{merges } f \ qs \ ss \vee$

$\text{merges } f \ qs \ ss = ss \wedge \{q. \exists t. (q,t) \in \text{set } qs \wedge t \sqcup_f \ ss!q \neq ss!q\} \cup (w - \{p\}) \subseteq w \langle \text{proof} \rangle$

**lemma** *iter-properties*[rule-format]: **assumes** *Semilat*  $A \ r \ f$

**shows**  $\llbracket \text{acc } r; \text{pres-type step } n \ A; \text{mono } r \ \text{step } n \ A;$

$\text{bounded step } n; \forall p \in w0. p < n; ss0 \in \text{list } n \ A;$

$\forall p < n. p \notin w0 \longrightarrow \text{stable } r \ \text{step } ss0 \ p \rrbracket \implies$

$iter\ f\ step\ ss0\ w0 = (ss', w')$   
 $\longrightarrow$   
 $ss' \in list\ n\ A \wedge stables\ r\ step\ ss' \wedge ss0\ [\sqsubseteq_r]\ ss' \wedge$   
 $(\forall ts \in list\ n\ A. ss0\ [\sqsubseteq_r]\ ts \wedge stables\ r\ step\ ts \longrightarrow ss' [\sqsubseteq_r] ts) \langle proof \rangle$

**lemma** *kildall-properties*: **assumes** *Semilat A r f*

**shows**  $\llbracket acc\ r; pres\text{-}type\ step\ n\ A; mono\ r\ step\ n\ A;$

$bounded\ step\ n; ss0 \in list\ n\ A \rrbracket \Longrightarrow$

$kildall\ r\ f\ step\ ss0 \in list\ n\ A \wedge$

$stables\ r\ step\ (kildall\ r\ f\ step\ ss0) \wedge$

$ss0\ [\sqsubseteq_r]\ kildall\ r\ f\ step\ ss0 \wedge$

$(\forall ts \in list\ n\ A. ss0\ [\sqsubseteq_r]\ ts \wedge stables\ r\ step\ ts \longrightarrow$

$kildall\ r\ f\ step\ ss0\ [\sqsubseteq_r]\ ts) \langle proof \rangle \langle proof \rangle$

**end**

## 4.10 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
  if pc'=pc+1 then s'  $\sqcup_f$  x
  else if s'  $\sqsubseteq_r$  cert!pc' then x
  else T)
```

**definition** *wtl-inst* :: 's certificate  $\Rightarrow$  's binop  $\Rightarrow$  's ord  $\Rightarrow$  's  $\Rightarrow$   
's step-type  $\Rightarrow$  nat  $\Rightarrow$  's  $\Rightarrow$  's

**where**

```
wtl-inst cert f r T step pc s = merge cert f r T pc (step pc s) (cert!(pc+1))
```

**definition** *wtl-cert* :: 's certificate  $\Rightarrow$  's binop  $\Rightarrow$  's ord  $\Rightarrow$  's  $\Rightarrow$  's  $\Rightarrow$   
's step-type  $\Rightarrow$  nat  $\Rightarrow$  's  $\Rightarrow$  's

**where**

```
wtl-cert cert f r T B step pc s =
(if cert!pc = B then
  wtl-inst cert f r T step pc s
else
  if s  $\sqsubseteq_r$  cert!pc then wtl-inst cert f r T step pc (cert!pc) else T)
```

**primrec** *wtl-inst-list* :: 'a list  $\Rightarrow$  's certificate  $\Rightarrow$  's binop  $\Rightarrow$  's ord  $\Rightarrow$  's  $\Rightarrow$  's  $\Rightarrow$   
's step-type  $\Rightarrow$  nat  $\Rightarrow$  's  $\Rightarrow$  's

**where**

```
wtl-inst-list [] cert f r T B step pc s = s
| wtl-inst-list (i#is) cert f r T B step pc s =
  (let s' = wtl-cert cert f r T B step pc s in
  if s' = T  $\vee$  s = T then T else wtl-inst-list is cert f r T B step (pc+1) s')
```

**definition** *cert-ok* :: 's certificate  $\Rightarrow$  nat  $\Rightarrow$  's  $\Rightarrow$  's  $\Rightarrow$  's set  $\Rightarrow$  bool

**where**

```
cert-ok cert n T B A  $\longleftrightarrow$  ( $\forall i < n. cert!i \in A \wedge cert!i \neq T$ )  $\wedge$  (cert!n = B)
```

**definition** *bottom* :: 'a ord  $\Rightarrow$  'a  $\Rightarrow$  bool

**where**

```
bottom r B  $\longleftrightarrow$  ( $\forall x. B \sqsubseteq_r x$ )
```

**locale** *lbv* = Semilat +

**fixes** *T* :: 'a ( $\top$ )

**fixes** *B* :: 'a ( $\perp$ )

**fixes** *step* :: 'a step-type

**assumes** *top*: top r  $\top$

**assumes** *T-A*:  $\top \in A$

**assumes** *bot*: bottom  $r \perp$   
**assumes** *B-A*:  $\perp \in A$

**fixes** *merge* :: 'a certificate  $\Rightarrow$  nat  $\Rightarrow$  (nat  $\times$  'a) list  $\Rightarrow$  'a  $\Rightarrow$  'a  
**defines** *mrg-def*: merge cert  $\equiv$  LBVSpec.merge cert f r  $\top$

**fixes** *wti* :: 'a certificate  $\Rightarrow$  nat  $\Rightarrow$  'a  $\Rightarrow$  'a  
**defines** *wti-def*: wti cert  $\equiv$  wtl-inst cert f r  $\top$  step

**fixes** *wtc* :: 'a certificate  $\Rightarrow$  nat  $\Rightarrow$  'a  $\Rightarrow$  'a  
**defines** *wtc-def*: wtc cert  $\equiv$  wtl-cert cert f r  $\top \perp$  step

**fixes** *wtl* :: 'b list  $\Rightarrow$  'a certificate  $\Rightarrow$  nat  $\Rightarrow$  'a  $\Rightarrow$  'a  
**defines** *wtl-def*: wtl ins cert  $\equiv$  wtl-inst-list ins cert f r  $\top \perp$  step

**lemma** (in *lbv*) *wti*:

*wti* c pc s = merge c pc (step pc s) (c!(pc+1))  
 <proof>

**lemma** (in *lbv*) *wtc*:

*wtc* c pc s = (if c!pc =  $\perp$  then *wti* c pc s else if s  $\sqsubseteq_r$  c!pc then *wti* c pc (c!pc) else  $\top$ )  
 <proof>

**lemma** *cert-okD1* [intro?]:

*cert-ok* c n T B A  $\Longrightarrow$  pc < n  $\Longrightarrow$  c!pc  $\in$  A  
 <proof>

**lemma** *cert-okD2* [intro?]:

*cert-ok* c n T B A  $\Longrightarrow$  c!n = B  
 <proof>

**lemma** *cert-okD3* [intro?]:

*cert-ok* c n T B A  $\Longrightarrow$  B  $\in$  A  $\Longrightarrow$  pc < n  $\Longrightarrow$  c!Suc pc  $\in$  A  
 <proof>

**lemma** *cert-okD4* [intro?]:

*cert-ok* c n T B A  $\Longrightarrow$  pc < n  $\Longrightarrow$  c!pc  $\neq$  T  
 <proof>

**declare** *Let-def* [simp]

#### 4.10.1 more semilattice lemmas

**lemma** (in *lbv*) *sup-top* [simp, elim]:

**assumes** *x*: x  $\in$  A  
**shows** x  $\sqcup_f$   $\top$  =  $\top$  <proof>

**lemma** (in *lbv*) *plusplussup-top* [simp, elim]:

set xs  $\subseteq$  A  $\Longrightarrow$  xs  $\sqcup_f$   $\top$  =  $\top$   
 <proof>

**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 <proof>

**lemma** (in *lbv*) *bottom-le* [simp, intro!]:  $\perp \sqsubseteq_r$  x

<proof>

**lemma** (in *lbv*) *le-bottom* [simp]: x  $\sqsubseteq_r$   $\perp$  = (x =  $\perp$ )



*<proof>*

### 4.10.2 merge

**lemma** (in *lbv*) *merge-Nil* [*simp*]:

$\text{merge } c \text{ pc } [] \ x = x$  *<proof>*

**lemma** (in *lbv*) *merge-Cons* [*simp*]:

$\text{merge } c \text{ pc } (l\#ls) \ x = \text{merge } c \text{ pc } ls$  (if  $\text{fst } l = \text{pc} + 1$  then  $\text{snd } l \ +\text{-f } x$   
else if  $\text{snd } l \sqsubseteq_r \ c!\text{fst } l$  then  $x$   
else  $\top$ )

*<proof>*

**lemma** (in *lbv*) *merge-Err* [*simp*]:

$\text{snd}'\text{set } ss \subseteq A \implies \text{merge } c \text{ pc } ss \ \top = \top$

*<proof>*

**lemma** (in *lbv*) *merge-not-top*:

$\bigwedge x. \text{snd}'\text{set } ss \subseteq A \implies \text{merge } c \text{ pc } ss \ x \neq \top \implies$   
 $\forall (pc', s') \in \text{set } ss. (pc' \neq \text{pc} + 1 \longrightarrow s' \sqsubseteq_r \ c!\text{pc}')$   
(is  $\bigwedge x. ?\text{set } ss \implies ?\text{merge } ss \ x \implies ?P \ ss$ )*<proof>*

**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  $\forall (pc', s') \in \text{set } ss. pc' \neq \text{pc} + 1 \longrightarrow s' \sqsubseteq_r \ c!\text{pc}'$  then  
 $\text{map } \text{snd } [(p', t') \leftarrow ss. p' = \text{pc} + 1] \sqcup_f x$   
else  $\top$ )

(is  $\bigwedge x. - \implies - \implies ?\text{merge } ss \ x = ?\text{if } ss \ x \text{ is } \bigwedge x. - \implies - \implies ?P \ ss \ x$ )*<proof>*

**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**  $\text{merge } c \text{ pc } ss \ x = (\text{map } \text{snd } [(p', t') \leftarrow ss. p' = \text{pc} + 1] \sqcup_f x)$ *<proof>*

### 4.10.3 wtl-inst-list

**lemmas** [*iff*] = *not-Err-eq*

**lemma** (in *lbv*) *wtl-Nil* [*simp*]:  $\text{wtl } [] \ c \ \text{pc } s = s$

*<proof>*

**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  $\text{wtl } is \ c \ (\text{pc} + 1) \ s'$ )  
*<proof>*

**lemma** (in *lbv*) *wtl-Cons-not-top*:

$\text{wtl } (i\#is) \ c \ \text{pc } s \neq \top =$   
( $\text{wtc } c \ \text{pc } s \neq \top \wedge s \neq \top \wedge \text{wtl } is \ c \ (\text{pc} + 1) \ (\text{wtc } c \ \text{pc } s) \neq \top$ )  
*<proof>*

**lemma** (in *lbv*) *wtl-top* [*simp*]:  $\text{wtl } ls \ c \ \text{pc } \top = \top$

*<proof>*

**lemma** (in *lbv*) *wtl-not-top*:  
 $wtl\ ls\ c\ pc\ s \neq \top \implies s \neq \top$   
 ⟨*proof*⟩

**lemma** (in *lbv*) *wtl-append* [*simp*]:  
 $\bigwedge pc\ s. wtl\ (a@b)\ c\ pc\ s = wtl\ b\ c\ (pc+length\ a)\ (wtl\ a\ c\ pc\ s)$   
 ⟨*proof*⟩

**lemma** (in *lbv*) *wtl-take*:  
 $wtl\ is\ c\ pc\ s \neq \top \implies wtl\ (take\ pc'\ is)\ c\ pc\ s \neq \top$   
 (is ?*wtl is*  $\neq - \implies -$ )⟨*proof*⟩

**lemma** *take-Suc*:  
 $\forall n. n < length\ l \implies take\ (Suc\ n)\ l = (take\ n\ l)@[!n]\ (is\ ?P\ l)$ ⟨*proof*⟩

**lemma** (in *lbv*) *wtl-Suc*:  
 assumes *suc*:  $pc+1 < length\ is$   
 assumes *wtl*:  $wtl\ (take\ pc\ is)\ c\ 0\ s \neq \top$   
 shows  $wtl\ (take\ (pc+1)\ is)\ c\ 0\ s = wtc\ c\ pc\ (wtl\ (take\ pc\ is)\ c\ 0\ s)$ ⟨*proof*⟩

**lemma** (in *lbv*) *wtl-all*:  
 assumes *all*:  $wtl\ is\ c\ 0\ s \neq \top$  (is ?*wtl is*  $\neq -$ )  
 assumes *pc*:  $pc < length\ is$   
 shows  $wtc\ c\ pc\ (wtl\ (take\ pc\ is)\ c\ 0\ s) \neq \top$ ⟨*proof*⟩

#### 4.10.4 preserves-type

**lemma** (in *lbv*) *merge-pres*:  
 assumes *s0*:  $snd'set\ ss \subseteq A$  and *x*:  $x \in A$   
 shows  $merge\ c\ pc\ ss\ x \in A$ ⟨*proof*⟩

**lemma** *pres-typeD2*:  
 $pres\_type\ step\ n\ A \implies s \in A \implies p < n \implies snd'set\ (step\ p\ s) \subseteq A$   
 ⟨*proof*⟩

**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  $wti\ c\ pc\ s \in A$ ⟨*proof*⟩

**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  $wtc\ c\ pc\ s \in A$ ⟨*proof*⟩

**lemma** (in *lbv*) *wtl-pres*:  
 assumes *pres*:  $pres\_type\ step\ (length\ is)\ A$   
 assumes *cert*:  $cert\_ok\ c\ (length\ is)\ \top \perp A$   
 assumes *s*:  $s \in A$   
 assumes *all*:  $wtl\ is\ c\ 0\ s \neq \top$   
 shows  $pc < length\ is \implies wtl\ (take\ pc\ is)\ c\ 0\ s \in A$   
 (is ?*len pc*  $\implies ?wtl\ pc \in A$ )⟨*proof*⟩

**end**

## 4.11 Correctness of the LBV

```

theory LBVCorrect
imports LBVSpec Typing-Framework
begin

locale lbs = lbv +
  fixes s0 :: 'a
  fixes c :: 'a list
  fixes ins :: 'b list
  fixes  $\tau s$  :: 'a list
  defines phi-def:
   $\tau s \equiv \text{map } (\lambda pc. \text{if } c!pc = \perp \text{ then wtl (take pc ins) c 0 } s_0 \text{ else } c!pc)$ 
    [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 lbs) phi-None [intro?]:
   $\llbracket pc < \text{size ins}; c!pc = \perp \rrbracket \implies \tau s!pc = \text{wtl (take pc ins) c 0 } s_0 \langle \text{proof} \rangle$ 
lemma (in lbs) phi-Some [intro?]:
   $\llbracket pc < \text{size ins}; c!pc \neq \perp \rrbracket \implies \tau s!pc = c!pc \langle \text{proof} \rangle$ 
lemma (in lbs) phi-len [simp]: size  $\tau s$  = size ins  $\langle \text{proof} \rangle$ 
lemma (in lbs) wtl-suc-pc:
  assumes all: wtl ins c 0 s0  $\neq \top$ 
  assumes pc: pc+1 < size ins
  shows wtl (take (pc+1) ins) c 0 s0  $\sqsubseteq_r \tau s!(pc+1) \langle \text{proof} \rangle$ 
lemma (in lbs) wtl-stable:
  assumes wtl: wtl ins c 0 s0  $\neq \top$ 
  assumes s0: s0  $\in A$  and pc: pc < size ins
  shows stable r step  $\tau s$  pc  $\langle \text{proof} \rangle$ 
lemma (in lbs) phi-not-top:
  assumes wtl: wtl ins c 0 s0  $\neq \top$  and pc: pc < size ins
  shows  $\tau s!pc \neq \top \langle \text{proof} \rangle$ 
lemma (in lbs) phi-in-A:
  assumes wtl: wtl ins c 0 s0  $\neq \top$  and s0: s0  $\in A$ 
  shows  $\tau s \in \text{list (size ins) } A \langle \text{proof} \rangle$ 
lemma (in lbs) phi0:
  assumes wtl: wtl ins c 0 s0  $\neq \top$  and 0: 0 < size ins
  shows s0  $\sqsubseteq_r \tau s!0 \langle \text{proof} \rangle$ 

theorem (in lbs) wtl-sound:
  assumes wtl: wtl ins c 0 s0  $\neq \top$  and s0: s0  $\in A$ 
  shows  $\exists \tau s. \text{wt-step } r \top \text{step } \tau s \langle \text{proof} \rangle$ 

theorem (in lbs) wtl-sound-strong:
  assumes wtl: wtl ins c 0 s0  $\neq \top$ 
  assumes s0: s0  $\in A$  and ins: 0 < size ins
  shows  $\exists \tau s \in \text{list (size ins) } A. \text{wt-step } r \top \text{step } \tau s \wedge s_0 \sqsubseteq_r \tau s!0 \langle \text{proof} \rangle$ 
end

```

## 4.12 Completeness of the LBV

**theory** *LBVComplete*

**imports** *LBVSpec Typing-Framework*

**begin**

**definition** *is-target* :: 's step-type  $\Rightarrow$  's list  $\Rightarrow$  nat  $\Rightarrow$  bool **where**

*is-target* step  $\tau s$   $pc' \longleftrightarrow (\exists pc\ s'. pc' \neq pc+1 \wedge pc < size\ \tau s \wedge (pc',s') \in set\ (step\ pc\ (\tau s!pc)))$

**definition** *make-cert* :: 's step-type  $\Rightarrow$  's list  $\Rightarrow$  's  $\Rightarrow$  's certificate **where**

*make-cert* step  $\tau s$   $B = map\ (\lambda pc. if\ is-target\ step\ \tau s\ pc\ then\ \tau s!pc\ else\ B)\ [0..<size\ \tau s]\ @\ [B]$

**lemma** [*code*]:

*is-target* step  $\tau s$   $pc' =$

*list-ex*  $(\lambda pc. pc' \neq pc+1 \wedge List.member\ (map\ fst\ (step\ pc\ (\tau s!pc)))\ pc')\ [0..<size\ \tau s]\langle proof \rangle$

**locale** *lbvc* = *lbv* +

**fixes**  $\tau s :: 'a\ list$

**fixes**  $c :: 'a\ list$

**defines** *cert-def*:  $c \equiv make-cert\ step\ \tau s\ \perp$

**assumes** *mono*: *mono*  $r\ step\ (size\ \tau s)\ A$

**assumes** *pres*: *pres-type*  $step\ (size\ \tau s)\ A$

**assumes**  $\tau s$ :  $\forall pc < size\ \tau s. \tau s!pc \in A \wedge \tau s!pc \neq \top$

**assumes** *bounded*: *bounded*  $step\ (size\ \tau s)$

**assumes** *B-neq-T*:  $\perp \neq \top$

**lemma** (**in** *lbvc*) *cert*: *cert-ok*  $c\ (size\ \tau s)\ \top\ \perp\ A\langle proof \rangle$

**lemmas** [*simp del*] = *split-paired-Ex*

**lemma** (**in** *lbvc*) *cert-target* [*intro?*]:

$\llbracket (pc',s') \in set\ (step\ pc\ (\tau s!pc));$   
 $pc' \neq pc+1; pc < size\ \tau s; pc' < size\ \tau s \rrbracket$   
 $\implies c!pc' = \tau s!pc'\langle proof \rangle$

**lemma** (**in** *lbvc*) *cert-approx* [*intro?*]:

$\llbracket pc < size\ \tau s; c!pc \neq \perp \rrbracket \implies c!pc = \tau s!pc\langle proof \rangle$

**lemma** (**in** *lbv*) *le-top* [*simp, intro*]:  $x <=-r\ \top\langle proof \rangle$

**lemma** (**in** *lbv*) *merge-mono*:

**assumes** *less*:  $set\ ss_2\ \{\sqsubseteq_r\}\ set\ ss_1$

**assumes**  $x$ :  $x \in A$

**assumes**  $ss_1$ :  $snd'set\ ss_1 \subseteq A$

**assumes**  $ss_2$ :  $snd'set\ ss_2 \subseteq A$

**shows** *merge*  $c\ pc\ ss_2\ x \sqsubseteq_r\ merge\ c\ pc\ ss_1\ x\ (is\ ?s_2 \sqsubseteq_r\ ?s_1)\langle proof \rangle$

**lemma** (**in** *lbvc*) *wti-mono*:

**assumes** *less*:  $s_2 \sqsubseteq_r\ s_1$

**assumes**  $pc$ :  $pc < size\ \tau s$  **and**  $s_1$ :  $s_1 \in A$  **and**  $s_2$ :  $s_2 \in A$

**shows** *wti*  $c\ pc\ s_2 \sqsubseteq_r\ wti\ c\ pc\ s_1\ (is\ ?s_2' \sqsubseteq_r\ ?s_1')\langle proof \rangle$

**lemma** (**in** *lbvc*) *wtc-mono*:

**assumes** *less*:  $s_2 \sqsubseteq_r\ s_1$

**assumes**  $pc$ :  $pc < size\ \tau s$  **and**  $s_1$ :  $s_1 \in A$  **and**  $s_2$ :  $s_2 \in A$

**shows** *wtc*  $c\ pc\ s_2 \sqsubseteq_r\ wtc\ c\ pc\ s_1\ (is\ ?s_2' \sqsubseteq_r\ ?s_1')\langle proof \rangle$

**lemma** (**in** *lbv*) *top-le-conv* [*simp*]:  $\top \sqsubseteq_r\ x = (x = \top)\langle proof \rangle$

**lemma** (**in** *lbv*) *neq-top* [*simp, elim*]:  $\llbracket x \sqsubseteq_r\ y; y \neq \top \rrbracket \implies x \neq \top\langle proof \rangle$

**lemma** (in *lbvc*) *stable-wti*:

**assumes** *stable*: *stable r step  $\tau s$  pc* **and** *pc*: *pc < size  $\tau s$*

**shows** *wti c pc* ( *$\tau s!pc$* )  $\neq \top$  (proof)

**lemma** (in *lbvc*) *wti-less*:

**assumes** *stable*: *stable r step  $\tau s$  pc* **and** *suc-pc*: *Suc pc < size  $\tau s$*

**shows** *wti c pc* ( *$\tau s!pc$* )  $\sqsubseteq_r \tau s!Suc\ pc$  (is ?*wti*  $\sqsubseteq_r$  -) (proof)

**lemma** (in *lbvc*) *stable-wtc*:

**assumes** *stable*: *stable r step  $\tau s$  pc* **and** *pc*: *pc < size  $\tau s$*

**shows** *wtc c pc* ( *$\tau s!pc$* )  $\neq \top$  (proof)

**lemma** (in *lbvc*) *wtc-less*:

**assumes** *stable*: *stable r step  $\tau s$  pc* **and** *suc-pc*: *Suc pc < size  $\tau s$*

**shows** *wtc c pc* ( *$\tau s!pc$* )  $\sqsubseteq_r \tau s!Suc\ pc$  (is ?*wtc*  $\sqsubseteq_r$  -) (proof)

**lemma** (in *lbvc*) *wt-step-wtl-lemma*:

**assumes** *wt-step*: *wt-step r  $\top$  step  $\tau s$*

**shows**  $\bigwedge pc\ s. pc + size\ ls = size\ \tau s \implies s \sqsubseteq_r \tau s!pc \implies s \in A \implies s \neq \top \implies$   
 $wtl\ ls\ c\ pc\ s \neq \top$

(is  $\bigwedge pc\ s. - \implies - \implies - \implies - \implies ?wtl\ ls\ pc\ s \neq -$ ) (proof)

**theorem** (in *lbvc*) *wtl-complete*:

**assumes** *wt*: *wt-step r  $\top$  step  $\tau s$*

**assumes** *s*: *s  $\sqsubseteq_r \tau s!0$  s  $\in A$  s  $\neq \top$*  **and** *eq*: *size ins = size  $\tau s$*

**shows** *wtl ins c 0 s*  $\neq \top$  (proof)

**end**

## 4.13 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*:  
 $(C, D) \in \text{subcls1 } P \implies \text{super } P \ C = D$   
 <proof>

**primrec** *the-Class* :: ty  $\Rightarrow$  cname  
**where**  
*the-Class* (Class C) = C

**definition** *sup* :: 'c prog  $\Rightarrow$  ty  $\Rightarrow$  ty  $\Rightarrow$  ty err  
**where**  
*sup* P T<sub>1</sub> T<sub>2</sub>  $\equiv$   
 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))  
 <proof>

**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*:  
 $\text{wf-prog } m \ P \implies \text{is-class } P \ C = P \vdash \ C \preceq^* \text{Object}$  <proof>

**lemma** *subcls-antisym*:  
 $\llbracket \text{wf-prog } m \ P; P \vdash \ C \preceq^* \ D; P \vdash \ D \preceq^* \ C \rrbracket \implies C = D$

$\langle proof \rangle$

**lemma** *widen-antisym*:

$\llbracket wf\text{-prog } m P; P \vdash T \leq U; P \vdash U \leq T \rrbracket \implies T = U \langle proof \rangle$

**lemma** *order-widen* [*intro, simp*]:

$wf\text{-prog } m P \implies order (subtype P) \langle proof \rangle$

**lemma** *NT-widen*:

$P \vdash NT \leq T = (T = NT \vee (\exists C. T = Class C)) \langle proof \rangle$

**lemma** *Class-widen2*:  $P \vdash Class C \leq T = (\exists D. T = Class D \wedge P \vdash C \preceq^* D) \langle proof \rangle$

**lemma** *wf-converse-subcls1-impl-acc-subtype*:

$wf ((subcls1 P) \hat{-} 1) \implies acc (subtype P) \langle proof \rangle$

**lemma** *wf-subtype-acc* [*intro, simp*]:

$wf\text{-prog } wf\text{-mb } P \implies acc (subtype P) \langle proof \rangle$

**lemma** *exec-lub-refl* [*simp*]:  $exec\text{-lub } r f T T = T \langle proof \rangle$

**lemma** *closed-err-types*:

$wf\text{-prog } wf\text{-mb } P \implies closed (err (types P)) (lift2 (sup P)) \langle proof \rangle$

**lemma** *sup-subtype-greater*:

$\llbracket wf\text{-prog } wf\text{-mb } P; is\text{-type } P t1; is\text{-type } P t2; sup P t1 t2 = OK s \rrbracket$   
 $\implies subtype P t1 s \wedge subtype P t2 s \langle proof \rangle$

**lemma** *sup-subtype-smallest*:

$\llbracket wf\text{-prog } wf\text{-mb } P; is\text{-type } P a; is\text{-type } P b; is\text{-type } P c;$   
 $subtype P a c; subtype P b c; sup P a b = OK d \rrbracket$   
 $\implies subtype P d c \langle proof \rangle$

**lemma** *sup-exists*:

$\llbracket subtype P a c; subtype P b c \rrbracket \implies EX T. sup P a b = OK T \langle proof \rangle$

**lemma** *err-semilat-JType-esl*:

$wf\text{-prog } wf\text{-mb } P \implies err\text{-semilat } (esl P) \langle proof \rangle$

**end**

## 4.14 The JVM Type System as Semilattice

**theory** *JVM-SemiType* **imports** *SemiType* **begin**

**type-synonym**  $ty_l = ty\ err\ list$   
**type-synonym**  $ty_s = ty\ list$   
**type-synonym**  $ty_i = ty_s \times ty_l$   
**type-synonym**  $ty_i' = ty_i\ option$   
**type-synonym**  $ty_m = ty_i'\ list$   
**type-synonym**  $ty_P = mname \Rightarrow cname \Rightarrow ty_m$

**definition**  $stk\ esl :: 'c\ prog \Rightarrow nat \Rightarrow ty_s\ esl$   
**where**  
 $stk\ esl\ P\ mxs \equiv upto\ esl\ mxs\ (SemiType.esl\ P)$

**definition**  $loc\ sl :: 'c\ prog \Rightarrow nat \Rightarrow ty_l\ sl$   
**where**  
 $loc\ sl\ P\ mxl \equiv Listn.sl\ mxl\ (Err.sl\ (SemiType.esl\ P))$

**definition**  $sl :: 'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ sl$   
**where**  
 $sl\ P\ mxs\ mxl \equiv$   
 $Err.sl(Opt.esl(Product.esl(stk\ esl\ P\ mxs)\ (Err.esl(loc\ sl\ P\ mxl))))$

**definition**  $states :: 'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ set$   
**where**  $states\ P\ mxs\ mxl \equiv fst(sl\ P\ mxs\ mxl)$

**definition**  $le :: 'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ ord$   
**where**  
 $le\ P\ mxs\ mxl \equiv fst(snd(sl\ P\ mxs\ mxl))$

**definition**  $sup :: 'c\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ err\ binop$   
**where**  
 $sup\ P\ mxs\ mxl \equiv snd(snd(sl\ P\ mxs\ mxl))$

**definition**  $sup\ ty\ opt :: ['c\ prog, ty\ err, ty\ err] \Rightarrow bool$   
 $(- | - - <= T - [71, 71, 71] 70)$

**where**  
 $sup\ ty\ opt\ P \equiv Err.le\ (subtype\ P)$

**definition**  $sup\ state :: ['c\ prog, ty_i, ty_i] \Rightarrow bool$   
 $(- | - - <= i - [71, 71, 71] 70)$

**where**  
 $sup\ state\ P \equiv Product.le\ (Listn.le\ (subtype\ P))\ (Listn.le\ (sup\ ty\ opt\ P))$

**definition**  $sup\ state\ opt :: ['c\ prog, ty_i', ty_i'] \Rightarrow bool$   
 $(- | - - <= ' - [71, 71, 71] 70)$

**where**  
 $sup\ state\ opt\ P \equiv Opt.le\ (sup\ state\ P)$

**abbreviation**



$sup\text{-}loc :: [c\ prog, ty_i, ty_l] \Rightarrow bool \ (- \mid - \ [-<=T] \ - \ [71, 71, 71] \ 70)$   
**where**  $P \mid - \ LT \ [-<=T] \ LT' \equiv list\text{-}all2 \ (sup\text{-}ty\text{-}opt \ P) \ LT \ LT'$

**notation** (*xsymbols*)

$sup\text{-}ty\text{-}opt \ (- \vdash \ - \ \leq_{\top} \ - \ [71, 71, 71] \ 70)$  **and**  
 $sup\text{-}state \ (- \vdash \ - \ \leq_i \ - \ [71, 71, 71] \ 70)$  **and**  
 $sup\text{-}state\text{-}opt \ (- \vdash \ - \ \leq' \ - \ [71, 71, 71] \ 70)$  **and**  
 $sup\text{-}loc \ (- \vdash \ - \ [\leq_{\top}] \ - \ [71, 71, 71] \ 70)$

#### 4.14.1 Unfolding

**lemma** *JVM-states-unfold*:

$states \ P \ mxs \ mxl \equiv err(opt((Union \ {list \ n \ (types \ P) \ \mid n. \ n \ \leq \ mxs\}) \ <*\rangle \ list \ mxl \ (err(types \ P))))\langle proof \rangle$

**lemma** *JVM-le-unfold*:

$le \ P \ m \ n \equiv Err.le(Opt.le(Product.le(Listn.le(subtype \ P))(Listn.le(Err.le(subtype \ P)))))\langle proof \rangle$

**lemma** *sl-def2*:

$JVM\text{-}SemiType.sl \ P \ mxs \ mxl \equiv (states \ P \ mxs \ mxl, \ JVM\text{-}SemiType.le \ P \ mxs \ mxl, \ JVM\text{-}SemiType.sup \ P \ mxs \ mxl)\langle proof \rangle$

**lemma** *JVM-le-conv*:

$le \ P \ m \ n \ (OK \ t1) \ (OK \ t2) = P \vdash \ t1 \ \leq' \ t2\langle proof \rangle$

**lemma** *JVM-le-Err-conv*:

$le \ P \ m \ n = Err.le \ (sup\text{-}state\text{-}opt \ P)\langle proof \rangle$

**lemma** *err-le-unfold* [*iff*]:

$Err.le \ r \ (OK \ a) \ (OK \ b) = r \ a \ b\langle proof \rangle$

#### 4.14.2 Semilattice

**lemma** *order-sup-state-opt* [*intro, simp*]:

$wf\text{-}prog \ wf\text{-}mb \ P \Longrightarrow order \ (sup\text{-}state\text{-}opt \ P)\langle proof \rangle$

**lemma** *semilat-JVM* [*intro?*]:

$wf\text{-}prog \ wf\text{-}mb \ P \Longrightarrow semilat \ (JVM\text{-}SemiType.sl \ P \ mxs \ mxl)\langle proof \rangle$

**lemma** *acc-JVM* [*intro*]:

$wf\text{-}prog \ wf\text{-}mb \ P \Longrightarrow acc \ (JVM\text{-}SemiType.le \ P \ mxs \ mxl)\langle proof \rangle$

#### 4.14.3 Widening with $\top$

**lemma** *subtype-refl*[*iff*]:  $subtype \ P \ t \ t \ \langle proof \rangle$

**lemma** *sup-ty-opt-refl* [*iff*]:  $P \vdash \ T \ \leq_{\top} \ T\langle proof \rangle$

**lemma** *Err-any-conv* [*iff*]:  $P \vdash \ Err \ \leq_{\top} \ T = (T = Err)\langle proof \rangle$

**lemma** *any-Err* [*iff*]:  $P \vdash \ T \ \leq_{\top} \ Err\langle proof \rangle$

**lemma** *OK-OK-conv* [*iff*]:

$P \vdash \ OK \ T \ \leq_{\top} \ OK \ T' = P \vdash \ T \ \leq \ T'\langle proof \rangle$

**lemma** *any-OK-conv* [*iff*]:

$P \vdash \ X \ \leq_{\top} \ OK \ T' = (\exists T. X = OK \ T \wedge P \vdash \ T \ \leq \ T')\langle proof \rangle$

**lemma** *OK-any-conv*:

$P \vdash \ OK \ T \ \leq_{\top} \ X = (X = Err \vee (\exists T'. X = OK \ T' \wedge P \vdash \ T \ \leq \ T'))\langle proof \rangle$

**lemma** *sup-ty-opt-trans* [*intro?*, *trans*]:

$\llbracket P \vdash \ a \ \leq_{\top} \ b; P \vdash \ b \ \leq_{\top} \ c \rrbracket \Longrightarrow P \vdash \ a \ \leq_{\top} \ c\langle proof \rangle$

#### 4.14.4 Stack and Registers

**lemma** *stk-convert*:

$P \vdash ST \leq ST' = \text{Listn.le } (\text{subtype } P) ST ST' \langle \text{proof} \rangle$   
**lemma** *sup-loc-refl* [iff]:  $P \vdash LT \leq_{\top} LT \langle \text{proof} \rangle$   
**lemmas** *sup-loc-Cons1* [iff] = *list-all2-Cons1* [of *sup-ty-opt*  $P$ ] **for**  $P$

**lemma** *sup-loc-def*:

$P \vdash LT \leq_{\top} LT' \equiv \text{Listn.le } (\text{sup-ty-opt } P) LT LT' \langle \text{proof} \rangle$

**lemma** *sup-loc-widens-conv* [iff]:

$P \vdash \text{map } OK Ts \leq_{\top} \text{map } OK Ts' = P \vdash Ts \leq Ts' \langle \text{proof} \rangle$

**lemma** *sup-loc-trans* [intro?, trans]:

$\llbracket P \vdash a \leq_{\top} b; P \vdash b \leq_{\top} c \rrbracket \implies P \vdash a \leq_{\top} c \langle \text{proof} \rangle$

#### 4.14.5 State Type

**lemma** *sup-state-conv* [iff]:

$P \vdash (ST, LT) \leq_i (ST', LT') = (P \vdash ST \leq ST' \wedge P \vdash LT \leq_{\top} LT') \langle \text{proof} \rangle$

**lemma** *sup-state-conv2*:

$P \vdash s1 \leq_i s2 = (P \vdash \text{fst } s1 \leq \text{fst } s2 \wedge P \vdash \text{snd } s1 \leq_{\top} \text{snd } s2) \langle \text{proof} \rangle$

**lemma** *sup-state-refl* [iff]:  $P \vdash s \leq_i s \langle \text{proof} \rangle$

**lemma** *sup-state-trans* [intro?, trans]:

$\llbracket P \vdash a \leq_i b; P \vdash b \leq_i c \rrbracket \implies P \vdash a \leq_i c \langle \text{proof} \rangle$

**lemma** *sup-state-opt-None-any* [iff]:

$P \vdash \text{None} \leq' s \langle \text{proof} \rangle$

**lemma** *sup-state-opt-any-None* [iff]:

$P \vdash s \leq' \text{None} = (s = \text{None}) \langle \text{proof} \rangle$

**lemma** *sup-state-opt-Some-Some* [iff]:

$P \vdash \text{Some } a \leq' \text{Some } b = P \vdash a \leq_i b \langle \text{proof} \rangle$

**lemma** *sup-state-opt-any-Some*:

$P \vdash (\text{Some } s) \leq' X = (\exists s'. X = \text{Some } s' \wedge P \vdash s \leq_i s') \langle \text{proof} \rangle$

**lemma** *sup-state-opt-refl* [iff]:  $P \vdash s \leq' s \langle \text{proof} \rangle$

**lemma** *sup-state-opt-trans* [intro?, trans]:

$\llbracket P \vdash a \leq' b; P \vdash b \leq' c \rrbracket \implies P \vdash a \leq' c \langle \text{proof} \rangle$

**end**

## 4.15 Effect of Instructions on the State Type

```
theory Effect
imports JVM-SemiType ../JVM/JVMExceptions
begin
```

— FIXME

```
locale prog =
  fixes P :: 'a prog
```

```
locale jvm-method = prog +
  fixes mxs :: nat
  fixes mxl0 :: nat
  fixes Ts :: ty list
  fixes Tτ :: ty
  fixes is :: instr list
  fixes xt :: ex-table
```

```
fixes mxl :: nat
defines mxl-def: mxl ≡ 1+size Ts+mxl0
```

Program counter of successor instructions:

```
primrec succs :: instr ⇒ tyi ⇒ pc ⇒ pc list where
  succs (Load idx) τ pc = [pc+1]
| succs (Store idx) τ pc = [pc+1]
| succs (Push v) τ pc = [pc+1]
| succs (Getfield F C) τ pc = [pc+1]
| succs (Putfield F C) τ pc = [pc+1]
| succs (New C) τ pc = [pc+1]
| succs (Checkcast C) τ pc = [pc+1]
| succs Pop τ pc = [pc+1]
| succs IAdd τ pc = [pc+1]
| succs CmpEq τ pc = [pc+1]
| succs-IfFalse:
  succs (IfFalse b) τ pc = [pc+1, nat (int pc + b)]
| succs-Goto:
  succs (Goto b) τ pc = [nat (int pc + b)]
| succs-Return:
  succs Return τ pc = []
| succs-Invoke:
  succs (Invoke M n) τ pc = (if (fst τ)!n = NT then [] else [pc+1])
| succs-Throw:
  succs Throw τ pc = []
```

Effect of instruction on the state type:

```
fun the-class :: ty ⇒ cname where
  the-class (Class C) = C
```

```
fun effi :: instr × 'm prog × tyi ⇒ tyi where
  effi-Load:
    effi (Load n, P, (ST, LT)) = (ok-val (LT ! n) # ST, LT)
| effi-Store:
    effi (Store n, P, (T#ST, LT)) = (ST, LT[n:= OK T])
| effi-Push:
```

$\text{eff}_i (\text{Push } v, P, (ST, LT)) = (\text{the } (\text{typeof } v) \# ST, LT)$   
 $\text{eff}_i\text{-Getfield:}$   
 $\text{eff}_i (\text{Getfield } F C, P, (T\#ST, LT)) = (\text{snd } (\text{field } P C F) \# ST, LT)$   
 $\text{eff}_i\text{-Putfield:}$   
 $\text{eff}_i (\text{Putfield } F C, P, (T_1\#T_2\#ST, LT)) = (ST, LT)$   
 $\text{eff}_i\text{-New:}$   
 $\text{eff}_i (\text{New } C, P, (ST, LT)) = (\text{Class } C \# ST, LT)$   
 $\text{eff}_i\text{-Checkcast:}$   
 $\text{eff}_i (\text{Checkcast } C, P, (T\#ST, LT)) = (\text{Class } C \# ST, LT)$   
 $\text{eff}_i\text{-Pop:}$   
 $\text{eff}_i (\text{Pop}, P, (T\#ST, LT)) = (ST, LT)$   
 $\text{eff}_i\text{-IAdd:}$   
 $\text{eff}_i (\text{IAdd}, P, (T_1\#T_2\#ST, LT)) = (\text{Integer}\#ST, LT)$   
 $\text{eff}_i\text{-CmpEq:}$   
 $\text{eff}_i (\text{CmpEq}, P, (T_1\#T_2\#ST, LT)) = (\text{Boolean}\#ST, LT)$   
 $\text{eff}_i\text{-IfFalse:}$   
 $\text{eff}_i (\text{IfFalse } b, P, (T_1\#ST, LT)) = (ST, LT)$   
 $\text{eff}_i\text{-Invoke:}$   
 $\text{eff}_i (\text{Invoke } M n, P, (ST, LT)) =$   
 $(\text{let } C = \text{the-class } (ST!n); (D, Ts, T_r, b) = \text{method } P C M$   
 $\text{in } (T_r \# \text{drop } (n+1) ST, LT))$   
 $\text{eff}_i\text{-Goto:}$   
 $\text{eff}_i (\text{Goto } n, P, s) = s$

**fun** *is-relevant-class* :: *instr*  $\Rightarrow$  *'m prog*  $\Rightarrow$  *cname*  $\Rightarrow$  *bool* **where**  
 $\text{rel-Getfield:}$   
 $\text{is-relevant-class } (\text{Getfield } F D) = (\lambda P C. P \vdash \text{NullPointer} \preceq^* C)$   
 $\text{rel-Putfield:}$   
 $\text{is-relevant-class } (\text{Putfield } F D) = (\lambda P C. P \vdash \text{NullPointer} \preceq^* C)$   
 $\text{rel-Checkcast:}$   
 $\text{is-relevant-class } (\text{Checkcast } D) = (\lambda P C. P \vdash \text{ClassCast} \preceq^* C)$   
 $\text{rel-New:}$   
 $\text{is-relevant-class } (\text{New } D) = (\lambda P C. P \vdash \text{OutOfMemory} \preceq^* C)$   
 $\text{rel-Throw:}$   
 $\text{is-relevant-class } \text{Throw} = (\lambda P C. \text{True})$   
 $\text{rel-Invoke:}$   
 $\text{is-relevant-class } (\text{Invoke } M n) = (\lambda P C. \text{True})$   
 $\text{rel-default:}$   
 $\text{is-relevant-class } i = (\lambda P C. \text{False})$

**definition** *is-relevant-entry* :: *'m prog*  $\Rightarrow$  *instr*  $\Rightarrow$  *pc*  $\Rightarrow$  *ex-entry*  $\Rightarrow$  *bool* **where**  
 $\text{is-relevant-entry } P i pc e \longleftrightarrow (\text{let } (f, t, C, h, d) = e \text{ in } \text{is-relevant-class } i P C \wedge pc \in \{f..<t\})$

**definition** *relevant-entries* :: *'m prog*  $\Rightarrow$  *instr*  $\Rightarrow$  *pc*  $\Rightarrow$  *ex-table*  $\Rightarrow$  *ex-table* **where**  
 $\text{relevant-entries } P i pc = \text{filter } (\text{is-relevant-entry } P i pc)$

**definition** *xcpt-eff* :: *instr*  $\Rightarrow$  *'m prog*  $\Rightarrow$  *pc*  $\Rightarrow$  *ty<sub>i</sub>*  
 $\Rightarrow$  *ex-table*  $\Rightarrow$   $(pc \times ty_i)$  *list* **where**  
 $\text{xcpt-eff } i P pc \tau et = (\text{let } (ST, LT) = \tau \text{ in}$   
 $\text{map } (\lambda (f, t, C, h, d). (h, \text{Some } (\text{Class } C \# \text{drop } (\text{size } ST - d) ST, LT))) (\text{relevant-entries } P i pc et))$

**definition** *norm-eff* :: *instr*  $\Rightarrow$  *'m prog*  $\Rightarrow$  *nat*  $\Rightarrow$  *ty<sub>i</sub>*  $\Rightarrow$   $(pc \times ty_i)$  *list* **where**  
 $\text{norm-eff } i P pc \tau = \text{map } (\lambda pc'. (pc', \text{Some } (\text{eff}_i (i, P, \tau)))) (\text{succs } i \tau pc)$

**definition**  $\text{eff} :: \text{instr} \Rightarrow 'm \text{ prog} \Rightarrow \text{pc} \Rightarrow \text{ex-table} \Rightarrow \text{ty}_i' \Rightarrow (\text{pc} \times \text{ty}_i')$  list **where**  
 $\text{eff } i \text{ P pc et } t = (\text{case } t \text{ of}$   
 $\text{None} \Rightarrow []$   
 $| \text{Some } \tau \Rightarrow (\text{norm-eff } i \text{ P pc } \tau) @ (\text{xcpt-eff } i \text{ P pc } \tau \text{ et}))$

**lemma**  $\text{eff-None}$ :  
 $\text{eff } i \text{ P pc xt None} = []$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{eff-Some}$ :  
 $\text{eff } i \text{ P pc xt (Some } \tau) = \text{norm-eff } i \text{ P pc } \tau @ \text{xcpt-eff } i \text{ P pc } \tau \text{ xt}$   
 $\langle \text{proof} \rangle$

Conditions under which  $\text{eff}$  is applicable:

**fun**  $\text{app}_i :: \text{instr} \times 'm \text{ prog} \times \text{pc} \times \text{nat} \times \text{ty} \times \text{ty}_i \Rightarrow \text{bool}$  **where**  
 $\text{app}_i\text{-Load}$ :  
 $\text{app}_i (\text{Load } n, P, \text{pc}, \text{mxs}, T_r, (ST, LT)) =$   
 $(n < \text{length } LT \wedge LT ! n \neq \text{Err} \wedge \text{length } ST < \text{mxs})$   
 $| \text{app}_i\text{-Store}$ :  
 $\text{app}_i (\text{Store } n, P, \text{pc}, \text{mxs}, T_r, (T\#ST, LT)) =$   
 $(n < \text{length } LT)$   
 $| \text{app}_i\text{-Push}$ :  
 $\text{app}_i (\text{Push } v, P, \text{pc}, \text{mxs}, T_r, (ST, LT)) =$   
 $(\text{length } ST < \text{mxs} \wedge \text{typeof } v \neq \text{None})$   
 $| \text{app}_i\text{-Getfield}$ :  
 $\text{app}_i (\text{Getfield } F \ C, P, \text{pc}, \text{mxs}, T_r, (T\#ST, LT)) =$   
 $(\exists T_f. P \vdash C \text{ sees } F:T_f \text{ in } C \wedge P \vdash T \leq \text{Class } C)$   
 $| \text{app}_i\text{-Putfield}$ :  
 $\text{app}_i (\text{Putfield } F \ C, P, \text{pc}, \text{mxs}, T_r, (T_1\#T_2\#ST, LT)) =$   
 $(\exists T_f. P \vdash C \text{ sees } F:T_f \text{ in } C \wedge P \vdash T_2 \leq (\text{Class } C) \wedge P \vdash T_1 \leq T_f)$   
 $| \text{app}_i\text{-New}$ :  
 $\text{app}_i (\text{New } C, P, \text{pc}, \text{mxs}, T_r, (ST, LT)) =$   
 $(\text{is-class } P \ C \wedge \text{length } ST < \text{mxs})$   
 $| \text{app}_i\text{-Checkcast}$ :  
 $\text{app}_i (\text{Checkcast } C, P, \text{pc}, \text{mxs}, T_r, (T\#ST, LT)) =$   
 $(\text{is-class } P \ C \wedge \text{is-refT } T)$   
 $| \text{app}_i\text{-Pop}$ :  
 $\text{app}_i (\text{Pop}, P, \text{pc}, \text{mxs}, T_r, (T\#ST, LT)) =$   
 $\text{True}$   
 $| \text{app}_i\text{-IAdd}$ :  
 $\text{app}_i (\text{IAdd } P, \text{pc}, \text{mxs}, T_r, (T_1\#T_2\#ST, LT)) = (T_1 = T_2 \wedge T_1 = \text{Integer})$   
 $| \text{app}_i\text{-CmpEq}$ :  
 $\text{app}_i (\text{CmpEq}, P, \text{pc}, \text{mxs}, T_r, (T_1\#T_2\#ST, LT)) =$   
 $(T_1 = T_2 \vee \text{is-refT } T_1 \wedge \text{is-refT } T_2)$   
 $| \text{app}_i\text{-IfFalse}$ :  
 $\text{app}_i (\text{IfFalse } b, P, \text{pc}, \text{mxs}, T_r, (\text{Boolean}\#ST, LT)) =$   
 $(0 \leq \text{int } \text{pc} + b)$   
 $| \text{app}_i\text{-Goto}$ :  
 $\text{app}_i (\text{Goto } b, P, \text{pc}, \text{mxs}, T_r, s) =$   
 $(0 \leq \text{int } \text{pc} + b)$   
 $| \text{app}_i\text{-Return}$ :  
 $\text{app}_i (\text{Return}, P, \text{pc}, \text{mxs}, T_r, (T\#ST, LT)) =$   
 $(P \vdash T \leq T_r)$

| *app<sub>i</sub>-Throw*:  
 $app_i (Throw, P, pc, mxs, T_r, (T\#ST,LT)) =$   
 $is-refT T$

| *app<sub>i</sub>-Invoke*:  
 $app_i (Invoke M n, P, pc, mxs, T_r, (ST,LT)) =$   
 $(n < length ST \wedge$   
 $(ST!n \neq NT \rightarrow$   
 $(\exists C D Ts T m. ST!n = Class C \wedge P \vdash C sees M:Ts \rightarrow T = m in D \wedge$   
 $P \vdash rev (take n ST) [\leq] Ts)))$

| *app<sub>i</sub>-default*:  
 $app_i (i,P, pc,mxs,T_r,s) = False$

**definition** *xcpt-app* :: *instr*  $\Rightarrow$  *'m prog*  $\Rightarrow$  *pc*  $\Rightarrow$  *nat*  $\Rightarrow$  *ex-table*  $\Rightarrow$  *ty<sub>i</sub>*  $\Rightarrow$  *bool* **where**  
 $xcpt-app i P pc mxs xt \tau \longleftrightarrow (\forall (f,t,C,h,d) \in set (relevant-entries P i pc xt). is-class P C \wedge d \leq$   
 $size (fst \tau) \wedge d < mxs)$

**definition** *app* :: *instr*  $\Rightarrow$  *'m prog*  $\Rightarrow$  *nat*  $\Rightarrow$  *ty*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *ex-table*  $\Rightarrow$  *ty<sub>i</sub>'*  $\Rightarrow$  *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*:  
 $app i P mxs T_r pc mpc xt (Some \tau) =$   
 $(app_i (i,P,pc,mxs,T_r,\tau) \wedge xcpt-app i P pc mxs xt \tau \wedge$   
 $(\forall (pc',s') \in set (eff i P pc xt (Some \tau)). pc' < mpc))$   
 $\langle proof \rangle$

**locale** *eff* = *jvm-method* +  
**fixes** *eff<sub>i</sub>* **and** *app<sub>i</sub>* **and** *eff* **and** *app*  
**fixes** *norm-eff* **and** *xcpt-app* **and** *xcpt-eff*

**fixes** *mpc*  
**defines** *mpc*  $\equiv$  *size is*

**defines** *eff<sub>i</sub>* *i*  $\tau \equiv Effect.eff_i (i,P,\tau)$   
**notes** *eff<sub>i</sub>-simps* [*simp*] = *Effect.eff<sub>i</sub>.simps* [**where** *P* = *P*, *folded eff<sub>i</sub>-def*]

**defines** *app<sub>i</sub>* *i* *pc*  $\tau \equiv Effect.app_i (i, P, pc, mxs, T_r, \tau)$   
**notes** *app<sub>i</sub>-simps* [*simp*] = *Effect.app<sub>i</sub>.simps* [**where** *P*=*P* **and** *mxs*=*mxs* **and** *T<sub>r</sub>*=*T<sub>r</sub>*, *folded app<sub>i</sub>-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<sub>i</sub>-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**  $xcpt\text{-}app = Effect.xcpt\text{-}app\text{-}def$  [of -  $P$  -  $mxs$   $xt$ , folded  $xcpt\text{-}app\text{-}def$ ]

**defines**  $app\ i\ pc \equiv Effect.app\ i\ P\ mxs\ T_r\ pc\ mpc\ xt$

**notes**  $app = Effect.app\text{-}def$  [of -  $P$   $mxs\ T_r$  -  $mpc\ xt$ , folded  $app\text{-}def\ xcpt\text{-}app\text{-}def\ app_i\text{-}def\ eff\text{-}def$ ]

**lemma**  $length\text{-}cases2$ :

**assumes**  $\bigwedge LT. P\ ([],LT)$

**assumes**  $\bigwedge l\ ST\ LT. P\ (l\#\!ST,LT)$

**shows**  $P\ s$

$\langle proof \rangle$

**lemma**  $length\text{-}cases3$ :

**assumes**  $\bigwedge LT. P\ ([],LT)$

**assumes**  $\bigwedge l\ LT. P\ ([l],LT)$

**assumes**  $\bigwedge l\ ST\ LT. P\ (l\#\!ST,LT)$

**shows**  $P\ s$   $\langle proof \rangle$

**lemma**  $length\text{-}cases4$ :

**assumes**  $\bigwedge LT. P\ ([],LT)$

**assumes**  $\bigwedge l\ LT. P\ ([l],LT)$

**assumes**  $\bigwedge l\ l'\ LT. P\ ([l,l'],LT)$

**assumes**  $\bigwedge l\ l'\ ST\ LT. P\ (l\#\!l'\#\!ST,LT)$

**shows**  $P\ s$   $\langle proof \rangle$

simp rules for  $app$

**lemma**  $appNone[simp]$ :  $app\ i\ P\ mxs\ T_r\ pc\ mpc\ et\ None = True$

$\langle proof \rangle$

**lemma**  $appLoad[simp]$ :

$app_i\ (Load\ idx, P, T_r, mxs, pc, s) = (\exists ST\ LT. s = (ST,LT) \wedge idx < length\ LT \wedge LT!idx \neq Err \wedge length\ ST < mxs)$

$\langle proof \rangle$

**lemma**  $appStore[simp]$ :

$app_i\ (Store\ idx, P, pc, mxs, T_r, s) = (\exists ts\ ST\ LT. s = (ts\#\!ST,LT) \wedge idx < length\ LT)$

$\langle proof \rangle$

**lemma**  $appPush[simp]$ :

$app_i\ (Push\ v, P, pc, mxs, T_r, s) =$

$(\exists ST\ LT. s = (ST,LT) \wedge length\ ST < mxs \wedge typeof\ v \neq None)$

$\langle proof \rangle$

**lemma**  $appGetField[simp]$ :

$app_i\ (Getfield\ F\ C, P, pc, mxs, T_r, s) =$

$(\exists oT\ vT\ ST\ LT. s = (oT\#\!ST, LT) \wedge$

$P \vdash C\ sees\ F:vT\ in\ C \wedge P \vdash oT \leq (Class\ C))$

$\langle proof \rangle$

**lemma**  $appPutField[simp]$ :

$app_i\ (Putfield\ F\ C, P, pc, mxs, T_r, s) =$

$(\exists vT\ vT'\ oT\ ST\ LT. s = (vT\#\!oT\#\!ST, LT) \wedge$

$P \vdash C\ sees\ F:vT'\ in\ C \wedge P \vdash oT \leq (Class\ C) \wedge P \vdash vT \leq vT')$

*<proof>*

**lemma** *appNew[simp]*:

$app_i (New\ C,P,pc,mxs,T_r,s) =$   
 $(\exists\ ST\ LT. s=(ST,LT) \wedge is-class\ P\ C \wedge length\ ST < mxs)$   
*<proof>*

**lemma** *appCheckcast[simp]*:

$app_i (Checkcast\ C,P,pc,mxs,T_r,s) =$   
 $(\exists\ T\ ST\ LT. s = (T\#\!ST,LT) \wedge is-class\ P\ C \wedge is-refT\ T)$   
*<proof>*

**lemma** *app\_iPop[simp]*:

$app_i (Pop,P,pc,mxs,T_r,s) = (\exists\ ts\ ST\ LT. s = (ts\#\!ST,LT))$   
*<proof>*

**lemma** *appIAdd[simp]*:

$app_i (IAdd,P,pc,mxs,T_r,s) = (\exists\ ST\ LT. s = (Integer\#\!Integer\#\!ST,LT))$ *<proof>*

**lemma** *appIfFalse [simp]*:

$app_i (IfFalse\ b,P,pc,mxs,T_r,s) =$   
 $(\exists\ ST\ LT. s = (Boolean\#\!ST,LT) \wedge 0 \leq int\ pc + b)$ *<proof>*

**lemma** *appCmpEq[simp]*:

$app_i (CmpEq,P,pc,mxs,T_r,s) =$   
 $(\exists\ T_1\ T_2\ ST\ LT. s = (T_1\#\!T_2\#\!ST,LT) \wedge (\neg is-refT\ T_1 \wedge T_2 = T_1 \vee is-refT\ T_1 \wedge is-refT\ T_2))$   
*<proof>*

**lemma** *appReturn[simp]*:

$app_i (Return,P,pc,mxs,T_r,s) = (\exists\ T\ ST\ LT. s = (T\#\!ST,LT) \wedge P \vdash T \leq T_r)$   
*<proof>*

**lemma** *appThrow[simp]*:

$app_i (Throw,P,pc,mxs,T_r,s) = (\exists\ T\ ST\ LT. s=(T\#\!ST,LT) \wedge is-refT\ T)$   
*<proof>*

**lemma** *effNone*:

$(pc', s') \in set\ (eff\ i\ P\ pc\ et\ None) \implies s' = None$   
*<proof>*

some helpers to make the specification directly executable:

**lemma** *relevant-entries-append [simp]*:

$relevant-entries\ P\ i\ pc\ (xt\ @\ xt') = relevant-entries\ P\ i\ pc\ xt\ @\ relevant-entries\ P\ i\ pc\ xt'$   
*<proof>*

**lemma** *xcpt-app-append [iff]*:

$xcpt-app\ i\ P\ pc\ mxs\ (xt\ @\ xt')\ \tau = (xcpt-app\ i\ P\ pc\ mxs\ xt\ \tau \wedge xcpt-app\ i\ P\ pc\ mxs\ xt'\ \tau)$   
*<proof>*

**lemma** *xcpt-eff-append [simp]*:

$xcpt-eff\ i\ P\ pc\ \tau\ (xt\ @\ xt') = xcpt-eff\ i\ P\ pc\ \tau\ xt\ @\ xcpt-eff\ i\ P\ pc\ \tau\ xt'$   
*<proof>*

**lemma** *app-append [simp]*:

$app\ i\ P\ pc\ T\ mxs\ mpc\ (xt\ @\ xt')\ \tau = (app\ i\ P\ pc\ T\ mxs\ mpc\ xt\ \tau \wedge app\ i\ P\ pc\ T\ mxs\ mpc\ xt'\ \tau)$



*<proof>*

**end**

## 4.16 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  $app_i (i, P, mxs, mpc, rT, \tau') \implies app_i (i, P, mxs, mpc, rT, \tau) \langle proof \rangle$

lemma *succs-mono*:

assumes *wf*: *wf-prog* *p P* and *app<sub>i</sub>*:  $app_i (i, P, mxs, mpc, rT, \tau')$

shows  $P \vdash \tau \leq_i \tau' \implies set (succs\ i\ \tau\ pc) \subseteq set (succs\ i\ \tau'\ pc) \langle proof \rangle$

lemma *app-mono*:

assumes *wf*: *wf-prog* *p P*

assumes *less'*:  $P \vdash \tau \leq' \tau'$

shows  $app\ i\ P\ m\ rT\ pc\ mpc\ xt\ \tau' \implies app\ i\ P\ m\ rT\ pc\ mpc\ xt\ \tau \langle proof \rangle$

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  $P \vdash eff_i (i, P, \tau) \leq_i eff_i (i, P, \tau') \langle proof \rangle$

end

## 4.17 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:  
 $check\text{-}types :: 'm\ prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i' err\ list \Rightarrow bool$

**where**

$check\text{-}types\ P\ m\ x\ s\ m\ x\ l\ \tau\ s \equiv set\ \tau\ s \subseteq states\ P\ m\ x\ s\ m\ x\ l$

— An instruction is welltyped if it is applicable and its effect  
 — is compatible with the type at all successor instructions:

**definition**

$wt\text{-}instr :: ['m\ prog, ty, nat, pc, ex\text{-}table, instr, pc, ty_m] \Rightarrow bool$   
 $(\neg, \neg, \neg, \neg \vdash \neg, \neg :: - [60, 0, 0, 0, 0, 0, 0, 61] 60)$

**where**

$P, T, m\ x\ s, m\ p\ c, x\ t \vdash i, pc :: \tau\ s \equiv$   
 $app\ i\ P\ m\ x\ s\ T\ pc\ m\ p\ c\ x\ t\ (\tau\ s!pc) \wedge$   
 $(\forall (pc', \tau') \in set\ (eff\ i\ P\ pc\ x\ t\ (\tau\ s!pc)).\ P \vdash \tau' \leq' \tau\ s!pc)$

— The type at  $pc=0$  conforms to the method calling convention:

**definition**  $wt\text{-}start :: ['m\ prog, cname, ty\ list, nat, ty_m] \Rightarrow bool$

**where**

$wt\text{-}start\ P\ C\ T\ s\ m\ x\ l_0\ \tau\ s \equiv$   
 $P \vdash Some\ ([], OK\ (Class\ C)\#map\ OK\ T\ s@replicate\ m\ x\ l_0\ Err) \leq' \tau\ s!0$

— A method is welltyped if the body is not empty,  
 — if the method type covers all instructions and mentions  
 — declared classes only, if the method calling convention is respected, and  
 — if all instructions are welltyped.

**definition**  $wt\text{-}method :: ['m\ prog, cname, ty\ list, ty, nat, nat, instr\ list,$   
 $ex\text{-}table, ty_m] \Rightarrow bool$

**where**

$wt\text{-}method\ P\ C\ T\ s\ T_r\ m\ x\ s\ m\ x\ l_0\ is\ x\ t\ \tau\ s \equiv$   
 $0 < size\ is \wedge size\ \tau\ s = size\ is \wedge$   
 $check\text{-}types\ P\ m\ x\ s\ (1+size\ T\ s+m\ x\ l_0)\ (map\ OK\ \tau\ s) \wedge$   
 $wt\text{-}start\ P\ C\ T\ s\ m\ x\ l_0\ \tau\ s \wedge$   
 $(\forall pc < size\ is.\ P, T_r, m\ x\ s, size\ is, x\ t \vdash is!pc, pc :: \tau\ s)$

— A program is welltyped if it is wellformed and all methods are welltyped

**definition**  $wf\text{-}jvm\text{-}prog\text{-}phi :: ty_P \Rightarrow jvm\text{-}prog \Rightarrow bool\ (wf'\text{-}jvm'\text{-}prog\text{-})$

**where**

$wf\text{-}jvm\text{-}prog_{\Phi} \equiv$   
 $wf\text{-}prog\ (\lambda P\ C\ (M, T\ s, T_r, (m\ x\ s, m\ x\ l_0, is, x\ t)).$   
 $wt\text{-}method\ P\ C\ T\ s\ T_r\ m\ x\ s\ m\ x\ l_0\ is\ x\ t\ (\Phi\ C\ M))$

**definition**  $wf\text{-}jvm\text{-}prog :: jvm\text{-}prog \Rightarrow bool$

**where**

$wf\text{-}jvm\text{-}prog\ P \equiv \exists \Phi.\ wf\text{-}jvm\text{-}prog_{\Phi}\ P$

**notation** (*input*)

*wf-jvm-prog-phi* (*wf'-jvm'-prog-* - [0,999] 1000)

**lemma** *wt-jvm-progD*:

*wf-jvm-prog<sub>Φ</sub>* *P*  $\implies \exists wt. wf-prog\ wt\ P\langle proof \rangle$

**lemma** *wt-jvm-prog-impl-wt-instr*:

$\llbracket wf-jvm-prog_{\Phi}\ P;$   
 $P \vdash C\ sees\ M:Ts \rightarrow T = (m\ x_s, m\ x_l_0, ins, xt)\ in\ C; pc < size\ ins \rrbracket$   
 $\implies P, T, m\ x_s, size\ ins, xt \vdash ins!pc, pc :: \Phi\ C\ M\langle proof \rangle$

**lemma** *wt-jvm-prog-impl-wt-start*:

$\llbracket wf-jvm-prog_{\Phi}\ P;$   
 $P \vdash C\ sees\ M:Ts \rightarrow T = (m\ x_s, m\ x_l_0, ins, xt)\ in\ C \rrbracket \implies$   
 $0 < size\ ins \wedge wt-start\ P\ C\ Ts\ m\ x_l_0\ (\Phi\ C\ M)\langle proof \rangle$

**end**

## 4.18 The Typing Framework for the JVM

**theory** *TF-JVM*

**imports** *../DFA/Typing-Framework-err EffectMono BVSpec*

**begin**

**definition** *exec* :: *jvm-prog*  $\Rightarrow$  *nat*  $\Rightarrow$  *ty*  $\Rightarrow$  *ex-table*  $\Rightarrow$  *instr list*  $\Rightarrow$  *ty<sub>i</sub>'* *err step-type*

**where**

*exec* *G mxs rT et bs*  $\equiv$

*err-step* (*size bs*) ( $\lambda pc. app (bs!pc) G mxs rT pc (size bs) et$ )  
 ( $\lambda pc. eff (bs!pc) G pc et$ )

**locale** *JVM-sl* =

**fixes** *P* :: *jvm-prog* **and** *mxs* **and** *mxl<sub>0</sub>*

**fixes** *Ts* :: *ty list* **and** *is* **and** *xt* **and** *T<sub>r</sub>*

**fixes** *mxl* **and** *A* **and** *r* **and** *f* **and** *app* **and** *eff* **and** *step*

**defines** [*simp*]: *mxl*  $\equiv 1 + size Ts + mxl_0$

**defines** [*simp*]: *A*  $\equiv states P mxs mxl$

**defines** [*simp*]: *r*  $\equiv JVM-SemiType.le P mxs mxl$

**defines** [*simp*]: *f*  $\equiv JVM-SemiType.sup P mxs mxl$

**defines** [*simp*]: *app*  $\equiv \lambda pc. Effect.app (is!pc) P mxs T_r pc (size is) xt$

**defines** [*simp*]: *eff*  $\equiv \lambda pc. Effect.eff (is!pc) P pc xt$

**defines** [*simp*]: *step*  $\equiv err-step (size is) app eff$

**locale** *start-context* = *JVM-sl* +

**fixes** *p* **and** *C*

**assumes** *wf*: *wf-prog p P*

**assumes** *C*: *is-class P C*

**assumes** *Ts*: *set Ts*  $\subseteq types P$

**fixes** *first* :: *ty<sub>i</sub>'* **and** *start*

**defines** [*simp*]:

*first*  $\equiv Some ([], OK (Class C) \# map OK Ts @ replicate mxl_0 Err)$

**defines** [*simp*]:

*start*  $\equiv OK first \# replicate (size is - 1) (OK None)$

### 4.18.1 Connecting JVM and Framework

**lemma** (**in** *JVM-sl*) *step-def-exec*: *step*  $\equiv exec P mxs T_r xt is$

*<proof>*

**lemma** *special-ex-swap-lemma* [*iff*]:

$(? X. (? n. X = A n \& P n) \& Q X) = (? n. Q(A n) \& P n)$

*<proof>*

**lemma** *ex-in-list* [*iff*]:

$(\exists n. ST \in list n A \wedge n \leq mxs) = (set ST \subseteq A \wedge size ST \leq mxs)$

*<proof>*

**lemma** *singleton-list*:

$(\exists n. [Class C] \in list n (types P) \wedge n \leq mxs) = (is-class P C \wedge 0 < mxs)$

*<proof>*

**lemma** *set-drop-subset*:

$set\ xs \subseteq A \implies set\ (drop\ n\ xs) \subseteq A$

*<proof>*

**lemma** *Suc-minus-minus-le*:

$n < mxs \implies Suc\ (n - (n - b)) \leq mxs$

*<proof>*

**lemma** *in-listE*:

$\llbracket xs \in list\ n\ A; \llbracket size\ xs = n; set\ xs \subseteq A \rrbracket \implies P \rrbracket \implies P$

*<proof>*

**declare** *is-relevant-entry-def* [*simp*]

**declare** *set-drop-subset* [*simp*]

**theorem** (*in start-context*) *exec-pres-type*:

*pres-type step (size is) A**<proof>*

**declare** *is-relevant-entry-def* [*simp del*]

**declare** *set-drop-subset* [*simp del*]

**lemma** *lesubstep-type-simple*:

$xs \sqsubseteq_{Product.le\ (op\ =)\ r} ys \implies set\ xs \{\sqsubseteq_r\} set\ ys$ *<proof>*

**declare** *is-relevant-entry-def* [*simp del*]

**lemma** *conjI2*:  $\llbracket A; A \implies B \rrbracket \implies A \wedge B$  *<proof>*

**lemma** (*in JVM-sl*) *eff-mono*:

$\llbracket wf\ prog\ p\ P; pc < length\ is; s \sqsubseteq_{sup\ state\ opt\ P} t; app\ pc\ t \rrbracket$

$\implies set\ (eff\ pc\ s) \{\sqsubseteq_{sup\ state\ opt\ P}\} set\ (eff\ pc\ t)$ *<proof>*

**lemma** (*in JVM-sl*) *bounded-step*: *bounded step (size is)**<proof>*

**theorem** (*in JVM-sl*) *step-mono*:

$wf\ prog\ wf\ mb\ P \implies mono\ r\ step\ (size\ is)\ A$ *<proof>*

**lemma** (*in start-context*) *first-in-A* [*iff*]: *OK first*  $\in A$

*<proof>*

**lemma** (*in JVM-sl*) *wt-method-def2*:

*wt-method*  $P\ C'\ Ts\ T_r\ mxs\ mxl_0\ is\ xt\ \tau s =$

$(is \neq [] \wedge$

$size\ \tau s = size\ is \wedge$

$OK\ 'set\ \tau s \subseteq states\ P\ mxs\ mxl \wedge$

$wt\ start\ P\ C'\ Ts\ mxl_0\ \tau s \wedge$

$wt\ app\ eff\ (sup\ state\ opt\ P)\ app\ eff\ \tau s)$ *<proof>*

**end**

## 4.19 Kildall for the JVM

**theory** BVExec

**imports** ../DFA/Abstract-BV TF-JVM

**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 T<sub>r</sub> is xt*  $\equiv$   
*kildall* (*JVM-SemiType.le* *P mxs mxl*) (*JVM-SemiType.sup* *P mxs mxl*)  
 (*exec* *P mxs T<sub>r</sub> xt is*)

**definition** *wt-kildall* :: *jvm-prog*  $\Rightarrow$  *cname*  $\Rightarrow$  *ty list*  $\Rightarrow$  *ty*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat*  $\Rightarrow$   
*instr list*  $\Rightarrow$  *ex-table*  $\Rightarrow$  *bool*

**where**

*wt-kildall* *P C' Ts T<sub>r</sub> mxs mxl<sub>0</sub> is xt*  $\equiv$   
 0 < *size is*  $\wedge$   
 (let *first* = *Some* ([],[*OK* (*Class* *C'*)]@(*map* *OK* *Ts*)@(replicate *mxl<sub>0</sub>* *Err*));  
   *start* = *OK first* # (replicate (*size is* - 1) (*OK None*));  
   *result* = *kiljvm* *P mxs* (1+*size Ts*+*mxl<sub>0</sub>*) *T<sub>r</sub> is xt start*  
 in  $\forall n < \text{size } is. \text{result!}n \neq \text{Err}$ )

**definition** *wf-jvm-prog<sub>k</sub>* :: *jvm-prog*  $\Rightarrow$  *bool*

**where**

*wf-jvm-prog<sub>k</sub>* *P*  $\equiv$   
*wf-prog* ( $\lambda P C' (M, Ts, T_r, (mxs, mxl_0, is, xt)). \text{wt-kildall } P C' Ts T_r mxs mxl_0 is xt$ ) *P*

**theorem** (**in** *start-context*) *is-bcv-kiljvm*:

*is-bcv* *r Err step* (*size is*) *A* (*kiljvm* *P mxs mxl T<sub>r</sub> is xt*)  $\langle \text{proof} \rangle$

**lemma** *subset-replicate* [*intro?*]: *set* (replicate *n x*)  $\subseteq$  {*x*}

$\langle \text{proof} \rangle$

**lemma** *in-set-replicate*:

**assumes** *x*  $\in$  *set* (replicate *n y*)

**shows** *x* = *y*  $\langle \text{proof} \rangle$

**lemma** (**in** *start-context*) *start-in-A* [*intro?*]:

0 < *size is*  $\implies$  *start*  $\in$  *list* (*size is*) *A*

$\langle \text{proof} \rangle$

**theorem** (**in** *start-context*) *wt-kil-correct*:

**assumes** *wtk*: *wt-kildall* *P C Ts T<sub>r</sub> mxs mxl<sub>0</sub> is xt*

**shows**  $\exists \tau s. \text{wt-method } P C Ts T_r mxs mxl_0 is xt \tau s$   $\langle \text{proof} \rangle$

**theorem** (**in** *start-context*) *wt-kil-complete*:

**assumes** *wtm*: *wt-method* *P C Ts T<sub>r</sub> mxs mxl<sub>0</sub> is xt*  $\tau s$

**shows** *wt-kildall* *P C Ts T<sub>r</sub> mxs mxl<sub>0</sub> is xt*  $\langle \text{proof} \rangle$

**theorem** *jvm-kildall-correct*:

*wf-jvm-prog<sub>k</sub>* *P* = *wf-jvm-prog* *P*  $\langle \text{proof} \rangle$

**end**

## 4.20 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*  $\wedge$   
 $(\forall i < n. \text{cert}!i \neq \text{Err}) \wedge \text{cert}!n = \text{OK None}$

**definition** *lbvjvm* :: *jvm-prog*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *ty*  $\Rightarrow$  *ex-table*  $\Rightarrow$

*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<sub>r</sub> 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<sub>r</sub> 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<sub>r</sub> mxs mxl<sub>0</sub> et cert ins*  $\equiv$

*check-cert* *P mxs (1+size Ts+mxl<sub>0</sub>) (size ins) cert*  $\wedge$

*0 < size ins*  $\wedge$

$(\text{let } \text{start} = \text{Some } ([], (\text{OK } (\text{Class } C)) \# ((\text{map } \text{OK } \text{Ts})) @ (\text{replicate } \text{mxl}_0 \text{ Err}));$

$\text{result} = \text{lbvjvm } P \text{ mxs } (1 + \text{size } \text{Ts} + \text{mxl}_0) \text{ T}_r \text{ et cert ins } (\text{OK } \text{start})$

$\text{in result} \neq \text{Err})$

**definition** *wt-jvm-prog-lbv* :: *jvm-prog*  $\Rightarrow$  *prog-cert*  $\Rightarrow$  *bool*

**where**

*wt-jvm-prog-lbv* *P cert*  $\equiv$

*wf-prog*  $(\lambda P C (mn, Ts, T_r, (mxs, mxl_0, b, et)). \text{wt-lbv } P C Ts T_r mxs mxl_0 \text{ et } (\text{cert } C mn) b) P$

**definition** *mk-cert* :: *jvm-prog*  $\Rightarrow$  *nat*  $\Rightarrow$  *ty*  $\Rightarrow$  *ex-table*  $\Rightarrow$  *instr list*

$\Rightarrow$  *ty<sub>m</sub>*  $\Rightarrow$  *ty<sub>i</sub>' err list*

**where**

*mk-cert* *P mxs T<sub>r</sub> et bs phi*  $\equiv$  *make-cert*  $(\text{exec } P \text{ mxs } T_r \text{ et bs}) (\text{map } \text{OK } \text{phi}) (\text{OK None})$

**definition** *prg-cert* :: *jvm-prog*  $\Rightarrow$  *ty<sub>P</sub>*  $\Rightarrow$  *prog-cert*

**where**

*prg-cert* *P phi C mn*  $\equiv$  *let*  $(C, Ts, T_r, (mxs, mxl_0, ins, et)) = \text{method } P C mn$

*in mk-cert* *P mxs T<sub>r</sub> et ins (phi C mn)*

**lemma** *check-certD* [*intro?*]:

*check-cert* *P mxs mxl n cert*  $\implies$  *cert-ok cert n Err (OK None) (states P mxs mxl)*

*(proof)*

**lemma** (*in start-context*) *wt-lbv-wt-step*:

**assumes** *lbv*: *wt-lbv* *P C Ts T<sub>r</sub> mxs mxl<sub>0</sub> xt cert* *is*

**shows**  $\exists \tau s \in \text{list } (\text{size } \text{is}) \text{ A. wt-step } r \text{ Err step } \tau s \wedge \text{OK first } \sqsubseteq_r \tau s!0$  *(proof)*



**lemma** (in *start-context*) *wt-lbv-wt-method*:  
**assumes** *lbv*: *wt-lbv*  $P$   $C$   $T_s$   $T_r$  *mxs*  $mxl_0$  *xt* *cert* *is*  
**shows**  $\exists \tau s$ . *wt-method*  $P$   $C$   $T_s$   $T_r$  *mxs*  $mxl_0$  *is* *xt*  $\tau s$ ⟨*proof*⟩

**lemma** (in *start-context*) *wt-method-wt-lbv*:  
**assumes** *wt*: *wt-method*  $P$   $C$   $T_s$   $T_r$  *mxs*  $mxl_0$  *is* *xt*  $\tau s$   
**defines** [*simp*]: *cert*  $\equiv$  *mk-cert*  $P$  *mxs*  $T_r$  *xt* *is*  $\tau s$   
**shows** *wt-lbv*  $P$   $C$   $T_s$   $T_r$  *mxs*  $mxl_0$  *xt* *cert* *is*⟨*proof*⟩

**theorem** *jvm-lbv-correct*:  
*wt-jvm-prog-lbv*  $P$  *Cert*  $\implies$  *wf-jvm-prog*  $P$ ⟨*proof*⟩

**theorem** *jvm-lbv-complete*:  
**assumes** *wt*: *wf-jvm-prog* $_{\Phi}$   $P$   
**shows** *wt-jvm-prog-lbv*  $P$  (*prg-cert*  $P$   $\Phi$ )⟨*proof*⟩  
**end**

## 4.21 BV Type Safety Invariant

**theory** *BVConform*

**imports** *BVSpec ../JVM/JVMExec ../Common/Conform*

**begin**

**definition** *confT* :: 'c prog ⇒ heap ⇒ val ⇒ ty err ⇒ bool  
 (·, · | - · :<=T - [51,51,51,51] 50)

**where**

$P, h \mid - v :<=T E \equiv \text{case } E \text{ of } \text{Err} \Rightarrow \text{True} \mid \text{OK } T \Rightarrow P, h \vdash v :< T$

**notation** (*xsymbols*)

*confT* (·, · | - · :<=T - [51,51,51,51] 50)

**abbreviation**

*confTs* :: 'c prog ⇒ heap ⇒ val list ⇒ ty<sub>l</sub> ⇒ bool  
 (·, · | - · [:<=T] - [51,51,51,51] 50) **where**  
 $P, h \mid - vs [:<=T] Ts \equiv \text{list-all2 } (\text{confT } P \ h) \ vs \ Ts$

**notation** (*xsymbols*)

*confTs* (·, · | - · [:<=T] - [51,51,51,51] 50)

**definition** *conf-f* :: jvm-prog ⇒ heap ⇒ ty<sub>i</sub> ⇒ bytecode ⇒ frame ⇒ bool

**where**

$\text{conf-f } P \ h \equiv \lambda(ST, LT) \text{ is } (stk, loc, C, M, pc).$   
 $P, h \vdash stk [:<] ST \wedge P, h \vdash loc [:<_{\top}] LT \wedge pc < \text{size is}$

**lemma** *conf-f-def2*:

$\text{conf-f } P \ h \ (ST, LT) \text{ is } (stk, loc, C, M, pc) \equiv$   
 $P, h \vdash stk [:<] ST \wedge P, h \vdash loc [:<_{\top}] LT \wedge pc < \text{size is}$   
 ⟨*proof*⟩

**primrec** *conf-fs* :: [jvm-prog, heap, ty<sub>P</sub>, mname, nat, ty, frame list] ⇒ bool

**where**

$\text{conf-fs } P \ h \ \Phi \ M_0 \ n_0 \ T_0 \ [] = \text{True}$   
 $\mid \text{conf-fs } P \ h \ \Phi \ M_0 \ n_0 \ T_0 \ (f\#\text{frs}) =$   
 (let (stk, loc, C, M, pc) = f in  
 (∃ ST LT Ts T mxs mxl<sub>0</sub> is xt.  
 Φ C M ! pc = Some (ST, LT) ∧  
 (P ⊢ C sees M:Ts → T = (mxs, mxl<sub>0</sub>, is, xt) in C) ∧  
 (∃ D Ts' T' m D'.  
 is!pc = (Invoke M<sub>0</sub> n<sub>0</sub>) ∧ ST!n<sub>0</sub> = Class D ∧  
 P ⊢ D sees M<sub>0</sub>:Ts' → T' = m in D' ∧ P ⊢ T<sub>0</sub> ≤ T') ∧  
 conf-f P h (ST, LT) is f ∧ conf-fs P h Φ M (size Ts) T frs))

**definition** *correct-state* :: [jvm-prog, ty<sub>P</sub>, jvm-state] ⇒ bool

(·, · | - · [ok] [61,0,0] 61)

**where**

$\text{correct-state } P \ \Phi \equiv \lambda(xp, h, frs).$   
 case xp of  
 None ⇒ (case frs of

$$\begin{aligned}
& [] \Rightarrow \text{True} \\
& | (f \# fs) \Rightarrow P \vdash h \surd \wedge \\
& \quad (\text{let } (stk, loc, C, M, pc) = f \\
& \quad \quad \text{in } \exists Ts \ T \ mxs \ mxl_0 \ \text{is } xt \ \tau. \\
& \quad \quad (P \vdash C \ \text{sees } M : Ts \rightarrow T = (mxs, mxl_0, is, xt) \ \text{in } C) \wedge \\
& \quad \quad \Phi \ C \ M \ ! \ pc = \text{Some } \tau \wedge \\
& \quad \quad \text{conf-f } P \ h \ \tau \ \text{is } f \wedge \text{conf-fs } P \ h \ \Phi \ M \ (\text{size } Ts) \ T \ fs)) \\
& | \text{Some } x \Rightarrow \text{frs} = []
\end{aligned}$$

**notation** (*xsymbols*)

*correct-state*  $(\neg, \vdash \neg \surd [61, 0, 0] \ 61)$

#### 4.21.1 Values and $\top$

**lemma** *confT-Err* [*iff*]:  $P, h \vdash x : \leq_{\top} \text{Err}$   
*(proof)*

**lemma** *confT-OK* [*iff*]:  $P, h \vdash x : \leq_{\top} \text{OK } T = (P, h \vdash x : \leq T)$   
*(proof)*

**lemma** *confT-cases*:

$P, h \vdash x : \leq_{\top} X = (X = \text{Err} \vee (\exists T. X = \text{OK } T \wedge P, h \vdash x : \leq T))$   
*(proof)*

**lemma** *confT-hest* [*intro?*, *trans*]:

$\llbracket P, h \vdash x : \leq_{\top} T; h \sqsubseteq h' \rrbracket \Longrightarrow P, h' \vdash x : \leq_{\top} T$   
*(proof)*

**lemma** *confT-widen* [*intro?*, *trans*]:

$\llbracket P, h \vdash x : \leq_{\top} T; P \vdash T \leq_{\top} T' \rrbracket \Longrightarrow P, h \vdash x : \leq_{\top} T'$   
*(proof)*

#### 4.21.2 Stack and Registers

**lemmas** *confTs-Cons1* [*iff*] = *list-all2-Cons1* [*of confT P h*] **for**  $P \ h$

**lemma** *confTs-confT-sup*:

$\llbracket P, h \vdash \text{loc } [:\leq_{\top}] \ LT; n < \text{size } LT; LT!n = \text{OK } T; P \vdash T \leq T' \rrbracket$   
 $\Longrightarrow P, h \vdash (\text{loc}!n) : \leq T'$  *(proof)*

**lemma** *confTs-hest* [*intro?*]:

$P, h \vdash \text{loc } [:\leq_{\top}] \ LT \Longrightarrow h \sqsubseteq h' \Longrightarrow P, h' \vdash \text{loc } [:\leq_{\top}] \ LT$   
*(proof)*

**lemma** *confTs-widen* [*intro?*, *trans*]:

$P, h \vdash \text{loc } [:\leq_{\top}] \ LT \Longrightarrow P \vdash LT [:\leq_{\top}] \ LT' \Longrightarrow P, h \vdash \text{loc } [:\leq_{\top}] \ LT'$   
*(proof)*

**lemma** *confTs-map* [*iff*]:

$\bigwedge \text{vs}. (P, h \vdash \text{vs } [:\leq_{\top}] \ \text{map } \text{OK } Ts) = (P, h \vdash \text{vs } [:\leq] \ Ts)$   
*(proof)*

**lemma** *reg-widen-Err* [*iff*]:

$\bigwedge \text{LT}. (P \vdash \text{replicate } n \ \text{Err} [:\leq_{\top}] \ \text{LT}) = (\text{LT} = \text{replicate } n \ \text{Err})$   
*(proof)*

**lemma** *confTs-Err* [iff]:

$P, h \vdash \text{replicate } n \ v \ [;\leq_{\top}] \ \text{replicate } n \ \text{Err}$   
 ⟨proof⟩

### 4.21.3 correct-frames

**lemmas** [*simp del*] = *fun-upd-apply*

**lemma** *conf-fs-hext*:

$\bigwedge M \ n \ T_r.$   
 $\llbracket \text{conf-fs } P \ h \ \Phi \ M \ n \ T_r \ \text{frs}; \ h \trianglelefteq h' \rrbracket \implies \text{conf-fs } P \ h' \ \Phi \ M \ n \ T_r \ \text{frs}$  ⟨proof⟩  
**end**

## 4.22 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.

### 4.22.1 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
```

### 4.22.2 Exception Handling

For the *Invoke* instruction the BV has checked all handlers that guard the current *pc*.

**lemma** *Invoke-handlers*:

```
match-ex-table P C pc xt = Some (pc',d')  $\implies$ 
 $\exists (f,t,D,h,d) \in \text{set } (\text{relevant-entries } P (\text{Invoke } n M) pc xt).$ 
 $P \vdash C \preceq^* D \wedge pc \in \{f..<t\} \wedge pc' = h \wedge d' = d$ 
<proof>
```

We can prove separately that the recursive search for exception handlers (*find-handler*) in the frame stack results in a conforming state (if there was no matching exception handler in the current frame). We require that the exception is a valid heap address, and that the state before the exception occurred conforms.

**term** *find-handler*

**lemma** *uncaught-xcpt-correct*:

```
assumes wt: wf-jvm-prog $\Phi$  P
assumes h: h xcp = Some obj
shows  $\bigwedge f. P, \Phi \vdash (\text{None}, h, f \# \text{frs}) \checkmark \implies P, \Phi \vdash (\text{find-handler } P xcp h \text{ frs}) \checkmark$ 
(is  $\bigwedge f. ?correct (\text{None}, h, f \# \text{frs}) \implies ?correct (?find \text{ frs})$ ) <proof>
```

The requirement of lemma *uncaught-xcpt-correct* (that the exception is a valid reference on the heap) is always met for welltyped instructions and conformant states:

**lemma** *exec-instr-xcpt-h*:

```
 $\llbracket \text{fst } (\text{exec-instr } (\text{ins!pc}) P h \text{ stk vars } Cl M pc \text{ frs}) = \text{Some } xcp;$ 
 $P, T, \text{mxs}, \text{size } \text{ins}, xt \vdash \text{ins!pc}, pc :: \Phi C M;$ 
 $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, pc) \# \text{frs}) \checkmark \rrbracket$ 
 $\implies \exists \text{obj}. h xcp = \text{Some } \text{obj}$ 
(is  $\llbracket ?xcpt; ?wt; ?correct \rrbracket \implies ?thesis$ ) <proof>
```

**lemma** *conf-sys-xcpt*:

```
 $\llbracket \text{preallocated } h; C \in \text{sys-xcpts} \rrbracket \implies P, h \vdash \text{Addr } (\text{addr-of-sys-xcpt } C) : \leq \text{Class } C$ 
<proof>
```

**lemma** *match-ex-table-SomeD*:

```
match-ex-table P C pc xt = Some (pc',d')  $\implies$ 
 $\exists (f,t,D,h,d) \in \text{set } xt. \text{matches-ex-entry } P C pc (f,t,D,h,d) \wedge h = pc' \wedge d = d'$ 
<proof>
```

Finally we can state that, whenever an exception occurs, the next state always conforms:

**lemma** *xcpt-correct*:  
**fixes**  $\sigma' :: \text{jvm-state}$   
**assumes** *wtp*:  $\text{wf-jvm-prog}_{\Phi} P$   
**assumes** *meth*:  $P \vdash C \text{ sees } M:Ts \rightarrow T=(\text{mxs}, \text{mxl}_0, \text{ins}, \text{xt}) \text{ in } C$   
**assumes** *wt*:  $P, T, \text{mxs}, \text{size } \text{ins}, \text{xt} \vdash \text{ins!pc}, \text{pc} :: \Phi C M$   
**assumes** *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** *s'*:  $\text{Some } \sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs})$   
**assumes** *correct*:  $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark$   
**shows**  $P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle$

### 4.22.3 Single Instructions

In this section we prove for each single (welltyped) instruction that the state after execution of the instruction still conforms. Since we have already handled exceptions above, we can now assume that no exception occurs in this step.

**declare** *defs1* [*simp*]

**lemma** *Invoke-correct*:  
**fixes**  $\sigma' :: \text{jvm-state}$   
**assumes** *wtprog*:  $\text{wf-jvm-prog}_{\Phi} P$   
**assumes** *meth-C*:  $P \vdash C \text{ sees } M:Ts \rightarrow T=(\text{mxs}, \text{mxl}_0, \text{ins}, \text{xt}) \text{ in } C$   
**assumes** *ins*:  $\text{ins} ! \text{pc} = \text{Invoke } M' n$   
**assumes** *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** *approx*:  $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark$   
**assumes** *no-xcp*:  $\text{fst} (\text{exec-instr} (\text{ins!pc}) P h \text{ stk } \text{loc } C M \text{ pc } \text{ frs}) = \text{None}$   
**shows**  $P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle$   
**declare** *list-all2-Cons2* [*iff*]

**lemma** *Return-correct*:  
**fixes**  $\sigma' :: \text{jvm-state}$   
**assumes** *wtprog*:  $\text{wf-jvm-prog}_{\Phi} P$   
**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*:  $\text{ins} ! \text{pc} = \text{Return}$   
**assumes** *wt*:  $P, T, \text{mxs}, \text{size } \text{ins}, \text{xt} \vdash \text{ins!pc}, \text{pc} :: \Phi C M$   
**assumes**  $\sigma'$ :  $\text{Some } \sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs})$   
**assumes** *correct*:  $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark$   
  
**shows**  $P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle$   
**declare** *sup-state-opt-any-Some* [*iff*]  
**declare** *not-Err-eq* [*iff*]

**lemma** *Load-correct*:  
 $\llbracket \text{wf-prog } \text{wt } P;$   
 $P \vdash C \text{ sees } M:Ts \rightarrow T=(\text{mxs}, \text{mxl}_0, \text{ins}, \text{xt}) \text{ in } C;$   
 $\text{ins!pc} = \text{Load } \text{idx};$   
 $P, T, \text{mxs}, \text{size } \text{ins}, \text{xt} \vdash \text{ins!pc}, \text{pc} :: \Phi C M;$   
 $\text{Some } \sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs});$   
 $P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, \text{pc}) \# \text{frs}) \checkmark \rrbracket$   
 $\implies P, \Phi \vdash \sigma' \checkmark$   
 $\langle \text{proof} \rangle$

**declare**  $[[\text{simproc del: list-to-set-comprehension}]]$

**lemma** *Store-correct:*

$[[ \text{wf-prog wt } P;$   
 $P \vdash C \text{ sees } M:Ts \rightarrow T=(m\text{xs},m\text{x}l_0,ins,xt) \text{ in } C;$   
 $ins!pc = \text{Store } id\text{x};$   
 $P, T, m\text{xs}, size \text{ ins}, xt \vdash ins!pc, pc :: \Phi \ C \ M;$   
 $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc) \# frs);$   
 $P, \Phi \vdash (\text{None}, h, (stk, loc, C, M, pc) \# frs) \checkmark ]]$   
 $\Rightarrow P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle$

**lemma** *Push-correct:*

$[[ \text{wf-prog wt } P;$   
 $P \vdash C \text{ sees } M:Ts \rightarrow T=(m\text{xs},m\text{x}l_0,ins,xt) \text{ in } C;$   
 $ins!pc = \text{Push } v;$   
 $P, T, m\text{xs}, size \text{ ins}, xt \vdash ins!pc, pc :: \Phi \ C \ M;$   
 $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc) \# frs);$   
 $P, \Phi \vdash (\text{None}, h, (stk, loc, C, M, pc) \# frs) \checkmark ]]$   
 $\Rightarrow P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle$

**lemma** *Cast-conf2:*

$[[ \text{wf-prog ok } P; P, h \vdash v : \leq T; is\text{-ref}T \ T; \text{cast-ok } P \ C \ h \ v;$   
 $P \vdash \text{Class } C \leq T'; is\text{-class } P \ C ]]$   
 $\Rightarrow P, h \vdash v : \leq T' \langle \text{proof} \rangle$

**lemma** *Checkcast-correct:*

$[[ \text{wf-jvm-prog}_{\Phi} \ P;$   
 $P \vdash C \text{ sees } M:Ts \rightarrow T=(m\text{xs},m\text{x}l_0,ins,xt) \text{ in } C;$   
 $ins!pc = \text{Checkcast } D;$   
 $P, T, m\text{xs}, size \text{ ins}, xt \vdash ins!pc, pc :: \Phi \ C \ M;$   
 $\text{Some } \sigma' = \text{exec } (P, \text{None}, h, (stk, loc, C, M, pc) \# frs) ;$   
 $P, \Phi \vdash (\text{None}, h, (stk, loc, C, M, pc) \# frs) \checkmark;$   
 $fst (\text{exec-instr } (ins!pc) \ P \ h \ stk \ loc \ C \ M \ pc \ frs) = \text{None} ]]$   
 $\Rightarrow P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle$

**declare** *split-paired-All*  $[\text{simp del}]$

**lemmas** *widens-Cons*  $[\text{iff}] = \text{list-all2-Cons1}$   $[\text{of widen } P]$  **for**  $P$

**lemma** *Getfield-correct:*

**fixes**  $\sigma' :: \text{jvm-state}$   
**assumes**  $\text{wf: wf-prog wt } P$   
**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**  $\text{wt: } P, T, m\text{xs}, size \text{ ins}, xt \vdash ins!pc, pc :: \Phi \ C \ M$   
**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**  $P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle$

**lemma** *Putfield-correct:*

**fixes**  $\sigma' :: \text{jvm-state}$   
**assumes**  $\text{wf: wf-prog wt } P$   
**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**  $wt: P, T, mxs, size\ ins, xt \vdash ins!pc, pc :: \Phi\ C\ M$   
**assumes**  $s': Some\ \sigma' = exec\ (P, None, h, (stk, loc, C, M, pc) \# frs)$   
**assumes**  $cf: P, \Phi \vdash (None, h, (stk, loc, C, M, pc) \# frs) \surd$   
**assumes**  $xc: fst\ (exec\ instr\ (ins!pc)\ P\ h\ stk\ loc\ C\ M\ pc\ frs) = None$   
**shows**  $P, \Phi \vdash \sigma' \surd \langle proof \rangle$

**lemma** *has-fields-b-fields*:

$P \vdash C\ has\ fields\ FDTs \implies fields\ P\ C = FDTs \langle proof \rangle$

**lemma** *oconf-blank* [intro, simp]:

$\llbracket is\ class\ P\ C; wf\ prog\ wt\ P \rrbracket \implies P, h \vdash blank\ P\ C \surd \langle proof \rangle$

**lemma** *obj-ty-blank* [iff]:  $obj\ ty\ (blank\ P\ C) = Class\ C$

$\langle proof \rangle$

**lemma** *New-correct*:

**fixes**  $\sigma' :: jvm\ state$

**assumes**  $wf: wf\ prog\ wt\ P$

**assumes**  $meth: P \vdash C\ sees\ M: Ts \rightarrow T = (mxs, mxl_0, ins, xt)\ in\ C$

**assumes**  $ins: ins!pc = New\ X$

**assumes**  $wt: P, T, mxs, size\ ins, xt \vdash ins!pc, pc :: \Phi\ C\ M$

**assumes**  $exec: Some\ \sigma' = exec\ (P, None, h, (stk, loc, C, M, pc) \# frs)$

**assumes**  $conf: P, \Phi \vdash (None, h, (stk, loc, C, M, pc) \# frs) \surd$

**assumes**  $no-x: fst\ (exec\ instr\ (ins!pc)\ P\ h\ stk\ loc\ C\ M\ pc\ frs) = None$

**shows**  $P, \Phi \vdash \sigma' \surd \langle proof \rangle$

**lemma** *Goto-correct*:

$\llbracket wf\ prog\ wt\ P;$

$P \vdash C\ sees\ M: Ts \rightarrow T = (mxs, mxl_0, ins, xt)\ in\ C;$

$ins!pc = Goto\ branch;$

$P, T, mxs, size\ ins, xt \vdash ins!pc, pc :: \Phi\ C\ M;$

$Some\ \sigma' = exec\ (P, None, h, (stk, loc, C, M, pc) \# frs) ;$

$P, \Phi \vdash (None, h, (stk, loc, C, M, pc) \# frs) \surd \rrbracket$

$\implies P, \Phi \vdash \sigma' \surd \langle proof \rangle$

**lemma** *IfFalse-correct*:

$\llbracket wf\ prog\ wt\ P;$

$P \vdash C\ sees\ M: Ts \rightarrow T = (mxs, mxl_0, ins, xt)\ in\ C;$

$ins!pc = IfFalse\ branch;$

$P, T, mxs, size\ ins, xt \vdash ins!pc, pc :: \Phi\ C\ M;$

$Some\ \sigma' = exec\ (P, None, h, (stk, loc, C, M, pc) \# frs) ;$

$P, \Phi \vdash (None, h, (stk, loc, C, M, pc) \# frs) \surd \rrbracket$

$\implies P, \Phi \vdash \sigma' \surd \langle proof \rangle$

**lemma** *CmpEq-correct*:

$\llbracket wf\ prog\ wt\ P;$

$P \vdash C\ sees\ M: Ts \rightarrow T = (mxs, mxl_0, ins, xt)\ in\ C;$

$ins!pc = CmpEq;$

$P, T, mxs, size\ ins, xt \vdash ins!pc, pc :: \Phi\ C\ M;$

$Some\ \sigma' = exec\ (P, None, h, (stk, loc, C, M, pc) \# frs) ;$

$P, \Phi \vdash (None, h, (stk, loc, C, M, pc) \# frs) \surd \rrbracket$

$\implies P, \Phi \vdash \sigma' \surd \langle proof \rangle$

**lemma** *Pop-correct*:

$\llbracket wf\ prog\ wt\ P;$

$P \vdash C\ sees\ M: Ts \rightarrow T = (mxs, mxl_0, ins, xt)\ in\ C;$



$$\begin{aligned}
& \text{ins} ! pc = \text{Pop}; \\
& P, T, \text{m\!xs}, \text{size ins}, xt \vdash \text{ins}!pc, pc :: \Phi C M; \\
& \text{Some } \sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, pc) \# \text{frs}) ; \\
& P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, pc) \# \text{frs}) \checkmark \ ] \\
\implies & P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *IAdd-correct*:

$$\begin{aligned}
& \llbracket \text{wf-prog wt } P; \\
& P \vdash C \text{ sees } M:Ts \rightarrow T = (\text{m\!xs}, \text{m\!xl}_0, \text{ins}, xt) \text{ in } C; \\
& \text{ins} ! pc = \text{IAdd}; \\
& P, T, \text{m\!xs}, \text{size ins}, xt \vdash \text{ins}!pc, pc :: \Phi C M; \\
& \text{Some } \sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, pc) \# \text{frs}) ; \\
& P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, pc) \# \text{frs}) \checkmark \ ] \\
\implies & P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *Throw-correct*:

$$\begin{aligned}
& \llbracket \text{wf-prog wt } P; \\
& P \vdash C \text{ sees } M:Ts \rightarrow T = (\text{m\!xs}, \text{m\!xl}_0, \text{ins}, xt) \text{ in } C; \\
& \text{ins} ! pc = \text{Throw}; \\
& \text{Some } \sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, pc) \# \text{frs}) ; \\
& P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, pc) \# \text{frs}) \checkmark ; \\
& \text{fst} (\text{exec-instr} (\text{ins}!pc) P h \text{stk loc } C M pc \text{ frs}) = \text{None} \ ] \\
\implies & P, \Phi \vdash \sigma' \checkmark \\
& \langle \text{proof} \rangle
\end{aligned}$$

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*:

$$\begin{aligned}
& \llbracket \text{wf-jvm-prog}_{\Phi} P; \\
& P \vdash C \text{ sees } M:Ts \rightarrow T = (\text{m\!xs}, \text{m\!xl}_0, \text{ins}, xt) \text{ in } C; \\
& \text{Some } \sigma' = \text{exec} (P, \text{None}, h, (\text{stk}, \text{loc}, C, M, pc) \# \text{frs}); \\
& P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, pc) \# \text{frs}) \checkmark \ ] \\
\implies & P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle
\end{aligned}$$

#### 4.22.4 Main

**lemma** *correct-state-impl-Some-method*:

$$\begin{aligned}
& P, \Phi \vdash (\text{None}, h, (\text{stk}, \text{loc}, C, M, pc) \# \text{frs}) \checkmark \\
\implies & \exists m Ts T. P \vdash C \text{ sees } M:Ts \rightarrow T = m \text{ in } C \\
& \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *BV-correct-1* [rule-format]:

$$\bigwedge \sigma. \llbracket \text{wf-jvm-prog}_{\Phi} P; P, \Phi \vdash \sigma \checkmark \ ] \implies P \vdash \sigma \text{ -jvm} \rightarrow_1 \sigma' \longrightarrow P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle$$

**theorem** *progress*:

$$\llbracket xp = \text{None}; \text{frs} \neq [] \ ] \implies \exists \sigma'. P \vdash (xp, h, \text{frs}) \text{ -jvm} \rightarrow_1 \sigma' \\
\langle \text{proof} \rangle$$

**lemma** *progress-conform*:

$$\begin{aligned}
& \llbracket \text{wf-jvm-prog}_{\Phi} P; P, \Phi \vdash (xp, h, \text{frs}) \checkmark ; xp = \text{None}; \text{frs} \neq [] \ ] \\
\implies & \exists \sigma'. P \vdash (xp, h, \text{frs}) \text{ -jvm} \rightarrow_1 \sigma' \wedge P, \Phi \vdash \sigma' \checkmark \langle \text{proof} \rangle
\end{aligned}$$

**theorem** *BV-correct* [rule-format]:

$\llbracket wf\text{-jvm}\text{-prog}_{\Phi} P; P \vdash \sigma \text{-jvm} \rightarrow \sigma' \rrbracket \Longrightarrow P, \Phi \vdash \sigma \surd \longrightarrow P, \Phi \vdash \sigma' \surd \langle \text{proof} \rangle$

**lemma** *hconf-start*:

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

**shows**  $P \vdash (start\text{-heap } P) \surd \langle \text{proof} \rangle$

**lemma** *BV-correct-initial*:

**shows**  $\llbracket wf\text{-jvm}\text{-prog}_{\Phi} P; P \vdash C \text{ sees } M:[] \rightarrow T = m \text{ in } C \rrbracket$

$\Longrightarrow P, \Phi \vdash start\text{-state } P C M \surd \langle \text{proof} \rangle$

**theorem** *typesafe*:

**assumes** *welltyped*:  $wf\text{-jvm}\text{-prog}_{\Phi} P$

**assumes** *main-method*:  $P \vdash C \text{ sees } M:[] \rightarrow T = m \text{ in } C$

**shows**  $P \vdash start\text{-state } P C M \text{-jvm} \rightarrow \sigma \Longrightarrow P, \Phi \vdash \sigma \surd \langle \text{proof} \rangle$

**end**

## 4.23 Welltyped Programs produce no Type Errors

```
theory BVNoTypeError
imports ../JVM/JVMDefensive BVSpecTypeSafe
begin
```

```
lemma has-methodI:
  P ⊢ C sees M:Ts→T = m in D ⇒ P ⊢ C has M
  ⟨proof⟩
```

Some simple lemmas about the type testing functions of the defensive JVM:

```
lemma typeof-NoneD [simp,dest]: typeof v = Some x ⇒ ¬is-Addr v
  ⟨proof⟩
```

```
lemma is-Ref-def2:
  is-Ref v = (v = Null ∨ (∃ a. v = Addr a))
  ⟨proof⟩
```

```
lemma [iff]: is-Ref Null ⟨proof⟩
```

```
lemma is-RefI [intro, simp]: P, h ⊢ v :≤ T ⇒ is-refT T ⇒ is-Ref v ⟨proof⟩
```

```
lemma is-IntgI [intro, simp]: P, h ⊢ v :≤ Integer ⇒ is-Intg v ⟨proof⟩
```

```
lemma is-BoolI [intro, simp]: P, h ⊢ v :≤ Boolean ⇒ is-Bool v ⟨proof⟩
```

```
declare defs1 [simp del]
```

```
lemma wt-jvm-prog-states:
  [[ wf-jvm-progΦ P; P ⊢ C sees M: Ts→T = (mxs, mxl, ins, et) in C;
    Φ C M ! pc = τ; pc < size ins ]]
  ⇒ OK τ ∈ states P mxs (1+size Ts+mxl) ⟨proof⟩
```

The main theorem: welltyped programs do not produce type errors if they are started in a conformant state.

```
theorem no-type-error:
  fixes σ :: jvm-state
  assumes welltyped: wf-jvm-progΦ P and conforms: P, Φ ⊢ σ √
  shows exec-d P σ ≠ TypeError ⟨proof⟩
```

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-progΦ P ⇒ P, Φ ⊢ σ √ ⇒ P ⊢ σ -jvm→ σ'
  ⇒ P ⊢ (Normal σ) -jvmd→ (Normal σ') ⟨proof⟩
```

As corollary we get that the aggressive and the defensive machine are equivalent for welltyped programs (if started in a conformant state or in the canonical start state)

```
corollary welltyped-commutes:
  fixes σ :: jvm-state
  assumes wf: wf-jvm-progΦ P and conforms: P, Φ ⊢ σ √
  shows P ⊢ (Normal σ) -jvmd→ (Normal σ') = P ⊢ σ -jvm→ σ'
  ⟨proof⟩
```

```
corollary welltyped-initial-commutes:
  assumes wf: wf-jvm-prog P
  assumes meth: P ⊢ C sees M:[]→T = b in C
```

**defines** *start*:  $\sigma \equiv \text{start-state } P \ C \ M$   
**shows**  $P \vdash (\text{Normal } \sigma) \text{ -jvmd} \rightarrow (\text{Normal } \sigma') = P \vdash \sigma \text{ -jvm} \rightarrow \sigma'$   
 ⟨*proof*⟩

**lemma** *not-TypeError-eq* [*iff*]:  
 $x \neq \text{TypeError} = (\exists t. x = \text{Normal } t)$   
 ⟨*proof*⟩

**locale** *cnf* =  
**fixes**  $P$  **and**  $\Phi$  **and**  $\sigma$   
**assumes** *wf*:  $wf\text{-jvm-prog}_{\Phi} \ P$   
**assumes** *cnf*:  $correct\text{-state } P \ \Phi \ \sigma$

**theorem** (**in** *cnf*) *no-type-errors*:  
 $P \vdash (\text{Normal } \sigma) \text{ -jvmd} \rightarrow \sigma' \implies \sigma' \neq \text{TypeError}$   
 ⟨*proof*⟩

**locale** *start* =  
**fixes**  $P$  **and**  $C$  **and**  $M$  **and**  $\sigma$  **and**  $T$  **and**  $b$   
**assumes** *wf*:  $wf\text{-jvm-prog } P$   
**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*⟩

**end**

## 4.24 Example Welltypings

```

theory BVExample
imports ../JVM/JVMListExample BVSpecTypeSafe BVExec
  ~~/src/HOL/Library/Code-Target-Numeral
begin

```

This theory shows type correctness of the example program in section 3.7 (p. 86) 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.

### 4.24.1 Setup

```

lemma distinct-classes':
  list-name ≠ test-name
  list-name ≠ Object
  list-name ≠ ClassCast
  list-name ≠ OutOfMemory
  list-name ≠ NullPointer
  test-name ≠ Object
  test-name ≠ OutOfMemory
  test-name ≠ ClassCast
  test-name ≠ NullPointer
  ClassCast ≠ NullPointer
  ClassCast ≠ Object
  NullPointer ≠ Object
  OutOfMemory ≠ ClassCast
  OutOfMemory ≠ NullPointer
  OutOfMemory ≠ Object
  ⟨proof⟩

```

```

lemmas distinct-classes = distinct-classes' distinct-classes' [symmetric]

```

```

lemma distinct-fields:
  val-name ≠ next-name
  next-name ≠ val-name
  ⟨proof⟩

```

Abbreviations for definitions we will have to use often in the proofs below:

```

lemmas system-defs = SystemClasses-def ObjectC-def NullPointerC-def
  OutOfMemoryC-def ClassCastC-def
lemmas class-defs = list-class-def test-class-def

```

These auxiliary proofs are for efficiency: class lookup, subclass relation, method and field lookup are computed only once:

```

lemma class-Object [simp]:
  class E Object = Some (undefined, [], [])
  ⟨proof⟩

```

```

lemma class-NullPointer [simp]:
  class E NullPointer = Some (Object, [], [])
  ⟨proof⟩

```

```

lemma class-OutOfMemory [simp]:

```

*class E OutOfMemory = Some (Object, [], [])*  
 ⟨proof⟩

**lemma** *class-ClassCast [simp]:*  
*class E ClassCast = Some (Object, [], [])*  
 ⟨proof⟩

**lemma** *class-list [simp]:*  
*class E list-name = Some list-class*  
 ⟨proof⟩

**lemma** *class-test [simp]:*  
*class E test-name = Some test-class*  
 ⟨proof⟩

**lemma** *E-classes [simp]:*  
 $\{C. \text{is-class } E \ C\} = \{\text{list-name}, \text{test-name}, \text{NullPointer},$   
 $\text{ClassCast}, \text{OutOfMemory}, \text{Object}\}$   
 ⟨proof⟩

The subclass relation spelled out:

**lemma** *subcls1:*  
 $\text{subcls1 } E = \{(\text{list-name}, \text{Object}), (\text{test-name}, \text{Object}), (\text{NullPointer}, \text{Object}),$   
 $(\text{ClassCast}, \text{Object}), (\text{OutOfMemory}, \text{Object})\}$ ⟨proof⟩

The subclass relation is acyclic; hence its converse is well founded:

**lemma** *notin-rtrancl:*  
 $(a, b) \in r^* \implies a \neq b \implies (\bigwedge y. (a, y) \notin r) \implies \text{False}$   
 ⟨proof⟩

**lemma** *acyclic-subcls1-E: acyclic (subcls1 E)*⟨proof⟩

**lemma** *wf-subcls1-E: wf ((subcls1 E)<sup>-1</sup>)*⟨proof⟩

Method and field lookup:

**lemma** *method-append [simp]:*  
*method E list-name append-name =*  
 $(\text{list-name}, [\text{Class list-name}], \text{Void}, 3, 0, \text{append-ins}, [(1, 2, \text{NullPointer}, 7, 0)])$ ⟨proof⟩

**lemma** *method-makelist [simp]:*  
*method E test-name makelist-name =*  
 $(\text{test-name}, [], \text{Void}, 3, 2, \text{make-list-ins}, [])$ ⟨proof⟩

**lemma** *field-val [simp]:*  
*field E list-name val-name = (list-name, Integer)*⟨proof⟩

**lemma** *field-next [simp]:*  
*field E list-name next-name = (list-name, Class list-name)*⟨proof⟩

**lemma** [simp]: *fields E Object = []*  
 ⟨proof⟩

**lemma** [simp]: *fields E NullPointer = []*  
 ⟨proof⟩

**lemma** [simp]: *fields E ClassCast = []*  
 ⟨proof⟩

**lemma** [simp]: *fields E OutOfMemory = []*

*<proof>*

**lemma** *[simp]: fields E test-name = []<proof>*

**lemmas** *[simp] = is-class-def*

#### 4.24.2 Program structure

The program is structurally wellformed:

**lemma** *wf-struct:*

*wf-prog* ( $\lambda G C mb. True$ ) *E* (**is** *wf-prog* ?*mb E*)<proof>

#### 4.24.3 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<sub>m</sub> ( $\varphi_a$ )*

**where**

$\varphi_a \equiv \text{map } (\lambda(x,y). \text{Some } (x, \text{map OK } y))$  [

- ( [], [Class list-name, Class list-name]),
- ( [Class list-name], [Class list-name, Class list-name]),
- ( [Class list-name], [Class list-name, Class list-name]),
- ( [Class list-name, Class list-name], [Class list-name, Class list-name]),
- ( [Class list-name, Class list-name], [Class list-name, Class list-name]),
- ( [NT, Class list-name, Class list-name], [Class list-name, Class list-name]),
- ( [Boolean, Class list-name], [Class list-name, Class list-name]),
- ( [Class Object], [Class list-name, Class list-name]),
- ( [], [Class list-name, Class list-name]),
- ( [Class list-name], [Class list-name, Class list-name]),
- ( [Class list-name, Class list-name], [Class list-name, Class list-name]),
- ( [], [Class list-name, Class list-name]),
- ( [Void], [Class list-name, Class list-name]),
- ( [Class list-name], [Class list-name, Class list-name]),
- ( [Class list-name, Class list-name], [Class list-name, Class list-name]),
- ( [Void], [Class list-name, Class list-name])

The next definition and three proof rules implement an algorithm to enumerate natural numbers. The command *apply (elim pc-end pc-next pc-0* transforms a goal of the form

$pc < n \implies P pc$

into a series of goals

$P (0::'a)$

$P (Suc 0)$

...

$P n$

**definition** *intervall* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* (*-* ∈ [*-*, *-*])

**where**

$$x \in [a, b] \equiv a \leq x \wedge x < b$$

**lemma** *pc-0*:  $x < n \implies (x \in [0, n] \implies P x) \implies P x$

*<proof>*

**lemma** *pc-next*:  $x \in [n0, n] \implies P n0 \implies (x \in [Suc n0, n] \implies P x) \implies P x$ *<proof>*

**lemma** *pc-end*:  $x \in [n, n] \implies P x$

*<proof>*

**lemma** *types-append* [*simp*]: *check-types E 3 (Suc (Suc 0)) (map OK  $\varphi_a$ )**<proof>*

**lemma** *wt-append* [*simp*]:

*wt-method E list-name [Class list-name] Void 3 0 append-ins*  
*[(Suc 0, 2, NullPointer, 7, 0)]  $\varphi_a$* *<proof>*

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 Ctest, Err , Err ]),
- ( [Clist], [OK Ctest, Err , Err ]),
- ( [], [OK Clist, Err , Err ]),
- ( [Clist], [OK Clist, Err , Err ]),
- ( [Integer, Clist], [OK Clist, Err , Err ]),
- ( [], [OK Clist, Err , Err ]),
- ( [Clist], [OK Clist, Err , Err ]),
- ( [], [OK Clist, OK Clist, Err ]),
- ( [Clist], [OK Clist, OK Clist, Err ]),
- ( [Integer, Clist], [OK Clist, OK Clist, 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]),
- ( [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  $\varphi_m$ )**<proof>*

**lemma** *wt-makelist* [*simp*]:

*wt-method E test-name [] Void 3 2 make-list-ins []  $\varphi_m$* *<proof>*



**lemma** *wf-md'E*:

```
[[ wf-prog wf-md P;
    $\wedge C S fs ms m. [(C, S, fs, ms) \in set P; m \in set ms] \implies wf-md' P C m ]$ 
   $\implies wf-prog wf-md' P \langle proof \rangle$ 
```

The whole program is welltyped:

**definition** *Phi* :: *ty<sub>P</sub>* ( $\Phi$ )

**where**

```
 $\Phi C mn \equiv$  if  $C = test-name \wedge mn = makelist-name$  then  $\varphi_m$  else
             if  $C = list-name \wedge mn = append-name$  then  $\varphi_a$  else []
```

**lemma** *wf-prog*:

```
wf-jvm-prog $\Phi$  E \langle proof \rangle
```

#### 4.24.4 Conformance

Execution of the program will be typesafe, because its start state conforms to the welltyping:

**lemma**  $E, \Phi \vdash start-state E test-name makelist-name \surd \langle proof \rangle$

#### 4.24.5 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] {}
\langle proof \rangle
```

**definition** *some-elem* :: '*a set*  $\Rightarrow$  '*a where* [*code del*]:

```
some-elem = (%S. SOME x. x : S)
```

**code-const** *some-elem*

```
(SML (case/ - of/ Set/ xs/ =>/ hd/ xs))
```

This code setup is just a demonstration and *not* sound!

**notepad begin**

```
\langle proof \rangle
```

**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)
\langle proof \rangle
```

**lemma** *JVM-sup-unfold* [*code*]:

```
JVM-SemiType.sup S m n = lift2 (Opt.sup
```

```

(Product.sup (Listn.sup (SemiType.sup S))
  (λx y. OK (map2 (lift2 (SemiType.sup S)) x y)))
⟨proof⟩

```

**lemmas** [code] = *SemiType.sup-def* [unfolding exec-lub-def] *JVM-le-unfold*

**lemmas** [code] = *lesub-def* *plussub-def*

**lemma** [code]:

```

is-refT T = (case T of NT ⇒ True | Class C ⇒ True | - ⇒ False)
⟨proof⟩

```

**declare** *app<sub>i</sub>.simps* [code]

**lemma** [code]:

```

appi (Getfield F C, P, pc, mxs, Tr, (T#ST, LT)) =
  Predicate.holds (Predicate.bind (sees-field-i-i-i-o-i P C F C) (λTf. if P ⊢ T ≤ Class C then
    Predicate.single () else bot))
⟨proof⟩

```

**lemma** [code]:

```

appi (Putfield F C, P, pc, mxs, Tr, (T1#T2#ST, LT)) =
  Predicate.holds (Predicate.bind (sees-field-i-i-i-o-i P C F C) (λTf. if P ⊢ T2 ≤ (Class C) ∧ P ⊢
    T1 ≤ Tf then Predicate.single () else bot))
⟨proof⟩

```

**lemma** [code]:

```

appi (Invoke M n, P, pc, mxs, Tr, (ST,LT)) =
  (n < length ST ∧
   (ST!n ≠ NT →
    (case ST!n of
     Class C ⇒ Predicate.holds (Predicate.bind (Method-i-i-i-o-o-o P C M) (λ(Ts, T, m, D). if
      P ⊢ rev (take n ST) [≤] Ts then Predicate.single () else bot))
     | - ⇒ False)))
⟨proof⟩

```

**lemmas** [code] =

```

SemiType.sup-def [unfolding 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* =  $\varphi_a$

**definition** *test4* **where** *test4* =  $\varphi_m$

⟨ML⟩

**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>*:

[[ *new-Addr h* = *Some a*; *P* ⊢ *C* has-fields *FDTs*; *h'* = *h(a ↦ (C, init-fields *FDTs*))* ]]

⇒⇒ *P* ⊢<sub>1</sub> ⟨*new C, (h, l)*⟩ ⇒ ⟨*addr a, (h', l)*⟩

| *NewFail<sub>1</sub>*:

*new-Addr h* = *None* ⇒⇒

*P* ⊢<sub>1</sub> ⟨*new C, (h, l)*⟩ ⇒ ⟨*THROW OutOfMemory, (h, l)*⟩

| *Cast<sub>1</sub>*:

[[ *P* ⊢<sub>1</sub> ⟨*e, s<sub>0</sub>*⟩ ⇒ ⟨*addr a, (h, l)*⟩; *h a* = *Some(D, fs)*; *P* ⊢ *D* ≼\* *C* ]]

⇒⇒ *P* ⊢<sub>1</sub> ⟨*Cast C e, s<sub>0</sub>*⟩ ⇒ ⟨*addr a, (h, l)*⟩

| *CastNull<sub>1</sub>*:

*P* ⊢<sub>1</sub> ⟨*e, s<sub>0</sub>*⟩ ⇒ ⟨*null, s<sub>1</sub>*⟩ ⇒⇒

*P* ⊢<sub>1</sub> ⟨*Cast C e, s<sub>0</sub>*⟩ ⇒ ⟨*null, s<sub>1</sub>*⟩

| *CastFail<sub>1</sub>*:

$$\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, l) \rangle; h \ a = \text{Some}(D, fs); \neg P \vdash D \preceq^* C \rrbracket$$

$$\implies P \vdash_1 \langle \text{Cast } C \ e, s_0 \rangle \Rightarrow \langle \text{THROW } \text{ClassCast}, (h, l) \rangle$$

| *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>:

$$P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \implies$$

$$P \vdash_1 \langle e \cdot F \{D\}, s_0 \rangle \Rightarrow \langle \text{THROW } \text{NullPointer}, s_1 \rangle$$

| *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>:

$$\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2) \rangle;$$

$$h_2 \ a = \text{Some}(C, fs); fs' = fs((F, D) \mapsto v); h_2' = h_2(a \mapsto (C, fs')) \rrbracket$$

$$\implies P \vdash_1 \langle e_1 \cdot F \{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{unit}, (h_2', l_2) \rangle$$

| *FAssNull*<sub>1</sub>:

$$\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, s_2 \rangle \rrbracket$$

$$\implies P \vdash_1 \langle e_1 \cdot F \{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{THROW } \text{NullPointer}, s_2 \rangle$$

| *FAssThrow*<sub>11</sub>:

$$P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle \implies$$

$$P \vdash_1 \langle e_1 \cdot F \{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_1 \rangle$$

| *FAssThrow*<sub>21</sub>:

$$\llbracket P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash_1 \langle e_2, s_1 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle \rrbracket$$

$$\implies P \vdash_1 \langle e_1 \cdot F \{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{throw } e', s_2 \rangle$$

| *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>:  
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle; P \vdash_1 \langle es, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_2, ls_2) \rangle;$   
 $h_2 \ a = \text{Some}(C, fs); P \vdash C \text{ sees } M: Ts \rightarrow T = \text{body in } D;$   
 $\text{size } vs = \text{size } Ts; ls_2' = (\text{Addr } a) \# vs \ @ \ \text{replicate } (\text{max-vars body}) \ \text{undefined};$   
 $P \vdash_1 \langle \text{body}, (h_2, ls_2') \rangle \Rightarrow \langle e', (h_3, ls_3) \rangle \rrbracket$   
 $\Longrightarrow P \vdash_1 \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle e', (h_3, ls_2) \rangle$

| *CallParamsThrow*<sub>1</sub>:  
 $\llbracket P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle; P \vdash_1 \langle es, s_1 \rangle [\Rightarrow] \langle es', s_2 \rangle;$   
 $es' = \text{map Val } vs \ @ \ \text{throw } ex \ \# \ es_2 \rrbracket$   
 $\Longrightarrow P \vdash_1 \langle e \cdot M(es), s_0 \rangle \Rightarrow \langle \text{throw } ex, s_2 \rangle$

| *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:

$$P \vdash_1 \langle es_0, (h_0, ls_0) \rangle [\Rightarrow] \langle es_1, (h_1, ls_1) \rangle \Longrightarrow \text{length } ls_0 = \text{length } ls_1 \langle \text{proof} \rangle$$

**lemma** *evals*<sub>1</sub>-preserves-len:

$$\bigwedge es' \ s \ s'. P \vdash_1 \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \Longrightarrow \text{length } es = \text{length } es' \langle \text{proof} \rangle$$

**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:  $P \vdash_1 \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \Longrightarrow \text{finals } es' \langle \text{proof} \rangle$

**end**

## 5.2 Well-Formedness of Intermediate Language

```
theory J1WellForm
imports ../J/JWellForm J1
begin
```

### 5.2.1 Well-Typedness

#### type-synonym

$env_1 = ty\ list$  — type environment indexed by variable number

#### inductive

```
WT1 :: [J1-prog, env1, expr1, ty] ⇒ bool
  ((-, - ⊢1 / - :: -) [51, 51, 51] 50)
and WTs1 :: [J1-prog, env1, expr1 list, ty list] ⇒ bool
  ((-, - ⊢1 / - :::] -) [51, 51, 51] 50)
for P :: J1-prog
```

#### where

```
WTNew1:
is-class P C ⇒
P, E ⊢1 new C :: Class C
```

```
| WTCast1:
[[ P, E ⊢1 e :: Class D; is-class P C; P ⊢ C ≤* D ∨ P ⊢ D ≤* C ]
⇒ P, E ⊢1 Cast C e :: Class C
```

```
| WTVal1:
typeof v = Some T ⇒
P, E ⊢1 Val v :: T
```

```
| WTVar1:
[[ E!i = T; i < size E ]
⇒ P, E ⊢1 Var i :: T
```

```
| WTBinOp1:
[[ P, E ⊢1 e1 :: T1; P, E ⊢1 e2 :: T2;
  case bop of Eq ⇒ (P ⊢ T1 ≤ T2 ∨ P ⊢ T2 ≤ T1) ∧ T = Boolean
  | Add ⇒ T1 = Integer ∧ T2 = Integer ∧ T = Integer ]
⇒ P, E ⊢1 e1 <<bop>> e2 :: T
```

```
| WTLAss1:
[[ E!i = T; i < size E; P, E ⊢1 e :: T'; P ⊢ T' ≤ T ]
⇒ P, E ⊢1 i:=e :: Void
```

```
| WTFAcc1:
[[ P, E ⊢1 e :: Class C; P ⊢ C sees F:T in D ]
⇒ P, E ⊢1 e.F{D} :: T
```

```
| WTFAss1:
[[ P, E ⊢1 e1 :: Class C; P ⊢ C sees F:T in D; P, E ⊢1 e2 :: T'; P ⊢ T' ≤ T ]
⇒ P, E ⊢1 e1.F{D} := e2 :: Void
```

```
| WTCall1:
```



$$\begin{aligned} & \llbracket P, E \vdash_1 e :: \text{Class } C; P \vdash C \text{ sees } M: Ts' \rightarrow T = m \text{ in } D; \\ & \quad P, E \vdash_1 es \llbracket :: Ts; P \vdash Ts \llbracket \leq Ts' \rrbracket \\ & \implies P, E \vdash_1 e \cdot M(es) :: T \end{aligned}$$

$$\begin{aligned} & | \text{WTBlock}_1: \\ & \llbracket \text{is-type } P T; P, E@[T] \vdash_1 e :: T' \rrbracket \\ & \implies P, E \vdash_1 \{i:T; e\} :: T' \end{aligned}$$

$$\begin{aligned} & | \text{WTSeq}_1: \\ & \llbracket P, E \vdash_1 e_1 :: T_1; P, E \vdash_1 e_2 :: T_2 \rrbracket \\ & \implies P, E \vdash_1 e_1; e_2 :: T_2 \end{aligned}$$

$$\begin{aligned} & | \text{WTCond}_1: \\ & \llbracket P, E \vdash_1 e :: \text{Boolean}; P, E \vdash_1 e_1 :: T_1; P, E \vdash_1 e_2 :: T_2; \\ & \quad P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1; P \vdash T_1 \leq T_2 \longrightarrow T = T_2; P \vdash T_2 \leq T_1 \longrightarrow T = T_1 \rrbracket \\ & \implies P, E \vdash_1 \text{if } (e) e_1 \text{ else } e_2 :: T \end{aligned}$$

$$\begin{aligned} & | \text{WTWhile}_1: \\ & \llbracket P, E \vdash_1 e :: \text{Boolean}; P, E \vdash_1 c :: T \rrbracket \\ & \implies P, E \vdash_1 \text{while } (e) c :: \text{Void} \end{aligned}$$

$$\begin{aligned} & | \text{WTThrow}_1: \\ & P, E \vdash_1 e :: \text{Class } C \implies \\ & P, E \vdash_1 \text{throw } e :: \text{Void} \end{aligned}$$

$$\begin{aligned} & | \text{WTTry}_1: \\ & \llbracket P, E \vdash_1 e_1 :: T; P, E@[ \text{Class } C ] \vdash_1 e_2 :: T; \text{is-class } P C \rrbracket \\ & \implies P, E \vdash_1 \text{try } e_1 \text{ catch}(C i) e_2 :: T \end{aligned}$$

$$\begin{aligned} & | \text{WTNil}_1: \\ & P, E \vdash_1 [] \llbracket :: \rrbracket [] \end{aligned}$$

$$\begin{aligned} & | \text{WTCons}_1: \\ & \llbracket P, E \vdash_1 e :: T; P, E \vdash_1 es \llbracket :: Ts \rrbracket \\ & \implies P, E \vdash_1 e \# es \llbracket :: T \# Ts \rrbracket \end{aligned}$$

**lemma**  $WTs_1$ -same-size:  $\bigwedge Ts. P, E \vdash_1 es \llbracket :: Ts \implies \text{size } es = \text{size } Ts \langle \text{proof} \rangle$

**lemma**  $WT_1$ -unique:

$$\begin{aligned} & P, E \vdash_1 e :: T_1 \implies (\bigwedge T_2. P, E \vdash_1 e :: T_2 \implies T_1 = T_2) \text{ and} \\ & P, E \vdash_1 es \llbracket :: Ts_1 \implies (\bigwedge Ts_2. P, E \vdash_1 es \llbracket :: Ts_2 \implies Ts_1 = Ts_2) \langle \text{proof} \rangle \end{aligned}$$

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

**shows**  $WT_1$ -is-type:  $P, E \vdash_1 e :: T \implies \text{set } E \subseteq \text{types } P \implies \text{is-type } P T$

**and**  $P, E \vdash_1 es \llbracket :: Ts \implies \text{True} \langle \text{proof} \rangle$

## 5.2.2 Well-formedness

— Indices in blocks increase by 1

**primrec**  $\mathcal{B} :: \text{expr}_1 \Rightarrow \text{nat} \Rightarrow \text{bool}$

**and**  $\mathcal{B}s :: \text{expr}_1 \text{ list} \Rightarrow \text{nat} \Rightarrow \text{bool}$  **where**

$\mathcal{B} (\text{new } C) i = \text{True} \mid$

$\mathcal{B} (\text{Cast } C e) i = \mathcal{B} e i \mid$

$$\begin{aligned}
& \mathcal{B} (\text{Val } v) i = \text{True} \mid \\
& \mathcal{B} (e_1 \ll \text{bop} \gg e_2) i = (\mathcal{B} e_1 i \wedge \mathcal{B} e_2 i) \mid \\
& \mathcal{B} (\text{Var } j) i = \text{True} \mid \\
& \mathcal{B} (e \cdot F\{D\}) i = \mathcal{B} e i \mid \\
& \mathcal{B} (j := e) i = \mathcal{B} e i \mid \\
& \mathcal{B} (e_1 \cdot F\{D\} := e_2) i = (\mathcal{B} e_1 i \wedge \mathcal{B} e_2 i) \mid \\
& \mathcal{B} (e \cdot M(es)) i = (\mathcal{B} e i \wedge \mathcal{B} s es i) \mid \\
& \mathcal{B} (\{j:T ; e\}) i = (i = j \wedge \mathcal{B} e (i+1)) \mid \\
& \mathcal{B} (e_1 ; e_2) i = (\mathcal{B} e_1 i \wedge \mathcal{B} e_2 i) \mid \\
& \mathcal{B} (\text{if } (e) e_1 \text{ else } e_2) i = (\mathcal{B} e i \wedge \mathcal{B} e_1 i \wedge \mathcal{B} e_2 i) \mid \\
& \mathcal{B} (\text{throw } e) i = \mathcal{B} e i \mid \\
& \mathcal{B} (\text{while } (e) c) i = (\mathcal{B} e i \wedge \mathcal{B} c i) \mid \\
& \mathcal{B} (\text{try } e_1 \text{ catch}(C j) e_2) i = (\mathcal{B} e_1 i \wedge i=j \wedge \mathcal{B} e_2 (i+1)) \mid \\
& \mathcal{B} s [] i = \text{True} \mid \\
& \mathcal{B} s (e \# es) i = (\mathcal{B} e i \wedge \mathcal{B} s es i)
\end{aligned}$$

**definition**  $wf\text{-}J_1\text{-mdecl} :: J_1\text{-prog} \Rightarrow \text{cname} \Rightarrow \text{expr}_1 \text{ mdecl} \Rightarrow \text{bool}$

**where**

$$\begin{aligned}
& wf\text{-}J_1\text{-mdecl } P C \equiv \lambda(M, Ts, T, body). \\
& (\exists T'. P, \text{Class } C \# Ts \vdash_1 body :: T' \wedge P \vdash T' \leq T) \wedge \\
& \mathcal{D} body [\{..size Ts\}] \wedge \mathcal{B} body (size Ts + 1)
\end{aligned}$$

**lemma**  $wf\text{-}J_1\text{-mdecl}[\text{simp}]$ :

$$\begin{aligned}
& wf\text{-}J_1\text{-mdecl } P C (M, Ts, T, body) \equiv \\
& ((\exists T'. P, \text{Class } C \# Ts \vdash_1 body :: T' \wedge P \vdash T' \leq T) \wedge \\
& \mathcal{D} body [\{..size Ts\}] \wedge \mathcal{B} body (size Ts + 1)) \langle \text{proof} \rangle
\end{aligned}$$

**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 ⇒ 'b) ⇒ 'a mdecl ⇒ 'b mdecl

**where**

compM f ≡ λ(M, Ts, T, m). (M, Ts, T, f m)

**definition** compC :: ('a ⇒ 'b) ⇒ 'a cdecl ⇒ 'b cdecl

**where**

compC f ≡ λ(C,D,Fdecls,Mdecls). (C,D,Fdecls, map (compM f) Mdecls)

**definition** compP :: ('a ⇒ 'b) ⇒ 'a prog ⇒ 'b prog

**where**

compP f ≡ 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-of-map4:

map-of (map (λ(x,a,b,c).(x,a,b,f c)) ts) =  
Option.map (λ(a,b,c).(a,b,f c)) ◦ (map-of ts)⟨proof⟩

**lemma** class-compP:

class P C = Some (D, fs, ms)  
⇒ class (compP f P) C = Some (D, fs, map (compM f) ms)⟨proof⟩

**lemma** class-compPD:

class (compP f P) C = Some (D, fs, cms)  
⇒ ∃ ms. class P C = Some(D,fs,ms) ∧ cms = map (compM f) ms⟨proof⟩

**lemma** [simp]: is-class (compP f P) C = is-class P C⟨proof⟩

**lemma** [simp]: class (compP f P) C = Option.map (λc. snd(compC f (C,c))) (class P C)⟨proof⟩

**lemma** sees-methods-compP:

P ⊢ C sees-methods Mm ⇒  
compP f P ⊢ C sees-methods (Option.map (λ((Ts,T,m),D). ((Ts,T,f m),D)) ◦ Mm)⟨proof⟩

**lemma** sees-method-compP:

P ⊢ C sees M: Ts→T = m in D ⇒  
compP f P ⊢ C sees M: Ts→T = (f m) in D⟨proof⟩

**lemma** [simp]:

P ⊢ C sees M: Ts→T = m in D ⇒  
method (compP f P) C M = (D,Ts,T,f m)⟨proof⟩

**lemma** sees-methods-compPD:

[[ cP ⊢ C sees-methods Mm'; cP = compP f P ]] ⇒  
∃ Mm. P ⊢ C sees-methods Mm ∧  
Mm' = (Option.map (λ((Ts,T,m),D). ((Ts,T,f m),D)) ◦ Mm)⟨proof⟩

**lemma** sees-method-compPD:

$compP f P \vdash C \text{ sees } M: Ts \rightarrow T = fm \text{ in } D \implies$   
 $\exists m. P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } D \wedge f m = fm \langle proof \rangle$

**lemma**  $[simp]: subcls1(compP f P) = subcls1 P \langle proof \rangle$

**lemma**  $compP\text{-widen}[simp]: (compP f P \vdash T \leq T') = (P \vdash T \leq T') \langle proof \rangle$

**lemma**  $[simp]: (compP f P \vdash Ts \leq Ts') = (P \vdash Ts \leq Ts') \langle proof \rangle$

**lemma**  $[simp]: is\text{-type } (compP f P) T = is\text{-type } P T \langle proof \rangle$

**lemma**  $[simp]: (compP (f::'a \Rightarrow 'b) P \vdash C \text{ has-fields } FDTs) = (P \vdash C \text{ has-fields } FDTs) \langle proof \rangle$

**lemma**  $[simp]: fields (compP f P) C = fields P C \langle proof \rangle$

**lemma**  $[simp]: (compP f P \vdash C \text{ sees } F:T \text{ in } D) = (P \vdash C \text{ sees } F:T \text{ in } D) \langle proof \rangle$

**lemma**  $[simp]: field (compP f P) F D = field P F D \langle proof \rangle$

### 5.3.1 Invariance of wf-prog under compilation

**lemma**  $[iff]: distinct\text{-fst } (compP f P) = distinct\text{-fst } P \langle proof \rangle$

**lemma**  $[iff]: distinct\text{-fst } (map (compM f) ms) = distinct\text{-fst } ms \langle proof \rangle$

**lemma**  $[iff]: wf\text{-syscls } (compP f P) = wf\text{-syscls } P \langle proof \rangle$

**lemma**  $[iff]: wf\text{-fdecl } (compP f P) = wf\text{-fdecl } P \langle proof \rangle$

**lemma**  $set\text{-compP}$ :  
 $((C, D, fs, ms') \in set(compP f P)) =$   
 $(\exists ms. (C, D, fs, ms) \in set P \wedge ms' = map (compM f) ms) \langle proof \rangle$

**lemma**  $wf\text{-cdecl}\text{-compPI}$ :  
 $\llbracket \bigwedge C M Ts T m.$   
 $\llbracket wf\text{-mdecl } wf_1 P C (M, Ts, T, m); P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } C \rrbracket$   
 $\implies wf\text{-mdecl } wf_2 (compP f P) C (M, Ts, T, f m);$   
 $\forall x \in set P. wf\text{-cdecl } wf_1 P x; x \in set (compP f P); wf\text{-prog } p P \rrbracket$   
 $\implies wf\text{-cdecl } wf_2 (compP f P) x \langle proof \rangle$

**lemma**  $wf\text{-prog}\text{-compPI}$ :

**assumes**  $lift$ :

$\bigwedge C M Ts T m.$

$\llbracket P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } C; wf\text{-mdecl } wf_1 P C (M, Ts, T, m) \rrbracket$   
 $\implies wf\text{-mdecl } wf_2 (compP f P) C (M, Ts, T, f m)$

**and**  $wf: wf\text{-prog } wf_1 P$

**shows**  $wf\text{-prog } wf_2 (compP f P) \langle proof \rangle$

**end**

**theory** *List-Index* **imports** *Main* **begin**

This theory defines three functions for finding the index of items in a list:

*find-index*  $P$   $xs$  finds the index of the first element in  $xs$  that satisfies  $P$ .

*index*  $xs$   $x$  finds the index of the first occurrence of  $x$  in  $xs$ .

*last-index*  $xs$   $x$  finds the index of the last occurrence of  $x$  in  $xs$ .

All functions return *length*  $xs$  if  $xs$  does not contain a suitable element.

The argument order of *find-index* follows the function of the same name in the Haskell standard library. For *index* (and *last-index*) the order is intentionally reversed: *index* maps lists to a mapping from elements to their indices, almost the inverse of function *nth*.

**primrec** *find-index* :: ('a ⇒ bool) ⇒ 'a list ⇒ nat **where**  
*find-index* - [] = 0 |  
*find-index*  $P$  (x#xs) = (if  $P$   $x$  then 0 else *find-index*  $P$  xs + 1)

**definition** *index* :: 'a list ⇒ 'a ⇒ nat **where**  
*index*  $xs$  = (λa. *find-index* (λx. x=a)  $xs$ )

**definition** *last-index* :: 'a list ⇒ 'a ⇒ nat **where**  
*last-index*  $xs$   $x$  =  
 (let  $i$  = *index* (rev  $xs$ )  $x$ ;  $n$  = *size*  $xs$   
 in if  $i$  =  $n$  then  $i$  else  $n - (i+1)$ )

**lemma** *find-index-le-size*: *find-index*  $P$   $xs$  ≤ *size*  $xs$   
 ⟨proof⟩

**lemma** *index-le-size*: *index*  $xs$   $x$  ≤ *size*  $xs$   
 ⟨proof⟩

**lemma** *last-index-le-size*: *last-index*  $xs$   $x$  ≤ *size*  $xs$   
 ⟨proof⟩

**lemma** *index-Nil[simp]*: *index* []  $a$  = 0  
 ⟨proof⟩

**lemma** *index-Cons[simp]*: *index* (x#xs)  $a$  = (if  $x=a$  then 0 else *index*  $xs$   $a$  + 1)  
 ⟨proof⟩

**lemma** *index-append*: *index* (xs @ ys)  $x$  =  
 (if  $x$  : set  $xs$  then *index*  $xs$   $x$  else *size*  $xs$  + *index*  $ys$   $x$ )  
 ⟨proof⟩

**lemma** *index-conv-size-if-notin[simp]*:  $x \notin$  set  $xs \implies$  *index*  $xs$   $x$  = *size*  $xs$   
 ⟨proof⟩

**lemma** *find-index-eq-size-conv*:  
*size*  $xs$  =  $n \implies$  (*find-index*  $P$   $xs$  =  $n$ ) = (ALL  $x$  : set  $xs$ . ~  $P$   $x$ )  
 ⟨proof⟩

**lemma** *size-eq-find-index-conv*:  
*size*  $xs$  =  $n \implies$  ( $n$  = *find-index*  $P$   $xs$ ) = (ALL  $x$  : set  $xs$ . ~  $P$   $x$ )  
 ⟨proof⟩

**lemma** *index-size-conv*: *size*  $xs$  =  $n \implies$  (*index*  $xs$   $x$  =  $n$ ) = ( $x \notin$  set  $xs$ )  
 ⟨proof⟩

**lemma** *size-index-conv*:  $size\ xs = n \implies (n = index\ xs\ x) = (x \notin set\ xs)$   
 ⟨proof⟩

**lemma** *last-index-size-conv*:  
 $size\ xs = n \implies (last-index\ xs\ x = n) = (x \notin set\ xs)$   
 ⟨proof⟩

**lemma** *size-last-index-conv*:  
 $size\ xs = n \implies (n = last-index\ xs\ x) = (x \notin set\ xs)$   
 ⟨proof⟩

**lemma** *find-index-less-size-conv*:  
 $(find-index\ P\ xs < size\ xs) = (EX\ x : set\ xs. P\ x)$   
 ⟨proof⟩

**lemma** *index-less-size-conv*:  
 $(index\ xs\ x < size\ xs) = (x \in set\ xs)$   
 ⟨proof⟩

**lemma** *last-index-less-size-conv*:  
 $(last-index\ xs\ x < size\ xs) = (x : set\ xs)$   
 ⟨proof⟩

**lemma** *index-less[simp]*:  
 $x : set\ xs \implies size\ xs \leq n \implies index\ xs\ x < n$   
 ⟨proof⟩

**lemma** *last-index-less[simp]*:  
 $x : set\ xs \implies size\ xs \leq n \implies last-index\ xs\ x < n$   
 ⟨proof⟩

**lemma** *last-index-Cons*:  $last-index\ (x\#\ xs)\ y =$   
 (if  $x=y$  then  
   if  $x \in set\ xs$  then  $last-index\ xs\ y + 1$  else 0  
 else  $last-index\ xs\ y + 1$ )  
 ⟨proof⟩

**lemma** *last-index-append*:  $last-index\ (xs\ @\ ys)\ x =$   
 (if  $x : set\ ys$  then  $size\ xs + last-index\ ys\ x$   
 else if  $x : set\ xs$  then  $last-index\ xs\ x$  else  $size\ xs + size\ ys$ )  
 ⟨proof⟩

**lemma** *last-index-Snoc[simp]*:  
 $last-index\ (xs\ @\ [x])\ y =$   
 (if  $x=y$  then  $size\ xs$   
 else if  $y : set\ xs$  then  $last-index\ xs\ y$  else  $size\ xs + 1$ )  
 ⟨proof⟩

**lemma** *nth-find-index*:  $find-index\ P\ xs < size\ xs \implies P(xs\ !\ find-index\ P\ xs)$   
 ⟨proof⟩

**lemma** *nth-index[simp]*:  $x \in set\ xs \implies xs\ !\ index\ xs\ x = x$   
 ⟨proof⟩

**lemma** *nth-last-index[simp]*:  $x \in \text{set } xs \implies xs ! \text{last-index } xs \ x = x$   
 ⟨proof⟩

**lemma** *index-eq-index-conv[simp]*:  $x \in \text{set } xs \vee y \in \text{set } xs \implies$   
 $(\text{index } xs \ x = \text{index } xs \ y) = (x = y)$   
 ⟨proof⟩

**lemma** *last-index-eq-index-conv[simp]*:  $x \in \text{set } xs \vee y \in \text{set } xs \implies$   
 $(\text{last-index } xs \ x = \text{last-index } xs \ y) = (x = y)$   
 ⟨proof⟩

**lemma** *inj-on-index*:  $\text{inj-on } (\text{index } xs) (\text{set } xs)$   
 ⟨proof⟩

**lemma** *inj-on-last-index*:  $\text{inj-on } (\text{last-index } xs) (\text{set } xs)$   
 ⟨proof⟩

**lemma** *index-conv-takeWhile*:  $\text{index } xs \ x = \text{size}(\text{takeWhile } (\lambda y. x \neq y) \ xs)$   
 ⟨proof⟩

**lemma** *index-take*:  $\text{index } xs \ x \geq i \implies x \notin \text{set}(\text{take } i \ xs)$   
 ⟨proof⟩

**lemma** *last-index-drop*:  
 $\text{last-index } xs \ x < i \implies x \notin \text{set}(\text{drop } i \ xs)$   
 ⟨proof⟩

**end**

**theory** *Hidden*

**imports** *../../List-Index/List-Index*

**begin**

**definition** *hidden* ::  $'a \ \text{list} \Rightarrow \text{nat} \Rightarrow \text{bool}$  **where**  
 $\text{hidden } xs \ i \equiv i < \text{size } xs \wedge xs ! i \in \text{set}(\text{drop } (i+1) \ xs)$

**lemma** *hidden-last-index*:  $x \in \text{set } xs \implies \text{hidden } (xs @ [x]) (\text{last-index } xs \ x)$   
 ⟨proof⟩

**lemma** *hidden-inacc*:  $\text{hidden } xs \ i \implies \text{last-index } xs \ x \neq i$   
 ⟨proof⟩

**lemma** *[simp]*:  $\text{hidden } xs \ i \implies \text{hidden } (xs @ [x]) \ i$   
 ⟨proof⟩

**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⟩

**lemma** *map-upds-apply-eq-Some*:

$((m(xs[\mapsto]ys)) x = \text{Some } y) =$   
 $(\text{let } xs' = \text{take } (\text{size } ys) \text{ } xs$   
 $\text{ in if } x \in \text{set } xs' \text{ then } ys ! \text{last-index } xs' \text{ } x = y \text{ else } m \text{ } x = \text{Some } y)$   
 <proof>

**lemma** *map-upds-upd-conv-last-index*:

$\llbracket x \in \text{set } xs; \text{size } xs \leq \text{size } ys \rrbracket$   
 $\implies m(xs[\mapsto]ys)(x \mapsto y) = m(xs[\mapsto]ys[\text{last-index } xs \text{ } x := y])$   
 <proof>

**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*]: *compEs<sub>1</sub> Vs es* = *map (compE<sub>1</sub> Vs) es*⟨*proof*⟩

**primrec** *fin<sub>1</sub>*:: *expr*  $\Rightarrow$  *expr<sub>1</sub>* **where**

```

  fin1(Val v) = Val v
| fin1(throw e) = throw(fin1 e)

```

**lemma** *comp-final*: *final e*  $\Longrightarrow$  *compE<sub>1</sub> Vs e* = *fin<sub>1</sub> e*⟨*proof*⟩

**lemma** [*simp*]:

```

   $\bigwedge$  Vs. max-vars (compE1 Vs e) = max-vars e
and  $\bigwedge$  Vs. max-varss (compEs1 Vs es) = max-varss es⟨proof⟩

```

Compiling programs:

**definition** *compP<sub>1</sub>* :: *J-prog*  $\Rightarrow$  *J<sub>1</sub>-prog*

**where**

```

  compP1  $\equiv$  compP ( $\lambda$ (pns,body). compE1 (this#pns) body)

```

**end**

## 5.5 Correctness of Stage 1

```
theory Correctness1
imports J1WellForm Compiler1
begin
```

### 5.5.1 Correctness of program compilation

```
primrec unmod :: expr1 ⇒ nat ⇒ bool
  and unmods :: expr1 list ⇒ nat ⇒ bool where
unmod (new C) i = True |
unmod (Cast C e) i = unmod e i |
unmod (Val v) i = True |
unmod (e1 «bop» e2) i = (unmod e1 i ∧ unmod e2 i) |
unmod (Var i) j = True |
unmod (i:=e) j = (i ≠ j ∧ unmod e j) |
unmod (e·F{D}) i = unmod e i |
unmod (e1·F{D}:=e2) i = (unmod e1 i ∧ unmod e2 i) |
unmod (e·M(es)) i = (unmod e i ∧ unmods es i) |
unmod {j:T; e} i = unmod e i |
unmod (e1;;e2) i = (unmod e1 i ∧ unmod e2 i) |
unmod (if (e) e1 else e2) i = (unmod e i ∧ unmod e1 i ∧ unmod e2 i) |
unmod (while (e) c) i = (unmod e i ∧ unmod c i) |
unmod (throw e) i = unmod e i |
unmod (try e1 catch(C i) e2) j = (unmod e1 j ∧ (if i=j then False else unmod e2 j)) |

unmods ([]) i = True |
unmods (e#es) i = (unmod e i ∧ unmods es i)
```

**lemma** *hidden-unmod*:  $\bigwedge Vs. \text{hidden } Vs \ i \implies \text{unmod } (\text{comp}E_1 \ Vs \ e) \ i$  **and**  
 $\bigwedge Vs. \text{hidden } Vs \ i \implies \text{unmods } (\text{comp}Es_1 \ Vs \ es) \ i$  *<proof>*

**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**  $\llbracket P \vdash_1 \langle es, (h, ls) \rangle \Rightarrow \langle es', (h', ls') \rangle; \text{unmods } es \ i; \ i < \text{size } ls \rrbracket$   
 $\implies ls \ ! \ i = ls' \ ! \ i$  *<proof>*

**lemma** *LAss-lem*:  
 $\llbracket x \in \text{set } xs; \text{size } xs \leq \text{size } ys \rrbracket$   
 $\implies m_1 \subseteq_m m_2(xs[\mapsto]ys) \implies m_1(x \mapsto y) \subseteq_m m_2(xs[\mapsto]ys[\text{last-index } xs \ x := y])$  *<proof>* **lemma**

*Block-lem*:

```
fixes l :: 'a → 'b
assumes 0: l ⊆m [Vs [↦] ls]
  and 1: l' ⊆m [Vs [↦] ls', V ↦ v]
  and hidden: V ∈ set Vs ⟹ ls ! last-index Vs V = ls' ! last-index Vs V
  and size: size ls = size ls'   size Vs < size ls'
shows l'(V := l V) ⊆m [Vs [↦] ls'] <proof>
```

The main theorem:

**theorem** *assumes wf*: *wuf-J-prog P*  
**shows** *eval<sub>1</sub>-eval*:  $P \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle$   
 $\implies (\bigwedge Vs \ ls. \llbracket \text{fv } e \subseteq \text{set } Vs; \ l \subseteq_m [Vs[\mapsto]ls]; \text{size } Vs + \text{max-vars } e \leq \text{size } ls \rrbracket$   
 $\implies \exists ls'. \text{comp}P_1 \ P \vdash_1 \langle \text{comp}E_1 \ Vs \ e, (h, ls) \rangle \Rightarrow \langle \text{fin}_1 \ e', (h', ls') \rangle \wedge l' \subseteq_m [Vs[\mapsto]ls']$ )

**and**  $evals_1$ - $evals$ :  $P \vdash \langle es, (h, l) \rangle [\Rightarrow] \langle es', (h', l') \rangle$   
 $\implies (\bigwedge Vs \text{ } ls. \llbracket fvs \text{ } es \subseteq set \text{ } Vs; l \subseteq_m [Vs \mapsto] ls \rrbracket; size \text{ } Vs + max\text{-}varss \text{ } es \leq size \text{ } ls \rrbracket$   
 $\implies \exists ls'. compP_1 P \vdash_1 \langle compEs_1 Vs \text{ } es, (h, ls) \rangle [\Rightarrow] \langle compEs_1 Vs \text{ } es', (h', ls') \rangle \wedge$   
 $l' \subseteq_m [Vs \mapsto] ls' \rangle \langle proof \rangle$

### 5.5.2 Preservation of well-formedness

The compiler preserves well-formedness. Is less trivial than it may appear. We start with two simple properties: preservation of well-typedness

**lemma**  $compE_1$ - $pres$ - $wt$ :  $\bigwedge Vs \text{ } Ts \text{ } U.$   
 $\llbracket P, [Vs \mapsto] Ts \vdash e :: U; size \text{ } Ts = size \text{ } Vs \rrbracket$   
 $\implies compP f P, Ts \vdash_1 compE_1 Vs e :: U$   
**and**  $\bigwedge Vs \text{ } Ts \text{ } Us.$   
 $\llbracket P, [Vs \mapsto] Ts \vdash es [::] Us; size \text{ } Ts = size \text{ } Vs \rrbracket$   
 $\implies compP f P, Ts \vdash_1 compEs_1 Vs es [::] Us \langle proof \rangle$

and the correct block numbering:

**lemma**  $\mathcal{B}$ :  $\bigwedge Vs \text{ } n. size \text{ } Vs = n \implies \mathcal{B} (compE_1 Vs e) n$   
**and**  $\mathcal{B}s$ :  $\bigwedge Vs \text{ } n. size \text{ } Vs = n \implies \mathcal{B}s (compEs_1 Vs es) n \langle proof \rangle$

The main complication is preservation of definite assignment  $\mathcal{D}$ .

**lemma**  $image$ - $last$ - $index$ :  $A \subseteq set(xs @ [x]) \implies last\text{-}index (xs @ [x]) ' A =$   
*(if  $x \in A$  then insert (size  $xs$ ) (last-index  $xs$  ' ( $A - \{x\}$ )) else last-index  $xs$  '  $A$ )*  $\langle proof \rangle$

**lemma**  $A$ - $compE_1$ - $None$ [ $simp$ ]:  
 $\bigwedge Vs. \mathcal{A} e = None \implies \mathcal{A} (compE_1 Vs e) = None$   
**and**  $\bigwedge Vs. \mathcal{A}s es = None \implies \mathcal{A}s (compEs_1 Vs es) = None \langle proof \rangle$

**lemma**  $A$ - $compE_1$ :  
 $\bigwedge A \text{ } Vs. \llbracket \mathcal{A} e = [A]; fv e \subseteq set \text{ } Vs \rrbracket \implies \mathcal{A} (compE_1 Vs e) = [last\text{-}index \text{ } Vs ' A]$   
**and**  $\bigwedge A \text{ } Vs. \llbracket \mathcal{A}s es = [A]; fvs es \subseteq set \text{ } Vs \rrbracket \implies \mathcal{A}s (compEs_1 Vs es) = [last\text{-}index \text{ } Vs ' A] \langle proof \rangle$

**lemma**  $D$ - $None$ [ $iff$ ]:  $\mathcal{D} (e :: 'a \text{ } exp) None$  **and** [ $iff$ ]:  $\mathcal{D}s (es :: 'a \text{ } exp \text{ } list) None \langle proof \rangle$

**lemma**  $D$ - $last$ - $index$ - $compE_1$ :  
 $\bigwedge A \text{ } Vs. \llbracket A \subseteq set \text{ } Vs; fv e \subseteq set \text{ } Vs \rrbracket \implies$   
 $\mathcal{D} e [A] \implies \mathcal{D} (compE_1 Vs e) [last\text{-}index \text{ } Vs ' A]$   
**and**  $\bigwedge A \text{ } Vs. \llbracket A \subseteq set \text{ } Vs; fvs es \subseteq set \text{ } Vs \rrbracket \implies$   
 $\mathcal{D}s es [A] \implies \mathcal{D}s (compEs_1 Vs es) [last\text{-}index \text{ } Vs ' A] \langle proof \rangle$

**lemma**  $last$ - $index$ - $image$ - $set$ :  $distinct \text{ } xs \implies last\text{-}index \text{ } xs ' set \text{ } xs = \{.. < size \text{ } xs\} \langle proof \rangle$

**lemma**  $D$ - $compE_1$ :  
 $\llbracket \mathcal{D} e [set \text{ } Vs]; fv e \subseteq set \text{ } Vs; distinct \text{ } Vs \rrbracket \implies \mathcal{D} (compE_1 Vs e) [\{.. < length \text{ } Vs\}] \langle proof \rangle$

**lemma**  $D$ - $compE_1$ ':  
**assumes**  $\mathcal{D} e [set(V \# Vs)]$  **and**  $fv e \subseteq set(V \# Vs)$  **and**  $distinct(V \# Vs)$   
**shows**  $\mathcal{D} (compE_1 (V \# Vs) e) [\{.. length \text{ } Vs\}] \langle proof \rangle$

**lemma**  $compP_1$ - $pres$ - $wf$ :  $wf\text{-}J\text{-}prog P \implies wf\text{-}J_1\text{-}prog (compP_1 P) \langle proof \rangle$

**end**

## 5.6 Compilation Stage 2

**theory** *Compiler2*

**imports** *PCompiler J1 ../JVM/JVMExec*

**begin**

**primrec** *compE<sub>2</sub>* :: *expr<sub>1</sub>* ⇒ *instr list*

**and** *compEs<sub>2</sub>* :: *expr<sub>1</sub> list* ⇒ *instr list* **where**

*compE<sub>2</sub>* (*new C*) = [*New C*]

| *compE<sub>2</sub>* (*Cast C e*) = *compE<sub>2</sub>* *e* @ [*Checkcast C*]

| *compE<sub>2</sub>* (*Val v*) = [*Push v*]

| *compE<sub>2</sub>* (*e<sub>1</sub> <<bop>> e<sub>2</sub>*) = *compE<sub>2</sub>* *e<sub>1</sub>* @ *compE<sub>2</sub>* *e<sub>2</sub>* @

(*case bop of Eq* ⇒ [*CmpEq*]

| *Add* ⇒ [*IAdd*])

| *compE<sub>2</sub>* (*Var i*) = [*Load i*]

| *compE<sub>2</sub>* (*i:=e*) = *compE<sub>2</sub>* *e* @ [*Store i, Push Unit*]

| *compE<sub>2</sub>* (*e·F{D}*) = *compE<sub>2</sub>* *e* @ [*Getfield F D*]

| *compE<sub>2</sub>* (*e<sub>1</sub>·F{D} := e<sub>2</sub>*) =

*compE<sub>2</sub>* *e<sub>1</sub>* @ *compE<sub>2</sub>* *e<sub>2</sub>* @ [*Putfield F D, Push Unit*]

| *compE<sub>2</sub>* (*e·M(es)*) = *compE<sub>2</sub>* *e* @ *compEs<sub>2</sub>* *es* @ [*Invoke M (size es)*]

| *compE<sub>2</sub>* (*{i:T; e}*) = *compE<sub>2</sub>* *e*

| *compE<sub>2</sub>* (*e<sub>1</sub>;;e<sub>2</sub>*) = *compE<sub>2</sub>* *e<sub>1</sub>* @ [*Pop*] @ *compE<sub>2</sub>* *e<sub>2</sub>*

| *compE<sub>2</sub>* (*if (e) e<sub>1</sub> else e<sub>2</sub>*) =

(*let cnd* = *compE<sub>2</sub>* *e*;

*thn* = *compE<sub>2</sub>* *e<sub>1</sub>*;

*els* = *compE<sub>2</sub>* *e<sub>2</sub>*;

*test* = *Iffalse (int(size thn + 2))*;

*thnex* = *Goto (int(size els + 1))*

*in cnd* @ [*test*] @ *thn* @ [*thnex*] @ *els*)

| *compE<sub>2</sub>* (*while (e) c*) =

(*let cnd* = *compE<sub>2</sub>* *e*;

*bdy* = *compE<sub>2</sub>* *c*;

*test* = *Iffalse (int(size bdy + 3))*;

*loop* = *Goto (-int(size bdy + size cnd + 2))*

*in cnd* @ [*test*] @ *bdy* @ [*Pop*] @ [*loop*] @ [*Push Unit*])

| *compE<sub>2</sub>* (*throw e*) = *compE<sub>2</sub>* *e* @ [*instr.Throw*]

| *compE<sub>2</sub>* (*try e<sub>1</sub> catch(C i) e<sub>2</sub>*) =

(*let catch* = *compE<sub>2</sub>* *e<sub>2</sub>*

*in compE<sub>2</sub>* *e<sub>1</sub>* @ [*Goto (int(size catch)+2), Store i*] @ *catch*)

| *compEs<sub>2</sub>* [] = []

| *compEs<sub>2</sub>* (*e#es*) = *compE<sub>2</sub>* *e* @ *compEs<sub>2</sub>* *es*

Compilation of exception table. Is given start address of code to compute absolute addresses necessary in exception table.

**primrec** *compxE<sub>2</sub>* :: *expr<sub>1</sub>* ⇒ *pc* ⇒ *nat* ⇒ *ex-table*

**and** *compxEs<sub>2</sub>* :: *expr<sub>1</sub> list* ⇒ *pc* ⇒ *nat* ⇒ *ex-table* **where**

*compxE<sub>2</sub>* (*new C*) *pc d* = []

| *compxE<sub>2</sub>* (*Cast C e*) *pc d* = *compxE<sub>2</sub>* *e* *pc d*

| *compxE<sub>2</sub>* (*Val v*) *pc d* = []

| *compxE<sub>2</sub>* (*e<sub>1</sub> <<bop>> e<sub>2</sub>*) *pc d* =

*compxE<sub>2</sub>* *e<sub>1</sub>* *pc d* @ *compxE<sub>2</sub>* *e<sub>2</sub>* (*pc + size(compE<sub>2</sub> e<sub>1</sub>)*) (*d+1*)

| *compxE<sub>2</sub>* (*Var i*) *pc d* = []

| *compxE<sub>2</sub>* (*i:=e*) *pc d* = *compxE<sub>2</sub>* *e* *pc d*

$| \text{compxE}_2 (e \cdot F\{D\}) \text{ pc } d = \text{compxE}_2 e \text{ pc } d$   
 $| \text{compxE}_2 (e_1 \cdot F\{D\} := e_2) \text{ pc } d =$   
 $\quad \text{compxE}_2 e_1 \text{ pc } d @ \text{compxE}_2 e_2 (\text{pc} + \text{size}(\text{compE}_2 e_1)) (d+1)$   
 $| \text{compxE}_2 (e \cdot M(es)) \text{ pc } d =$   
 $\quad \text{compxE}_2 e \text{ pc } d @ \text{compxEs}_2 es (\text{pc} + \text{size}(\text{compE}_2 e)) (d+1)$   
 $| \text{compxE}_2 (\{i:T; e\}) \text{ pc } d = \text{compxE}_2 e \text{ pc } d$   
 $| \text{compxE}_2 (e_1;;e_2) \text{ pc } d =$   
 $\quad \text{compxE}_2 e_1 \text{ pc } d @ \text{compxE}_2 e_2 (\text{pc} + \text{size}(\text{compE}_2 e_1) + 1) d$   
 $| \text{compxE}_2 (\text{if } (e) e_1 \text{ else } e_2) \text{ pc } d =$   
 $\quad (\text{let } \text{pc}_1 = \text{pc} + \text{size}(\text{compE}_2 e) + 1;$   
 $\quad \quad \text{pc}_2 = \text{pc}_1 + \text{size}(\text{compE}_2 e_1) + 1$   
 $\quad \text{in } \text{compxE}_2 e \text{ pc } d @ \text{compxE}_2 e_1 \text{ pc}_1 d @ \text{compxE}_2 e_2 \text{ pc}_2 d)$   
 $| \text{compxE}_2 (\text{while } (b) e) \text{ pc } d =$   
 $\quad \text{compxE}_2 b \text{ pc } d @ \text{compxE}_2 e (\text{pc} + \text{size}(\text{compE}_2 b) + 1) d$   
 $| \text{compxE}_2 (\text{throw } e) \text{ pc } d = \text{compxE}_2 e \text{ pc } d$   
 $| \text{compxE}_2 (\text{try } e_1 \text{ catch}(C i) e_2) \text{ pc } d =$   
 $\quad (\text{let } \text{pc}_1 = \text{pc} + \text{size}(\text{compE}_2 e_1)$   
 $\quad \text{in } \text{compxE}_2 e_1 \text{ pc } d @ \text{compxE}_2 e_2 (\text{pc}_1 + 2) d @ [(pc, \text{pc}_1, C, \text{pc}_1 + 1, d)])$

$| \text{compxEs}_2 [] \text{ pc } d = []$   
 $| \text{compxEs}_2 (e \# es) \text{ pc } d = \text{compxE}_2 e \text{ pc } d @ \text{compxEs}_2 es (\text{pc} + \text{size}(\text{compE}_2 e)) (d+1)$

**primrec** *max-stack* :: *expr*<sub>1</sub> ⇒ *nat*

**and** *max-stacks* :: *expr*<sub>1</sub> *list* ⇒ *nat* **where**

$\text{max-stack } (\text{new } C) = 1$   
 $\text{max-stack } (\text{Cast } C e) = \text{max-stack } e$   
 $\text{max-stack } (\text{Val } v) = 1$   
 $\text{max-stack } (e_1 \ll \text{bop} \gg e_2) = \max (\text{max-stack } e_1) (\text{max-stack } e_2) + 1$   
 $\text{max-stack } (\text{Var } i) = 1$   
 $\text{max-stack } (i := e) = \text{max-stack } e$   
 $\text{max-stack } (e \cdot F\{D\}) = \text{max-stack } e$   
 $\text{max-stack } (e_1 \cdot F\{D\} := e_2) = \max (\text{max-stack } e_1) (\text{max-stack } e_2) + 1$   
 $\text{max-stack } (e \cdot M(es)) = \max (\text{max-stack } e) (\text{max-stacks } es) + 1$   
 $\text{max-stack } (\{i:T; e\}) = \text{max-stack } e$   
 $\text{max-stack } (e_1;;e_2) = \max (\text{max-stack } e_1) (\text{max-stack } e_2)$   
 $\text{max-stack } (\text{if } (e) e_1 \text{ else } e_2) =$   
 $\quad \max (\text{max-stack } e) (\max (\text{max-stack } e_1) (\text{max-stack } e_2))$   
 $\text{max-stack } (\text{while } (e) c) = \max (\text{max-stack } e) (\text{max-stack } c)$   
 $\text{max-stack } (\text{throw } e) = \text{max-stack } e$   
 $\text{max-stack } (\text{try } e_1 \text{ catch}(C i) e_2) = \max (\text{max-stack } e_1) (\text{max-stack } e_2)$

$\text{max-stacks } [] = 0$   
 $\text{max-stacks } (e \# es) = \max (\text{max-stack } e) (1 + \text{max-stacks } es)$

**lemma** *max-stack1*:  $1 \leq \text{max-stack } e$  *<proof>*

**definition** *compMb*<sub>2</sub> :: *expr*<sub>1</sub> ⇒ *jvm-method*

**where**

$\text{compMb}_2 \equiv \lambda \text{body.}$   
 $\text{let } \text{ins} = \text{compE}_2 \text{ body } @ [\text{Return}];$   
 $\quad \text{xt} = \text{compxE}_2 \text{ body } 0 0$   
 $\text{in } (\text{max-stack } \text{body}, \text{max-vars } \text{body}, \text{ins}, \text{xt})$

**definition** *compP*<sub>2</sub> :: *J*<sub>1</sub>-*prog* ⇒ *jvm-prog*

**where**

$$\text{comp}P_2 \equiv \text{comp}P \text{ comp}Mb_2$$

**lemma** *compMb<sub>2</sub>* [*simp*]:

$$\text{comp}Mb_2 e = (\text{max-stack } e, \text{max-vars } e, \text{comp}E_2 e @ [\text{Return}], \text{comp}xE_2 e 0 0) \langle \text{proof} \rangle$$

**end**

## 5.7 Correctness of Stage 2

```
theory Correctness2
imports ~~/src/HOL/Library/Sublist Compiler2
begin
```

### 5.7.1 Instruction sequences

How to select individual instructions and subsequences of instructions from a program given the class, method and program counter.

**definition** *before* :: *jvm-prog*  $\Rightarrow$  *cname*  $\Rightarrow$  *mname*  $\Rightarrow$  *nat*  $\Rightarrow$  *instr list*  $\Rightarrow$  *bool*  
 $((-, -, -, - / \triangleright -)$  [51,0,0,0,51] 50) **where**  
 $P, C, M, pc \triangleright is \longleftrightarrow prefixeq\ is\ (drop\ pc\ (instrs\ of\ P\ C\ M))$

**definition** *at* :: *jvm-prog*  $\Rightarrow$  *cname*  $\Rightarrow$  *mname*  $\Rightarrow$  *nat*  $\Rightarrow$  *instr*  $\Rightarrow$  *bool*  
 $((-, -, -, - / \triangleright -)$  [51,0,0,0,51] 50) **where**  
 $P, C, M, pc \triangleright i \longleftrightarrow (\exists is. drop\ pc\ (instrs\ of\ P\ C\ M) = i\#\ is)$

**lemma** [*simp*]:  $P, C, M, pc \triangleright [] \langle proof \rangle$

**lemma** [*simp*]:  $P, C, M, pc \triangleright (i\#\ is) = (P, C, M, pc \triangleright i \wedge P, C, M, pc + 1 \triangleright is) \langle proof \rangle$

**lemma** [*simp*]:  $P, C, M, pc \triangleright (is_1 @ is_2) = (P, C, M, pc \triangleright is_1 \wedge P, C, M, pc + size\ is_1 \triangleright is_2) \langle proof \rangle$

**lemma** [*simp*]:  $P, C, M, pc \triangleright i \Longrightarrow instrs\ of\ P\ C\ M\ !\ pc = i \langle proof \rangle$

**lemma** *beforeM*:  
 $P \vdash C\ sees\ M: Ts \rightarrow T = body\ in\ D \Longrightarrow$   
 $compP_2\ P, D, M, 0 \triangleright compE_2\ body @ [Return] \langle proof \rangle$

This lemma executes a single instruction by rewriting:

**lemma** [*simp*]:  
 $P, C, M, pc \triangleright instr \Longrightarrow$   
 $(P \vdash (None, h, (vs, ls, C, M, pc) \# frs) -jvm \rightarrow \sigma') =$   
 $((None, h, (vs, ls, C, M, pc) \# frs) = \sigma' \vee$   
 $(\exists \sigma. exec(P, (None, h, (vs, ls, C, M, pc) \# frs)) = Some\ \sigma \wedge P \vdash \sigma -jvm \rightarrow \sigma')) \langle proof \rangle$

### 5.7.2 Exception tables

**definition** *pcs* :: *ex-table*  $\Rightarrow$  *nat set*  
**where**  
 $pcs\ xt \equiv \bigcup (f, t, C, h, d) \in set\ xt. \{f ..< t\}$

**lemma** *pcs-subset*:  
**shows**  $\bigwedge pc\ d. pcs(compxE_2\ e\ pc\ d) \subseteq \{pc..<pc+size(compE_2\ e)\}$   
**and**  $\bigwedge pc\ d. pcs(compxEs_2\ es\ pc\ d) \subseteq \{pc..<pc+size(compEs_2\ es)\} \langle proof \rangle$

**lemma** [*simp*]:  $pcs\ [] = \{\} \langle proof \rangle$

**lemma** [*simp*]:  $pcs\ (x\#\ xt) = \{fst\ x\ ..<fst(snd\ x)\} \cup pcs\ xt \langle proof \rangle$

**lemma** [*simp*]:  $pcs(xt_1 @ xt_2) = pcs\ xt_1 \cup pcs\ xt_2 \langle proof \rangle$

**lemma** [simp]:  $pc < pc_0 \vee pc_0 + \text{size}(\text{comp}E_2 e) \leq pc \implies pc \notin \text{pcs}(\text{comp}x E_2 e pc_0 d)$ ⟨proof⟩

**lemma** [simp]:  $pc < pc_0 \vee pc_0 + \text{size}(\text{comp}Es_2 es) \leq pc \implies pc \notin \text{pcs}(\text{comp}x Es_2 es pc_0 d)$ ⟨proof⟩

**lemma** [simp]:  $pc_1 + \text{size}(\text{comp}E_2 e_1) \leq pc_2 \implies \text{pcs}(\text{comp}x E_2 e_1 pc_1 d_1) \cap \text{pcs}(\text{comp}x E_2 e_2 pc_2 d_2) = \{\}$ ⟨proof⟩

**lemma** [simp]:  $pc_1 + \text{size}(\text{comp}E_2 e) \leq pc_2 \implies \text{pcs}(\text{comp}x E_2 e pc_1 d_1) \cap \text{pcs}(\text{comp}x Es_2 es pc_2 d_2) = \{\}$ ⟨proof⟩

**lemma** [simp]:  
 $pc \notin \text{pcs } xt_0 \implies \text{match-ex-table } P C pc (xt_0 @ xt_1) = \text{match-ex-table } P C pc xt_1$ ⟨proof⟩

**lemma** [simp]:  $\llbracket x \in \text{set } xt; pc \notin \text{pcs } xt \rrbracket \implies \neg \text{matches-ex-entry } P D pc x$ ⟨proof⟩

**lemma** [simp]:  
**assumes**  $xe: xe \in \text{set}(\text{comp}x E_2 e pc d)$  **and** **outside:**  $pc' < pc \vee pc + \text{size}(\text{comp}E_2 e) \leq pc'$   
**shows**  $\neg \text{matches-ex-entry } P C pc' xe$ ⟨proof⟩

**lemma** [simp]:  
**assumes**  $xe: xe \in \text{set}(\text{comp}x Es_2 es pc d)$  **and** **outside:**  $pc' < pc \vee pc + \text{size}(\text{comp}Es_2 es) \leq pc'$   
**shows**  $\neg \text{matches-ex-entry } P C pc' xe$ ⟨proof⟩

**lemma** *match-ex-table-app*[simp]:  
 $\forall xte \in \text{set } xt_1. \neg \text{matches-ex-entry } P D pc xte \implies$   
 $\text{match-ex-table } P D pc (xt_1 @ xt) = \text{match-ex-table } P D pc xt$ ⟨proof⟩

**lemma** [simp]:  
 $\forall x \in \text{set } xtab. \neg \text{matches-ex-entry } P C pc x \implies$   
 $\text{match-ex-table } P C pc xtab = \text{None}$ ⟨proof⟩

**lemma** *match-ex-entry*:  
 $\text{matches-ex-entry } P C pc (\text{start}, \text{end}, \text{catch-type}, \text{handler}) =$   
 $(\text{start} \leq pc \wedge pc < \text{end} \wedge P \vdash C \preceq^* \text{catch-type})$ ⟨proof⟩

**definition** *caught* ::  $\text{jvm-prog} \Rightarrow pc \Rightarrow \text{heap} \Rightarrow \text{addr} \Rightarrow \text{ex-table} \Rightarrow \text{bool}$  **where**  
 $\text{caught } P pc h a xt \longleftrightarrow$   
 $(\exists \text{entry} \in \text{set } xt. \text{matches-ex-entry } P (\text{cname-of } h a) pc \text{entry})$

**definition** *beforex* ::  $\text{jvm-prog} \Rightarrow \text{cname} \Rightarrow \text{mname} \Rightarrow \text{ex-table} \Rightarrow \text{nat set} \Rightarrow \text{nat} \Rightarrow \text{bool}$   
 $((\text{?}, /-, /- \triangleright / - /' / -, /-) [51, 0, 0, 0, 0, 51] 50)$  **where**  
 $P, C, M \triangleright xt / I, d \longleftrightarrow$   
 $(\exists xt_0 xt_1. \text{ex-table-of } P C M = xt_0 @ xt @ xt_1 \wedge \text{pcs } xt_0 \cap I = \{\} \wedge \text{pcs } xt \subseteq I \wedge$   
 $(\forall pc \in I. \forall C pc' d'. \text{match-ex-table } P C pc xt_1 = \lfloor (pc', d') \rfloor \longrightarrow d' \leq d))$

**definition** *dummyx* ::  $\text{jvm-prog} \Rightarrow \text{cname} \Rightarrow \text{mname} \Rightarrow \text{ex-table} \Rightarrow \text{nat set} \Rightarrow \text{nat} \Rightarrow \text{bool}$   $((\text{?}, -, \triangleright / - /' / -, -) [51, 0, 0, 0, 0, 51] 50)$  **where**  
 $P, C, M \triangleright xt / I, d \longleftrightarrow P, C, M \triangleright xt / I, d$

**lemma** *beforexD1*:  $P, C, M \triangleright xt / I, d \implies \text{pcs } xt \subseteq I$ ⟨proof⟩

**lemma** *beforex-mono*:  $\llbracket P, C, M \triangleright xt / I, d'; d' \leq d \rrbracket \implies P, C, M \triangleright xt / I, d$ ⟨proof⟩

**lemma** [simp]:  $P, C, M \triangleright xt / I, d \implies P, C, M \triangleright xt / I, \text{Suc } d$ ⟨proof⟩



**lemma** *beforex-append[simp]*:

$$\begin{aligned} & pcs\ x_1 \cap pcs\ x_2 = \{\} \implies \\ & P, C, M \triangleright x_1 @ x_2 / I, d = \\ & (P, C, M \triangleright x_1 / I - pcs\ x_2, d \wedge P, C, M \triangleright x_2 / I - pcs\ x_1, d \wedge P, C, M \triangleright x_1 @ x_2 / I, d) \langle proof \rangle \end{aligned}$$

**lemma** *beforex-appendD1*:

$$\begin{aligned} & \llbracket P, C, M \triangleright x_1 @ x_2 @ [(f, t, D, h, d)] / I, d; \\ & \quad pcs\ x_1 \subseteq J; J \subseteq I; J \cap pcs\ x_2 = \{\} \rrbracket \\ & \implies P, C, M \triangleright x_1 / J, d \langle proof \rangle \end{aligned}$$

**lemma** *beforex-appendD2*:

$$\begin{aligned} & \llbracket P, C, M \triangleright x_1 @ x_2 @ [(f, t, D, h, d)] / I, d; \\ & \quad pcs\ x_2 \subseteq J; J \subseteq I; J \cap pcs\ x_1 = \{\} \rrbracket \\ & \implies P, C, M \triangleright x_2 / J, d \langle proof \rangle \end{aligned}$$

**lemma** *beforexM*:

$$\begin{aligned} & P \vdash C\ sees\ M: Ts \rightarrow T = body\ in\ D \implies \\ & compP_2\ P, D, M \triangleright compxE_2\ body\ 0\ 0 / \{.. < size(compE_2\ body)\}, 0 \langle proof \rangle \end{aligned}$$

**lemma** *match-ex-table-SomeD2*:

$$\begin{aligned} & \llbracket match-ex-table\ P\ D\ pc\ (ex-table-of\ P\ C\ M) = [(pc', d')] ; \\ & \quad P, C, M \triangleright xt / I, d; \forall x \in set\ xt. \neg matches-ex-entry\ P\ D\ pc\ x; pc \in I \rrbracket \\ & \implies d' \leq d \langle proof \rangle \end{aligned}$$

**lemma** *match-ex-table-SomeD1*:

$$\begin{aligned} & \llbracket match-ex-table\ P\ D\ pc\ (ex-table-of\ P\ C\ M) = [(pc', d')] ; \\ & \quad P, C, M \triangleright xt / I, d; pc \in I; pc \notin pcs\ xt \rrbracket \implies d' \leq d \langle proof \rangle \end{aligned}$$

### 5.7.3 The correctness proof

**definition**

$$\begin{aligned} & handle :: jvm-prog \Rightarrow cname \Rightarrow mname \Rightarrow addr \Rightarrow heap \Rightarrow val\ list \Rightarrow val\ list \Rightarrow nat \Rightarrow frame\ list \\ & \quad \Rightarrow jvm-state\ \mathbf{where} \\ & handle\ P\ C\ M\ a\ h\ vs\ ls\ pc\ frs = find-handler\ P\ a\ h\ ((vs, ls, C, M, pc) \# frs) \end{aligned}$$

**lemma** *handle-Cons*:

$$\begin{aligned} & \llbracket P, C, M \triangleright xt / I, d; d \leq size\ vs; pc \in I; \\ & \quad \forall x \in set\ xt. \neg matches-ex-entry\ P\ (cname-of\ h\ xa)\ pc\ x \rrbracket \implies \\ & handle\ P\ C\ M\ xa\ h\ (v \# vs)\ ls\ pc\ frs = handle\ P\ C\ M\ xa\ h\ vs\ ls\ pc\ frs \langle proof \rangle \end{aligned}$$

**lemma** *handle-append*:

$$\begin{aligned} & \llbracket P, C, M \triangleright xt / I, d; d \leq size\ vs; pc \in I; pc \notin pcs\ xt \rrbracket \implies \\ & handle\ P\ C\ M\ xa\ h\ (ws @ vs)\ ls\ pc\ frs = handle\ P\ C\ M\ xa\ h\ vs\ ls\ pc\ frs \langle proof \rangle \end{aligned}$$

**lemma** *aux-isin[simp]*:  $\llbracket B \subseteq A; a \in B \rrbracket \implies a \in A \langle proof \rangle$

**lemma** *fixes P<sub>1</sub> defines [simp]*:  $P \equiv compP_2\ P_1$

**shows** *Jcc*:

$$\begin{aligned} & P_1 \vdash_1 \langle e, (h_0, ls_0) \rangle \Rightarrow \langle ef, (h_1, ls_1) \rangle \implies \\ & (\bigwedge C\ M\ pc\ v\ xa\ vs\ frs\ I. \\ & \quad \llbracket P, C, M, pc \triangleright compE_2\ e; P, C, M \triangleright compxE_2\ e\ pc\ (size\ vs) / I, size\ vs; \\ & \quad \quad \{pc.. < pc + size(compE_2\ e)\} \subseteq I \rrbracket \implies \end{aligned}$$

$$\begin{aligned}
& (ef = \text{Val } v \longrightarrow \\
& \quad P \vdash (\text{None}, h_0, (vs, ls_0, C, M, pc) \# frs) \text{--jvm--} \\
& \quad \quad (\text{None}, h_1, (v \# vs, ls_1, C, M, pc + \text{size}(\text{comp}E_2 \ e)) \# frs)) \\
& \wedge \\
& (ef = \text{Throw } xa \longrightarrow \\
& \quad (\exists pc_1. pc \leq pc_1 \wedge pc_1 < pc + \text{size}(\text{comp}E_2 \ e) \wedge \\
& \quad \quad \neg \text{caught } P \ pc_1 \ h_1 \ xa \ (\text{comp}xE_2 \ e \ pc \ (\text{size } vs)) \wedge \\
& \quad \quad P \vdash (\text{None}, h_0, (vs, ls_0, C, M, pc) \# frs) \text{--jvm--} \text{handle } P \ C \ M \ xa \ h_1 \ vs \ ls_1 \ pc_1 \ frs))) \\
\mathbf{and} \ P_1 \vdash_1 \langle es, (h_0, ls_0) \rangle [\Rightarrow] \langle fs, (h_1, ls_1) \rangle \Rightarrow \\
& (\wedge C \ M \ pc \ ws \ xa \ es' \ vs \ frs \ I. \\
& \quad \llbracket P, C, M, pc \triangleright \text{comp}Es_2 \ es; P, C, M \triangleright \text{comp}xEs_2 \ es \ pc \ (\text{size } vs) / I, \text{size } vs; \\
& \quad \quad \{pc.. < pc + \text{size}(\text{comp}Es_2 \ es)\} \subseteq I \rrbracket \Rightarrow \\
& (fs = \text{map } \text{Val } ws \longrightarrow \\
& \quad P \vdash (\text{None}, h_0, (vs, ls_0, C, M, pc) \# frs) \text{--jvm--} \\
& \quad \quad (\text{None}, h_1, (\text{rev } ws \ @ \ vs, ls_1, C, M, pc + \text{size}(\text{comp}Es_2 \ es)) \# frs)) \\
& \wedge \\
& (fs = \text{map } \text{Val } ws \ @ \ \text{Throw } xa \ \# \ es' \longrightarrow \\
& \quad (\exists pc_1. pc \leq pc_1 \wedge pc_1 < pc + \text{size}(\text{comp}Es_2 \ es) \wedge \\
& \quad \quad \neg \text{caught } P \ pc_1 \ h_1 \ xa \ (\text{comp}xEs_2 \ es \ pc \ (\text{size } vs)) \wedge \\
& \quad \quad P \vdash (\text{None}, h_0, (vs, ls_0, C, M, pc) \# frs) \text{--jvm--} \text{handle } P \ C \ M \ xa \ h_1 \ vs \ ls_1 \ pc_1 \ frs))) \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *atLeast0AtMost*[simp]:  $\{0::\text{nat}..n\} = \{..n\}$   
 $\langle \text{proof} \rangle$

**lemma** *atLeast0LessThan*[simp]:  $\{0::\text{nat}..<n\} = \{..<n\}$   
 $\langle \text{proof} \rangle$

**fun** *exception* :: 'a exp  $\Rightarrow$  addr option where  
*exception* (Throw a) = Some a  
| *exception* e = None

**lemma** *comp<sub>2</sub>-correct*:

**assumes** *method*:  $P_1 \vdash C \ \text{sees } M:Ts \rightarrow T = \text{body in } C$

**and** *eval*:  $P_1 \vdash_1 \langle \text{body}, (h, ls) \rangle \Rightarrow \langle e', (h', ls') \rangle$

**shows** *compP<sub>2</sub>*  $P_1 \vdash (\text{None}, h, ([], ls, C, M, 0)) \text{--jvm--} (\text{exception } e', h', []) \langle \text{proof} \rangle$

**end**

## 5.8 Combining Stages 1 and 2

**theory** *Compiler*

**imports** *Correctness1 Correctness2*

**begin**

**definition** *J2JVM* :: *J-prog*  $\Rightarrow$  *jvm-prog*

**where**

*J2JVM*  $\equiv$  *compP*<sub>2</sub>  $\circ$  *compP*<sub>1</sub>

**theorem** *comp-correct*:

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

**and** *method*:  $P \vdash C \text{ sees } M:Ts \rightarrow T = (pns, body) \text{ in } C$

**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*  $\vdash (None, h, ([], vs @ rest, C, M, 0)) \text{ --jvm--} \rightarrow (exception\ e', h', []) \langle proof \rangle$

**end**

## 5.9 Preservation of Well-Typedness

```

theory TypeComp
imports Compiler ../BV/BVSpec
begin

locale TC0 =
  fixes P :: J1-prog and mxl :: nat
begin

definition ty E e = (THE T. P, E ⊢1 e :: T)

definition tyl E A' = map (λi. if i ∈ A' ∧ i < size E then OK(E!i) else Err) [0..mxl]

definition tyi' ST E A = (case A of None ⇒ None | [A'] ⇒ Some(ST, tyl E A'))

definition after E A ST e = tyi' (ty E e # ST) E (A ⊔ A e)

end

lemma (in TC0) ty-def2 [simp]: P, E ⊢1 e :: T ⇒ ty E e = T⟨proof⟩
lemma (in TC0) [simp]: tyi' ST E None = None⟨proof⟩
lemma (in TC0) tyl-app-diff [simp]:
  tyl (E@[T]) (A - {size E}) = tyl E A⟨proof⟩

lemma (in TC0) tyi'-app-diff [simp]:
  tyi' ST (E @ [T]) (A ⊖ size E) = tyi' ST E A⟨proof⟩

lemma (in TC0) tyl-antimono:
  A ⊆ A' ⇒ P ⊢ tyl E A' [≤⊤] tyl E A⟨proof⟩

lemma (in TC0) tyi'-antimono:
  A ⊆ A' ⇒ P ⊢ tyi' ST E [A'] ≤' tyi' ST E [A]⟨proof⟩

lemma (in TC0) tyl-env-antimono:
  P ⊢ tyl (E@[T]) A [≤⊤] tyl E A⟨proof⟩

lemma (in TC0) tyi'-env-antimono:
  P ⊢ tyi' ST (E@[T]) A ≤' tyi' ST E A⟨proof⟩

lemma (in TC0) tyi'-incr:
  P ⊢ tyi' ST (E @ [T]) [insert (size E) A] ≤' tyi' ST E [A]⟨proof⟩

lemma (in TC0) tyl-incr:
  P ⊢ tyl (E @ [T]) (insert (size E) A) [≤⊤] tyl E A⟨proof⟩

lemma (in TC0) tyl-in-types:
  set E ⊆ types P ⇒ tyl E A ∈ list mxl (err (types P))⟨proof⟩
locale TC1 = TC0
begin

primrec compT :: ty list ⇒ nat hyperset ⇒ ty list ⇒ expr1 ⇒ tyi' list and
  compTs :: ty list ⇒ nat hyperset ⇒ ty list ⇒ expr1 list ⇒ tyi' 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 (e1 <bop> e2) =
  (let ST1 = ty E e1#ST; A1 = A ⊔ A e1 in
   compT E A ST e1 @ [after E A ST e1] @
   compT E A1 ST1 e2 @ [after E A1 ST1 e2])
| compT E A ST (Var i) = []
| compT E A ST (i := e) = compT E A ST e @
  [after E A ST e, ty_i' ST E (A ⊔ A e ⊔ [{i})]
| compT E A ST (e.F{D}) =
  compT E A ST e @ [after E A ST e]
| compT E A ST (e1.F{D} := e2) =
  (let ST1 = ty E e1#ST; A1 = A ⊔ A e1; A2 = A1 ⊔ A e2 in
   compT E A ST e1 @ [after E A ST e1] @
   compT E A1 ST1 e2 @ [after E A1 ST1 e2] @
   [ty_i' ST E A2])
| compT E A ST {i:T; e} = compT (E@[T]) (A⊖i) ST e
| compT E A ST (e1;;e2) =
  (let A1 = A ⊔ A e1 in
   compT E A ST e1 @ [after E A ST e1, ty_i' ST E A1] @
   compT E A1 ST e2)
| compT E A ST (if (e) e1 else e2) =
  (let A0 = A ⊔ A e; τ = ty_i' ST E A0 in
   compT E A ST e @ [after E A ST e, τ] @
   compT E A0 ST e1 @ [after E A0 ST e1, τ] @
   compT E A0 ST e2)
| compT E A ST (while (e) c) =
  (let A0 = A ⊔ A e; A1 = A0 ⊔ A c; τ = ty_i' ST E A0 in
   compT E A ST e @ [after E A ST e, τ] @
   compT E A0 ST c @ [after E A0 ST c, ty_i' ST E A1, ty_i' ST E A0])
| compT E A ST (throw e) = compT E A ST e @ [after E A ST e]
| compT E A ST (e.M(es)) =
  compT E A ST e @ [after E A ST e] @
  compTs E (A ⊔ A e) (ty E e # ST) es
| compT E A ST (try e1 catch(C i) e2) =
  compT E A ST e1 @ [after E A ST e1] @
  [ty_i' (Class C#ST) E A, ty_i' ST (E@[Class C]) (A ⊔ [{i})] @
   compT (E@[Class C]) (A ⊔ [{i}) ST e2
| compTs E A ST [] = []
| compTs E A ST (e#es) = compT E A ST e @ [after E A ST e] @
  compTs E (A ⊔ (A e)) (ty E e # ST) es

```

**definition**  $compT_a :: ty\ list \Rightarrow nat\ hyperset \Rightarrow ty\ list \Rightarrow expr_1 \Rightarrow ty_i'\ list$  **where**  
 $compT_a\ E\ A\ ST\ e = compT\ E\ A\ ST\ e\ @\ [after\ E\ A\ ST\ e]$

**end**

**lemma**  $compE_2\text{-not-Nil}[simp]: compE_2\ e \neq [] \langle proof \rangle$

**lemma** (in TC1)  $compT\text{-sizes}[simp]:$

**shows**  $\bigwedge E\ A\ ST. size(compT\ E\ A\ ST\ e) = size(compE_2\ e) - 1$

**and**  $\bigwedge E\ A\ ST. size(compTs\ E\ A\ ST\ es) = size(compEs_2\ es) \langle proof \rangle$

**lemma** (in TC1)  $[simp]: \bigwedge ST\ E. [\tau] \notin set\ (compT\ E\ None\ ST\ e)$

and [simp]:  $\bigwedge ST E. [\tau] \notin \text{set}(\text{compTs } E \text{ None } ST \text{ es})$ ⟨proof⟩

lemma (in TC0) pair-eq-ty<sub>i</sub>'-conv:

$([(ST, LT)] = \text{ty}_i' ST_0 E A) =$   
 $(\text{case } A \text{ of None} \Rightarrow \text{False} \mid \text{Some } A \Rightarrow (ST = ST_0 \wedge LT = \text{ty}_l E A))$ ⟨proof⟩

lemma (in TC0) pair-conv-ty<sub>i</sub>':

$[(ST, \text{ty}_l E A)] = \text{ty}_i' ST E [A]$ ⟨proof⟩

lemma (in TC1) compT-LT-prefix:

$\bigwedge E A ST_0. \llbracket [(ST, LT)] \in \text{set}(\text{compT } E A ST_0 e); \mathcal{B} e (\text{size } E) \rrbracket$   
 $\implies P \vdash [(ST, LT)] \leq' \text{ty}_i' ST E A$

and

$\bigwedge E A ST_0. \llbracket [(ST, LT)] \in \text{set}(\text{compTs } E A ST_0 \text{ es}); \mathcal{B} s \text{ es } (\text{size } E) \rrbracket$   
 $\implies P \vdash [(ST, LT)] \leq' \text{ty}_i' ST E A$ ⟨proof⟩

lemma [iff]:  $OK \text{ None} \in \text{states } P \text{ mxs mxl}$ ⟨proof⟩

lemma (in TC0) after-in-states:

$\llbracket \text{wf-prog } p P; P, E \vdash_1 e :: T; \text{set } E \subseteq \text{types } P; \text{set } ST \subseteq \text{types } P;$   
 $\text{size } ST + \text{max-stack } e \leq \text{mxs} \rrbracket$   
 $\implies OK (\text{after } E A ST e) \in \text{states } P \text{ mxs mxl}$ ⟨proof⟩

lemma (in TC0) OK-ty<sub>i</sub>'-in-statesI[simp]:

$\llbracket \text{set } E \subseteq \text{types } P; \text{set } ST \subseteq \text{types } P; \text{size } ST \leq \text{mxs} \rrbracket$   
 $\implies OK (\text{ty}_i' ST E A) \in \text{states } P \text{ mxs mxl}$ ⟨proof⟩

lemma is-class-type-aux:  $\text{is-class } P C \implies \text{is-type } P (\text{Class } C)$ ⟨proof⟩

theorem (in TC1) compT-states:

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

shows  $\bigwedge E T A ST.$

$\llbracket P, E \vdash_1 e :: T; \text{set } E \subseteq \text{types } P; \text{set } ST \subseteq \text{types } P;$   
 $\text{size } ST + \text{max-stack } e \leq \text{mxs}; \text{size } E + \text{max-vars } e \leq \text{mxl} \rrbracket$   
 $\implies OK ' \text{set}(\text{compT } E A ST e) \subseteq \text{states } P \text{ mxs mxl}$

and  $\bigwedge E Ts A ST.$

$\llbracket P, E \vdash_1 \text{es} :: Ts; \text{set } E \subseteq \text{types } P; \text{set } ST \subseteq \text{types } P;$   
 $\text{size } ST + \text{max-stacks } \text{es} \leq \text{mxs}; \text{size } E + \text{max-varss } \text{es} \leq \text{mxl} \rrbracket$   
 $\implies OK ' \text{set}(\text{compTs } E A ST \text{ es}) \subseteq \text{states } P \text{ mxs mxl}$ ⟨proof⟩

definition  $\text{shift} :: \text{nat} \Rightarrow \text{ex-table} \Rightarrow \text{ex-table}$

where

$\text{shift } n \text{ xt} \equiv \text{map } (\lambda(\text{from}, \text{to}, C, \text{handler}, \text{depth}). (\text{from} + n, \text{to} + n, C, \text{handler} + n, \text{depth})) \text{ xt}$

lemma [simp]:  $\text{shift } 0 \text{ xt} = \text{xt}$ ⟨proof⟩

lemma [simp]:  $\text{shift } n [] = []$ ⟨proof⟩

lemma [simp]:  $\text{shift } n (\text{xt}_1 @ \text{xt}_2) = \text{shift } n \text{xt}_1 @ \text{shift } n \text{xt}_2$ ⟨proof⟩

lemma [simp]:  $\text{shift } m (\text{shift } n \text{xt}) = \text{shift } (m+n) \text{xt}$ ⟨proof⟩

lemma [simp]:  $\text{pcs}(\text{shift } n \text{xt}) = \{\text{pc} + n \mid \text{pc}. \text{pc} \in \text{pcs } \text{xt}\}$ ⟨proof⟩

lemma shift-compxE<sub>2</sub>:

shows  $\bigwedge \text{pc } \text{pc}' d. \text{shift } \text{pc} (\text{compxE}_2 e \text{pc}' d) = \text{compxE}_2 e (\text{pc}' + \text{pc}) d$

and  $\bigwedge \text{pc } \text{pc}' d. \text{shift } \text{pc} (\text{compxEs}_2 \text{es } \text{pc}' d) = \text{compxEs}_2 \text{es} (\text{pc}' + \text{pc}) d$ ⟨proof⟩

lemma compxE<sub>2</sub>-size-convs[simp]:

shows  $n \neq 0 \implies \text{comp}xE_2 \ e \ n \ d = \text{shift } n \ (\text{comp}xE_2 \ e \ 0 \ d)$   
 and  $n \neq 0 \implies \text{comp}xEs_2 \ es \ n \ d = \text{shift } n \ (\text{comp}xEs_2 \ es \ 0 \ d)\langle \text{proof} \rangle$   
 locale  $TC2 = TC1 +$   
 fixes  $T_r :: ty$  and  $mxs :: pc$   
 begin

**definition**

$wt\text{-instrs} :: \text{instr list} \Rightarrow \text{ex-table} \Rightarrow ty_i' \text{ list} \Rightarrow \text{bool}$   
 $((\vdash -, - / [::] / -) [0,0,51] \ 50)$  **where**  
 $\vdash is, xt [::] \tau s \longleftrightarrow \text{size } is < \text{size } \tau s \wedge \text{pcs } xt \subseteq \{0..<\text{size } is\} \wedge$   
 $(\forall pc < \text{size } is. P, T_r, mxs, \text{size } \tau s, xt \vdash is!pc, pc :: \tau s)$

**end**

**notation**  $TC2.wt\text{-instrs} \ ((-, -, \vdash / -, - / [::] / -) [50,50,50,50,50,51] \ 50)$

**lemma** (in  $TC2$ )  $[simp]: \tau s \neq [] \implies \vdash [], [] [::] \tau s \langle \text{proof} \rangle$

**lemma**  $[simp]: \text{eff } i \ P \ pc \ \text{et } \text{None} = [] \langle \text{proof} \rangle$

**lemma**  $wt\text{-instr-appR}$ :

$\llbracket P, T, m, mpc, xt \vdash is!pc, pc :: \tau s;$   
 $pc < \text{size } is; \text{size } is < \text{size } \tau s; mpc \leq \text{size } \tau s; mpc \leq mpc' \rrbracket$   
 $\implies P, T, m, mpc', xt \vdash is!pc, pc :: \tau s @ \tau s' \langle \text{proof} \rangle$

**lemma**  $relevant\text{-entries-shift} [simp]:$

$relevant\text{-entries } P \ i \ (pc+n) \ (\text{shift } n \ xt) = \text{shift } n \ (relevant\text{-entries } P \ i \ pc \ xt) \langle \text{proof} \rangle$

**lemma**  $[simp]:$

$xcpt\text{-eff } i \ P \ (pc+n) \ \tau \ (\text{shift } n \ xt) =$   
 $\text{map } (\lambda(pc, \tau). (pc + n, \tau)) \ (xcpt\text{-eff } i \ P \ pc \ \tau \ xt) \langle \text{proof} \rangle$

**lemma**  $[simp]:$

$app_i \ (i, P, pc, m, T, \tau) \implies$   
 $\text{eff } i \ P \ (pc+n) \ (\text{shift } n \ xt) \ (\text{Some } \tau) =$   
 $\text{map } (\lambda(pc, \tau). (pc+n, \tau)) \ (\text{eff } i \ P \ pc \ xt \ (\text{Some } \tau)) \langle \text{proof} \rangle$

**lemma**  $[simp]:$

$xcpt\text{-app } i \ P \ (pc+n) \ mxs \ (\text{shift } n \ xt) \ \tau = xcpt\text{-app } i \ P \ pc \ mxs \ xt \ \tau \langle \text{proof} \rangle$

**lemma**  $wt\text{-instr-appL}$ :

$\llbracket P, T, m, mpc, xt \vdash i, pc :: \tau s; pc < \text{size } \tau s; mpc \leq \text{size } \tau s \rrbracket$   
 $\implies P, T, m, mpc + \text{size } \tau s', \text{shift } (\text{size } \tau s') \ xt \vdash i, pc + \text{size } \tau s' :: \tau s' @ \tau s \langle \text{proof} \rangle$

**lemma**  $wt\text{-instr-Cons}$ :

$\llbracket P, T, m, mpc - 1, [] \vdash i, pc - 1 :: \tau s;$   
 $0 < pc; 0 < mpc; pc < \text{size } \tau s + 1; mpc \leq \text{size } \tau s + 1 \rrbracket$   
 $\implies P, T, m, mpc, [] \vdash i, pc :: \tau \# \tau s \langle \text{proof} \rangle$

**lemma**  $wt\text{-instr-append}$ :

$\llbracket P, T, m, mpc - \text{size } \tau s', [] \vdash i, pc - \text{size } \tau s' :: \tau s;$   
 $\text{size } \tau s' \leq pc; \text{size } \tau s' \leq mpc; pc < \text{size } \tau s + \text{size } \tau s'; mpc \leq \text{size } \tau s + \text{size } \tau s' \rrbracket$   
 $\implies P, T, m, mpc, [] \vdash i, pc :: \tau s' @ \tau s \langle \text{proof} \rangle$

**lemma**  $xcpt\text{-app-pcs}$ :

$pc \notin \text{pcs } xt \implies xcpt\text{-app } i \ P \ pc \ mxs \ xt \ \tau \langle \text{proof} \rangle$

**lemma** *xcpt-eff-pcs*:

$$pc \notin pcs \, xt \implies xcpt\text{-}eff \, i \, P \, pc \, \tau \, xt = [] \langle proof \rangle$$

**lemma** *pcs-shift*:

$$pc < n \implies pc \notin pcs \, (shift \, n \, xt) \langle proof \rangle$$

**lemma** *wt-instr-appRx*:

$$\begin{aligned} & \llbracket P, T, m, mpc, xt \vdash is!pc, pc :: \tau s; pc < size \, is; size \, is < size \, \tau s; mpc \leq size \, \tau s \rrbracket \\ & \implies P, T, m, mpc, xt @ shift \, (size \, is) \, xt' \vdash is!pc, pc :: \tau s \langle proof \rangle \end{aligned}$$

**lemma** *wt-instr-appLx*:

$$\begin{aligned} & \llbracket P, T, m, mpc, xt \vdash i, pc :: \tau s; pc \notin pcs \, xt' \rrbracket \\ & \implies P, T, m, mpc, xt' @ xt \vdash i, pc :: \tau s \langle proof \rangle \end{aligned}$$

**lemma** (in *TC2*) *wt-instrs-extR*:

$$\vdash is, xt \, [::] \, \tau s \implies \vdash is, xt \, [::] \, \tau s @ \tau s' \langle proof \rangle$$

**lemma** (in *TC2*) *wt-instrs-ext*:

$$\begin{aligned} & \llbracket \vdash is_1, xt_1 \, [::] \, \tau s_1 @ \tau s_2; \vdash is_2, xt_2 \, [::] \, \tau s_2; size \, \tau s_1 = size \, is_1 \rrbracket \\ & \implies \vdash is_1 @ is_2, xt_1 @ shift \, (size \, is_1) \, xt_2 \, [::] \, \tau s_1 @ \tau s_2 \langle proof \rangle \end{aligned}$$

**corollary** (in *TC2*) *wt-instrs-ext2*:

$$\begin{aligned} & \llbracket \vdash is_2, xt_2 \, [::] \, \tau s_2; \vdash is_1, xt_1 \, [::] \, \tau s_1 @ \tau s_2; size \, \tau s_1 = size \, is_1 \rrbracket \\ & \implies \vdash is_1 @ is_2, xt_1 @ shift \, (size \, is_1) \, xt_2 \, [::] \, \tau s_1 @ \tau s_2 \langle proof \rangle \end{aligned}$$

**corollary** (in *TC2*) *wt-instrs-ext-prefix* [trans]:

$$\begin{aligned} & \llbracket \vdash is_1, xt_1 \, [::] \, \tau s_1 @ \tau s_2; \vdash is_2, xt_2 \, [::] \, \tau s_3; \\ & \quad size \, \tau s_1 = size \, is_1; prefixeq \, \tau s_3 \, \tau s_2 \rrbracket \\ & \implies \vdash is_1 @ is_2, xt_1 @ shift \, (size \, is_1) \, xt_2 \, [::] \, \tau s_1 @ \tau s_2 \langle proof \rangle \end{aligned}$$

**corollary** (in *TC2*) *wt-instrs-app*:

$$\begin{aligned} & \text{assumes } is_1: \vdash is_1, xt_1 \, [::] \, \tau s_1 @ [\tau] \\ & \text{assumes } is_2: \vdash is_2, xt_2 \, [::] \, \tau \# \tau s_2 \\ & \text{assumes } s: size \, \tau s_1 = size \, is_1 \\ & \text{shows } \vdash is_1 @ is_2, xt_1 @ shift \, (size \, is_1) \, xt_2 \, [::] \, \tau s_1 @ \tau \# \tau s_2 \langle proof \rangle \end{aligned}$$

**corollary** (in *TC2*) *wt-instrs-app-last* [trans]:

$$\begin{aligned} & \llbracket \vdash is_2, xt_2 \, [::] \, \tau \# \tau s_2; \vdash is_1, xt_1 \, [::] \, \tau s_1; \\ & \quad last \, \tau s_1 = \tau; size \, \tau s_1 = size \, is_1 + 1 \rrbracket \\ & \implies \vdash is_1 @ is_2, xt_1 @ shift \, (size \, is_1) \, xt_2 \, [::] \, \tau s_1 @ \tau s_2 \langle proof \rangle \end{aligned}$$

**corollary** (in *TC2*) *wt-instrs-append-last* [trans]:

$$\begin{aligned} & \llbracket \vdash is, xt \, [::] \, \tau s; P, T_r, mxs, mpc, [] \vdash i, pc :: \tau s; \\ & \quad pc = size \, is; mpc = size \, \tau s; size \, is + 1 < size \, \tau s \rrbracket \\ & \implies \vdash is @ [i], xt \, [::] \, \tau s \langle proof \rangle \end{aligned}$$

**corollary** (in *TC2*) *wt-instrs-app2*:

$$\begin{aligned} & \llbracket \vdash is_2, xt_2 \, [::] \, \tau' \# \tau s_2; \vdash is_1, xt_1 \, [::] \, \tau \# \tau s_1 @ [\tau']; \\ & \quad xt' = xt_1 @ shift \, (size \, is_1) \, xt_2; size \, \tau s_1 + 1 = size \, is_1 \rrbracket \\ & \implies \vdash is_1 @ is_2, xt' \, [::] \, \tau \# \tau s_1 @ \tau' \# \tau s_2 \langle proof \rangle \end{aligned}$$

**corollary** (in *TC2*) *wt-instrs-app2-simp* [trans, simp]:

$$\begin{aligned} & \llbracket \vdash is_2, xt_2 \, [::] \, \tau' \# \tau s_2; \vdash is_1, xt_1 \, [::] \, \tau \# \tau s_1 @ [\tau']; size \, \tau s_1 + 1 = size \, is_1 \rrbracket \\ & \implies \vdash is_1 @ is_2, xt_1 @ shift \, (size \, is_1) \, xt_2 \, [::] \, \tau \# \tau s_1 @ \tau' \# \tau s_2 \langle proof \rangle \end{aligned}$$



```

corollary (in TC2) wt-instrs-Cons[simp]:
  [[  $\tau s \neq []$ ;  $\vdash [i], [] [::] [\tau, \tau']$ ;  $\vdash is, xt [::] \tau' \# \tau s$  ]]
   $\implies \vdash i \# is, shift 1 xt [::] \tau \# \tau' \# \tau s$  <proof><proof><proof><proof><proof><proof><proof><proof><proof><proof><proof><proof><proof><proof><proof>
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.