

# Java Source and Bytecode Formalizations in Isabelle: Bali

Gerwin Klein      Tobias Nipkow      David von Oheimb      Leonor Prensa Nieto  
Norbert Schirmer      Martin Strecker

1st October 2005



# Contents

<b>1</b>	<b>Overview</b>	<b>7</b>
<b>2</b>	<b>Basis</b>	<b>11</b>
1	Definitions extending HOL as logical basis of Bali . . . . .	12
<b>3</b>	<b>Table</b>	<b>17</b>
2	Abstract tables and their implementation as lists . . . . .	18
<b>4</b>	<b>Name</b>	<b>25</b>
3	Java names . . . . .	26
<b>5</b>	<b>Value</b>	<b>29</b>
4	Java values . . . . .	30
<b>6</b>	<b>Type</b>	<b>31</b>
5	Java types . . . . .	32
<b>7</b>	<b>Term</b>	<b>33</b>
6	Java expressions and statements . . . . .	34
<b>8</b>	<b>Decl</b>	<b>43</b>
7	Field, method, interface, and class declarations, whole Java programs . . . .	44
8	Modifier . . . . .	44
9	Declaration (base "class" for member,interface and class declarations . . . .	46
10	Member (field or method) . . . . .	46
11	Field . . . . .	46
12	Method . . . . .	46
13	Interface . . . . .	48
14	Class . . . . .	49
<b>9</b>	<b>TypeRel</b>	<b>57</b>
15	The relations between Java types . . . . .	58
<b>10</b>	<b>DeclConcepts</b>	<b>67</b>
16	Advanced concepts on Java declarations like overriding, inheritance, dynamic method lookup . . . . .	68
17	accessibility of types (cf. 6.6.1) . . . . .	68
18	accessibility of members . . . . .	69
19	imethds . . . . .	89
20	accimethd . . . . .	90
21	methd . . . . .	90
22	accmethd . . . . .	91
23	dynmethd . . . . .	92

24	dynlookup . . . . .	94
25	fields . . . . .	94
26	accfield . . . . .	95
27	is methd . . . . .	95
<b>11</b>	<b>WellType</b>	<b>97</b>
28	Well-typedness of Java programs . . . . .	98
<b>12</b>	<b>DefiniteAssignment</b>	<b>109</b>
29	Definite Assignment . . . . .	110
30	Very restricted calculation fallback calculation . . . . .	111
31	Analysis of constant expressions . . . . .	113
32	Main analysis for boolean expressions . . . . .	114
33	Lifting set operations to range of tables (map to a set) . . . . .	115
<b>13</b>	<b>WellForm</b>	<b>125</b>
34	Well-formedness of Java programs . . . . .	126
35	accessibility concerns . . . . .	143
<b>14</b>	<b>State</b>	<b>147</b>
36	State for evaluation of Java expressions and statements . . . . .	148
37	access . . . . .	151
38	memory allocation . . . . .	152
39	initialization . . . . .	152
40	update . . . . .	152
41	update . . . . .	157
<b>15</b>	<b>Eval</b>	<b>163</b>
42	Operational evaluation (big-step) semantics of Java expressions and statements	164
<b>16</b>	<b>Example</b>	<b>181</b>
43	Example Bali program . . . . .	182
<b>17</b>	<b>Conform</b>	<b>199</b>
44	Conformance notions for the type soundness proof for Java . . . . .	200
<b>18</b>	<b>DefiniteAssignmentCorrect</b>	<b>209</b>
45	Correctness of Definite Assignment . . . . .	210
<b>19</b>	<b>TypeSafe</b>	<b>219</b>
46	The type soundness proof for Java . . . . .	220
47	accessibility . . . . .	228
48	Ideas for the future . . . . .	233
<b>20</b>	<b>Evaln</b>	<b>235</b>
49	Operational evaluation (big-step) semantics of Java expressions and statements	236
<b>21</b>	<b>Trans</b>	<b>243</b>
<b>22</b>	<b>AxSem</b>	<b>249</b>
50	Axiomatic semantics of Java expressions and statements (see also Eval.thy) .	250
51	peek-and . . . . .	251
52	assn-supd . . . . .	251
53	supd-assn . . . . .	251

54	subst-res . . . . .	252
55	subst-Bool . . . . .	252
56	peek-res . . . . .	252
57	ign-res . . . . .	253
58	peek-st . . . . .	253
59	ign-res-eq . . . . .	254
60	RefVar . . . . .	254
61	allocation . . . . .	254

## 23 AxSound 267

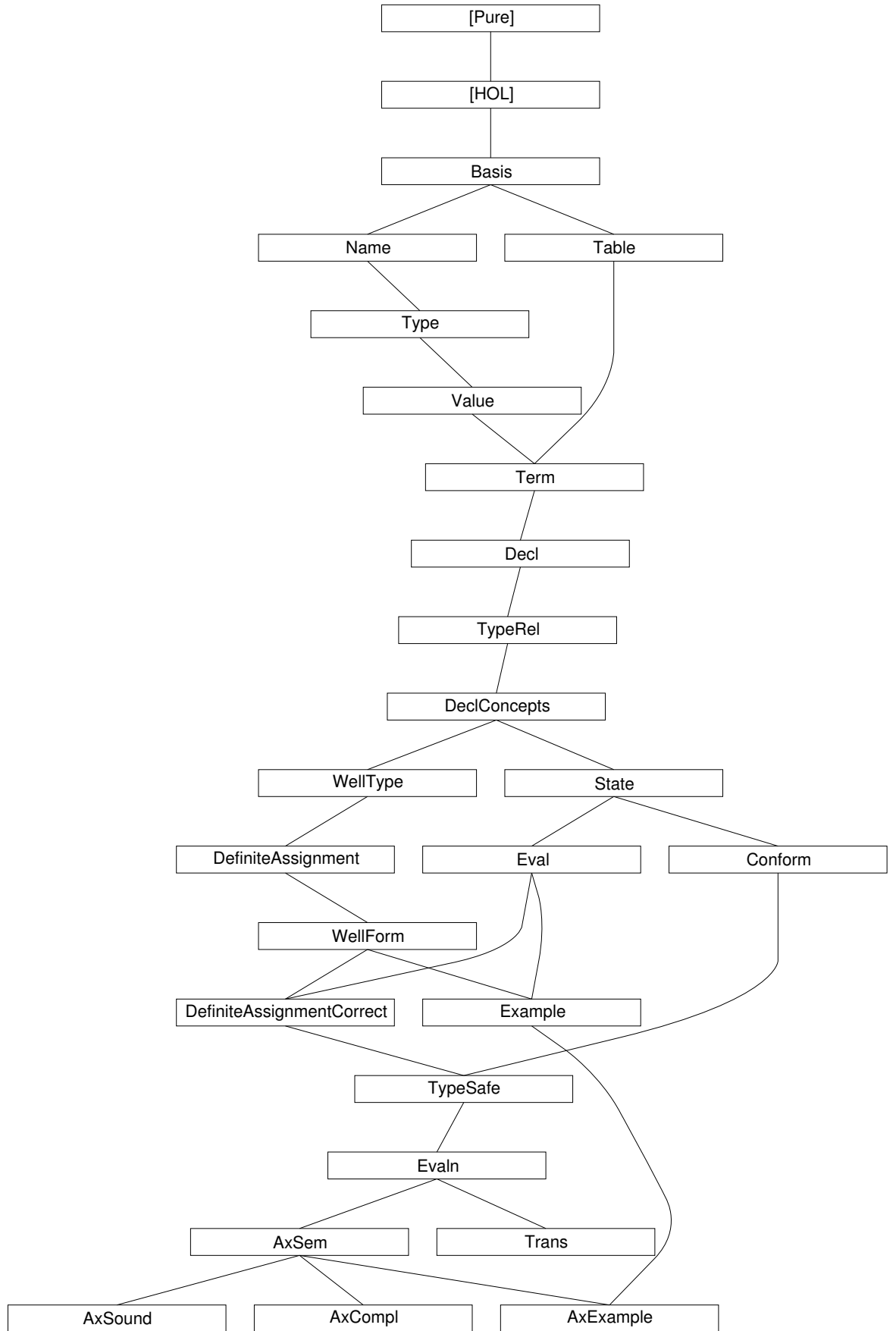
62	Soundness proof for Axiomatic semantics of Java expressions and statements	268
----	--	-----

## 24 AxCompl 273

63	Completeness proof for Axiomatic semantics of Java expressions and statements	274
----	---	-----

## 25 AxExample 281

64	Example of a proof based on the Bali axiomatic semantics . . . . .	282
----	--	-----



# Chapter 1

## Overview

These theories, called Bali, model and analyse different aspects of the JavaCard **source language**. The basis is an abstract model of the JavaCard source language. On it, a type system, an operational semantics and an axiomatic semantics (Hoare logic) are built. The execution of a wellformed program (with respect to the type system) according to the operational semantics is proved to be typesafe. The axiomatic semantics is proved to be sound and relative complete with respect to the operational semantics.

We have modelled large parts of the original JavaCard source language. It models features such as:

- The basic “primitive types” of Java
- Classes and related concepts
- Class fields and methods
- Instance fields and methods
- Interfaces and related concepts
- Arrays
- Static initialisation
- Static overloading of fields and methods
- Inheritance, overriding and hiding of methods, dynamic binding
- All cases of abrupt termination
  - Exception throwing and handling
  - `break`, `continue` and `return`
- Packages
- Access Modifiers (`private`, `protected`, `public`)
- A “definite assignment” check

The following features are missing in Bali wrt. JavaCard:

- Some primitive types (`byte`, `short`)
- Syntactic variants of statements (`do-loop`, `for-loop`)
- Interface fields

- Inner Classes

In addition, features are missing that are not part of the JavaCard language, such as multithreading and garbage collection. No attempt has been made to model peculiarities of JavaCard such as the applet firewall or the transaction mechanism.

Overview of the theories:

**Basis** Some basic definitions and settings not specific to JavaCard but missing in HOL.

**Table** Definition and some properties of a lookup table to map various names (like class names or method names) to some content (like classes or methods).

**Name** Definition of various names (class names, variable names, package names,...)

**Value** JavaCard expression values (Boolean, Integer, Addresses,...)

**Type** JavaCard types. Primitive types (Boolean, Integer,...) and reference types (Classes, Interfaces, Arrays,...)

**Term** JavaCard terms. Variables, expressions and statements.

**Decl** Class, interface and program declarations. Recursion operators for the class and the interface hierarchy.

**TypeRel** Various relations on types like the subclass-, subinterface-, widening-, narrowing- and casting-relation.

**DeclConcepts** Advanced concepts on the class and interface hierarchy like inheritance, overriding, hiding, accessibility of types and members according to the access modifiers, method lookup.

**WellType** Typesystem on the JavaCard term level.

**DefiniteAssignment** The definite assignment analysis on the JavaCard term level.

**WellForm** Typesystem on the JavaCard class, interface and program level.

**State** The program state (like object store) for the execution of JavaCard. Abrupt completion (exceptions, break, continue, return) is modelled as flag inside the state.

**Eval** Operational (big step) semantics for JavaCard.

**Example** An concrete example of a JavaCard program to validate the typesystem and the operational semantics.

**Conform** Conformance predicate for states. When does an execution state conform to the static types of the program given by the typesystem.

**DefiniteAssignmentCorrect** Correctness of the definite assignment analysis. If the analysis regards a variable as definitely assigned at a certain program point, the variable will actually be assigned there during execution.

**TypeSafe** Typesafety proof of the execution of JavaCard. "Welltyped programs don't go wrong" or more technical: The execution of a welltyped JavaCard program preserves the conformance of execution states.

**Evaln** Copy of the operational semantics given in theory Eval expanded with an annotation for the maximal recursive depth. The semantics is not altered. The annotation is needed for the soundness proof of the axiomatic semantics.



**Trans** A smallstep operational semantics for JavaCard.

**AxSem** An axiomatic semantics (Hoare logic) for JavaCard.

**AxSound** The soundness proof of the axiomatic semantics with respect to the operational semantics.

**AxComple** The proof of (relative) completeness of the axiomatic semantics with respect to the operational semantics.

**AxExample** An concrete example of the axiomatic semantics at work, applied to prove some properties of the JavaCard example given in theory Example.



## Chapter 2

# Basis

# 1 Definitions extending HOL as logical basis of Bali

theory *Basis* imports *Main* begin

$\langle ML \rangle$

**misc**

**declare** *same-fstI* [*intro!*]

$\langle ML \rangle$

**declare** *split-if-asm* [*split*] *option.split* [*split*] *option.split-asm* [*split*]

$\langle ML \rangle$

**declare** *if-weak-cong* [*cong del*] *option.weak-case-cong* [*cong del*]

**declare** *length-Suc-conv* [*iff*]

$\langle ML \rangle$

**lemma** *Collect-split-eq*:  $\{p. P (split\ f\ p)\} = \{(a,b). P (f\ a\ b)\}$

$\langle proof \rangle$

**lemma** *subset-insertD*:

$A \leq insert\ x\ B \implies A \leq B \ \& \ x \sim: A \mid (EX\ B'. A = insert\ x\ B' \ \& \ B' \leq B)$

$\langle proof \rangle$

**syntax**

$3 :: nat \quad (3)$

$4 :: nat \quad (4)$

**translations**

$3 == Suc\ 2$

$4 == Suc\ 3$

**lemma** *range-bool-domain*:  $range\ f = \{f\ True, f\ False\}$

$\langle proof \rangle$

**lemma** *irrefl-tranclI'*:  $r^{\wedge}-1\ Int\ r^{\wedge}+ = \{\} \implies !x. (x, x) \sim: r^{\wedge}+$

$\langle proof \rangle$

**lemma** *trancl-rtrancl-trancl*:

$\llbracket (x,y) \in r^{\wedge}+; (y,z) \in r^{\wedge}* \rrbracket \implies (x,z) \in r^{\wedge}+$

$\langle proof \rangle$

**lemma** *rtrancl-into-trancl3*:

$\llbracket (a,b) \in r^{\wedge}*; a \neq b \rrbracket \implies (a,b) \in r^{\wedge}+$

$\langle proof \rangle$

**lemma** *rtrancl-into-rtrancl2*:

$\llbracket (a, b) \in r; (b, c) \in r^* \rrbracket \implies (a, c) \in r^*$   
 $\langle \text{proof} \rangle$

**lemma** *triangle-lemma*:

$\llbracket \bigwedge a b c. \llbracket (a,b) \in r; (a,c) \in r \rrbracket \implies b=c; (a,x) \in r^*; (a,y) \in r^* \rrbracket$   
 $\implies (x,y) \in r^* \vee (y,x) \in r^*$   
 $\langle \text{proof} \rangle$

**lemma** *rtrancl-cases* [*consumes 1, case-names Refl Trancl*]:

$\llbracket (a,b) \in r^*; a = b \implies P; (a,b) \in r^+ \implies P \rrbracket \implies P$   
 $\langle \text{proof} \rangle$

**theorems** *converse-rtrancl-induct*

$= \text{converse-rtrancl-induct}$  [*consumes 1, case-names Id Step*]

**theorems** *converse-trancl-induct*

$= \text{converse-trancl-induct}$  [*consumes 1, case-names Single Step*]

**lemma** *Ball-weaken*:  $\llbracket \text{Ball } s \ P; \bigwedge x. P \ x \longrightarrow Q \ x \rrbracket \implies \text{Ball } s \ Q$

$\langle \text{proof} \rangle$

**lemma** *finite-SetCompr2*:  $\llbracket \text{finite } (\text{Collect } P); !y. P \ y \longrightarrow \text{finite } (\text{range } (f \ y)) \rrbracket \implies$   
 $\text{finite } \{f \ y \ x \mid x \ y. P \ y\}$

$\langle \text{proof} \rangle$

**lemma** *list-all2-trans*:  $\forall a \ b \ c. P1 \ a \ b \longrightarrow P2 \ b \ c \longrightarrow P3 \ a \ c \implies$

$\forall xs2 \ xs3. \text{list-all2 } P1 \ xs1 \ xs2 \longrightarrow \text{list-all2 } P2 \ xs2 \ xs3 \longrightarrow \text{list-all2 } P3 \ xs1 \ xs3$

$\langle \text{proof} \rangle$

**pairs**

**lemma** *surjective-pairing5*:  $p = (\text{fst } p, \text{fst } (\text{snd } p), \text{fst } (\text{snd } (\text{snd } p)), \text{fst } (\text{snd } (\text{snd } (\text{snd } p))),$   
 $\text{snd } (\text{snd } (\text{snd } (\text{snd } p))))$

$\langle \text{proof} \rangle$

**lemma** *fst-splitE* [*elim!*]:

$\llbracket \text{fst } s' = x'; !!x \ s. \llbracket s' = (x,s); x = x' \rrbracket \implies Q \rrbracket \implies Q$

$\langle \text{proof} \rangle$

**lemma** *fst-in-set-lemma* [*rule-format (no-asm)*]:  $(x, y) : \text{set } l \longrightarrow x : \text{fst } ' \text{set } l$

$\langle \text{proof} \rangle$

**quantifiers**

$\langle \text{ML} \rangle$

**lemma** *All-Ex-refl-eq2* [simp]:  
 $(!x. (? b. x = f b \ \& \ Q \ b) \longrightarrow P \ x) = (!b. Q \ b \longrightarrow P \ (f \ b))$   
 <proof>

**lemma** *ex-ex-miniscope1* [simp]:  
 $(EX \ w \ v. P \ w \ v \ \& \ Q \ v) = (EX \ v. (EX \ w. P \ w \ v) \ \& \ Q \ v)$   
 <proof>

**lemma** *ex-miniscope2* [simp]:  
 $(EX \ v. P \ v \ \& \ Q \ \& \ R \ v) = (Q \ \& \ (EX \ v. P \ v \ \& \ R \ v))$   
 <proof>

**lemma** *ex-reorder31*:  $(\exists z \ x \ y. P \ x \ y \ z) = (\exists x \ y \ z. P \ x \ y \ z)$   
 <proof>

**lemma** *All-Ex-refl-eq1* [simp]:  $(!x. (? b. x = f b) \longrightarrow P \ x) = (!b. P \ (f \ b))$   
 <proof>

## sums

**hide** *const In0 In1*

## syntax

*fun-sum* ::  $('a \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'c) \Rightarrow (('a + 'b) \Rightarrow 'c)$  (**infixr**  $'(+)'80$ )

## translations

*fun-sum* == *sum-case*

**consts**    *the-Inl* ::  $'a + 'b \Rightarrow 'a$   
             *the-Inr* ::  $'a + 'b \Rightarrow 'b$

**primrec** *the-Inl* (*Inl* *a*) = *a*

**primrec** *the-Inr* (*Inr* *b*) = *b*

**datatype**  $('a, 'b, 'c) \text{ sum3} = \text{In1 } 'a \mid \text{In2 } 'b \mid \text{In3 } 'c$

**consts**    *the-In1* ::  $('a, 'b, 'c) \text{ sum3} \Rightarrow 'a$   
             *the-In2* ::  $('a, 'b, 'c) \text{ sum3} \Rightarrow 'b$   
             *the-In3* ::  $('a, 'b, 'c) \text{ sum3} \Rightarrow 'c$

**primrec** *the-In1* (*In1* *a*) = *a*

**primrec** *the-In2* (*In2* *b*) = *b*

**primrec** *the-In3* (*In3* *c*) = *c*

## syntax

*In1l* ::  $'al \Rightarrow ('al + 'ar, 'b, 'c) \text{ sum3}$

*In1r* ::  $'ar \Rightarrow ('al + 'ar, 'b, 'c) \text{ sum3}$

## translations

*In1l* *e* == *In1* (*Inl* *e*)

*In1r* *c* == *In1* (*Inr* *c*)

**syntax** *the-In1l* ::  $('al + 'ar, 'b, 'c) \text{ sum3} \Rightarrow 'al$

*the-In1r* ::  $('al + 'ar, 'b, 'c) \text{ sum3} \Rightarrow 'ar$

## translations

*the-In1l* == *the-Inl*  $\circ$  *the-In1*

*the-In1r* == *the-Inr*  $\circ$  *the-In1*

$\langle ML \rangle$

### translations

$option \leq (type) \text{ Datatype.option}$   
 $list \leq (type) \text{ List.list}$   
 $sum3 \leq (type) \text{ Basis.sum3}$

### quantifiers for option type

#### syntax

$Oall :: [pttrn, 'a \text{ option}, bool] \Rightarrow bool \quad ((\exists! \text{ :-:} / -) [0,0,10] 10)$   
 $Oex :: [pttrn, 'a \text{ option}, bool] \Rightarrow bool \quad ((\exists? \text{ :-:} / -) [0,0,10] 10)$

#### syntax (symbols)

$Oall :: [pttrn, 'a \text{ option}, bool] \Rightarrow bool \quad ((\forall \text{ :-:} / -) [0,0,10] 10)$   
 $Oex :: [pttrn, 'a \text{ option}, bool] \Rightarrow bool \quad ((\exists \text{ :-:} / -) [0,0,10] 10)$

### translations

$! x:A: P \quad == \quad ! x:opts A. P$   
 $? x:A: P \quad == \quad ? x:opts A. P$

### unique association lists

#### constdefs

$unique :: ('a \times 'b) \text{ list} \Rightarrow bool$   
 $unique \equiv distinct \circ map \text{ fst}$

#### lemma uniqueD [rule-format (no-asm)]:

$unique \text{ l} \longrightarrow (!x \ y. (x,y):set \text{ l} \longrightarrow (!x' \ y'. (x',y'):set \text{ l} \longrightarrow x=x' \longrightarrow y=y'))$   
 $\langle proof \rangle$

#### lemma unique-Nil [simp]: unique []

$\langle proof \rangle$

#### lemma unique-Cons [simp]: unique ((x,y)#l) = (unique l & (!y. (x,y) ~: set l))

$\langle proof \rangle$

#### lemmas unique-ConsI = conjI [THEN unique-Cons [THEN iffD2], standard]

#### lemma unique-single [simp]: !!p. unique [p]

$\langle proof \rangle$

#### lemma unique-ConsD: unique (x#xs) ==> unique xs

$\langle proof \rangle$

#### lemma unique-append [rule-format (no-asm)]: unique l' ==> unique l -->

$(!(x,y):set \text{ l}. !(x',y'):set \text{ l}'. x' \sim x) \longrightarrow unique (\text{ l } @ \text{ l' })$   
 $\langle proof \rangle$

#### lemma unique-map-inj [rule-format (no-asm)]: unique l --> inj f --> unique (map (%(k,x). (f k, g k x)) l)

$\langle proof \rangle$

**lemma** *map-of-SomeI* [rule-format (no-asm)]: *unique*  $l \dashrightarrow (k, x) : set\ l \dashrightarrow map\ of\ l\ k = Some\ x$   
 $\langle proof \rangle$

## list patterns

### consts

*lsplit* :: [*'a*, *'a list*] => *'b*, *'a list*] => *'b*

### defs

*lsplit-def*: *lsplit* == %*f l. f (hd l) (tl l)*

### syntax

*-lpttrn* :: [*pttrn*, *pttrn*] => *pttrn* (-#/- [901,900] 900)

### translations

%*y#x#xs. b* == *lsplit* (%*y x#xs. b*)

%*x#xs . b* == *lsplit* (%*x xs . b*)

**lemma** *lsplit* [simp]: *lsplit* *c* (*x#xs*) = *c x xs*  
 $\langle proof \rangle$

**lemma** *lsplit2* [simp]: *lsplit* *P* (*x#xs*) *y z* = *P x xs y z*  
 $\langle proof \rangle$

## dummy pattern for quantifiers, let, etc.

### syntax

@*dummy-pat* :: *pttrn* ('-)

$\langle ML \rangle$

**end**



## Chapter 3

## Table

## 2 Abstract tables and their implementation as lists

**theory** *Table* **imports** *Basis* **begin**

design issues:

- definition of table: infinite map vs. list vs. finite set list chosen, because:
  - + a priori finite
  - + lookup is more operational than for finite set
    - not very abstract, but function table converts it to abstract mapping
- coding of lookup result: Some/None vs. value/arbitrary Some/None chosen, because:
  - ++ makes definedness check possible (applies also to finite set), which is important for the type standard, hiding/overriding, etc. (though it may perhaps be possible at least for the operational semantics to treat programs as infinite, i.e. where classes, fields, methods etc. of any name are considered to be defined)
  - sometimes awkward case distinctions, alleviated by operator 'the'

**types**  $('a, 'b)$  *table* — table with key type 'a and contents type 'b  
 $= 'a \multimap 'b$   
 $('a, 'b)$  *tables* — non-unique table with key 'a and contents 'b  
 $= 'a \Rightarrow 'b \text{ set}$

**map of / table of**

**syntax**

*table-of* ::  $('a \times 'b)$  *list*  $\Rightarrow ('a, 'b)$  *table* — concrete table

**translations**

*table-of* == *map-of*

$(type)'a \multimap 'b \leq (type)'a \Rightarrow 'b \text{ Option.option}$   
 $(type)('a, 'b) \text{ table} \leq (type)'a \multimap 'b$

**lemma** *map-add-find-left[simp]*:

$n \ k = \text{None} \implies (m \ ++ \ n) \ k = m \ k$

*<proof>*

### Conditional Override

**constdefs**

*cond-override*::

$('b \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a, 'b) \text{ table} \Rightarrow ('a, 'b) \text{ table} \Rightarrow ('a, 'b) \text{ table}$

— when merging tables old and new, only override an entry of table old when the condition cond holds

*cond-override cond old new*  $\equiv$

$\lambda k.$

$(\text{case new } k \text{ of}$   
 $\quad \text{None} \quad \Rightarrow \text{old } k$   
 $\quad | \text{Some new-val} \Rightarrow (\text{case old } k \text{ of}$   
 $\quad \quad \text{None} \quad \Rightarrow \text{Some new-val}$   
 $\quad \quad | \text{Some old-val} \Rightarrow (\text{if cond new-val old-val}$   
 $\quad \quad \quad \text{then Some new-val}$   
 $\quad \quad \quad \text{else Some old-val})))$

**lemma** *cond-override-empty1*[simp]: *cond-override c empty t = t*  
 ⟨proof⟩

**lemma** *cond-override-empty2*[simp]: *cond-override c t empty = t*  
 ⟨proof⟩

**lemma** *cond-override-None*[simp]:  
*old k = None  $\implies$  (cond-override c old new) k = new k*  
 ⟨proof⟩

**lemma** *cond-override-override*:  
 $\llbracket \text{old } k = \text{Some } ov; \text{new } k = \text{Some } nv; C \text{ nv } ov \rrbracket$   
 $\implies (\text{cond-override } C \text{ old new}) k = \text{Some } nv$   
 ⟨proof⟩

**lemma** *cond-override-noOverride*:  
 $\llbracket \text{old } k = \text{Some } ov; \text{new } k = \text{Some } nv; \neg (C \text{ nv } ov) \rrbracket$   
 $\implies (\text{cond-override } C \text{ old new}) k = \text{Some } ov$   
 ⟨proof⟩

**lemma** *dom-cond-override*: *dom (cond-override C s t)  $\subseteq$  dom s  $\cup$  dom t*  
 ⟨proof⟩

**lemma** *finite-dom-cond-override*:  
 $\llbracket \text{finite } (\text{dom } s); \text{finite } (\text{dom } t) \rrbracket \implies \text{finite } (\text{dom } (\text{cond-override } C \text{ s t}))$   
 ⟨proof⟩

## Filter on Tables

**constdefs**  
*filter-tab*:: ('a  $\Rightarrow$  'b  $\Rightarrow$  bool)  $\Rightarrow$  ('a, 'b) table  $\Rightarrow$  ('a, 'b) table  
*filter-tab c t*  $\equiv$   $\lambda k.$  (case t k of  
     None  $\Rightarrow$  None  
     | Some x  $\Rightarrow$  if c k x then Some x else None)

**lemma** *filter-tab-empty*[simp]: *filter-tab c empty = empty*  
 ⟨proof⟩

**lemma** *filter-tab-True*[simp]: *filter-tab ( $\lambda x y.$  True) t = t*  
 ⟨proof⟩

**lemma** *filter-tab-False*[simp]: *filter-tab ( $\lambda x y.$  False) t = empty*  
 ⟨proof⟩

**lemma** *filter-tab-ran-subset*: *ran (filter-tab c t)  $\subseteq$  ran t*  
 ⟨proof⟩

**lemma** *filter-tab-range-subset*:  $\text{range } (\text{filter-tab } c \ t) \subseteq \text{range } t \cup \{\text{None}\}$   
 ⟨proof⟩

**lemma** *finite-range-filter-tab*:  
 $\text{finite } (\text{range } t) \implies \text{finite } (\text{range } (\text{filter-tab } c \ t))$   
 ⟨proof⟩

**lemma** *filter-tab-SomeD[dest!]*:  
 $\text{filter-tab } c \ t \ k = \text{Some } x \implies (t \ k = \text{Some } x) \wedge c \ k \ x$   
 ⟨proof⟩

**lemma** *filter-tab-SomeI*:  $\llbracket t \ k = \text{Some } x; C \ k \ x \rrbracket \implies \text{filter-tab } C \ t \ k = \text{Some } x$   
 ⟨proof⟩

**lemma** *filter-tab-all-True*:  
 $\forall k \ y. \ t \ k = \text{Some } y \longrightarrow p \ k \ y \implies \text{filter-tab } p \ t = t$   
 ⟨proof⟩

**lemma** *filter-tab-all-True-Some*:  
 $\llbracket \forall k \ y. \ t \ k = \text{Some } y \longrightarrow p \ k \ y; t \ k = \text{Some } v \rrbracket \implies \text{filter-tab } p \ t \ k = \text{Some } v$   
 ⟨proof⟩

**lemma** *filter-tab-all-False*:  
 $\forall k \ y. \ t \ k = \text{Some } y \longrightarrow \neg p \ k \ y \implies \text{filter-tab } p \ t = \text{empty}$   
 ⟨proof⟩

**lemma** *filter-tab-None*:  $t \ k = \text{None} \implies \text{filter-tab } p \ t \ k = \text{None}$   
 ⟨proof⟩

**lemma** *filter-tab-dom-subset*:  $\text{dom } (\text{filter-tab } C \ t) \subseteq \text{dom } t$   
 ⟨proof⟩

**lemma** *filter-tab-eq*:  $\llbracket a=b \rrbracket \implies \text{filter-tab } C \ a = \text{filter-tab } C \ b$   
 ⟨proof⟩

**lemma** *finite-dom-filter-tab*:  
 $\text{finite } (\text{dom } t) \implies \text{finite } (\text{dom } (\text{filter-tab } C \ t))$   
 ⟨proof⟩

**lemma** *filter-tab-weaken*:  
 $\llbracket \forall a \in t \ k. \exists b \in s \ k. P \ a \ b; \bigwedge k \ x \ y. \llbracket t \ k = \text{Some } x; s \ k = \text{Some } y \rrbracket \implies \text{cond } k \ x \longrightarrow \text{cond } k \ y \rrbracket \implies \forall a \in \text{filter-tab } \text{cond } t \ k. \exists b \in \text{filter-tab } \text{cond } s \ k. P \ a \ b$   
 ⟨proof⟩

**lemma** *cond-override-filter*:

$$\begin{aligned} & \llbracket \bigwedge k \text{ old new. } \llbracket s \ k = \text{Some new}; t \ k = \text{Some old} \rrbracket \\ & \implies (\neg \text{overC new old} \longrightarrow \neg \text{filterC } k \text{ new}) \wedge \\ & \quad (\text{overC new old} \longrightarrow \text{filterC } k \text{ old} \longrightarrow \text{filterC } k \text{ new}) \\ & \rrbracket \implies \\ & \quad \text{cond-override overC (filter-tab filterC } t) \text{ (filter-tab filterC } s) \\ & \quad = \text{filter-tab filterC (cond-override overC } t \ s) \\ & \langle \text{proof} \rangle \end{aligned}$$

**Misc.**

**lemma** *Ball-set-table*:  $(\forall (x,y) \in \text{set } l. P \ x \ y) \implies \forall x. \forall y \in \text{map-of } l \ x: P \ x \ y$   
 $\langle \text{proof} \rangle$

**lemma** *Ball-set-tableD*:

$$\llbracket (\forall (x,y) \in \text{set } l. P \ x \ y); x \in \text{o2s (table-of } l \ x a) \rrbracket \implies P \ x a \ x$$
 $\langle \text{proof} \rangle$ 

**declare** *map-of-SomeD* [elim]

**lemma** *table-of-Some-in-set*:

$$\text{table-of } l \ k = \text{Some } x \implies (k, x) \in \text{set } l$$
 $\langle \text{proof} \rangle$ 

**lemma** *set-get-eq*:

$$\text{unique } l \implies (k, \text{the (table-of } l \ k)) \in \text{set } l = (\text{table-of } l \ k \neq \text{None})$$
 $\langle \text{proof} \rangle$ 

**lemma** *inj-Pair-const2*:  $\text{inj } (\lambda k. (k, C))$

$\langle \text{proof} \rangle$

**lemma** *table-of-mapconst-SomeI*:

$$\begin{aligned} & \llbracket \text{table-of } t \ k = \text{Some } y'; \text{snd } y=y'; \text{fst } y=c \rrbracket \implies \\ & \quad \text{table-of (map } (\lambda(k,x). (k,c,x)) \ t) \ k = \text{Some } y \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *table-of-mapconst-NoneI*:

$$\begin{aligned} & \llbracket \text{table-of } t \ k = \text{None} \rrbracket \implies \\ & \quad \text{table-of (map } (\lambda(k,x). (k,c,x)) \ t) \ k = \text{None} \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemmas** *table-of-map2-SomeI* = *inj-Pair-const2* [THEN *map-of-mapk-SomeI*, standard]

**lemma** *table-of-map-SomeI* [rule-format (no-asm)]:  $\text{table-of } t \ k = \text{Some } x \longrightarrow$

$$\text{table-of (map } (\lambda(k,x). (k, f \ x)) \ t) \ k = \text{Some (f } x)$$
 $\langle \text{proof} \rangle$ 

**lemma** *table-of-remap-SomeD* [rule-format (no-asm)]:

$$\begin{aligned} & \text{table-of (map } (\lambda((k,k'),x). (k,(k',x))) \ t) \ k = \text{Some (k',x)} \longrightarrow \\ & \text{table-of } t \ (k, k') = \text{Some } x \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *table-of-mapf-Some* [rule-format (no-asm)]:  $\forall x y. f x = f y \longrightarrow x = y \implies$   
*table-of* (map ( $\lambda(k,x). (k, f x)$ ) *t*) *k* = *Some* (*f x*)  $\longrightarrow$  *table-of* *t* *k* = *Some* *x*  
 <proof>

**lemma** *table-of-mapf-SomeD* [rule-format (no-asm), dest!]:  
*table-of* (map ( $\lambda(k,x). (k, f x)$ ) *t*) *k* = *Some* *z*  $\longrightarrow (\exists y \in \text{table-of } t \text{ } k: z = f y)$   
 <proof>

**lemma** *table-of-mapf-NoneD* [rule-format (no-asm), dest!]:  
*table-of* (map ( $\lambda(k,x). (k, f x)$ ) *t*) *k* = *None*  $\longrightarrow (\text{table-of } t \text{ } k = \text{None})$   
 <proof>

**lemma** *table-of-mapkey-SomeD* [rule-format (no-asm), dest!]:  
*table-of* (map ( $\lambda(k,x). ((k,C),x)$ ) *t*) (*k,D*) = *Some* *x*  $\longrightarrow C = D \wedge \text{table-of } t \text{ } k = \text{Some } x$   
 <proof>

**lemma** *table-of-mapkey-SomeD2* [rule-format (no-asm), dest!]:  
*table-of* (map ( $\lambda(k,x). ((k,C),x)$ ) *t*) *ek* = *Some* *x*  
 $\longrightarrow C = \text{snd } ek \wedge \text{table-of } t \text{ } (\text{fst } ek) = \text{Some } x$   
 <proof>

**lemma** *table-append-Some-iff*: *table-of* (*xs@ys*) *k* = *Some* *z* =  
 (*table-of* *xs* *k* = *Some* *z*  $\vee$  (*table-of* *xs* *k* = *None*  $\wedge$  *table-of* *ys* *k* = *Some* *z*))  
 <proof>

**lemma** *table-of-filter-unique-SomeD* [rule-format (no-asm)]:  
*table-of* (filter *P xs*) *k* = *Some* *z*  $\implies \text{unique } xs \longrightarrow \text{table-of } xs \text{ } k = \text{Some } z$   
 <proof>

### consts

*Un-tables* :: ('a, 'b) tables set  $\Rightarrow$  ('a, 'b) tables  
*overrides-t* :: ('a, 'b) tables  $\Rightarrow$  ('a, 'b) tables  $\Rightarrow$   
 ('a, 'b) tables (infixl  $\oplus \oplus$  100)  
*hidings-entails*:: ('a, 'b) tables  $\Rightarrow$  ('a, 'c) tables  $\Rightarrow$   
 ('b  $\Rightarrow$  'c  $\Rightarrow$  bool)  $\Rightarrow$  bool (- hidings - entails - 20)  
 — variant for unique table:  
*hiding-entails* :: ('a, 'b) table  $\Rightarrow$  ('a, 'c) table  $\Rightarrow$   
 ('b  $\Rightarrow$  'c  $\Rightarrow$  bool)  $\Rightarrow$  bool (- hiding - entails - 20)  
 — variant for a unique table and conditional overriding:  
*cond-hiding-entails* :: ('a, 'b) table  $\Rightarrow$  ('a, 'c) table  
 $\Rightarrow$  ('b  $\Rightarrow$  'c  $\Rightarrow$  bool)  $\Rightarrow$  ('b  $\Rightarrow$  'c  $\Rightarrow$  bool)  $\Rightarrow$  bool  
 (- hiding - under - entails - 20)

### defs

*Un-tables-def*: *Un-tables* *ts*  $\equiv \lambda k. \bigcup t \in ts. t \text{ } k$   
*overrides-t-def*: *s*  $\oplus \oplus$  *t*  $\equiv \lambda k. \text{if } t \text{ } k = \{\} \text{ then } s \text{ } k \text{ else } t \text{ } k$   
*hidings-entails-def*: *t* *hidings* *s* *entails* *R*  $\equiv \forall k. \forall x \in t \text{ } k. \forall y \in s \text{ } k. R \text{ } x \text{ } y$   
*hiding-entails-def*: *t* *hiding* *s* *entails* *R*  $\equiv \forall k. \forall x \in t \text{ } k: \forall y \in s \text{ } k: R \text{ } x \text{ } y$   
*cond-hiding-entails-def*: *t* *hiding* *s* *under* *C* *entails* *R*  
 $\equiv \forall k. \forall x \in t \text{ } k: \forall y \in s \text{ } k: C \text{ } x \text{ } y \longrightarrow R \text{ } x \text{ } y$

**Untables**

**lemma** *Un-tablesI* [intro]:  $\bigwedge x. \llbracket t \in ts; x \in t \ k \rrbracket \implies x \in \text{Un-tables } ts \ k$   
 ⟨proof⟩

**lemma** *Un-tablesD* [dest!]:  $\bigwedge x. x \in \text{Un-tables } ts \ k \implies \exists t. t \in ts \wedge x \in t \ k$   
 ⟨proof⟩

**lemma** *Un-tables-empty* [simp]:  $\text{Un-tables } \{\} = (\lambda k. \{\})$   
 ⟨proof⟩

**overrides**

**lemma** *empty-overrides-t* [simp]:  $(\lambda k. \{\}) \oplus \oplus m = m$   
 ⟨proof⟩

**lemma** *overrides-empty-t* [simp]:  $m \oplus \oplus (\lambda k. \{\}) = m$   
 ⟨proof⟩

**lemma** *overrides-t-Some-iff*:  
 $(x \in (s \oplus \oplus t) \ k) = (x \in t \ k \vee t \ k = \{\} \wedge x \in s \ k)$   
 ⟨proof⟩

**lemmas** *overrides-t-SomeD* = *overrides-t-Some-iff* [THEN iffD1, dest!]

**lemma** *overrides-t-right-empty* [simp]:  $n \ k = \{\} \implies (m \oplus \oplus n) \ k = m \ k$   
 ⟨proof⟩

**lemma** *overrides-t-find-right* [simp]:  $n \ k \neq \{\} \implies (m \oplus \oplus n) \ k = n \ k$   
 ⟨proof⟩

**hiding entails**

**lemma** *hiding-entailsD*:  
 $\llbracket t \text{ hiding } s \text{ entails } R; t \ k = \text{Some } x; s \ k = \text{Some } y \rrbracket \implies R \ x \ y$   
 ⟨proof⟩

**lemma** *empty-hiding-entails*: *empty hiding s entails R*  
 ⟨proof⟩

**lemma** *hiding-empty-entails*: *t hiding empty entails R*  
 ⟨proof⟩

**declare** *empty-hiding-entails* [simp] *hiding-empty-entails* [simp]

**cond hiding entails**

**lemma** *cond-hiding-entailsD*:  
 $\llbracket t \text{ hiding } s \text{ under } C \text{ entails } R; t \ k = \text{Some } x; s \ k = \text{Some } y; C \ x \ y \rrbracket \implies R \ x \ y$   
 ⟨proof⟩

**lemma** *empty-cond-hiding-entails*[simp]: *empty hiding s under C entails R*  
 ⟨proof⟩

**lemma** *cond-hiding-empty-entails*[simp]: *t hiding empty under C entails R*  
 ⟨proof⟩

**lemma** *hidings-entailsD*:  $\llbracket t \text{ hidings } s \text{ entails } R; x \in t \ k; y \in s \ k \rrbracket \implies R \ x \ y$   
 ⟨proof⟩

**lemma** *hidings-empty-entails*: *t hidings* ( $\lambda k. \{\}$ ) *entails R*  
 ⟨proof⟩

**lemma** *empty-hidings-entails*:  
 ( $\lambda k. \{\}$ ) *hidings s entails R*⟨proof⟩  
**declare** *empty-hidings-entails* [intro!] *hidings-empty-entails* [intro!]

**consts**  
*atleast-free* :: ('a  $\rightsquigarrow$  'b)  $\implies$  nat  $\implies$  bool  
**primrec**  
*atleast-free* *m* 0 = True  
*atleast-free-Suc*:  
*atleast-free* *m* (Suc *n*) = (? *a*. *m* *a* = None & (!*b*. *atleast-free* (*m*(*a*| $\rightarrow$ *b*)) *n*))

**lemma** *atleast-free-weaken* [rule-format (no-asm)]:  
 !*m*. *atleast-free* *m* (Suc *n*)  $\longrightarrow$  *atleast-free* *m* *n*  
 ⟨proof⟩

**lemma** *atleast-free-SucI*:  
 [| *h* *a* = None; !*obj*. *atleast-free* (*h*(*a*| $\rightarrow$ *obj*)) *n* |]  $\implies$  *atleast-free* *h* (Suc *n*)  
 ⟨proof⟩

**declare** *fun-upd-apply* [simp del]

**lemma** *atleast-free-SucD-lemma* [rule-format (no-asm)]:  
 !*m* *a*. *m* *a* = None  $\longrightarrow$  (!*c*. *atleast-free* (*m*(*a*| $\rightarrow$ *c*)) *n*)  $\longrightarrow$   
 (!*b* *d*. *a*  $\rightsquigarrow$  *b*  $\longrightarrow$  *atleast-free* (*m*(*b*| $\rightarrow$ *d*)) *n*)  
 ⟨proof⟩  
**declare** *fun-upd-apply* [simp]

**lemma** *atleast-free-SucD* [rule-format (no-asm)]: *atleast-free* *h* (Suc *n*)  $\implies$  *atleast-free* (*h*(*a*| $\rightarrow$ *b*)) *n*  
 ⟨proof⟩

**declare** *atleast-free-Suc* [simp del]  
**end**



## Chapter 4

Name

### 3 Java names

**theory** *Name* **imports** *Basis* **begin**

**typeddecl** *tnam* — ordinary type name, i.e. class or interface name

**typeddecl** *pname* — package name

**typeddecl** *mname* — method name

**typeddecl** *vname* — variable or field name

**typeddecl** *label* — label as destination of break or continue

**datatype** *ename* — expression name

= *VName vname*

| *Res* — special name to model the return value of methods

**datatype** *lname* — names for local variables and the This pointer

= *ENAME ename*

| *This*

**syntax**

*VName* :: *vname*  $\Rightarrow$  *lname*

*Result* :: *lname*

**translations**

*VName* *n* == *ENAME* (*VName* *n*)

*Result* == *ENAME* *Res*

**datatype** *xname* — names of standard exceptions

= *Throwable*

| *NullPointerException* | *OutOfMemory* | *ClassCast*

| *NegativeArraySize* | *IndexOutOfBoundsException* | *ArrayStore*

**lemma** *xn-cases*:

*xn* = *Throwable*  $\vee$  *xn* = *NullPointerException*  $\vee$

*xn* = *OutOfMemory*  $\vee$  *xn* = *ClassCast*  $\vee$

*xn* = *NegativeArraySize*  $\vee$  *xn* = *IndexOutOfBoundsException*  $\vee$  *xn* = *ArrayStore*

$\langle$ proof $\rangle$

**datatype** *tname* — type names for standard classes and other type names

= *Object-*

| *SXcpt-* *xname*

| *TName* *tnam*

**record** *qname* = — qualified tname cf. 6.5.3, 6.5.4

*pid* :: *pname*

*tid* :: *tname*

**axclass** *has-pname* < *type*

**consts** *pname*::'*a*::*has-pname*  $\Rightarrow$  *pname*

**instance** *pname*::*has-pname*  $\langle$ proof $\rangle$

**defs** (overloaded)

*pname-pname-def*: *pname* (*p*::*pname*)  $\equiv$  *p*

**axclass** *has-tname* < *type*

**consts** *tname*::'*a*::*has-tname*  $\Rightarrow$  *tname*

**instance** *tname::has-tname*  $\langle \text{proof} \rangle$

**defs** (overloaded)

*tname-tname-def*: *tname* (*t::tname*)  $\equiv t$

**axclass** *has-qtname* < *type*

**consts** *qtname::'a::has-qtname*  $\Rightarrow$  *qtname*

**instance** *qtname-ext-type* :: (*type*) *has-qtname*  $\langle \text{proof} \rangle$

**defs** (overloaded)

*qtname-qtname-def*: *qtname* (*q::qtname*)  $\equiv q$

**translations**

*mname* <= *Name.mname*

*xname* <= *Name.xname*

*tname* <= *Name.tname*

*ename* <= *Name.ename*

*qtname* <= (*type*) ( $\lfloor \text{pid}::\text{pname}, \text{tid}::\text{tname} \rfloor$ )

(*type*) '*a* *qtname-scheme* <= (*type*) ( $\lfloor \text{pid}::\text{pname}, \text{tid}::\text{tname}, \dots::'a \rfloor$ )

**consts** *java-lang::pname* — package *java.lang*

**consts**

*Object* :: *qtname*

*SXcpt* :: *xname*  $\Rightarrow$  *qtname*

**defs**

*Object-def*: *Object*  $\equiv \lfloor \text{pid} = \text{java-lang}, \text{tid} = \text{Object} \rfloor$

*SXcpt-def*: *SXcpt*  $\equiv \lambda x. \lfloor \text{pid} = \text{java-lang}, \text{tid} = \text{SXcpt- } x \rfloor$

**lemma** *Object-neq-SXcpt* [*simp*]: *Object*  $\neq$  *SXcpt xn*

$\langle \text{proof} \rangle$

**lemma** *SXcpt-inject* [*simp*]: (*SXcpt xn* = *SXcpt xm*) = (*xn* = *xm*)

$\langle \text{proof} \rangle$

**end**



## Chapter 5

# Value

## 4 Java values

**theory** *Value* **imports** *Type* **begin**

**typeddecl** *loc* — locations, i.e. abstract references on objects

**datatype** *val*

= *Unit* — dummy result value of void methods  
 | *Bool bool* — Boolean value  
 | *Intg int* — integer value  
 | *Null* — null reference  
 | *Addr loc* — addresses, i.e. locations of objects

**translations** *val* <= (*type*) *Term.val*  
                   *loc* <= (*type*) *Term.loc*

**consts** *the-Bool* :: *val* ⇒ *bool*

**primrec** *the-Bool* (*Bool b*) = *b*

**consts** *the-Intg* :: *val* ⇒ *int*

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

**consts** *the-Addr* :: *val* ⇒ *loc*

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

**types** *dyn-ty* = *loc* ⇒ *ty option*

**consts**

*typeof* :: *dyn-ty* ⇒ *val* ⇒ *ty option*

*defpval* :: *prim-ty* ⇒ *val* — default value for primitive types

*default-val* :: *ty* ⇒ *val* — default value for all types

**primrec** *typeof dt Unit* = *Some (PrimT Void)*

*typeof dt (Bool b)* = *Some (PrimT Boolean)*

*typeof dt (Intg i)* = *Some (PrimT Integer)*

*typeof dt Null* = *Some NT*

*typeof dt (Addr a)* = *dt a*

**primrec** *defpval Void* = *Unit*

*defpval Boolean* = *Bool False*

*defpval Integer* = *Intg 0*

**primrec** *default-val (PrimT pt)* = *defpval pt*

*default-val (RefT r)* = *Null*

**end**

## Chapter 6

# Type

## 5 Java types

**theory** *Type* **imports** *Name* **begin**

simplifications:

- only the most important primitive types
- the null type is regarded as reference type

**datatype** *prim-ty* — primitive type, cf. 4.2  
 = *Void* — result type of void methods  
 | *Boolean*  
 | *Integer*

**datatype** *ref-ty* — reference type, cf. 4.3  
 = *NullT* — null type, cf. 4.1  
 | *IfaceT qtname* — interface type  
 | *ClassT qtname* — class type  
 | *ArrayT ty* — array type

**and** *ty* — any type, cf. 4.1  
 = *PrimT prim-ty* — primitive type  
 | *RefT ref-ty* — reference type

**translations**

*prim-ty* <= (*type*) *Type.prim-ty*  
*ref-ty* <= (*type*) *Type.ref-ty*  
*ty* <= (*type*) *Type.ty*

**syntax**

*NT* :: *ty*  
*Iface* :: *qtname*  $\Rightarrow$  *ty*  
*Class* :: *qtname*  $\Rightarrow$  *ty*  
*Array* :: *ty*  $\Rightarrow$  *ty* ( $\cdot$ .[] [90] 90)

**translations**

*NT* == *RefT NullT*  
*Iface I* == *RefT (IfaceT I)*  
*Class C* == *RefT (ClassT C)*  
*T.[]* == *RefT (ArrayT T)*

**constdefs**

*the-Class* :: *ty*  $\Rightarrow$  *qtname*  
*the-Class T*  $\equiv$  *SOME C. T = Class C*

**lemma** *the-Class-eq [simp]: the-Class (Class C) = C*  
 <proof>

**end**



## Chapter 7

### Term

## 6 Java expressions and statements

**theory** *Term* **imports** *Value Table* **begin**

design issues:

- invocation frames for local variables could be reduced to special static objects (one per method). This would reduce redundancy, but yield a rather non-standard execution model more difficult to understand.
  - method bodies separated from calls to handle assumptions in axiomat. semantics NB: Body is intended to be in the environment of the called method.
  - class initialization is regarded as (auxiliary) statement (required for AxSem)
  - result expression of method return is handled by a special result variable result variable is treated uniformly with local variables
- + welltypedness and existence of the result/return expression is ensured without extra effort

simplifications:

- expression statement allowed for any expression
- This is modeled as a special non-assignable local variable
- Super is modeled as a general expression with the same value as This
- access to field x in current class via This.x
- NewA creates only one-dimensional arrays; initialization of further subarrays may be simulated with nested NewAs
- The 'Lit' constructor is allowed to contain a reference value. But this is assumed to be prohibited in the input language, which is enforced by the type-checking rules.
- a call of a static method via a type name may be simulated by a dummy variable
- no nested blocks with inner local variables
- no synchronized statements
- no secondary forms of if, while (e.g. no for) (may be easily simulated)
- no switch (may be simulated with if)
- the *try-catch-finally* statement is divided into the *try-catch* statement and a finally statement, which may be considered as try..finally with empty catch
- the *try-catch* statement has exactly one catch clause; multiple ones can be simulated with instanceof
- the compiler is supposed to add the annotations - during type-checking. This transformation is left out as its result is checked by the type rules anyway

**types** *locals* = (*lname*, *val*) *table* — local variables

**datatype** *jump*  
= *Break label* — break

| *Cont label* — continue  
 | *Ret* — return from method

**datatype** *xcpt* — exception  
 = *Loc loc* — location of allocated exception object  
 | *Std xname* — intermediate standard exception, see Eval.thy

**datatype** *error*  
 = *AccessViolation* — Access to a member that isn't permitted  
 | *CrossMethodJump* — Method exits with a break or continue

**datatype** *abrupt* — abrupt completion  
 = *Xcpt xcpt* — exception  
 | *Jump jump* — break, continue, return  
 | *Error error* — runtime errors, we wan't to detect and proof absent in welltyped programmms

**types**  
*abopt* = *abrupt option*

Local variable store and exception. Anticipation of State.thy used by smallstep semantics. For a method call, we save the local variables of the caller in the term Callee to restore them after method return. Also an exception must be restored after the finally statement

**translations**  
*locals* <= (*type*) (*lname, val*) *table*

**datatype** *inv-mode* — invocation mode for method calls  
 = *Static* — static  
 | *SuperM* — super  
 | *IntVir* — interface or virtual

**record** *sig* = — signature of a method, cf. 8.4.2  
*name* :: *mname* — acutally belongs to Decl.thy  
*parTs* :: *ty list*

**translations**  
*sig* <= (*type*) (*name* :: *mname*, *parTs* :: *ty list*)  
*sig* <= (*type*) (*name* :: *mname*, *parTs* :: *ty list*, . . . : 'a)

— function codes for unary operations

**datatype** *unop* = *UPlus* — + unary plus  
 | *UMinus* — - unary minus  
 | *UBitNot* — bitwise NOT  
 | *UNot* — ! logical complement

— function codes for binary operations

**datatype** *binop* = *Mul* — \* multiplication  
 | *Div* — / division  
 | *Mod* — % remainder  
 | *Plus* — + addition  
 | *Minus* — - subtraction  
 | *LShift* — << left shift  
 | *RShift* — >> signed right shift  
 | *RShiftU* — >>> unsigned right shift  
 | *Less* — < less than  
 | *Le* — <= less than or equal  
 | *Greater* — > greater than  
 | *Ge* — >= greater than or equal  
 | *Eq* — == equal  
 | *Neq* — != not equal

```

| BitAnd — & bitwise AND
| And — & boolean AND
| BitXor — ^ bitwise Xor
| Xor — ^ boolean Xor
| BitOr — | bitwise Or
| Or — | boolean Or
| CondAnd — && conditional And
| CondOr — || conditional Or

```

The boolean operators & and | strictly evaluate both of their arguments. The conditional operators && and || only evaluate the second argument if the value of the whole expression isn't already determined by the first argument. e.g.: `false && e e` is not evaluated; `true || e e` is not evaluated;

#### **datatype** *var*

```

= LVar lname — local variable (incl. parameters)
| FVar qname qname bool expr vname ({-, -, -}--[10,10,10,85,99]90)
    — class field
    — {accC, statDeclC, stat}e..fn
    — accC: accessing class (static class were
    — the code is declared. Annotation only needed for
    — evaluation to check accessibility)
    — statDeclC: static declaration class of field
    — stat: static or instance field?
    — e: reference to object
    — fn: field name
| AVar expr expr (-.[-][90,10 ]90)
    — array component
    — e1..e2: e1 array reference; e2 index
| InsInitV stmt var
    — insertion of initialization before evaluation
    — of var (technical term for smallstep semantics.)

```

#### **and** *expr*

```

= NewC qname — class instance creation
| NewA ty expr (New -.[-][99,10 ]85)
    — array creation
| Cast ty expr — type cast
| Inst expr ref-ty (- InstOf -.[85,99] 85)
    — instanceof
| Lit val — literal value, references not allowed
| UnOp unop expr — unary operation
| BinOp binop expr expr — binary operation

| Super — special Super keyword
| Acc var — variable access
| Ass var expr (-:= - [90,85 ]85)
    — variable assign
| Cond expr expr expr (- ? - : - [85,85,80]80) — conditional
| Call qname ref-ty inv-mode expr mname (ty list) (expr list)
    ({-, -, -}---'({-}'')[10,10,10,85,99,10,10]85)
    — method call
    — {accC, statT, mode}e..mn( {pTs}args) "
    — accC: accessing class (static class were
    — the call code is declared. Annotation only needed for
    — evaluation to check accessibility)
    — statT: static declaration class/interface of
    — method
    — mode: invocation mode
    — e: reference to object

```

— *mn*: field name  
 — *pTs*: types of parameters  
 — *args*: the actual parameters/arguments  
 | *Methd qname sig* — (folded) method (see below)  
 | *Body qname stmt* — (unfolded) method body  
 | *InsInitE stmt expr*  
   — insertion of initialization before  
   — evaluation of *expr* (technical term for smallstep sem.)  
 | *Callee locals expr* — save callers locals in callee-Frame  
   — (technical term for smallstep semantics)  
**and** *stmt*  
 = *Skip* — empty statement  
 | *Expr expr* — expression statement  
 | *Lab jump stmt* ( $\cdot - [99,66]66$ )  
   — labeled statement; handles break  
 | *Comp stmt stmt* ( $\cdot - [66,65]65$ )  
 | *If- expr stmt stmt* (*If'*( $\cdot$ ) - *Else* -  $[80,79,79]70$ )  
 | *Loop label expr stmt* ( $\cdot - \text{While}'(\cdot) - [99,80,79]70$ )  
 | *Jmp jump* — break, continue, return  
 | *Throw expr*  
 | *TryC stmt qname vname stmt* (*Try* - *Catch'*( $\cdot -$ ) -  $[79,99,80,79]70$ )  
   — *Try c1 Catch(C vn) c2*  
   — *c1*: block where exception may be thrown  
   — *C*: exception class to catch  
   — *vn*: local name for exception used in *c2*  
   — *c2*: block to execute when exception is caught  
 | *Fin stmt stmt* ( $\cdot - \text{Finally} - [79,79]70$ )  
 | *FinA abrupt stmt* — Save abrupt of first statement  
   — technical term for smallstep sem.)  
 | *Init qname* — class initialization

The expressions *Methd* and *Body* are artificial program constructs, in the sense that they are not used to define a concrete Bali program. In the operational semantics they are "generated on the fly" to decompose the task to define the behaviour of the *Call* expression. They are crucial for the axiomatic semantics to give a syntactic hook to insert some assertions (cf. *AxSem.thy*, *Eval.thy*). The *Init* statement (to initialize a class on its first use) is inserted in various places by the semantics. *Callee*, *InsInitV*, *InsInitE*, *FinA* are only needed as intermediate steps in the smallstep (transition) semantics (cf. *Trans.thy*). *Callee* is used to save the local variables of the caller for method return. So we avoid modelling a frame stack. The *InsInitV/E* terms are only used by the smallstep semantics to model the intermediate steps of class-initialisation.

**types** *term* = (*expr+stmt,var,expr list*) *sum3*

**translations**

*sig* <= (*type*) *mname*  $\times$  *ty list*  
*var* <= (*type*) *Term.var*  
*expr* <= (*type*) *Term.expr*  
*stmt* <= (*type*) *Term.stmt*  
*term* <= (*type*) (*expr+stmt,var,expr list*) *sum3*

**syntax**

*this* :: *expr*  
*LAcc* :: *vname*  $\Rightarrow$  *expr* (!)  
*LAss* :: *vname*  $\Rightarrow$  *expr*  $\Rightarrow$  *stmt* ( $\cdot - ::= - [90,85] 85$ )  
*Return* :: *expr*  $\Rightarrow$  *stmt*  
*StatRef* :: *ref-ty*  $\Rightarrow$  *expr*

**translations**

```

this      == Acc (LVar This)
!!v       == Acc (LVar (ENam (VNam v)))
v::=e     == Expr (Ass (LVar (ENam (VNam v))) e)
Return e  == Expr (Ass (LVar (ENam Res)) e);; Jmp Ret
          — Res := e;; Jmp Ret
StatRef rt == Cast (RefT rt) (Lit Null)

```

### constdefs

```

is-stmt :: term ⇒ bool
is-stmt t ≡ ∃ c. t=In1r c

```

⟨ML⟩

**declare** *is-stmt-rews* [simp]

Here is some syntactic stuff to handle the injections of statements, expressions, variables and expression lists into general terms.

### syntax

```

expr-inj-term:: expr ⇒ term (⟨-⟩e 1000)
stmt-inj-term:: stmt ⇒ term (⟨-⟩s 1000)
var-inj-term:: var ⇒ term (⟨-⟩v 1000)
lst-inj-term:: expr list ⇒ term (⟨-⟩l 1000)

```

### translations

```

⟨e⟩e ↦ In1l e
⟨c⟩s ↦ In1r c
⟨v⟩v ↦ In2 v
⟨es⟩l ↦ In3 es

```

It seems to be more elegant to have an overloaded injection like the following.

```

axclass inj-term < type
consts inj-term:: 'a::inj-term ⇒ term (⟨-⟩ 1000)

```

How this overloaded injections work can be seen in the theory *DefiniteAssignment*. Other big inductive relations on terms defined in theories *WellType*, *Eval*, *Evaln* and *AxSem* don't follow this convention right now, but introduce subtle syntactic sugar in the relations themselves to make a distinction on expressions, statements and so on. So unfortunately you will encounter a mixture of dealing with these injections. The translations above are used as bridge between the different conventions.

**instance** *stmt::inj-term* ⟨proof⟩

### defs (overloaded)

*stmt-inj-term-def*: ⟨*c::stmt*⟩ ≡ *In1r c*

**lemma** *stmt-inj-term-simp*: ⟨*c::stmt*⟩ = *In1r c*  
 ⟨proof⟩

**lemma** *stmt-inj-term [iff]*: ⟨*x::stmt*⟩ = ⟨*y*⟩ ≡ *x = y*  
 ⟨proof⟩

**instance** *expr::inj-term* ⟨proof⟩

### defs (overloaded)

*expr-inj-term-def*: ⟨*e::expr*⟩ ≡ *In1l e*

**lemma** *expr-inj-term-simp*:  $\langle e::\text{expr} \rangle = \text{In1 } e$   
 $\langle \text{proof} \rangle$

**lemma** *expr-inj-term* [iff]:  $\langle x::\text{expr} \rangle = \langle y \rangle \equiv x = y$   
 $\langle \text{proof} \rangle$

**instance** *var::inj-term*  $\langle \text{proof} \rangle$

**defs** (overloaded)  
*var-inj-term-def*:  $\langle v::\text{var} \rangle \equiv \text{In2 } v$

**lemma** *var-inj-term-simp*:  $\langle v::\text{var} \rangle = \text{In2 } v$   
 $\langle \text{proof} \rangle$

**lemma** *var-inj-term* [iff]:  $\langle x::\text{var} \rangle = \langle y \rangle \equiv x = y$   
 $\langle \text{proof} \rangle$

**instance** *list::(type) inj-term*  $\langle \text{proof} \rangle$

**defs** (overloaded)  
*expr-list-inj-term-def*:  $\langle es::\text{expr list} \rangle \equiv \text{In3 } es$

**lemma** *expr-list-inj-term-simp*:  $\langle es::\text{expr list} \rangle = \text{In3 } es$   
 $\langle \text{proof} \rangle$

**lemma** *expr-list-inj-term* [iff]:  $\langle x::\text{expr list} \rangle = \langle y \rangle \equiv x = y$   
 $\langle \text{proof} \rangle$

**lemmas** *inj-term-simps* = *stmt-inj-term-simp* *expr-inj-term-simp* *var-inj-term-simp*  
*expr-list-inj-term-simp*

**lemmas** *inj-term-sym-simps* = *stmt-inj-term-simp* [THEN sym]  
*expr-inj-term-simp* [THEN sym]  
*var-inj-term-simp* [THEN sym]  
*expr-list-inj-term-simp* [THEN sym]

**lemma** *stmt-expr-inj-term* [iff]:  $\langle t::\text{stmt} \rangle \neq \langle w::\text{expr} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *expr-stmt-inj-term* [iff]:  $\langle t::\text{expr} \rangle \neq \langle w::\text{stmt} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *stmt-var-inj-term* [iff]:  $\langle t::\text{stmt} \rangle \neq \langle w::\text{var} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *var-stmt-inj-term* [iff]:  $\langle t::\text{var} \rangle \neq \langle w::\text{stmt} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *stmt-elist-inj-term* [iff]:  $\langle t::\text{stmt} \rangle \neq \langle w::\text{expr list} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *elist-stmt-inj-term* [iff]:  $\langle t::\text{expr list} \rangle \neq \langle w::\text{stmt} \rangle$

$\langle \text{proof} \rangle$

**lemma** *expr-var-inj-term* [iff]:  $\langle t::\text{expr} \rangle \neq \langle w::\text{var} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *var-expr-inj-term* [iff]:  $\langle t::\text{var} \rangle \neq \langle w::\text{expr} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *expr-elist-inj-term* [iff]:  $\langle t::\text{expr} \rangle \neq \langle w::\text{expr list} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *elist-expr-inj-term* [iff]:  $\langle t::\text{expr list} \rangle \neq \langle w::\text{expr} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *var-elist-inj-term* [iff]:  $\langle t::\text{var} \rangle \neq \langle w::\text{expr list} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *elist-var-inj-term* [iff]:  $\langle t::\text{expr list} \rangle \neq \langle w::\text{var} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *term-cases*:

$\llbracket \bigwedge v. P \langle v \rangle_v; \bigwedge e. P \langle e \rangle_e; \bigwedge c. P \langle c \rangle_s; \bigwedge l. P \langle l \rangle_l \rrbracket$   
 $\implies P t$   
 $\langle \text{proof} \rangle$

## Evaluation of unary operations

**consts** *eval-unop* :: *unop*  $\Rightarrow$  *val*  $\Rightarrow$  *val*

**primrec**

*eval-unop* *UPlus*  $v = \text{Intg } (\text{the-Intg } v)$

*eval-unop* *UMinus*  $v = \text{Intg } (- (\text{the-Intg } v))$

*eval-unop* *UBitNot*  $v = \text{Intg } 42$  — FIXME: Not yet implemented

*eval-unop* *UNot*  $v = \text{Bool } (\neg \text{the-Bool } v)$

## Evaluation of binary operations

**consts** *eval-binop* :: *binop*  $\Rightarrow$  *val*  $\Rightarrow$  *val*  $\Rightarrow$  *val*

**primrec**

*eval-binop* *Mul*  $v1 \ v2 = \text{Intg } ((\text{the-Intg } v1) * (\text{the-Intg } v2))$

*eval-binop* *Div*  $v1 \ v2 = \text{Intg } ((\text{the-Intg } v1) \text{ div } (\text{the-Intg } v2))$

*eval-binop* *Mod*  $v1 \ v2 = \text{Intg } ((\text{the-Intg } v1) \text{ mod } (\text{the-Intg } v2))$

*eval-binop* *Plus*  $v1 \ v2 = \text{Intg } ((\text{the-Intg } v1) + (\text{the-Intg } v2))$

*eval-binop* *Minus*  $v1 \ v2 = \text{Intg } ((\text{the-Intg } v1) - (\text{the-Intg } v2))$

— Be aware of the explicit coercion of the shift distance to nat

*eval-binop* *LShift*  $v1 \ v2 = \text{Intg } ((\text{the-Intg } v1) * (2^{(\text{nat } (\text{the-Intg } v2))}))$

*eval-binop* *RShift*  $v1 \ v2 = \text{Intg } ((\text{the-Intg } v1) \text{ div } (2^{(\text{nat } (\text{the-Intg } v2))}))$

*eval-binop* *RShiftU*  $v1 \ v2 = \text{Intg } 42$  — FIXME: Not yet implemented

*eval-binop* *Less*  $v1 \ v2 = \text{Bool } ((\text{the-Intg } v1) < (\text{the-Intg } v2))$

*eval-binop* *Le*  $v1 \ v2 = \text{Bool } ((\text{the-Intg } v1) \leq (\text{the-Intg } v2))$

*eval-binop* *Greater*  $v1 \ v2 = \text{Bool } ((\text{the-Intg } v2) < (\text{the-Intg } v1))$

*eval-binop* *Ge*  $v1 \ v2 = \text{Bool } ((\text{the-Intg } v2) \leq (\text{the-Intg } v1))$

*eval-binop* *Eq*  $v1 \ v2 = \text{Bool } (v1=v2)$

*eval-binop* *Neq*  $v1 \ v2 = \text{Bool } (v1 \neq v2)$

*eval-binop* *BitAnd*  $v1 \ v2 = \text{Intg } 42$  — FIXME: Not yet implemented

*eval-binop* *And*  $v1 \ v2 = \text{Bool } ((\text{the-Bool } v1) \wedge (\text{the-Bool } v2))$



*eval-binop BitXor*  $v1\ v2 = \text{Intg } 42$  — FIXME: Not yet implemented  
*eval-binop Xor*  $v1\ v2 = \text{Bool } ((\text{the-Bool } v1) \neq (\text{the-Bool } v2))$   
*eval-binop BitOr*  $v1\ v2 = \text{Intg } 42$  — FIXME: Not yet implemented  
*eval-binop Or*  $v1\ v2 = \text{Bool } ((\text{the-Bool } v1) \vee (\text{the-Bool } v2))$   
*eval-binop CondAnd*  $v1\ v2 = \text{Bool } ((\text{the-Bool } v1) \wedge (\text{the-Bool } v2))$   
*eval-binop CondOr*  $v1\ v2 = \text{Bool } ((\text{the-Bool } v1) \vee (\text{the-Bool } v2))$

**constdefs** *need-second-arg* :: *binop*  $\Rightarrow$  *val*  $\Rightarrow$  *bool*  
*need-second-arg binop v1*  $\equiv \neg ((\text{binop} = \text{CondAnd} \wedge \neg \text{the-Bool } v1) \vee$   
 $(\text{binop} = \text{CondOr} \wedge \text{the-Bool } v1))$

*CondAnd* and *CondOr* only evaluate the second argument if the value isn't already determined by the first argument

**lemma** *need-second-arg-CondAnd* [*simp*]: *need-second-arg CondAnd (Bool b) = b*  
 <proof>

**lemma** *need-second-arg-CondOr* [*simp*]: *need-second-arg CondOr (Bool b) = ( $\neg$  b)*  
 <proof>

**lemma** *need-second-arg-strict* [*simp*]:  
 $\llbracket \text{binop} \neq \text{CondAnd}; \text{binop} \neq \text{CondOr} \rrbracket \Longrightarrow \text{need-second-arg binop } b$   
 <proof>  
**end**



## Chapter 8

### Decl

## 7 Field, method, interface, and class declarations, whole Java programs

**theory** *Decl imports Term Table begin*

improvements:

- clarification and correction of some aspects of the package/access concept (Also submitted as bug report to the Java Bug Database: Bug Id: 4485402 and Bug Id: 4493343 <http://developer.java.s>)

simplifications:

- the only field and method modifiers are static and the access modifiers
- no constructors, which may be simulated by new + suitable methods
- there is just one global initializer per class, which can simulate all others
- no throws clause
- a void method is replaced by one that returns Unit (of dummy type Void)
- no interface fields
- every class has an explicit superclass (unused for Object)
- the (standard) methods of Object and of standard exceptions are not specified
- no main method

## 8 Modifier

### Access modifier

**datatype** *acc-modi*  
 $= Private \mid Package \mid Protected \mid Public$

We can define a linear order for the access modifiers. With Private yielding the most restrictive access and public the most liberal access policy: Private ; Package ; Protected ; Public

**instance** *acc-modi:: ord*  $\langle proof \rangle$

**defs (overloaded)**

*less-acc-def:*

$$\begin{aligned}
 a < (b::acc-modi) \\
 \equiv (case\ a\ of \\
 \quad Private &\Rightarrow (b=Package \vee b=Protected \vee b=Public) \\
 \quad | Package &\Rightarrow (b=Protected \vee b=Public) \\
 \quad | Protected &\Rightarrow (b=Public) \\
 \quad | Public &\Rightarrow False)
 \end{aligned}$$

*le-acc-def:*

$$a \leq (b::acc-modi) \equiv (a = b) \vee (a < b)$$

**instance** *acc-modi:: order*  
 $\langle proof \rangle$

**instance** *acc-modi:: linorder*  
 $\langle proof \rangle$



## 9 Declaration (base "class" for member, interface and class declarations)

**record** *decl* =  
     *access* :: *acc-modi*

### translations

*decl* <= (*type*) ( $\downarrow$  *access*::*acc-modi*)  
*decl* <= (*type*) ( $\downarrow$  *access*::*acc-modi*,...::'*a*)

## 10 Member (field or method)

**record** *member* = *decl* +  
     *static* :: *stat-modi*

### translations

*member* <= (*type*) ( $\downarrow$  *access*::*acc-modi*, *static*::*bool*)  
*member* <= (*type*) ( $\downarrow$  *access*::*acc-modi*, *static*::*bool*,...::'*a*)

## 11 Field

**record** *field* = *member* +  
     *type* :: *ty*

### translations

*field* <= (*type*) ( $\downarrow$  *access*::*acc-modi*, *static*::*bool*, *type*::*ty*)  
*field* <= (*type*) ( $\downarrow$  *access*::*acc-modi*, *static*::*bool*, *type*::*ty*,...::'*a*)

### types

*fdecl*  
     = *vname*  $\times$  *field*

### translations

*fdecl* <= (*type*) *vname*  $\times$  *field*

## 12 Method

**record** *mhead* = *member* +  
     *pars* :: *vname list*  
     *resT* :: *ty*

**record** *mbody* =  
     *lcls*:: (*vname*  $\times$  *ty*) *list*  
     *stmt*:: *stmt*

**record** *methd* = *mhead* +  
     *mbody*::*mbody*

**types** *mdecl* = *sig*  $\times$  *methd*

### translations

*mhead* <= (*type*) ( $\downarrow$  *access*::*acc-modi*, *static*::*bool*,  
     *pars*::*vname list*, *resT*::*ty*)  
*mhead* <= (*type*) ( $\downarrow$  *access*::*acc-modi*, *static*::*bool*,  
     *pars*::*vname list*, *resT*::*ty*,...::'*a*)  
*mbody* <= (*type*) ( $\downarrow$  *lcls*::(*vname*  $\times$  *ty*) *list*, *stmt*::*stmt*)  
*mbody* <= (*type*) ( $\downarrow$  *lcls*::(*vname*  $\times$  *ty*) *list*, *stmt*::*stmt*,...::'*a*)  
*methd* <= (*type*) ( $\downarrow$  *access*::*acc-modi*, *static*::*bool*,  
     *pars*::*vname list*, *resT*::*ty*, *mbody*::*mbody*)  
*methd* <= (*type*) ( $\downarrow$  *access*::*acc-modi*, *static*::*bool*,

$$mdecl \leq (type) \text{ sig} \times methd$$
**constdefs**

$$mhead::methd \Rightarrow mhead$$

$$mhead \ m \equiv (\text{access} = \text{access } m, \text{static} = \text{static } m, \text{pars} = \text{pars } m, \text{resT} = \text{resT } m)$$

**lemma** *access-mhead* [simp]:  $\text{access } (mhead \ m) = \text{access } m$   
 ⟨proof⟩

**lemma** *static-mhead* [simp]:  $\text{static } (mhead \ m) = \text{static } m$   
 ⟨proof⟩

**lemma** *pars-mhead* [simp]:  $\text{pars } (mhead \ m) = \text{pars } m$   
 ⟨proof⟩

**lemma** *resT-mhead* [simp]:  $\text{resT } (mhead \ m) = \text{resT } m$   
 ⟨proof⟩

To be able to talk uniformly about field and method declarations we introduce the notion of a member declaration (e.g. useful to define accessibility)

**datatype** *memberdecl* = *fdecl* *fdecl* | *mdecl* *mdecl*

**datatype** *memberid* = *fid* *vname* | *mid* *sig*

**axclass** *has-memberid* < *type*

**consts**

$$memberid :: 'a::has-memberid \Rightarrow memberid$$

**instance** *memberdecl::has-memberid* ⟨proof⟩

**defs (overloaded)**

*memberdecl-memberid-def*:

$$memberid \ m \equiv (\text{case } m \text{ of} \\
\quad fdecl \ (vn, f) \Rightarrow fid \ vn \\
\quad | \ mdecl \ (sig, m) \Rightarrow mid \ sig)$$

**lemma** *memberid-fdecl-simp*[simp]:  $memberid \ (fdecl \ (vn, f)) = fid \ vn$   
 ⟨proof⟩

**lemma** *memberid-fdecl-simp1*:  $memberid \ (fdecl \ f) = fid \ (fst \ f)$   
 ⟨proof⟩

**lemma** *memberid-mdecl-simp*[simp]:  $memberid \ (mdecl \ (sig, m)) = mid \ sig$   
 ⟨proof⟩

**lemma** *memberid-mdecl-simp1*:  $memberid \ (mdecl \ m) = mid \ (fst \ m)$   
 ⟨proof⟩

**instance**  $*$  :: (*type*, *has-memberid*) *has-memberid* ⟨proof⟩

**defs (overloaded)**

*pair-memberid-def*:

$\text{memberid } p \equiv \text{memberid } (\text{snd } p)$

**lemma** *memberid-pair-simp*[*simp*]:  $\text{memberid } (c, m) = \text{memberid } m$   
 $\langle \text{proof} \rangle$

**lemma** *memberid-pair-simp1*:  $\text{memberid } p = \text{memberid } (\text{snd } p)$   
 $\langle \text{proof} \rangle$

**constdefs** *is-field* ::  $qname \times memberdecl \Rightarrow bool$   
*is-field*  $m \equiv \exists \text{ decl } C \text{ f. } m = (\text{decl } C, \text{fdecl } f)$

**lemma** *is-fieldD*:  $\text{is-field } m \Longrightarrow \exists \text{ decl } C \text{ f. } m = (\text{decl } C, \text{fdecl } f)$   
 $\langle \text{proof} \rangle$

**lemma** *is-fieldI*:  $\text{is-field } (C, \text{fdecl } f)$   
 $\langle \text{proof} \rangle$

**constdefs** *is-method* ::  $qname \times memberdecl \Rightarrow bool$   
*is-method*  $\text{membr} \equiv \exists \text{ decl } C \text{ m. } \text{membr} = (\text{decl } C, \text{mdecl } m)$

**lemma** *is-methodD*:  $\text{is-method } \text{membr} \Longrightarrow \exists \text{ decl } C \text{ m. } \text{membr} = (\text{decl } C, \text{mdecl } m)$   
 $\langle \text{proof} \rangle$

**lemma** *is-methodI*:  $\text{is-method } (C, \text{mdecl } m)$   
 $\langle \text{proof} \rangle$

### 13 Interface

**record** *ibody* = *decl* + — interface body  
*imethods* ::  $(sig \times mhead) \text{ list}$  — method heads

**record** *iface* = *ibody* + — interface  
*isuperIfs* ::  $qname \text{ list}$  — superinterface list

**types**  
*idecl* — interface declaration, cf. 9.1  
 $= qname \times \text{iface}$

#### translations

$\text{ibody} \leq (\text{type}) \ (\backslash \text{access} :: \text{acc-modi}, \text{imethods} :: (\text{sig} \times \text{mhead}) \text{ list})$   
 $\text{ibody} \leq (\text{type}) \ (\backslash \text{access} :: \text{acc-modi}, \text{imethods} :: (\text{sig} \times \text{mhead}) \text{ list}, \dots :: 'a)$   
 $\text{iface} \leq (\text{type}) \ (\backslash \text{access} :: \text{acc-modi}, \text{imethods} :: (\text{sig} \times \text{mhead}) \text{ list},$   
 $\text{isuperIfs} :: qname \text{ list})$   
 $\text{iface} \leq (\text{type}) \ (\backslash \text{access} :: \text{acc-modi}, \text{imethods} :: (\text{sig} \times \text{mhead}) \text{ list},$   
 $\text{isuperIfs} :: qname \text{ list}, \dots :: 'a)$   
 $\text{idecl} \leq (\text{type}) \ qname \times \text{iface}$

#### constdefs

$\text{ibody} :: \text{iface} \Rightarrow \text{ibody}$   
 $\text{ibody } i \equiv (\backslash \text{access} = \text{access } i, \text{imethods} = \text{imethods } i)$



**lemma** *access-ibody* [simp]: (*access* (*ibody* *i*)) = *access* *i*  
 ⟨*proof*⟩

**lemma** *imethods-ibody* [simp]: (*imethods* (*ibody* *i*)) = *imethods* *i*  
 ⟨*proof*⟩

## 14 Class

**record** *cbody* = *decl* + — class body  
     *cfields*:: *fdecl* *list*  
     *methods*:: *mdecl* *list*  
     *init* :: *stmt* — initializer

**record** *class* = *cbody* + — class  
     *super* :: *qtname* — superclass  
     *superIfs*:: *qtname* *list* — implemented interfaces

### types

*cdecl* — class declaration, cf. 8.1  
 = *qtname* × *class*

### translations

*cbody* <= (*type*) (⌊*access*::*acc-modi*,*cfields*::*fdecl* *list*,  
                   *methods*::*mdecl* *list*,*init*::*stmt*⌋)  
*cbody* <= (*type*) (⌊*access*::*acc-modi*,*cfields*::*fdecl* *list*,  
                   *methods*::*mdecl* *list*,*init*::*stmt*,...::'*a*'⌋)  
*class* <= (*type*) (⌊*access*::*acc-modi*,*cfields*::*fdecl* *list*,  
                   *methods*::*mdecl* *list*,*init*::*stmt*,  
                   *super*::*qtname*,*superIfs*::*qtname* *list*⌋)  
*class* <= (*type*) (⌊*access*::*acc-modi*,*cfields*::*fdecl* *list*,  
                   *methods*::*mdecl* *list*,*init*::*stmt*,  
                   *super*::*qtname*,*superIfs*::*qtname* *list*,...::'*a*'⌋)  
*cdecl* <= (*type*) *qtname* × *class*

### constdefs

*cbody* :: *class* ⇒ *cbody*  
*cbody* *c* ≡ (⌊*access*=*access* *c*, *cfields*=*cfields* *c*,*methods*=*methods* *c*,*init*=*init* *c*⌋)

**lemma** *access-cbody* [simp]: *access* (*cbody* *c*) = *access* *c*  
 ⟨*proof*⟩

**lemma** *cfields-cbody* [simp]: *cfields* (*cbody* *c*) = *cfields* *c*  
 ⟨*proof*⟩

**lemma** *methods-cbody* [simp]: *methods* (*cbody* *c*) = *methods* *c*  
 ⟨*proof*⟩

**lemma** *init-cbody* [simp]: *init* (*cbody* *c*) = *init* *c*  
 ⟨*proof*⟩

### standard classes

#### consts

*Object-mdecls* :: *mdecl list* — methods of Object  
*SXcpt-mdecls* :: *mdecl list* — methods of SXcpts  
*ObjectC* :: *cdecl* — declaration of root class  
*SXcptC* :: *xname*  $\Rightarrow$  *cdecl* — declarations of throwable classes

## defs

*ObjectC-def: ObjectC*  $\equiv$  (*Object*, ( $\text{access} = \text{Public}$ ,  $\text{cfields} = []$ ,  $\text{methods} = \text{Object-mdecls}$ ,  
 $\text{init} = \text{Skip}$ ,  $\text{super} = \text{arbitrary}$ ,  $\text{superIfs} = []$ ))  
*SXcptC-def: SXcptC xn*  $\equiv$  (*SXcpt xn*, ( $\text{access} = \text{Public}$ ,  $\text{cfields} = []$ ,  $\text{methods} = \text{SXcpt-mdecls}$ ,  
 $\text{init} = \text{Skip}$ ,  
 $\text{super} = \text{if } xn = \text{Throwable then Object}$   
 $\text{else SXcpt Throwable}$ ,  
 $\text{superIfs} = []$ ))

**lemma** *ObjectC-neq-SXcptC* [*simp*]: *ObjectC*  $\neq$  *SXcptC xn*  
 <proof>

**lemma** *SXcptC-inject* [*simp*]: (*SXcptC xn* = *SXcptC xm*) = (*xn* = *xm*)  
 <proof>

**constdefs** *standard-classes* :: *cdecl list*  
*standard-classes*  $\equiv$  [*ObjectC*, *SXcptC Throwable*,  
*SXcptC NullPointerException*, *SXcptC OutOfMemory*, *SXcptC ClassCast*,  
*SXcptC NegArrSize*, *SXcptC IndOutBound*, *SXcptC ArrStore*]

## programs

**record** *prog* =  
*ifaces* :: *idecl list*  
*classes* :: *cdecl list*

## translations

*prog*  $\leq$  (*type*) ( $\text{ifaces} :: \text{idecl list}$ ,  $\text{classes} :: \text{cdecl list}$ )  
*prog*  $\leq$  (*type*) ( $\text{ifaces} :: \text{idecl list}$ ,  $\text{classes} :: \text{cdecl list}$ , ... : 'a)

## syntax

*iface* :: *prog*  $\Rightarrow$  (*qtname*, *iface*) *table*  
*class* :: *prog*  $\Rightarrow$  (*qtname*, *class*) *table*  
*is-iface* :: *prog*  $\Rightarrow$  *qtname*  $\Rightarrow$  *bool*  
*is-class* :: *prog*  $\Rightarrow$  *qtname*  $\Rightarrow$  *bool*

## translations

*iface G I* == *table-of (ifaces G) I*  
*class G C* == *table-of (classes G) C*  
*is-iface G I* == *iface G I*  $\neq$  *None*  
*is-class G C* == *class G C*  $\neq$  *None*

## is type

### consts

*is-type* :: *prog*  $\Rightarrow$  *ty*  $\Rightarrow$  *bool*  
*isrtype* :: *prog*  $\Rightarrow$  *ref-ty*  $\Rightarrow$  *bool*

**primrec** *is-type G (PrimT pt)* = *True*  
*is-type G (RefT rt)* = *isrtype G rt*

```

isrtype G (NullT _) = True
isrtype G (IfaceT tn) = is-iface G tn
isrtype G (ClassT tn) = is-class G tn
isrtype G (ArrayT T) = is-type G T

```

**lemma** *type-is-iface*:  $is\text{-}type\ G\ (Iface\ I) \implies is\text{-}iface\ G\ I$   
 <proof>

**lemma** *type-is-class*:  $is\text{-}type\ G\ (Class\ C) \implies is\text{-}class\ G\ C$   
 <proof>

## subinterface and subclass relation, in anticipation of TypeRel.thy

### consts

```

subint1 :: prog => (qname × qname) set — direct subinterface
subcls1 :: prog => (qname × qname) set — direct subclass

```

### defs

```

subint1-def: subint1 G ≡ {(I,J). ∃ i∈iface G I: J∈set (isuperIfs i)}
subcls1-def: subcls1 G ≡ {(C,D). C≠Object ∧ (∃ c∈class G C: super c = D)}

```

### syntax

```

@subcls1 :: prog => [qname, qname] => bool (|-<:C1- [71,71,71] 70)
@subclseq:: prog => [qname, qname] => bool (|-<=:C-[71,71,71] 70)
@subcls  :: prog => [qname, qname] => bool (|-<:C-[71,71,71] 70)

```

### syntax (*xsymbols*)

```

@subcls1 :: prog => [qname, qname] => bool (⊢-<_{C1}- [71,71,71] 70)
@subclseq:: prog => [qname, qname] => bool (⊢-<_C - [71,71,71] 70)
@subcls  :: prog => [qname, qname] => bool (⊢-<_C - [71,71,71] 70)

```

### translations

```

G⊢C <_{C1} D == (C,D) ∈ subcls1 G
G⊢C ⊆_C D == (C,D) ∈ (subcls1 G)^*
G⊢C <_C D == (C,D) ∈ (subcls1 G)^+

```

**lemma** *subint1I*:  $\llbracket iface\ G\ I = Some\ i;\ J \in set\ (isuperIfs\ i) \rrbracket$   
 $\implies (I,J) \in subint1\ G$   
 <proof>

**lemma** *subcls1I*:  $\llbracket class\ G\ C = Some\ c;\ C \neq Object \rrbracket \implies (C,(super\ c)) \in subcls1\ G$   
 <proof>

**lemma** *subint1D*:  $(I,J) \in subint1\ G \implies \exists i \in iface\ G\ I: J \in set\ (isuperIfs\ i)$   
 <proof>

### lemma *subcls1D*:

$(C,D) \in subcls1\ G \implies C \neq Object \wedge (\exists c. class\ G\ C = Some\ c \wedge (super\ c = D))$   
 <proof>

**lemma** *subint1-def2*:

$\text{subint1 } G = (\text{SIGMA } I: \{I. \text{is-iface } G \ I\}. \text{set } (\text{isuperIfs } (\text{the } (\text{iface } G \ I))))$   
 $\langle \text{proof} \rangle$

**lemma** *subcls1-def2*:

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

**lemma** *subcls-is-class*:

$\llbracket G \vdash C \prec_C D \rrbracket \implies \exists c. \text{class } G \ C = \text{Some } c$   
 $\langle \text{proof} \rangle$

**lemma** *no-subcls1-Object*:  $G \vdash \text{Object} \prec_{C1} D \implies P$

$\langle \text{proof} \rangle$

**lemma** *no-subcls-Object*:  $G \vdash \text{Object} \prec_C D \implies P$

$\langle \text{proof} \rangle$

## well-structured programs

**constdefs**

$\text{ws-idecl} :: \text{prog} \Rightarrow \text{qname} \Rightarrow \text{qname list} \Rightarrow \text{bool}$   
 $\text{ws-idecl } G \ I \ si \equiv \forall J \in \text{set } si. \text{is-iface } G \ J \ \wedge \ (J, I) \notin (\text{subint1 } G)^\wedge +$

$\text{ws-cdecl} :: \text{prog} \Rightarrow \text{qname} \Rightarrow \text{qname} \Rightarrow \text{bool}$   
 $\text{ws-cdecl } G \ C \ sc \equiv C \neq \text{Object} \longrightarrow \text{is-class } G \ sc \wedge (sc, C) \notin (\text{subcls1 } G)^\wedge +$

$\text{ws-prog} :: \text{prog} \Rightarrow \text{bool}$   
 $\text{ws-prog } G \equiv (\forall (I, i) \in \text{set } (\text{ifaces } G). \text{ws-idecl } G \ I \ (\text{isuperIfs } i)) \wedge$   
 $(\forall (C, c) \in \text{set } (\text{classes } G). \text{ws-cdecl } G \ C \ (\text{super } c))$

**lemma** *ws-progI*:

$\llbracket \forall (I, i) \in \text{set } (\text{ifaces } G). \forall J \in \text{set } (\text{isuperIfs } i). \text{is-iface } G \ J \wedge$   
 $(J, I) \notin (\text{subint1 } G)^\wedge +;$   
 $\forall (C, c) \in \text{set } (\text{classes } G). C \neq \text{Object} \longrightarrow \text{is-class } G \ (\text{super } c) \wedge$   
 $((\text{super } c), C) \notin (\text{subcls1 } G)^\wedge +$   
 $\rrbracket \implies \text{ws-prog } G$   
 $\langle \text{proof} \rangle$

**lemma** *ws-prog-ideclD*:

$\llbracket \text{iface } G \ I = \text{Some } i; J \in \text{set } (\text{isuperIfs } i); \text{ws-prog } G \rrbracket \implies$   
 $\text{is-iface } G \ J \wedge (J, I) \notin (\text{subint1 } G)^\wedge +$   
 $\langle \text{proof} \rangle$

**lemma** *ws-prog-cdeclD*:

$\llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object}; \text{ws-prog } G \rrbracket \implies$   
 $\text{is-class } G \ (\text{super } c) \wedge (\text{super } c, C) \notin (\text{subcls1 } G)^\wedge +$   
 $\langle \text{proof} \rangle$

**well-foundedness**

**lemma** *finite-is-iface*: *finite*  $\{I. \text{is-iface } G \ I\}$   
 $\langle \text{proof} \rangle$

**lemma** *finite-is-class*: *finite*  $\{C. \text{is-class } G \ C\}$   
 $\langle \text{proof} \rangle$

**lemma** *finite-subint1*: *finite* (*subint1*  $G$ )  
 $\langle \text{proof} \rangle$

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

**lemma** *subint1-irrefl-lemma1*:  
 $\text{ws-prog } G \implies (\text{subint1 } G)^{-1} \cap (\text{subint1 } G)^+ = \{\}$   
 $\langle \text{proof} \rangle$

**lemma** *subcls1-irrefl-lemma1*:  
 $\text{ws-prog } G \implies (\text{subcls1 } G)^{-1} \cap (\text{subcls1 } G)^+ = \{\}$   
 $\langle \text{proof} \rangle$

**lemmas** *subint1-irrefl-lemma2* = *subint1-irrefl-lemma1*  $[ \text{THEN irrefl-tranclI} ]$   
**lemmas** *subcls1-irrefl-lemma2* = *subcls1-irrefl-lemma1*  $[ \text{THEN irrefl-tranclI} ]$

**lemma** *subint1-irrefl*:  $\llbracket (x, y) \in \text{subint1 } G; \text{ws-prog } G \rrbracket \implies x \neq y$   
 $\langle \text{proof} \rangle$

**lemma** *subcls1-irrefl*:  $\llbracket (x, y) \in \text{subcls1 } G; \text{ws-prog } G \rrbracket \implies x \neq y$   
 $\langle \text{proof} \rangle$

**lemmas** *subint1-acyclic* = *subint1-irrefl-lemma2*  $[ \text{THEN acyclicI, standard} ]$   
**lemmas** *subcls1-acyclic* = *subcls1-irrefl-lemma2*  $[ \text{THEN acyclicI, standard} ]$

**lemma** *wf-subint1*:  $\text{ws-prog } G \implies \text{wf } ((\text{subint1 } G)^{-1})$   
 $\langle \text{proof} \rangle$

**lemma** *wf-subcls1*:  $\text{ws-prog } G \implies \text{wf } ((\text{subcls1 } G)^{-1})$   
 $\langle \text{proof} \rangle$

**lemma** *subint1-induct*:  
 $\llbracket \text{ws-prog } G; \bigwedge x. \forall y. (x, y) \in \text{subint1 } G \longrightarrow P \ y \implies P \ x \rrbracket \implies P \ a$   
 $\langle \text{proof} \rangle$

**lemma** *subcls1-induct*  $[ \text{consumes } 1 ]$ :  
 $\llbracket \text{ws-prog } G; \bigwedge x. \forall y. (x, y) \in \text{subcls1 } G \longrightarrow P \ y \implies P \ x \rrbracket \implies P \ a$

$\langle \text{proof} \rangle$

**lemma** *ws-subint1-induct*:

$\llbracket \text{is-iface } G \ I; \text{ws-prog } G; \bigwedge I \ i. \llbracket \text{iface } G \ I = \text{Some } i \wedge$   
 $(\forall J \in \text{set } (\text{isuperIfs } i). (I,J) \in \text{subint1 } G \wedge P \ J \wedge \text{is-iface } G \ J) \rrbracket \implies P \ I$   
 $\rrbracket \implies P \ I$   
 $\langle \text{proof} \rangle$

**lemma** *ws-subcls1-induct*:  $\llbracket \text{is-class } G \ C; \text{ws-prog } G;$

$\bigwedge C \ c. \llbracket \text{class } G \ C = \text{Some } c;$   
 $(C \neq \text{Object} \longrightarrow (C, (\text{super } c)) \in \text{subcls1 } G \wedge$   
 $P (\text{super } c) \wedge \text{is-class } G (\text{super } c)) \rrbracket \implies P \ C$   
 $\rrbracket \implies P \ C$   
 $\langle \text{proof} \rangle$

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

$\llbracket \text{class } G \ C = \text{Some } c; \text{ws-prog } G;$   
 $\bigwedge co. \text{class } G \ \text{Object} = \text{Some } co \implies P \ \text{Object};$   
 $\bigwedge C \ c. \llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object}; P (\text{super } c) \rrbracket \implies P \ C$   
 $\rrbracket \implies P \ C$   
 $\langle \text{proof} \rangle$

**lemma** *ws-class-induct'* [consumes 2, case-names Object Subcls]:

$\llbracket \text{is-class } G \ C; \text{ws-prog } G;$   
 $\bigwedge co. \text{class } G \ \text{Object} = \text{Some } co \implies P \ \text{Object};$   
 $\bigwedge C \ c. \llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object}; P (\text{super } c) \rrbracket \implies P \ C$   
 $\rrbracket \implies P \ C$   
 $\langle \text{proof} \rangle$

**lemma** *ws-class-induct''* [consumes 2, case-names Object Subcls]:

$\llbracket \text{class } G \ C = \text{Some } c; \text{ws-prog } G;$   
 $\bigwedge co. \text{class } G \ \text{Object} = \text{Some } co \implies P \ \text{Object } co;$   
 $\bigwedge C \ c \ sc. \llbracket \text{class } G \ C = \text{Some } c; \text{class } G (\text{super } c) = \text{Some } sc;$   
 $C \neq \text{Object}; P (\text{super } c) \ sc \rrbracket \implies P \ C \ c$   
 $\rrbracket \implies P \ C \ c$   
 $\langle \text{proof} \rangle$

**lemma** *ws-interface-induct* [consumes 2, case-names Step]:

**assumes** *is-if-I*: *is-iface*  $G \ I$  **and**

*ws*: *ws-prog*  $G$  **and**

*hyp-sub*:  $\bigwedge I \ i. \llbracket \text{iface } G \ I = \text{Some } i;$

$\forall J \in \text{set } (\text{isuperIfs } i).$

$(I,J) \in \text{subint1 } G \wedge P \ J \wedge \text{is-iface } G \ J \rrbracket \implies P \ I$

**shows**  $P \ I$

$\langle \text{proof} \rangle$

## general recursion operators for the interface and class hierarchies

**consts**

*iface-rec* ::  $\text{prog} \times \text{qtname} \Rightarrow (\text{qtname} \Rightarrow \text{iface} \Rightarrow 'a \ \text{set} \Rightarrow 'a) \Rightarrow 'a$

*class-rec* ::  $\text{prog} \times \text{qtname} \Rightarrow 'a \Rightarrow (\text{qtname} \Rightarrow \text{class} \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a$

```

recdef iface-rec same-fst ws-prog ( $\lambda G. (subint1\ G)^{-1}$ )
iface-rec ( $G, I$ ) =
  ( $\lambda f. case\ iface\ G\ I\ of$ 
     $None \Rightarrow arbitrary$ 
    |  $Some\ i \Rightarrow if\ ws-prog\ G$ 
      then  $f\ I\ i$ 
       $((\lambda J. iface-rec\ (G, J)\ f)\ 'set\ (isuperIfs\ i))$ 
      else  $arbitrary$ )
(hints recdef-wf: wf-subint1 intro: subint1I)
declare iface-rec.simps [simp del]

```

```

lemma iface-rec:
 $\llbracket iface\ G\ I = Some\ i; ws-prog\ G \rrbracket \implies$ 
 $iface-rec\ (G, I)\ f = f\ I\ i\ ((\lambda J. iface-rec\ (G, J)\ f)\ 'set\ (isuperIfs\ i))$ 
 $\langle proof \rangle$ 

```

```

recdef class-rec same-fst ws-prog ( $\lambda G. (subcls1\ G)^{-1}$ )
class-rec( $G, C$ ) =
  ( $\lambda t f. case\ class\ G\ C\ of$ 
     $None \Rightarrow arbitrary$ 
    |  $Some\ c \Rightarrow if\ ws-prog\ G$ 
      then  $f\ C\ c$ 
       $(if\ C = Object\ then\ t$ 
        else  $class-rec\ (G, super\ c)\ t\ f)$ 
      else  $arbitrary$ )
(hints recdef-wf: wf-subcls1 intro: subcls1I)
declare class-rec.simps [simp del]

```

```

lemma class-rec:  $\llbracket class\ G\ C = Some\ c; ws-prog\ G \rrbracket \implies$ 
 $class-rec\ (G, C)\ t\ f =$ 
 $f\ C\ c\ (if\ C = Object\ then\ t\ else\ class-rec\ (G, super\ c)\ t\ f)$ 
 $\langle proof \rangle$ 

```

```

constdefs
imethds::  $prog \Rightarrow qtname \Rightarrow (sig, qtname \times mhead)\ tables$ 
  — methods of an interface, with overriding and inheritance, cf. 9.2
imethds  $G\ I$ 
   $\equiv iface-rec\ (G, I)$ 
   $(\lambda I\ i\ ts. (Un-tables\ ts) \oplus \oplus$ 
     $(o2s \circ table-of\ (map\ (\lambda(s, m). (s, I, m))\ (imethds\ i))))$ 

```

**end**





## Chapter 9

# TypeRel

## 15 The relations between Java types

**theory** *TypeRel* **imports** *Decl* **begin**

simplifications:

- subinterface, subclass and widening relation includes identity

improvements over Java Specification 1.0:

- narrowing reference conversion also in cases where the return types of a pair of methods common to both types are in widening (rather identity) relation
- one could add similar constraints also for other cases

design issues:

- the type relations do not require *is-type* for their arguments
- the *subint1* and *subcls1* relations imply *is-iface/is-class* for their first arguments, which is required for their finiteness

**consts**

```
implmt1  :: prog => (qname × qname) set — direct implementation
implmt   :: prog => (qname × qname) set — implementation
widen    :: prog => (ty   × ty   ) set — widening
narrow   :: prog => (ty   × ty   ) set — narrowing
cast     :: prog => (ty   × ty   ) set — casting
```

**syntax**

```
@subint1 :: prog => [qname, qname] => bool (|-<:I1- [71,71,71] 70)
@subint  :: prog => [qname, qname] => bool (|-<=:I -[71,71,71] 70)

@implmt1 :: prog => [qname, qname] => bool (|-~>1- [71,71,71] 70)
@implmt  :: prog => [qname, qname] => bool (|-~>- [71,71,71] 70)
@widen   :: prog => [ty   , ty   ] => bool (|-<=: - [71,71,71] 70)
@narrow  :: prog => [ty   , ty   ] => bool (|->:- [71,71,71] 70)
@cast    :: prog => [ty   , ty   ] => bool (|-<=: ? -[71,71,71] 70)
```

**syntax** (*symbols*)

```
@subint1 :: prog => [qname, qname] => bool (|-<:I1- [71,71,71] 70)
@subint  :: prog => [qname, qname] => bool (|-<=:I - [71,71,71] 70)

@implmt1 :: prog => [qname, qname] => bool (|-~>1- [71,71,71] 70)
@implmt  :: prog => [qname, qname] => bool (|-~>- [71,71,71] 70)
@widen   :: prog => [ty   , ty   ] => bool (|-<:- [71,71,71] 70)
@narrow  :: prog => [ty   , ty   ] => bool (|->- [71,71,71] 70)
@cast    :: prog => [ty   , ty   ] => bool (|-<:? - [71,71,71] 70)
```

**translations**

$$G \vdash I \prec_{I1} J == (I, J) \in \text{subint1 } G$$

$$G \vdash I \preceq_I J == (I, J) \in (\text{subint1 } G)^{\wedge*} \text{ — cf. 9.1.3}$$

$$G \vdash C \leadsto_1 I == (C, I) \in \text{implmt1 } G$$

$$G \vdash C \leadsto I == (C, I) \in \text{implmt } G$$

$$G \vdash S \preceq T == (S, T) \in \text{widen } G$$

$$G \vdash S \succ T == (S, T) \in \text{narrow } G$$

$$G \vdash S \preceq^? T == (S, T) \in \text{cast } G$$

### subclass and subinterface relations

**lemmas** *subcls-direct* = *subcls1I* [THEN *r-into-rtrancl*, *standard*]

**lemma** *subcls-direct1*:

$$\llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object}; D = \text{super } c \rrbracket \implies G \vdash C \preceq_C D$$

*<proof>*

**lemma** *subcls1I1*:

$$\llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object}; D = \text{super } c \rrbracket \implies G \vdash C \prec_{C1} D$$

*<proof>*

**lemma** *subcls-direct2*:

$$\llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object}; D = \text{super } c \rrbracket \implies G \vdash C \prec_C D$$

*<proof>*

**lemma** *subclseq-trans*:  $\llbracket G \vdash A \preceq_C B; G \vdash B \preceq_C C \rrbracket \implies G \vdash A \preceq_C C$

*<proof>*

**lemma** *subcls-trans*:  $\llbracket G \vdash A \prec_C B; G \vdash B \prec_C C \rrbracket \implies G \vdash A \prec_C C$

*<proof>*

**lemma** *SXcpt-subcls-Throwable-lemma*:

$$\llbracket \text{class } G \ (\text{SXcpt } xn) = \text{Some } xc;$$

$$\text{super } xc = (\text{if } xn = \text{Throwable} \text{ then } \text{Object} \text{ else } \text{SXcpt } \text{Throwable}) \rrbracket$$

$$\implies G \vdash \text{SXcpt } xn \preceq_C \text{SXcpt } \text{Throwable}$$

*<proof>*

**lemma** *subcls-ObjectI*:  $\llbracket \text{is-class } G \ C; \text{ws-prog } G \rrbracket \implies G \vdash C \preceq_C \text{Object}$

*<proof>*

**lemma** *subclseq-ObjectD* [*dest!*]:  $G \vdash \text{Object} \preceq_C C \implies C = \text{Object}$

*<proof>*

**lemma** *subcls-ObjectD* [*dest!*]:  $G \vdash \text{Object} \prec_C C \implies \text{False}$

*<proof>*

**lemma** *subcls-ObjectI1* [*intro!*]:

$$\llbracket C \neq \text{Object}; \text{is-class } G \ C; \text{ws-prog } G \rrbracket \implies G \vdash C \prec_C \text{Object}$$

*<proof>*

**lemma** *subcls-is-class*:  $(C, D) \in (\text{subcls1 } G)^+ \implies \text{is-class } G \ C$   
 $\langle \text{proof} \rangle$

**lemma** *subcls-is-class2* [rule-format (no-asm)]:  
 $G \vdash C \preceq_C D \implies \text{is-class } G \ D \longrightarrow \text{is-class } G \ C$   
 $\langle \text{proof} \rangle$

**lemma** *single-inheritance*:  
 $\llbracket G \vdash A \prec_{C1} B; G \vdash A \prec_{C1} C \rrbracket \implies B = C$   
 $\langle \text{proof} \rangle$

**lemma** *subcls-compareable*:  
 $\llbracket G \vdash A \preceq_C X; G \vdash A \preceq_C Y \rrbracket \implies G \vdash X \preceq_C Y \vee G \vdash Y \preceq_C X$   
 $\langle \text{proof} \rangle$

**lemma** *subcls1-irrefl*:  $\llbracket G \vdash C \prec_{C1} D; \text{ws-prog } G \rrbracket \implies C \neq D$   
 $\langle \text{proof} \rangle$

**lemma** *no-subcls-Object*:  $G \vdash C \prec_C D \implies C \neq \text{Object}$   
 $\langle \text{proof} \rangle$

**lemma** *subcls-acyclic*:  $\llbracket G \vdash C \prec_C D; \text{ws-prog } G \rrbracket \implies \neg G \vdash D \prec_C C$   
 $\langle \text{proof} \rangle$

**lemma** *subclseq-cases* [consumes 1, case-names Eq Subcls]:  
 $\llbracket G \vdash C \preceq_C D; C = D \implies P; G \vdash C \prec_C D \implies P \rrbracket \implies P$   
 $\langle \text{proof} \rangle$

**lemma** *subclseq-acyclic*:  
 $\llbracket G \vdash C \preceq_C D; G \vdash D \preceq_C C; \text{ws-prog } G \rrbracket \implies C = D$   
 $\langle \text{proof} \rangle$

**lemma** *subcls-irrefl*:  $\llbracket G \vdash C \prec_C D; \text{ws-prog } G \rrbracket \implies C \neq D$   
 $\langle \text{proof} \rangle$

**lemma** *invert-subclseq*:  
 $\llbracket G \vdash C \preceq_C D; \text{ws-prog } G \rrbracket \implies \neg G \vdash D \prec_C C$   
 $\langle \text{proof} \rangle$

**lemma** *invert-subcls*:  
 $\llbracket G \vdash C \prec_C D; \text{ws-prog } G \rrbracket \implies \neg G \vdash D \preceq_C C$

$\langle \text{proof} \rangle$

**lemma** *subcls-superD*:

$\llbracket G \vdash C \prec_C D; \text{class } G \ C = \text{Some } c \rrbracket \implies G \vdash (\text{super } c) \preceq_C D$   
 $\langle \text{proof} \rangle$

**lemma** *subclseq-superD*:

$\llbracket G \vdash C \preceq_C D; C \neq D; \text{class } G \ C = \text{Some } c \rrbracket \implies G \vdash (\text{super } c) \preceq_C D$   
 $\langle \text{proof} \rangle$

## implementation relation

**defs**

— direct implementation, cf. 8.1.3

*implmt1-def*:  $\text{implmt1 } G \equiv \{(C, I). C \neq \text{Object} \wedge (\exists c \in \text{class } G \ C: I \in \text{set } (\text{superIfs } c))\}$

**lemma** *implmt1D*:  $G \vdash C \rightsquigarrow 1I \implies C \neq \text{Object} \wedge (\exists c \in \text{class } G \ C: I \in \text{set } (\text{superIfs } c))$

$\langle \text{proof} \rangle$

**inductive** *implmt* *G* **intros**

— cf. 8.1.4

*direct*:  $G \vdash C \rightsquigarrow 1J \implies G \vdash C \rightsquigarrow J$   
*subint*:  $\llbracket G \vdash C \rightsquigarrow 1I; G \vdash I \preceq I \ J \rrbracket \implies G \vdash C \rightsquigarrow J$   
*subcls1*:  $\llbracket G \vdash C \prec_{C1} D; G \vdash D \rightsquigarrow J \rrbracket \implies G \vdash C \rightsquigarrow J$

**lemma** *implmtD*:  $G \vdash C \rightsquigarrow J \implies (\exists I. G \vdash C \rightsquigarrow 1I \wedge G \vdash I \preceq I \ J) \vee (\exists D. G \vdash C \prec_{C1} D \wedge G \vdash D \rightsquigarrow J)$

$\langle \text{proof} \rangle$

**lemma** *implmt-ObjectE* [*elim!*]:  $G \vdash \text{Object} \rightsquigarrow I \implies R$

$\langle \text{proof} \rangle$

**lemma** *subcls-implmt* [*rule-format* (*no-asm*)]:  $G \vdash A \preceq_C B \implies G \vdash B \rightsquigarrow K \longrightarrow G \vdash A \rightsquigarrow K$

$\langle \text{proof} \rangle$

**lemma** *implmt-subint2*:  $\llbracket G \vdash A \rightsquigarrow J; G \vdash J \preceq I \ K \rrbracket \implies G \vdash A \rightsquigarrow K$

$\langle \text{proof} \rangle$

**lemma** *implmt-is-class*:  $G \vdash C \rightsquigarrow I \implies \text{is-class } G \ C$

$\langle \text{proof} \rangle$

## widening relation

**inductive** *widen* *G* **intros**

— widening, viz. method invocation conversion, cf. 5.3 i.e. kind of syntactic subtyping

*refl*:  $G \vdash T \preceq T$  — identity conversion, cf. 5.1.1

*subint*:  $G \vdash I \preceq I \ J \implies G \vdash \text{Iface } I \preceq \text{Iface } J$  — wid.ref.conv., cf. 5.1.4

*int-obj*:  $G \vdash \text{Iface } I \preceq \text{Class } \text{Object}$

*subcls*:  $G \vdash C \preceq_C D \implies G \vdash \text{Class } C \preceq \text{Class } D$

*implmt*:  $G \vdash C \rightsquigarrow I \implies G \vdash \text{Class } C \preceq \text{Iface } I$

$null: \quad G \vdash NT \preceq RefT \ R$   
 $arr-obj: \quad G \vdash T.\boxed{\phantom{x}} \preceq Class \ Object$   
 $array: \quad G \vdash RefT \ S \preceq RefT \ T \implies G \vdash RefT \ S.\boxed{\phantom{x}} \preceq RefT \ T.\boxed{\phantom{x}}$

**declare** *widen.refl* [intro!]  
**declare** *widen.intros* [simp]

**lemma** *widen-PrimT*:  $G \vdash PrimT \ x \preceq T \implies (\exists y. \ T = PrimT \ y)$   
 <proof>

**lemma** *widen-PrimT2*:  $G \vdash S \preceq PrimT \ x \implies \exists y. \ S = PrimT \ y$   
 <proof>

These widening lemmata hold in Bali but are too strong for ordinary Java. They would not work for real Java Integral Types, like short, long, int. These lemmata are just for documentation and are not used.

**lemma** *widen-PrimT-strong*:  $G \vdash PrimT \ x \preceq T \implies T = PrimT \ x$   
 <proof>

**lemma** *widen-PrimT2-strong*:  $G \vdash S \preceq PrimT \ x \implies S = PrimT \ x$   
 <proof>

Specialized versions for booleans also would work for real Java

**lemma** *widen-Boolean*:  $G \vdash PrimT \ Boolean \preceq T \implies T = PrimT \ Boolean$   
 <proof>

**lemma** *widen-Boolean2*:  $G \vdash S \preceq PrimT \ Boolean \implies S = PrimT \ Boolean$   
 <proof>

**lemma** *widen-RefT*:  $G \vdash RefT \ R \preceq T \implies \exists t. \ T = RefT \ t$   
 <proof>

**lemma** *widen-RefT2*:  $G \vdash S \preceq RefT \ R \implies \exists t. \ S = RefT \ t$   
 <proof>

**lemma** *widen-Iface*:  $G \vdash Iface \ I \preceq T \implies T = Class \ Object \vee (\exists J. \ T = Iface \ J)$   
 <proof>

**lemma** *widen-Iface2*:  $G \vdash S \preceq Iface \ J \implies S = NT \vee (\exists I. \ S = Iface \ I) \vee (\exists D. \ S = Class \ D)$   
 <proof>

**lemma** *widen-Iface-Iface*:  $G \vdash Iface \ I \preceq Iface \ J \implies G \vdash I \preceq I \ J$   
 <proof>

**lemma** *widen-Iface-Iface-eq* [simp]:  $G \vdash Iface \ I \preceq Iface \ J = G \vdash I \preceq I \ J$   
 <proof>

**lemma** *widen-Class*:  $G \vdash \text{Class } C \preceq T \implies (\exists D. T = \text{Class } D) \vee (\exists I. T = \text{Iface } I)$   
 $\langle \text{proof} \rangle$

**lemma** *widen-Class2*:  $G \vdash S \preceq \text{Class } C \implies C = \text{Object} \vee S = NT \vee (\exists D. S = \text{Class } D)$   
 $\langle \text{proof} \rangle$

**lemma** *widen-Class-Class*:  $G \vdash \text{Class } C \preceq \text{Class } cm \implies G \vdash C \preceq_C cm$   
 $\langle \text{proof} \rangle$

**lemma** *widen-Class-Class-eq [simp]*:  $G \vdash \text{Class } C \preceq \text{Class } cm = G \vdash C \preceq_C cm$   
 $\langle \text{proof} \rangle$

**lemma** *widen-Class-Iface*:  $G \vdash \text{Class } C \preceq \text{Iface } I \implies G \vdash C \rightsquigarrow I$   
 $\langle \text{proof} \rangle$

**lemma** *widen-Class-Iface-eq [simp]*:  $G \vdash \text{Class } C \preceq \text{Iface } I = G \vdash C \rightsquigarrow I$   
 $\langle \text{proof} \rangle$

**lemma** *widen-Array*:  $G \vdash S.\square \preceq T \implies T = \text{Class Object} \vee (\exists T'. T = T'.\square \wedge G \vdash S \preceq T')$   
 $\langle \text{proof} \rangle$

**lemma** *widen-Array2*:  $G \vdash S \preceq T.\square \implies S = NT \vee (\exists S'. S = S'.\square \wedge G \vdash S' \preceq T)$   
 $\langle \text{proof} \rangle$

**lemma** *widen-ArrayPrimT*:  $G \vdash \text{PrimT } t.\square \preceq T \implies T = \text{Class Object} \vee T = \text{PrimT } t.\square$   
 $\langle \text{proof} \rangle$

**lemma** *widen-ArrayRefT*:  
 $G \vdash \text{RefT } t.\square \preceq T \implies T = \text{Class Object} \vee (\exists s. T = \text{RefT } s.\square \wedge G \vdash \text{RefT } t \preceq \text{RefT } s)$   
 $\langle \text{proof} \rangle$

**lemma** *widen-ArrayRefT-ArrayRefT-eq [simp]*:  
 $G \vdash \text{RefT } T.\square \preceq \text{RefT } T'.\square = G \vdash \text{RefT } T \preceq \text{RefT } T'$   
 $\langle \text{proof} \rangle$

**lemma** *widen-Array-Array*:  $G \vdash T.\square \preceq T'.\square \implies G \vdash T \preceq T'$   
 $\langle \text{proof} \rangle$

**lemma** *widen-Array-Class*:  $G \vdash S.\square \preceq \text{Class } C \implies C = \text{Object}$   
 $\langle \text{proof} \rangle$

**lemma** *widen-NT2*:  $G \vdash S \preceq NT \implies S = NT$

$\langle \text{proof} \rangle$

**lemma** *widen-Object*:  $\llbracket \text{isrtype } G \ T; \text{ws-prog } G \rrbracket \implies G \vdash \text{RefT } T \preceq \text{Class } \text{Object}$   
 $\langle \text{proof} \rangle$

**lemma** *widen-trans-lemma*  $[\text{rule-format } (\text{no-asm})]$ :  
 $\llbracket G \vdash S \preceq U; \forall C. \text{is-class } G \ C \longrightarrow G \vdash C \preceq_C \text{Object} \rrbracket \implies \forall T. G \vdash U \preceq T \longrightarrow G \vdash S \preceq T$   
 $\langle \text{proof} \rangle$

**lemma** *ws-widen-trans*:  $\llbracket G \vdash S \preceq U; G \vdash U \preceq T; \text{ws-prog } G \rrbracket \implies G \vdash S \preceq T$   
 $\langle \text{proof} \rangle$

**lemma** *widen-antisym-lemma*  $[\text{rule-format } (\text{no-asm})]$ :  $\llbracket G \vdash S \preceq T;$   
 $\forall I \ J. G \vdash I \preceq I \ J \wedge G \vdash J \preceq I \ I \longrightarrow I = J;$   
 $\forall C \ D. G \vdash C \preceq_C D \wedge G \vdash D \preceq_C C \longrightarrow C = D;$   
 $\forall I. G \vdash \text{Object} \rightsquigarrow I \longrightarrow \text{False} \rrbracket \implies G \vdash T \preceq S \longrightarrow S = T$   
 $\langle \text{proof} \rangle$

**lemmas** *subint-antisym* =  
 $\text{subint1-acyclic } [\text{THEN } \text{acyclic-impl-antisym-rtrancl, standard}]$   
**lemmas** *subcls-antisym* =  
 $\text{subcls1-acyclic } [\text{THEN } \text{acyclic-impl-antisym-rtrancl, standard}]$

**lemma** *widen-antisym*:  $\llbracket G \vdash S \preceq T; G \vdash T \preceq S; \text{ws-prog } G \rrbracket \implies S = T$   
 $\langle \text{proof} \rangle$

**lemma** *widen-ObjectD*  $[\text{dest!}]$ :  $G \vdash \text{Class } \text{Object} \preceq T \implies T = \text{Class } \text{Object}$   
 $\langle \text{proof} \rangle$

**constdefs**  
 $\text{widens} :: \text{prog} \Rightarrow [\text{ty list}, \text{ty list}] \Rightarrow \text{bool } (-\vdash-[\preceq]- [\text{?1}, \text{?1}, \text{?1}] \ \text{?0})$   
 $G \vdash Ts[\preceq] Ts' \equiv \text{list-all2 } (\lambda T \ T'. G \vdash T \preceq T') \ Ts \ Ts'$

**lemma** *widens-Nil*  $[\text{simp}]$ :  $G \vdash [][\preceq] []$   
 $\langle \text{proof} \rangle$

**lemma** *widens-Cons*  $[\text{simp}]$ :  $G \vdash (S \# Ss)[\preceq] (T \# Ts) = (G \vdash S \preceq T \wedge G \vdash Ss[\preceq] Ts)$   
 $\langle \text{proof} \rangle$

## narrowing relation

### inductive narrow $G$ intros

*subcls*:  $G \vdash C \preceq_C D \implies G \vdash \text{Class } D \succ \text{Class } C$   
*implmt*:  $\neg G \vdash C \rightsquigarrow I \implies G \vdash \text{Class } C \succ \text{Iface } I$   
*obj-arr*:  $G \vdash \text{Class } \text{Object} \succ T. []$   
*int-cls*:  $G \vdash \text{Iface } I \succ \text{Class } C$   
*subint*:  $\text{imethds } G \ I \text{ hidings } \text{imethds } G \ J \text{ entails}$   
 $(\lambda (md, mh) (md', mh'). G \vdash \text{mrt } mh \preceq \text{mrt } mh') \implies$   
 $\neg G \vdash I \preceq I \ J \implies G \vdash \text{Iface } I \succ \text{Iface } J$   
*array*:  $G \vdash \text{RefT } S \succ \text{RefT } T \implies G \vdash \text{RefT } S. [] \succ \text{RefT } T. []$



**lemma** *narrow-RefT*:  $G \vdash \text{RefT } R \succ T \implies \exists t. T = \text{RefT } t$   
 $\langle \text{proof} \rangle$

**lemma** *narrow-RefT2*:  $G \vdash S \succ \text{RefT } R \implies \exists t. S = \text{RefT } t$   
 $\langle \text{proof} \rangle$

**lemma** *narrow-PrimT*:  $G \vdash \text{PrimT } pt \succ T \implies \exists t. T = \text{PrimT } t$   
 $\langle \text{proof} \rangle$

**lemma** *narrow-PrimT2*:  $G \vdash S \succ \text{PrimT } pt \implies$   
 $\exists t. S = \text{PrimT } t \wedge G \vdash \text{PrimT } t \preceq \text{PrimT } pt$   
 $\langle \text{proof} \rangle$

These narrowing lemmata hold in Bali but are too strong for ordinary Java. They would not work for real Java Integral Types, like short, long, int. These lemmata are just for documentation and are not used.

**lemma** *narrow-PrimT-strong*:  $G \vdash \text{PrimT } pt \succ T \implies T = \text{PrimT } pt$   
 $\langle \text{proof} \rangle$

**lemma** *narrow-PrimT2-strong*:  $G \vdash S \succ \text{PrimT } pt \implies S = \text{PrimT } pt$   
 $\langle \text{proof} \rangle$

Specialized versions for booleans also would work for real Java

**lemma** *narrow-Boolean*:  $G \vdash \text{PrimT } \text{Boolean} \succ T \implies T = \text{PrimT } \text{Boolean}$   
 $\langle \text{proof} \rangle$

**lemma** *narrow-Boolean2*:  $G \vdash S \succ \text{PrimT } \text{Boolean} \implies S = \text{PrimT } \text{Boolean}$   
 $\langle \text{proof} \rangle$

## casting relation

**inductive** *cast* *G intros* — casting conversion, cf. 5.5

*widen*:  $G \vdash S \preceq T \implies G \vdash S \preceq^? T$   
*narrow*:  $G \vdash S \succ T \implies G \vdash S \preceq^? T$

**lemma** *cast-RefT*:  $G \vdash \text{RefT } R \preceq^? T \implies \exists t. T = \text{RefT } t$   
 $\langle \text{proof} \rangle$

**lemma** *cast-RefT2*:  $G \vdash S \preceq^? \text{RefT } R \implies \exists t. S = \text{RefT } t$   
 $\langle \text{proof} \rangle$

**lemma** *cast-PrimT*:  $G \vdash \text{PrimT } pt \preceq^? T \implies \exists t. T = \text{PrimT } t$   
 $\langle \text{proof} \rangle$

**lemma** *cast-PrimT2*:  $G \vdash S \preceq^? \text{PrimT } pt \implies \exists t. S = \text{PrimT } t \wedge G \vdash \text{PrimT } t \preceq \text{PrimT } pt$   
 $\langle \text{proof} \rangle$

**lemma** *cast-Boolean*:  
**assumes** *bool-cast*:  $G \vdash \text{PrimT } \text{Boolean} \preceq^? T$   
**shows**  $T = \text{PrimT } \text{Boolean}$   
 $\langle \text{proof} \rangle$

**lemma** *cast-Boolean2*:  
**assumes** *bool-cast*:  $G \vdash S \preceq^? \text{PrimT } \text{Boolean}$   
**shows**  $S = \text{PrimT } \text{Boolean}$   
 $\langle \text{proof} \rangle$

**end**

## Chapter 10

# DeclConcepts

## 16 Advanced concepts on Java declarations like overriding, inheritance, dynamic method lookup

theory *DeclConcepts* imports *TypeRel* begin

access control (cf. 6.6), overriding and hiding (cf. 8.4.6.1)

**constdefs**

*is-public* :: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  *bool*  
*is-public* *G* *qn*  $\equiv$  (case class *G* *qn* of  
     None  $\Rightarrow$  (case iface *G* *qn* of  
         None  $\Rightarrow$  False  
         | Some iface  $\Rightarrow$  access iface = Public)  
     | Some class  $\Rightarrow$  access class = Public)

## 17 accessibility of types (cf. 6.6.1)

Primitive types are always accessible, interfaces and classes are accessible in their package or if they are defined public, an array type is accessible if its element type is accessible

**consts** *accessible-in* :: *prog*  $\Rightarrow$  *ty*  $\Rightarrow$  *pname*  $\Rightarrow$  *bool*  
     (-  $\vdash$  - *accessible'-in* - [61,61,61] 60)  
     *rt-accessible-in*:: *prog*  $\Rightarrow$  *ref-ty*  $\Rightarrow$  *pname*  $\Rightarrow$  *bool*  
     (-  $\vdash$  - *accessible'-in'* - [61,61,61] 60)

**primrec**

$G \vdash (\text{PrimT } p) \text{ accessible-in pack} = \text{True}$   
*accessible-in-RefT-simp*:  
 $G \vdash (\text{RefT } r) \text{ accessible-in pack} = G \vdash r \text{ accessible-in' pack}$   
  
 $G \vdash (\text{NullT}) \text{ accessible-in' pack} = \text{True}$   
 $G \vdash (\text{IfaceT } I) \text{ accessible-in' pack} = ((\text{pid } I = \text{pack}) \vee \text{is-public } G \ I)$   
 $G \vdash (\text{ClassT } C) \text{ accessible-in' pack} = ((\text{pid } C = \text{pack}) \vee \text{is-public } G \ C)$   
 $G \vdash (\text{ArrayT } ty) \text{ accessible-in' pack} = G \vdash ty \text{ accessible-in pack}$

**declare** *accessible-in-RefT-simp* [*simp del*]

**constdefs**

*is-acc-class* :: *prog*  $\Rightarrow$  *pname*  $\Rightarrow$  *qname*  $\Rightarrow$  *bool*  
*is-acc-class* *G* *P* *C*  $\equiv$  *is-class* *G* *C*  $\wedge$   $G \vdash (\text{Class } C) \text{ accessible-in } P$   
*is-acc-iface* :: *prog*  $\Rightarrow$  *pname*  $\Rightarrow$  *qname*  $\Rightarrow$  *bool*  
*is-acc-iface* *G* *P* *I*  $\equiv$  *is-iface* *G* *I*  $\wedge$   $G \vdash (\text{Iface } I) \text{ accessible-in } P$   
*is-acc-type* :: *prog*  $\Rightarrow$  *pname*  $\Rightarrow$  *ty*  $\Rightarrow$  *bool*  
*is-acc-type* *G* *P* *T*  $\equiv$  *is-type* *G* *T*  $\wedge$   $G \vdash T \text{ accessible-in } P$   
*is-acc-reftype* :: *prog*  $\Rightarrow$  *pname*  $\Rightarrow$  *ref-ty*  $\Rightarrow$  *bool*  
*is-acc-reftype* *G* *P* *T*  $\equiv$  *isrtype* *G* *T*  $\wedge$   $G \vdash T \text{ accessible-in' } P$

**lemma** *is-acc-classD*:

*is-acc-class* *G* *P* *C*  $\Longrightarrow$  *is-class* *G* *C*  $\wedge$   $G \vdash (\text{Class } C) \text{ accessible-in } P$   
 <proof>

**lemma** *is-acc-class-is-class*: *is-acc-class* *G* *P* *C*  $\Longrightarrow$  *is-class* *G* *C*

<proof>

**lemma** *is-acc-ifaceD*:

*is-acc-iface* *G* *P* *I*  $\Longrightarrow$  *is-iface* *G* *I*  $\wedge$   $G \vdash (\text{Iface } I) \text{ accessible-in } P$   
 <proof>

**lemma** *is-acc-typeD*:  
*is-acc-type*  $G P T \implies is-type\ G\ T \ \wedge\ G \vdash T\ accessible-in\ P$   
 $\langle proof \rangle$

**lemma** *is-acc-reftypeD*:  
*is-acc-reftype*  $G P T \implies isrtype\ G\ T \ \wedge\ G \vdash T\ accessible-in'\ P$   
 $\langle proof \rangle$

## 18 accessibility of members

The accessibility of members is more involved as the accessibility of types. We have to distinguish several cases to model the different effects of accessibility during inheritance, overriding and ordinary member access

### Various technical conversion and selection functions

overloaded selector *accmodi* to select the access modifier out of various HOL types

**axclass** *has-accmodi* < type  
**consts** *accmodi*:: '*a*::*has-accmodi*  $\Rightarrow$  *acc-modi*

**instance** *acc-modi*::*has-accmodi*  $\langle proof \rangle$

**defs** (overloaded)  
*acc-modi-accmodi-def*: *accmodi* (*a*::*acc-modi*)  $\equiv a$

**lemma** *acc-modi-accmodi-simp*[*simp*]: *accmodi* (*a*::*acc-modi*) = *a*  
 $\langle proof \rangle$

**instance** *decl-ext-type*:: (*type*) *has-accmodi*  $\langle proof \rangle$

**defs** (overloaded)  
*decl-acc-modi-def*: *accmodi* (*d*::('a:: *type*) *decl-scheme*)  $\equiv access\ d$

**lemma** *decl-acc-modi-simp*[*simp*]: *accmodi* (*d*::('a::*type*) *decl-scheme*) = *access d*  
 $\langle proof \rangle$

**instance** \* :: (*type*,*has-accmodi*) *has-accmodi*  $\langle proof \rangle$

**defs** (overloaded)  
*pair-acc-modi-def*: *accmodi* *p*  $\equiv (accmodi\ (snd\ p))$

**lemma** *pair-acc-modi-simp*[*simp*]: *accmodi* (*x*,*a*) = (*accmodi a*)  
 $\langle proof \rangle$

**instance** *memberdecl* :: *has-accmodi*  $\langle proof \rangle$

**defs** (overloaded)  
*memberdecl-acc-modi-def*: *accmodi* *m*  $\equiv$  (case *m* of  
     *fdecl f*  $\Rightarrow accmodi\ f$   
     | *mdecl m*  $\Rightarrow accmodi\ m$ )

**lemma** *memberdecl-fdecl-acc-modi-simp*[simp]:  
 $\text{accmodi } (fdecl\ m) = \text{accmodi } m$   
 ⟨proof⟩

**lemma** *memberdecl-mdecl-acc-modi-simp*[simp]:  
 $\text{accmodi } (mdecl\ m) = \text{accmodi } m$   
 ⟨proof⟩

overloaded selector *declclass* to select the declaring class out of various HOL types

**axclass** *has-declclass* < type  
**consts** *declclass*:: 'a::has-declclass  $\Rightarrow$  qname

**instance** *qname-ext-type*::(type) *has-declclass* ⟨proof⟩

**defs** (overloaded)  
*qname-declclass-def*:  $\text{declclass } (q::\text{qname}) \equiv q$

**lemma** *qname-declclass-simp*[simp]:  $\text{declclass } (q::\text{qname}) = q$   
 ⟨proof⟩

**instance** \* :: (has-declclass,type) *has-declclass* ⟨proof⟩

**defs** (overloaded)  
*pair-declclass-def*:  $\text{declclass } p \equiv \text{declclass } (\text{fst } p)$

**lemma** *pair-declclass-simp*[simp]:  $\text{declclass } (c,x) = \text{declclass } c$   
 ⟨proof⟩

overloaded selector *is-static* to select the static modifier out of various HOL types

**axclass** *has-static* < type  
**consts** *is-static* :: 'a::has-static  $\Rightarrow$  bool

**instance** *decl-ext-type* :: (has-static) *has-static* ⟨proof⟩

**defs** (overloaded)  
*decl-is-static-def*:  
 $\text{is-static } (m::('a::\text{has-static})\ \text{decl-scheme}) \equiv \text{is-static } (\text{Decl.decl.more } m)$

**instance** *member-ext-type* :: (type) *has-static* ⟨proof⟩

**defs** (overloaded)  
*static-field-type-is-static-def*:  
 $\text{is-static } (m::('b::\text{type})\ \text{member-ext-type}) \equiv \text{static-sel } m$

**lemma** *member-is-static-simp*:  $\text{is-static } (m::'a\ \text{member-scheme}) = \text{static } m$   
 ⟨proof⟩

**instance** \* :: (type,has-static) *has-static* ⟨proof⟩

**defs** (overloaded)  
*pair-is-static-def*:  $\text{is-static } p \equiv \text{is-static } (\text{snd } p)$

**lemma** *pair-is-static-simp* [simp]:  $\text{is-static } (x,s) = \text{is-static } s$



**lemma** *mbr-simp*[simp]:  $mbr\ (C,m) = m$   
 $\langle proof \rangle$

**lemma** *access-mbr-simp* [simp]:  $(accmodi\ (mbr\ m)) = accmodi\ m$   
 $\langle proof \rangle$

**lemma** *mthd-simp*[simp]:  $mthd\ (C,m) = m$   
 $\langle proof \rangle$

**lemma** *fld-simp*[simp]:  $fld\ (C,f) = f$   
 $\langle proof \rangle$

**lemma** *accmodi-simp*[simp]:  $accmodi\ (C,m) = access\ m$   
 $\langle proof \rangle$

**lemma** *access-mthd-simp* [simp]:  $(access\ (mthd\ m)) = accmodi\ m$   
 $\langle proof \rangle$

**lemma** *access-fld-simp* [simp]:  $(access\ (fld\ f)) = accmodi\ f$   
 $\langle proof \rangle$

**lemma** *static-mthd-simp*[simp]:  $static\ (mthd\ m) = is-static\ m$   
 $\langle proof \rangle$

**lemma** *mthd-is-static-simp* [simp]:  $is-static\ (mthd\ m) = is-static\ m$   
 $\langle proof \rangle$

**lemma** *static-fld-simp*[simp]:  $static\ (fld\ f) = is-static\ f$   
 $\langle proof \rangle$

**lemma** *ext-field-simp* [simp]:  $(declclass\ f,fld\ f) = f$   
 $\langle proof \rangle$

**lemma** *ext-method-simp* [simp]:  $(declclass\ m,mthd\ m) = m$   
 $\langle proof \rangle$

**lemma** *ext-mbr-simp* [simp]:  $(declclass\ m,mbr\ m) = m$   
 $\langle proof \rangle$

**lemma** *fname-simp*[simp]:  $fname\ (n,c) = n$   
 $\langle proof \rangle$

**lemma** *declclassf-simp*[simp]:  $declclassf\ (n,c) = c$   
 $\langle proof \rangle$



**constdefs** — some mnemonic selectors for  $(vname \times qname)$

$fldname :: (vname \times qname) \Rightarrow vname$

$fldname \equiv fst$

$fldclass :: (vname \times qname) \Rightarrow qname$

$fldclass \equiv snd$

**lemma** *fldname-simp*[simp]:  $fldname\ (n,c) = n$

*<proof>*

**lemma** *fldclass-simp*[simp]:  $fldclass\ (n,c) = c$

*<proof>*

**lemma** *ext-fldname-simp*[simp]:  $(fldname\ f, fldclass\ f) = f$

*<proof>*

Convert a qualified method declaration (qualified with its declaring class) to a qualified member declaration: *methdMembr*

**constdefs**

$methdMembr :: (qname \times mdecl) \Rightarrow (qname \times memberdecl)$

$methdMembr\ m \equiv (fst\ m, mdecl\ (snd\ m))$

**lemma** *methdMembr-simp*[simp]:  $methdMembr\ (c,m) = (c, mdecl\ m)$

*<proof>*

**lemma** *accmodi-methdMembr-simp*[simp]:  $accmodi\ (methdMembr\ m) = accmodi\ m$

*<proof>*

**lemma** *is-static-methdMembr-simp*[simp]:  $is-static\ (methdMembr\ m) = is-static\ m$

*<proof>*

**lemma** *declclass-methdMembr-simp*[simp]:  $declclass\ (methdMembr\ m) = declclass\ m$

*<proof>*

Convert a qualified method (qualified with its declaring class) to a qualified member declaration: *method*

**constdefs**

$method :: sig \Rightarrow (qname \times methd) \Rightarrow (qname \times memberdecl)$

$method\ sig\ m \equiv (declclass\ m, mdecl\ (sig, mthd\ m))$

**lemma** *method-simp*[simp]:  $method\ sig\ (C,m) = (C, mdecl\ (sig, m))$

*<proof>*

**lemma** *accmodi-method-simp*[simp]:  $accmodi\ (method\ sig\ m) = accmodi\ m$

*<proof>*

**lemma** *declclass-method-simp*[simp]:  $declclass\ (method\ sig\ m) = declclass\ m$

*<proof>*

**lemma** *is-static-method-simp[simp]*: *is-static* (method sig m) = *is-static* m  
 ⟨proof⟩

**lemma** *mbr-method-simp[simp]*: *mbr* (method sig m) = *mdecl* (sig, mthd m)  
 ⟨proof⟩

**lemma** *memberid-method-simp[simp]*: *memberid* (method sig m) = *mid* sig  
 ⟨proof⟩

**constdefs**

*fieldm* :: *vname*  $\Rightarrow$  (*qtname*  $\times$  *field*)  $\Rightarrow$  (*qtname*  $\times$  *memberdecl*)  
*fieldm* n f  $\equiv$  (*declclass* f, *fdecl* (n, fld f))

**lemma** *fieldm-simp[simp]*: *fieldm* n (C, f) = (C, *fdecl* (n, f))  
 ⟨proof⟩

**lemma** *accmodi-fieldm-simp[simp]*: *accmodi* (*fieldm* n f) = *accmodi* f  
 ⟨proof⟩

**lemma** *declclass-fieldm-simp[simp]*: *declclass* (*fieldm* n f) = *declclass* f  
 ⟨proof⟩

**lemma** *is-static-fieldm-simp[simp]*: *is-static* (*fieldm* n f) = *is-static* f  
 ⟨proof⟩

**lemma** *mbr-fieldm-simp[simp]*: *mbr* (*fieldm* n f) = *fdecl* (n, fld f)  
 ⟨proof⟩

**lemma** *memberid-fieldm-simp[simp]*: *memberid* (*fieldm* n f) = *fid* n  
 ⟨proof⟩

Select the signature out of a qualified method declaration: *msig*

**constdefs** *msig*:: (*qtname*  $\times$  *mdecl*)  $\Rightarrow$  *sig*  
*msig* m  $\equiv$  *fst* (*snd* m)

**lemma** *msig-simp[simp]*: *msig* (c, (s, m)) = s  
 ⟨proof⟩

Convert a qualified method (qualified with its declaring class) to a qualified method declaration:  
*qmdecl*

**constdefs** *qmdecl* :: *sig*  $\Rightarrow$  (*qtname*  $\times$  *methd*)  $\Rightarrow$  (*qtname*  $\times$  *mdecl*)  
*qmdecl* sig m  $\equiv$  (*declclass* m, (sig, mthd m))

**lemma** *qmdecl-simp[simp]*: *qmdecl* sig (C, m) = (C, (sig, m))  
 ⟨proof⟩

**lemma** *declclass-qmdecl-simp*[simp]: *declclass* (*qmdecl sig m*) = *declclass m*  
 ⟨proof⟩

**lemma** *accmodi-qmdecl-simp*[simp]: *accmodi* (*qmdecl sig m*) = *accmodi m*  
 ⟨proof⟩

**lemma** *is-static-qmdecl-simp*[simp]: *is-static* (*qmdecl sig m*) = *is-static m*  
 ⟨proof⟩

**lemma** *msig-qmdecl-simp*[simp]: *msig* (*qmdecl sig m*) = *sig*  
 ⟨proof⟩

**lemma** *mdecl-qmdecl-simp*[simp]:  
*mdecl* (*mthd* (*qmdecl sig new*)) = *mdecl* (*sig*, *mthd new*)  
 ⟨proof⟩

**lemma** *methdMembr-qmdecl-simp* [simp]:  
*methdMembr* (*qmdecl sig old*) = *method sig old*  
 ⟨proof⟩

overloaded selector *resTy* to select the result type out of various HOL types

**axclass** *has-resTy* < *type*  
**consts** *resTy*:: 'a::has-resTy  $\Rightarrow$  *ty*

**instance** *decl-ext-type* :: (*has-resTy*) *has-resTy* ⟨proof⟩

**defs** (overloaded)  
*decl-resTy-def*:  
*resTy* (*m*::('a::has-resTy) *decl-scheme*)  $\equiv$  *resTy* (*Decl.decl.more m*)

**instance** *member-ext-type* :: (*has-resTy*) *has-resTy* ⟨proof⟩

**defs** (overloaded)  
*member-ext-type-resTy-def*:  
*resTy* (*m*::('b::has-resTy) *member-ext-type*)  
 $\equiv$  *resTy* (*member.more-sel m*)

**instance** *mhead-ext-type* :: (*type*) *has-resTy* ⟨proof⟩

**defs** (overloaded)  
*mhead-ext-type-resTy-def*:  
*resTy* (*m*::('b *mhead-ext-type*))  
 $\equiv$  *resT-sel m*

**lemma** *mhead-resTy-simp*: *resTy* (*m*::'a *mhead-scheme*) = *resT m*  
 ⟨proof⟩

**lemma** *resTy-mhead* [simp]: *resTy* (*mhead m*) = *resTy m*  
 ⟨proof⟩

**instance** \* :: (*type*, *has-resTy*) *has-resTy* ⟨proof⟩

**defs (overloaded)**

*pair-resTy-def*:  $\text{resTy } p \equiv \text{resTy } (\text{snd } p)$

**lemma** *pair-resTy-simp*[*simp*]:  $\text{resTy } (x, m) = \text{resTy } m$   
*<proof>*

**lemma** *qmdecl-resTy-simp* [*simp*]:  $\text{resTy } (\text{qmdecl sig } m) = \text{resTy } m$   
*<proof>*

**lemma** *resTy-mthd* [*simp*]:  $\text{resTy } (\text{mthd } m) = \text{resTy } m$   
*<proof>*

**inheritable-in**

$G \vdash m$  *inheritable-in*  $P$ :  $m$  can be inherited by classes in package  $P$  if:

- the declaration class of  $m$  is accessible in  $P$  and
- the member  $m$  is declared with protected or public access or if it is declared with default (package) access, the package of the declaration class of  $m$  is also  $P$ . If the member  $m$  is declared with private access it is not accessible for inheritance at all.

**constdefs**

*inheritable-in*::

$\text{prog} \Rightarrow (\text{qname} \times \text{memberdecl}) \Rightarrow \text{pname} \Rightarrow \text{bool}$   
 $(- \vdash - \text{inheritable}'\text{-in} - [61, 61, 61] \ 60)$

$G \vdash \text{membr}$  *inheritable-in* *pack*

$\equiv (\text{case } (\text{accmodi } \text{membr}) \text{ of}$   
 $\quad \text{Private} \Rightarrow \text{False}$   
 $\quad | \text{Package} \Rightarrow (\text{pid } (\text{declclass } \text{membr})) = \text{pack}$   
 $\quad | \text{Protected} \Rightarrow \text{True}$   
 $\quad | \text{Public} \Rightarrow \text{True})$

**syntax**

*Method-inheritable-in*::

$\text{prog} \Rightarrow (\text{qname} \times \text{mdecl}) \Rightarrow \text{pname} \Rightarrow \text{bool}$   
 $(- \vdash \text{Method} - \text{inheritable}'\text{-in} - [61, 61, 61] \ 60)$

**translations**

$G \vdash \text{Method } m$  *inheritable-in*  $p == G \vdash \text{methdMembr } m$  *inheritable-in*  $p$

**syntax**

*Methd-inheritable-in*::

$\text{prog} \Rightarrow \text{sig} \Rightarrow (\text{qname} \times \text{methd}) \Rightarrow \text{pname} \Rightarrow \text{bool}$   
 $(- \vdash \text{Methd} - - \text{inheritable}'\text{-in} - [61, 61, 61, 61] \ 60)$

**translations**

$G \vdash \text{Methd } s \ m$  *inheritable-in*  $p == G \vdash (\text{method } s \ m)$  *inheritable-in*  $p$

**declared-in/undeclared-in**

**constdefs** *cdeclaredmethd*::  $\text{prog} \Rightarrow \text{qname} \Rightarrow (\text{sig}, \text{methd}) \text{ table}$

*cdeclaredmethd*  $G \ C$

$\equiv (\text{case class } G \ C \text{ of}$   
 $\quad \text{None} \Rightarrow \lambda \text{ sig. None}$

| *Some c*  $\Rightarrow$  *table-of (methods c)*  
 )

**constdefs**

*cdeclaredfield*:: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  (*vname,field*) *table*

*cdeclaredfield G C*

$\equiv$  (*case class G C of*  
     *None*  $\Rightarrow \lambda$  *sig. None*  
     | *Some c*  $\Rightarrow$  *table-of (cfields c)*  
 )

**constdefs**

*declared-in*:: *prog*  $\Rightarrow$  *memberdecl*  $\Rightarrow$  *qname*  $\Rightarrow$  *bool*

( $\vdash$  - *declared'-in* - [61,61,61] 60)

$G \vdash m$  *declared-in C*  $\equiv$  (*case m of*

*fdecl (fn,f)*  $\Rightarrow$  *cdeclaredfield G C fn = Some f*  
     | *mdecl (sig,m)*  $\Rightarrow$  *cdeclaredmethd G C sig = Some m*)

**syntax**

*method-declared-in*:: *prog*  $\Rightarrow$  (*qname*  $\times$  *mdecl*)  $\Rightarrow$  *qname*  $\Rightarrow$  *bool*

( $\vdash$  *Method* - *declared'-in* - [61,61,61] 60)

**translations**

$G \vdash \text{Method } m \text{ declared-in } C == G \vdash mdecl (mthd m) \text{ declared-in } C$

**syntax**

*methd-declared-in*:: *prog*  $\Rightarrow$  *sig*  $\Rightarrow$  (*qname*  $\times$  *methd*)  $\Rightarrow$  *qname*  $\Rightarrow$  *bool*

( $\vdash$  *Methd* - - *declared'-in* - [61,61,61,61] 60)

**translations**

$G \vdash \text{Methd } s m \text{ declared-in } C == G \vdash mdecl (s, mthd m) \text{ declared-in } C$

**lemma** *declared-in-classD*:

$G \vdash m \text{ declared-in } C \implies \text{is-class } G C$

$\langle \text{proof} \rangle$

**constdefs**

*undeclared-in*:: *prog*  $\Rightarrow$  *memberid*  $\Rightarrow$  *qname*  $\Rightarrow$  *bool*

( $\vdash$  - *undeclared'-in* - [61,61,61] 60)

$G \vdash m$  *undeclared-in C*  $\equiv$  (*case m of*

*fid fn*  $\Rightarrow$  *cdeclaredfield G C fn = None*  
     | *mid sig*  $\Rightarrow$  *cdeclaredmethd G C sig = None*)

**members****consts**

*members*:: *prog*  $\Rightarrow$  (*qname*  $\times$  (*qname*  $\times$  *memberdecl*)) *set*

**syntax**

*member-of*:: *prog*  $\Rightarrow$  (*qname*  $\times$  *memberdecl*)  $\Rightarrow$  *qname*  $\Rightarrow$  *bool*

( $\vdash$  - *member'-of* - [61,61,61] 60)

**translations**

$G \vdash m \text{ member-of } C \Leftrightarrow (C, m) \in \text{members } G$

**inductive members G intros**

*Immediate:*  $\llbracket G \vdash \text{mbr } m \text{ declared-in } C; \text{declclass } m = C \rrbracket \implies G \vdash m \text{ member-of } C$

*Inherited:*  $\llbracket G \vdash m \text{ inheritable-in } (\text{pid } C); G \vdash \text{memberid } m \text{ undeclared-in } C;$   
 $G \vdash C \prec_{C_1} S; G \vdash (\text{Class } S) \text{ accessible-in } (\text{pid } C); G \vdash m \text{ member-of } S$   
 $\rrbracket \implies G \vdash m \text{ member-of } C$

Note that in the case of an inherited member only the members of the direct superclass are concerned. If a member of a superclass of the direct superclass isn't inherited in the direct superclass (not member of the direct superclass) than it can't be a member of the class. E.g. If a member of a class A is defined with package access it isn't member of a subclass S if S isn't in the same package as A. Any further subclasses of S will not inherit the member, regardless if they are in the same package as A or not.

#### **syntax**

*method-member-of::*  $\text{prog} \Rightarrow (qname \times mdecl) \Rightarrow qname \Rightarrow \text{bool}$   
 $(- \vdash \text{Method} - \text{member'-of} - [61,61,61] \ 60)$

#### **translations**

$G \vdash \text{Method } m \text{ member-of } C \Leftrightarrow G \vdash (\text{methdMembr } m) \text{ member-of } C$

#### **syntax**

*methd-member-of::*  $\text{prog} \Rightarrow sig \Rightarrow (qname \times \text{methd}) \Rightarrow qname \Rightarrow \text{bool}$   
 $(- \vdash \text{Methd} - - \text{member'-of} - [61,61,61,61] \ 60)$

#### **translations**

$G \vdash \text{Methd } s \ m \text{ member-of } C \Leftrightarrow G \vdash (\text{method } s \ m) \text{ member-of } C$

#### **syntax**

*fieldm-member-of::*  $\text{prog} \Rightarrow vname \Rightarrow (qname \times \text{field}) \Rightarrow qname \Rightarrow \text{bool}$   
 $(- \vdash \text{Field} - - \text{member'-of} - [61,61,61] \ 60)$

#### **translations**

$G \vdash \text{Field } n \ f \text{ member-of } C \Leftrightarrow G \vdash \text{fieldm } n \ f \text{ member-of } C$

#### **constdefs**

*inherits::*  $\text{prog} \Rightarrow qname \Rightarrow (qname \times \text{memberdecl}) \Rightarrow \text{bool}$   
 $(- \vdash - \text{inherits} - [61,61,61] \ 60)$

$G \vdash C \text{ inherits } m$

$\equiv G \vdash m \text{ inheritable-in } (\text{pid } C) \wedge G \vdash \text{memberid } m \text{ undeclared-in } C \wedge$   
 $(\exists S. G \vdash C \prec_{C_1} S \wedge G \vdash (\text{Class } S) \text{ accessible-in } (\text{pid } C) \wedge G \vdash m \text{ member-of } S)$

**lemma** *inherits-member:*  $G \vdash C \text{ inherits } m \implies G \vdash m \text{ member-of } C$   
 $\langle \text{proof} \rangle$

**constdefs** *member-in::*  $\text{prog} \Rightarrow (qname \times \text{memberdecl}) \Rightarrow qname \Rightarrow \text{bool}$   
 $(- \vdash - \text{member'-in} - [61,61,61] \ 60)$

$G \vdash m \text{ member-in } C \equiv \exists \text{ provC}. G \vdash C \preceq_C \text{ provC} \wedge G \vdash m \text{ member-of } \text{ provC}$

A member is in a class if it is member of the class or a superclass. If a member is in a class we can select this member. This additional notion is necessary since not all members are inherited to subclasses. So such members are not member-of the subclass but member-in the subclass.

#### **syntax**

*method-member-in::*  $\text{prog} \Rightarrow (qname \times mdecl) \Rightarrow qname \Rightarrow \text{bool}$   
 $(- \vdash \text{Method} - \text{member'-in} - [61,61,61] \ 60)$

#### **translations**

$G \vdash \text{Method } m \text{ member-in } C \Leftrightarrow G \vdash (\text{methdMembr } m) \text{ member-in } C$

**syntax**

*methd-member-in*::  $prog \Rightarrow sig \Rightarrow (qname \times methd) \Rightarrow qname \Rightarrow bool$   
 $(- \vdash Methd - - member'-in - [61,61,61,61] 60)$

**translations**

$G \vdash Methd s m member-in C \Leftrightarrow G \vdash (method s m) member-in C$

**consts** *stat-overridesR*::

$prog \Rightarrow ((qname \times mdecl) \times (qname \times mdecl)) set$

**lemma** *member-inD*:  $G \vdash m member-in C$

$\Rightarrow \exists provC. G \vdash C \preceq_C provC \wedge G \vdash m member-of provC$   
 $\langle proof \rangle$

**lemma** *member-inI*:  $\llbracket G \vdash m member-of provC; G \vdash C \preceq_C provC \rrbracket \Rightarrow G \vdash m member-in C$

$\langle proof \rangle$

**lemma** *member-of-to-member-in*:  $G \vdash m member-of C \Rightarrow G \vdash m member-in C$

$\langle proof \rangle$

**overriding**

Unfortunately the static notion of overriding (used during the typecheck of the compiler) and the dynamic notion of overriding (used during execution in the JVM) are not exactly the same.

Static overriding (used during the typecheck of the compiler)

**syntax**

*stat-overrides*::  $prog \Rightarrow (qname \times mdecl) \Rightarrow (qname \times mdecl) \Rightarrow bool$   
 $(- \vdash - overrides_S - [61,61,61] 60)$

**translations**

$G \vdash new overrides_S old == (new, old) \in stat-overridesR G$

**inductive** *stat-overridesR* *G* **intros**

*Direct*:  $\llbracket \neg is-static new; msig new = msig old;$   
 $G \vdash Method new declared-in (declclass new);$   
 $G \vdash Method old declared-in (declclass old);$   
 $G \vdash Method old inheritable-in pid (declclass new);$   
 $G \vdash (declclass new) \prec_{C1} superNew;$   
 $G \vdash Method old member-of superNew$   
 $\rrbracket \Rightarrow G \vdash new overrides_S old$

*Indirect*:  $\llbracket G \vdash new overrides_S inter; G \vdash inter overrides_S old \rrbracket$   
 $\Rightarrow G \vdash new overrides_S old$

Dynamic overriding (used during the typecheck of the compiler)

**consts** *overridesR*::

$prog \Rightarrow ((qname \times mdecl) \times (qname \times mdecl)) set$

*overrides*::  $prog \Rightarrow (qname \times mdecl) \Rightarrow (qname \times mdecl) \Rightarrow bool$   
 $(- \vdash - overrides - [61,61,61] 60)$

**translations**

$G \vdash new overrides old == (new, old) \in overridesR G$

**inductive overridesR G intros**

*Direct:*  $\llbracket \neg \text{is-static new}; \neg \text{is-static old}; \text{accmodi new} \neq \text{Private};$   
 $\text{msig new} = \text{msig old};$   
 $G \vdash (\text{declclass new}) \prec_C (\text{declclass old});$   
 $G \vdash \text{Method new declared-in } (\text{declclass new});$   
 $G \vdash \text{Method old declared-in } (\text{declclass old});$   
 $G \vdash \text{Method old inheritable-in pid } (\text{declclass new});$   
 $G \vdash \text{resTy new} \preceq \text{resTy old}$   
 $\rrbracket \implies G \vdash \text{new overrides old}$

*Indirect:*  $\llbracket G \vdash \text{new overrides inter}; G \vdash \text{inter overrides old} \rrbracket$   
 $\implies G \vdash \text{new overrides old}$

**syntax**

*sig-stat-overrides::*

$\text{prog} \Rightarrow \text{sig} \Rightarrow (qname \times \text{methd}) \Rightarrow (qname \times \text{methd}) \Rightarrow \text{bool}$   
 $(-, \vdash - \text{overrides}_S - [61, 61, 61, 61] \ 60)$

**translations**

$G, s \vdash \text{new overrides}_S \text{ old} \rightarrow G \vdash (q\text{mdecl } s \text{ new}) \text{ overrides}_S (q\text{mdecl } s \text{ old})$

**syntax**

*sig-overrides::*  $\text{prog} \Rightarrow \text{sig} \Rightarrow (qname \times \text{methd}) \Rightarrow (qname \times \text{methd}) \Rightarrow \text{bool}$   
 $(-, \vdash - \text{overrides} - [61, 61, 61, 61] \ 60)$

**translations**

$G, s \vdash \text{new overrides old} \rightarrow G \vdash (q\text{mdecl } s \text{ new}) \text{ overrides } (q\text{mdecl } s \text{ old})$

**Hiding****constdefs hides::**

$\text{prog} \Rightarrow (qname \times \text{mdecl}) \Rightarrow (qname \times \text{mdecl}) \Rightarrow \text{bool}$   
 $(\vdash - \text{hides} - [61, 61, 61] \ 60)$

$G \vdash \text{new hides old}$

$\equiv \text{is-static new} \wedge \text{msig new} = \text{msig old} \wedge$   
 $G \vdash (\text{declclass new}) \prec_C (\text{declclass old}) \wedge$   
 $G \vdash \text{Method new declared-in } (\text{declclass new}) \wedge$   
 $G \vdash \text{Method old declared-in } (\text{declclass old}) \wedge$   
 $G \vdash \text{Method old inheritable-in pid } (\text{declclass new})$

**syntax**

*sig-hides::*  $\text{prog} \Rightarrow \text{sig} \Rightarrow (qname \times \text{mdecl}) \Rightarrow (qname \times \text{mdecl}) \Rightarrow \text{bool}$   
 $(-, \vdash - \text{hides} - [61, 61, 61, 61] \ 60)$

**translations**

$G, s \vdash \text{new hides old} \rightarrow G \vdash (q\text{mdecl } s \text{ new}) \text{ hides } (q\text{mdecl } s \text{ old})$

**lemma hidesI:**

$\llbracket \text{is-static new}; \text{msig new} = \text{msig old};$   
 $G \vdash (\text{declclass new}) \prec_C (\text{declclass old});$   
 $G \vdash \text{Method new declared-in } (\text{declclass new});$   
 $G \vdash \text{Method old declared-in } (\text{declclass old});$   
 $G \vdash \text{Method old inheritable-in pid } (\text{declclass new})$   
 $\rrbracket \implies G \vdash \text{new hides old}$   
 $\langle \text{proof} \rangle$

**lemma hidesD:**

$\llbracket G \vdash \text{new hides old} \rrbracket \implies$   
 $\text{declclass new} \neq \text{Object} \wedge \text{is-static new} \wedge \text{msig new} = \text{msig old} \wedge$



$G \vdash (\text{declclass new}) \prec_C (\text{declclass old}) \wedge$   
 $G \vdash \text{Method new declared-in } (\text{declclass new}) \wedge$   
 $G \vdash \text{Method old declared-in } (\text{declclass old})$   
 $\langle \text{proof} \rangle$

**lemma** *overrides-commonD*:

$\llbracket G \vdash \text{new overrides old} \rrbracket \implies$   
 $\text{declclass new} \neq \text{Object} \wedge \neg \text{is-static new} \wedge \neg \text{is-static old} \wedge$   
 $\text{accmodi new} \neq \text{Private} \wedge$   
 $\text{msig new} = \text{msig old} \wedge$   
 $G \vdash (\text{declclass new}) \prec_C (\text{declclass old}) \wedge$   
 $G \vdash \text{Method new declared-in } (\text{declclass new}) \wedge$   
 $G \vdash \text{Method old declared-in } (\text{declclass old})$   
 $\langle \text{proof} \rangle$

**lemma** *ws-overrides-commonD*:

$\llbracket G \vdash \text{new overrides old}; \text{ws-prog } G \rrbracket \implies$   
 $\text{declclass new} \neq \text{Object} \wedge \neg \text{is-static new} \wedge \neg \text{is-static old} \wedge$   
 $\text{accmodi new} \neq \text{Private} \wedge G \vdash \text{resTy new} \preceq \text{resTy old} \wedge$   
 $\text{msig new} = \text{msig old} \wedge$   
 $G \vdash (\text{declclass new}) \prec_C (\text{declclass old}) \wedge$   
 $G \vdash \text{Method new declared-in } (\text{declclass new}) \wedge$   
 $G \vdash \text{Method old declared-in } (\text{declclass old})$   
 $\langle \text{proof} \rangle$

**lemma** *overrides-eq-sigD*:

$\llbracket G \vdash \text{new overrides old} \rrbracket \implies \text{msig old} = \text{msig new}$   
 $\langle \text{proof} \rangle$

**lemma** *hides-eq-sigD*:

$\llbracket G \vdash \text{new hides old} \rrbracket \implies \text{msig old} = \text{msig new}$   
 $\langle \text{proof} \rangle$

## permits access

### constdefs

*permits-acc*:

$\text{prog} \Rightarrow (\text{qname} \times \text{memberdecl}) \Rightarrow \text{qname} \Rightarrow \text{qname} \Rightarrow \text{bool}$   
 $(- \vdash - \text{ in } - \text{ permits'-acc'-from } - [61, 61, 61, 61] \ 60)$

$G \vdash \text{membr in class permits-acc-from accclass}$   
 $\equiv (\text{case } (\text{accmodi membr}) \text{ of}$   
 $\quad \text{Private} \Rightarrow (\text{declclass membr} = \text{accclass})$   
 $\quad | \text{Package} \Rightarrow (\text{pid } (\text{declclass membr}) = \text{pid accclass})$   
 $\quad | \text{Protected} \Rightarrow (\text{pid } (\text{declclass membr}) = \text{pid accclass})$   
 $\quad \vee$   
 $\quad (G \vdash \text{accclass} \prec_C \text{declclass membr}$   
 $\quad \wedge (G \vdash \text{class} \preceq_C \text{accclass} \vee \text{is-static membr}))$   
 $\quad | \text{Public} \Rightarrow \text{True})$

The subcondition of the *Protected* case:  $G \vdash \text{accclass} \prec_C \text{declclass membr}$  could also be relaxed to:  $G \vdash \text{accclass} \preceq_C \text{declclass membr}$  since in case both classes are the same the other condition  $\text{pid } (\text{declclass membr}) = \text{pid accclass}$  holds anyway.

Like in case of overriding, the static and dynamic accessibility of members is not uniform.

- Statically the class/interface of the member must be accessible for the member to be accessible. During runtime this is not necessary. For Example, if a class is accessible and we are allowed to access a member of this class (statically) we expect that we can access this member in an arbitrary subclass (during runtime). It's not intended to restrict the access to accessible subclasses during runtime.
- Statically the member we want to access must be "member of" the class. Dynamically it must only be "member in" the class.

**consts***accessible-fromR::* $prog \Rightarrow qname \Rightarrow ((qname \times memberdecl) \times qname) \text{ set}$ **syntax***accessible-from::* $prog \Rightarrow (qname \times memberdecl) \Rightarrow qname \Rightarrow qname \Rightarrow bool$   
 $(- \vdash \text{of} - \text{accessible}'\text{-from} - [61,61,61,61] \ 60)$ **translations** $G \vdash \text{membr of cls accessible-from accclass}$  $\Rightarrow (\text{membr}, \text{cls}) \in \text{accessible-fromR } G \text{ accclass}$ **syntax***method-accessible-from::* $prog \Rightarrow (qname \times mdecl) \Rightarrow qname \Rightarrow qname \Rightarrow bool$   
 $(- \vdash \text{Method} - \text{of} - \text{accessible}'\text{-from} - [61,61,61,61] \ 60)$ **translations** $G \vdash \text{Method } m \text{ of cls accessible-from accclass}$  $\Rightarrow G \vdash \text{methdMembr } m \text{ of cls accessible-from accclass}$ **syntax***methd-accessible-from::* $prog \Rightarrow sig \Rightarrow (qname \times methd) \Rightarrow qname \Rightarrow qname \Rightarrow bool$   
 $(- \vdash \text{Methd} - - \text{of} - \text{accessible}'\text{-from} - [61,61,61,61,61] \ 60)$ **translations** $G \vdash \text{Methd } s \ m \text{ of cls accessible-from accclass}$  $\Rightarrow G \vdash (\text{method } s \ m) \text{ of cls accessible-from accclass}$ **syntax***field-accessible-from::* $prog \Rightarrow vname \Rightarrow (qname \times field) \Rightarrow qname \Rightarrow qname \Rightarrow bool$   
 $(- \vdash \text{Field} - - \text{of} - \text{accessible}'\text{-from} - [61,61,61,61,61] \ 60)$ **translations** $G \vdash \text{Field } fn \ f \text{ of } C \text{ accessible-from accclass}$  $\Rightarrow G \vdash (\text{fieldm } fn \ f) \text{ of } C \text{ accessible-from accclass}$ **inductive accessible-fromR G accclass intros***Immediate:*  $\llbracket G \vdash \text{membr member-of class};$  $G \vdash (\text{Class class}) \text{ accessible-in } (pid \text{ accclass});$  $G \vdash \text{membr in class permits-acc-from accclass}$  $\rrbracket \Rightarrow G \vdash \text{membr of class accessible-from accclass}$ *Overriding:*  $\llbracket G \vdash \text{membr member-of class};$  $G \vdash (\text{Class class}) \text{ accessible-in } (pid \text{ accclass});$  $\text{membr} = (C, mdecl \text{ new});$

$$\begin{aligned}
&G \vdash (C, new) \text{ overrides } old; \\
&G \vdash class \prec_C sup; \\
&G \vdash Method \text{ old of sup accessible-from accclass} \\
&\Downarrow \Rightarrow G \vdash \text{membr of class accessible-from accclass}
\end{aligned}$$
**consts**

*dyn-accessible-fromR*::  
 $prog \Rightarrow qname \Rightarrow ((qname \times memberdecl) \times qname) \text{ set}$

**syntax**

*dyn-accessible-from*::  
 $prog \Rightarrow (qname \times memberdecl) \Rightarrow qname \Rightarrow qname \Rightarrow bool$   
 $(- \vdash - \text{ in } - \text{ dyn'-accessible'-from } - [61, 61, 61, 61] \ 60)$

**translations**

$G \vdash \text{membr in } C \text{ dyn-accessible-from accC}$   
 $\Leftrightarrow (\text{membr}, C) \in \text{dyn-accessible-fromR } G \text{ accC}$

**syntax**

*method-dyn-accessible-from*::  
 $prog \Rightarrow (qname \times mdecl) \Rightarrow qname \Rightarrow qname \Rightarrow bool$   
 $(- \vdash Method - \text{ in } - \text{ dyn'-accessible'-from } - [61, 61, 61, 61] \ 60)$

**translations**

$G \vdash Method \ m \text{ in } C \text{ dyn-accessible-from accC}$   
 $\Leftrightarrow G \vdash \text{methdMembr } m \text{ in } C \text{ dyn-accessible-from accC}$

**syntax**

*methd-dyn-accessible-from*::  
 $prog \Rightarrow sig \Rightarrow (qname \times methd) \Rightarrow qname \Rightarrow qname \Rightarrow bool$   
 $(- \vdash Methd - - \text{ in } - \text{ dyn'-accessible'-from } - [61, 61, 61, 61, 61] \ 60)$

**translations**

$G \vdash Methd \ s \ m \text{ in } C \text{ dyn-accessible-from accC}$   
 $\Leftrightarrow G \vdash (\text{method } s \ m) \text{ in } C \text{ dyn-accessible-from accC}$

**syntax**

*field-dyn-accessible-from*::  
 $prog \Rightarrow vname \Rightarrow (qname \times field) \Rightarrow qname \Rightarrow qname \Rightarrow bool$   
 $(- \vdash Field - - \text{ in } - \text{ dyn'-accessible'-from } - [61, 61, 61, 61, 61] \ 60)$

**translations**

$G \vdash Field \ fn \ f \text{ in } dynC \text{ dyn-accessible-from accC}$   
 $\Leftrightarrow G \vdash (\text{fieldm } fn \ f) \text{ in } dynC \text{ dyn-accessible-from accC}$

**inductive dyn-accessible-fromR G accclass intros**

*Immediate*:  $\Downarrow G \vdash \text{membr member-in class};$   
 $G \vdash \text{membr in class permits-acc-from accclass}$   
 $\Downarrow \Rightarrow G \vdash \text{membr in class dyn-accessible-from accclass}$

*Overriding*:  $\Downarrow G \vdash \text{membr member-in class};$   
 $\text{membr} = (C, mdecl \ new);$   
 $G \vdash (C, new) \text{ overrides } old;$   
 $G \vdash class \prec_C sup;$   
 $G \vdash Method \ old \text{ in } sup \text{ dyn-accessible-from accclass}$   
 $\Downarrow \Rightarrow G \vdash \text{membr in class dyn-accessible-from accclass}$

**lemma** *accessible-from-commonD*:  $G \vdash m$  of  $C$  accessible-from  $S$   
 $\implies G \vdash m$  member-of  $C \wedge G \vdash (\text{Class } C) \text{ accessible-in } (\text{pid } S)$   
 ⟨proof⟩

**lemma** *unique-declaration*:  
 $\llbracket G \vdash m \text{ declared-in } C; G \vdash n \text{ declared-in } C; \text{memberid } m = \text{memberid } n \rrbracket$   
 $\implies m = n$   
 ⟨proof⟩

**lemma** *declared-not-undeclared*:  
 $G \vdash m \text{ declared-in } C \implies \neg G \vdash \text{memberid } m \text{ undeclared-in } C$   
 ⟨proof⟩

**lemma** *undeclared-not-declared*:  
 $G \vdash \text{memberid } m \text{ undeclared-in } C \implies \neg G \vdash m \text{ declared-in } C$   
 ⟨proof⟩

**lemma** *not-undeclared-declared*:  
 $\neg G \vdash \text{membr-id undeclared-in } C \implies (\exists m. G \vdash m \text{ declared-in } C \wedge$   
 $\text{membr-id} = \text{memberid } m)$   
 ⟨proof⟩

**lemma** *unique-declared-in*:  
 $\llbracket G \vdash m \text{ declared-in } C; G \vdash n \text{ declared-in } C; \text{memberid } m = \text{memberid } n \rrbracket$   
 $\implies m = n$   
 ⟨proof⟩

**lemma** *unique-member-of*:  
 assumes  $n: G \vdash n \text{ member-of } C$  and  
 $m: G \vdash m \text{ member-of } C$  and  
 $\text{eqid: memberid } n = \text{memberid } m$   
 shows  $n=m$   
 ⟨proof⟩

**lemma** *member-of-is-classD*:  $G \vdash m \text{ member-of } C \implies \text{is-class } G \ C$   
 ⟨proof⟩

**lemma** *member-of-declC*:  
 $G \vdash m \text{ member-of } C$   
 $\implies G \vdash \text{mbr } m \text{ declared-in } (\text{declclass } m)$   
 ⟨proof⟩

**lemma** *member-of-member-of-declC*:  
 $G \vdash m \text{ member-of } C$   
 $\implies G \vdash m \text{ member-of } (\text{declclass } m)$   
 ⟨proof⟩

**lemma** *member-of-class-relation*:  
 $G \vdash m \text{ member-of } C \implies G \vdash C \preceq_C \text{ declclass } m$

$\langle \text{proof} \rangle$

**lemma** *member-in-class-relation*:

$G \vdash m \text{ member-in } C \implies G \vdash C \preceq_C \text{ declclass } m$   
 $\langle \text{proof} \rangle$

**lemma** *stat-override-declclasses-relation*:

$\llbracket G \vdash (\text{declclass } \text{new}) \prec_{C1} \text{superNew}; G \vdash \text{Method } \text{old} \text{ member-of } \text{superNew} \rrbracket$   
 $\implies G \vdash (\text{declclass } \text{new}) \prec_C (\text{declclass } \text{old})$   
 $\langle \text{proof} \rangle$

**lemma** *stat-overrides-commonD*:

$\llbracket G \vdash \text{new overrides}_S \text{old} \rrbracket \implies$   
 $\text{declclass } \text{new} \neq \text{Object} \wedge \neg \text{is-static } \text{new} \wedge \text{msig } \text{new} = \text{msig } \text{old} \wedge$   
 $G \vdash (\text{declclass } \text{new}) \prec_C (\text{declclass } \text{old}) \wedge$   
 $G \vdash \text{Method } \text{new} \text{ declared-in } (\text{declclass } \text{new}) \wedge$   
 $G \vdash \text{Method } \text{old} \text{ declared-in } (\text{declclass } \text{old})$   
 $\langle \text{proof} \rangle$

**lemma** *member-of-Package*:

$\llbracket G \vdash m \text{ member-of } C; \text{accmodi } m = \text{Package} \rrbracket$   
 $\implies \text{pid } (\text{declclass } m) = \text{pid } C$   
 $\langle \text{proof} \rangle$

**lemma** *member-in-declC*:  $G \vdash m \text{ member-in } C \implies G \vdash m \text{ member-in } (\text{declclass } m)$

$\langle \text{proof} \rangle$

**lemma** *dyn-accessible-from-commonD*:  $G \vdash m \text{ in } C \text{ dyn-accessible-from } S$

$\implies G \vdash m \text{ member-in } C$   
 $\langle \text{proof} \rangle$

**lemma** *no-Private-stat-override*:

$\llbracket G \vdash \text{new overrides}_S \text{old} \rrbracket \implies \text{accmodi } \text{old} \neq \text{Private}$   
 $\langle \text{proof} \rangle$

**lemma** *no-Private-override*:  $\llbracket G \vdash \text{new overrides } \text{old} \rrbracket \implies \text{accmodi } \text{old} \neq \text{Private}$

$\langle \text{proof} \rangle$

**lemma** *permits-acc-inheritance*:

$\llbracket G \vdash m \text{ in } \text{statC} \text{ permits-acc-from } \text{accC}; G \vdash \text{dynC} \preceq_C \text{statC} \rrbracket$   
 $\implies G \vdash m \text{ in } \text{dynC} \text{ permits-acc-from } \text{accC}$   
 $\langle \text{proof} \rangle$

**lemma** *permits-acc-static-declC*:

$\llbracket G \vdash m \text{ in } C \text{ permits-acc-from } \text{accC}; G \vdash m \text{ member-in } C; \text{is-static } m \rrbracket$   
 $\implies G \vdash m \text{ in } (\text{declclass } m) \text{ permits-acc-from } \text{accC}$   
 $\langle \text{proof} \rangle$

**lemma** *dyn-accessible-from-static-declC*:

**assumes**  $\text{acc-}C$ :  $G \vdash m \text{ in } C \text{ dyn-accessible-from } \text{acc}C$  **and**  
*static*: *is-static*  $m$

**shows**  $G \vdash m \text{ in } (\text{declclass } m) \text{ dyn-accessible-from } \text{acc}C$

$\langle \text{proof} \rangle$

**lemma** *field-accessible-fromD*:

$\llbracket G \vdash \text{membr of } C \text{ accessible-from } \text{acc}C; \text{is-field membr} \rrbracket$

$\implies G \vdash \text{membr member-of } C \wedge$

$G \vdash (\text{Class } C) \text{ accessible-in } (\text{pid } \text{acc}C) \wedge$

$G \vdash \text{membr in } C \text{ permits-acc-from } \text{acc}C$

$\langle \text{proof} \rangle$

**lemma** *field-accessible-from-permits-acc-inheritance*:

$\llbracket G \vdash \text{membr of } \text{stat}C \text{ accessible-from } \text{acc}C; \text{is-field membr}; G \vdash \text{dyn}C \preceq_C \text{stat}C \rrbracket$

$\implies G \vdash \text{membr in } \text{dyn}C \text{ permits-acc-from } \text{acc}C$

$\langle \text{proof} \rangle$

**lemma** *accessible-fieldD*:

$\llbracket G \vdash \text{membr of } C \text{ accessible-from } \text{acc}C; \text{is-field membr} \rrbracket$

$\implies G \vdash \text{membr member-of } C \wedge$

$G \vdash (\text{Class } C) \text{ accessible-in } (\text{pid } \text{acc}C) \wedge$

$G \vdash \text{membr in } C \text{ permits-acc-from } \text{acc}C$

$\langle \text{proof} \rangle$

**lemma** *member-of-Private*:

$\llbracket G \vdash m \text{ member-of } C; \text{accmodi } m = \text{Private} \rrbracket \implies \text{declclass } m = C$

$\langle \text{proof} \rangle$

**lemma** *member-of-subclseq-declC*:

$G \vdash m \text{ member-of } C \implies G \vdash C \preceq_C \text{declclass } m$

$\langle \text{proof} \rangle$

**lemma** *member-of-inheritance*:

**assumes**  $m$ :  $G \vdash m \text{ member-of } D$  **and**

*subclseq-D-C*:  $G \vdash D \preceq_C C$  **and**

*subclseq-C-m*:  $G \vdash C \preceq_C \text{declclass } m$  **and**

*ws*: *ws-prog*  $G$

**shows**  $G \vdash m \text{ member-of } C$

$\langle \text{proof} \rangle$

**lemma** *member-of-subcls*:

**assumes** *old*:  $G \vdash \text{old member-of } C$  **and**

*new*:  $G \vdash \text{new member-of } D$  **and**

*eqid*:  $\text{memberid new} = \text{memberid old}$  **and**

*subclseq-D-C*:  $G \vdash D \preceq_C C$  **and**

*subcls-new-old*:  $G \vdash \text{declclass new} \prec_C \text{declclass old}$  **and**

*ws: ws-prog G*  
**shows**  $G \vdash D \prec_C C$   
 ⟨proof⟩

**corollary** *member-of-overrides-subcls:*

$\llbracket G \vdash \text{Methd sig old member-of } C; G \vdash \text{Methd sig new member-of } D; G \vdash D \preceq_C C;$   
 $G, \text{sig} \vdash \text{new overrides old}; \text{ws-prog } G \rrbracket$   
 $\implies G \vdash D \prec_C C$   
 ⟨proof⟩

**corollary** *member-of-stat-overrides-subcls:*

$\llbracket G \vdash \text{Methd sig old member-of } C; G \vdash \text{Methd sig new member-of } D; G \vdash D \preceq_C C;$   
 $G, \text{sig} \vdash \text{new overrides}_S \text{ old}; \text{ws-prog } G \rrbracket$   
 $\implies G \vdash D \prec_C C$   
 ⟨proof⟩

**lemma** *inherited-field-access:*

**assumes** *stat-acc:*  $G \vdash \text{membr of stat} C \text{ accessible-from acc} C$  **and**  
*is-field:* *is-field membr* **and**  
*subclseq:*  $G \vdash \text{dyn} C \preceq_C \text{stat} C$   
**shows**  $G \vdash \text{membr in dyn} C \text{ dyn-accessible-from acc} C$   
 ⟨proof⟩

**lemma** *accessible-inheritance:*

**assumes** *stat-acc:*  $G \vdash m \text{ of stat} C \text{ accessible-from acc} C$  **and**  
*subclseq:*  $G \vdash \text{dyn} C \preceq_C \text{stat} C$  **and**  
*member-dynC:*  $G \vdash m \text{ member-of dyn} C$  **and**  
*dynC-acc:*  $G \vdash (\text{Class dyn} C) \text{ accessible-in (pid acc} C)$   
**shows**  $G \vdash m \text{ of dyn} C \text{ accessible-from acc} C$   
 ⟨proof⟩

## fields and methods

### types

$f\text{spec} = \text{vname} \times \text{qtname}$

### translations

$f\text{spec} \leq (\text{type}) \text{vname} \times \text{qtname}$

### constdefs

*imethds:: prog  $\Rightarrow$  qtname  $\Rightarrow$  (sig, qtname  $\times$  mhead) tables*  
*imethds G I*  
 $\equiv \text{iface-rec } (G, I)$   
 $(\lambda I \ i \ ts. (\text{Un-tables } ts) \oplus \oplus$   
 $(o2s \circ \text{table-of } (\text{map } (\lambda (s, m). (s, I, m)) (\text{imethds } i))))$

methods of an interface, with overriding and inheritance, cf. 9.2

### constdefs

*accimethds:: prog  $\Rightarrow$  pname  $\Rightarrow$  qtname  $\Rightarrow$  (sig, qtname  $\times$  mhead) tables*  
*accimethds G pack I*  
 $\equiv \text{if } G \vdash \text{Iface } I \text{ accessible-in pack}$   
 $\text{then imethds } G \ I$   
 $\text{else } \lambda k. \{\}$

only returns imethds if the interface is accessible

**constdefs**

*methd*:: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  (*sig*, *qname*  $\times$  *methd*) *table*

*methd* *G* *C*

$\equiv$  *class-rec* (*G*, *C*) *empty*  
     ( $\lambda C$  *c* *subcls-mthds*.  
         *filter-tab* ( $\lambda sig$  *m*. *G*  $\vdash$  *C* inherits method *sig* *m*)  
             *subcls-mthds*  
         ++  
         *table-of* (*map* ( $\lambda (s, m)$ . (*s*, *C*, *m*)) (*methods* *c*)))

*methd* *G* *C*: methods of a class *C* (statically visible from *C*), with inheritance and hiding cf. 8.4.6; Overriding is captured by *dynmethd*. Every new method with the same signature coalesces the method of a superclass.

**constdefs**

*accmethd*:: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  *qname*  $\Rightarrow$  (*sig*, *qname*  $\times$  *methd*) *table*

*accmethd* *G* *S* *C*

$\equiv$  *filter-tab* ( $\lambda sig$  *m*. *G*  $\vdash$  method *sig* *m* of *C* accessible-from *S*)  
     (*methd* *G* *C*)

*accmethd* *G* *S* *C*: only those methods of *methd* *G* *C*, accessible from *S*

Note the class component in the accessibility filter. The class where method *m* is declared (*declC*) isn't necessarily accessible from the current scope *S*. The method can be made accessible through inheritance, too. So we must test accessibility of method *m* of class *C* (not *declclass* *m*)

**constdefs**

*dynmethd*:: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  *qname*  $\Rightarrow$  (*sig*, *qname*  $\times$  *methd*) *table*

*dynmethd* *G* *statC* *dynC*

$\equiv$   $\lambda sig$ .  
     (*if* *G*  $\vdash$  *dynC*  $\preceq_C$  *statC*  
         then (*case* *methd* *G* *statC* *sig* of

*None*  $\Rightarrow$  *None*  
         | *Some* *statM*  
              $\Rightarrow$  (*class-rec* (*G*, *dynC*) *empty*  
                 ( $\lambda C$  *c* *subcls-mthds*.  
                     *subcls-mthds*  
                     ++  
                     (*filter-tab*  
                         ( $\lambda -$  *dynM*. *G*, *sig*  $\vdash$  *dynM* overrides *statM*  $\vee$  *dynM* = *statM*)  
                         (*methd* *G* *C*))  
                     )  
                 )  
         else *None*)

*dynmethd* *G* *statC* *dynC*: dynamic method lookup of a reference with dynamic class *dynC* and static class *statC*

Note some kind of duality between *methd* and *dynmethd* in the *class-rec* arguments. Whereas *methd* filters the subclass methods (to get only the inherited ones), *dynmethd* filters the new methods (to get only those methods which actually override the methods of the static class)

**constdefs**

*dynimethd*:: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  *qname*  $\Rightarrow$  (*sig*, *qname*  $\times$  *methd*) *table*

*dynimethd* *G* *I* *dynC*

$\equiv$   $\lambda sig$ . *if* *imethds* *G* *I* *sig*  $\neq$  {}  
     then *methd* *G* *dynC* *sig*  
     else *dynmethd* *G* *Object* *dynC* *sig*

*dynimethd* *G* *I* *dynC*: dynamic method lookup of a reference with dynamic class *dynC* and static interface type *I*



When calling an interface method, we must distinguish if the method signature was defined in the interface or if it must be an Object method in the other case. If it was an interface method we search the class hierarchy starting at the dynamic class of the object up to Object to find the first matching method (*methd*). Since all interface methods have public access the method can't be coalesced due to some odd visibility effects like in case of *dynmethd*. The method will be inherited or overridden in all classes from the first class implementing the interface down to the actual dynamic class.

#### constdefs

*dynlookup*:: *prog*  $\Rightarrow$  *ref-ty*  $\Rightarrow$  *qname*  $\Rightarrow$  (*sig*, *qname*  $\times$  *methd*) *table*

*dynlookup* *G* *statT* *dynC*

$\equiv$  (case *statT* of  
   *NullT*  $\Rightarrow$  *empty*  
   | *IfaceT* *I*  $\Rightarrow$  *dynimethd* *G* *I* *dynC*  
   | *ClassT* *statC*  $\Rightarrow$  *dynmethd* *G* *statC* *dynC*  
   | *ArrayT* *ty*  $\Rightarrow$  *dynmethd* *G* *Object* *dynC*)

*dynlookup* *G* *statT* *dynC*: dynamic lookup of a method within the static reference type *statT* and the dynamic class *dynC*. In a wellformd context *statT* will not be *NullT* and in case *statT* is an array type, *dynC*=Object

#### constdefs

*fields*:: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  ((*vname*  $\times$  *qname*)  $\times$  *field*) *list*

*fields* *G* *C*

$\equiv$  *class-rec* (*G*, *C*) [] ( $\lambda C$  *c* *ts*. *map* ( $\lambda(n, t)$ . ((*n*, *C*), *t*)) (*cfields* *c*) @ *ts*)

*DeclConcepts.fields* *G* *C* list of fields of a class, including all the fields of the superclasses (private, inherited and hidden ones) not only the accessible ones (an instance of a object allocates all these fields)

#### constdefs

*accfield*:: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  *qname*  $\Rightarrow$  (*vname*, *qname*  $\times$  *field*) *table*

*accfield* *G* *S* *C*

$\equiv$  let *field-tab* = *table-of*((*map* ( $\lambda((n, d), f)$ . (*n*, (*d*, *f*)))) (*fields* *G* *C*))  
   in *filter-tab* ( $\lambda n$  (*declC*, *f*). *G*  $\vdash$  (*declC*, *fdecl* (*n*, *f*)) of *C* accessible-from *S*)

*field-tab*

*accfield* *G* *C* *S*: fields of a class *C* which are accessible from scope of class *S* with inheritance and hiding, cf. 8.3

note the class component in the accessibility filter (see also *methd*). The class declaring field *f* (*declC*) isn't necessarily accessible from scope *S*. The field can be made visible through inheritance, too. So we must test accessibility of field *f* of class *C* (not *declclass* *f*)

#### constdefs

*is-methd* :: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  *sig*  $\Rightarrow$  *bool*

*is-methd* *G*  $\equiv$   $\lambda C$  *sig*. *is-class* *G* *C*  $\wedge$  *methd* *G* *C* *sig*  $\neq$  *None*

**constdefs** *efname*:: ((*vname*  $\times$  *qname*)  $\times$  *field*)  $\Rightarrow$  (*vname*  $\times$  *qname*)

*efname*  $\equiv$  *fst*

**lemma** *efname-simp*[*simp*]: *efname* (*n*, *f*) = *n*

$\langle$ *proof* $\rangle$

## 19 imethds

**lemma** *imethds-rec*:  $\llbracket$ *iface* *G* *I* = *Some* *i*; *ws-prog* *G* $\rrbracket \Longrightarrow$

*imethds* *G* *I* = *Un-tables* (( $\lambda J$ . *imethds* *G* *J*) 'set (*isuperIfs* *i*))  $\oplus \oplus$   
   (*o2s*  $\circ$  *table-of* (*map* ( $\lambda(s, mh)$ . (*s*, *I*, *mh*)) (*imethds* *i*)))

$\langle \text{proof} \rangle$

**lemma** *imethds-norec*:

$\llbracket \text{iface } G \text{ md} = \text{Some } i; \text{ws-prog } G; \text{table-of } (\text{imethds } i) \text{ sig} = \text{Some } mh \rrbracket \implies$   
 $(\text{md}, mh) \in \text{imethds } G \text{ md sig}$

$\langle \text{proof} \rangle$

**lemma** *imethds-declI*:  $\llbracket m \in \text{imethds } G \text{ I sig}; \text{ws-prog } G; \text{is-iface } G \text{ I} \rrbracket \implies$

$(\exists i. \text{iface } G (\text{decliface } m) = \text{Some } i \wedge$   
 $\text{table-of } (\text{imethds } i) \text{ sig} = \text{Some } (\text{mthd } m)) \wedge$   
 $(I, \text{decliface } m) \in (\text{subint1 } G) \hat{*} \wedge m \in \text{imethds } G (\text{decliface } m) \text{ sig}$

$\langle \text{proof} \rangle$

**lemma** *imethds-cases* [consumes 3, case-names *NewMethod InheritedMethod*]:

**assumes** *im*:  $im \in \text{imethds } G \text{ I sig}$  **and**

*ifI*:  $\text{iface } G \text{ I} = \text{Some } i$  **and**

*ws*:  $\text{ws-prog } G$  **and**

*hyp-new*:  $\text{table-of } (\text{map } (\lambda(s, mh). (s, I, mh)) (\text{imethds } i)) \text{ sig}$   
 $= \text{Some } im \implies P$  **and**

*hyp-inh*:  $\bigwedge J. \llbracket J \in \text{set } (\text{isuperIfs } i); im \in \text{imethds } G \text{ J sig} \rrbracket \implies P$

**shows** *P*

$\langle \text{proof} \rangle$

## 20 accimethd

**lemma** *accimethds-simp* [simp]:

$G \vdash \text{Iface } I \text{ accessible-in pack} \implies \text{accimethds } G \text{ pack } I = \text{imethds } G \text{ I}$

$\langle \text{proof} \rangle$

**lemma** *accimethdsD*:

$im \in \text{accimethds } G \text{ pack } I \text{ sig}$

$\implies im \in \text{imethds } G \text{ I sig} \wedge G \vdash \text{Iface } I \text{ accessible-in pack}$

$\langle \text{proof} \rangle$

**lemma** *accimethdsI*:

$\llbracket im \in \text{imethds } G \text{ I sig}; G \vdash \text{Iface } I \text{ accessible-in pack} \rrbracket$

$\implies im \in \text{accimethds } G \text{ pack } I \text{ sig}$

$\langle \text{proof} \rangle$

## 21 methd

**lemma** *methd-rec*:  $\llbracket \text{class } G \text{ C} = \text{Some } c; \text{ws-prog } G \rrbracket \implies$

$\text{methd } G \text{ C}$

$= (\text{if } C = \text{Object}$

$\text{then empty}$

$\text{else filter-tab } (\lambda \text{sig } m. G \vdash C \text{ inherits method sig } m)$   
 $(\text{methd } G (\text{super } c)))$

$++ \text{table-of } (\text{map } (\lambda(s, m). (s, C, m)) (\text{methods } c))$

$\langle \text{proof} \rangle$

**lemma** *methd-norec*:

$\llbracket \text{class } G \text{ decl}C = \text{Some } c; \text{ws-prog } G; \text{table-of } (\text{methods } c) \text{ sig} = \text{Some } m \rrbracket$   
 $\implies \text{methd } G \text{ decl}C \text{ sig} = \text{Some } (\text{decl}C, m)$   
 $\langle \text{proof} \rangle$

**lemma** *methd-declC*:

$\llbracket \text{methd } G \text{ } C \text{ sig} = \text{Some } m; \text{ws-prog } G; \text{is-class } G \text{ } C \rrbracket \implies$   
 $(\exists d. \text{class } G \text{ (declclass } m) = \text{Some } d \wedge \text{table-of } (\text{methods } d) \text{ sig} = \text{Some } (\text{methd } m)) \wedge$   
 $G \vdash C \preceq_C (\text{declclass } m) \wedge \text{methd } G \text{ (declclass } m) \text{ sig} = \text{Some } m$   
 $\langle \text{proof} \rangle$

**lemma** *methd-inheritedD*:

$\llbracket \text{class } G \text{ } C = \text{Some } c; \text{ws-prog } G; \text{methd } G \text{ } C \text{ sig} = \text{Some } m \rrbracket$   
 $\implies (\text{declclass } m \neq C \longrightarrow G \vdash C \text{ inherits method sig } m)$   
 $\langle \text{proof} \rangle$

**lemma** *methd-diff-cls*:

$\llbracket \text{ws-prog } G; \text{is-class } G \text{ } C; \text{is-class } G \text{ } D;$   
 $\text{methd } G \text{ } C \text{ sig} = m; \text{methd } G \text{ } D \text{ sig} = n; m \neq n$   
 $\rrbracket \implies C \neq D$   
 $\langle \text{proof} \rangle$

**lemma** *method-declared-inI*:

$\llbracket \text{table-of } (\text{methods } c) \text{ sig} = \text{Some } m; \text{class } G \text{ } C = \text{Some } c \rrbracket$   
 $\implies G \vdash \text{mdecl } (\text{sig}, m) \text{ declared-in } C$   
 $\langle \text{proof} \rangle$

**lemma** *methd-declared-in-declclass*:

$\llbracket \text{methd } G \text{ } C \text{ sig} = \text{Some } m; \text{ws-prog } G; \text{is-class } G \text{ } C \rrbracket$   
 $\implies G \vdash \text{Methd sig } m \text{ declared-in } (\text{declclass } m)$   
 $\langle \text{proof} \rangle$

**lemma** *member-methd*:

**assumes** *member-of*:  $G \vdash \text{Methd sig } m \text{ member-of } C$  **and**  
 $\text{ws: ws-prog } G$   
**shows**  $\text{methd } G \text{ } C \text{ sig} = \text{Some } m$   
 $\langle \text{proof} \rangle$

**lemma** *finite-methd:ws-prog*  $G \implies \text{finite } \{\text{methd } G \text{ } C \text{ sig} \mid \text{sig } C. \text{is-class } G \text{ } C\}$

$\langle \text{proof} \rangle$

**lemma** *finite-dom-methd*:

$\llbracket \text{ws-prog } G; \text{is-class } G \text{ } C \rrbracket \implies \text{finite } (\text{dom } (\text{methd } G \text{ } C))$   
 $\langle \text{proof} \rangle$

## 22 accmethd

**lemma** *accmethd-SomeD*:

$\text{accmethd } G \text{ } S \text{ } C \text{ sig} = \text{Some } m$

$\implies \text{methd } G \ C \ \text{sig} = \text{Some } m \wedge G \vdash \text{method } \text{sig } m \text{ of } C \text{ accessible-from } S$   
 <proof>

**lemma** *accmethd-SomeI*:

$\llbracket \text{methd } G \ C \ \text{sig} = \text{Some } m; G \vdash \text{method } \text{sig } m \text{ of } C \text{ accessible-from } S \rrbracket$   
 $\implies \text{accmethd } G \ S \ C \ \text{sig} = \text{Some } m$   
 <proof>

**lemma** *accmethd-declC*:

$\llbracket \text{accmethd } G \ S \ C \ \text{sig} = \text{Some } m; \text{ws-prog } G; \text{is-class } G \ C \rrbracket \implies$   
 $(\exists d. \text{class } G \ (\text{declclass } m) = \text{Some } d \wedge$   
 $\text{table-of } (\text{methods } d) \ \text{sig} = \text{Some } (\text{methd } m)) \wedge$   
 $G \vdash C \preceq_C (\text{declclass } m) \wedge \text{methd } G \ (\text{declclass } m) \ \text{sig} = \text{Some } m \wedge$   
 $G \vdash \text{method } \text{sig } m \text{ of } C \text{ accessible-from } S$   
 <proof>

**lemma** *finite-dom-accmethd*:

$\llbracket \text{ws-prog } G; \text{is-class } G \ C \rrbracket \implies \text{finite } (\text{dom } (\text{accmethd } G \ S \ C))$   
 <proof>

## 23 dynmethd

**lemma** *dynmethd-rec*:

$\llbracket \text{class } G \ \text{dynC} = \text{Some } c; \text{ws-prog } G \rrbracket \implies$   
 $\text{dynmethd } G \ \text{statC} \ \text{dynC} \ \text{sig}$   
 $= (\text{if } G \vdash \text{dynC} \preceq_C \text{statC}$   
 $\text{then } (\text{case methd } G \ \text{statC} \ \text{sig} \text{ of}$   
 $\text{None} \Rightarrow \text{None}$   
 $\mid \text{Some } \text{statM}$   
 $\Rightarrow (\text{case methd } G \ \text{dynC} \ \text{sig} \text{ of}$   
 $\text{None} \Rightarrow \text{dynmethd } G \ \text{statC} \ (\text{super } c) \ \text{sig}$   
 $\mid \text{Some } \text{dynM} \Rightarrow$   
 $(\text{if } G, \text{sig} \vdash \text{dynM} \text{ overrides } \text{statM} \vee \text{dynM} = \text{statM}$   
 $\text{then } \text{Some } \text{dynM}$   
 $\text{else } (\text{dynmethd } G \ \text{statC} \ (\text{super } c) \ \text{sig})$   
 $)))$   
 $\text{else } \text{None})$   
 $(\text{is } - \implies - \implies ?\text{Dynmethd-def } \text{dynC} \ \text{sig} = ?\text{Dynmethd-rec } \text{dynC} \ c \ \text{sig})$   
 <proof>

**lemma** *dynmethd-C-C*:  $\llbracket \text{is-class } G \ C; \text{ws-prog } G \rrbracket$

$\implies \text{dynmethd } G \ C \ C \ \text{sig} = \text{methd } G \ C \ \text{sig}$   
 <proof>

**lemma** *dynmethdSomeD*:

$\llbracket \text{dynmethd } G \ \text{statC} \ \text{dynC} \ \text{sig} = \text{Some } \text{dynM}; \text{is-class } G \ \text{dynC}; \text{ws-prog } G \rrbracket$   
 $\implies G \vdash \text{dynC} \preceq_C \text{statC} \wedge (\exists \text{statM}. \text{methd } G \ \text{statC} \ \text{sig} = \text{Some } \text{statM})$   
 <proof>

**lemma** *dynmethd-Some-cases* [consumes 3, case-names Static Overrides]:

**assumes**  $\text{dynM}: \text{dynmethd } G \ \text{statC} \ \text{dynC} \ \text{sig} = \text{Some } \text{dynM}$  **and**  
 $\text{is-cls-dynC}: \text{is-class } G \ \text{dynC}$  **and**

$ws: ws\text{-}prog\ G$  **and**  
 $hyp\text{-}static: methd\ G\ statC\ sig = Some\ dynM \implies P$  **and**  
 $hyp\text{-}override: \bigwedge statM. \llbracket methd\ G\ statC\ sig = Some\ statM; dynM \neq statM;$   
 $G, sig \vdash dynM\ overrides\ statM \rrbracket \implies P$   
**shows**  $P$   
 $\langle proof \rangle$

**lemma** *no-override-in-Object*:

**assumes**  $dynM: dynmethd\ G\ statC\ dynC\ sig = Some\ dynM$  **and**  
 $is\text{-}cls\text{-}dynC: is\text{-}class\ G\ dynC$  **and**  
 $ws: ws\text{-}prog\ G$  **and**  
 $statM: methd\ G\ statC\ sig = Some\ statM$  **and**  
 $neq\text{-}dynM\text{-}statM: dynM \neq statM$   
**shows**  $dynC \neq Object$   
 $\langle proof \rangle$

**lemma** *dynmethd-Some-rec-cases* [consumes 3,

*case-names Static Override Recursion*]:

**assumes**  $dynM: dynmethd\ G\ statC\ dynC\ sig = Some\ dynM$  **and**  
 $clsDynC: class\ G\ dynC = Some\ c$  **and**  
 $ws: ws\text{-}prog\ G$  **and**  
 $hyp\text{-}static: methd\ G\ statC\ sig = Some\ dynM \implies P$  **and**  
 $hyp\text{-}override: \bigwedge statM. \llbracket methd\ G\ statC\ sig = Some\ statM;$   
 $methd\ G\ dynC\ sig = Some\ dynM; statM \neq dynM;$   
 $G, sig \vdash dynM\ overrides\ statM \rrbracket \implies P$  **and**  
 $hyp\text{-}recursion: \llbracket dynC \neq Object;$   
 $dynmethd\ G\ statC\ (super\ c)\ sig = Some\ dynM \rrbracket \implies P$   
**shows**  $P$   
 $\langle proof \rangle$

**lemma** *dynmethd-declC*:

$\llbracket dynmethd\ G\ statC\ dynC\ sig = Some\ m;$   
 $is\text{-}class\ G\ statC; ws\text{-}prog\ G$   
 $\rrbracket \implies$   
 $(\exists d. class\ G\ (declclass\ m) = Some\ d \wedge table\text{-}of\ (methods\ d)\ sig = Some\ (methd\ m)) \wedge$   
 $G \vdash dynC \preceq_C (declclass\ m) \wedge methd\ G\ (declclass\ m)\ sig = Some\ m$   
 $\langle proof \rangle$

**lemma** *methd-Some-dynmethd-Some*:

**assumes**  $statM: methd\ G\ statC\ sig = Some\ statM$  **and**  
 $subclseq: G \vdash dynC \preceq_C statC$  **and**  
 $is\text{-}cls\text{-}statC: is\text{-}class\ G\ statC$  **and**  
 $ws: ws\text{-}prog\ G$   
**shows**  $\exists dynM. dynmethd\ G\ statC\ dynC\ sig = Some\ dynM$   
 $(is\ ?P\ dynC)$   
 $\langle proof \rangle$

**lemma** *dynmethd-cases* [consumes 4, case-names Static Overrides]:

**assumes**  $statM: methd\ G\ statC\ sig = Some\ statM$  **and**  
 $subclseq: G \vdash dynC \preceq_C statC$  **and**  
 $is\text{-}cls\text{-}statC: is\text{-}class\ G\ statC$  **and**  
 $ws: ws\text{-}prog\ G$  **and**

*hyp-static*:  $\text{dynmethd } G \text{ statC dynC sig} = \text{Some statM} \implies P$  **and**  
*hyp-override*:  $\bigwedge \text{dynM}. \llbracket \text{dynmethd } G \text{ statC dynC sig} = \text{Some dynM};$   
 $\text{dynM} \neq \text{statM};$   
 $G, \text{sig} \vdash \text{dynM overrides statM} \rrbracket \implies P$

**shows**  $P$   
 $\langle \text{proof} \rangle$

**lemma** *ws-dynmethd*:

**assumes**  $\text{statM}: \text{methd } G \text{ statC sig} = \text{Some statM}$  **and**  
 $\text{subclseq}: G \vdash \text{dynC} \preceq_C \text{statC}$  **and**  
 $\text{is-clc-statC}: \text{is-class } G \text{ statC}$  **and**  
 $\text{ws}: \text{ws-prog } G$

**shows**

$\exists \text{dynM}. \text{dynmethd } G \text{ statC dynC sig} = \text{Some dynM} \wedge$   
 $\text{is-static dynM} = \text{is-static statM} \wedge G \vdash \text{resTy dynM} \preceq_{\text{resTy}} \text{statM}$

$\langle \text{proof} \rangle$

## 24 dynlookup

**lemma** *dynlookup-cases* [*consumes 1, case-names NullT IfaceT ClassT ArrayT*]:

$\llbracket \text{dynlookup } G \text{ statT dynC sig} = x;$   
 $\llbracket \text{statT} = \text{NullT} \quad ; \text{empty sig} = x \rrbracket \implies P;$   
 $\bigwedge I. \llbracket \text{statT} = \text{IfaceT } I \quad ; \text{dynimethd } G \text{ } I \quad \text{dynC sig} = x \rrbracket \implies P;$   
 $\bigwedge \text{statC}. \llbracket \text{statT} = \text{ClassT statC}; \text{dynmethd } G \text{ statC dynC sig} = x \rrbracket \implies P;$   
 $\bigwedge \text{ty}. \llbracket \text{statT} = \text{ArrayT ty} \quad ; \text{dynmethd } G \text{ Object dynC sig} = x \rrbracket \implies P$   
 $\rrbracket \implies P$   
 $\langle \text{proof} \rangle$

## 25 fields

**lemma** *fields-rec*:  $\llbracket \text{class } G \text{ } C = \text{Some } c; \text{ws-prog } G \rrbracket \implies$

$\text{fields } G \text{ } C = \text{map } (\lambda(fn, ft). ((fn, C), ft)) (\text{cfields } c) @$   
 $(\text{if } C = \text{Object then } [] \text{ else } \text{fields } G \text{ (super } c))$

$\langle \text{proof} \rangle$

**lemma** *fields-norec*:

$\llbracket \text{class } G \text{ fd} = \text{Some } c; \text{ws-prog } G; \text{table-of } (\text{cfields } c) \text{ fn} = \text{Some } f \rrbracket$   
 $\implies \text{table-of } (\text{fields } G \text{ fd}) (\text{fn}, \text{fd}) = \text{Some } f$

$\langle \text{proof} \rangle$

**lemma** *table-of-fieldsD*:

$\text{table-of } (\text{map } (\lambda(fn, ft). ((fn, C), ft)) (\text{cfields } c)) \text{ efn} = \text{Some } f$   
 $\implies (\text{declclassf efn}) = C \wedge \text{table-of } (\text{cfields } c) (\text{fname efn}) = \text{Some } f$

$\langle \text{proof} \rangle$

**lemma** *fields-declC*:

$\llbracket \text{table-of } (\text{fields } G \text{ } C) \text{ efn} = \text{Some } f; \text{ws-prog } G; \text{is-class } G \text{ } C \rrbracket \implies$   
 $(\exists d. \text{class } G (\text{declclassf efn}) = \text{Some } d \wedge$   
 $\text{table-of } (\text{cfields } d) (\text{fname efn}) = \text{Some } f) \wedge$   
 $G \vdash C \preceq_C (\text{declclassf efn}) \wedge \text{table-of } (\text{fields } G (\text{declclassf efn})) \text{ efn} = \text{Some } f$

$\langle \text{proof} \rangle$

**lemma** *fields-emptyI*:  $\bigwedge y. \llbracket ws\text{-prog } G; \text{class } G \ C = \text{Some } c; cfields \ c = [] \rrbracket;$   
 $C \neq \text{Object} \longrightarrow \text{class } G \ (\text{super } c) = \text{Some } y \wedge \text{fields } G \ (\text{super } c) = [] \implies$   
 $\text{fields } G \ C = []$   
 $\langle \text{proof} \rangle$

**lemma** *fields-mono-lemma*:  
 $\llbracket x \in \text{set } (\text{fields } G \ C); G \vdash D \preceq_C C; ws\text{-prog } G \rrbracket$   
 $\implies x \in \text{set } (\text{fields } G \ D)$   
 $\langle \text{proof} \rangle$

**lemma** *ws-unique-fields-lemma*:  
 $\llbracket (efn, fd) \in \text{set } (\text{fields } G \ (\text{super } c)); fc \in \text{set } (cfields \ c); ws\text{-prog } G;$   
 $fname \ efn = fname \ fc; declclassf \ efn = C;$   
 $\text{class } G \ C = \text{Some } c; C \neq \text{Object}; \text{class } G \ (\text{super } c) = \text{Some } d \rrbracket \implies R$   
 $\langle \text{proof} \rangle$

**lemma** *ws-unique-fields*:  $\llbracket is\text{-class } G \ C; ws\text{-prog } G;$   
 $\bigwedge C \ c. \llbracket \text{class } G \ C = \text{Some } c \rrbracket \implies \text{unique } (cfields \ c) \rrbracket \implies$   
 $\text{unique } (\text{fields } G \ C)$   
 $\langle \text{proof} \rangle$

## 26 accfield

**lemma** *accfield-fields*:  
 $\text{accfield } G \ S \ C \ fn = \text{Some } f$   
 $\implies \text{table-of } (\text{fields } G \ C) \ (fn, \text{declclass } f) = \text{Some } (fld \ f)$   
 $\langle \text{proof} \rangle$

**lemma** *accfield-declC-is-class*:  
 $\llbracket is\text{-class } G \ C; \text{accfield } G \ S \ C \ en = \text{Some } (fd, f); ws\text{-prog } G \rrbracket \implies$   
 $is\text{-class } G \ fd$   
 $\langle \text{proof} \rangle$

**lemma** *accfield-accessibleD*:  
 $\text{accfield } G \ S \ C \ fn = \text{Some } f \implies G \vdash \text{Field } fn \ f \text{ of } C \text{ accessible-from } S$   
 $\langle \text{proof} \rangle$

## 27 is methd

**lemma** *is-methdI*:  
 $\llbracket \text{class } G \ C = \text{Some } y; \text{methd } G \ C \ sig = \text{Some } b \rrbracket \implies is\text{-methd } G \ C \ sig$   
 $\langle \text{proof} \rangle$

**lemma** *is-methdD*:  
 $is\text{-methd } G \ C \ sig \implies \text{class } G \ C \neq \text{None} \wedge \text{methd } G \ C \ sig \neq \text{None}$   
 $\langle \text{proof} \rangle$

**lemma** *finite-is-methd*:  
 $ws\text{-prog } G \implies \text{finite } (\text{Collect } (\text{split } (is\text{-methd } G)))$

$\langle proof \rangle$

### calculation of the superclasses of a class

#### constdefs

```

superclasses:: prog  $\Rightarrow$  qtname  $\Rightarrow$  qtname set
superclasses G C  $\equiv$  class-rec (G,C) {}
                    ( $\lambda$  C c superclss. (if C=Object
                                     then {}
                                     else insert (super c) superclss))

```

**lemma** superclasses-rec:  $\llbracket \text{class } G \ C = \text{Some } c; \text{ws-prog } G \rrbracket \Longrightarrow$

```

superclasses G C
= (if (C=Object)
    then {}
    else insert (super c) (superclasses G (super c)))

```

$\langle proof \rangle$

**lemma** superclasses-mono:

```

 $\llbracket G \vdash C \prec_C D; \text{ws-prog } G; \text{class } G \ C = \text{Some } c; \\
\bigwedge C \ c. \llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object} \rrbracket \Longrightarrow \exists \text{ sc. class } G \ (\text{super } c) = \text{Some } \text{sc}; \\
x \in \text{superclasses } G \ D \rrbracket \Longrightarrow x \in \text{superclasses } G \ C \\
\langle proof \rangle$ 
```

**lemma** subclsEval:

```

 $\llbracket G \vdash C \prec_C D; \text{ws-prog } G; \text{class } G \ C = \text{Some } c; \\
\bigwedge C \ c. \llbracket \text{class } G \ C = \text{Some } c; C \neq \text{Object} \rrbracket \Longrightarrow \exists \text{ sc. class } G \ (\text{super } c) = \text{Some } \text{sc} \\
\rrbracket \Longrightarrow D \in \text{superclasses } G \ C \\
\langle proof \rangle$ 
```

**end**



## Chapter 11

# WellType

## 28 Well-typedness of Java programs

**theory** *WellType* **imports** *DeclConcepts* **begin**

improvements over Java Specification 1.0:

- methods of Object can be called upon references of interface or array type

simplifications:

- the type rules include all static checks on statements and expressions, e.g. definedness of names (of parameters, locals, fields, methods)

design issues:

- unified type judgment for statements, variables, expressions, expression lists
- statements are typed like expressions with dummy type Void
- the typing rules take an extra argument that is capable of determining the dynamic type of objects. Therefore, they can be used for both checking static types and determining runtime types in transition semantics.

**types** *lenv*

= (*lname*, *ty*) *table* — local variables, including This and Result

**record** *env* =

*prg*:: *prog* — program  
*cls*:: *qtname* — current package and class name  
*lcl*:: *lenv* — local environment

**translations**

*lenv* <= (*type*) (*lname*, *ty*) *table*  
*lenv* <= (*type*) *lname*  $\Rightarrow$  *ty option*  
*env* <= (*type*) (*prg*::*prog*, *cls*::*qtname*, *lcl*::*lenv*)  
*env* <= (*type*) (*prg*::*prog*, *cls*::*qtname*, *lcl*::*lenv*, ...::'*a*)

**syntax**

*pkg* :: *env*  $\Rightarrow$  *pname* — select the current package from an environment

**translations**

*pkg e* == *pid (cls e)*

### Static overloading: maximally specific methods

**types**

*emhead* = *ref-ty*  $\times$  *mhead*

— Some mnemonic selectors for *emhead*

**constdefs**

*declrefT* :: *emhead*  $\Rightarrow$  *ref-ty*  
*declrefT*  $\equiv$  *fst*

*mhd* :: *emhead*  $\Rightarrow$  *mhead*  
*mhd*  $\equiv$  *snd*

**lemma** *declrefT-simp[simp]:declrefT (r,m) = r*

$\langle \text{proof} \rangle$

**lemma** *mhd-simp*[simp]: *mhd* (*r,m*) = *m*

$\langle \text{proof} \rangle$

**lemma** *static-mhd-simp*[simp]: *static* (*mhd m*) = *is-static m*

$\langle \text{proof} \rangle$

**lemma** *mhd-resTy-simp* [simp]: *resTy* (*mhd m*) = *resTy m*

$\langle \text{proof} \rangle$

**lemma** *mhd-is-static-simp* [simp]: *is-static* (*mhd m*) = *is-static m*

$\langle \text{proof} \rangle$

**lemma** *mhd-accmodi-simp* [simp]: *accmodi* (*mhd m*) = *accmodi m*

$\langle \text{proof} \rangle$

**consts**

*cmheads* :: *prog*  $\Rightarrow$  *qtname*  $\Rightarrow$  *qtname*  $\Rightarrow$  *sig*  $\Rightarrow$  *emhead set*

*Objectmheads* :: *prog*  $\Rightarrow$  *qtname*  $\Rightarrow$  *sig*  $\Rightarrow$  *emhead set*

*accObjectmheads*:: *prog*  $\Rightarrow$  *qtname*  $\Rightarrow$  *ref-ty*  $\Rightarrow$  *sig*  $\Rightarrow$  *emhead set*

*mheads* :: *prog*  $\Rightarrow$  *qtname*  $\Rightarrow$  *ref-ty*  $\Rightarrow$  *sig*  $\Rightarrow$  *emhead set*

**defs**

*cmheads-def*:

*cmheads G S C*

$\equiv \lambda \text{sig}. (\lambda (\text{Cls}, \text{mthd}). (\text{ClassT Cls}, (\text{mhead mthd}))) \text{ 'o2s (accmethd G S C sig)}$

*Objectmheads-def*:

*Objectmheads G S*

$\equiv \lambda \text{sig}. (\lambda (\text{Cls}, \text{mthd}). (\text{ClassT Cls}, (\text{mhead mthd})))$

$\text{ 'o2s (filter-tab } (\lambda \text{sig m}. \text{accmodi m} \neq \text{Private}) (\text{accmethd G S Object}) \text{ sig)}$

*accObjectmheads-def*:

*accObjectmheads G S T*

$\equiv \text{if } G \vdash \text{RefT T accessible-in (pid S)}$

$\text{ then Objectmheads G S}$

$\text{ else } \lambda \text{sig}. \{\}$

**primrec**

*mheads G S NullT* =  $(\lambda \text{sig}. \{\})$

*mheads G S (IfaceT I)* =  $(\lambda \text{sig}. (\lambda (I, h). (\text{IfaceT I}, h)))$

$\text{ 'accimethds G (pid S) I sig} \cup$

$\text{ accObjectmheads G S (IfaceT I) sig)}$

*mheads G S (ClassT C)* = *cmheads G S C*

*mheads G S (ArrayT T)* = *accObjectmheads G S (ArrayT T)*

**constdefs**

— applicable methods, cf. 15.11.2.1

*appl-methds* :: *prog*  $\Rightarrow$  *qtname*  $\Rightarrow$  *ref-ty*  $\Rightarrow$  *sig*  $\Rightarrow$  (*emhead*  $\times$  *ty list*) *set*

*appl-methds G S rt*  $\equiv \lambda \text{sig}.$

$\{(mh, pTs') \mid mh \text{ pTs}'. mh \in mheads G S rt \text{ (name=name sig, parTs=pTs')} \wedge$   
 $G \vdash (\text{parTs sig}) [\preceq] pTs'\}$

— more specific methods, cf. 15.11.2.2

*more-spec* :: *prog*  $\Rightarrow$  *emhead*  $\times$  *ty list*  $\Rightarrow$  *emhead*  $\times$  *ty list*  $\Rightarrow$  *bool*

*more-spec G*  $\equiv \lambda (mh, pTs). \lambda (mh', pTs'). G \vdash pTs [\preceq] pTs'$

— maximally specific methods, cf. 15.11.2.2

$max-spec \quad :: prog \Rightarrow qname \Rightarrow ref-ty \Rightarrow sig \Rightarrow (emhead \times ty\ list) \quad set$

$max-spec\ G\ S\ rt\ sig \equiv \{m. m \in appl-methds\ G\ S\ rt\ sig \wedge$   
 $(\forall m' \in appl-methds\ G\ S\ rt\ sig. more-spec\ G\ m'\ m \longrightarrow m' = m)\}$

**lemma** *max-spec2appl-meths*:

$x \in max-spec\ G\ S\ T\ sig \implies x \in appl-methds\ G\ S\ T\ sig$   
 $\langle proof \rangle$

**lemma** *appl-methsD*:  $(mh, pTs') \in appl-methds\ G\ S\ T\ (\llbracket name = mn, parTs = pTs \rrbracket) \implies$   
 $mh \in mheads\ G\ S\ T\ (\llbracket name = mn, parTs = pTs \rrbracket) \wedge G \vdash pTs [\preceq] pTs'$   
 $\langle proof \rangle$

**lemma** *max-spec2mheads*:

$max-spec\ G\ S\ rt\ (\llbracket name = mn, parTs = pTs \rrbracket) = insert\ (mh, pTs')\ A$   
 $\implies mh \in mheads\ G\ S\ rt\ (\llbracket name = mn, parTs = pTs \rrbracket) \wedge G \vdash pTs [\preceq] pTs'$   
 $\langle proof \rangle$

**constdefs**

$empty-dt :: dyn-ty$

$empty-dt \equiv \lambda a. None$

$invmode :: ('a::type)member-scheme \Rightarrow expr \Rightarrow inv-mode$

$invmode\ m\ e \equiv if\ is-static\ m$

$\quad then\ Static$

$\quad else\ if\ e = Super\ then\ SuperM\ else\ IntVir$

**lemma** *invmode-nonstatic* [simp]:

$invmode\ (\llbracket access = a, static = False, \dots = x \rrbracket)\ (Acc\ (LVar\ e)) = IntVir$   
 $\langle proof \rangle$

**lemma** *invmode-Static-eq* [simp]:  $(invmode\ m\ e = Static) = is-static\ m$

$\langle proof \rangle$

**lemma** *invmode-IntVir-eq*:  $(invmode\ m\ e = IntVir) = (\neg(is-static\ m) \wedge e \neq Super)$

$\langle proof \rangle$

**lemma** *Null-staticD*:

$a' = Null \longrightarrow (is-static\ m) \implies invmode\ m\ e = IntVir \longrightarrow a' \neq Null$   
 $\langle proof \rangle$

## Typing for unary operations

**consts** *unop-type* :: *unop*  $\Rightarrow$  *prim-ty*

**primrec**

*unop-type* *UPlus* = *Integer*

*unop-type UMinus* = *Integer*  
*unop-type UBitNot* = *Integer*  
*unop-type UNot* = *Boolean*

**consts** *wt-unop* :: *unop*  $\Rightarrow$  *ty*  $\Rightarrow$  *bool*

**primrec**

*wt-unop UPlus* *t* = (*t* = *PrimT Integer*)  
*wt-unop UMinus* *t* = (*t* = *PrimT Integer*)  
*wt-unop UBitNot* *t* = (*t* = *PrimT Integer*)  
*wt-unop UNot* *t* = (*t* = *PrimT Boolean*)

## Typing for binary operations

**consts** *binop-type* :: *binop*  $\Rightarrow$  *prim-ty*

**primrec**

*binop-type Mul* = *Integer*  
*binop-type Div* = *Integer*  
*binop-type Mod* = *Integer*  
*binop-type Plus* = *Integer*  
*binop-type Minus* = *Integer*  
*binop-type LShift* = *Integer*  
*binop-type RShift* = *Integer*  
*binop-type RShiftU* = *Integer*  
*binop-type Less* = *Boolean*  
*binop-type Le* = *Boolean*  
*binop-type Greater* = *Boolean*  
*binop-type Ge* = *Boolean*  
*binop-type Eq* = *Boolean*  
*binop-type Neq* = *Boolean*  
*binop-type BitAnd* = *Integer*  
*binop-type And* = *Boolean*  
*binop-type BitXor* = *Integer*  
*binop-type Xor* = *Boolean*  
*binop-type BitOr* = *Integer*  
*binop-type Or* = *Boolean*  
*binop-type CondAnd* = *Boolean*  
*binop-type CondOr* = *Boolean*

**consts** *wt-binop* :: *prog*  $\Rightarrow$  *binop*  $\Rightarrow$  *ty*  $\Rightarrow$  *ty*  $\Rightarrow$  *bool*

**primrec**

*wt-binop G Mul* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop G Div* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop G Mod* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop G Plus* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop G Minus* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop G LShift* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop G RShift* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop G RShiftU* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop G Less* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop G Le* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop G Greater* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop G Ge* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop G Eq* *t1 t2* = ( $G \vdash t1 \preceq t2 \vee G \vdash t2 \preceq t1$ )  
*wt-binop G Neq* *t1 t2* = ( $G \vdash t1 \preceq t2 \vee G \vdash t2 \preceq t1$ )  
*wt-binop G BitAnd* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop G And* *t1 t2* = ((*t1* = *PrimT Boolean*)  $\wedge$  (*t2* = *PrimT Boolean*))  
*wt-binop G BitXor* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))  
*wt-binop G Xor* *t1 t2* = ((*t1* = *PrimT Boolean*)  $\wedge$  (*t2* = *PrimT Boolean*))  
*wt-binop G BitOr* *t1 t2* = ((*t1* = *PrimT Integer*)  $\wedge$  (*t2* = *PrimT Integer*))

$wt\_binop\ G\ Or\ \quad t1\ t2 = ((t1 = PrimT\ Boolean) \wedge (t2 = PrimT\ Boolean))$   
 $wt\_binop\ G\ CondAnd\ t1\ t2 = ((t1 = PrimT\ Boolean) \wedge (t2 = PrimT\ Boolean))$   
 $wt\_binop\ G\ CondOr\ t1\ t2 = ((t1 = PrimT\ Boolean) \wedge (t2 = PrimT\ Boolean))$

## Typing for terms

**types**  $tys = \quad ty + ty\ list$

**translations**

$tys \leq (type)\ ty + ty\ list$

**consts**

$wt :: (env \times dyn\_ty \times term \times tys)\ set$

**syntax**

$wt :: env \Rightarrow dyn\_ty \Rightarrow [term, tys] \Rightarrow bool\ (-, | = :- [51, 51, 51, 51]\ 50)$   
 $wt\_stmt :: env \Rightarrow dyn\_ty \Rightarrow stmt \Rightarrow bool\ (-, | = :- <> [51, 51, 51]\ 50)$   
 $ty\_expr :: env \Rightarrow dyn\_ty \Rightarrow [expr, ty] \Rightarrow bool\ (-, | = :- [51, 51, 51, 51]\ 50)$   
 $ty\_var :: env \Rightarrow dyn\_ty \Rightarrow [var, ty] \Rightarrow bool\ (-, | = :- [51, 51, 51, 51]\ 50)$   
 $ty\_exprs :: env \Rightarrow dyn\_ty \Rightarrow [expr\ list,$   
 $\quad ty\ list] \Rightarrow bool\ (-, | = :- \# [51, 51, 51, 51]\ 50)$

**syntax** ( $xsymbols$ )

$wt :: env \Rightarrow dyn\_ty \Rightarrow [term, tys] \Rightarrow bool\ (-, | = :- [51, 51, 51, 51]\ 50)$   
 $wt\_stmt :: env \Rightarrow dyn\_ty \Rightarrow stmt \Rightarrow bool\ (-, | = :- \surd [51, 51, 51]\ 50)$   
 $ty\_expr :: env \Rightarrow dyn\_ty \Rightarrow [expr, ty] \Rightarrow bool\ (-, | = :- [51, 51, 51, 51]\ 50)$   
 $ty\_var :: env \Rightarrow dyn\_ty \Rightarrow [var, ty] \Rightarrow bool\ (-, | = :- [51, 51, 51, 51]\ 50)$   
 $ty\_exprs :: env \Rightarrow dyn\_ty \Rightarrow [expr\ list,$   
 $\quad ty\ list] \Rightarrow bool\ (-, | = :- \doteq [51, 51, 51, 51]\ 50)$

**translations**

$E, dt \models t :: T == (E, dt, t, T) \in wt$   
 $E, dt \models s :: \surd == E, dt \models In1r\ s :: Inl\ (PrimT\ Void)$   
 $E, dt \models e :: -T == E, dt \models In1l\ e :: Inl\ T$   
 $E, dt \models e :: =T == E, dt \models In2\ e :: Inl\ T$   
 $E, dt \models e :: \doteq T == E, dt \models In3\ e :: Inr\ T$

**syntax**

$wt- :: env \Rightarrow [term, tys] \Rightarrow bool\ (-, | = :- [51, 51, 51]\ 50)$   
 $wt\_stmt- :: env \Rightarrow stmt \Rightarrow bool\ (-, | = :- <> [51, 51]\ 50)$   
 $ty\_expr- :: env \Rightarrow [expr, ty] \Rightarrow bool\ (-, | = :- [51, 51, 51]\ 50)$   
 $ty\_var- :: env \Rightarrow [var, ty] \Rightarrow bool\ (-, | = :- [51, 51, 51]\ 50)$   
 $ty\_exprs- :: env \Rightarrow [expr\ list,$   
 $\quad ty\ list] \Rightarrow bool\ (-, | = :- \# [51, 51, 51]\ 50)$

**syntax** ( $xsymbols$ )

$wt- :: env \Rightarrow [term, tys] \Rightarrow bool\ (+, | = :- [51, 51, 51]\ 50)$   
 $wt\_stmt- :: env \Rightarrow stmt \Rightarrow bool\ (+, | = :- \surd [51, 51]\ 50)$   
 $ty\_expr- :: env \Rightarrow [expr, ty] \Rightarrow bool\ (+, | = :- [51, 51, 51]\ 50)$   
 $ty\_var- :: env \Rightarrow [var, ty] \Rightarrow bool\ (+, | = :- [51, 51, 51]\ 50)$   
 $ty\_exprs- :: env \Rightarrow [expr\ list,$   
 $\quad ty\ list] \Rightarrow bool\ (+, | = :- \doteq [51, 51, 51]\ 50)$

**translations**

$E \vdash t :: T == E, empty\_dt \models t :: T$   
 $E \vdash s :: \surd == E \vdash In1r\ s :: Inl\ (PrimT\ Void)$   
 $E \vdash e :: -T == E \vdash In1l\ e :: Inl\ T$   
 $E \vdash e :: =T == E \vdash In2\ e :: Inl\ T$   
 $E \vdash e :: \doteq T == E \vdash In3\ e :: Inr\ T$

**inductive wt intros**

— well-typed statements

$$\text{Skip: } E, dt \models \text{Skip} :: \checkmark$$

$$\text{Expr: } \llbracket E, dt \models e :: - T \rrbracket \implies E, dt \models \text{Expr } e :: \checkmark$$

— cf. 14.6

$$\text{Lab: } E, dt \models c :: \checkmark \implies E, dt \models l \cdot c :: \checkmark$$

$$\text{Comp: } \llbracket E, dt \models c1 :: \checkmark; E, dt \models c2 :: \checkmark \rrbracket \implies E, dt \models c1 ;; c2 :: \checkmark$$

— cf. 14.8

$$\text{If: } \llbracket E, dt \models e :: - \text{PrimT Boolean}; E, dt \models c1 :: \checkmark; E, dt \models c2 :: \checkmark \rrbracket \implies E, dt \models \text{If}(e) \ c1 \ \text{Else} \ c2 :: \checkmark$$

— cf. 14.10

$$\text{Loop: } \llbracket E, dt \models e :: - \text{PrimT Boolean}; E, dt \models c :: \checkmark \rrbracket \implies E, dt \models l \cdot \text{While}(e) \ c :: \checkmark$$

— cf. 14.13, 14.15, 14.16

$$\text{Jmp: } E, dt \models \text{Jmp } \text{jump} :: \checkmark$$

— cf. 14.16

$$\text{Throw: } \llbracket E, dt \models e :: - \text{Class } tn; \text{prg } E \vdash tn \preceq_C \text{SXcpt Throwable} \rrbracket \implies E, dt \models \text{Throw } e :: \checkmark$$

— cf. 14.18

$$\text{Try: } \llbracket E, dt \models c1 :: \checkmark; \text{prg } E \vdash tn \preceq_C \text{SXcpt Throwable}; \text{lcl } E \ (V\text{Name } vn) = \text{None}; E \ (\text{lcl} := \text{lcl } E \ (V\text{Name } vn \mapsto \text{Class } tn)) \rrbracket, dt \models c2 :: \checkmark \implies E, dt \models \text{Try } c1 \ \text{Catch}(tn \ vn) \ c2 :: \checkmark$$

— cf. 14.18

$$\text{Fin: } \llbracket E, dt \models c1 :: \checkmark; E, dt \models c2 :: \checkmark \rrbracket \implies E, dt \models c1 \ \text{Finally} \ c2 :: \checkmark$$

$$\text{Init: } \llbracket \text{is-class } (\text{prg } E) \ C \rrbracket \implies E, dt \models \text{Init } C :: \checkmark$$

— *Init* is created on the fly during evaluation (see Eval.thy). The class isn't necessarily accessible from the points *Init* is called. Therefor we only demand *is-class* and not *is-acc-class* here.

— well-typed expressions

— cf. 15.8

$$\text{NewC: } \llbracket \text{is-acc-class } (\text{prg } E) \ (\text{pkg } E) \ C \rrbracket \implies E, dt \models \text{NewC } C :: - \text{Class } C$$

— cf. 15.9

$$\text{NewA: } \llbracket \text{is-acc-type } (\text{prg } E) \ (\text{pkg } E) \ T; E, dt \models i :: - \text{PrimT Integer} \rrbracket \implies$$

$$E, dt \models \text{New } T[i] :: - T. []$$

— cf. 15.15

$$\begin{aligned} \text{Cast: } \llbracket E, dt \models e :: - T; \text{ is-acc-type } (\text{prg } E) (\text{pkg } E) T'; \\ \text{prg } E \vdash T \preceq^? T' \rrbracket \implies \\ E, dt \models \text{Cast } T' e :: - T' \end{aligned}$$

— cf. 15.19.2

$$\begin{aligned} \text{Inst: } \llbracket E, dt \models e :: - \text{RefT } T; \text{ is-acc-type } (\text{prg } E) (\text{pkg } E) (\text{RefT } T'); \\ \text{prg } E \vdash \text{RefT } T \preceq^? \text{RefT } T' \rrbracket \implies \\ E, dt \models_e \text{InstOf } T' :: - \text{PrimT Boolean} \end{aligned}$$

— cf. 15.7.1

$$\begin{aligned} \text{Lit: } \llbracket \text{typeof } dt \ x = \text{Some } T \rrbracket \implies \\ E, dt \models \text{Lit } x :: - T \end{aligned}$$

$$\begin{aligned} \text{UnOp: } \llbracket E, dt \models e :: - T_e; \text{ wt-unop } \text{unop } T_e; T = \text{PrimT } (\text{unop-type } \text{unop}) \rrbracket \\ \implies \\ E, dt \models \text{UnOp } \text{unop } e :: - T \end{aligned}$$

$$\begin{aligned} \text{BinOp: } \llbracket E, dt \models e1 :: - T1; E, dt \models e2 :: - T2; \text{ wt-binop } (\text{prg } E) \text{ binop } T1 \ T2; \\ T = \text{PrimT } (\text{binop-type } \text{binop}) \rrbracket \\ \implies \\ E, dt \models \text{BinOp } \text{binop } e1 \ e2 :: - T \end{aligned}$$

— cf. 15.10.2, 15.11.1

$$\begin{aligned} \text{Super: } \llbracket \text{lcl } E \ \text{This} = \text{Some } (\text{Class } C); C \neq \text{Object}; \\ \text{class } (\text{prg } E) \ C = \text{Some } c \rrbracket \implies \\ E, dt \models \text{Super} :: - \text{Class } (\text{super } c) \end{aligned}$$

— cf. 15.13.1, 15.10.1, 15.12

$$\begin{aligned} \text{Acc: } \llbracket E, dt \models va :: = T \rrbracket \implies \\ E, dt \models \text{Acc } va :: - T \end{aligned}$$

— cf. 15.25, 15.25.1

$$\begin{aligned} \text{Ass: } \llbracket E, dt \models va :: = T; va \neq \text{LVar } \text{This}; \\ E, dt \models v :: - T'; \\ \text{prg } E \vdash T' \preceq T \rrbracket \implies \\ E, dt \models va :: = v :: - T' \end{aligned}$$

— cf. 15.24

$$\begin{aligned} \text{Cond: } \llbracket E, dt \models e0 :: - \text{PrimT Boolean}; \\ E, dt \models e1 :: - T1; E, dt \models e2 :: - T2; \\ \text{prg } E \vdash T1 \preceq T2 \wedge T = T2 \vee \text{prg } E \vdash T2 \preceq T1 \wedge T = T1 \rrbracket \implies \\ E, dt \models e0 \ ? \ e1 : e2 :: - T \end{aligned}$$

— cf. 15.11.1, 15.11.2, 15.11.3

$$\begin{aligned} \text{Call: } \llbracket E, dt \models e :: - \text{RefT } \text{statT}; \\ E, dt \models ps :: \dot{=} pTs; \\ \text{max-spec } (\text{prg } E) (\text{cls } E) \ \text{statT } (\text{name} = mn, \text{parTs} = pTs) \\ = \{((\text{statDeclT}, m), pTs')\} \\ \rrbracket \implies \\ E, dt \models \{ \text{cls } E, \text{statT}, \text{invmode } m \ e \} e \cdot mn(\{pTs'\} ps) :: - (\text{resTy } m) \end{aligned}$$

*Methd*:  $\llbracket \text{is-class } (\text{prg } E) \ C;$

$$\begin{aligned} \text{methd } (\text{prg } E) \ C \ \text{sig} = \text{Some } m; \\ E, dt \models \text{Body } (\text{declclass } m) (\text{stmt } (\text{mbody } (\text{methd } m))) :: - T \rrbracket \implies \\ E, dt \models \text{Methd } C \ \text{sig} :: - T \end{aligned}$$

— The class  $C$  is the dynamic class of the method call (cf. Eval.thy). It hasn't got to be directly accessible



from the current package  $pkg\ E$ . Only the static class must be accessible (ensured indirectly by  $Call$ ). Note that  $l$  is just a dummy value. It is only used in the smallstep semantics. To proof typesafety directly for the smallstep semantics we would have to assume conformance of  $l$  here!

*Body*:  $\llbracket is\_class\ (prg\ E)\ D;$   
 $E, dt \models blk::\checkmark;$   
 $(lcl\ E)\ Result = Some\ T;$   
 $is\_type\ (prg\ E)\ T \rrbracket \implies$   
 $E, dt \models Body\ D\ blk:: - T$

— The class  $D$  implementing the method must not directly be accessible from the current package  $pkg\ E$ , but can also be indirectly accessible due to inheritance (ensured in  $Call$ ) The result type hasn't got to be accessible in Java! (If it is not accessible you can only assign it to Object). For dummy value  $l$  see rule *Method*.

— well-typed variables

— cf. 15.13.1  
 $LVar$ :  $\llbracket lcl\ E\ vn = Some\ T;\ is\_acc\_type\ (prg\ E)\ (pkg\ E)\ T \rrbracket \implies$   
 $E, dt \models LVar\ vn:: = T$

— cf. 15.10.1  
 $FVar$ :  $\llbracket E, dt \models e:: - Class\ C;$   
 $accfield\ (prg\ E)\ (cls\ E)\ C\ fn = Some\ (statDeclC, f) \rrbracket \implies$   
 $E, dt \models \{ cls\ E, statDeclC, is\_static\ f \} e..fn:: = (type\ f)$

— cf. 15.12  
 $AVar$ :  $\llbracket E, dt \models e:: - T.; \rrbracket$   
 $E, dt \models i:: - PrimT\ Integer \rrbracket \implies$   
 $E, dt \models e.[i]:: = T$

— well-typed expression lists

— cf. 15.11.???  
 $Nil$ :  $E, dt \models []:: \doteq []$

— cf. 15.11.???  
 $Cons$ :  $\llbracket E, dt \models e:: - T;$   
 $E, dt \models es:: \doteq Ts \rrbracket \implies$   
 $E, dt \models e \# es:: \doteq T \# Ts$

**declare** *not-None-eq* [*simp del*]  
**declare** *split-if* [*split del*] *split-if-asm* [*split del*]  
**declare** *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]  
 $\langle ML \rangle$

**inductive-cases** *wt-elim-cases* [*cases set*]:  
 $E, dt \models In2\ (LVar\ vn) \quad :: T$   
 $E, dt \models In2\ (\{ accC, statDeclC, s \} e..fn) :: T$   
 $E, dt \models In2\ (e.[i]) \quad :: T$   
 $E, dt \models In1l\ (NewC\ C) \quad :: T$   
 $E, dt \models In1l\ (New\ T'[i]) \quad :: T$   
 $E, dt \models In1l\ (Cast\ T'\ e) \quad :: T$   
 $E, dt \models In1l\ (e\ InstOf\ T') \quad :: T$   
 $E, dt \models In1l\ (Lit\ x) \quad :: T$   
 $E, dt \models In1l\ (UnOp\ unop\ e) \quad :: T$   
 $E, dt \models In1l\ (BinOp\ binop\ e1\ e2) \quad :: T$   
 $E, dt \models In1l\ (Super) \quad :: T$   
 $E, dt \models In1l\ (Acc\ va) \quad :: T$   
 $E, dt \models In1l\ (Ass\ va\ v) \quad :: T$

$$\begin{array}{ll}
E, dt \models_{In1l} (e0 \text{ ? } e1 : e2) & :: T \\
E, dt \models_{In1l} (\{accC, statT, mode\} e \cdot mn(\{pT\}p)) & :: T \\
E, dt \models_{In1l} (Methd \ C \ sig) & :: T \\
E, dt \models_{In1l} (Body \ D \ blk) & :: T \\
E, dt \models_{In3} ([]) & :: Ts \\
E, dt \models_{In3} (e \# es) & :: Ts \\
E, dt \models_{In1r} Skip & :: x \\
E, dt \models_{In1r} (Expr \ e) & :: x \\
E, dt \models_{In1r} (c1 ;; c2) & :: x \\
E, dt \models_{In1r} (l \cdot c) & :: x \\
E, dt \models_{In1r} (If(e) \ c1 \ Else \ c2) & :: x \\
E, dt \models_{In1r} (l \cdot While(e) \ c) & :: x \\
E, dt \models_{In1r} (Jmp \ jump) & :: x \\
E, dt \models_{In1r} (Throw \ e) & :: x \\
E, dt \models_{In1r} (Try \ c1 \ Catch(tn \ vn) \ c2) & :: x \\
E, dt \models_{In1r} (c1 \ Finally \ c2) & :: x \\
E, dt \models_{In1r} (Init \ C) & :: x
\end{array}$$

**declare** *not-None-eq* [simp]  
**declare** *split-if* [split] *split-if-asm* [split]  
**declare** *split-paired-All* [simp] *split-paired-Ex* [simp]  
 $\langle ML \rangle$

**lemma** *is-acc-class-is-accessible*:  
 $is-acc-class \ G \ P \ C \Longrightarrow G \vdash (Class \ C) \text{ accessible-in } P$   
 $\langle proof \rangle$

**lemma** *is-acc-iface-is-iface*:  $is-acc-iface \ G \ P \ I \Longrightarrow is-iface \ G \ I$   
 $\langle proof \rangle$

**lemma** *is-acc-iface-Iface-is-accessible*:  
 $is-acc-iface \ G \ P \ I \Longrightarrow G \vdash (Iface \ I) \text{ accessible-in } P$   
 $\langle proof \rangle$

**lemma** *is-acc-type-is-type*:  $is-acc-type \ G \ P \ T \Longrightarrow is-type \ G \ T$   
 $\langle proof \rangle$

**lemma** *is-acc-iface-is-accessible*:  
 $is-acc-type \ G \ P \ T \Longrightarrow G \vdash T \text{ accessible-in } P$   
 $\langle proof \rangle$

**lemma** *wt-Methd-is-methd*:  
 $E \vdash_{In1l} (Methd \ C \ sig) :: T \Longrightarrow is-methd \ (prg \ E) \ C \ sig$   
 $\langle proof \rangle$

Special versions of some typing rules, better suited to pattern match the conclusion (no selectors in the conclusion)

**lemma** *wt-Call*:  
 $\llbracket E, dt \models e :: -RefT \ statT; E, dt \models ps :: pTs;$   
 $\max-spec \ (prg \ E) \ (cls \ E) \ statT \ (\llbracket name = mn, parTs = pTs \rrbracket)$   
 $= \{((statDeclC, m), pTs')\}; rT = (resTy \ m); accC = cls \ E;$   
 $mode = invmode \ m \ e \rrbracket \Longrightarrow E, dt \models \{accC, statT, mode\} e \cdot mn(\{pTs'\}ps) :: -rT$   
 $\langle proof \rangle$

**lemma** *invocationTypeExpr-noClassD*:

$\llbracket E \vdash e :: -\text{RefT } \text{statT} \rrbracket$   
 $\implies (\forall \text{ statC}. \text{statT} \neq \text{ClassT } \text{statC}) \longrightarrow \text{invmode } m \ e \neq \text{SuperM}$   
 $\langle \text{proof} \rangle$

**lemma** *wt-Super*:

$\llbracket \text{lcl } E \text{ This} = \text{Some } (\text{Class } C); C \neq \text{Object}; \text{class } (\text{prg } E) \ C = \text{Some } c; D = \text{super } c \rrbracket$   
 $\implies E, dt \models \text{Super} :: -\text{Class } D$   
 $\langle \text{proof} \rangle$

**lemma** *wt-FVar*:

$\llbracket E, dt \models e :: -\text{Class } C; \text{accfield } (\text{prg } E) \ (\text{cls } E) \ C \text{ fn} = \text{Some } (\text{statDeclC}, f);$   
 $\text{sf} = \text{is-static } f; fT = (\text{type } f); \text{accC} = \text{cls } E \rrbracket$   
 $\implies E, dt \models \{ \text{accC}, \text{statDeclC}, \text{sf} \} e.. \text{fn} :: fT$   
 $\langle \text{proof} \rangle$

**lemma** *wt-init* [iff]:  $E, dt \models \text{Init } C :: \surd = \text{is-class } (\text{prg } E) \ C$

$\langle \text{proof} \rangle$

**declare** *wt.Skip* [iff]

**lemma** *wt-StatRef*:

$\text{is-acc-type } (\text{prg } E) \ (\text{pkg } E) \ (\text{RefT } rt) \implies E \vdash \text{StatRef } rt :: -\text{RefT } rt$   
 $\langle \text{proof} \rangle$

**lemma** *wt-Inj-elim*:

$\bigwedge E. E, dt \models t :: U \implies \text{case } t \text{ of}$   
 $\quad \text{In1 } ec \Rightarrow (\text{case } ec \text{ of}$   
 $\quad \quad \text{Inl } e \Rightarrow \exists T. U = \text{Inl } T$   
 $\quad \quad | \text{Inr } s \Rightarrow U = \text{Inl } (\text{PrimT } \text{Void}))$   
 $\quad | \text{In2 } e \Rightarrow (\exists T. U = \text{Inl } T)$   
 $\quad | \text{In3 } e \Rightarrow (\exists T. U = \text{Inr } T)$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

**lemma** *wt-elim-BinOp*:

$\llbracket E, dt \models \text{In1l } (\text{BinOp } \text{binop } e1 \ e2) :: T; \bigwedge T1 \ T2 \ T3.$   
 $\llbracket E, dt \models e1 :: -T1; E, dt \models e2 :: -T2; \text{wt-binop } (\text{prg } E) \ \text{binop } T1 \ T2;$   
 $E, dt \models (\text{if } b \text{ then } \text{In1l } e2 \text{ else } \text{In1r } \text{Skip}) :: T3;$   
 $T = \text{Inl } (\text{PrimT } (\text{binop-type } \text{binop})) \rrbracket$   
 $\implies P$   
 $\implies P$   
 $\langle \text{proof} \rangle$

**lemma** *Inj-eq-lemma* [simp]:

$(\forall T. (\exists T'. T = \text{Inj } T' \wedge P \ T') \longrightarrow Q \ T) = (\forall T'. P \ T' \longrightarrow Q \ (\text{Inj } T'))$   
 $\langle \text{proof} \rangle$

**lemma** *single-valued-tys-lemma* [rule-format (no-asm)]:  
 $\forall S\ T. G \vdash S \preceq T \longrightarrow G \vdash T \preceq S \longrightarrow S = T \implies E, dt \models t :: T \implies$   
 $G = \text{prg } E \longrightarrow (\forall T'. E, dt \models t :: T' \longrightarrow T = T')$   
 <proof>

**lemma** *single-valued-tys*:  
 $\text{ws-prog } (\text{prg } E) \implies \text{single-valued } \{(t, T). E, dt \models t :: T\}$   
 <proof>

**lemma** *typeof-empty-is-type* [rule-format (no-asm)]:  
 $\text{typeof } (\lambda a. \text{None})\ v = \text{Some } T \longrightarrow \text{is-type } G\ T$   
 <proof>

**lemma** *typeof-is-type* [rule-format (no-asm)]:  
 $(\forall a. v \neq \text{Addr } a) \longrightarrow (\exists T. \text{typeof } dt\ v = \text{Some } T \wedge \text{is-type } G\ T)$   
 <proof>

**end**

## Chapter 12

# DefiniteAssignment

## 29 Definite Assignment

**theory** *DefiniteAssignment* **imports** *WellType* **begin**

Definite Assignment Analysis (cf. 16)

The definite assignment analysis approximates the sets of local variables that will be assigned at a certain point of evaluation, and ensures that we will only read variables which previously were assigned. It should conform to the following idea: If the evaluation of a term completes normally (no abrupton (exception, break, continue, return) appeared) , the set of local variables calculated by the analysis is a subset of the variables that were actually assigned during evaluation.

To get more precise information about the sets of assigned variables the analysis includes the following optimisations:

- Inside of a while loop we also take care of the variables assigned before break statements, since the break causes the while loop to continue normally.
- For conditional statements we take care of constant conditions to statically determine the path of evaluation.
- Inside a distinct path of a conditional statements we know to which boolean value the condition has evaluated to, and so can retrieve more information about the variables assigned during evaluation of the boolean condition.

Since in our model of Java the return values of methods are stored in a local variable we also ensure that every path of (normal) evaluation will assign the result variable, or in the sense of real Java every path ends up in and return instruction.

Not covered yet:

- analysis of definite unassigned
- special treatment of final fields

### Correct nesting of jump statements

For definite assignment it becomes crucial, that jumps (break, continue, return) are nested correctly i.e. a continue jump is nested in a matching while statement, a break jump is nested in a proper label statement, a class initialiser does not terminate abruptly with a return. With this we can for example ensure that evaluation of an expression will never end up with a jump, since no breaks, continues or returns are allowed in an expression.

**consts** *jumpNestingOkS* :: *jump set*  $\Rightarrow$  *stmt*  $\Rightarrow$  *bool*

**primrec**

*jumpNestingOkS jmps (Skip)* = *True*

*jumpNestingOkS jmps (Expr e)* = *True*

*jumpNestingOkS jmps (j• s)* = *jumpNestingOkS* ( $\{j\} \cup$  *jmps*) *s*

*jumpNestingOkS jmps (c1;;c2)* = (*jumpNestingOkS jmps c1*  $\wedge$   
*jumpNestingOkS jmps c2*)

*jumpNestingOkS jmps (If(e) c1 Else c2)* = (*jumpNestingOkS jmps c1*  $\wedge$   
*jumpNestingOkS jmps c2*)

*jumpNestingOkS jmps (l• While(e) c)* = *jumpNestingOkS* ( $\{Cont\ l\} \cup$  *jmps*) *c*

— The label of the while loop only handles continue jumps. Breaks are only handled by *Lab*

*jumpNestingOkS jmps (Jmp j)* = ( $j \in$  *jmps*)

*jumpNestingOkS jmps (Throw e)* = *True*

*jumpNestingOkS jmps (Try c1 Catch(*C vn*) c2)* = (*jumpNestingOkS jmps c1*  $\wedge$   
*jumpNestingOkS jmps c2*)

*jumpNestingOkS jmps (c1 Finally c2)* = (*jumpNestingOkS jmps c1*  $\wedge$

*jumpNestingOkS jmps c2)*

*jumpNestingOkS jmps (Init C) = True*  
 — wellformedness of the program must enshure that for all initializers *jumpNestingOkS* holds  
 — Dummy analysis for intermediate smallstep term *FinA*  
*jumpNestingOkS jmps (FinA a c) = False*

**constdefs** *jumpNestingOk* :: *jump set*  $\Rightarrow$  *term*  $\Rightarrow$  *bool*  
*jumpNestingOk jmps t*  $\equiv$  (case *t* of  
     *In1 se*  $\Rightarrow$  (case *se* of  
         *Inl e*  $\Rightarrow$  *True*  
         | *Inr s*  $\Rightarrow$  *jumpNestingOkS jmps s*)  
     | *In2 v*  $\Rightarrow$  *True*  
     | *In3 es*  $\Rightarrow$  *True*)

**lemma** *jumpNestingOk-expr-simp* [*simp*]: *jumpNestingOk jmps (In1l e) = True*  
 <proof>

**lemma** *jumpNestingOk-expr-simp1* [*simp*]: *jumpNestingOk jmps <e::expr> = True*  
 <proof>

**lemma** *jumpNestingOk-stmt-simp* [*simp*]:  
*jumpNestingOk jmps (In1r s) = jumpNestingOkS jmps s*  
 <proof>

**lemma** *jumpNestingOk-stmt-simp1* [*simp*]:  
*jumpNestingOk jmps <s::stmt> = jumpNestingOkS jmps s*  
 <proof>

**lemma** *jumpNestingOk-var-simp* [*simp*]: *jumpNestingOk jmps (In2 v) = True*  
 <proof>

**lemma** *jumpNestingOk-var-simp1* [*simp*]: *jumpNestingOk jmps <v::var> = True*  
 <proof>

**lemma** *jumpNestingOk-expr-list-simp* [*simp*]: *jumpNestingOk jmps (In3 es) = True*  
 <proof>

**lemma** *jumpNestingOk-expr-list-simp1* [*simp*]:  
*jumpNestingOk jmps <es::expr list> = True*  
 <proof>

## Calculation of assigned variables for boolean expressions

### 30 Very restricted calculation fallback calculation

**consts** *the-LVar-name*:: *var*  $\Rightarrow$  *lname*  
**primrec**  
*the-LVar-name* (*LVar n*) = *n*

**consts** *assignsE* :: *expr*  $\Rightarrow$  *lname set*

$assignsV :: var \Rightarrow lname\ set$   
 $assignsEs :: expr\ list \Rightarrow lname\ set$

**primrec**

$assignsE\ (NewC\ c) = \{\}$   
 $assignsE\ (NewA\ t\ e) = assignsE\ e$   
 $assignsE\ (Cast\ t\ e) = assignsE\ e$   
 $assignsE\ (e\ InstOf\ r) = assignsE\ e$   
 $assignsE\ (Lit\ val) = \{\}$   
 $assignsE\ (UnOp\ unop\ e) = assignsE\ e$   
 $assignsE\ (BinOp\ binop\ e1\ e2) = (if\ binop=CondAnd\ \vee\ binop=CondOr$   
 $\quad then\ (assignsE\ e1)$   
 $\quad else\ (assignsE\ e1) \cup (assignsE\ e2))$   
 $assignsE\ (Super) = \{\}$   
 $assignsE\ (Acc\ v) = assignsV\ v$   
 $assignsE\ (v:=e)$   
 $\quad = (assignsV\ v) \cup (assignsE\ e) \cup$   
 $\quad (if\ \exists\ n.\ v=(LVar\ n)\ then\ \{the-LVar-name\ v\}$   
 $\quad \quad else\ \{\})$

$assignsE\ (b? e1 : e2) = (assignsE\ b) \cup ((assignsE\ e1) \cap (assignsE\ e2))$   
 $assignsE\ (\{accC, statT, mode\}objRef.mn(\{pTs\}args))$   
 $\quad = (assignsE\ objRef) \cup (assignsEs\ args)$

— Only dummy analysis for intermediate expressions *Method*, *Body*, *InsInitE* and *Callee*

$assignsE\ (Method\ C\ sig) = \{\}$   
 $assignsE\ (Body\ C\ s) = \{\}$   
 $assignsE\ (InsInitE\ s\ e) = \{\}$   
 $assignsE\ (Callee\ l\ e) = \{\}$

$assignsV\ (LVar\ n) = \{\}$   
 $assignsV\ (\{accC, statDeclC, stat\}objRef..fn) = assignsE\ objRef$   
 $assignsV\ (e1.[e2]) = assignsE\ e1 \cup assignsE\ e2$

$assignsEs\ [] = \{\}$   
 $assignsEs\ (e\#es) = assignsE\ e \cup assignsEs\ es$

**constdefs**  $assigns :: term \Rightarrow lname\ set$

$assigns\ t \equiv (case\ t\ of$   
 $\quad In1\ se \Rightarrow (case\ se\ of$   
 $\quad \quad Inl\ e \Rightarrow assignsE\ e$   
 $\quad \quad | Inr\ s \Rightarrow \{\})$   
 $\quad | In2\ v \Rightarrow assignsV\ v$   
 $\quad | In3\ es \Rightarrow assignsEs\ es)$

**lemma**  $assigns-expr-simp\ [simp]:\ assigns\ (In1l\ e) = assignsE\ e$   
 $\langle proof \rangle$

**lemma**  $assigns-expr-simp1\ [simp]:\ assigns\ (\langle e \rangle) = assignsE\ e$   
 $\langle proof \rangle$

**lemma**  $assigns-stmt-simp\ [simp]:\ assigns\ (In1r\ s) = \{\}$   
 $\langle proof \rangle$

**lemma**  $assigns-stmt-simp1\ [simp]:\ assigns\ (\langle s::stmt \rangle) = \{\}$   
 $\langle proof \rangle$



**lemma** *assigns-var-simp* [simp]: *assigns (In2 v) = assignsV v*  
 ⟨proof⟩

**lemma** *assigns-var-simp1* [simp]: *assigns (⟨v⟩) = assignsV v*  
 ⟨proof⟩

**lemma** *assigns-expr-list-simp* [simp]: *assigns (In3 es) = assignsEs es*  
 ⟨proof⟩

**lemma** *assigns-expr-list-simp1* [simp]: *assigns (⟨es⟩) = assignsEs es*  
 ⟨proof⟩

### 31 Analysis of constant expressions

**consts** *constVal* :: *expr*  $\Rightarrow$  *val option*

**primrec**

*constVal* (*NewC* *c*) = *None*

*constVal* (*NewA* *t e*) = *None*

*constVal* (*Cast* *t e*) = *None*

*constVal* (*Inst* *e r*) = *None*

*constVal* (*Lit* *val*) = *Some val*

*constVal* (*UnOp* *unop e*) = (case (*constVal* *e*) of  
     *None*  $\Rightarrow$  *None*  
     | *Some v*  $\Rightarrow$  *Some (eval-unop unop v)*)

*constVal* (*BinOp* *binop e1 e2*) = (case (*constVal* *e1*) of  
     *None*  $\Rightarrow$  *None*  
     | *Some v1*  $\Rightarrow$  (case (*constVal* *e2*) of  
         *None*  $\Rightarrow$  *None*  
         | *Some v2*  $\Rightarrow$  *Some (eval-binop binop v1 v2)*)))

*constVal* (*Super*) = *None*

*constVal* (*Acc* *v*) = *None*

*constVal* (*Ass* *v e*) = *None*

*constVal* (*Cond* *b e1 e2*) = (case (*constVal* *b*) of  
     *None*  $\Rightarrow$  *None*  
     | *Some bv*  $\Rightarrow$  (case *the-Bool* *bv* of  
         *True*  $\Rightarrow$  (case (*constVal* *e2*) of  
             *None*  $\Rightarrow$  *None*  
             | *Some v*  $\Rightarrow$  *constVal* *e1*)  
         | *False*  $\Rightarrow$  (case (*constVal* *e1*) of  
             *None*  $\Rightarrow$  *None*  
             | *Some v*  $\Rightarrow$  *constVal* *e2*))))

— Note that *constVal* (*Cond* *b e1 e2*) is stricter as it could be. It requires that all tree expressions are constant even if we can decide which branch to choose, provided the constant value of *b*

*constVal* (*Call* *accC statT mode objRef mn pTs args*) = *None*

*constVal* (*Method* *C sig*) = *None*

*constVal* (*Body* *C s*) = *None*

*constVal* (*InsInitE* *s e*) = *None*

*constVal* (*Callee* *l e*) = *None*

**lemma** *constVal-Some-induct* [consumes 1, case-names *Lit UnOp BinOp CondL CondR*]:

**assumes** *const*: *constVal* *e* = *Some v* **and**

*hyp-Lit*:  $\bigwedge v. P (Lit v)$  **and**

*hyp-UnOp*:  $\bigwedge unop e'. P e' \Longrightarrow P (UnOp unop e')$  **and**

*hyp-BinOp*:  $\bigwedge binop e1 e2. \llbracket P e1; P e2 \rrbracket \Longrightarrow P (BinOp binop e1 e2)$  **and**

$$\text{hyp-CondL: } \bigwedge b \text{ bv } e1 \ e2. \llbracket \text{constVal } b = \text{Some } bv; \text{the-Bool } bv; P \ b; P \ e1 \rrbracket$$

$$\implies P \ (b? \ e1 : e2) \text{ and}$$

$$\text{hyp-CondR: } \bigwedge b \text{ bv } e1 \ e2. \llbracket \text{constVal } b = \text{Some } bv; \neg \text{the-Bool } bv; P \ b; P \ e2 \rrbracket$$

$$\implies P \ (b? \ e1 : e2)$$

**shows**  $P \ e$   
 ⟨proof⟩

**lemma** *assignsE-const-simp*:  $\text{constVal } e = \text{Some } v \implies \text{assignsE } e = \{\}$   
 ⟨proof⟩

## 32 Main analysis for boolean expressions

Assigned local variables after evaluating the expression if it evaluates to a specific boolean value. If the expression cannot evaluate to a *Boolean* value UNIV is returned. If we expect true/false the opposite constant false/true will also lead to UNIV.

**consts** *assigns-if*::  $\text{bool} \Rightarrow \text{expr} \Rightarrow \text{lname set}$

**primrec**

$$\begin{aligned} \text{assigns-if } b \ (\text{NewC } c) &= \text{UNIV} \text{ — can never evaluate to Boolean} \\ \text{assigns-if } b \ (\text{NewA } t \ e) &= \text{UNIV} \text{ — can never evaluate to Boolean} \\ \text{assigns-if } b \ (\text{Cast } t \ e) &= \text{assigns-if } b \ e \\ \text{assigns-if } b \ (\text{Inst } e \ r) &= \text{assignsE } e \text{ — Inst has type Boolean but } e \text{ is a reference type} \\ \text{assigns-if } b \ (\text{Lit } val) &= (\text{if } val = \text{Bool } b \text{ then } \{\} \text{ else UNIV}) \\ \text{assigns-if } b \ (\text{UnOp } unop \ e) &= (\text{case constVal } (\text{UnOp } unop \ e) \text{ of} \\ &\quad \text{None} \Rightarrow (\text{if } unop = \text{UNot} \\ &\quad \text{then assigns-if } (\neg b) \ e \\ &\quad \text{else UNIV}) \\ &\quad | \text{Some } v \Rightarrow (\text{if } v = \text{Bool } b \\ &\quad \text{then } \{\} \\ &\quad \text{else UNIV})) \end{aligned}$$

$$\begin{aligned} \text{assigns-if } b \ (\text{BinOp } binop \ e1 \ e2) &= (\text{case constVal } (\text{BinOp } binop \ e1 \ e2) \text{ of} \\ &\quad \text{None} \Rightarrow (\text{if } binop = \text{CondAnd} \text{ then} \\ &\quad \quad (\text{case } b \text{ of} \\ &\quad \quad \quad \text{True} \Rightarrow \text{assigns-if True } e1 \cup \text{assigns-if True } e2 \\ &\quad \quad \quad | \text{False} \Rightarrow \text{assigns-if False } e1 \cap \\ &\quad \quad \quad \quad (\text{assigns-if True } e1 \cup \text{assigns-if False } e2)) \\ &\quad \text{else} \\ &\quad (\text{if } binop = \text{CondOr} \text{ then} \\ &\quad \quad (\text{case } b \text{ of} \\ &\quad \quad \quad \text{True} \Rightarrow \text{assigns-if True } e1 \cap \\ &\quad \quad \quad \quad (\text{assigns-if False } e1 \cup \text{assigns-if True } e2) \\ &\quad \quad \quad | \text{False} \Rightarrow \text{assigns-if False } e1 \cup \text{assigns-if False } e2) \\ &\quad \quad \text{else assignsE } e1 \cup \text{assignsE } e2)) \\ &\quad | \text{Some } v \Rightarrow (\text{if } v = \text{Bool } b \text{ then } \{\} \text{ else UNIV})) \end{aligned}$$

$$\begin{aligned} \text{assigns-if } b \ (\text{Super}) &= \text{UNIV} \text{ — can never evaluate to Boolean} \\ \text{assigns-if } b \ (\text{Acc } v) &= (\text{assignsV } v) \\ \text{assigns-if } b \ (v := e) &= (\text{assignsE } (\text{Ass } v \ e)) \\ \text{assigns-if } b \ (c? \ e1 : e2) &= (\text{assignsE } c) \cup \\ &\quad (\text{case } (\text{constVal } c) \text{ of} \\ &\quad \quad \text{None} \Rightarrow (\text{assigns-if } b \ e1) \cap \\ &\quad \quad \quad (\text{assigns-if } b \ e2) \\ &\quad \quad | \text{Some } bv \Rightarrow (\text{case the-Bool } bv \text{ of} \\ &\quad \quad \quad \text{True} \Rightarrow \text{assigns-if } b \ e1 \\ &\quad \quad \quad | \text{False} \Rightarrow \text{assigns-if } b \ e2)) \end{aligned}$$

$$\begin{aligned} \text{assigns-if } b \ (\{\text{accC}, \text{statT}, \text{mode}\} \text{objRef} \cdot \text{mn}(\{pTs\} \text{args})) &= \text{assignsE } (\{\text{accC}, \text{statT}, \text{mode}\} \text{objRef} \cdot \text{mn}(\{pTs\} \text{args})) \end{aligned}$$

— Only dummy analysis for intermediate expressions *Method*, *Body*, *InsInitE* and *Callee*

*assigns-if* *b* (*Method* *C* *sig*) = {}  
*assigns-if* *b* (*Body* *C* *s*) = {}  
*assigns-if* *b* (*InsInitE* *s* *e*) = {}  
*assigns-if* *b* (*Callee* *l* *e*) = {}

**lemma** *assigns-if-const-b-simp*:

**assumes** *boolConst*: *constVal* *e* = *Some* (*Bool* *b*) (**is** ?*Const* *b* *e*)  
**shows** *assigns-if* *b* *e* = {} (**is** ?*Ass* *b* *e*)  
 <proof>

**lemma** *assigns-if-const-not-b-simp*:

**assumes** *boolConst*: *constVal* *e* = *Some* (*Bool* *b*) (**is** ?*Const* *b* *e*)  
**shows** *assigns-if* ( $\neg$ *b*) *e* = *UNIV* (**is** ?*Ass* *b* *e*)  
 <proof>

### 33 Lifting set operations to range of tables (map to a set)

**constdefs**

*union-ts*:: ('*a*, '*b*) *tables*  $\Rightarrow$  ('*a*, '*b*) *tables*  $\Rightarrow$  ('*a*, '*b*) *tables*  
 ( $- \Rightarrow \cup$  - [67,67] 65)  
 $A \Rightarrow \cup B \equiv \lambda k. A\ k \cup B\ k$

**constdefs**

*intersect-ts*:: ('*a*, '*b*) *tables*  $\Rightarrow$  ('*a*, '*b*) *tables*  $\Rightarrow$  ('*a*, '*b*) *tables*  
 ( $- \Rightarrow \cap$  - [72,72] 71)  
 $A \Rightarrow \cap B \equiv \lambda k. A\ k \cap B\ k$

**constdefs**

*all-union-ts*:: ('*a*, '*b*) *tables*  $\Rightarrow$  '*b* *set*  $\Rightarrow$  ('*a*, '*b*) *tables*  
 (**infixl**  $\Rightarrow \cup_{\forall}$  40)  
 $A \Rightarrow \cup_{\forall} B \equiv \lambda k. A\ k \cup B$

### Binary union of tables

**lemma** *union-ts-iff* [*simp*]: ( $c \in (A \Rightarrow \cup B)\ k$ ) = ( $c \in A\ k \vee c \in B\ k$ )  
 <proof>

**lemma** *union-tsI1* [*elim?*]:  $c \in A\ k \Longrightarrow c \in (A \Rightarrow \cup B)\ k$   
 <proof>

**lemma** *union-tsI2* [*elim?*]:  $c \in B\ k \Longrightarrow c \in (A \Rightarrow \cup B)\ k$   
 <proof>

**lemma** *union-tsCI* [*intro!*]: ( $c \notin B\ k \Longrightarrow c \in A\ k$ )  $\Longrightarrow c \in (A \Rightarrow \cup B)\ k$   
 <proof>

**lemma** *union-tsE* [*elim!*]:

$\llbracket c \in (A \Rightarrow \cup B)\ k; (c \in A\ k \Longrightarrow P); (c \in B\ k \Longrightarrow P) \rrbracket \Longrightarrow P$   
 <proof>

### Binary intersection of tables

**lemma** *intersect-ts-iff* [simp]:  $c \in (A \Rightarrow \cap B) \ k = (c \in A \ k \wedge c \in B \ k)$   
 ⟨proof⟩

**lemma** *intersect-tsI* [intro!]:  $\llbracket c \in A \ k; c \in B \ k \rrbracket \Longrightarrow c \in (A \Rightarrow \cap B) \ k$   
 ⟨proof⟩

**lemma** *intersect-tsD1*:  $c \in (A \Rightarrow \cap B) \ k \Longrightarrow c \in A \ k$   
 ⟨proof⟩

**lemma** *intersect-tsD2*:  $c \in (A \Rightarrow \cap B) \ k \Longrightarrow c \in B \ k$   
 ⟨proof⟩

**lemma** *intersect-tsE* [elim!]:  
 $\llbracket c \in (A \Rightarrow \cap B) \ k; \llbracket c \in A \ k; c \in B \ k \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$   
 ⟨proof⟩

### All-Union of tables and set

**lemma** *all-union-ts-iff* [simp]:  $(c \in (A \Rightarrow \cup B) \ k) = (c \in A \ k \vee c \in B)$   
 ⟨proof⟩

**lemma** *all-union-tsI1* [elim?]:  $c \in A \ k \Longrightarrow c \in (A \Rightarrow \cup B) \ k$   
 ⟨proof⟩

**lemma** *all-union-tsI2* [elim?]:  $c \in B \Longrightarrow c \in (A \Rightarrow \cup B) \ k$   
 ⟨proof⟩

**lemma** *all-union-tsCI* [intro!]:  $(c \notin B \Longrightarrow c \in A \ k) \Longrightarrow c \in (A \Rightarrow \cup B) \ k$   
 ⟨proof⟩

**lemma** *all-union-tsE* [elim!]:  
 $\llbracket c \in (A \Rightarrow \cup B) \ k; (c \in A \ k \Longrightarrow P); (c \in B \Longrightarrow P) \rrbracket \Longrightarrow P$   
 ⟨proof⟩

### The rules of definite assignment

**types** *breakass* = (*label*, *lname*) *tables*

— Mapping from a break label, to the set of variables that will be assigned if the evaluation terminates with this break

**record** *assigned* =

*nrm* :: *lname set* — Definetly assigned variables for normal completion

*brk* :: *breakass* — Definetly assigned variables for abrupt completion with a break

**consts** *da* :: (*env* × *lname set* × *term* × *assigned*) *set*

The environment *env* is only needed for the conditional - ? - : -. The definite assignment rules refer to the typing rules here to distinguish boolean and other expressions.

**syntax**

$da :: env \Rightarrow lname\ set \Rightarrow term \Rightarrow assigned \Rightarrow bool$   
 $(+ - \gg - [65,65,65,65] 71)$

### translations

$E \vdash B \gg t \gg A == (E, B, t, A) \in da$

$B$ : the "assigned" variables before evaluating term  $t$ ;  $A$ : the "assigned" variables after evaluating term  $t$

**constdefs**  $rmlab :: 'a \Rightarrow ('a, 'b)\ tables \Rightarrow ('a, 'b)\ tables$   
 $rmlab\ k\ A \equiv \lambda\ x. \text{ if } x=k \text{ then } UNIV \text{ else } A\ x$

**constdefs**  $range\text{-}inter\text{-}ts :: ('a, 'b)\ tables \Rightarrow 'b\ set (\Rightarrow \bigcap - 80)$   
 $\Rightarrow \bigcap A \equiv \{x \mid x. \forall\ k. x \in A\ k\}$

### inductive $da$ intros

*Skip*:  $Env \vdash B \gg \langle Skip \rangle \gg (\text{norm}=B, \text{brk}=\lambda\ l. UNIV)$

*Expr*:  $Env \vdash B \gg \langle e \rangle \gg A$

$\Rightarrow$

$Env \vdash B \gg \langle Expr\ e \rangle \gg A$

*Lab*:  $\llbracket Env \vdash B \gg \langle c \rangle \gg C; \text{norm}\ A = \text{norm}\ C \cap (\text{brk}\ C)\ l; \text{brk}\ A = rmlab\ l\ (\text{brk}\ C) \rrbracket$

$\Rightarrow$

$Env \vdash B \gg \langle Break\ l\ c \rangle \gg A$

*Comp*:  $\llbracket Env \vdash B \gg \langle c1 \rangle \gg C1; Env \vdash \text{norm}\ C1 \gg \langle c2 \rangle \gg C2; \text{norm}\ A = \text{norm}\ C2; \text{brk}\ A = (\text{brk}\ C1) \Rightarrow \cap (\text{brk}\ C2) \rrbracket$

$\Rightarrow$

$Env \vdash B \gg \langle c1;; c2 \rangle \gg A$

*If*:  $\llbracket Env \vdash B \gg \langle e \rangle \gg E;$

$Env \vdash (B \cup \text{assigns-if}\ True\ e) \gg \langle c1 \rangle \gg C1;$

$Env \vdash (B \cup \text{assigns-if}\ False\ e) \gg \langle c2 \rangle \gg C2;$

$\text{norm}\ A = \text{norm}\ C1 \cap \text{norm}\ C2;$

$\text{brk}\ A = \text{brk}\ C1 \Rightarrow \cap \text{brk}\ C2 \rrbracket$

$\Rightarrow$

$Env \vdash B \gg \langle If(e)\ c1\ Else\ c2 \rangle \gg A$

— Note that  $E$  is not further used, because we take the specialized sets that also consider if the expression evaluates to true or false. Inside of  $e$  there is no **break** or **finally**, so the break map of  $E$  will be the trivial one. So  $Env \vdash B \gg \langle e \rangle \gg E$  is just used to ensure the definite assignment in expression  $e$ . Notice the implicit analysis of a constant boolean expression  $e$  in this rule. For example, if  $e$  is constantly *True* then *assigns-if False e* = *UNIV* and therefore  $\text{norm}\ C2 = UNIV$ . So finally  $\text{norm}\ A = \text{norm}\ C1$ . For the break maps this trick works too, because the trivial break map will map all labels to *UNIV*. In the example, if no break occurs in  $c2$  the break maps will trivially map to *UNIV* and if a break occurs it will map to *UNIV* too, because *assigns-if False e* = *UNIV*. So in the intersection of the break maps the path  $c2$  will have no contribution.

*Loop*:  $\llbracket Env \vdash B \gg \langle e \rangle \gg E;$

$Env \vdash (B \cup \text{assigns-if}\ True\ e) \gg \langle c \rangle \gg C;$

$\text{norm}\ A = \text{norm}\ C \cap (B \cup \text{assigns-if}\ False\ e);$

$\text{brk}\ A = \text{brk}\ C \rrbracket$

$\Rightarrow$

$Env \vdash B \gg \langle l.\ While(e)\ c \rangle \gg A$

— The *Loop* rule resembles some of the ideas of the *If* rule. For the  $\text{norm}\ A$  the set  $B \cup \text{assigns-if False e}$  will be *UNIV* if the condition is constantly true. To normally exit the while loop, we must consider the body  $c$  to be completed normally ( $\text{norm}\ C$ ) or with a break. But in this model, the label  $l$  of the loop only handles continue labels, not break labels. The break label will be handled by an enclosing *Lab* statement. So we don't

have to handle the breaks specially.

$$\begin{aligned}
 & \text{Jump: } \llbracket \text{jump} = \text{Ret} \longrightarrow \text{Result} \in B; \\
 & \quad \text{nrm } A = \text{UNIV}; \\
 & \quad \text{brk } A = (\text{case jump of} \\
 & \quad \quad \text{Break } l \Rightarrow \lambda k. \text{ if } k=l \text{ then } B \text{ else UNIV} \\
 & \quad \quad | \text{Cont } l \Rightarrow \lambda k. \text{ UNIV} \\
 & \quad \quad | \text{Ret} \Rightarrow \lambda k. \text{ UNIV}) \rrbracket \\
 & \implies \\
 & \quad \text{Env} \vdash B \gg \langle \text{Jump jump} \rangle \gg A
 \end{aligned}$$

— In case of a break to label  $l$  the corresponding break set is all variables assigned before the break. The assigned variables for normal completion of the *Jump* is *UNIV*, because the statement will never complete normally. For continue and return the break map is the trivial one. In case of a return we ensure that the result value is assigned.

$$\begin{aligned}
 & \text{Throw: } \llbracket \text{Env} \vdash B \gg \langle e \rangle \gg E; \text{nrm } A = \text{UNIV}; \text{brk } A = (\lambda l. \text{ UNIV}) \rrbracket \\
 & \implies \text{Env} \vdash B \gg \langle \text{Throw } e \rangle \gg A
 \end{aligned}$$

$$\begin{aligned}
 & \text{Try: } \llbracket \text{Env} \vdash B \gg \langle c1 \rangle \gg C1; \\
 & \quad \text{Env}(\text{lcl} := \text{lcl Env}(\text{VName } vn \mapsto \text{Class } C)) \vdash (B \cup \{ \text{VName } vn \}) \gg \langle c2 \rangle \gg C2; \\
 & \quad \text{nrm } A = \text{nrm } C1 \cap \text{nrm } C2; \\
 & \quad \text{brk } A = \text{brk } C1 \Rightarrow \cap \text{brk } C2 \rrbracket \\
 & \implies \text{Env} \vdash B \gg \langle \text{Try } c1 \text{ Catch}(C \text{ } vn) \text{ } c2 \rangle \gg A
 \end{aligned}$$

$$\begin{aligned}
 & \text{Fin: } \llbracket \text{Env} \vdash B \gg \langle c1 \rangle \gg C1; \\
 & \quad \text{Env} \vdash B \gg \langle c2 \rangle \gg C2; \\
 & \quad \text{nrm } A = \text{nrm } C1 \cup \text{nrm } C2; \\
 & \quad \text{brk } A = ((\text{brk } C1) \Rightarrow \cup (\text{nrm } C2)) \Rightarrow \cap (\text{brk } C2) \rrbracket \\
 & \implies \\
 & \quad \text{Env} \vdash B \gg \langle c1 \text{ Finally } c2 \rangle \gg A
 \end{aligned}$$

— The set of assigned variables before execution  $c2$  are the same as before execution  $c1$ , because  $c1$  could throw an exception and so we can't guarantee that any variable will be assigned in  $c1$ . The *Finally* statement completes normally if both  $c1$  and  $c2$  complete normally. If  $c1$  completes abruptly with a break, then  $c2$  also will be executed and may terminate normally or with a break. The overall break map then is the intersection of the maps of both paths. If  $c2$  terminates normally we have to extend all break sets in  $\text{brk } C1$  with  $\text{nrm } C2$  ( $\Rightarrow \cup$ ). If  $c2$  exits with a break this break will appear in the overall result state. We don't know if  $c1$  completed normally or abruptly (maybe with an exception not only a break) so  $c1$  has no contribution to the break map following this path.

— Evaluation of expressions and the break sets of definite assignment: Thinking of a Java expression we assume that we can never have a break statement inside of an expression. So for all expressions the break sets could be set to the trivial one:  $\lambda l. \text{ UNIV}$ . But we can't trivially prove, that evaluating an expression will never result in a break, although Java expressions already syntactically don't allow nested statements in them. The reason are the nested class initialization statements which are inserted by the evaluation rules. So to prove the absence of a break we need to ensure, that the initialization statements will never end up in a break. In a wellformed initialization statement, of course, where breaks are nested correctly inside of *Lab* or *Loop* statements evaluation of the whole initialization statement will never result in a break, because this break will be handled inside of the statement. But for simplicity we haven't added the analysis of the correct nesting of breaks in the typing judgments right now. So we have decided to adjust the rules of definite assignment to fit to these circumstances. If an initialization is involved during evaluation of the expression (evaluation rules *FVar*, *NewC* and *NewA*)

$$\text{Init: } \text{Env} \vdash B \gg \langle \text{Init } C \rangle \gg (\text{nrm} = B, \text{brk} = \lambda l. \text{ UNIV})$$

— Wellformedness of a program will ensure, that every static initialiser is definitely assigned and the jumps are nested correctly. The case here for *Init* is just for convenience, to get a proper precondition for the induction hypothesis in various proofs, so that we don't have to expand the initialisation on every point where it is triggered by the evaluation rules.

$$\text{NewC: } \text{Env} \vdash B \gg \langle \text{NewC } C \rangle \gg (\text{nrm} = B, \text{brk} = \lambda l. \text{ UNIV})$$

*NewA*:  $Env \vdash B \gg \langle e \rangle \gg A$

$\implies$

$Env \vdash B \gg \langle New\ T[e] \rangle \gg A$

*Cast*:  $Env \vdash B \gg \langle e \rangle \gg A$

$\implies$

$Env \vdash B \gg \langle Cast\ T\ e \rangle \gg A$

*Inst*:  $Env \vdash B \gg \langle e \rangle \gg A$

$\implies$

$Env \vdash B \gg \langle e\ InstOf\ T \rangle \gg A$

*Lit*:  $Env \vdash B \gg \langle Lit\ v \rangle \gg (\llbracket nrm=B, brk=\lambda\ l.\ UNIV \rrbracket)$

*UnOp*:  $Env \vdash B \gg \langle e \rangle \gg A$

$\implies$

$Env \vdash B \gg \langle UnOp\ unop\ e \rangle \gg A$

*CondAnd*:  $\llbracket Env \vdash B \gg \langle e1 \rangle \gg E1; Env \vdash (B \cup assigns\text{-}if\ True\ e1) \gg \langle e2 \rangle \gg E2;$

$nrm\ A = B \cup (assigns\text{-}if\ True\ (BinOp\ CondAnd\ e1\ e2) \cap$

$assigns\text{-}if\ False\ (BinOp\ CondAnd\ e1\ e2));$

$brk\ A = (\lambda\ l.\ UNIV) \rrbracket$

$\implies$

$Env \vdash B \gg \langle BinOp\ CondAnd\ e1\ e2 \rangle \gg A$

*CondOr*:  $\llbracket Env \vdash B \gg \langle e1 \rangle \gg E1; Env \vdash (B \cup assigns\text{-}if\ False\ e1) \gg \langle e2 \rangle \gg E2;$

$nrm\ A = B \cup (assigns\text{-}if\ True\ (BinOp\ CondOr\ e1\ e2) \cap$

$assigns\text{-}if\ False\ (BinOp\ CondOr\ e1\ e2));$

$brk\ A = (\lambda\ l.\ UNIV) \rrbracket$

$\implies$

$Env \vdash B \gg \langle BinOp\ CondOr\ e1\ e2 \rangle \gg A$

*BinOp*:  $\llbracket Env \vdash B \gg \langle e1 \rangle \gg E1; Env \vdash nrm\ E1 \gg \langle e2 \rangle \gg A;$

$binop \neq CondAnd; binop \neq CondOr \rrbracket$

$\implies$

$Env \vdash B \gg \langle BinOp\ binop\ e1\ e2 \rangle \gg A$

*Super*:  $This \in B$

$\implies$

$Env \vdash B \gg \langle Super \rangle \gg (\llbracket nrm=B, brk=\lambda\ l.\ UNIV \rrbracket)$

*AccLVar*:  $\llbracket vn \in B;$

$nrm\ A = B; brk\ A = (\lambda\ k.\ UNIV) \rrbracket$

$\implies$

$Env \vdash B \gg \langle Acc\ (LVar\ vn) \rangle \gg A$

— To properly access a local variable we have to test the definite assignment here. The variable must occur in the set  $B$

*Acc*:  $\llbracket \forall\ vn.\ v \neq LVar\ vn;$

$Env \vdash B \gg \langle v \rangle \gg A \rrbracket$

$\implies$

$Env \vdash B \gg \langle Acc\ v \rangle \gg A$

*AssLVar*:  $\llbracket Env \vdash B \gg \langle e \rangle \gg E; nrm\ A = nrm\ E \cup \{vn\}; brk\ A = brk\ E \rrbracket$

$\implies$

$Env \vdash B \gg \langle (LVar\ vn) := e \rangle \gg A$

*Ass*:  $\llbracket \forall\ vn.\ v \neq LVar\ vn; Env \vdash B \gg \langle v \rangle \gg V; Env \vdash nrm\ V \gg \langle e \rangle \gg A \rrbracket$

$\implies$

$$Env \vdash B \gg \langle v := e \rangle \gg A$$

$$\begin{aligned} \text{CondBool: } & \llbracket Env \vdash (c \ ? \ e1 : e2) :: \neg (PrimT \ Boolean); \\ & Env \vdash B \gg \langle c \rangle \gg C; \\ & Env \vdash (B \cup \text{assigns-if True } c) \gg \langle e1 \rangle \gg E1; \\ & Env \vdash (B \cup \text{assigns-if False } c) \gg \langle e2 \rangle \gg E2; \\ & nrm \ A = B \cup (\text{assigns-if True } (c \ ? \ e1 : e2) \cap \\ & \quad \text{assigns-if False } (c \ ? \ e1 : e2)); \\ & brk \ A = (\lambda l. UNIV) \rrbracket \\ \implies & \\ & Env \vdash B \gg \langle c \ ? \ e1 : e2 \rangle \gg A \end{aligned}$$

$$\begin{aligned} \text{Cond: } & \llbracket \neg Env \vdash (c \ ? \ e1 : e2) :: \neg (PrimT \ Boolean); \\ & Env \vdash B \gg \langle c \rangle \gg C; \\ & Env \vdash (B \cup \text{assigns-if True } c) \gg \langle e1 \rangle \gg E1; \\ & Env \vdash (B \cup \text{assigns-if False } c) \gg \langle e2 \rangle \gg E2; \\ & nrm \ A = nrm \ E1 \cap nrm \ E2; \ brk \ A = (\lambda l. UNIV) \rrbracket \\ \implies & \\ & Env \vdash B \gg \langle c \ ? \ e1 : e2 \rangle \gg A \end{aligned}$$

$$\begin{aligned} \text{Call: } & \llbracket Env \vdash B \gg \langle e \rangle \gg E; \ Env \vdash nrm \ E \gg \langle args \rangle \gg A \rrbracket \\ \implies & \\ & Env \vdash B \gg \langle \{accC, statT, mode\} e \cdot mn(\{pTs\} args) \rangle \gg A \end{aligned}$$

— The interplay of *Call*, *Method* and *Body*: Why rules for *Method* and *Body* at all? Note that a Java source program will not include bare *Method* or *Body* terms. These terms are just introduced during evaluation. So definite assignment of *Call* does not consider *Method* or *Body* at all. So for definite assignment alone we could omit the rules for *Method* and *Body*. But since evaluation of the method invocation is split up into three rules we must ensure that we have enough information about the call even in the *Body* term to make sure that we can proof type safety. Also we must be able transport this information from *Call* to *Method* and then further to *Body* during evaluation to establish the definite assignment of *Method* during evaluation of *Call*, and of *Body* during evaluation of *Method*. This is necessary since definite assignment will be a precondition for each induction hypothesis coming out of the evaluation rules, and therefor we have to establish the definite assignment of the sub-evaluation during the type-safety proof. Note that well-typedness is also a precondition for type-safety and so we can omit some assertion that are already ensured by well-typedness.

$$\begin{aligned} \text{Method: } & \llbracket \text{methd } (prg \ Env) \ D \ sig = Some \ m; \\ & Env \vdash B \gg \langle Body \ (declclass \ m) \ (stmt \ (mbody \ (methd \ m))) \rangle \gg A \rrbracket \\ \implies & \\ & Env \vdash B \gg \langle Method \ D \ sig \rangle \gg A \end{aligned}$$

$$\begin{aligned} \text{Body: } & \llbracket Env \vdash B \gg \langle c \rangle \gg C; \ jumpNestingOkS \ \{Ret\} \ c; \ Result \in nrm \ C; \\ & nrm \ A = B; \ brk \ A = (\lambda l. UNIV) \rrbracket \\ \implies & \\ & Env \vdash B \gg \langle Body \ D \ c \rangle \gg A \end{aligned}$$

— Note that  $A$  is not correlated to  $C$ . If the body statement returns abruptly with return, evaluation of *Body* will absorb this return and complete normally. So we cannot trivially get the assigned variables of the body statement since it has not completed normally or with a break. If the body completes normally we guarantee that the result variable is set with this rule. But if the body completes abruptly with a return we can't guarantee that the result variable is set here, since definite assignment only talks about normal completion and breaks. So for a return the *Jump* rule ensures that the result variable is set and then this information must be carried over to the *Body* rule by the conformance predicate of the state.

$$\text{LVar: } Env \vdash B \gg \langle LVar \ vn \rangle \gg (\llbracket nrm=B, \ brk=\lambda l. UNIV \rrbracket)$$

$$\begin{aligned} \text{FVar: } & Env \vdash B \gg \langle e \rangle \gg A \\ \implies & \\ & Env \vdash B \gg \langle \{accC, statDeclC, stat\} e \cdot fn \rangle \gg A \end{aligned}$$

$$\text{AVar: } \llbracket Env \vdash B \gg \langle e1 \rangle \gg E1; \ Env \vdash nrm \ E1 \gg \langle e2 \rangle \gg A \rrbracket$$



$$\begin{array}{c} \Rightarrow \\ Env \vdash B \gg \langle e1.[e2] \rangle \gg A \end{array}$$

*Nil*:  $Env \vdash B \gg \langle [] :: expr \ list \rangle \gg (\text{norm} = B, \text{brk} = \lambda l. UNIV)$

*Cons*:  $\llbracket Env \vdash B \gg \langle e :: expr \rangle \gg E; Env \vdash \text{norm } E \gg \langle es \rangle \gg A \rrbracket$

$$\begin{array}{c} \Rightarrow \\ Env \vdash B \gg \langle e \# es \rangle \gg A \end{array}$$

**declare** *inj-term-sym-simps* [*simp*]  
**declare** *assigns-if.simps* [*simp del*]  
**declare** *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]  
 $\langle ML \rangle$

**inductive-cases** *da-elim-cases* [*cases set*]:

$Env \vdash B \gg \langle Skip \rangle \gg A$   
 $Env \vdash B \gg \langle In1r \ Skip \rangle \gg A$   
 $Env \vdash B \gg \langle Expr \ e \rangle \gg A$   
 $Env \vdash B \gg \langle In1r \ (Expr \ e) \rangle \gg A$   
 $Env \vdash B \gg \langle l \cdot c \rangle \gg A$   
 $Env \vdash B \gg \langle In1r \ (l \cdot c) \rangle \gg A$   
 $Env \vdash B \gg \langle c1;; c2 \rangle \gg A$   
 $Env \vdash B \gg \langle In1r \ (c1;; c2) \rangle \gg A$   
 $Env \vdash B \gg \langle If(e) \ c1 \ Else \ c2 \rangle \gg A$   
 $Env \vdash B \gg \langle In1r \ (If(e) \ c1 \ Else \ c2) \rangle \gg A$   
 $Env \vdash B \gg \langle l \cdot While(e) \ c \rangle \gg A$   
 $Env \vdash B \gg \langle In1r \ (l \cdot While(e) \ c) \rangle \gg A$   
 $Env \vdash B \gg \langle Jmp \ jump \rangle \gg A$   
 $Env \vdash B \gg \langle In1r \ (Jmp \ jump) \rangle \gg A$   
 $Env \vdash B \gg \langle Throw \ e \rangle \gg A$   
 $Env \vdash B \gg \langle In1r \ (Throw \ e) \rangle \gg A$   
 $Env \vdash B \gg \langle Try \ c1 \ Catch(C \ vn) \ c2 \rangle \gg A$   
 $Env \vdash B \gg \langle In1r \ (Try \ c1 \ Catch(C \ vn) \ c2) \rangle \gg A$   
 $Env \vdash B \gg \langle c1 \ Finally \ c2 \rangle \gg A$   
 $Env \vdash B \gg \langle In1r \ (c1 \ Finally \ c2) \rangle \gg A$   
 $Env \vdash B \gg \langle Init \ C \rangle \gg A$   
 $Env \vdash B \gg \langle In1r \ (Init \ C) \rangle \gg A$   
 $Env \vdash B \gg \langle NewC \ C \rangle \gg A$   
 $Env \vdash B \gg \langle In1l \ (NewC \ C) \rangle \gg A$   
 $Env \vdash B \gg \langle New \ T[e] \rangle \gg A$   
 $Env \vdash B \gg \langle In1l \ (New \ T[e]) \rangle \gg A$   
 $Env \vdash B \gg \langle Cast \ T \ e \rangle \gg A$   
 $Env \vdash B \gg \langle In1l \ (Cast \ T \ e) \rangle \gg A$   
 $Env \vdash B \gg \langle e \ InstOf \ T \rangle \gg A$   
 $Env \vdash B \gg \langle In1l \ (e \ InstOf \ T) \rangle \gg A$   
 $Env \vdash B \gg \langle Lit \ v \rangle \gg A$   
 $Env \vdash B \gg \langle In1l \ (Lit \ v) \rangle \gg A$   
 $Env \vdash B \gg \langle UnOp \ unop \ e \rangle \gg A$   
 $Env \vdash B \gg \langle In1l \ (UnOp \ unop \ e) \rangle \gg A$   
 $Env \vdash B \gg \langle BinOp \ binop \ e1 \ e2 \rangle \gg A$   
 $Env \vdash B \gg \langle In1l \ (BinOp \ binop \ e1 \ e2) \rangle \gg A$   
 $Env \vdash B \gg \langle Super \rangle \gg A$   
 $Env \vdash B \gg \langle In1l \ (Super) \rangle \gg A$   
 $Env \vdash B \gg \langle Acc \ v \rangle \gg A$   
 $Env \vdash B \gg \langle In1l \ (Acc \ v) \rangle \gg A$   
 $Env \vdash B \gg \langle v := e \rangle \gg A$   
 $Env \vdash B \gg \langle In1l \ (v := e) \rangle \gg A$   
 $Env \vdash B \gg \langle c \ ? \ e1 : e2 \rangle \gg A$   
 $Env \vdash B \gg \langle In1l \ (c \ ? \ e1 : e2) \rangle \gg A$

$Env \vdash B \gg \langle \{accC, statT, mode\} e \cdot mn(\{pTs\} args) \rangle \gg A$   
 $Env \vdash B \gg In1l (\{accC, statT, mode\} e \cdot mn(\{pTs\} args)) \gg A$   
 $Env \vdash B \gg \langle Methd C sig \rangle \gg A$   
 $Env \vdash B \gg In1l (Methd C sig) \gg A$   
 $Env \vdash B \gg \langle Body D c \rangle \gg A$   
 $Env \vdash B \gg In1l (Body D c) \gg A$   
 $Env \vdash B \gg \langle LVar vn \rangle \gg A$   
 $Env \vdash B \gg In2 (LVar vn) \gg A$   
 $Env \vdash B \gg \langle \{accC, statDeclC, stat\} e \cdot fn \rangle \gg A$   
 $Env \vdash B \gg In2 (\{accC, statDeclC, stat\} e \cdot fn) \gg A$   
 $Env \vdash B \gg \langle e1.[e2] \rangle \gg A$   
 $Env \vdash B \gg In2 (e1.[e2]) \gg A$   
 $Env \vdash B \gg \langle [] :: expr list \rangle \gg A$   
 $Env \vdash B \gg In3 ([] :: expr list) \gg A$   
 $Env \vdash B \gg \langle e \# es \rangle \gg A$   
 $Env \vdash B \gg In3 (e \# es) \gg A$   
**declare** *inj-term-sym-simps* [*simp del*]  
**declare** *assigns-if.simps* [*simp*]  
**declare** *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]  
 $\langle ML \rangle$

**lemma** *da-Skip*:  $A = \langle nrm=B, brk=\lambda l. UNIV \rangle \implies Env \vdash B \gg \langle Skip \rangle \gg A$   
 $\langle proof \rangle$

**lemma** *da-NewC*:  $A = \langle nrm=B, brk=\lambda l. UNIV \rangle \implies Env \vdash B \gg \langle NewC C \rangle \gg A$   
 $\langle proof \rangle$

**lemma** *da-Lit*:  $A = \langle nrm=B, brk=\lambda l. UNIV \rangle \implies Env \vdash B \gg \langle Lit v \rangle \gg A$   
 $\langle proof \rangle$

**lemma** *da-Super*:  $\llbracket This \in B; A = \langle nrm=B, brk=\lambda l. UNIV \rangle \rrbracket \implies Env \vdash B \gg \langle Super \rangle \gg A$   
 $\langle proof \rangle$

**lemma** *da-Init*:  $A = \langle nrm=B, brk=\lambda l. UNIV \rangle \implies Env \vdash B \gg \langle Init C \rangle \gg A$   
 $\langle proof \rangle$

**lemma** *assignsE-subseteq-assigns-ifs*:  
**assumes** *boolEx*:  $E \vdash e :: -PrimT Boolean$  (**is** *?Boolean e*)  
**shows** *assignsE e*  $\subseteq$  *assigns-if True e*  $\cap$  *assigns-if False e* (**is** *?Incl e*)  
 $\langle proof \rangle$

**lemma** *rmlab-same-label* [*simp*]:  $(rmlab l A) l = UNIV$   
 $\langle proof \rangle$

**lemma** *rmlab-same-label1* [simp]:  $l=l' \implies (rmlab\ l\ A)\ l' = UNIV$   
 ⟨proof⟩

**lemma** *rmlab-other-label* [simp]:  $l \neq l' \implies (rmlab\ l\ A)\ l' = A\ l'$   
 ⟨proof⟩

**lemma** *range-inter-ts-subseteq* [intro]:  $\forall\ k.\ A\ k \subseteq B\ k \implies \Rightarrow \bigcap A \subseteq \Rightarrow \bigcap B$   
 ⟨proof⟩

**lemma** *range-inter-ts-subseteq'*:  
 $\llbracket \forall\ k.\ A\ k \subseteq B\ k; x \in \Rightarrow \bigcap A \rrbracket \implies x \in \Rightarrow \bigcap B$   
 ⟨proof⟩

**lemma** *da-monotone*:  
 assumes  $da: Env \vdash B \gg t \gg A$  and  
 $subsepeq\ B\ B': B \subseteq B'$  and  
 $da': Env \vdash B' \gg t \gg A'$   
 shows  $(nrm\ A \subseteq nrm\ A') \wedge (\forall\ l.\ (brk\ A\ l \subseteq brk\ A'\ l))$   
 ⟨proof⟩

**lemma** *da-weaken*:  
 assumes  $da: Env \vdash B \gg t \gg A$  and  
 $subsepeq\ B\ B': B \subseteq B'$   
 shows  $\exists\ A'. Env \vdash B' \gg t \gg A'$   
 ⟨proof⟩

**corollary** *da-weakenE* [consumes 2]:  
 assumes  $da: Env \vdash B \gg t \gg A$  and  
 $B': B \subseteq B'$  and  
 $ex\ mono: \bigwedge\ A'. \llbracket Env \vdash B' \gg t \gg A'; nrm\ A \subseteq nrm\ A';$   
 $\bigwedge\ l.\ brk\ A\ l \subseteq brk\ A'\ l \rrbracket \implies P$   
 shows  $P$   
 ⟨proof⟩

**end**



## Chapter 13

# WellForm

### 34 Well-formedness of Java programs

**theory** *WellForm* **imports** *DefiniteAssignment* **begin**

For static checks on expressions and statements, see *WellType.thy*  
improvements over Java Specification 1.0 (cf. 8.4.6.3, 8.4.6.4, 9.4.1):

- a method implementing or overwriting another method may have a result type that widens to the result type of the other method (instead of identical type)
- if a method hides another method (both methods have to be static!) there are no restrictions to the result type since the methods have to be static and there is no dynamic binding of static methods
- if an interface inherits more than one method with the same signature, the methods need not have identical return types

simplifications:

- Object and standard exceptions are assumed to be declared like normal classes

#### well-formed field declarations

well-formed field declaration (common part for classes and interfaces), cf. 8.3 and (9.3)

**constdefs**

$$\begin{aligned} wf\_fdecl &:: prog \Rightarrow pname \Rightarrow fdecl \Rightarrow bool \\ wf\_fdecl \ G \ P &\equiv \lambda(fn,f). \ is\_acc\_type \ G \ P \ (type \ f) \end{aligned}$$

**lemma** *wf-fdecl-def2*:  $\bigwedge fd. \ wf\_fdecl \ G \ P \ fd = is\_acc\_type \ G \ P \ (type \ (snd \ fd))$   
*<proof>*

#### well-formed method declarations

A method head is wellformed if:

- the signature and the method head agree in the number of parameters
- all types of the parameters are visible
- the result type is visible
- the parameter names are unique

**constdefs**

$$\begin{aligned} wf\_mhead &:: prog \Rightarrow pname \Rightarrow sig \Rightarrow mhead \Rightarrow bool \\ wf\_mhead \ G \ P &\equiv \lambda \ sig \ mh. \ length \ (parTs \ sig) = length \ (pars \ mh) \wedge \\ &\quad (\ \forall \ T \in set \ (parTs \ sig). \ is\_acc\_type \ G \ P \ T) \wedge \\ &\quad is\_acc\_type \ G \ P \ (resTy \ mh) \wedge \\ &\quad distinct \ (pars \ mh) \end{aligned}$$

A method declaration is wellformed if:

- the method head is wellformed
- the names of the local variables are unique
- the types of the local variables must be accessible

- the local variables don't shadow the parameters
- the class of the method is defined
- the body statement is welltyped with respect to the modified environment of local names, were the local variables, the parameters the special result variable (Res) and This are assoziated with there types.

**constdefs** *callee-lcl* :: *qname*  $\Rightarrow$  *sig*  $\Rightarrow$  *methd*  $\Rightarrow$  *lenv*  
*callee-lcl* *C* *sig* *m*  
 $\equiv \lambda k. (case\ k\ of$   
 $\quad EName\ e$   
 $\quad \Rightarrow (case\ e\ of$   
 $\quad \quad VName\ v$   
 $\quad \quad \Rightarrow (table-of\ (lcls\ (mbody\ m))((pars\ m)[\mapsto](parTs\ sig)))\ v$   
 $\quad \quad | Res \Rightarrow Some\ (resTy\ m))$   
 $\quad | This \Rightarrow if\ is-static\ m\ then\ None\ else\ Some\ (Class\ C))$

**constdefs** *parameters* :: *methd*  $\Rightarrow$  *lname* *set*  
*parameters* *m*  $\equiv set\ (map\ (EName\ o\ VName)\ (pars\ m))$   
 $\cup\ (if\ (static\ m)\ then\ \{\}\ else\ \{This\})$

**constdefs**  
*wf-mdecl* :: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  *mdecl*  $\Rightarrow$  *bool*  
*wf-mdecl* *G* *C*  $\equiv$   
 $\lambda(sig,m).$   
 $wf-mhead\ G\ (pid\ C)\ sig\ (mhead\ m) \wedge$   
 $unique\ (lcls\ (mbody\ m)) \wedge$   
 $(\forall (vn,T) \in set\ (lcls\ (mbody\ m)).\ is-acc-type\ G\ (pid\ C)\ T) \wedge$   
 $(\forall pn \in set\ (pars\ m).\ table-of\ (lcls\ (mbody\ m))\ pn = None) \wedge$   
 $jumpNestingOkS\ \{Ret\}\ (stmt\ (mbody\ m)) \wedge$   
 $is-class\ G\ C \wedge$   
 $(\langle prg=G, cls=C, lcl=callee-lcl\ C\ sig\ m \rangle \vdash (stmt\ (mbody\ m))) :: \checkmark \wedge$   
 $(\exists\ A.\ \langle prg=G, cls=C, lcl=callee-lcl\ C\ sig\ m \rangle$   
 $\quad \vdash\ parameters\ m \gg (stmt\ (mbody\ m)) \gg A$   
 $\quad \wedge\ Result \in nrm\ A)$

**lemma** *callee-lcl-VName-simp* [simp]:  
*callee-lcl* *C* *sig* *m* (*EName* (*VName* *v*))  
 $= (table-of\ (lcls\ (mbody\ m))((pars\ m)[\mapsto](parTs\ sig)))\ v$   
 $\langle proof \rangle$

**lemma** *callee-lcl-Res-simp* [simp]:  
*callee-lcl* *C* *sig* *m* (*EName* *Res*) = *Some* (*resTy* *m*)  
 $\langle proof \rangle$

**lemma** *callee-lcl-This-simp* [simp]:  
*callee-lcl* *C* *sig* *m* (*This*) = (*if* *is-static* *m* *then* *None* *else* *Some* (*Class* *C*))  
 $\langle proof \rangle$

**lemma** *callee-lcl-This-static-simp*:  
*is-static* *m*  $\implies$  *callee-lcl* *C* *sig* *m* (*This*) = *None*  
 $\langle proof \rangle$

**lemma** *callee-lcl-This-not-static-simp*:

$\neg \text{is-static } m \implies \text{callee-lcl } C \text{ sig } m \text{ (This)} = \text{Some (Class } C)$

$\langle \text{proof} \rangle$

**lemma** *wf-mheadI*:

$\llbracket \text{length (parTs sig)} = \text{length (pars m)}; \forall T \in \text{set (parTs sig)}. \text{is-acc-type } G \text{ P } T;$   
 $\text{is-acc-type } G \text{ P (resTy m)}; \text{distinct (pars m)} \rrbracket \implies$   
 $\text{wf-mhead } G \text{ P sig } m$

$\langle \text{proof} \rangle$

**lemma** *wf-mdeclI*:  $\llbracket$

$\text{wf-mhead } G \text{ (pid } C) \text{ sig (mhead m)}; \text{unique (lcls (mbody m))};$   
 $(\forall pn \in \text{set (pars m)}. \text{table-of (lcls (mbody m)) } pn = \text{None});$   
 $\forall (vn, T) \in \text{set (lcls (mbody m))}. \text{is-acc-type } G \text{ (pid } C) \text{ } T;$   
 $\text{jumpNestingOkS \{Ret\} (stmt (mbody m))};$   
 $\text{is-class } G \text{ } C;$   
 $(\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \text{ sig } m \rrbracket \vdash \text{stmt (mbody m)})::\sqrt{}$   
 $(\exists A. (\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \text{ sig } m \rrbracket \vdash \text{parameters } m \gg \langle \text{stmt (mbody m)} \rangle \gg A$   
 $\wedge \text{Result} \in \text{nrm } A)$

$\rrbracket \implies$

$\text{wf-mdecl } G \text{ } C \text{ (sig, m)}$

$\langle \text{proof} \rangle$

**lemma** *wf-mdeclE [consumes 1]*:

$\llbracket \text{wf-mdecl } G \text{ } C \text{ (sig, m)};$   
 $\llbracket \text{wf-mhead } G \text{ (pid } C) \text{ sig (mhead m)}; \text{unique (lcls (mbody m))};$   
 $\forall pn \in \text{set (pars m)}. \text{table-of (lcls (mbody m)) } pn = \text{None};$   
 $\forall (vn, T) \in \text{set (lcls (mbody m))}. \text{is-acc-type } G \text{ (pid } C) \text{ } T;$   
 $\text{jumpNestingOkS \{Ret\} (stmt (mbody m))};$   
 $\text{is-class } G \text{ } C;$   
 $(\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \text{ sig } m \rrbracket \vdash \text{stmt (mbody m)})::\sqrt{}$   
 $(\exists A. (\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \text{ sig } m \rrbracket \vdash \text{parameters } m \gg \langle \text{stmt (mbody m)} \rangle \gg A$   
 $\wedge \text{Result} \in \text{nrm } A)$

$\rrbracket \implies P$

$\rrbracket \implies P$

$\langle \text{proof} \rangle$

**lemma** *wf-mdeclD1*:

$\text{wf-mdecl } G \text{ } C \text{ (sig, m)} \implies$   
 $\text{wf-mhead } G \text{ (pid } C) \text{ sig (mhead m)} \wedge \text{unique (lcls (mbody m))} \wedge$   
 $(\forall pn \in \text{set (pars m)}. \text{table-of (lcls (mbody m)) } pn = \text{None}) \wedge$   
 $(\forall (vn, T) \in \text{set (lcls (mbody m))}. \text{is-acc-type } G \text{ (pid } C) \text{ } T)$

$\langle \text{proof} \rangle$

**lemma** *wf-mdecl-bodyD*:

$\text{wf-mdecl } G \text{ } C \text{ (sig, m)} \implies$   
 $(\exists T. (\llbracket \text{prg} = G, \text{cls} = C, \text{lcl} = \text{callee-lcl } C \text{ sig } m \rrbracket \vdash \text{Body } C \text{ (stmt (mbody m))})::\neg T \wedge$   
 $G \vdash T \preceq (\text{resTy } m))$

$\langle \text{proof} \rangle$



**lemma** *rT-is-acc-type*:

*wf-mhead G P sig m  $\implies$  is-acc-type G P (resTy m)*  
*<proof>*

### well-formed interface declarations

A interface declaration is wellformed if:

- the interface hierarchy is wellstructured
- there is no class with the same name
- the method heads are wellformed and not static and have Public access
- the methods are uniquely named
- all superinterfaces are accessible
- the result type of a method overriding a method of Object widens to the result type of the overridden method. Shadowing static methods is forbidden.
- the result type of a method overriding a set of methods defined in the superinterfaces widens to each of the corresponding result types

**constdefs**

*wf-idecl :: prog  $\Rightarrow$  idecl  $\Rightarrow$  bool*  
*wf-idecl G  $\equiv$*   
 $\lambda(I, i).$   
 $ws-idecl\ G\ I\ (isuperIfs\ i) \wedge$   
 $\neg is-class\ G\ I \wedge$   
 $(\forall (sig, mh) \in set\ (imethods\ i). wf-mhead\ G\ (pid\ I)\ sig\ mh \wedge$   
 $\neg is-static\ mh \wedge$   
 $accmodi\ mh = Public) \wedge$   
 $unique\ (imethods\ i) \wedge$   
 $(\forall J \in set\ (isuperIfs\ i). is-acc-iface\ G\ (pid\ I)\ J) \wedge$   
 $(table-of\ (imethods\ i)$   
 $hiding\ (methd\ G\ Object)$   
 $under\ (\lambda new\ old. accmodi\ old \neq Private)$   
 $entails\ (\lambda new\ old. G \vdash resTy\ new \preceq resTy\ old \wedge$   
 $is-static\ new = is-static\ old)) \wedge$   
 $(o2s \circ table-of\ (imethods\ i)$   
 $hidings\ Un-tables((\lambda J. (imethds\ G\ J)) 'set\ (isuperIfs\ i))$   
 $entails\ (\lambda new\ old. G \vdash resTy\ new \preceq resTy\ old))$

**lemma** *wf-idecl-mhead*:  $\llbracket wf-idecl\ G\ (I, i); (sig, mh) \in set\ (imethods\ i) \rrbracket \implies$

*wf-mhead G (pid I) sig mh  $\wedge \neg is-static\ mh \wedge accmodi\ mh = Public$*   
*<proof>*

**lemma** *wf-idecl-hidings*:

*wf-idecl G (I, i)  $\implies$*   
 $(\lambda s. o2s\ (table-of\ (imethods\ i)\ s))$   
 $hidings\ Un-tables\ ((\lambda J. imethds\ G\ J)\ 'set\ (isuperIfs\ i))$   
 $entails\ \lambda new\ old. G \vdash resTy\ new \preceq resTy\ old$   
*<proof>*

**lemma** *wf-idecl-hiding*:

*wf-idecl*  $G (I, i) \implies$   
*table-of* (*imethods*  $i$ )  
           *hiding* (*methd*  $G$  *Object*)  
           *under*  $(\lambda \text{ new old. } \text{accmodi } \text{old} \neq \text{Private})$   
           *entails*  $(\lambda \text{ new old. } G \vdash_{\text{resTy}} \text{new} \leq_{\text{resTy}} \text{old} \wedge$   
   *is-static*  $\text{new} = \text{is-static } \text{old}))$   
 $\langle \text{proof} \rangle$

**lemma** *wf-idecl-supD*:

$\llbracket \text{wf-idecl } G (I, i); J \in \text{set } (\text{isuperIfs } i) \rrbracket$   
 $\implies \text{is-acc-iface } G (\text{pid } I) J \wedge (J, I) \notin (\text{subint1 } G) \hat{+}$   
 $\langle \text{proof} \rangle$

## well-formed class declarations

A class declaration is wellformed if:

- there is no interface with the same name
- all superinterfaces are accessible and for all methods implementing an interface method the result type widens to the result type of the interface method, the method is not static and offers at least as much access (this actually means that the method has Public access, since all interface methods have public access)
- all field declarations are wellformed and the field names are unique
- all method declarations are wellformed and the method names are unique
- the initialization statement is welltyped
- the classhierarchy is wellstructured
- Unless the class is *Object*:
  - the superclass is accessible
  - for all methods overriding another method (of a superclass) the result type widens to the result type of the overridden method, the access modifier of the new method provides at least as much access as the overwritten one.
  - for all methods hiding a method (of a superclass) the hidden method must be static and offer at least as much access rights. Remark: In contrast to the Java Language Specification we don't restrict the result types of the method (as in case of overriding), because there seems to be no reason, since there is no dynamic binding of static methods. (cf. 8.4.6.3 vs. 15.12.1). Stricly speaking the restrictions on the access rights aren't necessary to, since the static type and the access rights together determine which method is to be called statically. But if a class gains more then one static method with the same signature due to inheritance, it is confusing when the method selection depends on the access rights only: e.g. Class C declares static public method *foo()*. Class D is subclass of C and declares static method *foo()* with default package access. *D.foo()* ? if this call is in the same package as D then *foo* of class D is called, otherwise *foo* of class C.

**constdefs** *entails*::  $(\text{'a, 'b}) \text{ table} \Rightarrow (\text{'b} \Rightarrow \text{bool}) \Rightarrow \text{bool}$   
    $(- \text{ entails } - 20)$

$t \text{ entails } P \equiv \forall k. \forall x \in t \ k: P \ x$

**lemma** *entailsD*:

$\llbracket t \text{ entails } P; t \text{ k} = \text{Some } x \rrbracket \implies P \ x$

$\langle \text{proof} \rangle$

**lemma** *empty-entails[simp]*: *empty entails P*

$\langle \text{proof} \rangle$

**constdefs**

*wf-cdecl* :: *prog*  $\Rightarrow$  *cdecl*  $\Rightarrow$  *bool*

*wf-cdecl* *G*  $\equiv$

$\lambda(C, c).$   
 $\neg \text{is-iface } G \ C \wedge$   
 $(\forall I \in \text{set } (\text{superIfs } c). \text{is-acc-iface } G \ (\text{pid } C) \ I \wedge$   
 $(\forall s. \forall im \in \text{imethds } G \ I \ s.$   
 $(\exists cm \in \text{methd } G \ C \ s: G \vdash \text{resTy } cm \preceq \text{resTy } im \wedge$   
 $\neg \text{is-static } cm \wedge$   
 $\text{accmodi } im \leq \text{accmodi } cm))) \wedge$   
 $(\forall f \in \text{set } (\text{cfields } c). \text{wf-fdecl } G \ (\text{pid } C) \ f) \wedge \text{unique } (\text{cfields } c) \wedge$   
 $(\forall m \in \text{set } (\text{methods } c). \text{wf-mdecl } G \ C \ m) \wedge \text{unique } (\text{methods } c) \wedge$   
 $\text{jumpNestingOkS } \{\} \ (\text{init } c) \wedge$   
 $(\exists A. (\text{prg}=G, \text{cls}=C, \text{lcl}=\text{empty}) \vdash \{\} \gg \langle \text{init } c \rangle \gg A) \wedge$   
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=\text{empty}) \vdash (\text{init } c) :: \checkmark \wedge \text{ws-cdecl } G \ C \ (\text{super } c) \wedge$   
 $(C \neq \text{Object} \longrightarrow$   
 $(\text{is-acc-class } G \ (\text{pid } C) \ (\text{super } c) \wedge$   
 $(\text{table-of } (\text{map } (\lambda (s, m). (s, C, m)) (\text{methods } c))$   
 $\text{entails } (\lambda \text{ new}. \forall \text{ old sig.}$   
 $(G, \text{sig} \vdash \text{new overrides}_S \text{ old}$   
 $\longrightarrow (G \vdash \text{resTy } \text{new} \preceq \text{resTy } \text{old} \wedge$   
 $\text{accmodi } \text{old} \leq \text{accmodi } \text{new} \wedge$   
 $\neg \text{is-static } \text{old})) \wedge$   
 $(G, \text{sig} \vdash \text{new hides old}$   
 $\longrightarrow (\text{accmodi } \text{old} \leq \text{accmodi } \text{new} \wedge$   
 $\text{is-static } \text{old}))))$   
 $)$

**lemma** *wf-cdeclE* [*consumes 1*]:

$\llbracket \text{wf-cdecl } G \ (C, c);$

$\llbracket \neg \text{is-iface } G \ C;$

$(\forall I \in \text{set } (\text{superIfs } c). \text{is-acc-iface } G \ (\text{pid } C) \ I \wedge$

$(\forall s. \forall im \in \text{imethds } G \ I \ s.$

$(\exists cm \in \text{methd } G \ C \ s: G \vdash \text{resTy } cm \preceq \text{resTy } im \wedge$

$\neg \text{is-static } cm \wedge$

$\text{accmodi } im \leq \text{accmodi } cm))));$

$\forall f \in \text{set } (\text{cfields } c). \text{wf-fdecl } G \ (\text{pid } C) \ f; \text{unique } (\text{cfields } c);$

$\forall m \in \text{set } (\text{methods } c). \text{wf-mdecl } G \ C \ m; \text{unique } (\text{methods } c);$

$\text{jumpNestingOkS } \{\} \ (\text{init } c);$

$\exists A. (\text{prg}=G, \text{cls}=C, \text{lcl}=\text{empty}) \vdash \{\} \gg \langle \text{init } c \rangle \gg A;$

$(\text{prg}=G, \text{cls}=C, \text{lcl}=\text{empty}) \vdash (\text{init } c) :: \checkmark;$

$\text{ws-cdecl } G \ C \ (\text{super } c);$

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

$(\text{is-acc-class } G \ (\text{pid } C) \ (\text{super } c) \wedge$

$(\text{table-of } (\text{map } (\lambda (s, m). (s, C, m)) (\text{methods } c))$

$\text{entails } (\lambda \text{ new}. \forall \text{ old sig.}$

$(G, \text{sig} \vdash \text{new overrides}_S \text{ old}$

$\longrightarrow (G \vdash \text{resTy } \text{new} \preceq \text{resTy } \text{old} \wedge$

$$\begin{aligned}
& \text{accmodi } old \leq \text{accmodi } new \wedge \\
& \neg \text{is-static } old)) \wedge \\
& (G, \text{sig} \vdash \text{new hides } old \\
& \longrightarrow (\text{accmodi } old \leq \text{accmodi } new \wedge \\
& \text{is-static } old))) \\
& )) \Rightarrow P \\
& \Rightarrow P \\
& \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *wf-cdecl-unique*:

$\text{wf-cdecl } G \ (C, c) \Rightarrow \text{unique } (c\text{fields } c) \wedge \text{unique } (\text{methods } c)$   
 $\langle \text{proof} \rangle$

**lemma** *wf-cdecl-fdecl*:

$\llbracket \text{wf-cdecl } G \ (C, c); f \in \text{set } (c\text{fields } c) \rrbracket \Rightarrow \text{wf-fdecl } G \ (\text{pid } C) \ f$   
 $\langle \text{proof} \rangle$

**lemma** *wf-cdecl-mdecl*:

$\llbracket \text{wf-cdecl } G \ (C, c); m \in \text{set } (\text{methods } c) \rrbracket \Rightarrow \text{wf-mdecl } G \ C \ m$   
 $\langle \text{proof} \rangle$

**lemma** *wf-cdecl-impD*:

$\llbracket \text{wf-cdecl } G \ (C, c); I \in \text{set } (\text{superIfs } c) \rrbracket$   
 $\Rightarrow \text{is-acc-iface } G \ (\text{pid } C) \ I \wedge$   
 $(\forall s. \forall im \in \text{imethds } G \ I \ s.$   
 $(\exists cm \in \text{methd } G \ C \ s: G \vdash \text{resTy } cm \preceq \text{resTy } im \wedge \neg \text{is-static } cm \wedge$   
 $\text{accmodi } im \leq \text{accmodi } cm))$   
 $\langle \text{proof} \rangle$

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

$\llbracket \text{wf-cdecl } G \ (C, c); C \neq \text{Object} \rrbracket \Rightarrow$   
 $\text{is-acc-class } G \ (\text{pid } C) \ (\text{super } c) \wedge (\text{super } c, C) \notin (\text{subcls1 } G) \wedge$   
 $(\text{table-of } (\text{map } (\lambda (s, m). (s, C, m)) (\text{methods } c))$   
 $\text{entails } (\lambda \text{new}. \forall \text{old sig.}$   
 $(G, \text{sig} \vdash \text{new overrides}_S \text{old}$   
 $\longrightarrow (G \vdash \text{resTy } \text{new} \preceq \text{resTy } \text{old} \wedge$   
 $\text{accmodi } \text{old} \leq \text{accmodi } \text{new} \wedge$   
 $\neg \text{is-static } \text{old})) \wedge$   
 $(G, \text{sig} \vdash \text{new hides } \text{old}$   
 $\longrightarrow (\text{accmodi } \text{old} \leq \text{accmodi } \text{new} \wedge$   
 $\text{is-static } \text{old}))))$   
 $\langle \text{proof} \rangle$

**lemma** *wf-cdecl-overrides-SomeD*:

$\llbracket \text{wf-cdecl } G \ (C, c); C \neq \text{Object}; \text{table-of } (\text{methods } c) \text{ sig} = \text{Some } \text{newM};$   
 $G, \text{sig} \vdash (C, \text{newM}) \text{ overrides}_S \text{old}$   
 $\rrbracket \Rightarrow G \vdash \text{resTy } \text{newM} \preceq \text{resTy } \text{old} \wedge$   
 $\text{accmodi } \text{old} \leq \text{accmodi } \text{newM} \wedge$   
 $\neg \text{is-static } \text{old}$   
 $\langle \text{proof} \rangle$

**lemma** *wf-cdecl-hides-SomeD*:

$\llbracket wf\text{-}cdecl\ G\ (C, c); C \neq Object; table\text{-}of\ (methods\ c)\ sig = Some\ newM;$   
 $G, sig \vdash (C, newM)\ hides\ old$   
 $\rrbracket \implies accmodi\ old \leq access\ newM \wedge$   
 $is\text{-}static\ old$   
 $\langle proof \rangle$

**lemma** *wf-cdecl-wt-init*:

$wf\text{-}cdecl\ G\ (C, c) \implies (\llbracket prg = G, cls = C, lcl = empty \rrbracket) \vdash init\ c :: \checkmark$   
 $\langle proof \rangle$

## well-formed programs

A program declaration is wellformed if:

- the class ObjectC of Object is defined
- every method of Object has an access modifier distinct from Package. This is necessary since every interface automatically inherits from Object. We must know, that every time a Object method is "overridden" by an interface method this is also overridden by the class implementing the the interface (see *implement-dynmethd* and *class-mheadsD*)
- all standard Exceptions are defined
- all defined interfaces are wellformed
- all defined classes are wellformed

**constdefs**

$wf\text{-}prog :: prog \Rightarrow bool$   
 $wf\text{-}prog\ G \equiv let\ is = ifaces\ G; cs = classes\ G\ in$   
 $ObjectC \in set\ cs \wedge$   
 $(\forall m \in set\ Object\text{-}mdecls. accmodi\ m \neq Package) \wedge$   
 $(\forall xn. SXcptC\ xn \in set\ cs) \wedge$   
 $(\forall i \in set\ is. wf\text{-}idecl\ G\ i) \wedge unique\ is \wedge$   
 $(\forall c \in set\ cs. wf\text{-}cdecl\ G\ c) \wedge unique\ cs$

**lemma** *wf-prog-idecl*:  $\llbracket iface\ G\ I = Some\ i; wf\text{-}prog\ G \rrbracket \implies wf\text{-}idecl\ G\ (I, i)$

$\langle proof \rangle$

**lemma** *wf-prog-cdecl*:  $\llbracket class\ G\ C = Some\ c; wf\text{-}prog\ G \rrbracket \implies wf\text{-}cdecl\ G\ (C, c)$

$\langle proof \rangle$

**lemma** *wf-prog-Object-mdecls*:

$wf\text{-}prog\ G \implies (\forall m \in set\ Object\text{-}mdecls. accmodi\ m \neq Package)$   
 $\langle proof \rangle$

**lemma** *wf-prog-acc-superD*:

$\llbracket wf\text{-}prog\ G; class\ G\ C = Some\ c; C \neq Object \rrbracket$   
 $\implies is\text{-}acc\text{-}class\ G\ (pid\ C)\ (super\ c)$   
 $\langle proof \rangle$

**lemma** *wf-ws-prog* [*elim!*, *simp*]:  $wf\text{-}prog\ G \implies ws\text{-}prog\ G$

$\langle \text{proof} \rangle$

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

$\text{wf-prog } G \implies$

$\text{class } G \text{ Object} = \text{Some } (\llbracket \text{access}=\text{Public}, \text{cfields}=\llbracket, \text{methods}=\text{Object-mdecls},$   
 $\text{init}=\text{Skip}, \text{super}=\text{arbitrary}, \text{superIfs}=\llbracket \rrbracket)$

$\langle \text{proof} \rangle$

**lemma** *methd-Object*[simp]:  $\text{wf-prog } G \implies \text{methd } G \text{ Object} =$

$\text{table-of } (\text{map } (\lambda(s,m). (s, \text{Object}, m)) \text{ Object-mdecls})$

$\langle \text{proof} \rangle$

**lemma** *wf-prog-Object-methd*:

$\llbracket \text{wf-prog } G; \text{methd } G \text{ Object sig} = \text{Some } m \rrbracket \implies \text{accmodi } m \neq \text{Package}$

$\langle \text{proof} \rangle$

**lemma** *wf-prog-Object-is-public*[intro]:

$\text{wf-prog } G \implies \text{is-public } G \text{ Object}$

$\langle \text{proof} \rangle$

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

$\text{wf-prog } G \implies$

$\text{class } G (\text{SXcpt } xn) = \text{Some } (\llbracket \text{access}=\text{Public}, \text{cfields}=\llbracket, \text{methods}=\text{SXcpt-mdecls},$   
 $\text{init}=\text{Skip},$   
 $\text{super}=\text{if } xn = \text{Throwable} \text{ then } \text{Object}$   
 $\text{else } \text{SXcpt } \text{Throwable},$   
 $\text{superIfs}=\llbracket \rrbracket)$

$\langle \text{proof} \rangle$

**lemma** *wf-ObjectC* [simp]:

$\text{wf-cdecl } G \text{ ObjectC} = (\neg \text{is-iface } G \text{ Object} \wedge \text{Ball } (\text{set } \text{Object-mdecls})$   
 $(\text{wf-mdecl } G \text{ Object}) \wedge \text{unique } \text{Object-mdecls})$

$\langle \text{proof} \rangle$

**lemma** *Object-is-class* [simp, elim!]:  $\text{wf-prog } G \implies \text{is-class } G \text{ Object}$

$\langle \text{proof} \rangle$

**lemma** *Object-is-acc-class* [simp, elim!]:  $\text{wf-prog } G \implies \text{is-acc-class } G \text{ S Object}$

$\langle \text{proof} \rangle$

**lemma** *SXcpt-is-class* [simp, elim!]:  $\text{wf-prog } G \implies \text{is-class } G (\text{SXcpt } xn)$

$\langle \text{proof} \rangle$

**lemma** *SXcpt-is-acc-class* [simp, elim!]:

$\text{wf-prog } G \implies \text{is-acc-class } G \text{ S } (\text{SXcpt } xn)$

$\langle \text{proof} \rangle$

**lemma** *fields-Object* [simp]:  $\text{wf-prog } G \implies \text{DeclConcepts.fields } G \text{ Object} = \llbracket$

$\langle \text{proof} \rangle$

**lemma** *accfield-Object* [simp]:  
 $\text{wf-prog } G \implies \text{accfield } G \text{ S Object} = \text{empty}$   
 $\langle \text{proof} \rangle$

**lemma** *fields-Throwable* [simp]:  
 $\text{wf-prog } G \implies \text{DeclConcepts.fields } G \text{ (SXcpt Throwable)} = []$   
 $\langle \text{proof} \rangle$

**lemma** *fields-SXcpt* [simp]:  $\text{wf-prog } G \implies \text{DeclConcepts.fields } G \text{ (SXcpt } xn) = []$   
 $\langle \text{proof} \rangle$

**lemmas** *widen-trans* = *ws-widen-trans* [OF - - wf-ws-prog, elim]

**lemma** *widen-trans2* [elim]:  $\llbracket G \vdash U \preceq T; G \vdash S \preceq U; \text{wf-prog } G \rrbracket \implies G \vdash S \preceq T$   
 $\langle \text{proof} \rangle$

**lemma** *Xcpt-subcls-Throwable* [simp]:  
 $\text{wf-prog } G \implies G \vdash \text{SXcpt } xn \preceq_C \text{ SXcpt Throwable}$   
 $\langle \text{proof} \rangle$

**lemma** *unique-fields*:  
 $\llbracket \text{is-class } G \text{ C}; \text{wf-prog } G \rrbracket \implies \text{unique } (\text{DeclConcepts.fields } G \text{ C})$   
 $\langle \text{proof} \rangle$

**lemma** *fields-mono*:  
 $\llbracket \text{table-of } (\text{DeclConcepts.fields } G \text{ C}) \text{ fn} = \text{Some } f; G \vdash D \preceq_C C; \text{is-class } G \text{ D}; \text{wf-prog } G \rrbracket$   
 $\implies \text{table-of } (\text{DeclConcepts.fields } G \text{ D}) \text{ fn} = \text{Some } f$   
 $\langle \text{proof} \rangle$

**lemma** *fields-is-type* [elim]:  
 $\llbracket \text{table-of } (\text{DeclConcepts.fields } G \text{ C}) \text{ m} = \text{Some } f; \text{wf-prog } G; \text{is-class } G \text{ C} \rrbracket \implies$   
 $\text{is-type } G \text{ (type } f)$   
 $\langle \text{proof} \rangle$

**lemma** *imethds-wf-mhead* [rule-format (no-asm)]:  
 $\llbracket m \in \text{imethds } G \text{ I sig}; \text{wf-prog } G; \text{is-iface } G \text{ I} \rrbracket \implies$   
 $\text{wf-mhead } G \text{ (pid (decliface m)) sig (mthd m)} \wedge$   
 $\neg \text{is-static } m \wedge \text{accmodi } m = \text{Public}$   
 $\langle \text{proof} \rangle$

**lemma** *methd-wf-mdecl*:  
 $\llbracket \text{methd } G \text{ C sig} = \text{Some } m; \text{wf-prog } G; \text{class } G \text{ C} = \text{Some } y \rrbracket \implies$   
 $G \vdash C \preceq_C (\text{declclass } m) \wedge \text{is-class } G \text{ (declclass } m) \wedge$   
 $\text{wf-mdecl } G \text{ (declclass } m) \text{ (sig, (mthd } m))$   
 $\langle \text{proof} \rangle$

**lemma** *methd-rT-is-type*:  
 $\llbracket wf\text{-prog } G; methd\ G\ C\ sig = Some\ m; \\ class\ G\ C = Some\ y \rrbracket \\ \implies is\text{-type}\ G\ (resTy\ m) \\ \langle proof \rangle$

**lemma** *accmethd-rT-is-type*:  
 $\llbracket wf\text{-prog } G; accmethd\ G\ S\ C\ sig = Some\ m; \\ class\ G\ C = Some\ y \rrbracket \\ \implies is\text{-type}\ G\ (resTy\ m) \\ \langle proof \rangle$

**lemma** *methd-Object-SomeD*:  
 $\llbracket wf\text{-prog } G; methd\ G\ Object\ sig = Some\ m \rrbracket \\ \implies declclass\ m = Object \\ \langle proof \rangle$

**lemma** *wf-imethdsD*:  
 $\llbracket im \in imethds\ G\ I\ sig; wf\text{-prog } G; is\text{-iface}\ G\ I \rrbracket \\ \implies \neg is\text{-static}\ im \wedge accmodi\ im = Public \\ \langle proof \rangle$

**lemma** *wf-prog-hidesD*:  
**assumes** *hides*:  $G \vdash new\ overrides\ old$  **and** *wf*:  $wf\text{-prog } G$   
**shows**  
 $accmodi\ old \leq accmodi\ new \wedge \\ is\text{-static}\ old \\ \langle proof \rangle$

Compare this lemma about static overriding  $G \vdash new\ overrides_S\ old$  with the definition of dynamic overriding  $G \vdash new\ overrides\ old$ . Conforming result types and restrictions on the access modifiers of the old and the new method are not part of the predicate for static overriding. But they are enshured in a wellfromed program. Dynamic overriding has no restrictions on the access modifiers but enforces confrom result types as precondition. But with some efford we can guarantee the access modifier restriction for dynamic overriding, too. See lemma *wf-prog-dyn-override-prop*.

**lemma** *wf-prog-stat-overridesD*:  
**assumes** *stat-override*:  $G \vdash new\ overrides_S\ old$  **and** *wf*:  $wf\text{-prog } G$   
**shows**  
 $G \vdash resTy\ new \preceq resTy\ old \wedge \\ accmodi\ old \leq accmodi\ new \wedge \\ \neg is\text{-static}\ old \\ \langle proof \rangle$

**lemma** *static-to-dynamic-overriding*:  
**assumes** *stat-override*:  $G \vdash new\ overrides_S\ old$  **and** *wf* :  $wf\text{-prog } G$   
**shows**  $G \vdash new\ overrides\ old \\ \langle proof \rangle$



**lemma** *non-Package-instance-method-inheritance*:

**assumes** *old-inheritable*:  $G \vdash \text{Method old inheritable-in } (\text{pid } C)$  **and**  
*accmodi-old*:  $\text{accmodi old} \neq \text{Package}$  **and**  
*instance-method*:  $\neg \text{is-static old}$  **and**  
*subcls*:  $G \vdash C \prec_C \text{declclass old}$  **and**  
*old-declared*:  $G \vdash \text{Method old declared-in } (\text{declclass old})$  **and**  
*wf*:  $\text{wf-prog } G$   
**shows**  $G \vdash \text{Method old member-of } C \vee$   
 $(\exists \text{ new. } G \vdash \text{new overrides}_S \text{ old} \wedge G \vdash \text{Method new member-of } C)$   
 $\langle \text{proof} \rangle$

**lemma** *non-Package-instance-method-inheritance-cases* [consumes 6, case-names *Inheritance Overriding*]:

**assumes** *old-inheritable*:  $G \vdash \text{Method old inheritable-in } (\text{pid } C)$  **and**  
*accmodi-old*:  $\text{accmodi old} \neq \text{Package}$  **and**  
*instance-method*:  $\neg \text{is-static old}$  **and**  
*subcls*:  $G \vdash C \prec_C \text{declclass old}$  **and**  
*old-declared*:  $G \vdash \text{Method old declared-in } (\text{declclass old})$  **and**  
*wf*:  $\text{wf-prog } G$  **and**  
*inheritance*:  $G \vdash \text{Method old member-of } C \implies P$  **and**  
*overriding*:  $\bigwedge \text{ new. } \llbracket G \vdash \text{new overrides}_S \text{ old}; G \vdash \text{Method new member-of } C \rrbracket \implies P$   
**shows**  $P$   
 $\langle \text{proof} \rangle$

**lemma** *dynamic-to-static-overriding*:

**assumes** *dyn-override*:  $G \vdash \text{new overrides old}$  **and**  
*accmodi-old*:  $\text{accmodi old} \neq \text{Package}$  **and**  
*wf*:  $\text{wf-prog } G$   
**shows**  $G \vdash \text{new overrides}_S \text{ old}$   
 $\langle \text{proof} \rangle$

**lemma** *wf-prog-dyn-override-prop*:

**assumes** *dyn-override*:  $G \vdash \text{new overrides old}$  **and**  
*wf*:  $\text{wf-prog } G$   
**shows**  $\text{accmodi old} \leq \text{accmodi new}$   
 $\langle \text{proof} \rangle$

**lemma** *overrides-Package-old*:

**assumes** *dyn-override*:  $G \vdash \text{new overrides old}$  **and**  
*accmodi-new*:  $\text{accmodi new} = \text{Package}$  **and**  
*wf*:  $\text{wf-prog } G$   
**shows**  $\text{accmodi old} = \text{Package}$   
 $\langle \text{proof} \rangle$

**lemma** *dyn-override-Package*:

**assumes** *dyn-override*:  $G \vdash \text{new overrides old}$  **and**  
*accmodi-old*:  $\text{accmodi old} = \text{Package}$  **and**  
*accmodi-new*:  $\text{accmodi new} = \text{Package}$  **and**  
*wf*:  $\text{wf-prog } G$   
**shows**  $\text{pid } (\text{declclass old}) = \text{pid } (\text{declclass new})$   
 $\langle \text{proof} \rangle$

**lemma** *dyn-override-Package-escape*:

**assumes** *dyn-override*:  $G \vdash \text{new overrides old}$  **and**  
*accmodi-old*:  $\text{accmodi old} = \text{Package}$  **and**  
*outside-pack*:  $\text{pid}(\text{declclass old}) \neq \text{pid}(\text{declclass new})$  **and**  
*wf*: *wf-prog*  $G$   
**shows**  $\exists \text{ inter. } G \vdash \text{new overrides inter} \wedge G \vdash \text{inter overrides old} \wedge$   
 $\text{pid}(\text{declclass old}) = \text{pid}(\text{declclass inter}) \wedge$   
 $\text{Protected} \leq \text{accmodi inter}$

$\langle \text{proof} \rangle$

**lemma** *declclass-widen*[*rule-format*]:

*wf-prog*  $G$   
 $\longrightarrow (\forall c \ m. \text{class } G \ C = \text{Some } c \longrightarrow \text{methd } G \ C \ \text{sig} = \text{Some } m$   
 $\longrightarrow G \vdash C \preceq_C \text{declclass } m) \text{ (is ?P } G \ C)$

$\langle \text{proof} \rangle$

**lemma** *declclass-methd-Object*:

$\llbracket \text{wf-prog } G; \text{methd } G \ \text{Object} \ \text{sig} = \text{Some } m \rrbracket \Longrightarrow \text{declclass } m = \text{Object}$

$\langle \text{proof} \rangle$

**lemma** *methd-declaredD*:

$\llbracket \text{wf-prog } G; \text{is-class } G \ C; \text{methd } G \ C \ \text{sig} = \text{Some } m \rrbracket$   
 $\Longrightarrow G \vdash (\text{mdecl}(\text{sig}, \text{methd } m)) \text{ declared-in } (\text{declclass } m)$

$\langle \text{proof} \rangle$

**lemma** *methd-rec-Some-cases* [*consumes 4*, *case-names NewMethod InheritedMethod*]:

**assumes** *methd-C*:  $\text{methd } G \ C \ \text{sig} = \text{Some } m$  **and**  
*ws*: *ws-prog*  $G$  **and**  
*clsC*:  $\text{class } G \ C = \text{Some } c$  **and**  
*neq-C-Obj*:  $C \neq \text{Object}$

**shows**

$\llbracket \text{table-of}(\text{map}(\lambda(s, m). (s, C, m))(\text{methods } c)) \ \text{sig} = \text{Some } m \rrbracket \Longrightarrow P;$   
 $\llbracket G \vdash C \text{ inherits } (\text{method } \text{sig } m); \text{methd } G \ (\text{super } c) \ \text{sig} = \text{Some } m \rrbracket \Longrightarrow P$   
 $\rrbracket \Longrightarrow P$

$\langle \text{proof} \rangle$

**lemma** *methd-member-of*:

**assumes** *wf*: *wf-prog*  $G$

**shows**

$\llbracket \text{is-class } G \ C; \text{methd } G \ C \ \text{sig} = \text{Some } m \rrbracket \Longrightarrow G \vdash \text{Methd } \text{sig } m \text{ member-of } C$   
 $(\text{is ?Class } C \Longrightarrow \text{?Method } C \Longrightarrow \text{?MemberOf } C)$

$\langle \text{proof} \rangle$

**lemma** *current-methd*:

$\llbracket \text{table-of}(\text{methods } c) \ \text{sig} = \text{Some } \text{new};$   
*ws-prog*  $G; \text{class } G \ C = \text{Some } c; C \neq \text{Object};$   
 $\text{methd } G \ (\text{super } c) \ \text{sig} = \text{Some } \text{old} \rrbracket$   
 $\Longrightarrow \text{methd } G \ C \ \text{sig} = \text{Some } (C, \text{new})$

$\langle \text{proof} \rangle$

**lemma** *wf-prog-staticD*:

**assumes** *wf*: wf-prog *G* **and**  
*clsC*: class *G* *C* = Some *c* **and**  
*neq-C-Obj*: *C*  $\neq$  Object **and**  
*old*: methd *G* (super *c*) sig = Some *old* **and**  
*accmodi-old*: Protected  $\leq$  accmodi *old* **and**  
*new*: table-of (methods *c*) sig = Some *new*  
**shows** is-static *new* = is-static *old*  
 ⟨proof⟩

**lemma** *inheritable-instance-methd*:

**assumes** *subclseq-C-D*:  $G \vdash C \preceq_C D$  **and**  
*is-cls-D*: is-class *G* *D* **and**  
*wf*: wf-prog *G* **and**  
*old*: methd *G* *D* sig = Some *old* **and**  
*accmodi-old*: Protected  $\leq$  accmodi *old* **and**  
*not-static-old*:  $\neg$  is-static *old*  
**shows**  
 $\exists$  *new*. methd *G* *C* sig = Some *new*  $\wedge$   
 (*new* = *old*  $\vee$  *G*,sig $\vdash$ *new* overrides<sub>*S*</sub> *old*)  
 (is ( $\exists$  *new*. (?Constraint *C* *new* *old*)))  
 ⟨proof⟩

**lemma** *inheritable-instance-methd-cases* [consumes 6

, case-names *Inheritance Overriding*]:

**assumes** *subclseq-C-D*:  $G \vdash C \preceq_C D$  **and**  
*is-cls-D*: is-class *G* *D* **and**  
*wf*: wf-prog *G* **and**  
*old*: methd *G* *D* sig = Some *old* **and**  
*accmodi-old*: Protected  $\leq$  accmodi *old* **and**  
*not-static-old*:  $\neg$  is-static *old* **and**  
*inheritance*: methd *G* *C* sig = Some *old*  $\implies$  *P* **and**  
*overriding*:  $\bigwedge$  *new*.  $\llbracket$ methd *G* *C* sig = Some *new*;  
*G*,sig $\vdash$ *new* overrides<sub>*S*</sub> *old* $\rrbracket \implies$  *P*  
**shows** *P*  
 ⟨proof⟩

**lemma** *inheritable-instance-methd-props*:

**assumes** *subclseq-C-D*:  $G \vdash C \preceq_C D$  **and**  
*is-cls-D*: is-class *G* *D* **and**  
*wf*: wf-prog *G* **and**  
*old*: methd *G* *D* sig = Some *old* **and**  
*accmodi-old*: Protected  $\leq$  accmodi *old* **and**  
*not-static-old*:  $\neg$  is-static *old*  
**shows**  
 $\exists$  *new*. methd *G* *C* sig = Some *new*  $\wedge$   
 $\neg$  is-static *new*  $\wedge$   $G \vdash \text{resTy } \text{new} \preceq \text{resTy } \text{old} \wedge \text{accmodi } \text{old} \leq \text{accmodi } \text{new}$   
 (is ( $\exists$  *new*. (?Constraint *C* *new* *old*)))  
 ⟨proof⟩

⟨ML⟩

**lemma** *subint-widen-imethds*:

$\llbracket G \vdash I \preceq I \ J; \text{wf-prog } G; \text{is-iface } G \ J; j m \in \text{imethds } G \ J \ \text{sig} \rrbracket \implies$

$\exists im \in imethds\ G\ I\ sig.\ is-static\ im = is-static\ jm \wedge$   
 $accmodi\ im = accmodi\ jm \wedge$   
 $G \vdash resTy\ im \preceq resTy\ jm$   
 <proof>

**lemma** *implmt1-methd*:

$\bigwedge sig.\ \llbracket G \vdash C \rightsquigarrow 1I; wf-prog\ G; im \in imethds\ G\ I\ sig \rrbracket \implies$   
 $\exists cm \in methd\ G\ C\ sig:\ \neg is-static\ cm \wedge \neg is-static\ im \wedge$   
 $G \vdash resTy\ cm \preceq resTy\ im \wedge$   
 $accmodi\ im = Public \wedge accmodi\ cm = Public$   
 <proof>

**lemma** *implmt-methd* [rule-format (no-asm)]:

$\llbracket wf-prog\ G; G \vdash C \rightsquigarrow I \rrbracket \implies is-iface\ G\ I \longrightarrow$   
 $(\forall im \in imethds\ G\ I\ sig.$   
 $\exists cm \in methd\ G\ C\ sig:\ \neg is-static\ cm \wedge \neg is-static\ im \wedge$   
 $G \vdash resTy\ cm \preceq resTy\ im \wedge$   
 $accmodi\ im = Public \wedge accmodi\ cm = Public)$   
 <proof>

**lemma** *mheadsD* [rule-format (no-asm)]:

$emh \in mheads\ G\ S\ t\ sig \longrightarrow wf-prog\ G \longrightarrow$   
 $(\exists C\ D\ m.\ t = ClassT\ C \wedge declrefT\ emh = ClassT\ D \wedge$   
 $accmethd\ G\ S\ C\ sig = Some\ m \wedge$   
 $(declclass\ m = D) \wedge mhead\ (mthd\ m) = (mhd\ emh)) \vee$   
 $(\exists I.\ t = IfaceT\ I \wedge ((\exists im.\ im \in accimethds\ G\ (pid\ S)\ I\ sig \wedge$   
 $mthd\ im = mhd\ emh) \vee$   
 $(\exists m.\ G \vdash Iface\ I\ accessible-in\ (pid\ S) \wedge accmethd\ G\ S\ Object\ sig = Some\ m \wedge$   
 $accmodi\ m \neq Private \wedge$   
 $declrefT\ emh = ClassT\ Object \wedge mhead\ (mthd\ m) = mhd\ emh))) \vee$   
 $(\exists T\ m.\ t = ArrayT\ T \wedge G \vdash Array\ T\ accessible-in\ (pid\ S) \wedge$   
 $accmethd\ G\ S\ Object\ sig = Some\ m \wedge accmodi\ m \neq Private \wedge$   
 $declrefT\ emh = ClassT\ Object \wedge mhead\ (mthd\ m) = mhd\ emh)$   
 <proof>

**lemma** *mheads-cases* [consumes 2, case-names Class-methd

*Iface-methd Iface-Object-methd Array-Object-methd*]:

$\llbracket emh \in mheads\ G\ S\ t\ sig; wf-prog\ G;$   
 $\bigwedge C\ D\ m.\ \llbracket t = ClassT\ C; declrefT\ emh = ClassT\ D; accmethd\ G\ S\ C\ sig = Some\ m;$   
 $(declclass\ m = D); mhead\ (mthd\ m) = (mhd\ emh) \rrbracket \implies P\ emh;$   
 $\bigwedge I\ im.\ \llbracket t = IfaceT\ I; im \in accimethds\ G\ (pid\ S)\ I\ sig; mthd\ im = mhd\ emh \rrbracket$   
 $\implies P\ emh;$   
 $\bigwedge I\ m.\ \llbracket t = IfaceT\ I; G \vdash Iface\ I\ accessible-in\ (pid\ S);$   
 $accmethd\ G\ S\ Object\ sig = Some\ m; accmodi\ m \neq Private;$   
 $declrefT\ emh = ClassT\ Object; mhead\ (mthd\ m) = mhd\ emh \rrbracket \implies P\ emh;$   
 $\bigwedge T\ m.\ \llbracket t = ArrayT\ T; G \vdash Array\ T\ accessible-in\ (pid\ S);$   
 $accmethd\ G\ S\ Object\ sig = Some\ m; accmodi\ m \neq Private;$   
 $declrefT\ emh = ClassT\ Object; mhead\ (mthd\ m) = mhd\ emh \rrbracket \implies P\ emh$

$\llbracket \Rightarrow P \text{ emh} \rrbracket$   
 $\langle \text{proof} \rangle$

**lemma** *declclassD[rule-format]*:  
 $\llbracket \text{wf-prog } G; \text{class } G \ C = \text{Some } c; \text{methd } G \ C \ \text{sig} = \text{Some } m; \text{class } G \ (\text{declclass } m) = \text{Some } d \rrbracket$   
 $\Rightarrow \text{table-of } (\text{methods } d) \ \text{sig} = \text{Some } (\text{methd } m)$   
 $\langle \text{proof} \rangle$

**lemma** *dynmethd-Object*:  
**assumes**  $\text{statM}: \text{methd } G \ \text{Object} \ \text{sig} = \text{Some } \text{statM}$  **and**  
 $\text{private}: \text{accmodi } \text{statM} = \text{Private}$  **and**  
 $\text{is-clc-}C: \text{is-class } G \ C$  **and**  
 $\text{wf}: \text{wf-prog } G$   
**shows**  $\text{dynmethd } G \ \text{Object } C \ \text{sig} = \text{Some } \text{statM}$   
 $\langle \text{proof} \rangle$

**lemma** *wf-imethds-hiding-objmethdsD*:  
**assumes**  $\text{old}: \text{methd } G \ \text{Object} \ \text{sig} = \text{Some } \text{old}$  **and**  
 $\text{is-if-}I: \text{is-iface } G \ I$  **and**  
 $\text{wf}: \text{wf-prog } G$  **and**  
 $\text{not-private}: \text{accmodi } \text{old} \neq \text{Private}$  **and**  
 $\text{new}: \text{new} \in \text{imethds } G \ I \ \text{sig}$   
**shows**  $G \vdash \text{resTy } \text{new} \leq \text{resTy } \text{old} \wedge \text{is-static } \text{new} = \text{is-static } \text{old} \ (\text{is } ?P \ \text{new})$   
 $\langle \text{proof} \rangle$

Which dynamic classes are valid to look up a member of a distinct static type? We have to distinct class members (named static members in Java) from instance members. Class members are global to all Objects of a class, instance members are local to a single Object instance. If a member is equipped with the static modifier it is a class member, else it is an instance member. The following table gives an overview of the current framework. We assume to have a reference with static type  $\text{statT}$  and a dynamic class  $\text{dynC}$ . Between both of these types the widening relation holds  $G \vdash \text{Class } \text{dynC} \leq \text{statT}$ . Unfortunately this ordinary widening relation isn't enough to describe the valid lookup classes, since we must cope the special cases of arrays and interfaces, too. If we statically expect an array or interface we may lookup a field or a method in Object which isn't covered in the widening relation.

statT	field	instance	method	static	(class)	method
NullT	/	/	/	Iface	/	dynC
Object	Class	dynC	dynC	dynC	dynC	Array
Object	Object					

In most cases we can lookup the member in the dynamic class. But as an interface can't declare new static methods, nor an array can define new methods at all, we have to lookup methods in the base class Object.

The limitation to classes in the field column is artificial and comes out of the typing rule for the field access (see rule *FVar* in the welltyping relation *wt* in theory WellType). It stems out of the fact, that Object indeed has no non private fields. So interfaces and arrays can actually have no fields at all and a field access would be senseless. (In Java interfaces are allowed to declare new fields but in current Bali not!). So there is no principal reason why we should not allow Objects to declare non private fields. Then we would get the following column:

statT	field	NullT	/	Iface	Object	Class	dynC	Array	Object

**consts** *valid-lookup-clc*:  $\text{prog} \Rightarrow \text{ref-ty} \Rightarrow \text{qname} \Rightarrow \text{bool} \Rightarrow \text{bool}$

( $\cdot, \vdash$  - *valid'-lookup'-cls'-for* - [61,61,61,61] 60)

**primrec**

$G, \text{Null}T \vdash \text{dyn}C \text{ valid-lookup-cls-for static-membr} = \text{False}$

$G, \text{Iface}T I \vdash \text{dyn}C \text{ valid-lookup-cls-for static-membr}$

$= (\text{if } \text{static-membr}$   
      $\text{then } \text{dyn}C = \text{Object}$   
      $\text{else } G \vdash \text{Class } \text{dyn}C \preceq \text{Iface } I)$

$G, \text{Class}T C \vdash \text{dyn}C \text{ valid-lookup-cls-for static-membr} = G \vdash \text{Class } \text{dyn}C \preceq \text{Class } C$

$G, \text{Array}T T \vdash \text{dyn}C \text{ valid-lookup-cls-for static-membr} = (\text{dyn}C = \text{Object})$

**lemma** *valid-lookup-cls-is-class*:

**assumes**  $\text{dyn}C$ :  $G, \text{stat}T \vdash \text{dyn}C \text{ valid-lookup-cls-for static-membr}$  **and**

$\text{ty-stat}T$ :  $\text{isrtype } G \text{ stat}T$  **and**

$\text{wf}$ :  $\text{wf-prog } G$

**shows**  $\text{is-class } G \text{ dyn}C$

$\langle \text{proof} \rangle$

**declare** *split-paired-All* [simp del] *split-paired-Ex* [simp del]

$\langle \text{ML} \rangle$

**lemma** *dynamic-mheadsD*:

$\llbracket \text{emh} \in \text{mheads } G \text{ S stat}T \text{ sig};$

$G, \text{stat}T \vdash \text{dyn}C \text{ valid-lookup-cls-for } (\text{is-static } \text{emh});$

$\text{isrtype } G \text{ stat}T; \text{wf-prog } G$

$\rrbracket \implies \exists m \in \text{dynlookup } G \text{ stat}T \text{ dyn}C \text{ sig}:$

$\text{is-static } m = \text{is-static } \text{emh} \wedge G \vdash \text{resTy } m \preceq \text{resTy } \text{emh}$

$\langle \text{proof} \rangle$

**declare** *split-paired-All* [simp] *split-paired-Ex* [simp]

$\langle \text{ML} \rangle$

**lemma** *methd-declclass*:

$\llbracket \text{class } G \text{ C} = \text{Some } c; \text{wf-prog } G; \text{methd } G \text{ C sig} = \text{Some } m \rrbracket$

$\implies \text{methd } G (\text{declclass } m) \text{ sig} = \text{Some } m$

$\langle \text{proof} \rangle$

**lemma** *dynmethd-declclass*:

$\llbracket \text{dynmethd } G \text{ stat}C \text{ dyn}C \text{ sig} = \text{Some } m;$

$\text{wf-prog } G; \text{is-class } G \text{ stat}C$

$\rrbracket \implies \text{methd } G (\text{declclass } m) \text{ sig} = \text{Some } m$

$\langle \text{proof} \rangle$

**lemma** *dynlookup-declC*:

$\llbracket \text{dynlookup } G \text{ stat}T \text{ dyn}C \text{ sig} = \text{Some } m; \text{wf-prog } G;$

$\text{is-class } G \text{ dyn}C; \text{isrtype } G \text{ stat}T$

$\rrbracket \implies G \vdash \text{dyn}C \preceq_C (\text{declclass } m) \wedge \text{is-class } G (\text{declclass } m)$

$\langle \text{proof} \rangle$

**lemma** *dynlookup-Array-declclassD* [simp]:

$\llbracket \text{dynlookup } G (\text{Array}T T) \text{ Object sig} = \text{Some } dm; \text{wf-prog } G \rrbracket$

$\implies \text{declclass } dm = \text{Object}$

$\langle \text{proof} \rangle$

**declare** *split-paired-All* [simp del] *split-paired-Ex* [simp del]  
 $\langle \text{ML} \rangle$

**lemma** *wt-is-type*:  $E, dt \models v :: T \implies \text{wf-prog } (\text{prg } E) \longrightarrow$   
 $dt = \text{empty-dt} \longrightarrow (\text{case } T \text{ of}$   
 $\quad \text{Inl } T \Rightarrow \text{is-type } (\text{prg } E) \ T$   
 $\quad | \text{Inr } Ts \Rightarrow \text{Ball } (\text{set } Ts) (\text{is-type } (\text{prg } E)))$

$\langle \text{proof} \rangle$

**declare** *split-paired-All* [simp] *split-paired-Ex* [simp]  
 $\langle \text{ML} \rangle$

**lemma** *ty-expr-is-type*:  
 $\llbracket E \vdash e :: - T; \text{wf-prog } (\text{prg } E) \rrbracket \implies \text{is-type } (\text{prg } E) \ T$   
 $\langle \text{proof} \rangle$

**lemma** *ty-var-is-type*:  
 $\llbracket E \vdash v :: T; \text{wf-prog } (\text{prg } E) \rrbracket \implies \text{is-type } (\text{prg } E) \ T$   
 $\langle \text{proof} \rangle$

**lemma** *ty-exprs-is-type*:  
 $\llbracket E \vdash es :: Ts; \text{wf-prog } (\text{prg } E) \rrbracket \implies \text{Ball } (\text{set } Ts) (\text{is-type } (\text{prg } E))$   
 $\langle \text{proof} \rangle$

**lemma** *static-mheadsD*:  
 $\llbracket emh \in \text{mheads } G \ S \ t \ \text{sig}; \text{wf-prog } G; E \vdash e :: - \text{RefT } t; \text{prg } E = G ;$   
 $\quad \text{invmode } (\text{mhd } emh) \ e \neq \text{IntVir}$   
 $\rrbracket \implies \exists m. ( (\exists C. t = \text{ClassT } C \wedge \text{accmethd } G \ S \ C \ \text{sig} = \text{Some } m)$   
 $\quad \vee (\forall C. t \neq \text{ClassT } C \wedge \text{accmethd } G \ S \ \text{Object} \ \text{sig} = \text{Some } m) ) \wedge$   
 $\quad \text{declrefT } emh = \text{ClassT } (\text{declclass } m) \wedge \text{mhead } (\text{mthd } m) = (\text{mhd } emh)$   
 $\langle \text{proof} \rangle$

**lemma** *wt-MethdI*:  
 $\llbracket \text{methd } G \ C \ \text{sig} = \text{Some } m; \text{wf-prog } G;$   
 $\quad \text{class } G \ C = \text{Some } c \rrbracket \implies$   
 $\exists T. (\llbracket \text{prg} = G, \text{cls} = (\text{declclass } m),$   
 $\quad \text{lcl} = \text{callee-lcl } (\text{declclass } m) \ \text{sig } (\text{mthd } m) \rrbracket \vdash \text{Methd } C \ \text{sig} :: - T \wedge G \vdash T \preceq_{\text{resTy}} m$   
 $\langle \text{proof} \rangle$

### 35 accessibility concerns

**lemma** *mheads-type-accessible*:  
 $\llbracket emh \in \text{mheads } G \ S \ T \ \text{sig}; \text{wf-prog } G \rrbracket$   
 $\implies G \vdash \text{RefT } T \text{ accessible-in } (\text{pid } S)$   
 $\langle \text{proof} \rangle$

**lemma** *static-to-dynamic-accessible-from-aux*:  
 $\llbracket G \vdash m \text{ of } C \text{ accessible-from } \text{accC}; \text{wf-prog } G \rrbracket$   
 $\implies G \vdash m \text{ in } C \text{ dyn-accessible-from } \text{accC}$   
 $\langle \text{proof} \rangle$

**lemma** *static-to-dynamic-accessible-from*:  
**assumes** *stat-acc*:  $G \vdash m$  of *statC* accessible-from *accC* **and**  
*subclseq*:  $G \vdash \text{dynC} \preceq_C \text{statC}$  **and**  
*wf*: *wf-prog* *G*  
**shows**  $G \vdash m$  in *dynC* dyn-accessible-from *accC*  
 ⟨proof⟩

**lemma** *static-to-dynamic-accessible-from-static*:  
**assumes** *stat-acc*:  $G \vdash m$  of *statC* accessible-from *accC* **and**  
*static*: *is-static* *m* **and**  
*wf*: *wf-prog* *G*  
**shows**  $G \vdash m$  in (*declclass* *m*) dyn-accessible-from *accC*  
 ⟨proof⟩

**lemma** *dynmethd-member-in*:  
**assumes** *m*: *dynmethd* *G* *statC* *dynC* *sig* = *Some m* **and**  
*iscls-statC*: *is-class* *G* *statC* **and**  
*wf*: *wf-prog* *G*  
**shows**  $G \vdash \text{Methd } \text{sig } m$  member-in *dynC*  
 ⟨proof⟩

**lemma** *dynmethd-access-prop*:  
**assumes** *statM*: *methd* *G* *statC* *sig* = *Some statM* **and**  
*stat-acc*:  $G \vdash \text{Methd } \text{sig } \text{statM}$  of *statC* accessible-from *accC* **and**  
*dynM*: *dynmethd* *G* *statC* *dynC* *sig* = *Some dynM* **and**  
*wf*: *wf-prog* *G*  
**shows**  $G \vdash \text{Methd } \text{sig } \text{dynM}$  in *dynC* dyn-accessible-from *accC*  
 ⟨proof⟩

**lemma** *implmt-methd-access*:  
**fixes** *accC*::*qtname*  
**assumes** *iface-methd*: *imethds* *G* *I* *sig*  $\neq \{\}$  **and**  
*implements*:  $G \vdash \text{dynC} \rightsquigarrow I$  **and**  
*isif-I*: *is-iface* *G* *I* **and**  
*wf*: *wf-prog* *G*  
**shows**  $\exists \text{ dynM. } \text{methd } G \text{ dynC } \text{sig} = \text{Some dynM} \wedge$   
 $G \vdash \text{Methd } \text{sig } \text{dynM}$  in *dynC* dyn-accessible-from *accC*  
 ⟨proof⟩

**corollary** *implmt-dynimethd-access*:  
**fixes** *accC*::*qtname*  
**assumes** *iface-methd*: *imethds* *G* *I* *sig*  $\neq \{\}$  **and**  
*implements*:  $G \vdash \text{dynC} \rightsquigarrow I$  **and**  
*isif-I*: *is-iface* *G* *I* **and**  
*wf*: *wf-prog* *G*  
**shows**  $\exists \text{ dynM. } \text{dynimethd } G \text{ I } \text{ dynC } \text{sig} = \text{Some dynM} \wedge$   
 $G \vdash \text{Methd } \text{sig } \text{dynM}$  in *dynC* dyn-accessible-from *accC*  
 ⟨proof⟩

**lemma** *dynlookup-access-prop*:  
**assumes** *emh*: *emh*  $\in$  *mheads* *G* *accC* *statT* *sig* **and**  
*dynM*: *dynlookup* *G* *statT* *dynC* *sig* = *Some dynM* **and**  
*dynC-prop*:  $G, \text{statT} \vdash \text{dynC}$  valid-lookup-cls-for *is-static* *emh* **and**  
*isT-statT*: *isrtype* *G* *statT* **and**



*wf*: *wf-prog G*  
**shows**  $G \vdash \text{Methd sig dynM in dynC dyn-accessible-from accC}$   
 ⟨proof⟩

**lemma** *dynlookup-access*:

**assumes** *emh*:  $\text{emh} \in \text{mheads } G \text{ accC statT sig}$  **and**  
*dynC-prop*:  $G, \text{statT} \vdash \text{dynC valid-lookup-cls-for (is-static emh)}$  **and**  
*isT-statT*:  $\text{isrtype } G \text{ statT}$  **and**  
*wf*: *wf-prog G*  
**shows**  $\exists \text{ dynM. dynlookup } G \text{ statT dynC sig} = \text{Some dynM} \wedge$   
 $G \vdash \text{Methd sig dynM in dynC dyn-accessible-from accC}$   
 ⟨proof⟩

**lemma** *stat-overrides-Package-old*:

**assumes** *stat-override*:  $G \vdash \text{new overrides}_S \text{ old}$  **and**  
*accmodi-new*:  $\text{accmodi new} = \text{Package}$  **and**  
*wf*: *wf-prog G*  
**shows**  $\text{accmodi old} = \text{Package}$   
 ⟨proof⟩

## Properties of dynamic accessibility

**lemma** *dyn-accessible-Private*:

**assumes** *dyn-acc*:  $G \vdash m \text{ in } C \text{ dyn-accessible-from accC}$  **and**  
*priv*:  $\text{accmodi } m = \text{Private}$   
**shows**  $\text{accC} = \text{declclass } m$   
 ⟨proof⟩

*dyn-accessible-Package* only works with the *wf-prog* assumption. Without it. it is easy to leaf the Package!

**lemma** *dyn-accessible-Package*:

$\llbracket G \vdash m \text{ in } C \text{ dyn-accessible-from accC}; \text{accmodi } m = \text{Package};$   
*wf-prog G* $\rrbracket$   
 $\implies \text{pid accC} = \text{pid (declclass } m)$   
 ⟨proof⟩

For fields we don't need the wellformedness of the program, since there is no overriding

**lemma** *dyn-accessible-field-Package*:

**assumes** *dyn-acc*:  $G \vdash f \text{ in } C \text{ dyn-accessible-from accC}$  **and**  
*pack*:  $\text{accmodi } f = \text{Package}$  **and**  
*field*:  $\text{is-field } f$   
**shows**  $\text{pid accC} = \text{pid (declclass } f)$   
 ⟨proof⟩

*dyn-accessible-instance-field-Protected* only works for fields since methods can break the package bounds due to overriding

**lemma** *dyn-accessible-instance-field-Protected*:

**assumes** *dyn-acc*:  $G \vdash f \text{ in } C \text{ dyn-accessible-from accC}$  **and**  
*prot*:  $\text{accmodi } f = \text{Protected}$  **and**  
*field*:  $\text{is-field } f$  **and**  
*instance-field*:  $\neg \text{is-static } f$  **and**  
*outside*:  $\text{pid (declclass } f) \neq \text{pid accC}$   
**shows**  $G \vdash C \preceq_C \text{ accC}$   
 ⟨proof⟩

**lemma** *dyn-accessible-static-field-Protected*:  
**assumes** *dyn-acc*:  $G \vdash f \text{ in } C \text{ dyn-accessible-from } accC$  **and**  
 $prot: accmod\ f = Protected$  **and**  
 $field: is-field\ f$  **and**  
 $static-field: is-static\ f$  **and**  
 $outside: pid\ (declclass\ f) \neq pid\ accC$   
**shows**  $G \vdash accC \preceq_C declclass\ f \wedge G \vdash C \preceq_C declclass\ f$   
 $\langle proof \rangle$   
**end**

## Chapter 14

# State

### 36 State for evaluation of Java expressions and statements

**theory** *State* **imports** *DeclConcepts* **begin**

design issues:

- all kinds of objects (class instances, arrays, and class objects) are handled via a general object abstraction
- the heap and the map for class objects are combined into a single table (*recall* (*loc*, *obj*) *table*  $\times$  (*qname*, *obj*) *table*  $\sim =$  (*loc* + *qname*, *obj*) *table*)

#### objects

**datatype** *obj-tag* = — tag for generic object  
           *CInst qname* — class instance  
           | *Arr ty int* — array with component type and length  
           — — CStat *qname* the tag is irrelevant for a class object, i.e. the static fields of a class, since its type is given already by the reference to it (see below)

**types** *vn* = *fspec* + *int* — variable name  
**record** *obj* =  
           *tag* :: *obj-tag* — generalized object  
           *values* :: (*vn*, *val*) *table*

#### translations

*fspec* <= (*type*) *vname*  $\times$  *qname*  
*vn* <= (*type*) *fspec* + *int*  
*obj* <= (*type*) (*tag*::*obj-tag*, *values*::*vn*  $\Rightarrow$  *val option*)  
*obj* <= (*type*) (*tag*::*obj-tag*, *values*::*vn*  $\Rightarrow$  *val option*,...::'*a*)

#### constdefs

*the-Arr* :: *obj option*  $\Rightarrow$  *ty*  $\times$  *int*  $\times$  (*vn*, *val*) *table*  
*the-Arr obj*  $\equiv$  *SOME* (*T,k,t*). *obj* = *Some* (*tag=Arr T k,values=t*)

**lemma** *the-Arr-Arr* [*simp*]: *the-Arr* (*Some* (*tag=Arr T k,values=cs*)) = (*T,k,cs*)  
 <proof>

**lemma** *the-Arr-Arr1* [*simp,intro,dest*]:  
 $\llbracket \text{tag } obj = \text{Arr } T \ k \rrbracket \Longrightarrow \text{the-Arr } (\text{Some } obj) = (T,k,\text{values } obj)$   
 <proof>

#### constdefs

*upd-obj* :: *vn*  $\Rightarrow$  *val*  $\Rightarrow$  *obj*  $\Rightarrow$  *obj*  
*upd-obj n v*  $\equiv$   $\lambda$  *obj* . *obj* (*values:=*(*values obj*)(*n* $\mapsto$ *v*))

**lemma** *upd-obj-def2* [*simp*]:  
*upd-obj n v obj* = *obj* (*values:=*(*values obj*)(*n* $\mapsto$ *v*))  
 <proof>

#### constdefs

*obj-ty* :: *obj*  $\Rightarrow$  *ty*  
*obj-ty obj*  $\equiv$  *case tag obj of*

$$\begin{array}{l} CInst\ C \Rightarrow Class\ C \\ | Arr\ T\ k \Rightarrow T.[] \end{array}$$

**lemma** *obj-ty-eq* [intro!]: *obj-ty* ( $\langle tag=oi, values=x \rangle$ ) = *obj-ty* ( $\langle tag=oi, values=y \rangle$ )  
 ⟨proof⟩

**lemma** *obj-ty-eq1* [intro!,dest]:  
 $tag\ obj = tag\ obj' \implies obj\text{-}ty\ obj = obj\text{-}ty\ obj'$   
 ⟨proof⟩

**lemma** *obj-ty-cong* [simp]:  
 $obj\text{-}ty\ (obj\ \langle values:=vs \rangle) = obj\text{-}ty\ obj$   
 ⟨proof⟩

**lemma** *obj-ty-CInst* [simp]:  
 $obj\text{-}ty\ \langle tag=CInst\ C, values=vs \rangle = Class\ C$   
 ⟨proof⟩

**lemma** *obj-ty-CInst1* [simp,intro!,dest]:  
 $\llbracket tag\ obj = CInst\ C \rrbracket \implies obj\text{-}ty\ obj = Class\ C$   
 ⟨proof⟩

**lemma** *obj-ty-Arr* [simp]:  
 $obj\text{-}ty\ \langle tag=Arr\ T\ i, values=vs \rangle = T.[]$   
 ⟨proof⟩

**lemma** *obj-ty-Arr1* [simp,intro!,dest]:  
 $\llbracket tag\ obj = Arr\ T\ i \rrbracket \implies obj\text{-}ty\ obj = T.[]$   
 ⟨proof⟩

**lemma** *obj-ty-widenD*:  
 $G \vdash obj\text{-}ty\ obj \preceq RefT\ t \implies (\exists C. tag\ obj = CInst\ C) \vee (\exists T\ k. tag\ obj = Arr\ T\ k)$   
 ⟨proof⟩

**constdefs**

$$\begin{array}{l} obj\text{-}class :: obj \Rightarrow qname \\ obj\text{-}class\ obj \equiv case\ tag\ obj\ of \\ \quad CInst\ C \Rightarrow C \\ \quad | Arr\ T\ k \Rightarrow Object \end{array}$$

**lemma** *obj-class-CInst* [simp]: *obj-class* ( $\langle tag=CInst\ C, values=vs \rangle$ ) = *C*  
 ⟨proof⟩

**lemma** *obj-class-CInst1* [simp,intro!,dest]:  
 $tag\ obj = CInst\ C \implies obj\text{-}class\ obj = C$   
 ⟨proof⟩

**lemma** *obj-class-Arr* [simp]: *obj-class* ( $\langle \text{tag} = \text{Arr } T \ k, \text{values} = \text{vs} \rangle$ ) = *Object*  
 ⟨proof⟩

**lemma** *obj-class-Arr1* [simp,intro!,dest]:  
 $\text{tag } \text{obj} = \text{Arr } T \ k \implies \text{obj-class } \text{obj} = \text{Object}$   
 ⟨proof⟩

**lemma** *obj-ty-obj-class*:  $G \vdash \text{obj-ty } \text{obj} \preceq \text{Class } \text{statC} = G \vdash \text{obj-class } \text{obj} \preceq_C \text{statC}$   
 ⟨proof⟩

## object references

**types** *oref* = *loc* + *qname* — generalized object reference

### syntax

*Heap* :: *loc*  $\Rightarrow$  *oref*  
*Stat* :: *qname*  $\Rightarrow$  *oref*

### translations

*Heap*  $\Rightarrow$  *Inl*  
*Stat*  $\Rightarrow$  *Inr*  
*oref*  $\leq$  (*type*) *loc* + *qname*

### constdefs

*fields-table*::  
 $\text{prog} \Rightarrow \text{qname} \Rightarrow (\text{fspec} \Rightarrow \text{field} \Rightarrow \text{bool}) \Rightarrow (\text{fspec}, \text{ty}) \text{ table}$   
*fields-table* *G C P*  
 $\equiv \text{option-map } \text{type} \circ \text{table-of } (\text{filter } (\text{split } P) (\text{DeclConcepts.fields } G \ C))$

**lemma** *fields-table-SomeI*:  
 $\llbracket \text{table-of } (\text{DeclConcepts.fields } G \ C) \ n = \text{Some } f; P \ n \ f \rrbracket$   
 $\implies \text{fields-table } G \ C \ P \ n = \text{Some } (\text{type } f)$   
 ⟨proof⟩

**lemma** *fields-table-SomeD'*: *fields-table* *G C P fn* = *Some T*  $\implies$   
 $\exists f. (fn, f) \in \text{set}(\text{DeclConcepts.fields } G \ C) \wedge \text{type } f = T$   
 ⟨proof⟩

**lemma** *fields-table-SomeD*:  
 $\llbracket \text{fields-table } G \ C \ P \ fn = \text{Some } T; \text{unique } (\text{DeclConcepts.fields } G \ C) \rrbracket \implies$   
 $\exists f. \text{table-of } (\text{DeclConcepts.fields } G \ C) \ fn = \text{Some } f \wedge \text{type } f = T$   
 ⟨proof⟩

### constdefs

*in-bounds* :: *int*  $\Rightarrow$  *int*  $\Rightarrow$  *bool* (( $\neg$  *in'-bounds* -) [50, 51] 50)  
 $i \text{ in-bounds } k \equiv 0 \leq i \wedge i < k$

*arr-comps* :: '*a*  $\Rightarrow$  *int*  $\Rightarrow$  *int*  $\Rightarrow$  '*a* option  
 $\text{arr-comps } T \ k \equiv \lambda i. \text{if } i \text{ in-bounds } k \text{ then } \text{Some } T \text{ else } \text{None}$

*var-tys* :: *prog*  $\Rightarrow$  *obj-tag*  $\Rightarrow$  *oref*  $\Rightarrow$  (*vn*, *ty*) *table*  
*var-tys* *G oi r*

$$\begin{aligned}
&\equiv \text{case } r \text{ of} \\
&\quad \text{Heap } a \Rightarrow (\text{case } oi \text{ of} \\
&\quad \quad CInst \ C \Rightarrow \text{fields-table } G \ C \ (\lambda n \ f. \neg \text{static } f) \ (+) \ \text{empty} \\
&\quad \quad | \text{Arr } T \ k \Rightarrow \text{empty } (+) \ \text{arr-comps } T \ k) \\
&\quad | \text{Stat } C \Rightarrow \text{fields-table } G \ C \ (\lambda fn \ f. \text{declclassf } fn = C \wedge \text{static } f) \\
&\quad \quad (+) \ \text{empty}
\end{aligned}$$

**lemma** *var-tys-Some-eq*:

*var-tys*  $G \ oi \ r \ n = \text{Some } T$

$$\begin{aligned}
&= (\text{case } r \text{ of} \\
&\quad Inl \ a \Rightarrow (\text{case } oi \text{ of} \\
&\quad \quad CInst \ C \Rightarrow (\exists nt. n = Inl \ nt \wedge \text{fields-table } G \ C \ (\lambda n \ f. \\
&\quad \quad \quad \neg \text{static } f) \ nt = \text{Some } T)) \\
&\quad \quad | \text{Arr } t \ k \Rightarrow (\exists i. n = Inr \ i \wedge i \text{ in-bounds } k \wedge t = T)) \\
&\quad | Inr \ C \Rightarrow (\exists nt. n = Inl \ nt \wedge \\
&\quad \quad \text{fields-table } G \ C \ (\lambda fn \ f. \text{declclassf } fn = C \wedge \text{static } f) \ nt \\
&\quad \quad = \text{Some } T))
\end{aligned}$$

$\langle \text{proof} \rangle$

## stores

**types** *globs* — global variables: heap and static variables

$$\begin{aligned}
&= (\text{oref} \ , \ \text{obj}) \ \text{table} \\
&\text{heap} \\
&= (\text{loc} \ , \ \text{obj}) \ \text{table}
\end{aligned}$$

## translations

$$\begin{aligned}
&\text{globs} \leq (\text{type}) \ (\text{oref} \ , \ \text{obj}) \ \text{table} \\
&\text{heap} \leq (\text{type}) \ (\text{loc} \ , \ \text{obj}) \ \text{table}
\end{aligned}$$

**datatype** *st* =

*st* *globs* *locals*

## 37 access

### constdefs

$$\begin{aligned}
&\text{globs} :: st \Rightarrow \text{globs} \\
&\text{globs} \equiv \text{st-case } (\lambda g \ l. \ g)
\end{aligned}$$

$$\begin{aligned}
&\text{locals} :: st \Rightarrow \text{locals} \\
&\text{locals} \equiv \text{st-case } (\lambda g \ l. \ l)
\end{aligned}$$

$$\begin{aligned}
&\text{heap} :: st \Rightarrow \text{heap} \\
&\text{heap } s \equiv \text{globs } s \circ \text{Heap}
\end{aligned}$$

**lemma** *globs-def2 [simp]*:  $\text{globs } (st \ g \ l) = g$

$\langle \text{proof} \rangle$

**lemma** *locals-def2 [simp]*:  $\text{locals } (st \ g \ l) = l$

$\langle \text{proof} \rangle$

**lemma** *heap-def2 [simp]: heap s a = globs s (Heap a)*  
 ⟨proof⟩

#### **syntax**

*val-this*     :: *st*  $\Rightarrow$  *val*  
*lookup-obj*   :: *st*  $\Rightarrow$  *val*  $\Rightarrow$  *obj*

#### **translations**

*val-this s*        == *the (locals s This)*  
*lookup-obj s a'* == *the (heap s (the-Addr a'))*

### **38 memory allocation**

#### **constdefs**

*new-Addr*     :: *heap*  $\Rightarrow$  *loc option*  
*new-Addr h*   ≡ *if (∀ a. h a ≠ None) then None else Some (SOME a. h a = None)*

**lemma** *new-AddrD: new-Addr h = Some a  $\implies$  h a = None*  
 ⟨proof⟩

**lemma** *new-AddrD2: new-Addr h = Some a  $\implies$  ∀ b. h b ≠ None  $\longrightarrow$  b ≠ a*  
 ⟨proof⟩

**lemma** *new-Addr-SomeI: h a = None  $\implies$  ∃ b. new-Addr h = Some b ∧ h b = None*  
 ⟨proof⟩

### **39 initialization**

#### **syntax**

*init-vals*     :: (*a*, *ty*) *table*  $\Rightarrow$  (*a*, *val*) *table*

#### **translations**

*init-vals vs*    == *option-map default-val*  $\circ$  *vs*

**lemma** *init-arr-comps-base [simp]: init-vals (arr-comps T 0) = empty*  
 ⟨proof⟩

**lemma** *init-arr-comps-step [simp]:*  
*0 < j  $\implies$  init-vals (arr-comps T j) =*  
*init-vals (arr-comps T (j - 1))(j - 1  $\mapsto$  default-val T)*  
 ⟨proof⟩

### **40 update**

#### **constdefs**

*gupd*        :: *oref*  $\Rightarrow$  *obj*  $\Rightarrow$  *st*  $\Rightarrow$  *st*        (*gupd'(- $\mapsto$ -')[10,10]1000*)  
*gupd r obj* ≡ *st-case (λg l. st (g(r $\mapsto$ obj)) l)*

*lupd*        :: *lname*  $\Rightarrow$  *val*  $\Rightarrow$  *st*  $\Rightarrow$  *st*        (*lupd'(- $\mapsto$ -')[10,10]1000*)  
*lupd vn v*   ≡ *st-case (λg l. st g (l(vn $\mapsto$ v)))*

*upd-gobj*    :: *oref*  $\Rightarrow$  *vn*  $\Rightarrow$  *val*  $\Rightarrow$  *st*  $\Rightarrow$  *st*



$upd-gobj\ r\ n\ v \equiv st-case\ (\lambda g\ l.\ st\ (chg-map\ (upd-obj\ n\ v)\ r\ g)\ l)$

$set-locals\ ::\ locals \Rightarrow st \Rightarrow st$   
 $set-locals\ l \equiv st-case\ (\lambda g\ l'.\ st\ g\ l)$

$init-obj\ ::\ prog \Rightarrow obj-tag \Rightarrow oref \Rightarrow st \Rightarrow st$   
 $init-obj\ G\ oi\ r \equiv gupd(r \mapsto \langle tag=oi, values=init-vals\ (var-tys\ G\ oi\ r) \rangle)$

### **syntax**

$init-class-obj\ ::\ prog \Rightarrow qtname \Rightarrow st \Rightarrow st$

### **translations**

$init-class-obj\ G\ C == init-obj\ G\ arbitrary\ (Inr\ C)$

**lemma**  $gupd-def2\ [simp]:\ gupd(r \mapsto obj)\ (st\ g\ l) = st\ (g(r \mapsto obj))\ l$   
 $\langle proof \rangle$

**lemma**  $lupd-def2\ [simp]:\ lupd(vn \mapsto v)\ (st\ g\ l) = st\ g\ (l(vn \mapsto v))$   
 $\langle proof \rangle$

**lemma**  $globs-gupd\ [simp]:\ globs\ (gupd(r \mapsto obj)\ s) = globs\ s(r \mapsto obj)$   
 $\langle proof \rangle$

**lemma**  $globs-lupd\ [simp]:\ globs\ (lupd(vn \mapsto v)\ s) = globs\ s$   
 $\langle proof \rangle$

**lemma**  $locals-gupd\ [simp]:\ locals\ (gupd(r \mapsto obj)\ s) = locals\ s$   
 $\langle proof \rangle$

**lemma**  $locals-lupd\ [simp]:\ locals\ (lupd(vn \mapsto v)\ s) = locals\ s(vn \mapsto v)$   
 $\langle proof \rangle$

**lemma**  $globs-upd-gobj-new\ [rule-format\ (no-asm),\ simp]:$   
 $globs\ s\ r = None \longrightarrow globs\ (upd-gobj\ r\ n\ v\ s) = globs\ s$   
 $\langle proof \rangle$

**lemma**  $globs-upd-gobj-upd\ [rule-format\ (no-asm),\ simp]:$   
 $globs\ s\ r = Some\ obj \longrightarrow globs\ (upd-gobj\ r\ n\ v\ s) = globs\ s(r \mapsto upd-obj\ n\ v\ obj)$   
 $\langle proof \rangle$

**lemma**  $locals-upd-gobj\ [simp]:\ locals\ (upd-gobj\ r\ n\ v\ s) = locals\ s$   
 $\langle proof \rangle$

**lemma**  $globs-init-obj\ [simp]:\ globs\ (init-obj\ G\ oi\ r\ s)\ t =$   
 $(if\ t=r\ then\ Some\ \langle tag=oi, values=init-vals\ (var-tys\ G\ oi\ r) \rangle\ else\ globs\ s\ t)$   
 $\langle proof \rangle$

**lemma** *locals-init-obj* [simp]:  $\text{locals } (\text{init-obj } G \text{ oi } r \ s) = \text{locals } s$   
 ⟨proof⟩

**lemma** *surjective-st* [simp]:  $\text{st } (\text{globs } s) (\text{locals } s) = s$   
 ⟨proof⟩

**lemma** *surjective-st-init-obj*:  
 $\text{st } (\text{globs } (\text{init-obj } G \text{ oi } r \ s)) (\text{locals } s) = \text{init-obj } G \text{ oi } r \ s$   
 ⟨proof⟩

**lemma** *heap-heap-upd* [simp]:  
 $\text{heap } (\text{st } (g(\text{Inl } a \mapsto \text{obj}))) \ l = \text{heap } (\text{st } g \ l)(a \mapsto \text{obj})$   
 ⟨proof⟩

**lemma** *heap-stat-upd* [simp]:  $\text{heap } (\text{st } (g(\text{Inr } C \mapsto \text{obj}))) \ l = \text{heap } (\text{st } g \ l)$   
 ⟨proof⟩

**lemma** *heap-local-upd* [simp]:  $\text{heap } (\text{st } g \ (l(vn \mapsto v))) = \text{heap } (\text{st } g \ l)$   
 ⟨proof⟩

**lemma** *heap-gupd-Heap* [simp]:  $\text{heap } (\text{gupd}(\text{Heap } a \mapsto \text{obj}) \ s) = \text{heap } s(a \mapsto \text{obj})$   
 ⟨proof⟩

**lemma** *heap-gupd-Stat* [simp]:  $\text{heap } (\text{gupd}(\text{Stat } C \mapsto \text{obj}) \ s) = \text{heap } s$   
 ⟨proof⟩

**lemma** *heap-lupd* [simp]:  $\text{heap } (\text{lupd}(vn \mapsto v) \ s) = \text{heap } s$   
 ⟨proof⟩

**lemma** *heap-upd-gobj-Stat* [simp]:  $\text{heap } (\text{upd-gobj } (\text{Stat } C) \ n \ v \ s) = \text{heap } s$   
 ⟨proof⟩

**lemma** *set-locals-def2* [simp]:  $\text{set-locals } l \ (\text{st } g \ l') = \text{st } g \ l$   
 ⟨proof⟩

**lemma** *set-locals-id* [simp]:  $\text{set-locals } (\text{locals } s) \ s = s$   
 ⟨proof⟩

**lemma** *set-set-locals* [simp]:  $\text{set-locals } l \ (\text{set-locals } l' \ s) = \text{set-locals } l \ s$   
 ⟨proof⟩

**lemma** *locals-set-locals* [simp]:  $\text{locals } (\text{set-locals } l \ s) = l$   
 ⟨proof⟩

**lemma** *globs-set-locals* [simp]:  $\text{globs } (\text{set-locals } l \ s) = \text{globs } s$   
 ⟨proof⟩

**lemma** *heap-set-locals* [simp]:  $\text{heap } (\text{set-locals } l \ s) = \text{heap } s$

$\langle proof \rangle$

## abrupt completion

### consts

$the-Xcpt :: abrupt \Rightarrow xcpt$   
 $the-Jump :: abrupt \Rightarrow jump$   
 $the-Loc :: xcpt \Rightarrow loc$   
 $the-Std :: xcpt \Rightarrow xname$

**primrec**  $the-Xcpt (Xcpt x) = x$   
**primrec**  $the-Jump (Jump j) = j$   
**primrec**  $the-Loc (Loc a) = a$   
**primrec**  $the-Std (Std x) = x$

### constdefs

$abrupt-if :: bool \Rightarrow abopt \Rightarrow abopt \Rightarrow abopt$   
 $abrupt-if\ c\ x'\ x \equiv if\ c \wedge (x = None)\ then\ x'\ else\ x$

**lemma**  $abrupt-if-True-None [simp]: abrupt-if\ True\ x\ None = x$   
 $\langle proof \rangle$

**lemma**  $abrupt-if-True-not-None [simp]: x \neq None \implies abrupt-if\ True\ x\ y \neq None$   
 $\langle proof \rangle$

**lemma**  $abrupt-if-False [simp]: abrupt-if\ False\ x\ y = y$   
 $\langle proof \rangle$

**lemma**  $abrupt-if-Some [simp]: abrupt-if\ c\ x\ (Some\ y) = Some\ y$   
 $\langle proof \rangle$

**lemma**  $abrupt-if-not-None [simp]: y \neq None \implies abrupt-if\ c\ x\ y = y$   
 $\langle proof \rangle$

**lemma**  $split-abrupt-if:$   
 $P\ (abrupt-if\ c\ x'\ x) =$   
 $((c \wedge x = None \longrightarrow P\ x') \wedge (\neg (c \wedge x = None) \longrightarrow P\ x))$   
 $\langle proof \rangle$

### syntax

$raise-if :: bool \Rightarrow xname \Rightarrow abopt \Rightarrow abopt$   
 $np :: val \Rightarrow abopt \Rightarrow abopt$   
 $check-neg :: val \Rightarrow abopt \Rightarrow abopt$   
 $error-if :: bool \Rightarrow error \Rightarrow abopt \Rightarrow abopt$

### translations

$\text{raise-if } c \text{ } xn == \text{abrupt-if } c \text{ } (\text{Some } (Xcpt \text{ } (Std \text{ } xn)))$   
 $\text{np } v == \text{raise-if } (v = \text{Null}) \text{ } \text{NullPointer}$   
 $\text{check-neg } i' == \text{raise-if } (\text{the-Intg } i' < 0) \text{ } \text{NegArrSize}$   
 $\text{error-if } c \text{ } e == \text{abrupt-if } c \text{ } (\text{Some } (\text{Error } e))$

**lemma** *raise-if-None* [simp]:  $(\text{raise-if } c \text{ } x \text{ } y = \text{None}) = (\neg c \wedge y = \text{None})$   
 $\langle \text{proof} \rangle$   
**declare** *raise-if-None* [THEN iffD1, dest!]

**lemma** *if-raise-if-None* [simp]:  
 $((\text{if } b \text{ then } y \text{ else } \text{raise-if } c \text{ } x \text{ } y) = \text{None}) = ((c \longrightarrow b) \wedge y = \text{None})$   
 $\langle \text{proof} \rangle$

**lemma** *raise-if-SomeD* [dest!]:  
 $\text{raise-if } c \text{ } x \text{ } y = \text{Some } z \implies c \wedge z = (Xcpt \text{ } (Std \text{ } x)) \wedge y = \text{None} \vee (y = \text{Some } z)$   
 $\langle \text{proof} \rangle$

**lemma** *error-if-None* [simp]:  $(\text{error-if } c \text{ } e \text{ } y = \text{None}) = (\neg c \wedge y = \text{None})$   
 $\langle \text{proof} \rangle$   
**declare** *error-if-None* [THEN iffD1, dest!]

**lemma** *if-error-if-None* [simp]:  
 $((\text{if } b \text{ then } y \text{ else } \text{error-if } c \text{ } e \text{ } y) = \text{None}) = ((c \longrightarrow b) \wedge y = \text{None})$   
 $\langle \text{proof} \rangle$

**lemma** *error-if-SomeD* [dest!]:  
 $\text{error-if } c \text{ } e \text{ } y = \text{Some } z \implies c \wedge z = (\text{Error } e) \wedge y = \text{None} \vee (y = \text{Some } z)$   
 $\langle \text{proof} \rangle$

**constdefs**  
 $\text{absorb} :: \text{jump} \Rightarrow \text{abopt} \Rightarrow \text{abopt}$   
 $\text{absorb } j \text{ } a \equiv \text{if } a = \text{Some } (\text{Jump } j) \text{ then } \text{None} \text{ else } a$

**lemma** *absorb-SomeD* [dest!]:  $\text{absorb } j \text{ } a = \text{Some } x \implies a = \text{Some } x$   
 $\langle \text{proof} \rangle$

**lemma** *absorb-same* [simp]:  $\text{absorb } j \text{ } (\text{Some } (\text{Jump } j)) = \text{None}$   
 $\langle \text{proof} \rangle$

**lemma** *absorb-other* [simp]:  $a \neq \text{Some } (\text{Jump } j) \implies \text{absorb } j \text{ } a = a$   
 $\langle \text{proof} \rangle$

**lemma** *absorb-Some-NoneD*:  $\text{absorb } j \text{ } (\text{Some } \text{abr}) = \text{None} \implies \text{abr} = \text{Jump } j$   
 $\langle \text{proof} \rangle$

**lemma** *absorb-Some-JumpD*:  $\text{absorb } j \text{ } s = \text{Some } (\text{Jump } j') \implies j' \neq j$   
 $\langle \text{proof} \rangle$

**full program state****types**

$state = abopt \times st$  — state including abrupton information

**syntax**

$Norm :: st \Rightarrow state$   
 $abrupt :: state \Rightarrow abopt$   
 $store :: state \Rightarrow st$

**translations**

$Norm\ s == (None, s)$   
 $abrupt ==> fst$   
 $store ==> snd$   
 $abopt <= (type)\ State.abrupt\ option$   
 $abopt <= (type)\ abrupt\ option$   
 $state <= (type)\ abopt \times State.st$   
 $state <= (type)\ abopt \times st$

**lemma** *single-stateE*:  $\forall Z. Z = (s::state) \implies False$   
 $\langle proof \rangle$

**lemma** *state-not-single*:  $All\ (op = (x::state)) \implies R$   
 $\langle proof \rangle$

**constdefs**

$normal :: state \Rightarrow bool$   
 $normal \equiv \lambda s. abrupt\ s = None$

**lemma** *normal-def2* [*simp*]:  $normal\ s = (abrupt\ s = None)$   
 $\langle proof \rangle$

**constdefs**

$heap-free :: nat \Rightarrow state \Rightarrow bool$   
 $heap-free\ n \equiv \lambda s. atleast-free\ (heap\ (store\ s))\ n$

**lemma** *heap-free-def2* [*simp*]:  $heap-free\ n\ s = atleast-free\ (heap\ (store\ s))\ n$   
 $\langle proof \rangle$

**41 update****constdefs**

$abupd :: (abopt \Rightarrow abopt) \Rightarrow state \Rightarrow state$   
 $abupd\ f \equiv prod-fun\ f\ id$

$supd :: (st \Rightarrow st) \Rightarrow state \Rightarrow state$   
 $supd \equiv prod-fun\ id$

**lemma** *abupd-def2* [*simp*]:  $abupd\ f\ (x, s) = (f\ x, s)$

$\langle \text{proof} \rangle$

**lemma** *abupd-abrupt-if-False* [simp]:  $\bigwedge s. \text{abupd} (\text{abrupt-if False } xo) s = s$   
 $\langle \text{proof} \rangle$

**lemma** *supd-def2* [simp]:  $\text{supd } f (x, s) = (x, f s)$   
 $\langle \text{proof} \rangle$

**lemma** *supd-lupd* [simp]:  
 $\bigwedge s. \text{supd} (\text{lupd } vn v) s = (\text{abrupt } s, \text{lupd } vn v (\text{store } s))$   
 $\langle \text{proof} \rangle$

**lemma** *supd-gupd* [simp]:  
 $\bigwedge s. \text{supd} (\text{gupd } r \text{ obj}) s = (\text{abrupt } s, \text{gupd } r \text{ obj} (\text{store } s))$   
 $\langle \text{proof} \rangle$

**lemma** *supd-init-obj* [simp]:  
 $\text{supd} (\text{init-obj } G \text{ oi } r) s = (\text{abrupt } s, \text{init-obj } G \text{ oi } r (\text{store } s))$   
 $\langle \text{proof} \rangle$

**lemma** *abupd-store-invariant* [simp]:  $\text{store} (\text{abupd } f s) = \text{store } s$   
 $\langle \text{proof} \rangle$

**lemma** *supd-abrupt-invariant* [simp]:  $\text{abrupt} (\text{supd } f s) = \text{abrupt } s$   
 $\langle \text{proof} \rangle$

## syntax

*set-lvars* :: *locals*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  
*restore-lvars* :: *state*  $\Rightarrow$  *state*  $\Rightarrow$  *state*

## translations

*set-lvars* *l* == *supd* (*set-locals* *l*)  
*restore-lvars* *s'* *s* == *set-lvars* (*locals* (*store* *s'*)) *s*

**lemma** *set-set-lvars* [simp]:  $\bigwedge s. \text{set-lvars } l (\text{set-lvars } l' s) = \text{set-lvars } l s$   
 $\langle \text{proof} \rangle$

**lemma** *set-lvars-id* [simp]:  $\bigwedge s. \text{set-lvars} (\text{locals} (\text{store } s)) s = s$   
 $\langle \text{proof} \rangle$

## initialisation test

### constdefs

*inited* :: *qtname*  $\Rightarrow$  *globs*  $\Rightarrow$  *bool*  
*inited* *C* *g*  $\equiv g (\text{Stat } C) \neq \text{None}$

$initd \quad :: \text{qname} \Rightarrow \text{state} \Rightarrow \text{bool}$   
 $initd \ C \equiv initd \ C \circ \text{globs} \circ \text{store}$

**lemma** *not-initd-empty* [simp]:  $\neg initd \ C \ \text{empty}$   
 $\langle \text{proof} \rangle$

**lemma** *initd-gupdate* [simp]:  $initd \ C \ (g(r \mapsto obj)) = (initd \ C \ g \ \vee \ r = \text{Stat } C)$   
 $\langle \text{proof} \rangle$

**lemma** *initd-init-class-obj* [intro!]:  $initd \ C \ (\text{globs} \ (\text{init-class-obj } G \ C \ s))$   
 $\langle \text{proof} \rangle$

**lemma** *not-initdD*:  $\neg initd \ C \ g \implies g \ (\text{Stat } C) = \text{None}$   
 $\langle \text{proof} \rangle$

**lemma** *initdD*:  $initd \ C \ g \implies \exists \text{ obj. } g \ (\text{Stat } C) = \text{Some obj}$   
 $\langle \text{proof} \rangle$

**lemma** *initd-def2* [simp]:  $initd \ C \ s = initd \ C \ (\text{globs} \ (\text{store } s))$   
 $\langle \text{proof} \rangle$

*error-free*

**constdefs** *error-free*::  $\text{state} \Rightarrow \text{bool}$   
 $\text{error-free } s \equiv \neg (\exists \text{ err. abrupt } s = \text{Some } (\text{Error err}))$

**lemma** *error-free-Norm* [simp,intro]:  $\text{error-free } (\text{Norm } s)$   
 $\langle \text{proof} \rangle$

**lemma** *error-free-normal* [simp,intro]:  $\text{normal } s \implies \text{error-free } s$   
 $\langle \text{proof} \rangle$

**lemma** *error-free-Xcpt* [simp]:  $\text{error-free } (\text{Some } (\text{Xcpt } x), s)$   
 $\langle \text{proof} \rangle$

**lemma** *error-free-Jump* [simp,intro]:  $\text{error-free } (\text{Some } (\text{Jump } j), s)$   
 $\langle \text{proof} \rangle$

**lemma** *error-free-Error* [simp]:  $\text{error-free } (\text{Some } (\text{Error } e), s) = \text{False}$   
 $\langle \text{proof} \rangle$

**lemma** *error-free-Some* [simp,intro]:  
 $\neg (\exists \text{ err. } x = \text{Error err}) \implies \text{error-free } ((\text{Some } x), s)$   
 $\langle \text{proof} \rangle$

**lemma** *error-free-abupd-absorb* [simp,intro]:

$error\text{-}free\ s \implies error\text{-}free\ (abupd\ (absorb\ j)\ s)$   
 $\langle proof \rangle$

**lemma** *error-free-absorb* [simp,intro]:  
 $error\text{-}free\ (a,s) \implies error\text{-}free\ (absorb\ j\ a,\ s)$   
 $\langle proof \rangle$

**lemma** *error-free-abrupt-if* [simp,intro]:  
 $\llbracket error\text{-}free\ s; \neg (\exists\ err.\ x=Error\ err) \rrbracket$   
 $\implies error\text{-}free\ (abupd\ (abrupt\text{-}if\ p\ (Some\ x))\ s)$   
 $\langle proof \rangle$

**lemma** *error-free-abrupt-if1* [simp,intro]:  
 $\llbracket error\text{-}free\ (a,s); \neg (\exists\ err.\ x=Error\ err) \rrbracket$   
 $\implies error\text{-}free\ (abrupt\text{-}if\ p\ (Some\ x)\ a,\ s)$   
 $\langle proof \rangle$

**lemma** *error-free-abrupt-if-Xcpt* [simp,intro]:  
 $error\text{-}free\ s$   
 $\implies error\text{-}free\ (abupd\ (abrupt\text{-}if\ p\ (Some\ (Xcpt\ x)))\ s)$   
 $\langle proof \rangle$

**lemma** *error-free-abrupt-if-Xcpt1* [simp,intro]:  
 $error\text{-}free\ (a,s)$   
 $\implies error\text{-}free\ (abrupt\text{-}if\ p\ (Some\ (Xcpt\ x))\ a,\ s)$   
 $\langle proof \rangle$

**lemma** *error-free-abrupt-if-Jump* [simp,intro]:  
 $error\text{-}free\ s$   
 $\implies error\text{-}free\ (abupd\ (abrupt\text{-}if\ p\ (Some\ (Jump\ j)))\ s)$   
 $\langle proof \rangle$

**lemma** *error-free-abrupt-if-Jump1* [simp,intro]:  
 $error\text{-}free\ (a,s)$   
 $\implies error\text{-}free\ (abrupt\text{-}if\ p\ (Some\ (Jump\ j))\ a,\ s)$   
 $\langle proof \rangle$

**lemma** *error-free-raise-if* [simp,intro]:  
 $error\text{-}free\ s \implies error\text{-}free\ (abupd\ (raise\text{-}if\ p\ x)\ s)$   
 $\langle proof \rangle$

**lemma** *error-free-raise-if1* [simp,intro]:  
 $error\text{-}free\ (a,s) \implies error\text{-}free\ ((raise\text{-}if\ p\ x\ a),\ s)$   
 $\langle proof \rangle$

**lemma** *error-free-supd* [simp,intro]:  
 $error\text{-}free\ s \implies error\text{-}free\ (supd\ f\ s)$   
 $\langle proof \rangle$



**lemma** *error-free-supd1* [*simp,intro*]:  
 $\text{error-free } (a,s) \implies \text{error-free } (a,f\ s)$   
 ⟨*proof*⟩

**lemma** *error-free-set-lvars* [*simp,intro*]:  
 $\text{error-free } s \implies \text{error-free } ((\text{set-lvars } l)\ s)$   
 ⟨*proof*⟩

**lemma** *error-free-set-locals* [*simp,intro*]:  
 $\text{error-free } (x, s) \implies \text{error-free } (x, \text{set-locals } l\ s')$   
 ⟨*proof*⟩

**end**



## Chapter 15

### Eval

## 42 Operational evaluation (big-step) semantics of Java expressions and statements

theory *Eval* imports *State DeclConcepts* begin

improvements over Java Specification 1.0:

- dynamic method lookup does not need to consider the return type (cf.15.11.4.4)
- throw raises a NullPointerException if a null reference is given, and each throw of a standard exception yield a fresh exception object (was not specified)
- if there is not enough memory even to allocate an OutOfMemory exception, evaluation/execution fails, i.e. simply stops (was not specified)
- array assignment checks lhs (and may throw exceptions) before evaluating rhs
- fixed exact positions of class initializations (immediate at first active use)

design issues:

- evaluation vs. (single-step) transition semantics evaluation semantics chosen, because:
  - ++ less verbose and therefore easier to read (and to handle in proofs)
  - + more abstract
  - + intermediate values (appearing in recursive rules) need not be stored explicitly, e.g. no call body construct or stack of invocation frames containing local variables and return addresses for method calls needed
  - + convenient rule induction for subject reduction theorem
    - no interleaving (for parallelism) can be described
    - stating a property of infinite executions requires the meta-level argument that this property holds for any finite prefixes of it (e.g. stopped using a counter that is decremented to zero and then throwing an exception)
- unified evaluation for variables, expressions, expression lists, statements
- the value entry in statement rules is redundant
- the value entry in rules is irrelevant in case of exceptions, but its full inclusion helps to make the rule structure independent of exception occurrence.
- as irrelevant value entries are ignored, it does not matter if they are unique. For simplicity, (fixed) arbitrary values are preferred over "free" values.
- the rule format is such that the start state may contain an exception.
  - ++ facilitates exception handling
  - + symmetry
- the rules are defined carefully in order to be applicable even in not type-correct situations (yielding undefined values), e.g. *the-Addr (Val (Bool b)) = arbitrary*.
  - ++ fewer rules
    - less readable because of auxiliary functions like *the-Addr*

Alternative: "defensive" evaluation throwing some InternalError exception in case of (impossible, for correct programs) type mismatches

- there is exactly one rule per syntactic construct
  - + no redundancy in case distinctions
- `hallocc` fails iff there is no free heap address. When there is only one free heap address left, it returns an `OutOfMemory` exception. In this way it is guaranteed that when an `OutOfMemory` exception is thrown for the first time, there is a free location on the heap to allocate it.
- the allocation of objects that represent standard exceptions is deferred until execution of any enclosing catch clause, which is transparent to the program.
  - requires an auxiliary execution relation
  - ++ avoids copies of allocation code and awkward case distinctions (whether there is enough memory to allocate the exception) in evaluation rules
- unfortunately `new-Addr` is not directly executable because of Hilbert operator.

simplifications:

- local variables are initialized with default values (no definite assignment)
- garbage collection not considered, therefore also no finalizers
- stack overflow and memory overflow during class initialization not modelled
- exceptions in initializations not replaced by `ExceptionInInitializerError`

**types**  $vvar = val \times (val \Rightarrow state \Rightarrow state)$   
 $vals = (val, vvar, val\ list)\ sum3$

**translations**

$vvar \leq (type)\ val \times (val \Rightarrow state \Rightarrow state)$   
 $vals \leq (type)(val, vvar, val\ list)\ sum3$

To avoid redundancy and to reduce the number of rules, there is only one evaluation rule for each syntactic term. This is also true for variables (e.g. see the rules below for *LVar*, *FVar* and *AVar*). So evaluation of a variable must capture both possible further uses: read (rule *Acc*) or write (rule *Ass*) to the variable. Therefore a variable evaluates to a special value *vvar*, which is a pair, consisting of the current value (for later read access) and an update function (for later write access). Because during assignment to an array variable an exception may occur if the types don't match, the update function is very generic: it transforms the full state. This generic update function causes some technical trouble during some proofs (e.g. type safety, correctness of definite assignment). There we need to prove some additional invariant on this update function to prove the assignment correct, since the update function could potentially alter the whole state in an arbitrary manner. This invariant must be carried around through the whole induction. So for future approaches it may be better not to take such a generic update function, but only to store the address and the kind of variable (array (+ element type), local variable or field) for later assignment.

**syntax** (*xsymbols*)  
 $dummy-res :: vals\ (\Diamond)$

**translations**

$\Diamond == In1\ Unit$

**syntax**

$val-inj-vals :: expr \Rightarrow term\ ([\_]\_e\ 1000)$   
 $var-inj-vals :: var \Rightarrow term\ ([\_]\_v\ 1000)$   
 $lst-inj-vals :: expr\ list \Rightarrow term\ ([\_]\_l\ 1000)$

**translations**

$$\begin{aligned} [e]_e &\rightarrow In1\ e \\ [v]_v &\rightarrow In2\ v \\ [es]_l &\rightarrow In3\ es \end{aligned}$$
**constdefs**

$$\begin{aligned} arbitrary3 &:: ('al + 'ar, 'b, 'c)\ sum3 \Rightarrow vals \\ arbitrary3 &\equiv sum3\text{-}case\ (In1 \circ sum\text{-}case\ (\lambda x. arbitrary))\ (\lambda x. Unit)) \\ &\quad (\lambda x. In2\ arbitrary)\ (\lambda x. In3\ arbitrary) \end{aligned}$$

**lemma**  $[simp]: arbitrary3\ (In1\ x) = In1\ arbitrary$   
 $\langle proof \rangle$

**lemma**  $[simp]: arbitrary3\ (In1r\ x) = \Diamond$   
 $\langle proof \rangle$

**lemma**  $[simp]: arbitrary3\ (In2\ x) = In2\ arbitrary$   
 $\langle proof \rangle$

**lemma**  $[simp]: arbitrary3\ (In3\ x) = In3\ arbitrary$   
 $\langle proof \rangle$

**exception throwing and catching****constdefs**

$$\begin{aligned} throw &:: val \Rightarrow abopt \Rightarrow abopt \\ throw\ a'\ x &\equiv abrupt\text{-}if\ True\ (Some\ (Xcpt\ (Loc\ (the\text{-}Addr\ a'))))\ (np\ a'\ x) \end{aligned}$$
**lemma** *throw-def2*:
$$throw\ a'\ x = abrupt\text{-}if\ True\ (Some\ (Xcpt\ (Loc\ (the\text{-}Addr\ a'))))\ (np\ a'\ x)$$
 $\langle proof \rangle$ 
**constdefs**

$$\begin{aligned} fits &:: prog \Rightarrow st \Rightarrow val \Rightarrow ty \Rightarrow bool\ (\neg, \vdash\text{-}fits\text{-}[61,61,61,61]60) \\ G, s \vdash a' fits\ T &\equiv (\exists rt. T = RefT\ rt) \longrightarrow a' = Null \vee G \vdash obj\text{-}ty(lookup\text{-}obj\ s\ a') \preceq T \end{aligned}$$

**lemma** *fits-Null*  $[simp]: G, s \vdash Null fits\ T$   
 $\langle proof \rangle$

**lemma** *fits-Addr-RefT*  $[simp]$ :
$$G, s \vdash Addr\ a fits\ RefT\ t = G \vdash obj\text{-}ty\ (the\ (heap\ s\ a)) \preceq RefT\ t$$
 $\langle proof \rangle$ 

**lemma** *fitsD*:  $\bigwedge X. G, s \vdash a' fits\ T \implies (\exists pt. T = PrimT\ pt) \vee$   
 $(\exists t. T = RefT\ t) \wedge a' = Null \vee$   
 $(\exists t. T = RefT\ t) \wedge a' \neq Null \wedge G \vdash obj\text{-}ty\ (lookup\text{-}obj\ s\ a') \preceq T$   
 $\langle proof \rangle$

**constdefs**

$$\begin{aligned} catch &:: prog \Rightarrow state \Rightarrow qname \Rightarrow bool\ (\neg, \vdash\text{-}catch\text{-}[61,61,61]60) \\ G, s \vdash catch\ C &\equiv \exists xc. abrupt\ s = Some\ (Xcpt\ xc) \wedge \end{aligned}$$

$G, store\ s \vdash Addr\ (the-Loc\ xc)\ fits\ Class\ C$

**lemma** *catch-Norm* [simp]:  $\neg G, Norm\ s \vdash catch\ tn$   
 $\langle proof \rangle$

**lemma** *catch-XcptLoc* [simp]:  
 $G, (Some\ (Xcpt\ (Loc\ a)), s) \vdash catch\ C = G, s \vdash Addr\ a\ fits\ Class\ C$   
 $\langle proof \rangle$

**lemma** *catch-Jump* [simp]:  $\neg G, (Some\ (Jump\ j), s) \vdash catch\ tn$   
 $\langle proof \rangle$

**lemma** *catch-Error* [simp]:  $\neg G, (Some\ (Error\ e), s) \vdash catch\ tn$   
 $\langle proof \rangle$

### constdefs

$new-xcpt-var :: vname \Rightarrow state \Rightarrow state$   
 $new-xcpt-var\ vn \equiv$   
 $\lambda(x, s). Norm\ (lupd\ (VName\ vn \mapsto Addr\ (the-Loc\ (the-Xcpt\ (the\ x))))\ s)$

**lemma** *new-xcpt-var-def2* [simp]:  
 $new-xcpt-var\ vn\ (x, s) =$   
 $Norm\ (lupd\ (VName\ vn \mapsto Addr\ (the-Loc\ (the-Xcpt\ (the\ x))))\ s)$   
 $\langle proof \rangle$

### misc

### constdefs

$assign :: ('a \Rightarrow state \Rightarrow state) \Rightarrow 'a \Rightarrow state \Rightarrow state$   
 $assign\ f\ v \equiv \lambda(x, s). let\ (x', s') = (if\ x = None\ then\ f\ v\ else\ id)\ (x, s)$   
 $in\ (x', if\ x' = None\ then\ s'\ else\ s)$

**lemma** *assign-Norm-Norm* [simp]:  
 $f\ v\ (Norm\ s) = Norm\ s' \implies assign\ f\ v\ (Norm\ s) = Norm\ s'$   
 $\langle proof \rangle$

**lemma** *assign-Norm-Some* [simp]:  
 $\llbracket abrupt\ (f\ v\ (Norm\ s)) = Some\ y \rrbracket$   
 $\implies assign\ f\ v\ (Norm\ s) = (Some\ y, s)$   
 $\langle proof \rangle$

**lemma** *assign-Some* [simp]:  
 $assign\ f\ v\ (Some\ x, s) = (Some\ x, s)$   
 $\langle proof \rangle$

**lemma** *assign-Some1* [simp]:  $\neg \text{normal } s \implies \text{assign } f \ v \ s = s$   
 ⟨proof⟩

**lemma** *assign-supd* [simp]:  
 $\text{assign } (\lambda v. \text{supd } (f \ v)) \ v \ (x, s)$   
 $= (x, \text{if } x = \text{None} \text{ then } f \ v \ s \text{ else } s)$   
 ⟨proof⟩

**lemma** *assign-raise-if* [simp]:  
 $\text{assign } (\lambda v \ (x, s). ((\text{raise-if } (b \ s \ v) \ \text{xcpt}) \ x, f \ v \ s)) \ v \ (x, s) =$   
 $(\text{raise-if } (b \ s \ v) \ \text{xcpt} \ x, \text{if } x = \text{None} \wedge \neg b \ s \ v \text{ then } f \ v \ s \text{ else } s)$   
 ⟨proof⟩

### constdefs

*init-comp-ty* :: *ty*  $\Rightarrow$  *stmt*  
*init-comp-ty* *T*  $\equiv$  *if* ( $\exists C. T = \text{Class } C$ ) *then* *Init* (*the-Class* *T*) *else* *Skip*

**lemma** *init-comp-ty-PrimT* [simp]: *init-comp-ty* (*PrimT* *pt*) = *Skip*  
 ⟨proof⟩

### constdefs

*invocation-class* :: *inv-mode*  $\Rightarrow$  *st*  $\Rightarrow$  *val*  $\Rightarrow$  *ref-ty*  $\Rightarrow$  *qname*  
*invocation-class* *m* *s* *a'* *statT*  
 $\equiv$  (*case* *m* *of*  
   *Static*  $\Rightarrow$  *if* ( $\exists \text{statC}. \text{statT} = \text{ClassT } \text{statC}$ )  
     *then* *the-Class* (*RefT* *statT*)  
     *else* *Object*  
   | *SuperM*  $\Rightarrow$  *the-Class* (*RefT* *statT*)  
   | *IntVir*  $\Rightarrow$  *obj-class* (*lookup-obj* *s* *a'*)

*invocation-declclass*::*prog*  $\Rightarrow$  *inv-mode*  $\Rightarrow$  *st*  $\Rightarrow$  *val*  $\Rightarrow$  *ref-ty*  $\Rightarrow$  *sig*  $\Rightarrow$  *qname*  
*invocation-declclass* *G* *m* *s* *a'* *statT* *sig*  
 $\equiv$  *declclass* (*the* (*dynlookup* *G* *statT*)  
   (*invocation-class* *m* *s* *a'* *statT*)  
   *sig*)

**lemma** *invocation-class-IntVir* [simp]:  
*invocation-class* *IntVir* *s* *a'* *statT* = *obj-class* (*lookup-obj* *s* *a'*)  
 ⟨proof⟩

**lemma** *dynclass-SuperM* [simp]:  
*invocation-class* *SuperM* *s* *a'* *statT* = *the-Class* (*RefT* *statT*)  
 ⟨proof⟩

**lemma** *invocation-class-Static* [simp]:  
*invocation-class* *Static* *s* *a'* *statT* = (*if* ( $\exists \text{statC}. \text{statT} = \text{ClassT } \text{statC}$ )  
   *then* *the-Class* (*RefT* *statT*)



*else Object)*

*<proof>*

### constdefs

*init-lvars* :: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  *sig*  $\Rightarrow$  *inv-mode*  $\Rightarrow$  *val*  $\Rightarrow$  *val list*  $\Rightarrow$   
*state*  $\Rightarrow$  *state*

*init-lvars G C sig mode a' pvs*

$\equiv \lambda (x,s).$

*let m = methd (the (methd G C sig));*

*l = \lambda k.*

*(case k of*

*EName e*

$\Rightarrow$  *(case e of*

*VNam v*  $\Rightarrow$  *(empty ((pars m)[ $\mapsto$ ]pvs)) v*

*| Res*  $\Rightarrow$  *None)*

*| This*

$\Rightarrow$  *(if mode=Static then None else Some a')*

*in set-lvars l (if mode = Static then x else np a' x,s)*

**lemma** *init-lvars-def2*: — better suited for simplification

*init-lvars G C sig mode a' pvs (x,s) =*

*set-lvars*

*(\lambda k.*

*(case k of*

*EName e*

$\Rightarrow$  *(case e of*

*VNam v*

$\Rightarrow$  *(empty ((pars (methd (the (methd G C sig))))[ $\mapsto$ ]pvs)) v*

*| Res*  $\Rightarrow$  *None)*

*| This*

$\Rightarrow$  *(if mode=Static then None else Some a')*

*(if mode = Static then x else np a' x,s)*

*<proof>*

### constdefs

*body* :: *prog*  $\Rightarrow$  *qname*  $\Rightarrow$  *sig*  $\Rightarrow$  *expr*

*body G C sig*  $\equiv$  *let m = the (methd G C sig)*

*in Body (declclass m) (stmt (mbody (methd m)))*

**lemma** *body-def2*: — better suited for simplification

*body G C sig = Body (declclass (the (methd G C sig)))*

*(stmt (mbody (methd (the (methd G C sig)))))*

*<proof>*

### variables

#### constdefs

*lvar* :: *lname*  $\Rightarrow$  *st*  $\Rightarrow$  *vvar*

*lvar vn s*  $\equiv$  *(the (locals s vn), \lambda v. supd (lupd(vn $\mapsto$ v)))*

*fvar* :: *qname*  $\Rightarrow$  *bool*  $\Rightarrow$  *vname*  $\Rightarrow$  *val*  $\Rightarrow$  *state*  $\Rightarrow$  *vvar*  $\times$  *state*

*fvar C stat fn a' s*

$\equiv$  *let (oref,xf) = if stat then (Stat C,id)*

*else (Heap (the-Addr a'),np a');*

$$\begin{aligned}
& n = \text{Inl } (fn, C); \\
& f = (\lambda v. \text{supd } (\text{upd-gobj } \text{oref } n \ v)) \\
& \text{in } ((\text{the } (\text{values } (\text{the } (\text{globs } (\text{store } s) \ \text{oref}))) \ n), f), \text{abupd } xf \ s) \\
\\
& \text{avar} :: \text{prog} \Rightarrow \text{val} \Rightarrow \text{val} \Rightarrow \text{state} \Rightarrow \text{vvar} \times \text{state} \\
& \text{avar } G \ i' \ a' \ s = \\
& \quad \equiv \text{let } \text{oref} = \text{Heap } (\text{the-Addr } a'); \\
& \quad \quad i = \text{the-Intg } i'; \\
& \quad \quad n = \text{Inr } i; \\
& \quad \quad (T, k, cs) = \text{the-Arr } (\text{globs } (\text{store } s) \ \text{oref}); \\
& \quad \quad f = (\lambda v \ (x, s). \ (\text{raise-if } (\neg G, s \vdash v \ \text{fits } T) \\
& \quad \quad \quad \text{ArrStore } x \\
& \quad \quad \quad , \text{upd-gobj } \text{oref } n \ v \ s)) \\
& \text{in } ((\text{the } (cs \ n), f) \\
& \quad , \text{abupd } (\text{raise-if } (\neg i \ \text{in-bounds } k) \ \text{IndOutBound} \circ np \ a') \ s)
\end{aligned}$$

**lemma** *fvar-def2*: — better suited for simplification

$$\begin{aligned}
& \text{fvar } C \ \text{stat} \ fn \ a' \ s = \\
& \quad ((\text{the} \\
& \quad \quad (\text{values} \\
& \quad \quad \quad (\text{the } (\text{globs } (\text{store } s) \ (\text{if } \text{stat} \ \text{then } \text{Stat } C \ \text{else } \text{Heap } (\text{the-Addr } a')))) \\
& \quad \quad \quad (\text{Inl } (fn, C))) \\
& \quad \quad , (\lambda v. \text{supd } (\text{upd-gobj } (\text{if } \text{stat} \ \text{then } \text{Stat } C \ \text{else } \text{Heap } (\text{the-Addr } a')) \\
& \quad \quad \quad \quad (\text{Inl } (fn, C)) \\
& \quad \quad \quad \quad v))) \\
& \quad \quad , \text{abupd } (\text{if } \text{stat} \ \text{then } \text{id} \ \text{else } np \ a') \ s)
\end{aligned}$$

$\langle \text{proof} \rangle$

**lemma** *avar-def2*: — better suited for simplification

$$\begin{aligned}
& \text{avar } G \ i' \ a' \ s = \\
& \quad ((\text{the } ((\text{snd}(\text{snd}(\text{the-Arr } (\text{globs } (\text{store } s) \ (\text{Heap } (\text{the-Addr } a')))))) \\
& \quad \quad (\text{Inr } (\text{the-Intg } i')) \\
& \quad \quad , (\lambda v \ (x, s'). \ (\text{raise-if } (\neg G, s \vdash v \ \text{fits } (\text{fst}(\text{the-Arr } (\text{globs } (\text{store } s) \\
& \quad \quad \quad \quad (\text{Heap } (\text{the-Addr } a')))))) \\
& \quad \quad \quad \text{ArrStore } x \\
& \quad \quad \quad , \text{upd-gobj } (\text{Heap } (\text{the-Addr } a')) \\
& \quad \quad \quad \quad (\text{Inr } (\text{the-Intg } i')) \ v \ s')) \\
& \quad \quad , \text{abupd } (\text{raise-if } (\neg(\text{the-Intg } i') \ \text{in-bounds } (\text{fst}(\text{snd}(\text{the-Arr } (\text{globs } (\text{store } s) \\
& \quad \quad \quad \quad (\text{Heap } (\text{the-Addr } a')))))) \ \text{IndOutBound} \circ np \ a') \\
& \quad \quad s)
\end{aligned}$$

$\langle \text{proof} \rangle$

**constdefs**

*check-field-access*:

$\text{prog} \Rightarrow \text{qname} \Rightarrow \text{qname} \Rightarrow \text{vname} \Rightarrow \text{bool} \Rightarrow \text{val} \Rightarrow \text{state} \Rightarrow \text{state}$

*check-field-access*  $G \ \text{accC} \ \text{statDeclC} \ fn \ \text{stat} \ a' \ s$

$$\begin{aligned}
& \equiv \text{let } \text{oref} = \text{if } \text{stat} \ \text{then } \text{Stat } \text{statDeclC} \\
& \quad \quad \text{else } \text{Heap } (\text{the-Addr } a'); \\
& \quad \text{dynC} = \text{case } \text{oref} \ \text{of} \\
& \quad \quad \quad \text{Heap } a \Rightarrow \text{obj-class } (\text{the } (\text{globs } (\text{store } s) \ \text{oref})) \\
& \quad \quad \quad | \ \text{Stat } C \Rightarrow C; \\
& \quad f = (\text{the } (\text{table-of } (\text{DeclConcepts.fields } G \ \text{dynC}) \ (fn, \text{statDeclC}))) \\
& \text{in } \text{abupd} \\
& \quad (\text{error-if } (\neg G \vdash \text{Field } fn \ (\text{statDeclC}, f) \ \text{in } \text{dynC} \ \text{dyn-accessible-from } \text{accC}) \\
& \quad \quad \text{AccessViolation})
\end{aligned}$$

$s$

**constdefs**

*check-method-access::*

$prog \Rightarrow qname \Rightarrow ref\text{-}ty \Rightarrow inv\text{-}mode \Rightarrow sig \Rightarrow val \Rightarrow state \Rightarrow state$   
*check-method-access*  $G \text{ acc}C \text{ stat}T \text{ mode } sig \text{ a}' s$   
 $\equiv let \text{ inv}C = invocation\text{-}class \text{ mode } (store \text{ s}) \text{ a}' \text{ stat}T;$   
 $\text{ dyn}M = the \text{ (dynlookup } G \text{ stat}T \text{ inv}C \text{ sig)}$   
*in* *abupd*  
 $(error\text{-}if (\neg G \vdash Methd \text{ sig } \text{ dyn}M \text{ in } \text{ inv}C \text{ dyn-accessible-from } \text{ acc}C)$   
 $\text{ AccessViolation})$   
 $s$

**evaluation judgments****consts**

*eval* ::  $prog \Rightarrow (state \times term \times vals \times state) \text{ set}$   
*hallo*::  $prog \Rightarrow (state \times obj\text{-}tag \times loc \times state) \text{ set}$   
*salloc*::  $prog \Rightarrow (state \times state) \text{ set}$

**syntax**

*eval* ::  $[prog, state, term, vals * state] \Rightarrow bool(-|- \rightarrow - \text{ - } [61, 61, 80, 61] 60)$   
*exec* ::  $[prog, state, stmt, state] \Rightarrow bool(-|- \rightarrow - \text{ - } [61, 61, 65, 61] 60)$   
*evar* ::  $[prog, state, var, vvar, state] \Rightarrow bool(-|- \rightarrow - \text{ - } [61, 61, 90, 61, 61] 60)$   
*eval*::  $[prog, state, expr, val, state] \Rightarrow bool(-|- \rightarrow - \text{ - } [61, 61, 80, 61, 61] 60)$   
*evals*::  $[prog, state, expr \text{ list }, val \text{ list }, state] \Rightarrow bool(-|- \rightarrow - \text{ - } [61, 61, 61, 61, 61] 60)$   
*hallo*::  $[prog, state, obj\text{-}tag, loc, state] \Rightarrow bool(-|- \rightarrow - \text{ - } [61, 61, 61, 61, 61] 60)$   
*sallo*::  $[prog, state, state] \Rightarrow bool(-|- \rightarrow - \text{ - } [61, 61, 61] 60)$

**syntax** (*x*symbols)

*eval* ::  $[prog, state, term, vals \times state] \Rightarrow bool(\vdash - \rightarrow - \text{ - } [61, 61, 80, 61] 60)$   
*exec* ::  $[prog, state, stmt, state] \Rightarrow bool(\vdash - \rightarrow - \text{ - } [61, 61, 65, 61] 60)$   
*evar* ::  $[prog, state, var, vvar, state] \Rightarrow bool(\vdash - \rightarrow - \text{ - } [61, 61, 90, 61, 61] 60)$   
*eval*::  $[prog, state, expr, val, state] \Rightarrow bool(\vdash - \rightarrow - \text{ - } [61, 61, 80, 61, 61] 60)$   
*evals*::  $[prog, state, expr \text{ list }, val \text{ list }, state] \Rightarrow bool(\vdash - \rightarrow - \text{ - } [61, 61, 61, 61, 61] 60)$   
*hallo*::  $[prog, state, obj\text{-}tag, loc, state] \Rightarrow bool(\vdash - \rightarrow - \text{ - } [61, 61, 61, 61, 61] 60)$   
*sallo*::  $[prog, state, state] \Rightarrow bool(\vdash - \rightarrow - \text{ - } [61, 61, 61] 60)$

**translations**

$G \vdash s - t \rightarrow w \text{ --- } s' \equiv (s, t, w \text{ --- } s') \in eval \text{ } G$   
 $G \vdash s - t \rightarrow (w, s') \leq (s, t, w, s') \in eval \text{ } G$   
 $G \vdash s - t \rightarrow (w, x, s') \leq (s, t, w, x, s') \in eval \text{ } G$   
 $G \vdash s - c \rightarrow (x, s') \leq G \vdash s - In1r \text{ } c \rightarrow (\Diamond, x, s')$   
 $G \vdash s - c \rightarrow s' \equiv G \vdash s - In1r \text{ } c \rightarrow (\Diamond, s')$   
 $G \vdash s - e \rightarrow v \rightarrow (x, s') \leq G \vdash s - In1l \text{ } e \rightarrow (In1 \text{ } v, x, s')$   
 $G \vdash s - e \rightarrow v \rightarrow s' \equiv G \vdash s - In1l \text{ } e \rightarrow (In1 \text{ } v, s')$   
 $G \vdash s - e \rightarrow vf \rightarrow (x, s') \leq G \vdash s - In2 \text{ } e \rightarrow (In2 \text{ } vf, x, s')$   
 $G \vdash s - e \rightarrow vf \rightarrow s' \equiv G \vdash s - In2 \text{ } e \rightarrow (In2 \text{ } vf, s')$   
 $G \vdash s - e \rightarrow v \rightarrow (x, s') \leq G \vdash s - In3 \text{ } e \rightarrow (In3 \text{ } v, x, s')$   
 $G \vdash s - e \rightarrow v \rightarrow s' \equiv G \vdash s - In3 \text{ } e \rightarrow (In3 \text{ } v, s')$   
 $G \vdash s - hallo \text{ } oi \rightarrow a \rightarrow (x, s') \leq (s, oi, a, x, s') \in hallo \text{ } G$   
 $G \vdash s - hallo \text{ } oi \rightarrow a \rightarrow s' \equiv (s, oi, a, s') \in hallo \text{ } G$   
 $G \vdash s - salloc \rightarrow (x, s') \leq (s, x, s') \in salloc \text{ } G$   
 $G \vdash s - salloc \rightarrow s' \equiv (s, s') \in salloc \text{ } G$

**inductive *halloc* *G* intros** — allocating objects on the heap, cf. 12.5

*Abrupt*:

$$G \vdash (\text{Some } x, s) \text{ --halloc } oi \succ \text{arbitrary} \rightarrow (\text{Some } x, s)$$

$$\begin{aligned} \text{New: } & \llbracket \text{new-Addr } (\text{heap } s) = \text{Some } a; \\ & (x, oi') = (\text{if atleast-free } (\text{heap } s) (\text{Suc } (\text{Suc } 0)) \text{ then } (\text{None}, oi) \\ & \quad \text{else } (\text{Some } (\text{Xcpt } (\text{Loc } a)), \text{CInst } (\text{SXcpt OutOfMemory}))) \rrbracket \\ & \implies \\ & G \vdash \text{Norm } s \text{ --halloc } oi \succ a \rightarrow (x, \text{init-obj } G \text{ } oi' (\text{Heap } a) \text{ } s) \end{aligned}$$

**inductive *sxalloc* *G* intros** — allocating exception objects for standard exceptions (other than OutOfMemory)

$$\text{Norm: } G \vdash \text{Norm } s \text{ --sxalloc} \rightarrow \text{Norm } s$$

$$\text{Jmp: } G \vdash (\text{Some } (\text{Jump } j), s) \text{ --sxalloc} \rightarrow (\text{Some } (\text{Jump } j), s)$$

$$\text{Error: } G \vdash (\text{Some } (\text{Error } e), s) \text{ --sxalloc} \rightarrow (\text{Some } (\text{Error } e), s)$$

$$\text{XcptL: } G \vdash (\text{Some } (\text{Xcpt } (\text{Loc } a)), s) \text{ --sxalloc} \rightarrow (\text{Some } (\text{Xcpt } (\text{Loc } a)), s)$$

$$\begin{aligned} \text{SXcpt: } & \llbracket G \vdash \text{Norm } s0 \text{ --halloc } (\text{CInst } (\text{SXcpt } xn)) \succ a \rightarrow (x, s1) \rrbracket \implies \\ & G \vdash (\text{Some } (\text{Xcpt } (\text{Std } xn)), s0) \text{ --sxalloc} \rightarrow (\text{Some } (\text{Xcpt } (\text{Loc } a)), s1) \end{aligned}$$

**inductive *eval* *G* intros**

— propagation of abrupt completion

— cf. 14.1, 15.5

*Abrupt*:

$$G \vdash (\text{Some } xc, s) \text{ --t} \succ \rightarrow (\text{arbitrary3 } t, (\text{Some } xc, s))$$

— execution of statements

— cf. 14.5

$$\text{Skip: } G \vdash \text{Norm } s \text{ --Skip} \rightarrow \text{Norm } s$$

— cf. 14.7

$$\begin{aligned} \text{Expr: } & \llbracket G \vdash \text{Norm } s0 \text{ --e} \succ v \rightarrow s1 \rrbracket \implies \\ & G \vdash \text{Norm } s0 \text{ --Expr } e \rightarrow s1 \end{aligned}$$

$$\begin{aligned} \text{Lab: } & \llbracket G \vdash \text{Norm } s0 \text{ --c} \rightarrow s1 \rrbracket \implies \\ & G \vdash \text{Norm } s0 \text{ --l} \cdot c \rightarrow \text{abupd } (\text{absorb } l) \text{ } s1 \end{aligned}$$

— cf. 14.2

$$\begin{aligned} \text{Comp: } & \llbracket G \vdash \text{Norm } s0 \text{ --c1} \rightarrow s1; \\ & G \vdash s1 \text{ --c2} \rightarrow s2 \rrbracket \implies \\ & G \vdash \text{Norm } s0 \text{ --c1;; c2} \rightarrow s2 \end{aligned}$$

— cf. 14.8.2

$$\begin{aligned} \text{If: } & \llbracket G \vdash \text{Norm } s0 \text{ --e} \succ b \rightarrow s1; \\ & G \vdash s1 \text{ --(if the-Bool } b \text{ then } c1 \text{ else } c2) \rightarrow s2 \rrbracket \implies \\ & G \vdash \text{Norm } s0 \text{ --If}(e) \text{ } c1 \text{ Else } c2 \rightarrow s2 \end{aligned}$$

— cf. 14.10, 14.10.1

— A continue jump from the while body *c* is handled by this rule. If a continue jump with the proper label was invoked inside *c* this label (Cont *l*) is deleted out of the abrupt component of the state before the

iterative evaluation of the while statement. A break jump is handled by the Lab Statement *Lab l (while...)*.

*Loop*:  $\llbracket G \vdash \text{Norm } s0 \text{ } -e \rightarrow b \rightarrow s1; \\ \text{if the-Bool } b \\ \text{then } (G \vdash s1 \text{ } -c \rightarrow s2 \wedge \\ G \vdash (\text{abupd } (\text{absorb } (\text{Cont } l)) \text{ } s2) \text{ } -l \cdot \text{While}(e) \text{ } c \rightarrow s3) \\ \text{else } s3 = s1 \rrbracket \Longrightarrow \\ G \vdash \text{Norm } s0 \text{ } -l \cdot \text{While}(e) \text{ } c \rightarrow s3$

*Jmp*:  $G \vdash \text{Norm } s \text{ } -\text{Jmp } j \rightarrow (\text{Some } (\text{Jump } j), s)$

— cf. 14.16

*Throw*:  $\llbracket G \vdash \text{Norm } s0 \text{ } -e \rightarrow a' \rightarrow s1 \rrbracket \Longrightarrow \\ G \vdash \text{Norm } s0 \text{ } -\text{Throw } e \rightarrow \text{abupd } (\text{throw } a') \text{ } s1$

— cf. 14.18.1

*Try*:  $\llbracket G \vdash \text{Norm } s0 \text{ } -c1 \rightarrow s1; G \vdash s1 \text{ } -\text{salloc} \rightarrow s2; \\ \text{if } G, s2 \vdash \text{catch } C \text{ then } G \vdash \text{new-xcpt-var } vn \text{ } s2 \text{ } -c2 \rightarrow s3 \text{ else } s3 = s2 \rrbracket \Longrightarrow \\ G \vdash \text{Norm } s0 \text{ } -\text{Try } c1 \text{ } \text{Catch}(C \text{ } vn) \text{ } c2 \rightarrow s3$

— cf. 14.18.2

*Fin*:  $\llbracket G \vdash \text{Norm } s0 \text{ } -c1 \rightarrow (x1, s1); \\ G \vdash \text{Norm } s1 \text{ } -c2 \rightarrow s2; \\ s3 = (\text{if } (\exists \text{ err. } x1 = \text{Some } (\text{Error } \text{err})) \\ \text{then } (x1, s1) \\ \text{else } \text{abupd } (\text{abrupt-if } (x1 \neq \text{None}) \text{ } x1) \text{ } s2) \rrbracket \\ \Longrightarrow \\ G \vdash \text{Norm } s0 \text{ } -c1 \text{ } \text{Finally } c2 \rightarrow s3$

— cf. 12.4.2, 8.5

*Init*:  $\llbracket \text{the } (\text{class } G \text{ } C) = c; \\ \text{if } \text{inited } C \text{ } (\text{globs } s0) \text{ then } s3 = \text{Norm } s0 \\ \text{else } (G \vdash \text{Norm } (\text{init-class-obj } G \text{ } C \text{ } s0) \\ -(\text{if } C = \text{Object then Skip else Init } (\text{super } c)) \rightarrow s1 \wedge \\ G \vdash \text{set-lvars empty } s1 \text{ } -\text{init } c \rightarrow s2 \wedge s3 = \text{restore-lvars } s1 \text{ } s2) \rrbracket \\ \Longrightarrow \\ G \vdash \text{Norm } s0 \text{ } -\text{Init } C \rightarrow s3$

— This class initialisation rule is a little bit inaccurate. Look at the exact sequence: (1) The current class object (the static fields) are initialised (*init-class-obj*), (2) the superclasses are initialised, (3) the static initialiser of the current class is invoked. More precisely we should expect another ordering, namely 2 1 3. But we can't just naively toggle 1 and 2. By calling *init-class-obj* before initialising the superclasses, we also implicitly record that we have started to initialise the current class (by setting an value for the class object). This becomes crucial for the completeness proof of the axiomatic semantics *AxCompl.thy*. Static initialisation requires an induction on the number of classes not yet initialised (or to be more precise, classes where the initialisation has not yet begun). So we could first assign a dummy value to the class before superclass initialisation and afterwards set the correct values. But as long as we don't take memory overflow into account when allocating class objects, we can leave things as they are for convenience.

— evaluation of expressions

— cf. 15.8.1, 12.4.1

*NewC*:  $\llbracket G \vdash \text{Norm } s0 \text{ } -\text{Init } C \rightarrow s1; \\ G \vdash s1 \text{ } -\text{halloc } (C \text{Inst } C) \rightarrow a \rightarrow s2 \rrbracket \Longrightarrow \\ G \vdash \text{Norm } s0 \text{ } -\text{NewC } C \rightarrow \text{Addr } a \rightarrow s2$

— cf. 15.9.1, 12.4.1

*NewA*:  $\llbracket G \vdash \text{Norm } s0 \text{ } -\text{init-comp-ty } T \rightarrow s1; G \vdash s1 \text{ } -e \rightarrow i' \rightarrow s2; \\ G \vdash \text{abupd } (\text{check-neg } i') \text{ } s2 \text{ } -\text{halloc } (\text{Arr } T \text{ } (\text{the-Intg } i')) \rightarrow a \rightarrow s3 \rrbracket \Longrightarrow \\ G \vdash \text{Norm } s0 \text{ } -\text{New } T[e] \rightarrow \text{Addr } a \rightarrow s3$

— cf. 15.15

*Cast*:  $\llbracket G \vdash \text{Norm } s0 \text{ } -e \rightarrow v \rightarrow s1; \\$

$$s2 = \text{abupd } (\text{raise-if } (\neg G, \text{store } s1 \vdash v \text{ fits } T) \text{ ClassCast}) s1 \rhd \implies \\ G \vdash \text{Norm } s0 - \text{Cast } T \text{ e} - \succ v \rightarrow s2$$

— cf. 15.19.2

$$\text{Inst: } \llbracket G \vdash \text{Norm } s0 - \text{e} - \succ v \rightarrow s1; \\ b = (v \neq \text{Null} \wedge G, \text{store } s1 \vdash v \text{ fits } \text{RefT } T) \rrbracket \implies \\ G \vdash \text{Norm } s0 - \text{e InstOf } T - \succ \text{Bool } b \rightarrow s1$$

— cf. 15.7.1

$$\text{Lit: } G \vdash \text{Norm } s - \text{Lit } v - \succ v \rightarrow \text{Norm } s$$

$$\text{UnOp: } \llbracket G \vdash \text{Norm } s0 - \text{e} - \succ v \rightarrow s1 \rrbracket \\ \implies G \vdash \text{Norm } s0 - \text{UnOp } \text{unop } \text{e} - \succ (\text{eval-unop } \text{unop } v) \rightarrow s1$$

$$\text{BinOp: } \llbracket G \vdash \text{Norm } s0 - \text{e1} - \succ v1 \rightarrow s1; \\ G \vdash s1 - (\text{if need-second-arg binop } v1 \text{ then } (\text{In1l } e2) \text{ else } (\text{In1r Skip})) \\ \succ \rightarrow (\text{In1 } v2, s2) \rrbracket \\ \implies G \vdash \text{Norm } s0 - \text{BinOp } \text{binop } e1 \text{ e2} - \succ (\text{eval-binop } \text{binop } v1 \text{ v2}) \rightarrow s2$$

— cf. 15.10.2

$$\text{Super: } G \vdash \text{Norm } s - \text{Super} - \succ \text{val-this } s \rightarrow \text{Norm } s$$

— cf. 15.2

$$\text{Acc: } \llbracket G \vdash \text{Norm } s0 - \text{va} = \succ (v, f) \rightarrow s1 \rrbracket \implies \\ G \vdash \text{Norm } s0 - \text{Acc } \text{va} - \succ v \rightarrow s1$$

— cf. 15.25.1

$$\text{Ass: } \llbracket G \vdash \text{Norm } s0 - \text{va} = \succ (w, f) \rightarrow s1; \\ G \vdash s1 - \text{e} - \succ v \rightarrow s2 \rrbracket \implies \\ G \vdash \text{Norm } s0 - \text{va} := \text{e} - \succ v \rightarrow \text{assign } f \text{ v } s2$$

— cf. 15.24

$$\text{Cond: } \llbracket G \vdash \text{Norm } s0 - \text{e0} - \succ b \rightarrow s1; \\ G \vdash s1 - (\text{if the-Bool } b \text{ then } e1 \text{ else } e2) - \succ v \rightarrow s2 \rrbracket \implies \\ G \vdash \text{Norm } s0 - \text{e0} ? e1 : e2 - \succ v \rightarrow s2$$

— The interplay of *Call*, *Methd* and *Body*: Method invocation is split up into these three rules:

*Call* Calculates the target address and evaluates the arguments of the method, and then performs dynamic or static lookup of the method, corresponding to the call mode. Then the *Methd* rule is evaluated on the calculated declaration class of the method invocation.

*Methd* A syntactic bridge for the folded method body. It is used by the axiomatic semantics to add the proper hypothesis for recursive calls of the method.

*Body* An extra syntactic entity for the unfolded method body was introduced to properly trigger class initialisation. Without class initialisation we could just evaluate the body statement.

— cf. 15.11.4.1, 15.11.4.2, 15.11.4.4, 15.11.4.5

*Call:*

$$\llbracket G \vdash \text{Norm } s0 - \text{e} - \succ a' \rightarrow s1; G \vdash s1 - \text{args} \dot{=} \succ vs \rightarrow s2; \\ D = \text{invocation-declclass } G \text{ mode } (\text{store } s2) \text{ a' statT } (\llbracket \text{name} = \text{mn}, \text{parTs} = \text{pTs} \rrbracket); \\ s3 = \text{init-lvars } G \text{ D } (\llbracket \text{name} = \text{mn}, \text{parTs} = \text{pTs} \rrbracket) \text{ mode } a' \text{ vs } s2; \\ s3' = \text{check-method-access } G \text{ accC statT mode } (\llbracket \text{name} = \text{mn}, \text{parTs} = \text{pTs} \rrbracket) \text{ a' } s3; \\ G \vdash s3' - \text{Methd } D (\llbracket \text{name} = \text{mn}, \text{parTs} = \text{pTs} \rrbracket) - \succ v \rightarrow s4 \rrbracket \\ \implies \\ G \vdash \text{Norm } s0 - \{ \text{accC}, \text{statT}, \text{mode} \} \text{e.mn}(\{ \text{pTs} \} \text{args}) - \succ v \rightarrow (\text{restore-lvars } s2 \text{ s4})$$



$$G \vdash (\text{Some } xc, s) \text{ --halloc } oi \succ a \rightarrow s'$$

$$G \vdash (\text{Norm } s) \text{ --halloc } oi \succ a \rightarrow s'$$

**inductive-cases** *sxalloc-elim-cases*:

$$G \vdash \text{Norm } s \text{ --sxalloc} \rightarrow s'$$

$$G \vdash (\text{Some } (\text{Jump } j), s) \text{ --sxalloc} \rightarrow s'$$

$$G \vdash (\text{Some } (\text{Error } e), s) \text{ --sxalloc} \rightarrow s'$$

$$G \vdash (\text{Some } (\text{Xcpt } (\text{Loc } a)), s) \text{ --sxalloc} \rightarrow s'$$

$$G \vdash (\text{Some } (\text{Xcpt } (\text{Std } xn)), s) \text{ --sxalloc} \rightarrow s'$$

**inductive-cases** *sxalloc-cases*:  $G \vdash s \text{ --sxalloc} \rightarrow s'$

**lemma** *sxalloc-elim-cases2*:  $\llbracket G \vdash s \text{ --sxalloc} \rightarrow s' \rrbracket$ ;

$$\bigwedge s. \llbracket s' = \text{Norm } s \rrbracket \implies P;$$

$$\bigwedge j s. \llbracket s' = (\text{Some } (\text{Jump } j), s) \rrbracket \implies P;$$

$$\bigwedge e s. \llbracket s' = (\text{Some } (\text{Error } e), s) \rrbracket \implies P;$$

$$\bigwedge a s. \llbracket s' = (\text{Some } (\text{Xcpt } (\text{Loc } a)), s) \rrbracket \implies P$$

$$\rrbracket \implies P$$

$\langle \text{proof} \rangle$

**declare** *not-None-eq* [*simp del*]

**declare** *split-paired-All* [*simp del*] *split-paired-Ex* [*simp del*]

$\langle \text{ML} \rangle$

**inductive-cases** *eval-cases*:  $G \vdash s \text{ --}t \succ \rightarrow vs'$

**inductive-cases** *eval-elim-cases* [*cases set*]:

$$G \vdash (\text{Some } xc, s) \text{ --}t \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In1r Skip} \succ \rightarrow xs'$$

$$G \vdash \text{Norm } s \text{ --In1r (Jmp } j) \succ \rightarrow xs'$$

$$G \vdash \text{Norm } s \text{ --In1r (l. c)} \succ \rightarrow xs'$$

$$G \vdash \text{Norm } s \text{ --In3 } (\llbracket \rrbracket) \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In3 } (e \# es) \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In1l (Lit } w) \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In1l (UnOp unop } e) \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In1l (BinOp binop } e1 \ e2) \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In2 (LVar } vn) \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In1l (Cast } T \ e) \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In1l (e InstOf } T) \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In1l (Super)} \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In1l (Acc } va) \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In1r (Expr } e) \succ \rightarrow xs'$$

$$G \vdash \text{Norm } s \text{ --In1r (c1;; c2)} \succ \rightarrow xs'$$

$$G \vdash \text{Norm } s \text{ --In1l (Methd } C \ sig) \succ \rightarrow xs'$$

$$G \vdash \text{Norm } s \text{ --In1l (Body } D \ c) \succ \rightarrow xs'$$

$$G \vdash \text{Norm } s \text{ --In1l (e0 ? e1 : e2)} \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In1r (If(e) c1 Else c2)} \succ \rightarrow xs'$$

$$G \vdash \text{Norm } s \text{ --In1r (l. While(e) c)} \succ \rightarrow xs'$$

$$G \vdash \text{Norm } s \text{ --In1r (c1 Finally c2)} \succ \rightarrow xs'$$

$$G \vdash \text{Norm } s \text{ --In1r (Throw } e) \succ \rightarrow xs'$$

$$G \vdash \text{Norm } s \text{ --In1l (NewC } C) \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In1l (New } T[e]) \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In1l (Ass } va \ e) \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In1r (Try c1 Catch(tn vn) c2)} \succ \rightarrow xs'$$

$$G \vdash \text{Norm } s \text{ --In2 } (\{accC, statDeclC, stat\} e..fn) \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In2 } (e1.[e2]) \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In1l } (\{accC, statT, mode\} e.mn(\{pT\}p)) \succ \rightarrow vs'$$

$$G \vdash \text{Norm } s \text{ --In1r (Init } C) \succ \rightarrow xs'$$

**declare** *not-None-eq* [*simp*]

**declare** *split-paired-All* [*simp*] *split-paired-Ex* [*simp*]



$\langle ML \rangle$

**declare** *split-if* [split] *split-if-asm* [split]  
*option.split* [split] *option.split-asm* [split]

**lemma** *eval-Inj-elim*:

$G \vdash s -t \rightarrow (w, s')$   
 $\implies \text{case } t \text{ of}$   
      $In1 \text{ } ec \Rightarrow (\text{case } ec \text{ of}$   
          $Inl \text{ } e \Rightarrow (\exists v. w = In1 \text{ } v)$   
          $| Inr \text{ } c \Rightarrow w = \Diamond)$   
      $| In2 \text{ } e \Rightarrow (\exists v. w = In2 \text{ } v)$   
      $| In3 \text{ } e \Rightarrow (\exists v. w = In3 \text{ } v)$   
 $\langle \text{proof} \rangle$

The following simplification procedures set up the proper injections of terms and their corresponding values in the evaluation relation: E.g. an expression (injection *In1l* into terms) always evaluates to ordinary values (injection *In1* into generalised values *vals*).

$\langle ML \rangle$

**declare** *halloc.Abrupt* [intro!] *eval.Abrupt* [intro!] *AbruptIs* [intro!]

*Callee, InsInitE, InsInitV, FinA* are only used in smallstep semantics, not in the bigstep semantics. So there is no valid evaluation of these terms

**lemma** *eval-Callee*:  $G \vdash \text{Norm } s - \text{Callee } l \text{ } e -\succ v \rightarrow s' = \text{False}$   
 $\langle \text{proof} \rangle$

**lemma** *eval-InsInitE*:  $G \vdash \text{Norm } s - \text{InsInitE } c \text{ } e -\succ v \rightarrow s' = \text{False}$   
 $\langle \text{proof} \rangle$

**lemma** *eval-InsInitV*:  $G \vdash \text{Norm } s - \text{InsInitV } c \text{ } w -\succ v \rightarrow s' = \text{False}$   
 $\langle \text{proof} \rangle$

**lemma** *eval-FinA*:  $G \vdash \text{Norm } s - \text{FinA } a \text{ } c \rightarrow s' = \text{False}$   
 $\langle \text{proof} \rangle$

**lemma** *eval-no-abrupt-lemma*:

$\bigwedge s \text{ } s'. G \vdash s -t \rightarrow (w, s') \implies \text{normal } s' \longrightarrow \text{normal } s$   
 $\langle \text{proof} \rangle$

**lemma** *eval-no-abrupt*:

$G \vdash (x, s) -t \rightarrow (w, \text{Norm } s') =$   
 $(x = \text{None} \wedge G \vdash \text{Norm } s -t \rightarrow (w, \text{Norm } s'))$   
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

**lemma** *eval-abrupt-lemma*:

$G \vdash s -t \rightarrow (v, s') \implies \text{abrupt } s = \text{Some } xc \longrightarrow s' = s \wedge v = \text{arbitrary3 } t$   
 $\langle \text{proof} \rangle$

**lemma** *eval-abrupt*:

$$\begin{aligned} & G \vdash (\text{Some } xc, s) -t \succ \rightarrow (w, s') = \\ & (s' = (\text{Some } xc, s) \wedge w = \text{arbitrary3 } t \wedge \\ & G \vdash (\text{Some } xc, s) -t \succ \rightarrow (\text{arbitrary3 } t, (\text{Some } xc, s))) \\ & \langle \text{proof} \rangle \end{aligned}$$

$\langle ML \rangle$

**lemma** *LitI*:  $G \vdash s -\text{Lit } v \succ (\text{if normal } s \text{ then } v \text{ else arbitrary}) \rightarrow s$

$\langle \text{proof} \rangle$

**lemma** *SkipI* [*intro!*]:  $G \vdash s -\text{Skip} \rightarrow s$

$\langle \text{proof} \rangle$

**lemma** *ExprI*:  $G \vdash s -e \succ v \rightarrow s' \implies G \vdash s -\text{Expr } e \rightarrow s'$

$\langle \text{proof} \rangle$

**lemma** *CompI*:  $\llbracket G \vdash s -c1 \rightarrow s1; G \vdash s1 -c2 \rightarrow s2 \rrbracket \implies G \vdash s -c1;; c2 \rightarrow s2$

$\langle \text{proof} \rangle$

**lemma** *CondI*:

$$\begin{aligned} & \bigwedge s1. \llbracket G \vdash s -e \succ b \rightarrow s1; G \vdash s1 -(\text{if the-Bool } b \text{ then } e1 \text{ else } e2) \succ v \rightarrow s2 \rrbracket \implies \\ & G \vdash s -e ? e1 : e2 \succ (\text{if normal } s1 \text{ then } v \text{ else arbitrary}) \rightarrow s2 \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *IfI*:  $\llbracket G \vdash s -e \succ v \rightarrow s1; G \vdash s1 -(\text{if the-Bool } v \text{ then } c1 \text{ else } c2) \rightarrow s2 \rrbracket$

$\implies G \vdash s -\text{If}(e) \ c1 \ \text{Else } c2 \rightarrow s2$

$\langle \text{proof} \rangle$

**lemma** *MethdI*:  $G \vdash s -\text{body } G \ C \ \text{sig} \succ v \rightarrow s'$

$\implies G \vdash s -\text{Methd } C \ \text{sig} \succ v \rightarrow s'$

$\langle \text{proof} \rangle$

**lemma** *eval-Call*:

$$\begin{aligned} & \llbracket G \vdash \text{Norm } s0 -e \succ a' \rightarrow s1; G \vdash s1 -ps \dot{=} \succ pvs \rightarrow s2; \\ & D = \text{invocation-declclass } G \ \text{mode } (\text{store } s2) \ a' \ \text{statT } (\llbracket \text{name} = mn, \text{parTs} = pTs \rrbracket); \\ & s3 = \text{init-lvars } G \ D \ (\llbracket \text{name} = mn, \text{parTs} = pTs \rrbracket) \ \text{mode } a' \ pvs \ s2; \\ & s3' = \text{check-method-access } G \ \text{accC} \ \text{statT} \ \text{mode } (\llbracket \text{name} = mn, \text{parTs} = pTs \rrbracket) \ a' \ s3; \\ & G \vdash s3' -\text{Methd } D \ (\llbracket \text{name} = mn, \text{parTs} = pTs \rrbracket) \succ v \rightarrow s4; \\ & s4' = \text{restore-lvars } s2 \ s4 \rrbracket \implies \\ & G \vdash \text{Norm } s0 -\{\text{accC}, \text{statT}, \text{mode}\} e \cdot mn(\{pTs\} ps) \succ v \rightarrow s4' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *eval-Init*:

$\llbracket \text{if inited } C \ (\text{globs } s0) \text{ then } s3 = \text{Norm } s0$

$\text{else } G \vdash \text{Norm } (\text{init-class-obj } G \ C \ s0)$

$-(\text{if } C = \text{Object} \text{ then } \text{Skip} \text{ else } \text{Init } (\text{super } (\text{the } (\text{class } G \ C)))) \rightarrow s1 \wedge$

$G \vdash \text{set-lvars empty } s1 \text{ } \neg (\text{init } (\text{the } (\text{class } G \ C))) \rightarrow s2 \wedge$   
 $s3 = \text{restore-lvars } s1 \ s2 \parallel \implies$   
 $G \vdash \text{Norm } s0 \text{ } \neg \text{Init } C \rightarrow s3$   
 $\langle \text{proof} \rangle$

**lemma** *init-done*:  $\text{initd } C \ s \implies G \vdash s \text{ } \neg \text{Init } C \rightarrow s$   
 $\langle \text{proof} \rangle$

**lemma** *eval-StatRef*:  
 $G \vdash s \text{ } \neg \text{StatRef } rt \neg \succ (\text{if abrupt } s = \text{None then Null else arbitrary}) \rightarrow s$   
 $\langle \text{proof} \rangle$

**lemma** *SkipD* [*dest!*]:  $G \vdash s \text{ } \neg \text{Skip} \rightarrow s' \implies s' = s$   
 $\langle \text{proof} \rangle$

**lemma** *Skip-eq* [*simp*]:  $G \vdash s \text{ } \neg \text{Skip} \rightarrow s' = (s = s')$   
 $\langle \text{proof} \rangle$

**lemma** *init-retains-locals* [*rule-format (no-asm)*]:  $G \vdash s \text{ } \neg t \succ \rightarrow (w, s') \implies$   
 $(\forall C. t = \text{In1r } (\text{Init } C) \longrightarrow \text{locals } (\text{store } s) = \text{locals } (\text{store } s'))$   
 $\langle \text{proof} \rangle$

**lemma** *halloc-xcpt* [*dest!*]:  
 $\bigwedge s'. G \vdash (\text{Some } xc, s) \text{ } \neg \text{halloc } oi \succ a \rightarrow s' \implies s' = (\text{Some } xc, s)$   
 $\langle \text{proof} \rangle$

**lemma** *eval-Methd*:  
 $G \vdash s \text{ } \neg \text{In1l}(\text{body } G \ C \ sig) \succ \rightarrow (w, s')$   
 $\implies G \vdash s \text{ } \neg \text{In1l}(\text{Methd } C \ sig) \succ \rightarrow (w, s')$   
 $\langle \text{proof} \rangle$

**lemma** *eval-Body*:  $\parallel G \vdash \text{Norm } s0 \text{ } \neg \text{Init } D \rightarrow s1; G \vdash s1 \text{ } \neg c \rightarrow s2;$   
 $\text{res} = \text{the } (\text{locals } (\text{store } s2) \text{ Result});$   
 $s3 = (\text{if } (\exists l. \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Break } l))) \vee$   
 $\text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Cont } l)))$   
 $\text{then abupd } (\lambda x. \text{Some } (\text{Error CrossMethodJump})) \ s2$   
 $\text{else } s2);$   
 $s4 = \text{abupd } (\text{absorb Ret}) \ s3 \parallel \implies$   
 $G \vdash \text{Norm } s0 \text{ } \neg \text{Body } D \ c \neg \succ \text{res} \rightarrow s4$   
 $\langle \text{proof} \rangle$

**lemma** *eval-binop-arg2-indep*:  
 $\neg \text{need-second-arg binop } v1 \implies \text{eval-binop binop } v1 \ x = \text{eval-binop binop } v1 \ y$   
 $\langle \text{proof} \rangle$

**lemma** *eval-BinOp-arg2-indepI*:

**assumes** *eval-e1*:  $G \vdash \text{Norm } s0 \text{ } -e1 \text{ } \rightarrow v1 \rightarrow s1$  **and**

*no-need*:  $\neg \text{need-second-arg binop } v1$

**shows**  $G \vdash \text{Norm } s0 \text{ } -\text{BinOp binop } e1 \text{ } e2 \text{ } \rightarrow (eval\text{-binop binop } v1 \text{ } v2) \rightarrow s1$   
**(is ?EvalBinOp v2)**

$\langle proof \rangle$

**single valued**

**lemma** *unique-halloc* [rule-format (no-asm)]:

$\bigwedge s \text{ as } as'. (s, oi, as) \in \text{halloc } G \implies (s, oi, as') \in \text{halloc } G \longrightarrow as' = as$

$\langle proof \rangle$

**lemma** *single-valued-halloc*:

*single-valued*  $\{((s, oi), (a, s')). G \vdash s \text{ } -\text{halloc } oi \text{ } \rightarrow a \rightarrow s'\}$

$\langle proof \rangle$

**lemma** *unique-sxalloc* [rule-format (no-asm)]:

$\bigwedge s \text{ s}'. G \vdash s \text{ } -\text{sxalloc} \rightarrow s' \implies G \vdash s \text{ } -\text{sxalloc} \rightarrow s'' \longrightarrow s'' = s'$

$\langle proof \rangle$

**lemma** *single-valued-sxalloc*: *single-valued*  $\{(s, s'). G \vdash s \text{ } -\text{sxalloc} \rightarrow s'\}$

$\langle proof \rangle$

**lemma** *split-pairD*:  $(x, y) = p \implies x = \text{fst } p \ \& \ y = \text{snd } p$

$\langle proof \rangle$

**lemma** *unique-eval* [rule-format (no-asm)]:

$G \vdash s \text{ } -t \text{ } \rightarrow ws \implies (\forall ws'. G \vdash s \text{ } -t \text{ } \rightarrow ws' \longrightarrow ws' = ws)$

$\langle proof \rangle$

**lemma** *single-valued-eval*:

*single-valued*  $\{((s, t), vs'). G \vdash s \text{ } -t \text{ } \rightarrow vs'\}$

$\langle proof \rangle$

**end**

## Chapter 16

### Example

### 43 Example Bali program

**theory** *Example* **imports** *Eval WellForm* **begin**

The following example Bali program includes:

- class and interface declarations with inheritance, hiding of fields, overriding of methods (with refined result type), array type,
- method call (with dynamic binding), parameter access, return expressions,
- expression statements, sequential composition, literal values, local assignment, local access, field assignment, type cast,
- exception generation and propagation, try and catch statement, throw statement
- instance creation and (default) static initialization

```
package java_lang

public interface HasFoo {
  public Base foo(Base z);
}

public class Base implements HasFoo {
  static boolean arr[] = new boolean[2];
  public HasFoo vee;
  public Base foo(Base z) {
    return z;
  }
}

public class Ext extends Base {
  public int vee;
  public Ext foo(Base z) {
    ((Ext)z).vee = 1;
    return null;
  }
}

public class Main {
  public static void main(String args[]) throws Throwable {
    Base e = new Ext();
    try {e.foo(null); }
    catch(NullPointerException z) {
      while(Ext.arr[2]) ;
    }
  }
}
```

**declare** *widen.null* [*intro*]

**lemma** *wf-fdecl-def2*:  $\bigwedge fd. wf-fdecl\ G\ P\ fd = is-acc-type\ G\ P\ (type\ (snd\ fd))$   
*<proof>*

**declare** *wf-fdecl-def2* [*iff*]

**type and expression names**

**datatype** *tnam-* = *HasFoo-* | *Base-* | *Ext-* | *Main-*

**datatype** *vnam-* = *arr-* | *vee-* | *z-* | *e-*

**datatype** *label-* = *lab1-*

**consts**

*tnam-* :: *tnam-*  $\Rightarrow$  *tnam*

*vnam-* :: *vnam-*  $\Rightarrow$  *vname*

*label-* :: *label-*  $\Rightarrow$  *label*

**axioms**

*inj-tnam-* [simp]: (*tnam-* *x* = *tnam-* *y*) = (*x* = *y*)

*inj-vnam-* [simp]: (*vnam-* *x* = *vnam-* *y*) = (*x* = *y*)

*inj-label-* [simp]: (*label-* *x* = *label-* *y*) = (*x* = *y*)

*surj-tnam-*:  $\exists m. n = \text{tnam- } m$

*surj-vnam-*:  $\exists m. n = \text{vnam- } m$

*surj-label-*:  $\exists m. n = \text{label- } m$

**syntax**

*HasFoo* :: *qtname*

*Base* :: *qtname*

*Ext* :: *qtname*

*Main* :: *qtname*

*arr* :: *ename*

*vee* :: *ename*

*z* :: *ename*

*e* :: *ename*

*lab1* :: *label*

**translations**

*HasFoo* == ( $\text{pid}=\text{java-lang}, \text{tid}=\text{TName } (\text{tnam- } \text{HasFoo-})$ )

*Base* == ( $\text{pid}=\text{java-lang}, \text{tid}=\text{TName } (\text{tnam- } \text{Base-})$ )

*Ext* == ( $\text{pid}=\text{java-lang}, \text{tid}=\text{TName } (\text{tnam- } \text{Ext-})$ )

*Main* == ( $\text{pid}=\text{java-lang}, \text{tid}=\text{TName } (\text{tnam- } \text{Main-})$ )

*arr* == (*vnam-* *arr-*)

*vee* == (*vnam-* *vee-*)

*z* == (*vnam-* *z-*)

*e* == (*vnam-* *e-*)

*lab1* == *label-* *lab1-*

**lemma** *neq-Base-Object* [simp]: *Base*  $\neq$  *Object*  
 <proof>

**lemma** *neq-Ext-Object* [simp]: *Ext*  $\neq$  *Object*  
 <proof>

**lemma** *neq-Main-Object* [simp]: *Main*  $\neq$  *Object*  
 <proof>

**lemma** *neq-Base-SXcpt* [simp]: *Base* ≠ *SXcpt xn*  
 ⟨proof⟩

**lemma** *neq-Ext-SXcpt* [simp]: *Ext* ≠ *SXcpt xn*  
 ⟨proof⟩

**lemma** *neq-Main-SXcpt* [simp]: *Main* ≠ *SXcpt xn*  
 ⟨proof⟩

## classes and interfaces

### defs

*Object-mdecls-def*: *Object-mdecls* ≡ []  
*SXcpt-mdecls-def*: *SXcpt-mdecls* ≡ []

### consts

*foo* :: *mname*

### constdefs

*foo-sig* :: *sig*  
*foo-sig* ≡ (⟦name=foo,parTs=[Class Base]⟧)  
  
*foo-mhead* :: *mhead*  
*foo-mhead* ≡ (⟦access=Public,static=False,pars=[z],resT=Class Base⟧)

### constdefs

*Base-foo* :: *mdecl*  
*Base-foo* ≡ (*foo-sig*, (⟦access=Public,static=False,pars=[z],resT=Class Base,  
 mbody=(⟦lcls=[],stmt=Return (!!z)⟧)⟧))

### constdefs

*Ext-foo* :: *mdecl*  
*Ext-foo* ≡ (*foo-sig*,  
 (⟦access=Public,static=False,pars=[z],resT=Class Ext,  
 mbody=(⟦lcls=[  
 ,stmt=Expr({Ext,Ext,False} Cast (Class Ext) (!!z)..vee :=  
 Lit (Intg 1)) ;;  
 Return (Lit Null)⟧)  
 ⟧))

### constdefs

*arr-viewed-from* :: *qtname* ⇒ *qtname* ⇒ *var*  
*arr-viewed-from accC C* ≡ {*accC*,*Base*,*True*}StatRef (ClassT C)..*arr*

*BaseCl* :: *class*  
*BaseCl* ≡ (⟦access=Public,  
 cfields=[(arr, (⟦access=Public,static=True,type=PrimT Boolean.[]⟧)),  
 (vee, (⟦access=Public,static=False,type=Iface HasFoo []⟧)),  
 methods=[*Base-foo*],  
 init=Expr(arr-viewed-from Base Base  
 :=New (PrimT Boolean)[Lit (Intg 2)]),  
 super=Object,



*superIfs*=[*HasFoo*])

*ExtCl* :: class

*ExtCl* ≡ (|*access*=*Public*,  
           *cfields*=[(*vee*, (|*access*=*Public*,*static*=*False*,*type*= *PrimT Integer*))],  
           *methods*=[*Ext-foo*],  
           *init*=*Skip*,  
           *super*=*Base*,  
           *superIfs*=[])

*MainCl* :: class

*MainCl* ≡ (|*access*=*Public*,  
           *cfields*=[],  
           *methods*=[],  
           *init*=*Skip*,  
           *super*=*Object*,  
           *superIfs*=[])

### constdefs

*HasFooInt* :: iface

*HasFooInt* ≡ (|*access*=*Public*,*imethods*=[(*foo-sig*, *foo-mhead*)],*isuperIfs*=[])

*Ifaces* ::idecl list

*Ifaces* ≡ [(*HasFoo*,*HasFooInt*)]

*Classes* ::cdecl list

*Classes* ≡ [(*Base*,*BaseCl*),(*Ext*,*ExtCl*),(*Main*,*MainCl*)]@*standard-classes*

**lemmas** *table-classes-defs* =

*Classes-def standard-classes-def ObjectC-def SXcptC-def*

**lemma** *table-ifaces* [simp]: *table-of Ifaces* = *empty*(*HasFoo*→*HasFooInt*)

⟨*proof*⟩

**lemma** *table-classes-Object* [simp]:

*table-of Classes Object* = *Some* (|*access*=*Public*,*cfields*=[]  
                                   ,*methods*=*Object-mdecls*  
                                   ,*init*=*Skip*,*super*=*arbitrary*,*superIfs*=[])

⟨*proof*⟩

**lemma** *table-classes-SXcpt* [simp]:

*table-of Classes (SXcpt xn)*  
 = *Some* (|*access*=*Public*,*cfields*=[],*methods*=*SXcpt-mdecls*,  
           *init*=*Skip*,  
           *super*=if *xn* = *Throwable* then *Object* else *SXcpt Throwable*,  
           *superIfs*=[])

⟨*proof*⟩

**lemma** *table-classes-HasFoo* [simp]: *table-of Classes HasFoo* = *None*

⟨*proof*⟩

**lemma** *table-classes-Base* [simp]: *table-of Classes Base* = *Some BaseCl*

$\langle \text{proof} \rangle$

**lemma** *table-classes-Ext* [simp]: *table-of Classes Ext = Some ExtCl*

$\langle \text{proof} \rangle$

**lemma** *table-classes-Main* [simp]: *table-of Classes Main = Some MainCl*

$\langle \text{proof} \rangle$

## program

### syntax

*tprg* :: *prog*

### translations

*tprg* == (*ifaces=Ifaces, classes=Classes*)

### constdefs

*test* :: (*ty*)*list*  $\Rightarrow$  *stmt*

*test* *pTs*  $\equiv$  *e::=NewC Ext;;*

*Try Expr*(*{Main, ClassT Base, IntVir}*!!*e.foo*(*{pTs}*[*Lit Null*]))

*Catch*((*SXcpt NullPointer*) *z*)

(*lab1* • *While*(*Acc*

(*Acc* (*arr-viewed-from Main Ext*).[*Lit* (*Intg 2*)])) *Skip*)

## well-structuredness

**lemma** *not-Object-subcls-any* [elim!]: (*Object, C*)  $\in$  (*subcls1 tprg*)<sup>+</sup>  $\Longrightarrow$  *R*

$\langle \text{proof} \rangle$

**lemma** *not-Throwable-subcls-SXcpt* [elim!]:

(*SXcpt Throwable, SXcpt xn*)  $\in$  (*subcls1 tprg*)<sup>+</sup>  $\Longrightarrow$  *R*

$\langle \text{proof} \rangle$

**lemma** *not-SXcpt-n-subcls-SXcpt-n* [elim!]:

(*SXcpt xn, SXcpt xn*)  $\in$  (*subcls1 tprg*)<sup>+</sup>  $\Longrightarrow$  *R*

$\langle \text{proof} \rangle$

**lemma** *not-Base-subcls-Ext* [elim!]: (*Base, Ext*)  $\in$  (*subcls1 tprg*)<sup>+</sup>  $\Longrightarrow$  *R*

$\langle \text{proof} \rangle$

**lemma** *not-TName-n-subcls-TName-n* [rule-format (no-asm), elim!]:

((*pid=java-lang, tid=TName tn*), (*pid=java-lang, tid=TName tn*))

$\in$  (*subcls1 tprg*)<sup>+</sup>  $\longrightarrow$  *R*

$\langle \text{proof} \rangle$

**lemma** *ws-idecl-HasFoo*: *ws-idecl tprg HasFoo* []

$\langle \text{proof} \rangle$

**lemma** *ws-cdecl-Object*: *ws-cdecl tprg Object any*

$\langle \text{proof} \rangle$

**lemma** *ws-cdecl-Throwable*: *ws-cdecl tprg (SXcpt Throwable) Object*  
 ⟨proof⟩

**lemma** *ws-cdecl-SXcpt*: *ws-cdecl tprg (SXcpt xn) (SXcpt Throwable)*  
 ⟨proof⟩

**lemma** *ws-cdecl-Base*: *ws-cdecl tprg Base Object*  
 ⟨proof⟩

**lemma** *ws-cdecl-Ext*: *ws-cdecl tprg Ext Base*  
 ⟨proof⟩

**lemma** *ws-cdecl-Main*: *ws-cdecl tprg Main Object*  
 ⟨proof⟩

**lemmas** *ws-cdecls = ws-cdecl-SXcpt ws-cdecl-Object ws-cdecl-Throwable*  
*ws-cdecl-Base ws-cdecl-Ext ws-cdecl-Main*

**declare** *not-Object-subcls-any* [rule del]  
*not-Throwable-subcls-SXcpt* [rule del]  
*not-SXcpt-n-subcls-SXcpt-n* [rule del]  
*not-Base-subcls-Ext* [rule del] *not-TName-n-subcls-TName-n* [rule del]

**lemma** *ws-idecl-all*:  
 $G = \text{tprg} \implies (\forall (I, i) \in \text{set Ifaces}. \text{ws-idecl } G \ I \ (\text{isuperIfs } i))$   
 ⟨proof⟩

**lemma** *ws-cdecl-all*:  $G = \text{tprg} \implies (\forall (C, c) \in \text{set Classes}. \text{ws-cdecl } G \ C \ (\text{super } c))$   
 ⟨proof⟩

**lemma** *ws-tprg*: *ws-prog tprg*  
 ⟨proof⟩

### **misc program properties (independent of well-structuredness)**

**lemma** *single-iface* [simp]: *is-iface tprg I = (I = HasFoo)*  
 ⟨proof⟩

**lemma** *empty-subint1* [simp]: *subint1 tprg = {}*  
 ⟨proof⟩

**lemma** *unique-ifaces*: *unique Ifaces*  
 ⟨proof⟩

**lemma** *unique-classes*: *unique Classes*  
 ⟨proof⟩

**lemma** *SXcpt-subcls-Throwable* [simp]:  $\text{tprg} \vdash \text{SXcpt } xn \preceq_C \text{ SXcpt Throwable}$   
 ⟨proof⟩

**lemma** *Ext-subclseq-Base* [simp]:  $\text{tprg} \vdash \text{Ext} \preceq_C \text{ Base}$   
 ⟨proof⟩

**lemma** *Ext-subcls-Base* [simp]:  $\text{tprg} \vdash \text{Ext} \prec_C \text{ Base}$   
 ⟨proof⟩

### fields and method lookup

**lemma** *fields-tprg-Object* [simp]:  $\text{DeclConcepts.fields tprg Object} = []$   
 ⟨proof⟩

**lemma** *fields-tprg-Throwable* [simp]:  
 $\text{DeclConcepts.fields tprg (SXcpt Throwable)} = []$   
 ⟨proof⟩

**lemma** *fields-tprg-SXcpt* [simp]:  $\text{DeclConcepts.fields tprg (SXcpt } xn) = []$   
 ⟨proof⟩

**lemmas** *fields-rec-* = *fields-rec* [OF - ws-tprg]

**lemma** *fields-Base* [simp]:  
 $\text{DeclConcepts.fields tprg Base}$   
 $= [((\text{arr}, \text{Base}), (\text{access}=\text{Public}, \text{static}=\text{True}, \text{type}=\text{PrimT Boolean}, [])),$   
 $((\text{vee}, \text{Base}), (\text{access}=\text{Public}, \text{static}=\text{False}, \text{type}=\text{Iface HasFoo}, []))]$   
 ⟨proof⟩

**lemma** *fields-Ext* [simp]:  
 $\text{DeclConcepts.fields tprg Ext}$   
 $= [((\text{vee}, \text{Ext}), (\text{access}=\text{Public}, \text{static}=\text{False}, \text{type}=\text{PrimT Integer}))]$   
 $@ \text{DeclConcepts.fields tprg Base}$   
 ⟨proof⟩

**lemmas** *imethds-rec-* = *imethds-rec* [OF - ws-tprg]

**lemmas** *methd-rec-* = *methd-rec* [OF - ws-tprg]

**lemma** *imethds-HasFoo* [simp]:  
 $\text{imethds tprg HasFoo} = \text{o2s} \circ \text{empty}(\text{foo-sig} \mapsto (\text{HasFoo}, \text{foo-mhead}))$   
 ⟨proof⟩

**lemma** *methd-tprg-Object* [simp]:  $\text{methd tprg Object} = \text{empty}$   
 ⟨proof⟩

**lemma** *methd-Base* [simp]:  
 $\text{methd tprg Base} = \text{table-of } [(\lambda(s, m). (s, \text{Base}, m)) \text{ Base-foo}]$   
 ⟨proof⟩

**lemma** *memberid-Base-foo-simp* [simp]:  
 $\text{memberid } (\text{mdecl Base-foo}) = \text{mid foo-sig}$   
 ⟨proof⟩

**lemma** *memberid-Ext-foo-simp* [simp]:  
 $\text{memberid } (\text{mdecl Ext-foo}) = \text{mid foo-sig}$   
 ⟨proof⟩

**lemma** *Base-declares-foo*:  
 $\text{tprg} \vdash \text{mdecl Base-foo declared-in Base}$   
 ⟨proof⟩

**lemma** *foo-sig-not-undeclared-in-Base*:  
 $\neg \text{tprg} \vdash \text{mid foo-sig undeclared-in Base}$   
 ⟨proof⟩

**lemma** *Ext-declares-foo*:  
 $\text{tprg} \vdash \text{mdecl Ext-foo declared-in Ext}$   
 ⟨proof⟩

**lemma** *foo-sig-not-undeclared-in-Ext*:  
 $\neg \text{tprg} \vdash \text{mid foo-sig undeclared-in Ext}$   
 ⟨proof⟩

**lemma** *Base-foo-not-inherited-in-Ext*:  
 $\neg \text{tprg} \vdash \text{Ext inherits } (\text{Base}, \text{mdecl Base-foo})$   
 ⟨proof⟩

**lemma** *Ext-method-inheritance*:  
 $\text{filter-tab } (\lambda \text{sig } m. \text{tprg} \vdash \text{Ext inherits method sig } m)$   
 $(\text{empty}(\text{fst } ((\lambda(s, m). (s, \text{Base}, m)) \text{Base-foo}) \mapsto$   
 $\text{snd } ((\lambda(s, m). (s, \text{Base}, m)) \text{Base-foo})))$   
 $= \text{empty}$   
 ⟨proof⟩

**lemma** *methd-Ext* [simp]:  $\text{methd tprg Ext} =$   
 $\text{table-of } [(\lambda(s, m). (s, \text{Ext}, m)) \text{Ext-foo}]$   
 ⟨proof⟩

## accessibility

**lemma** *classesDefined*:  
 $\llbracket \text{class tprg } C = \text{Some } c; C \neq \text{Object} \rrbracket \implies \exists \text{ sc. class tprg } (\text{super } c) = \text{Some } \text{sc}$   
 ⟨proof⟩

**lemma** *superclassesBase* [simp]:  $\text{superclasses tprg Base} = \{\text{Object}\}$   
 ⟨proof⟩

**lemma** *superclassesExt* [simp]: *superclasses tprg Ext*={*Base, Object*}  
 ⟨*proof*⟩

**lemma** *superclassesMain* [simp]: *superclasses tprg Main*={*Object*}  
 ⟨*proof*⟩

**lemma** *HasFoo-accessible*[simp]:*tprg*⊢(*Iface HasFoo*) *accessible-in P*  
 ⟨*proof*⟩

**lemma** *HasFoo-is-acc-iface*[simp]: *is-acc-iface tprg P HasFoo*  
 ⟨*proof*⟩

**lemma** *HasFoo-is-acc-type*[simp]: *is-acc-type tprg P (Iface HasFoo)*  
 ⟨*proof*⟩

**lemma** *Base-accessible*[simp]:*tprg*⊢(*Class Base*) *accessible-in P*  
 ⟨*proof*⟩

**lemma** *Base-is-acc-class*[simp]: *is-acc-class tprg P Base*  
 ⟨*proof*⟩

**lemma** *Base-is-acc-type*[simp]: *is-acc-type tprg P (Class Base)*  
 ⟨*proof*⟩

**lemma** *Ext-accessible*[simp]:*tprg*⊢(*Class Ext*) *accessible-in P*  
 ⟨*proof*⟩

**lemma** *Ext-is-acc-class*[simp]: *is-acc-class tprg P Ext*  
 ⟨*proof*⟩

**lemma** *Ext-is-acc-type*[simp]: *is-acc-type tprg P (Class Ext)*  
 ⟨*proof*⟩

**lemma** *accmethd-tprg-Object* [simp]: *accmethd tprg S Object* = *empty*  
 ⟨*proof*⟩

**lemma** *snd-special-simp*: *snd ((λ(*s, m*). (*s, a, m*)) *x*)* = (*a, snd x*)  
 ⟨*proof*⟩

**lemma** *fst-special-simp*: *fst ((λ(*s, m*). (*s, a, m*)) *x*)* = *fst x*  
 ⟨*proof*⟩

**lemma** *foo-sig-undeclared-in-Object*:  
*tprg*⊢*mid foo-sig undeclared-in Object*

$\langle \text{proof} \rangle$

**lemma** *unique-sig-Base-foo*:

$\text{tprg} \vdash \text{mdecl } (\text{sig}, \text{snd Base-foo}) \text{ declared-in Base} \implies \text{sig} = \text{foo-sig}$   
 $\langle \text{proof} \rangle$

**lemma** *Base-foo-no-override*:

$\text{tprg}, \text{sig} \vdash (\text{Base}, (\text{snd Base-foo})) \text{ overrides old} \implies P$   
 $\langle \text{proof} \rangle$

**lemma** *Base-foo-no-stat-override*:

$\text{tprg}, \text{sig} \vdash (\text{Base}, (\text{snd Base-foo})) \text{ overrides}_S \text{ old} \implies P$   
 $\langle \text{proof} \rangle$

**lemma** *Base-foo-no-hide*:

$\text{tprg}, \text{sig} \vdash (\text{Base}, (\text{snd Base-foo})) \text{ hides old} \implies P$   
 $\langle \text{proof} \rangle$

**lemma** *Ext-foo-no-hide*:

$\text{tprg}, \text{sig} \vdash (\text{Ext}, (\text{snd Ext-foo})) \text{ hides old} \implies P$   
 $\langle \text{proof} \rangle$

**lemma** *unique-sig-Ext-foo*:

$\text{tprg} \vdash \text{mdecl } (\text{sig}, \text{snd Ext-foo}) \text{ declared-in Ext} \implies \text{sig} = \text{foo-sig}$   
 $\langle \text{proof} \rangle$

**lemma** *Ext-foo-override*:

$\text{tprg}, \text{sig} \vdash (\text{Ext}, (\text{snd Ext-foo})) \text{ overrides old}$   
 $\implies \text{old} = (\text{Base}, (\text{snd Base-foo}))$   
 $\langle \text{proof} \rangle$

**lemma** *Ext-foo-stat-override*:

$\text{tprg}, \text{sig} \vdash (\text{Ext}, (\text{snd Ext-foo})) \text{ overrides}_S \text{ old}$   
 $\implies \text{old} = (\text{Base}, (\text{snd Base-foo}))$   
 $\langle \text{proof} \rangle$

**lemma** *Base-foo-member-of-Base*:

$\text{tprg} \vdash (\text{Base}, \text{mdecl Base-foo}) \text{ member-of Base}$   
 $\langle \text{proof} \rangle$

**lemma** *Base-foo-member-in-Base*:

$\text{tprg} \vdash (\text{Base}, \text{mdecl Base-foo}) \text{ member-in Base}$   
 $\langle \text{proof} \rangle$

**lemma** *Ext-foo-member-of-Ext*:

$\text{tprg} \vdash (\text{Ext}, \text{mdecl Ext-foo}) \text{ member-of Ext}$   
 $\langle \text{proof} \rangle$

**lemma** *Ext-foo-member-in-Ext*:

$\text{tprg} \vdash (\text{Ext}, \text{mdecl } \text{Ext-foo}) \text{ member-in } \text{Ext}$   
 $\langle \text{proof} \rangle$

**lemma** *Base-foo-permits-acc*:

$\text{tprg} \vdash (\text{Base}, \text{mdecl } \text{Base-foo}) \text{ in } \text{Base} \text{ permits-acc-from } S$   
 $\langle \text{proof} \rangle$

**lemma** *Base-foo-accessible [simp]*:

$\text{tprg} \vdash (\text{Base}, \text{mdecl } \text{Base-foo}) \text{ of } \text{Base} \text{ accessible-from } S$   
 $\langle \text{proof} \rangle$

**lemma** *Base-foo-dyn-accessible [simp]*:

$\text{tprg} \vdash (\text{Base}, \text{mdecl } \text{Base-foo}) \text{ in } \text{Base} \text{ dyn-accessible-from } S$   
 $\langle \text{proof} \rangle$

**lemma** *accmethd-Base [simp]*:

$\text{accmethd } \text{tprg } S \text{ Base} = \text{methd } \text{tprg } \text{Base}$   
 $\langle \text{proof} \rangle$

**lemma** *Ext-foo-permits-acc*:

$\text{tprg} \vdash (\text{Ext}, \text{mdecl } \text{Ext-foo}) \text{ in } \text{Ext} \text{ permits-acc-from } S$   
 $\langle \text{proof} \rangle$

**lemma** *Ext-foo-accessible [simp]*:

$\text{tprg} \vdash (\text{Ext}, \text{mdecl } \text{Ext-foo}) \text{ of } \text{Ext} \text{ accessible-from } S$   
 $\langle \text{proof} \rangle$

**lemma** *Ext-foo-dyn-accessible [simp]*:

$\text{tprg} \vdash (\text{Ext}, \text{mdecl } \text{Ext-foo}) \text{ in } \text{Ext} \text{ dyn-accessible-from } S$   
 $\langle \text{proof} \rangle$

**lemma** *Ext-foo-overrides-Base-foo*:

$\text{tprg} \vdash (\text{Ext}, \text{Ext-foo}) \text{ overrides } (\text{Base}, \text{Base-foo})$   
 $\langle \text{proof} \rangle$

**lemma** *accmethd-Ext [simp]*:

$\text{accmethd } \text{tprg } S \text{ Ext} = \text{methd } \text{tprg } \text{Ext}$   
 $\langle \text{proof} \rangle$

**lemma** *cls-Ext*:  $\text{class } \text{tprg } \text{Ext} = \text{Some } \text{ExtCl}$

$\langle \text{proof} \rangle$

**lemma** *dynmethd-Ext-foo*:

$\text{dynmethd } \text{tprg } \text{Base } \text{Ext} (\text{name} = \text{foo}, \text{parTs} = [\text{Class } \text{Base}])$   
 $= \text{Some } (\text{Ext}, \text{snd } \text{Ext-foo})$   
 $\langle \text{proof} \rangle$



**lemma** *Base-fields-accessible*[simp]:  
*accfield* *tprg* *S* *Base*  
 = *table-of*((*map* ( $\lambda((n,d),f).(n,(d,f))$ )) (*DeclConcepts.fields tprg Base*))  
 <proof>

**lemma** *arr-member-of-Base*:  
*tprg*⊢(*Base*, *fdecl* (*arr*,  
     ⊥*access* = *Public*, *static* = *True*, *type* = *PrimT Boolean*.[]))  
     *member-of Base*  
 <proof>

**lemma** *arr-member-in-Base*:  
*tprg*⊢(*Base*, *fdecl* (*arr*,  
     ⊥*access* = *Public*, *static* = *True*, *type* = *PrimT Boolean*.[]))  
     *member-in Base*  
 <proof>

**lemma** *arr-member-of-Ext*:  
*tprg*⊢(*Base*, *fdecl* (*arr*,  
     ⊥*access* = *Public*, *static* = *True*, *type* = *PrimT Boolean*.[]))  
     *member-of Ext*  
 <proof>

**lemma** *arr-member-in-Ext*:  
*tprg*⊢(*Base*, *fdecl* (*arr*,  
     ⊥*access* = *Public*, *static* = *True*, *type* = *PrimT Boolean*.[]))  
     *member-in Ext*  
 <proof>

**lemma** *Ext-fields-accessible*[simp]:  
*accfield* *tprg* *S* *Ext*  
 = *table-of*((*map* ( $\lambda((n,d),f).(n,(d,f))$ )) (*DeclConcepts.fields tprg Ext*))  
 <proof>

**lemma** *arr-Base-dyn-accessible* [simp]:  
*tprg*⊢(*Base*, *fdecl* (*arr*, ⊥*access*=*Public*,*static*=*True* ,*type*=*PrimT Boolean*.[]))  
     *in Base dyn-accessible-from S*  
 <proof>

**lemma** *arr-Ext-dyn-accessible*[simp]:  
*tprg*⊢(*Base*, *fdecl* (*arr*, ⊥*access*=*Public*,*static*=*True* ,*type*=*PrimT Boolean*.[]))  
     *in Ext dyn-accessible-from S*  
 <proof>

**lemma** *array-of-PrimT-acc* [simp]:  
*is-acc-type tprg java-lang* (*PrimT t*.[])  
 <proof>

**lemma** *PrimT-acc* [simp]:  
*is-acc-type tprg java-lang (PrimT t)*  
 ⟨proof⟩

**lemma** *Object-acc* [simp]:  
*is-acc-class tprg java-lang Object*  
 ⟨proof⟩

### well-formedness

**lemma** *wf-HasFoo*: *wf-idecl tprg (HasFoo, HasFooInt)*  
 ⟨proof⟩

**declare** *member-is-static-simp* [simp]  
**declare** *wt.Skip* [rule del] *wt.Init* [rule del]  
 ⟨ML⟩  
**lemmas** *wtIs* = *wt-Call wt-Super wt-FVar wt-StatRef wt-intros*  
**lemmas** *daIs* = *assigned.select-convs da-Skip da-NewC da-Lit da-Super da.intros*  
  
**lemmas** *Base-foo-defs* = *Base-foo-def foo-sig-def foo-mhead-def*  
**lemmas** *Ext-foo-defs* = *Ext-foo-def foo-sig-def*

**lemma** *wf-Base-foo*: *wf-mdecl tprg Base Base-foo*  
 ⟨proof⟩

**lemma** *wf-Ext-foo*: *wf-mdecl tprg Ext Ext-foo*  
 ⟨proof⟩

**declare** *mhead-resTy-simp* [simp add]  
**declare** *member-is-static-simp* [simp add]

**lemma** *wf-BaseC*: *wf-cdecl tprg (Base, BaseCl)*  
 ⟨proof⟩

**lemma** *wf-ExtC*: *wf-cdecl tprg (Ext, ExtCl)*  
 ⟨proof⟩

**lemma** *wf-MainC*: *wf-cdecl tprg (Main, MainCl)*  
 ⟨proof⟩

**lemma** *wf-idecl-all*: *p=tprg  $\implies$  Ball (set Ifaces) (wf-idecl p)*  
 ⟨proof⟩

**lemma** *wf-cdecl-all-standard-classes*:

*Ball* (set standard-classes) (wf-cdecl tprg)  
 ⟨proof⟩

**lemma** *wf-cdecl-all*:  $p = \text{tprg} \implies \text{Ball (set Classes) (wf-cdecl } p)$   
 ⟨proof⟩

**theorem** *wf-tprg*: *wf-prog tprg*  
 ⟨proof⟩

### max spec

**lemma** *appl-methds-Base-foo*:  
*appl-methds tprg S (ClassT Base) (name=foo, parTs=[NT])* =  
 {((ClassT Base, (access=Public,static=False,pars=[z],resT=Class Base))  
 , [Class Base])}  
 ⟨proof⟩

**lemma** *max-spec-Base-foo*: *max-spec tprg S (ClassT Base) (name=foo,parTs=[NT])* =  
 {((ClassT Base, (access=Public,static=False,pars=[z],resT=Class Base))  
 , [Class Base])}  
 ⟨proof⟩

### well-typedness

**lemma** *wt-test*:  $(\text{prg} = \text{tprg}, \text{cls} = \text{Main}, \text{lcl} = \text{empty}(\text{VName } e \mapsto \text{Class Base})) \vdash \text{test } ?pTs :: \checkmark$   
 ⟨proof⟩

### definite assignment

**lemma** *da-test*:  $(\text{prg} = \text{tprg}, \text{cls} = \text{Main}, \text{lcl} = \text{empty}(\text{VName } e \mapsto \text{Class Base}))$   
 $\vdash \{ \} \gg \langle \text{test } ?pTs \rangle \gg (\text{nrm} = \{ \text{VName } e \}, \text{brk} = \lambda l. \text{UNIV})$   
 ⟨proof⟩

### execution

**lemma** *alloc-one*:  $\bigwedge a \text{ obj. } \llbracket \text{the (new-Addr } h) = a; \text{atleast-free } h \text{ (Suc } n) \rrbracket \implies$   
 $\text{new-Addr } h = \text{Some } a \wedge \text{atleast-free } (h(a \mapsto \text{obj})) \text{ } n$   
 ⟨proof⟩

**declare** *fvar-def2* [simp] *avar-def2* [simp] *init-lvars-def2* [simp]  
**declare** *init-obj-def* [simp] *var-tys-def* [simp] *fields-table-def* [simp]  
**declare** *BaseCl-def* [simp] *ExtCl-def* [simp] *Ext-foo-def* [simp]  
*Base-foo-defs* [simp]

⟨ML⟩

**lemmas** *eval-Is* = *eval-Init eval-StatRef AbruptIs eval-intros*

### consts

*a* :: *loc*  
*b* :: *loc*  
*c* :: *loc*

### syntax

*tprg* :: *prog*

*obj-a* :: *obj*  
*obj-b* :: *obj*

```

obj-c :: obj
arr-N :: (vn, val) table
arr-a :: (vn, val) table
globs1 :: globs
globs2 :: globs
globs3 :: globs
globs8 :: globs
locs3 :: locals
locs4 :: locals
locs8 :: locals
s0 :: state
s0' :: state
s9' :: state
s1 :: state
s1' :: state
s2 :: state
s2' :: state
s3 :: state
s3' :: state
s4 :: state
s4' :: state
s6' :: state
s7' :: state
s8 :: state
s8' :: state

```

#### translations

```

tprg == (ifaces=Ifaces, classes=Classes)

obj-a <= (tag=Arr (PrimT Boolean) two
          ,values=empty(Inr 0 ↦ Bool False)(Inr one ↦ Bool False))
obj-b <= (tag=CInst Ext
          ,values=(empty(Inl (vee, Base) ↦ Null
                          (Inl (vee, Ext) ↦ Intg 0)))
obj-c == (tag=CInst (SXcpt NullPointer), values=empty)
arr-N == empty(Inl (arr, Base) ↦ Null)
arr-a == empty(Inl (arr, Base) ↦ Addr a)
globs1 == empty(Inr Ext ↦ (tag=arbitrary, values=empty))
          (Inr Base ↦ (tag=arbitrary, values=arr-N))
          (Inr Object ↦ (tag=arbitrary, values=empty))
globs2 == empty(Inr Ext ↦ (tag=arbitrary, values=empty))
          (Inr Object ↦ (tag=arbitrary, values=empty))
          (Inl a ↦ obj-a)
          (Inr Base ↦ (tag=arbitrary, values=arr-a))
globs3 == globs2(Inl b ↦ obj-b)
globs8 == globs3(Inl c ↦ obj-c)
locs3 == empty(VName e ↦ Addr b)
locs4 == empty(VName z ↦ Null)(Inr() ↦ Addr b)
locs8 == locs3(VName z ↦ Addr c)
s0 == st empty empty
s0' == Norm s0
s1 == st globs1 empty
s1' == Norm s1
s2 == st globs2 empty
s2' == Norm s2
s3 == st globs3 locs3
s3' == Norm s3
s4 == st globs3 locs4

```

```

s4' == Norm s4
s6' == (Some (Xcpt (Std NullPointer)), s4)
s7' == (Some (Xcpt (Std NullPointer)), s3)
s8 == st globs8 locs8
s8' == Norm s8
s9' == (Some (Xcpt (Std IndOutBound)), s8)

```

**syntax** *four::nat*

*tree::nat*

*two ::nat*

*one ::nat*

**translations**

*one* == *Suc 0*

*two* == *Suc one*

*tree* == *Suc two*

*four* == *Suc tree*

**declare** *Pair-eq* [*simp del*]

**lemma** *exec-test*:

```

[[the (new-Addr (heap s1)) = a;
 the (new-Addr (heap ?s2)) = b;
 the (new-Addr (heap ?s3)) = c]] ==>
atleast-free (heap s0) four ==>
tprg|-s0'-test [Class Base]-> ?s9'

```

⟨*proof*⟩

**declare** *Pair-eq* [*simp*]

**end**



## Chapter 17

# Conform

#### 44 Conformance notions for the type soundness proof for Java

**theory** *Conform* **imports** *State* **begin**

design issues:

- lconf allows for (arbitrary) inaccessible values
- "conforms" does not directly imply that the dynamic types of all objects on the heap are indeed existing classes. Yet this can be inferred for all referenced objs.

**types**  $env = prog \times (lname, ty) \text{ table}$

**extension of global store**

**constdefs**

$$\begin{aligned} gext &:: st \Rightarrow st \Rightarrow bool & (-\leq|- & [71,71] \quad 70) \\ s\leq|s' &\equiv \forall r. \forall obj \in globs \ s \ r: \exists obj' \in globs \ s' \ r: tag \ obj' = tag \ obj \end{aligned}$$

For the the proof of type soundness we will need the property that during execution, objects are not lost and moreover retain the values of their tags. So the object store grows conservatively. Note that if we considered garbage collection, we would have to restrict this property to accessible objects.

**lemma** *gext-objD*:

$$\begin{aligned} &\llbracket s\leq|s'; globs \ s \ r = Some \ obj \rrbracket \\ \implies &\exists obj'. globs \ s' \ r = Some \ obj' \wedge tag \ obj' = tag \ obj \\ &\langle proof \rangle \end{aligned}$$

**lemma** *rev-gext-objD*:

$$\begin{aligned} &\llbracket globs \ s \ r = Some \ obj; s\leq|s' \rrbracket \\ \implies &\exists obj'. globs \ s' \ r = Some \ obj' \wedge tag \ obj' = tag \ obj \\ &\langle proof \rangle \end{aligned}$$

**lemma** *init-class-obj-inited*:

$$\begin{aligned} &init-class-obj \ G \ C \ s1\leq|s2 \implies init-ed \ C \ (globs \ s2) \\ &\langle proof \rangle \end{aligned}$$

**lemma** *gext-refl* [*intro!*, *simp*]:  $s\leq|s$

$\langle proof \rangle$

**lemma** *gext-gupd* [*simp*, *elim!*]:  $\bigwedge s. globs \ s \ r = None \implies s\leq|gupd(r \mapsto x)s$

$\langle proof \rangle$

**lemma** *gext-new* [*simp*, *elim!*]:  $\bigwedge s. globs \ s \ r = None \implies s\leq|init-obj \ G \ oi \ r \ s$

$\langle proof \rangle$

**lemma** *gext-trans* [*elim*]:  $\bigwedge X. \llbracket s\leq|s'; s'\leq|s'' \rrbracket \implies s\leq|s''$

$\langle proof \rangle$

**lemma** *gext-upd-gobj* [*intro!*]:  $s\leq|upd-gobj \ r \ n \ v \ s$

$\langle proof \rangle$



**lemma** *gext-cong1* [simp]:  $set\_locals\ l\ s1 \leq |s2 = s1 \leq |s2$   
 $\langle proof \rangle$

**lemma** *gext-cong2* [simp]:  $s1 \leq |set\_locals\ l\ s2 = s1 \leq |s2$   
 $\langle proof \rangle$

**lemma** *gext-lupd1* [simp]:  $lupd(vn \mapsto v)s1 \leq |s2 = s1 \leq |s2$   
 $\langle proof \rangle$

**lemma** *gext-lupd2* [simp]:  $s1 \leq |lupd(vn \mapsto v)s2 = s1 \leq |s2$   
 $\langle proof \rangle$

**lemma** *inited-gext*:  $\llbracket inited\ C\ (globs\ s); s \leq |s \rrbracket \implies inited\ C\ (globs\ s')$   
 $\langle proof \rangle$

## value conformance

### constdefs

$conf :: prog \Rightarrow st \Rightarrow val \Rightarrow ty \Rightarrow bool \quad (-, \vdash :: \preceq - \quad [71, 71, 71, 71]\ 70)$   
 $G, s \vdash v :: \preceq T \equiv \exists T' \in typeof\ (\lambda a. option\_map\ obj\_ty\ (heap\ s\ a))\ v : G \vdash T' \preceq T$

**lemma** *conf-cong* [simp]:  $G, set\_locals\ l\ s \vdash v :: \preceq T = G, s \vdash v :: \preceq T$   
 $\langle proof \rangle$

**lemma** *conf-lupd* [simp]:  $G, lupd(vn \mapsto va)s \vdash v :: \preceq T = G, s \vdash v :: \preceq T$   
 $\langle proof \rangle$

**lemma** *conf-PrimT* [simp]:  $\forall dt. typeof\ dt\ v = Some\ (PrimT\ t) \implies G, s \vdash v :: \preceq PrimT\ t$   
 $\langle proof \rangle$

**lemma** *conf-Boolean*:  $G, s \vdash v :: \preceq PrimT\ Boolean \implies \exists b. v = Bool\ b$   
 $\langle proof \rangle$

**lemma** *conf-litval* [rule-format (no-asm)]:  
 $typeof\ (\lambda a. None)\ v = Some\ T \longrightarrow G, s \vdash v :: \preceq T$   
 $\langle proof \rangle$

**lemma** *conf-Null* [simp]:  $G, s \vdash Null :: \preceq T = G \vdash NT \preceq T$   
 $\langle proof \rangle$

**lemma** *conf-Addr*:  
 $G, s \vdash Addr\ a :: \preceq T = (\exists obj. heap\ s\ a = Some\ obj \wedge G \vdash obj\_ty\ obj \preceq T)$   
 $\langle proof \rangle$

**lemma** *conf-AddrI*:  $\llbracket \text{heap } s \ a = \text{Some } \text{obj}; G \vdash \text{obj-ty } \text{obj} \preceq T \rrbracket \implies G, s \vdash \text{Addr } a :: \preceq T$   
 $\langle \text{proof} \rangle$

**lemma** *defval-conf* [*rule-format* (*no-asm*), *elim*]:  
 $\text{is-type } G \ T \longrightarrow G, s \vdash \text{default-val } T :: \preceq T$   
 $\langle \text{proof} \rangle$

**lemma** *conf-widen* [*rule-format* (*no-asm*), *elim*]:  
 $G \vdash T \preceq T' \implies G, s \vdash x :: \preceq T \longrightarrow \text{ws-prog } G \longrightarrow G, s \vdash x :: \preceq T'$   
 $\langle \text{proof} \rangle$

**lemma** *conf-gext* [*rule-format* (*no-asm*), *elim*]:  
 $G, s \vdash v :: \preceq T \longrightarrow s \leq |s' \longrightarrow G, s \upharpoonright v :: \preceq T$   
 $\langle \text{proof} \rangle$

**lemma** *conf-list-widen* [*rule-format* (*no-asm*)]:  
 $\text{ws-prog } G \implies$   
 $\forall Ts \ Ts'. \text{list-all2 } (\text{conf } G \ s) \ \text{vs } Ts$   
 $\longrightarrow G \vdash Ts[\preceq] \ Ts' \longrightarrow \text{list-all2 } (\text{conf } G \ s) \ \text{vs } Ts'$   
 $\langle \text{proof} \rangle$

**lemma** *conf-RefTD* [*rule-format* (*no-asm*)]:  
 $G, s \vdash a' :: \preceq \text{RefT } T$   
 $\longrightarrow a' = \text{Null} \vee (\exists a \ \text{obj } T'. a' = \text{Addr } a \wedge \text{heap } s \ a = \text{Some } \text{obj} \wedge$   
 $\text{obj-ty } \text{obj} = T' \wedge G \vdash T' \preceq \text{RefT } T)$   
 $\langle \text{proof} \rangle$

## value list conformance

### constdefs

$\text{lconf} :: \text{prog} \Rightarrow \text{st} \Rightarrow ('a, \text{val}) \text{table} \Rightarrow ('a, \text{ty}) \text{table} \Rightarrow \text{bool}$   
 $(-, \vdash -[\preceq]) - [71, 71, 71, 71] \ 70)$   
 $G, s \vdash \text{vs}[\preceq] \ Ts \equiv \forall n. \forall T \in Ts \ n: \exists v \in \text{vs } n: G, s \vdash v :: \preceq T$

**lemma** *lconfD*:  $\llbracket G, s \vdash \text{vs}[\preceq] \ Ts; Ts \ n = \text{Some } T \rrbracket \implies G, s \vdash (\text{the } (\text{vs } n)) :: \preceq T$   
 $\langle \text{proof} \rangle$

**lemma** *lconf-cong* [*simp*]:  $\bigwedge s. G, \text{set-locals } x \ s \vdash l[\preceq] \ L = G, s \vdash l[\preceq] \ L$   
 $\langle \text{proof} \rangle$

**lemma** *lconf-lupd* [*simp*]:  $G, \text{lupd}(vn \mapsto v) \ s \vdash l[\preceq] \ L = G, s \vdash l[\preceq] \ L$   
 $\langle \text{proof} \rangle$

**lemma** *lconf-new*:  $\llbracket L \ vn = \text{None}; G, s \vdash l[\preceq] \ L \rrbracket \implies G, s \vdash l(vn \mapsto v)[\preceq] \ L$

$\langle \text{proof} \rangle$

**lemma** *lconf-upd*:  $\llbracket G, s \vdash l[::\preceq] L; G, s \vdash v :: \preceq T; L \text{ vn} = \text{Some } T \rrbracket \implies$   
 $G, s \vdash l(vn \mapsto v)[::\preceq] L$   
 $\langle \text{proof} \rangle$

**lemma** *lconf-ext*:  $\llbracket G, s \vdash l[::\preceq] L; G, s \vdash v :: \preceq T \rrbracket \implies G, s \vdash l(vn \mapsto v)[::\preceq] L(vn \mapsto T)$   
 $\langle \text{proof} \rangle$

**lemma** *lconf-map-sum* [simp]:  
 $G, s \vdash l1 (+) l2[::\preceq] L1 (+) L2 = (G, s \vdash l1[::\preceq] L1 \wedge G, s \vdash l2[::\preceq] L2)$   
 $\langle \text{proof} \rangle$

**lemma** *lconf-ext-list* [rule-format (no-asm)]:  
 $\bigwedge X. \llbracket G, s \vdash l[::\preceq] L \rrbracket \implies$   
 $\forall vs \ Ts. \text{distinct } vns \longrightarrow \text{length } Ts = \text{length } vns$   
 $\longrightarrow \text{list-all2 } (\text{conf } G \ s) \ vs \ Ts \longrightarrow G, s \vdash l(vns \mapsto vs)[::\preceq] L(vns \mapsto Ts)$   
 $\langle \text{proof} \rangle$

**lemma** *lconf-deallocL*:  $\llbracket G, s \vdash l[::\preceq] L(vn \mapsto T); L \text{ vn} = \text{None} \rrbracket \implies G, s \vdash l[::\preceq] L$   
 $\langle \text{proof} \rangle$

**lemma** *lconf-gext* [elim]:  $\llbracket G, s \vdash l[::\preceq] L; s \leq |s| \rrbracket \implies G, s \vdash l[::\preceq] L$   
 $\langle \text{proof} \rangle$

**lemma** *lconf-empty* [simp, intro!]:  $G, s \vdash vs[::\preceq] \text{empty}$   
 $\langle \text{proof} \rangle$

**lemma** *lconf-init-vals* [intro!]:  
 $\forall n. \forall T \in fs \ n:\text{is-type } G \ T \implies G, s \vdash \text{init-vals } fs[::\preceq] fs$   
 $\langle \text{proof} \rangle$

## weak value list conformance

Only if the value is defined it has to conform to its type. This is the contribution of the definite assignment analysis to the notion of conformance. The definite assignment analysis ensures that the program only attempts to access local variables that actually have a defined value in the state. So conformance must only ensure that the defined values are of the right type, and not also that the value is defined.

## constdefs

$wlconf :: \text{prog} \Rightarrow st \Rightarrow ('a, \text{val}) \text{ table} \Rightarrow ('a, \text{ty}) \text{ table} \Rightarrow \text{bool}$   
 $(\neg, \vdash, \vdash[::\preceq]) - [71, 71, 71, 71] \ 70)$   
 $G, s \vdash vs[\sim::\preceq] Ts \equiv \forall n. \forall T \in Ts \ n: \forall v \in vs \ n: G, s \vdash v :: \preceq T$

**lemma** *wlconfD*:  $\llbracket G, s \vdash vs[\sim::\preceq] Ts; Ts \ n = \text{Some } T; vs \ n = \text{Some } v \rrbracket \implies G, s \vdash v :: \preceq T$   
 $\langle \text{proof} \rangle$

**lemma** *wlconf-cong* [simp]:  $\bigwedge s. G, \text{set-locals } x \vdash l[\sim::\preceq] L = G, s \vdash l[\sim::\preceq] L$   
 $\langle \text{proof} \rangle$

**lemma** *wlconf-lupd* [simp]:  $G, \text{lupd}(vn \mapsto v) \vdash l[\sim::\preceq] L = G, s \vdash l[\sim::\preceq] L$   
 $\langle \text{proof} \rangle$

**lemma** *wlconf-upd*:  $\llbracket G, s \vdash l[\sim::\preceq] L; G, s \vdash v::\preceq T; L \text{ } vn = \text{Some } T \rrbracket \implies$   
 $G, s \vdash l(vn \mapsto v)[\sim::\preceq] L$   
 $\langle \text{proof} \rangle$

**lemma** *wlconf-ext*:  $\llbracket G, s \vdash l[\sim::\preceq] L; G, s \vdash v::\preceq T \rrbracket \implies G, s \vdash l(vn \mapsto v)[\sim::\preceq] L(vn \mapsto T)$   
 $\langle \text{proof} \rangle$

**lemma** *wlconf-map-sum* [simp]:  
 $G, s \vdash l1 (+) l2[\sim::\preceq] L1 (+) L2 = (G, s \vdash l1[\sim::\preceq] L1 \wedge G, s \vdash l2[\sim::\preceq] L2)$   
 $\langle \text{proof} \rangle$

**lemma** *wlconf-ext-list* [rule-format (no-asm)]:  
 $\bigwedge X. \llbracket G, s \vdash l[\sim::\preceq] L \rrbracket \implies$   
 $\forall vs \ Ts. \text{distinct } vns \longrightarrow \text{length } Ts = \text{length } vns$   
 $\longrightarrow \text{list-all2 } (\text{conf } G \ s) \ vs \ Ts \longrightarrow G, s \vdash l(vns[\mapsto] vs)[\sim::\preceq] L(vns[\mapsto] Ts)$   
 $\langle \text{proof} \rangle$

**lemma** *wlconf-deallocL*:  $\llbracket G, s \vdash l[\sim::\preceq] L(vn \mapsto T); L \text{ } vn = \text{None} \rrbracket \implies G, s \vdash l[\sim::\preceq] L$   
 $\langle \text{proof} \rangle$

**lemma** *wlconf-geat* [elim]:  $\llbracket G, s \vdash l[\sim::\preceq] L; s \leq s' \rrbracket \implies G, s' \vdash l[\sim::\preceq] L$   
 $\langle \text{proof} \rangle$

**lemma** *wlconf-empty* [simp, intro!]:  $G, s \vdash vs[\sim::\preceq] \text{empty}$   
 $\langle \text{proof} \rangle$

**lemma** *wlconf-empty-vals*:  $G, s \vdash \text{empty}[\sim::\preceq] ts$   
 $\langle \text{proof} \rangle$

**lemma** *wlconf-init-vals* [intro!]:  
 $\forall n. \forall T \in fs \ n:\text{is-type } G \ T \implies G, s \vdash \text{init-vals } fs[\sim::\preceq] fs$   
 $\langle \text{proof} \rangle$

**lemma** *lconf-wlconf*:  
 $G, s \vdash l[\sim::\preceq] L \implies G, s \vdash l[\sim::\preceq] L$   
 $\langle \text{proof} \rangle$

**object conformance****constdefs**

$$\begin{aligned}
oconf &:: prog \Rightarrow st \Rightarrow obj \Rightarrow oref \Rightarrow bool \quad (-, \vdash :: \preceq \sqrt{-} \quad [71, 71, 71, 71] \quad 70) \\
G, st \vdash obj &:: \preceq \sqrt{r} \equiv G, st \vdash values \, obj [:: \preceq] var-tys \, G \, (tag \, obj) \, r \wedge \\
&\quad (case \, r \, of \\
&\quad \quad Heap \, a \Rightarrow is-type \, G \, (obj-ty \, obj) \\
&\quad \quad | Stat \, C \Rightarrow True)
\end{aligned}$$

**lemma** *oconf-is-type*:  $G, st \vdash obj :: \preceq \sqrt{Heap \, a} \implies is-type \, G \, (obj-ty \, obj)$   
 $\langle proof \rangle$

**lemma** *oconf-lconf*:  $G, st \vdash obj :: \preceq \sqrt{r} \implies G, st \vdash values \, obj [:: \preceq] var-tys \, G \, (tag \, obj) \, r$   
 $\langle proof \rangle$

**lemma** *oconf-cong [simp]*:  $G, set-locals \, l \, st \vdash obj :: \preceq \sqrt{r} = G, st \vdash obj :: \preceq \sqrt{r}$   
 $\langle proof \rangle$

**lemma** *oconf-init-obj-lemma*:

$$\begin{aligned}
&\llbracket \bigwedge C \, c. \, class \, G \, C = Some \, c \implies unique \, (DeclConcepts.fields \, G \, C); \\
&\quad \bigwedge C \, c \, fld. \llbracket class \, G \, C = Some \, c; \\
&\quad \quad table-of \, (DeclConcepts.fields \, G \, C) \, f = Some \, fld \rrbracket \\
&\quad \implies is-type \, G \, (type \, fld); \\
&\quad (case \, r \, of \\
&\quad \quad Heap \, a \Rightarrow is-type \, G \, (obj-ty \, obj) \\
&\quad \quad | Stat \, C \Rightarrow is-class \, G \, C) \\
&\rrbracket \implies G, st \vdash obj \, (\vdash values := init-vals \, (var-tys \, G \, (tag \, obj) \, r)) :: \preceq \sqrt{r} \\
&\langle proof \rangle
\end{aligned}$$
**state conformance****constdefs**

$$\begin{aligned}
conforms &:: state \Rightarrow env \Rightarrow bool \quad ( \quad - :: \preceq - \quad [71, 71] \quad 70) \\
xs :: \preceq E &\equiv let \, (G, L) = E; \, s = snd \, xs; \, l = locals \, s \, in \\
&\quad (\forall r. \, \forall obj \in globs \, s \, r: \quad G, st \vdash obj \quad :: \preceq \sqrt{r}) \wedge \\
&\quad \quad G, st \vdash l \quad [\sim :: \preceq] L \wedge \\
&\quad (\forall a. \, fst \, xs = Some(Xcpt \, (Loc \, a)) \longrightarrow G, st \vdash Addr \, a :: \preceq Class \, (SXcpt \, Throwable)) \wedge \\
&\quad (fst \, xs = Some(Jump \, Ret) \longrightarrow l \, Result \neq None)
\end{aligned}$$
**conforms**

**lemma** *conforms-globsD*:

$$\llbracket (x, s) :: \preceq (G, L); \, globs \, s \, r = Some \, obj \rrbracket \implies G, st \vdash obj :: \preceq \sqrt{r} \\
\langle proof \rangle$$

**lemma** *conforms-localD*:  $(x, s) :: \preceq (G, L) \implies G, st \vdash locals \, s [\sim :: \preceq] L$   
 $\langle proof \rangle$

**lemma** *conforms-XcptLocD*:  $\llbracket (x, s) :: \preceq (G, L); \, x = Some \, (Xcpt \, (Loc \, a)) \rrbracket \implies$   
 $G, st \vdash Addr \, a :: \preceq Class \, (SXcpt \, Throwable)$   
 $\langle proof \rangle$

**lemma** *conforms-RetD*:  $\llbracket (x, s) :: \preceq (G, L); x = \text{Some } (\text{Jump Ret}) \rrbracket \implies$   
 $(\text{locals } s) \text{ Result} \neq \text{None}$   
 $\langle \text{proof} \rangle$

**lemma** *conforms-RefTD*:  
 $\llbracket G, s \vdash a' :: \preceq \text{RefT } t; a' \neq \text{Null}; (x, s) :: \preceq (G, L) \rrbracket \implies$   
 $\exists a \text{ obj. } a' = \text{Addr } a \wedge \text{globs } s (\text{Inl } a) = \text{Some obj} \wedge$   
 $G \vdash \text{obj-ty obj} \preceq \text{RefT } t \wedge \text{is-type } G (\text{obj-ty obj})$   
 $\langle \text{proof} \rangle$

**lemma** *conforms-Jump* [iff]:  
 $j = \text{Ret} \longrightarrow \text{locals } s \text{ Result} \neq \text{None}$   
 $\implies ((\text{Some } (\text{Jump } j), s) :: \preceq (G, L)) = (\text{Norm } s :: \preceq (G, L))$   
 $\langle \text{proof} \rangle$

**lemma** *conforms-StdXcpt* [iff]:  
 $((\text{Some } (\text{Xcpt } (\text{Std } xn)), s) :: \preceq (G, L)) = (\text{Norm } s :: \preceq (G, L))$   
 $\langle \text{proof} \rangle$

**lemma** *conforms-Err* [iff]:  
 $((\text{Some } (\text{Error } e), s) :: \preceq (G, L)) = (\text{Norm } s :: \preceq (G, L))$   
 $\langle \text{proof} \rangle$

**lemma** *conforms-raise-if* [iff]:  
 $((\text{raise-if } c \text{ } xn \text{ } x, s) :: \preceq (G, L)) = ((x, s) :: \preceq (G, L))$   
 $\langle \text{proof} \rangle$

**lemma** *conforms-error-if* [iff]:  
 $((\text{error-if } c \text{ } err \text{ } x, s) :: \preceq (G, L)) = ((x, s) :: \preceq (G, L))$   
 $\langle \text{proof} \rangle$

**lemma** *conforms-NormI*:  $(x, s) :: \preceq (G, L) \implies \text{Norm } s :: \preceq (G, L)$   
 $\langle \text{proof} \rangle$

**lemma** *conforms-absorb* [rule-format]:  
 $(a, b) :: \preceq (G, L) \longrightarrow (\text{absorb } j \text{ } a, b) :: \preceq (G, L)$   
 $\langle \text{proof} \rangle$

**lemma** *conformsI*:  $\llbracket \forall r. \forall \text{obj} \in \text{globs } s \text{ } r: G, s \vdash \text{obj} :: \preceq \sqrt{r};$   
 $G, s \vdash \text{locals } s [\sim :: \preceq] L;$   
 $\forall a. x = \text{Some } (\text{Xcpt } (\text{Loc } a)) \longrightarrow G, s \vdash \text{Addr } a :: \preceq \text{Class } (\text{SXcpt Throwable});$   
 $x = \text{Some } (\text{Jump Ret}) \longrightarrow \text{locals } s \text{ Result} \neq \text{None} \rrbracket \implies$   
 $(x, s) :: \preceq (G, L)$   
 $\langle \text{proof} \rangle$

**lemma** *conforms-xconf*:  $\llbracket (x, s) :: \preceq (G, L);$   
 $\forall a. x' = \text{Some } (\text{Xcpt } (\text{Loc } a)) \longrightarrow G, s \vdash \text{Addr } a :: \preceq \text{Class } (\text{SXcpt Throwable});$   
 $\rrbracket$

$x' = \text{Some } (\text{Jump Ret}) \longrightarrow \text{locals } s \text{ Result} \neq \text{None}] \Longrightarrow$   
 $\langle (x', s) :: \preceq (G, L) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *conforms-lupd*:

$\llbracket (x, s) :: \preceq (G, L); L \text{ vn} = \text{Some } T; G, s \vdash v :: \preceq T \rrbracket \Longrightarrow (x, \text{lupd}(v \mapsto v) s) :: \preceq (G, L)$   
 $\langle \text{proof} \rangle$

**lemmas** *conforms-allocL-aux* = *conforms-localD* [THEN *wlconf-ext*]

**lemma** *conforms-allocL*:

$\llbracket (x, s) :: \preceq (G, L); G, s \vdash v :: \preceq T \rrbracket \Longrightarrow (x, \text{lupd}(v \mapsto v) s) :: \preceq (G, L(v \mapsto T))$   
 $\langle \text{proof} \rangle$

**lemmas** *conforms-deallocL-aux* = *conforms-localD* [THEN *wlconf-deallocL*]

**lemma** *conforms-deallocL*:  $\bigwedge s. \llbracket s :: \preceq (G, L(v \mapsto T)); L \text{ vn} = \text{None} \rrbracket \Longrightarrow s :: \preceq (G, L)$

$\langle \text{proof} \rangle$

**lemma** *conforms-gext*:  $\llbracket (x, s) :: \preceq (G, L); s \leq | s' \rrbracket;$

$\forall r. \forall \text{obj} \in \text{globs } s' r: G, s \vdash \text{obj} :: \preceq \sqrt{r};$

$\text{locals } s' = \text{locals } s \rrbracket \Longrightarrow (x, s') :: \preceq (G, L)$

$\langle \text{proof} \rangle$

**lemma** *conforms-xgext*:

$\llbracket (x, s) :: \preceq (G, L); (x', s') :: \preceq (G, L); s' \leq | s; \text{dom } (\text{locals } s') \subseteq \text{dom } (\text{locals } s) \rrbracket$

$\Longrightarrow (x', s) :: \preceq (G, L)$

$\langle \text{proof} \rangle$

**lemma** *conforms-gupd*:  $\bigwedge \text{obj}. \llbracket (x, s) :: \preceq (G, L); G, s \vdash \text{obj} :: \preceq \sqrt{r}; s \leq | \text{gupd}(r \mapsto \text{obj}) s \rrbracket$

$\Longrightarrow (x, \text{gupd}(r \mapsto \text{obj}) s) :: \preceq (G, L)$

$\langle \text{proof} \rangle$

**lemma** *conforms-upd-gobj*:  $\llbracket (x, s) :: \preceq (G, L); \text{globs } s \text{ } r = \text{Some } \text{obj};$

$\text{var-tys } G \text{ (tag obj) } r \text{ n} = \text{Some } T; G, s \vdash v :: \preceq T \rrbracket \Longrightarrow (x, \text{upd-gobj } r \text{ n } v s) :: \preceq (G, L)$

$\langle \text{proof} \rangle$

**lemma** *conforms-set-locals*:

$\llbracket (x, s) :: \preceq (G, L); G, s \vdash l [\sim :: \preceq] L; x = \text{Some } (\text{Jump Ret}) \longrightarrow l \text{ Result} \neq \text{None} \rrbracket$

$\Longrightarrow (x, \text{set-locals } l s) :: \preceq (G, L)$

$\langle \text{proof} \rangle$

**lemma** *conforms-locals*:

$\llbracket (a, b) :: \preceq (G, L); L x = \text{Some } T; \text{locals } b \text{ } x \neq \text{None} \rrbracket$

$\Longrightarrow G, b \vdash \text{the } (\text{locals } b \text{ } x) :: \preceq T$

$\langle \text{proof} \rangle$

**lemma** *conforms-return:*

$\bigwedge s'. \llbracket (x, s) :: \preceq (G, L); (x', s') :: \preceq (G, L'); s \leq |s'; x' \neq \text{Some } (\text{Jump Ret}) \rrbracket \implies$   
 $(x', \text{set-locals } (\text{locals } s) s') :: \preceq (G, L)$   
 $\langle \text{proof} \rangle$

**end**



## Chapter 18

# DefiniteAssignmentCorrect

## 45 Correctness of Definite Assignment

**theory** *DefiniteAssignmentCorrect* **imports** *WellForm Eval begin*

$\langle ML \rangle$

**lemma** *sxalloc-no-jump*:

**assumes** *sxalloc*:  $G \vdash s0 \text{ --sxalloc--> } s1$  **and**  
*no-jmp*:  $\text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$   
**shows**  $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$

$\langle \text{proof} \rangle$

**lemma** *sxalloc-no-jump'*:

**assumes** *sxalloc*:  $G \vdash s0 \text{ --sxalloc--> } s1$  **and**  
*jump*:  $\text{abrupt } s1 = \text{Some } (\text{Jump } j)$   
**shows**  $\text{abrupt } s0 = \text{Some } (\text{Jump } j)$

$\langle \text{proof} \rangle$

**lemma** *halloc-no-jump*:

**assumes** *halloc*:  $G \vdash s0 \text{ --halloc } oi \succ a \text{ --> } s1$  **and**  
*no-jmp*:  $\text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$   
**shows**  $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$

$\langle \text{proof} \rangle$

**lemma** *halloc-no-jump'*:

**assumes** *halloc*:  $G \vdash s0 \text{ --halloc } oi \succ a \text{ --> } s1$  **and**  
*jump*:  $\text{abrupt } s1 = \text{Some } (\text{Jump } j)$   
**shows**  $\text{abrupt } s0 = \text{Some } (\text{Jump } j)$

$\langle \text{proof} \rangle$

**lemma** *Body-no-jump*:

**assumes** *eval*:  $G \vdash s0 \text{ --Body } D \text{ c--}\succ v \text{ --> } s1$  **and**  
*jump*:  $\text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$   
**shows**  $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$

$\langle \text{proof} \rangle$

**lemma** *Methd-no-jump*:

**assumes** *eval*:  $G \vdash s0 \text{ --Methd } D \text{ sig--}\succ v \text{ --> } s1$  **and**  
*jump*:  $\text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$   
**shows**  $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$

$\langle \text{proof} \rangle$

**lemma** *jumpNestingOkS-mono*:

**assumes** *jumpNestingOk-l'*:  $\text{jumpNestingOkS } \text{jmps}' \text{ } c$   
**and** *subset*:  $\text{jmps}' \subseteq \text{jmps}$   
**shows**  $\text{jumpNestingOkS } \text{jmps} \text{ } c$

$\langle \text{proof} \rangle$

**corollary** *jumpNestingOk-mono*:

**assumes** *jmpOk*:  $\text{jumpNestingOk } \text{jmps}' \text{ } t$   
**and** *subset*:  $\text{jmps}' \subseteq \text{jmps}$   
**shows**  $\text{jumpNestingOk } \text{jmps} \text{ } t$

$\langle \text{proof} \rangle$

**lemma** *assign-abrupt-propagation*:

**assumes** *f-ok*:  $\text{abrupt } (f \ n \ s) \neq x$   
**and** *ass*:  $\text{abrupt } (\text{assign } f \ n \ s) = x$   
**shows**  $\text{abrupt } s = x$

$\langle \text{proof} \rangle$

**lemma** *wt-init-comp-ty'*:

*is-acc-type* (*prg Env*) (*pid* (*cls Env*))  $T \implies \text{Env} \vdash \text{init-comp-ty } T :: \checkmark$

$\langle \text{proof} \rangle$

**lemma** *fvar-upd-no-jump*:

**assumes** *upd*:  $\text{upd} = \text{snd } (\text{fst } (\text{fvar statDeclC stat fn } a \ s'))$   
**and** *noJmp*:  $\text{abrupt } s \neq \text{Some } (\text{Jump } j)$   
**shows**  $\text{abrupt } (\text{upd val } s) \neq \text{Some } (\text{Jump } j)$

$\langle \text{proof} \rangle$

**lemma** *avar-state-no-jump*:

**assumes** *jmp*:  $\text{abrupt } (\text{snd } (\text{avar } G \ i \ a \ s)) = \text{Some } (\text{Jump } j)$   
**shows**  $\text{abrupt } s = \text{Some } (\text{Jump } j)$

$\langle \text{proof} \rangle$

**lemma** *avar-upd-no-jump*:

**assumes** *upd*:  $\text{upd} = \text{snd } (\text{fst } (\text{avar } G \ i \ a \ s'))$   
**and** *noJmp*:  $\text{abrupt } s \neq \text{Some } (\text{Jump } j)$   
**shows**  $\text{abrupt } (\text{upd val } s) \neq \text{Some } (\text{Jump } j)$

$\langle \text{proof} \rangle$

The next theorem expresses: If jumps (breaks, continues, returns) are nested correctly, we won't find an unexpected jump in the result state of the evaluation. For example, a break can't leave its enclosing loop, an return can't leave its enclosing method. To prove this, the method call is critical. Although the wellformedness of the whole program guarantees that the jumps (breaks, continues and returns) are nested correctly in all method bodies, the call rule alone does not guarantee that I will call a method or even a class that is part of the program due to dynamic binding! To be able to ensure this we need a kind of conformance of the state, like in the typesafety proof. But then we will redo the typesafety proof here. It would be nice if we could find an easy precondition that will guarantee that all calls will actually call classes and methods of the current program, which can be instantiated in the typesafety proof later on. To fix this problem, I have instrumented the semantic definition of a call to filter out any breaks in the state and to throw an error instead.

To get an induction hypothesis which is strong enough to perform the proof, we can't just assume *jumpNestingOk* for the empty set and conclude, that no jump at all will be in the resulting state, because the set is altered by the statements *Lab* and *While*.

The wellformedness of the program is used to ensure that for all class initialisations and methods the nesting of jumps is wellformed, too.

**theorem** *jumpNestingOk-eval*:

**assumes** *eval*:  $G \vdash s0 \multimap \rightarrow (v, s1)$   
**and** *jmpOk*:  $\text{jumpNestingOk } \text{jmps } t$   
**and** *wt*:  $\text{Env} \vdash t :: T$   
**and** *wf*:  $\text{wf-prog } G$

**and**  $G: \text{prg Env} = G$   
**and**  $\text{no-jmp}: \forall j. \text{abrupt } s0 = \text{Some } (\text{Jump } j) \longrightarrow j \in \text{jmps}$   
 $\quad (\text{is } ?\text{Jmp } \text{jmps } s0)$   
**shows**  $?Jmp \text{jmps } s1 \wedge$   
 $\quad (\text{normal } s1 \longrightarrow$   
 $\quad (\forall w \text{ upd}. v = \text{In2 } (w, \text{upd})$   
 $\quad \longrightarrow (\forall s \text{ j val}.$   
 $\quad \quad \text{abrupt } s \neq \text{Some } (\text{Jump } j) \longrightarrow$   
 $\quad \quad \text{abrupt } (\text{upd val } s) \neq \text{Some } (\text{Jump } j))))$   
 $(\text{is } ?Jmp \text{jmps } s1 \wedge ?Upd v s1)$   
 $\langle \text{proof} \rangle$

**lemmas**  $\text{jumpNestingOk-evalE} = \text{jumpNestingOk-eval } [\text{THEN } \text{conjE}, \text{rule-format}]$

**lemma**  $\text{jumpNestingOk-eval-no-jump}$ :  
**assumes**  $\text{eval}: \text{prg Env} \vdash s0 \dashv t \longrightarrow (v, s1)$  **and**  
 $\text{jmpOk}: \text{jumpNestingOk } \{ \} t$  **and**  
 $\text{no-jmp}: \text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$  **and**  
 $\text{wt}: \text{Env} \vdash t :: T$  **and**  
 $\text{wf}: \text{wf-prog } (\text{prg Env})$   
**shows**  $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j) \wedge$   
 $(\text{normal } s1 \longrightarrow v = \text{In2 } (w, \text{upd})$   
 $\longrightarrow \text{abrupt } s \neq \text{Some } (\text{Jump } j')$   
 $\longrightarrow \text{abrupt } (\text{upd val } s) \neq \text{Some } (\text{Jump } j'))$   
 $\langle \text{proof} \rangle$

**lemmas**  $\text{jumpNestingOk-eval-no-jumpE}$   
 $= \text{jumpNestingOk-eval-no-jump } [\text{THEN } \text{conjE}, \text{rule-format}]$

**corollary**  $\text{eval-expression-no-jump}$ :  
**assumes**  $\text{eval}: \text{prg Env} \vdash s0 \dashv e \dashv v \longrightarrow s1$  **and**  
 $\text{no-jmp}: \text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$  **and**  
 $\text{wt}: \text{Env} \vdash e :: \neg T$  **and**  
 $\text{wf}: \text{wf-prog } (\text{prg Env})$   
**shows**  $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$   
 $\langle \text{proof} \rangle$

**corollary**  $\text{eval-var-no-jump}$ :  
**assumes**  $\text{eval}: \text{prg Env} \vdash s0 \dashv \text{var} \dashv (w, \text{upd}) \longrightarrow s1$  **and**  
 $\text{no-jmp}: \text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$  **and**  
 $\text{wt}: \text{Env} \vdash \text{var} :: T$  **and**  
 $\text{wf}: \text{wf-prog } (\text{prg Env})$   
**shows**  $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j) \wedge$   
 $(\text{normal } s1 \longrightarrow$   
 $(\text{abrupt } s \neq \text{Some } (\text{Jump } j')$   
 $\longrightarrow \text{abrupt } (\text{upd val } s) \neq \text{Some } (\text{Jump } j'))$   
 $\langle \text{proof} \rangle$

**lemmas**  $\text{eval-var-no-jumpE} = \text{eval-var-no-jump } [\text{THEN } \text{conjE}, \text{rule-format}]$

**corollary**  $\text{eval-statement-no-jump}$ :  
**assumes**  $\text{eval}: \text{prg Env} \vdash s0 \dashv c \longrightarrow s1$  **and**  
 $\text{jmpOk}: \text{jumpNestingOkS } \{ \} c$  **and**  
 $\text{no-jmp}: \text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$  **and**  
 $\text{wt}: \text{Env} \vdash c :: \checkmark$  **and**  
 $\text{wf}: \text{wf-prog } (\text{prg Env})$   
**shows**  $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$

$\langle \text{proof} \rangle$

**corollary** *eval-expression-list-no-jump*:

**assumes** *eval*:  $\text{prg } Env \vdash s0 \text{ } \text{--es} \dot{=} \text{>} v \rightarrow s1$  **and**  
*no-jmp*:  $\text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$  **and**  
*wt*:  $Env \vdash \text{es} :: \dot{=} T$  **and**  
*wf*:  $\text{wf-prog } (\text{prg } Env)$   
**shows**  $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$

$\langle \text{proof} \rangle$

**lemma** *union-subseteq-elim* [elim]:  $\llbracket A \cup B \subseteq C; \llbracket A \subseteq C; B \subseteq C \rrbracket \implies P \rrbracket \implies P$

$\langle \text{proof} \rangle$

**lemma** *dom-locals-halloc-mono*:

**assumes** *halloc*:  $G \vdash s0 \text{--halloc } oi \text{>} a \rightarrow s1$   
**shows**  $\text{dom } (\text{locals } (\text{store } s0)) \subseteq \text{dom } (\text{locals } (\text{store } s1))$

$\langle \text{proof} \rangle$

**lemma** *dom-locals-sxalloc-mono*:

**assumes** *sxalloc*:  $G \vdash s0 \text{--sxalloc} \rightarrow s1$   
**shows**  $\text{dom } (\text{locals } (\text{store } s0)) \subseteq \text{dom } (\text{locals } (\text{store } s1))$

$\langle \text{proof} \rangle$

**lemma** *dom-locals-assign-mono*:

**assumes** *f-ok*:  $\text{dom } (\text{locals } (\text{store } s)) \subseteq \text{dom } (\text{locals } (\text{store } (f \ n \ s)))$   
**shows**  $\text{dom } (\text{locals } (\text{store } s)) \subseteq \text{dom } (\text{locals } (\text{store } (\text{assign } f \ n \ s)))$

$\langle \text{proof} \rangle$

**lemma** *dom-locals-lvar-mono*:

$\text{dom } (\text{locals } (\text{store } s)) \subseteq \text{dom } (\text{locals } (\text{store } (\text{snd } (\text{lvar } vn \ s') \ \text{val } s)))$

$\langle \text{proof} \rangle$

**lemma** *dom-locals-fvar-vvar-mono*:

$\text{dom } (\text{locals } (\text{store } s))$   
 $\subseteq \text{dom } (\text{locals } (\text{store } (\text{snd } (\text{fst } (\text{fvar } \text{statDeclC } \text{stat } fn \ a \ s') \ \text{val } s))))$

$\langle \text{proof} \rangle$

**lemma** *dom-locals-fvar-mono*:

$\text{dom } (\text{locals } (\text{store } s))$   
 $\subseteq \text{dom } (\text{locals } (\text{store } (\text{snd } (\text{fvar } \text{statDeclC } \text{stat } fn \ a \ s))))$

$\langle \text{proof} \rangle$

**lemma** *dom-locals-avar-vvar-mono*:

$\text{dom } (\text{locals } (\text{store } s))$   
 $\subseteq \text{dom } (\text{locals } (\text{store } (\text{snd } (\text{fst } (\text{avar } G \ i \ a \ s') \ \text{val } s))))$

$\langle \text{proof} \rangle$

**lemma** *dom-locals-avar-mono*:

$\text{dom} (\text{locals} (\text{store } s))$   
 $\subseteq \text{dom} (\text{locals} (\text{store} (\text{snd} (\text{avar } G \ i \ a \ s))))$   
 $\langle \text{proof} \rangle$

Since assignments are modelled as functions from states to states, we must take into account these functions. They appear only in the assignment rule and as result from evaluating a variable. That's why we need the complicated second part of the conjunction in the goal. The reason for the very generic way to treat assignments was the aim to omit redundancy. There is only one evaluation rule for each kind of variable (locals, fields, arrays). These rules are used for both accessing variables and updating variables. That's why the evaluation rules for variables result in a pair consisting of a value and an update function. Of course we could also think of a pair of a value and a reference in the store, instead of the generic update function. But as only array updates can cause a special exception (if the types mismatch) and not array reads we then have to introduce two different rules to handle array reads and updates

**lemma** *dom-locals-eval-mono*:

**assumes**  $\text{eval}: G \vdash s0 \rightarrow t \rightarrow (v, s1)$   
**shows**  $\text{dom} (\text{locals} (\text{store } s0)) \subseteq \text{dom} (\text{locals} (\text{store } s1)) \wedge$   
 $(\forall \text{ vv}. v = \text{In2 } vv \wedge \text{normal } s1$   
 $\rightarrow (\forall s \text{ val}. \text{dom} (\text{locals} (\text{store } s))$   
 $\subseteq \text{dom} (\text{locals} (\text{store} ((\text{snd } vv) \text{ val } s))))$

$\langle \text{proof} \rangle$

**lemma** *dom-locals-eval-mono-elim* [consumes 1]:

**assumes**  $\text{eval}: G \vdash s0 \rightarrow t \rightarrow (v, s1)$  **and**  
 $\text{hyps}: \llbracket \text{dom} (\text{locals} (\text{store } s0)) \subseteq \text{dom} (\text{locals} (\text{store } s1)) \rrbracket$   
 $\wedge \text{vv } s \text{ val}. \llbracket v = \text{In2 } vv; \text{normal } s1 \rrbracket$   
 $\implies \text{dom} (\text{locals} (\text{store } s))$   
 $\subseteq \text{dom} (\text{locals} (\text{store} ((\text{snd } vv) \text{ val } s))) \rrbracket \implies P$

**shows**  $P$

$\langle \text{proof} \rangle$

**lemma** *halloc-no-abrupt*:

**assumes**  $\text{halloc}: G \vdash s0 \rightarrow \text{halloc } oi \rightarrow a \rightarrow s1$  **and**  
 $\text{normal}: \text{normal } s1$

**shows**  $\text{normal } s0$

$\langle \text{proof} \rangle$

**lemma** *sxalloc-mono-no-abrupt*:

**assumes**  $\text{sxalloc}: G \vdash s0 \rightarrow \text{sxalloc} \rightarrow s1$  **and**  
 $\text{normal}: \text{normal } s1$

**shows**  $\text{normal } s0$

$\langle \text{proof} \rangle$

**lemma** *union-subseteqI*:  $\llbracket A \cup B \subseteq C; A' \subseteq A; B' \subseteq B \rrbracket \implies A' \cup B' \subseteq C$

$\langle \text{proof} \rangle$

**lemma** *union-subseteqII*:  $\llbracket A \cup B \subseteq C; A' \subseteq A \rrbracket \implies A' \cup B \subseteq C$

$\langle \text{proof} \rangle$

**lemma** *union-subseteqIr*:  $\llbracket A \cup B \subseteq C; B' \subseteq B \rrbracket \implies A \cup B' \subseteq C$   
 ⟨proof⟩

**lemma** *subseteq-union-transl* [trans]:  $\llbracket A \subseteq B; B \cup C \subseteq D \rrbracket \implies A \cup C \subseteq D$   
 ⟨proof⟩

**lemma** *subseteq-union-transr* [trans]:  $\llbracket A \subseteq B; C \cup B \subseteq D \rrbracket \implies A \cup C \subseteq D$   
 ⟨proof⟩

**lemma** *union-subseteq-weaken*:  $\llbracket A \cup B \subseteq C; \llbracket A \subseteq C; B \subseteq C \rrbracket \implies P \rrbracket \implies P$   
 ⟨proof⟩

**lemma** *assigns-good-approx*:  
**assumes**  
   *eval*:  $G \vdash s0 \dashv t \succ \rightarrow (v, s1)$  **and**  
   *normal*: *normal* *s1*  
**shows**  $\text{assigns } t \subseteq \text{dom } (\text{locals } (\text{store } s1))$   
 ⟨proof⟩

**corollary** *assignsE-good-approx*:  
**assumes**  
   *eval*:  $\text{prg } Env \vdash s0 \dashv e \dashv \succ v \rightarrow s1$  **and**  
   *normal*: *normal* *s1*  
**shows**  $\text{assignsE } e \subseteq \text{dom } (\text{locals } (\text{store } s1))$   
 ⟨proof⟩

**corollary** *assignsV-good-approx*:  
**assumes**  
   *eval*:  $\text{prg } Env \vdash s0 \dashv v = \succ v f \rightarrow s1$  **and**  
   *normal*: *normal* *s1*  
**shows**  $\text{assignsV } v \subseteq \text{dom } (\text{locals } (\text{store } s1))$   
 ⟨proof⟩

**corollary** *assignsEs-good-approx*:  
**assumes**  
   *eval*:  $\text{prg } Env \vdash s0 \dashv es \doteq \succ v s \rightarrow s1$  **and**  
   *normal*: *normal* *s1*  
**shows**  $\text{assignsEs } es \subseteq \text{dom } (\text{locals } (\text{store } s1))$   
 ⟨proof⟩

**lemma** *constVal-eval*:  
**assumes** *const*: *constVal* *e* = *Some* *c* **and**  
   *eval*:  $G \vdash \text{Norm } s0 \dashv e \dashv \succ v \rightarrow s$   
**shows**  $v = c \wedge \text{normal } s$   
 ⟨proof⟩

**lemmas** *constVal-eval-elim* = *constVal-eval* [THEN *conjE*]

**lemma** *eval-unop-type*:  
*typeof* *dt* (*eval-unop* *unop* *v*) = *Some* (*PrimT* (*unop-type* *unop*))  
 ⟨proof⟩

**lemma** *eval-binop-type*:

*typeof* *dt* (*eval-binop binop v1 v2*) = *Some (PrimT (binop-type binop))*  
 ⟨*proof*⟩

**lemma** *constVal-Boolean*:

**assumes** *const*: *constVal e* = *Some c* **and**  
*wt*: *Env* ⊢ *e* :: − *PrimT Boolean*  
**shows** *typeof empty-dt c* = *Some (PrimT Boolean)*  
 ⟨*proof*⟩

**lemma** *assigns-if-good-approx*:

**assumes**  
*eval*: *prg Env* ⊢ *s0* − *e* −  $\succ$  *b* → *s1* **and**  
*normal*: *normal s1* **and**  
*bool*: *Env* ⊢ *e* :: − *PrimT Boolean*  
**shows** *assigns-if (the-Bool b) e* ⊆ *dom (locals (store s1))*  
 ⟨*proof*⟩

**lemma** *assigns-if-good-approx'*:

**assumes** *eval*: *G* ⊢ *s0* − *e* −  $\succ$  *b* → *s1*  
**and** *normal*: *normal s1*  
**and** *bool*: (⊢ *prg* = *G*, *cls* = *C*, *lcl* = *L*) ⊢ *e* :: − (*PrimT Boolean*)  
**shows** *assigns-if (the-Bool b) e* ⊆ *dom (locals (store s1))*  
 ⟨*proof*⟩

**lemma** *subset-Intl*: *A* ⊆ *C* ⇒ *A* ∩ *B* ⊆ *C*

⟨*proof*⟩

**lemma** *subset-Intr*: *B* ⊆ *C* ⇒ *A* ∩ *B* ⊆ *C*

⟨*proof*⟩

**lemma** *da-good-approx*:

**assumes** *eval*: *prg Env* ⊢ *s0* − *t*  $\succ$  → (*v*, *s1*) **and**  
*wt*: *Env* ⊢ *t* :: *T* (**is** ?*Wt Env t T*) **and**  
*da*: *Env* ⊢ *dom (locals (store s0))* » *t* » *A* (**is** ?*Da Env s0 t A*) **and**  
*wf*: *wf-prog (prg Env)*  
**shows** (*normal s1* → (*nrm A* ⊆ *dom (locals (store s1))*)) ∧  
 (∀ *l*. *abrupt s1* = *Some (Jump (Break l))* ∧ *normal s0*  
 → (*brk A l* ⊆ *dom (locals (store s1))*)) ∧  
 (*abrupt s1* = *Some (Jump Ret)* ∧ *normal s0*  
 → *Result* ∈ *dom (locals (store s1))*)  
 (**is** ?*NormalAssigned s1 A* ∧ ?*BreakAssigned s0 s1 A* ∧ ?*ResAssigned s0 s1*)  
 ⟨*proof*⟩

**lemma** *da-good-approxE* [*consumes 4*]:

⊢ *prg Env* ⊢ *s0* − *t*  $\succ$  → (*v*, *s1*); *Env* ⊢ *t* :: *T*; *Env* ⊢ *dom (locals (store s0))* » *t* » *A*;  
*wf-prog (prg Env)*;  
 ⊢ *normal s1* ⇒ *nrm A* ⊆ *dom (locals (store s1))*;  
 ∧ *l*. ⊢ *abrupt s1* = *Some (Jump (Break l))*; *normal s0*⊢  
 ⇒ *brk A l* ⊆ *dom (locals (store s1))*;



$\llbracket \text{abrupt } s1 = \text{Some } (\text{Jump Ret}); \text{normal } s0 \rrbracket \Longrightarrow \text{Result} \in \text{dom } (\text{locals } (\text{store } s1))$   
 $\rrbracket \Longrightarrow P$   
 $\rrbracket \Longrightarrow P$   
 $\langle \text{proof} \rangle$

**lemma** *da-good-approxE'* [consumes 4]:  
**assumes** *eval*:  $G \vdash s0 \rightarrow t \rightarrow (v, s1)$   
**and** *wt*:  $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t :: T$   
**and** *da*:  $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg t \gg A$   
**and** *wf*: *wf-prog* *G*  
**and** *elim*:  $\llbracket \text{normal } s1 \rrbracket \Longrightarrow \text{nrm } A \subseteq \text{dom } (\text{locals } (\text{store } s1));$   
 $\bigwedge l. \llbracket \text{abrupt } s1 = \text{Some } (\text{Jump } (\text{Break } l)); \text{normal } s0 \rrbracket$   
 $\Longrightarrow \text{brk } A \ l \subseteq \text{dom } (\text{locals } (\text{store } s1));$   
 $\llbracket \text{abrupt } s1 = \text{Some } (\text{Jump Ret}); \text{normal } s0 \rrbracket$   
 $\Longrightarrow \text{Result} \in \text{dom } (\text{locals } (\text{store } s1))$   
 $\rrbracket \Longrightarrow P$   
**shows** *P*  
 $\langle \text{proof} \rangle$   
 $\langle ML \rangle$   
**end**



## Chapter 19

# TypeSafe

## 46 The type soundness proof for Java

**theory** *TypeSafe* **imports** *DefiniteAssignmentCorrect* *Conform* **begin**

**error free**

**lemma** *error-free-halloc*:

**assumes** *halloc*:  $G \vdash s0 \text{ --halloc } oi \succ a \rightarrow s1$  **and**

*error-free-s0*: *error-free s0*

**shows** *error-free s1*

$\langle \text{proof} \rangle$

**lemma** *error-free-sxalloc*:

**assumes** *sxalloc*:  $G \vdash s0 \text{ --sxalloc} \rightarrow s1$  **and** *error-free-s0*: *error-free s0*

**shows** *error-free s1*

$\langle \text{proof} \rangle$

**lemma** *error-free-check-field-access-eq*:

*error-free* (*check-field-access*  $G \text{ acc}C \text{ statDecl}C \text{ fn stat } a \ s$ )

$\implies$  (*check-field-access*  $G \text{ acc}C \text{ statDecl}C \text{ fn stat } a \ s$ ) =  $s$

$\langle \text{proof} \rangle$

**lemma** *error-free-check-method-access-eq*:

*error-free* (*check-method-access*  $G \text{ acc}C \text{ stat}T \text{ mode sig } a' \ s$ )

$\implies$  (*check-method-access*  $G \text{ acc}C \text{ stat}T \text{ mode sig } a' \ s$ ) =  $s$

$\langle \text{proof} \rangle$

**lemma** *error-free-FVar-lemma*:

*error-free s*

$\implies$  *error-free* (*abupd* (*if stat then id else np a*)  $s$ )

$\langle \text{proof} \rangle$

**lemma** *error-free-init-lvars* [*simp,intro*]:

*error-free s*  $\implies$

*error-free* (*init-lvars*  $G \ C \ \text{sig} \ \text{mode} \ a \ \text{pvs} \ s$ )

$\langle \text{proof} \rangle$

**lemma** *error-free-LVar-lemma*:

*error-free s*  $\implies$  *error-free* (*assign* ( $\lambda v. \text{supd lupd}(vn \mapsto v)$ )  $w \ s$ )

$\langle \text{proof} \rangle$

**lemma** *error-free-throw* [*simp,intro*]:

*error-free s*  $\implies$  *error-free* (*abupd* (*throw x*)  $s$ )

$\langle \text{proof} \rangle$

**result conformance**

**constdefs**

*assign-conforms* ::  $st \Rightarrow (val \Rightarrow state \Rightarrow state) \Rightarrow ty \Rightarrow env \Rightarrow bool$

( $\leq$  |  $\preceq$  ::  $\leq$  - [71,71,71,71] 70)

$s \leq | f \preceq T :: \preceq E \equiv$

( $\forall s' w. \text{Norm } s' :: \preceq E \longrightarrow \text{fst } E, s \vdash w :: \preceq T \longrightarrow s \leq | s' \longrightarrow \text{assign } f \ w \ (\text{Norm } s') :: \preceq E$ )  $\wedge$

( $\forall s' w. \text{error-free } s' \longrightarrow (\text{error-free } (\text{assign } f \ w \ s'))$ )

**constdefs**

$$\begin{aligned}
& rconf :: prog \Rightarrow lenv \Rightarrow st \Rightarrow term \Rightarrow vals \Rightarrow tys \Rightarrow bool \\
& \quad (-, -, \vdash, \succ, ::, \preceq - \quad [71, 71, 71, 71, 71, 71] \ 70) \\
& G, L, s \vdash t \succ v :: \preceq T \\
& \equiv \text{case } T \text{ of} \\
& \quad Inl \ T \Rightarrow \text{if } (\exists \text{ var. } t = In2 \text{ var}) \\
& \quad \quad \text{then } (\forall \ n. (the-In2 \ t) = LVar \ n \\
& \quad \quad \quad \longrightarrow (fst \ (the-In2 \ v) = the \ (locals \ s \ n)) \wedge \\
& \quad \quad \quad (locals \ s \ n \neq None \longrightarrow G, s \vdash fst \ (the-In2 \ v) :: \preceq T)) \wedge \\
& \quad \quad (\neg (\exists \ n. the-In2 \ t = LVar \ n) \longrightarrow (G, s \vdash fst \ (the-In2 \ v) :: \preceq T)) \wedge \\
& \quad \quad (s \leq |snd \ (the-In2 \ v) \preceq T :: \preceq (G, L)) \\
& \quad \quad \text{else } G, s \vdash the-In1 \ v :: \preceq T \\
& \quad | Inr \ Ts \Rightarrow list-all2 \ (conf \ G \ s) \ (the-In3 \ v) \ Ts
\end{aligned}$$

With *rconf* we describe the conformance of the result value of a term. This definition gets rather complicated because of the relations between the injections of the different terms, types and values. The main case distinction is between single values and value lists. In case of value lists, every value has to conform to its type. For single values we have to do a further case distinction, between values of variables  $\exists \text{ var. } t = In2 \text{ var}$  and ordinary values. Values of variables are modelled as pairs consisting of the current value and an update function which will perform an assignment to the variable. This stems from the decision, that we only have one evaluation rule for each kind of variable. The decision if we read or write to the variable is made by syntactic enclosing rules. So conformance of variable-values must ensure that both the current value and an update will conform to the type. With the introduction of definite assignment of local variables we have to do another case distinction. For the notion of conformance local variables are allowed to be *None*, since the definedness is not ensured by conformance but by definite assignment. Field and array variables must contain a value.

**lemma** *rconf-In1* [simp]:
$$G, L, s \vdash In1 \ ec \succ In1 \ v :: \preceq Inl \ T = G, s \vdash v :: \preceq T$$

⟨proof⟩

**lemma** *rconf-In2-no-LVar* [simp]:
$$\forall \ n. va \neq LVar \ n \implies$$

$$G, L, s \vdash In2 \ va \succ In2 \ vf :: \preceq Inl \ T = (G, s \vdash fst \ vf :: \preceq T \wedge s \leq |snd \ vf \preceq T :: \preceq (G, L))$$

⟨proof⟩

**lemma** *rconf-In2-LVar* [simp]:
$$va = LVar \ n \implies$$

$$G, L, s \vdash In2 \ va \succ In2 \ vf :: \preceq Inl \ T$$

$$= ((fst \ vf = the \ (locals \ s \ n)) \wedge$$

$$(locals \ s \ n \neq None \longrightarrow G, s \vdash fst \ vf :: \preceq T) \wedge s \leq |snd \ vf \preceq T :: \preceq (G, L))$$

⟨proof⟩

**lemma** *rconf-In3* [simp]:
$$G, L, s \vdash In3 \ es \succ In3 \ vs :: \preceq Inr \ Ts = list-all2 \ (\lambda v \ T. G, s \vdash v :: \preceq T) \ vs \ Ts$$

⟨proof⟩

**fits and conf****lemma** *conf-fits*:  $G, s \vdash v :: \preceq T \implies G, s \vdash v \text{ fits } T$ 

⟨proof⟩

**lemma** *fits-conf*:

$\llbracket G, s \vdash v :: \preceq T; G \vdash T \preceq? T'; G, s \vdash v \text{ fits } T'; \text{ws-prog } G \rrbracket \implies G, s \vdash v :: \preceq T'$   
 $\langle \text{proof} \rangle$

**lemma** *fits-Array*:

$\llbracket G, s \vdash v :: \preceq T; G \vdash T'.[] \preceq T.[]; G, s \vdash v \text{ fits } T'; \text{ws-prog } G \rrbracket \implies G, s \vdash v :: \preceq T'$   
 $\langle \text{proof} \rangle$

**gext**

**lemma** *halloc-gext*:  $\bigwedge s1\ s2. G \vdash s1 \text{ --halloc } oi \rightarrow s2 \implies \text{snd } s1 \leq | \text{snd } s2$   
 $\langle \text{proof} \rangle$

**lemma** *sxalloc-gext*:  $\bigwedge s1\ s2. G \vdash s1 \text{ --sxalloc } \rightarrow s2 \implies \text{snd } s1 \leq | \text{snd } s2$   
 $\langle \text{proof} \rangle$

**lemma** *eval-gext-lemma* [rule-format (no-asm)]:

$G \vdash s \text{ --t} \rightarrow (w, s') \implies \text{snd } s \leq | \text{snd } s' \wedge (\text{case } w \text{ of}$   
 $\quad \text{In1 } v \Rightarrow \text{True}$   
 $\quad | \text{In2 } vf \Rightarrow \text{normal } s \longrightarrow (\forall v\ x\ s. s \leq | \text{snd } (\text{assign } (\text{snd } vf)\ v\ (x, s)))$   
 $\quad | \text{In3 } vs \Rightarrow \text{True})$   
 $\langle \text{proof} \rangle$

**lemma** *evar-gext-f*:

$G \vdash \text{Norm } s1 \text{ --e} \rightarrow vf \rightarrow s2 \implies s \leq | \text{snd } (\text{assign } (\text{snd } vf)\ v\ (x, s))$   
 $\langle \text{proof} \rangle$

**lemmas** *eval-gext* = *eval-gext-lemma* [THEN conjunct1]

**lemma** *eval-gext'*:  $G \vdash (x1, s1) \text{ --t} \rightarrow (w, x2, s2) \implies s1 \leq | s2$   
 $\langle \text{proof} \rangle$

**lemma** *init-yields-initd*:  $G \vdash \text{Norm } s1 \text{ --Init } C \rightarrow s2 \implies \text{initd } C\ s2$   
 $\langle \text{proof} \rangle$

## Lemmas

**lemma** *obj-ty-obj-class1*:

$\llbracket \text{wf-prog } G; \text{is-type } G\ (\text{obj-ty } \text{obj}) \rrbracket \implies \text{is-class } G\ (\text{obj-class } \text{obj})$   
 $\langle \text{proof} \rangle$

**lemma** *oconf-init-obj*:

$\llbracket \text{wf-prog } G;$   
 $\quad (\text{case } r \text{ of Heap } a \Rightarrow \text{is-type } G\ (\text{obj-ty } \text{obj}) \mid \text{Stat } C \Rightarrow \text{is-class } G\ C)$   
 $\rrbracket \implies G, s \vdash \text{obj } (\text{values} := \text{init-vals } (\text{var-tys } G\ (\text{tag } \text{obj})\ r)) :: \preceq \sqrt{r}$   
 $\langle \text{proof} \rangle$

**lemma** *conforms-newG*:  $\llbracket \text{globs } s\ \text{oref} = \text{None}; (x, s) :: \preceq (G, L);$

$\text{wf-prog } G; \text{case } \text{oref} \text{ of Heap } a \Rightarrow \text{is-type } G\ (\text{obj-ty } (\text{tag} = \text{oi}, \text{values} = \text{vs}))$   
 $\mid \text{Stat } C \Rightarrow \text{is-class } G\ C \rrbracket \implies$

$(x, \text{init-obj } G \text{ oi } \text{oref } s) :: \preceq (G, L)$   
 $\langle \text{proof} \rangle$

**lemma** *conforms-init-class-obj*:

$\llbracket (x, s) :: \preceq (G, L); \text{wf-prog } G; \text{class } G \text{ C} = \text{Some } y; \neg \text{init-ed } C \text{ (globs } s) \rrbracket \implies$   
 $(x, \text{init-class-obj } G \text{ C } s) :: \preceq (G, L)$   
 $\langle \text{proof} \rangle$

**lemma** *fst-init-lvars[simp]*:

$\text{fst } (\text{init-lvars } G \text{ C sig } (\text{invmode } m \text{ e}) \text{ a' pvs } (x, s)) =$   
 $(\text{if is-static } m \text{ then } x \text{ else } (\text{np } a') \text{ x})$   
 $\langle \text{proof} \rangle$

**lemma** *halloc-conforms*:  $\bigwedge s1. \llbracket G \vdash s1 \text{ -halloc } \text{oi} \succ a \rightarrow s2; \text{wf-prog } G; s1 :: \preceq (G, L);$   
 $\text{is-type } G \text{ (obj-ty } (\text{tag}=\text{oi}, \text{values}=\text{fs})) \rrbracket \implies s2 :: \preceq (G, L)$   
 $\langle \text{proof} \rangle$

**lemma** *halloc-type-sound*:

$\bigwedge s1. \llbracket G \vdash s1 \text{ -halloc } \text{oi} \succ a \rightarrow (x, s); \text{wf-prog } G; s1 :: \preceq (G, L);$   
 $T = \text{obj-ty } (\text{tag}=\text{oi}, \text{values}=\text{fs}); \text{is-type } G \text{ T} \rrbracket \implies$   
 $(x, s) :: \preceq (G, L) \wedge (x = \text{None} \longrightarrow G, s \vdash \text{Addr } a :: \preceq T)$   
 $\langle \text{proof} \rangle$

**lemma** *sxalloc-type-sound*:

$\bigwedge s1 \text{ s2}. \llbracket G \vdash s1 \text{ -sxalloc} \rightarrow s2; \text{wf-prog } G \rrbracket \implies$   
 $\text{case fst } s1 \text{ of}$   
 $\quad \text{None} \Rightarrow s2 = s1$   
 $\quad | \text{Some } \text{abr} \Rightarrow (\text{case } \text{abr} \text{ of}$   
 $\quad \quad \text{Xcpt } x \Rightarrow (\exists a. \text{fst } s2 = \text{Some}(\text{Xcpt } (\text{Loc } a)) \wedge$   
 $\quad \quad \quad (\forall L. s1 :: \preceq (G, L) \longrightarrow s2 :: \preceq (G, L)))$   
 $\quad \quad | \text{Jump } j \Rightarrow s2 = s1$   
 $\quad \quad | \text{Error } e \Rightarrow s2 = s1)$   
 $\langle \text{proof} \rangle$

**lemma** *wt-init-comp-ty*:

$\text{is-acc-type } G \text{ (pid } C) \text{ T} \implies (\text{prg}=\text{G}, \text{cls}=\text{C}, \text{lcl}=\text{L}) \vdash \text{init-comp-ty } T :: \checkmark$   
 $\langle \text{proof} \rangle$

**declare** *fun-upd-same* [simp]

**declare** *fun-upd-apply* [simp del]

**consts**

$\text{DynT-prop} :: [\text{prog}, \text{inv-mode}, \text{qtname}, \text{ref-ty}] \Rightarrow \text{bool}$   
 $(\vdash \longrightarrow \preceq \text{--} [\text{71}, \text{71}, \text{71}, \text{71}] \text{70})$   
 $G \vdash \text{mode} \rightarrow D \preceq t \equiv \text{mode} = \text{IntVir} \longrightarrow \text{is-class } G \text{ D} \wedge$   
 $(\text{if } (\exists T. t = \text{ArrayT } T) \text{ then } D = \text{Object} \text{ else } G \vdash \text{Class } D \preceq \text{RefT } t)$

**lemma** *DynT-propI*:

$\llbracket (x, s) :: \preceq (G, L); G, s \vdash a' :: \preceq \text{RefT } \text{statT}; \text{wf-prog } G; \text{mode} = \text{IntVir} \longrightarrow a' \neq \text{Null} \rrbracket$   
 $\implies G \vdash \text{mode} \rightarrow \text{invocation-class mode } s \ a' \ \text{statT} \preceq \text{statT}$   
 <proof>

**lemma** *invocation-method*:

$\llbracket \text{wf-prog } G; \text{statT} \neq \text{NullT};$   
 $(\forall \text{ statC}. \text{statT} = \text{ClassT } \text{statC} \longrightarrow \text{is-class } G \ \text{statC});$   
 $(\forall \ I. \text{statT} = \text{IfaceT } I \longrightarrow \text{is-iface } G \ I \wedge \text{mode} \neq \text{SuperM});$   
 $(\forall \ T. \text{statT} = \text{ArrayT } T \longrightarrow \text{mode} \neq \text{SuperM});$   
 $G \vdash \text{mode} \rightarrow \text{invocation-class mode } s \ a' \ \text{statT} \preceq \text{statT};$   
 $\text{dynlookup } G \ \text{statT} \ (\text{invocation-class mode } s \ a' \ \text{statT}) \ \text{sig} = \text{Some } m \rrbracket$   
 $\implies \text{methd } G \ (\text{invocation-declclass } G \ \text{mode } s \ a' \ \text{statT} \ \text{sig}) \ \text{sig} = \text{Some } m$   
 <proof>

**lemma** *DynT-mheadsD*:

$\llbracket G \vdash \text{invmode } sm \ e \rightarrow \text{invC} \preceq \text{statT};$   
 $\text{wf-prog } G; (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash e :: -\text{RefT } \text{statT};$   
 $(\text{statDeclT}, sm) \in \text{mheads } G \ C \ \text{statT} \ \text{sig};$   
 $\text{invC} = \text{invocation-class } (\text{invmode } sm \ e) \ s \ a' \ \text{statT};$   
 $\text{declC} = \text{invocation-declclass } G \ (\text{invmode } sm \ e) \ s \ a' \ \text{statT} \ \text{sig}$   
 $\rrbracket \implies$   
 $\exists \ dm.$   
 $\text{methd } G \ \text{declC} \ \text{sig} = \text{Some } dm \wedge \text{dynlookup } G \ \text{statT} \ \text{invC} \ \text{sig} = \text{Some } dm \wedge$   
 $G \vdash \text{resTy } (\text{methd } dm) \preceq \text{resTy } sm \wedge$   
 $\text{wf-mdecl } G \ \text{declC} \ (\text{sig}, \text{methd } dm) \wedge$   
 $\text{declC} = \text{declclass } dm \wedge$   
 $\text{is-static } dm = \text{is-static } sm \wedge$   
 $\text{is-class } G \ \text{invC} \wedge \text{is-class } G \ \text{declC} \wedge G \vdash \text{invC} \preceq_C \ \text{declC} \wedge$   
 $(\text{if } \text{invmode } sm \ e = \text{IntVir}$   
    $\text{then } (\forall \ \text{statC}. \text{statT} = \text{ClassT } \text{statC} \longrightarrow G \vdash \text{invC} \preceq_C \ \text{statC})$   
    $\text{else } ( (\exists \ \text{statC}. \text{statT} = \text{ClassT } \text{statC} \wedge G \vdash \text{statC} \preceq_C \ \text{declC})$   
      $\vee (\forall \ \text{statC}. \text{statT} \neq \text{ClassT } \text{statC} \wedge \text{declC} = \text{Object})) \wedge$   
      $\text{statDeclT} = \text{ClassT } (\text{declclass } dm))$   
 $\rrbracket \implies$   
 <proof>

**corollary** *DynT-mheadsE* [consumes 7]:

— Same as *DynT-mheadsD* but better suited for application in typesafety proof

**assumes** *invC-compatible*:  $G \vdash \text{mode} \rightarrow \text{invC} \preceq \text{statT}$

**and** *wf*:  $\text{wf-prog } G$

**and** *wt-e*:  $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash e :: -\text{RefT } \text{statT}$

**and** *mheads*:  $(\text{statDeclT}, sm) \in \text{mheads } G \ C \ \text{statT} \ \text{sig}$

**and** *mode*:  $\text{mode} = \text{invmode } sm \ e$

**and** *invC*:  $\text{invC} = \text{invocation-class mode } s \ a' \ \text{statT}$

**and** *declC*:  $\text{declC} = \text{invocation-declclass } G \ \text{mode } s \ a' \ \text{statT} \ \text{sig}$

**and** *dm*:  $\bigwedge \ dm. \llbracket \text{methd } G \ \text{declC} \ \text{sig} = \text{Some } dm;$

$\text{dynlookup } G \ \text{statT} \ \text{invC} \ \text{sig} = \text{Some } dm;$

$G \vdash \text{resTy } (\text{methd } dm) \preceq \text{resTy } sm;$

$\text{wf-mdecl } G \ \text{declC} \ (\text{sig}, \text{methd } dm);$

$\text{declC} = \text{declclass } dm;$

$\text{is-static } dm = \text{is-static } sm;$

$\text{is-class } G \ \text{invC}; \text{is-class } G \ \text{declC}; G \vdash \text{invC} \preceq_C \ \text{declC};$

$(\text{if } \text{invmode } sm \ e = \text{IntVir}$

$\text{then } (\forall \ \text{statC}. \text{statT} = \text{ClassT } \text{statC} \longrightarrow G \vdash \text{invC} \preceq_C \ \text{statC})$

$\text{else } ( (\exists \ \text{statC}. \text{statT} = \text{ClassT } \text{statC} \wedge G \vdash \text{statC} \preceq_C \ \text{declC})$

$\vee (\forall \ \text{statC}. \text{statT} \neq \text{ClassT } \text{statC} \wedge \text{declC} = \text{Object})) \wedge$

$\text{statDeclT} = \text{ClassT } (\text{declclass } dm)) \rrbracket \implies P$



**shows**  $P$   
 $\langle \text{proof} \rangle$

**lemma** *DynT-conf*:  $\llbracket G \vdash \text{invocation-class mode } s \text{ } a' \text{ } \text{statT} \preceq_C \text{ declC}; \text{wf-prog } G;$   
 $\text{isrtype } G \text{ } (\text{statT});$   
 $G, s \vdash a' :: \preceq \text{RefT } \text{statT}; \text{ mode } = \text{IntVir} \longrightarrow a' \neq \text{Null};$   
 $\text{mode} \neq \text{IntVir} \longrightarrow (\exists \text{ statC}. \text{statT} = \text{ClassT } \text{statC} \wedge G \vdash \text{statC} \preceq_C \text{ declC})$   
 $\vee (\forall \text{ statC}. \text{statT} \neq \text{ClassT } \text{statC} \wedge \text{declC} = \text{Object}) \rrbracket$   
 $\implies G, s \vdash a' :: \preceq \text{Class } \text{declC}$   
 $\langle \text{proof} \rangle$

**lemma** *Ass-lemma*:  
 $\llbracket G \vdash \text{Norm } s0 \text{ } -\text{var} \Rightarrow (w, f) \rightarrow \text{Norm } s1; G \vdash \text{Norm } s1 \text{ } -e-\succ v \rightarrow \text{Norm } s2;$   
 $G, s2 \vdash v :: \preceq eT; s1 \leq s2 \longrightarrow \text{assign } f \text{ } v \text{ } (\text{Norm } s2) :: \preceq (G, L) \rrbracket$   
 $\implies \text{assign } f \text{ } v \text{ } (\text{Norm } s2) :: \preceq (G, L) \wedge$   
 $(\text{normal } (\text{assign } f \text{ } v \text{ } (\text{Norm } s2))) \longrightarrow G, \text{store } (\text{assign } f \text{ } v \text{ } (\text{Norm } s2)) \vdash v :: \preceq eT$   
 $\langle \text{proof} \rangle$

**lemma** *Throw-lemma*:  $\llbracket G \vdash \text{tn} \preceq_C \text{SXcpt } \text{Throwable}; \text{wf-prog } G; (x1, s1) :: \preceq (G, L);$   
 $x1 = \text{None} \longrightarrow G, s1 \vdash a' :: \preceq \text{Class } \text{tn} \rrbracket \implies (\text{throw } a' \text{ } x1, s1) :: \preceq (G, L)$   
 $\langle \text{proof} \rangle$

**lemma** *Try-lemma*:  $\llbracket G \vdash \text{obj-ty } (\text{the } (\text{globs } s1' \text{ } (\text{Heap } a))) \preceq \text{Class } \text{tn};$   
 $(\text{Some } (\text{Xcpt } (\text{Loc } a)), s1') :: \preceq (G, L); \text{wf-prog } G \rrbracket$   
 $\implies \text{Norm } (\text{lupd}(vn \mapsto \text{Addr } a) \text{ } s1') :: \preceq (G, L(vn \mapsto \text{Class } \text{tn}))$   
 $\langle \text{proof} \rangle$

**lemma** *Fin-lemma*:  
 $\llbracket G \vdash \text{Norm } s1 \text{ } -c2 \rightarrow (x2, s2); \text{wf-prog } G; (\text{Some } a, s1) :: \preceq (G, L); (x2, s2) :: \preceq (G, L);$   
 $\text{dom } (\text{locals } s1) \subseteq \text{dom } (\text{locals } s2) \rrbracket$   
 $\implies (\text{abrupt-if } \text{True } (\text{Some } a) \text{ } x2, s2) :: \preceq (G, L)$   
 $\langle \text{proof} \rangle$

**lemma** *FVar-lemma1*:  
 $\llbracket \text{table-of } (\text{DeclConcepts.fields } G \text{ } \text{statC}) \text{ } (\text{fn}, \text{statDeclC}) = \text{Some } f ;$   
 $x2 = \text{None} \longrightarrow G, s2 \vdash a :: \preceq \text{Class } \text{statC}; \text{wf-prog } G; G \vdash \text{statC} \preceq_C \text{statDeclC};$   
 $\text{statDeclC} \neq \text{Object};$   
 $\text{class } G \text{ } \text{statDeclC} = \text{Some } y; (x2, s2) :: \preceq (G, L); s1 \leq s2;$   
 $\text{initd } \text{statDeclC } (\text{globs } s1);$   
 $(\text{if static } f \text{ then id else np } a) \text{ } x2 = \text{None} \rrbracket$   
 $\implies$   
 $\exists \text{ obj. } \text{globs } s2 \text{ } (\text{if static } f \text{ then Inr } \text{statDeclC} \text{ else Inl } (\text{the-Addr } a))$   
 $= \text{Some obj} \wedge$   
 $\text{var-tys } G \text{ } (\text{tag obj}) \text{ } (\text{if static } f \text{ then Inr } \text{statDeclC} \text{ else Inl } (\text{the-Addr } a))$   
 $(\text{Inl}(\text{fn}, \text{statDeclC})) = \text{Some } (\text{type } f)$   
 $\langle \text{proof} \rangle$

**lemma** *FVar-lemma2*: *error-free state*  
 $\implies \text{error-free}$   
 $(\text{assign}$   
 $(\lambda v. \text{supd}$

```

      (upd-gobj
       (if static field then Inr statDeclC
        else Inl (the-Addr a))
       (Inl (fn, statDeclC)) v))
    w state)
  <proof>

```

```

declare split-paired-All [simp del] split-paired-Ex [simp del]
declare split-if [split del] split-if-asm [split del]
      option.split [split del] option.split-asm [split del]
  <ML>

```

**lemma** *FVar-lemma*:

```

[[((v, f), Norm s2') = fvar statDeclC (static field) fn a (x2, s2);
  G ⊢ statC ≤C statDeclC;
  table-of (DeclConcepts.fields G statC) (fn, statDeclC) = Some field;
  wf-prog G;
  x2 = None ⟶ G, s2 ⊢ a :: ≤Class statC;
  statDeclC ≠ Object; class G statDeclC = Some y;
  (x2, s2) :: ≤1(G, L); s1 ≤ |s2; inited statDeclC (globs s1)]] ⟹
  G, s2 ⊢ v :: ≤type field ∧ s2' ≤ |f ≤type field :: ≤1(G, L)
  <proof>
declare split-paired-All [simp] split-paired-Ex [simp]
declare split-if [split] split-if-asm [split]
      option.split [split] option.split-asm [split]
  <ML>

```

**lemma** *AVar-lemma1*:  $\llbracket \text{globs } s \text{ (Inl } a) = \text{Some obj}; \text{tag obj} = \text{Arr ty } i; \text{the-Intg } i' \text{ in-bounds } i; \text{wf-prog } G; G \vdash \text{ty} \cdot [] \preceq \text{Tb} \cdot []; \text{Norm } s :: \preceq (G, L) \rrbracket \implies G, s \vdash \text{the } ((\text{values obj}) (\text{Inr } (\text{the-Intg } i')) :: \preceq \text{Tb}$

<proof>

**lemma** *obj-split*:  $\exists t \text{ vs. obj} = (\text{tag} = t, \text{values} = \text{vs})$

<proof>

**lemma** *AVar-lemma2*: *error-free state*

```

  ⟹ error-free
  (assign
   (λv (x, s').
    ((raise-if (¬ G, s ⊢ v fits T) ArrStore) x,
     upd-gobj (Inl a) (Inr (the-Intg i)) v s')))
  w state)
  <proof>

```

**lemma** *AVar-lemma*:  $\llbracket \text{wf-prog } G; G \vdash (x1, s1) -e2 \rightarrow i \rightarrow (x2, s2); \llbracket (v, f), \text{Norm } s2' \rrbracket = \text{avar } G \text{ } i \text{ } a (x2, s2); x1 = \text{None} \implies G, s1 \vdash a :: \preceq \text{Ta} \cdot []; (x2, s2) :: \preceq (G, L); s1 \leq |s2 \rrbracket \implies G, s2 \vdash v :: \preceq \text{Ta} \wedge s2' \leq |f \preceq \text{Ta} :: \preceq (G, L) \rrbracket$

<proof>

## Call

**lemma** *conforms-init-lvars-lemma*:  $\llbracket \text{wf-prog } G; \text{wf-mhead } G \text{ } P \text{ sig } mh; \text{list-all2 (conf } G \text{ } s) \text{ pvs } p\text{Tsa}; G \vdash p\text{Tsa} [\preceq] (\text{parTs sig}) \rrbracket \implies$

$G, s \vdash \text{empty } (\text{pars } mh[\mapsto] pvs)$   
 $[\sim::\preceq] \text{table-of lvars}(\text{pars } mh[\mapsto] parTs \text{ sig})$   
 $\langle \text{proof} \rangle$

**lemma** *lconf-map-lname* [simp]:  
 $G, s \vdash (\text{lname-case } l1 \ l2)[::\preceq](\text{lname-case } L1 \ L2)$   
 $=$   
 $(G, s \vdash l1[::\preceq] L1 \wedge G, s \vdash (\lambda x::unit . l2)[::\preceq](\lambda x::unit. L2))$   
 $\langle \text{proof} \rangle$

**lemma** *wlconf-map-lname* [simp]:  
 $G, s \vdash (\text{lname-case } l1 \ l2)[\sim::\preceq](\text{lname-case } L1 \ L2)$   
 $=$   
 $(G, s \vdash l1[\sim::\preceq] L1 \wedge G, s \vdash (\lambda x::unit . l2)[\sim::\preceq](\lambda x::unit. L2))$   
 $\langle \text{proof} \rangle$

**lemma** *lconf-map-ename* [simp]:  
 $G, s \vdash (\text{ename-case } l1 \ l2)[::\preceq](\text{ename-case } L1 \ L2)$   
 $=$   
 $(G, s \vdash l1[::\preceq] L1 \wedge G, s \vdash (\lambda x::unit. l2)[::\preceq](\lambda x::unit. L2))$   
 $\langle \text{proof} \rangle$

**lemma** *wlconf-map-ename* [simp]:  
 $G, s \vdash (\text{ename-case } l1 \ l2)[\sim::\preceq](\text{ename-case } L1 \ L2)$   
 $=$   
 $(G, s \vdash l1[\sim::\preceq] L1 \wedge G, s \vdash (\lambda x::unit. l2)[\sim::\preceq](\lambda x::unit. L2))$   
 $\langle \text{proof} \rangle$

**lemma** *defval-conf1* [rule-format (no-asm), elim]:  
 $\text{is-type } G \ T \longrightarrow (\exists v \in \text{Some } (\text{default-val } T): G, s \vdash v::\preceq T)$   
 $\langle \text{proof} \rangle$

**lemma** *np-no-jump*:  $x \neq \text{Some } (\text{Jump } j) \implies (\text{np } a') \ x \neq \text{Some } (\text{Jump } j)$   
 $\langle \text{proof} \rangle$

**declare** *split-paired-All* [simp del] *split-paired-Ex* [simp del]  
**declare** *split-if* [split del] *split-if-asm* [split del]  
 $\text{option.split}$  [split del]  $\text{option.split-asm}$  [split del]  
 $\langle \text{ML} \rangle$

**lemma** *conforms-init-lvars*:  
 $\llbracket \text{wf-mhead } G \ (\text{pid declC}) \ \text{sig } (\text{mhead } (\text{mthd } dm)); \text{wf-prog } G;$   
 $\text{list-all2 } (\text{conf } G \ s) \ pvs \ pTsa; \ G \vdash pTsa[\preceq](\text{parTs } \text{sig});$   
 $(x, s)::\preceq(G, L);$   
 $\text{methd } G \ \text{declC } \text{sig} = \text{Some } dm;$   
 $\text{isrtype } G \ \text{statT};$   
 $G \vdash \text{invC} \preceq_C \ \text{declC};$   
 $G, s \vdash a'::\preceq \text{RefT } \text{statT};$   
 $\text{invmode } (\text{mhd } sm) \ e = \text{IntVir} \longrightarrow a' \neq \text{Null};$   
 $\text{invmode } (\text{mhd } sm) \ e \neq \text{IntVir} \longrightarrow$

$(\exists \text{ statC}. \text{statT} = \text{ClassT statC} \wedge G \vdash \text{statC} \preceq_C \text{declC})$   
 $\vee (\forall \text{ statC}. \text{statT} \neq \text{ClassT statC} \wedge \text{declC} = \text{Object});$   
 $\text{invC} = \text{invocation-class (invmode (mhd sm) e) s a' statT};$   
 $\text{declC} = \text{invocation-declclass G (invmode (mhd sm) e) s a' statT sig};$   
 $x \neq \text{Some (Jump Ret)}$   
 $\parallel \Rightarrow$   
 $\text{init-lvars G declC sig (invmode (mhd sm) e) a'}$   
 $\text{pvs (x,s)::}\preceq(G, \lambda k.$   
 $\quad (\text{case k of}$   
 $\quad \quad \text{EName e} \Rightarrow (\text{case e of}$   
 $\quad \quad \quad \text{VName v}$   
 $\quad \quad \quad \Rightarrow (\text{table-of (lcls (mbody (mthd dm)))}$   
 $\quad \quad \quad \quad (\text{pars (mthd dm)} \mapsto \text{parTs sig})) \text{ v}$   
 $\quad \quad \quad | \text{Res} \Rightarrow \text{Some (resTy (mthd dm))})$   
 $\quad \quad | \text{This} \Rightarrow \text{if (is-static (mthd sm))}$   
 $\quad \quad \quad \text{then None else Some (Class declC))})$   
 $\langle \text{proof} \rangle$   
**declare** *split-paired-All* [simp] *split-paired-Ex* [simp]  
**declare** *split-if* [split] *split-if-asm* [split]  
 $\text{option.split [split] option.split-asm [split]}$   
 $\langle \text{ML} \rangle$

## 47 accessibility

**theorem** *dynamic-field-access-ok:*

**assumes** *wf*: *wf-prog G* **and**  
 $\text{not-Null: } \neg \text{stat} \longrightarrow a \neq \text{Null}$  **and**  
 $\text{conform-a: } G, (\text{store s}) \vdash a :: \preceq \text{Class statC}$  **and**  
 $\text{conform-s: } s :: \preceq (G, L)$  **and**  
 $\text{normal-s: normal s}$  **and**  
 $\text{wt-e: } (\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e :: \neg \text{Class statC}$  **and**  
 $f: \text{accfield } G \text{ accC statC fn} = \text{Some } f$  **and**  
 $\text{dynC: if stat then dynC = declclass f}$   
 $\quad \text{else dynC = obj-class (lookup-obj (store s) a)}$  **and**  
 $\text{stat: if stat then (is-static f) else } (\neg \text{is-static f})$   
**shows**  $\text{table-of (DeclConcepts.fields G dynC) (fn, declclass f) = Some (fld f)} \wedge$   
 $G \vdash \text{Field fn f in dynC dyn-accessible-from accC}$   
 $\langle \text{proof} \rangle$

**lemma** *error-free-field-access:*

**assumes** *accfield*:  $\text{accfield } G \text{ accC statC fn} = \text{Some (statDeclC, f)}$  **and**  
 $\text{wt-e: } (\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e :: \neg \text{Class statC}$  **and**  
 $\text{eval-init: } G \vdash \text{Norm s0} \neg \text{Init statDeclC} \rightarrow s1$  **and**  
 $\text{eval-e: } G \vdash s1 \neg e \rightarrow a \rightarrow s2$  **and**  
 $\text{conf-s2: } s2 :: \preceq (G, L)$  **and**  
 $\text{conf-a: normal s2} \implies G, \text{store s2} \vdash a :: \preceq \text{Class statC}$  **and**  
 $\text{fvar: (v, s2') = fvar statDeclC (is-static f) fn a s2}$  **and**  
 $\text{wf: wf-prog G}$   
**shows**  $\text{check-field-access } G \text{ accC statDeclC fn (is-static f) a s2' = s2'}$   
 $\langle \text{proof} \rangle$

**lemma** *call-access-ok:*

**assumes** *invC-prop*:  $G \vdash \text{invmode statM e} \rightarrow \text{invC} \preceq \text{statT}$   
**and**  $\text{wf: wf-prog G}$   
**and**  $\text{wt-e: } (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash e :: \neg \text{RefT statT}$   
**and**  $\text{statM: (statDeclT, statM) } \in \text{mheads } G \text{ accC statT sig}$   
**and**  $\text{invC: invC} = \text{invocation-class (invmode statM e) s a statT}$

**shows**  $\exists \text{ dynM}. \text{dynlookup } G \text{ statT invC sig} = \text{Some dynM} \wedge$   
 $G \vdash \text{Method sig dynM in invC dyn-accessible-from accC}$   
 $\langle \text{proof} \rangle$

**lemma** *error-free-call-access*:

**assumes**

*eval-args*:  $G \vdash s1 \multimap \text{args} \Rightarrow vs \rightarrow s2$  **and**

*wt-e*:  $(\text{prg} = G, \text{cls} = \text{accC}, \text{lcl} = L) \vdash e :: \neg(\text{RefT statT})$  **and**

*statM*:  $\text{max-spec } G \text{ accC statT } (\text{name} = mn, \text{parTs} = pTs)$   
 $= \{((\text{statDeclT}, \text{statM}), pTs')\}$  **and**

*conf-s2*:  $s2 :: \preceq(G, L)$  **and**

*conf-a*:  $\text{normal } s1 \Rightarrow G, \text{store } s1 \vdash a :: \preceq \text{RefT statT}$  **and**

*invProp*:  $\text{normal } s3 \Rightarrow$

$G \vdash \text{invmode statM } e \rightarrow \text{invC} \preceq \text{statT}$  **and**

*s3*:  $s3 = \text{init-lvars } G \text{ invDeclC } (\text{name} = mn, \text{parTs} = pTs')$

$(\text{invmode statM } e) \text{ a vs } s2$  **and**

*invC*:  $\text{invC} = \text{invocation-class } (\text{invmode statM } e) (\text{store } s2) \text{ a statT}$  **and**

*invDeclC*:  $\text{invDeclC} = \text{invocation-declclass } G (\text{invmode statM } e) (\text{store } s2)$   
 $\text{a statT } (\text{name} = mn, \text{parTs} = pTs')$  **and**

*wf*:  $\text{wf-prog } G$

**shows**  $\text{check-method-access } G \text{ accC statT } (\text{invmode statM } e) (\text{name} = mn, \text{parTs} = pTs') \text{ a } s3$   
 $= s3$

$\langle \text{proof} \rangle$

**lemma** *map-upds-eq-length-append-simp*:

$\bigwedge \text{tab } qs. \text{length } ps = \text{length } qs \Rightarrow \text{tab}(ps[\mapsto]qs @ zs) = \text{tab}(ps[\mapsto]qs)$

$\langle \text{proof} \rangle$

**lemma** *map-upds-upd-eq-length-simp*:

$\bigwedge \text{tab } qs \text{ x y. } \text{length } ps = \text{length } qs$

$\Rightarrow \text{tab}(ps[\mapsto]qs)(x \mapsto y) = \text{tab}(ps @ [x][\mapsto]qs @ [y])$

$\langle \text{proof} \rangle$

**lemma** *map-upd-cong*:  $\text{tab} = \text{tab}' \Rightarrow \text{tab}(x \mapsto y) = \text{tab}'(x \mapsto y)$

$\langle \text{proof} \rangle$

**lemma** *map-upd-cong-ext*:  $\text{tab } z = \text{tab}' z \Rightarrow (\text{tab}(x \mapsto y)) z = (\text{tab}'(x \mapsto y)) z$

$\langle \text{proof} \rangle$

**lemma** *map-upds-cong*:  $\text{tab} = \text{tab}' \Rightarrow \text{tab}(xs[\mapsto]ys) = \text{tab}'(xs[\mapsto]ys)$

$\langle \text{proof} \rangle$

**lemma** *map-upds-cong-ext*:

$\bigwedge \text{tab } \text{tab}' \text{ ys. } \text{tab } z = \text{tab}' z \Rightarrow (\text{tab}(xs[\mapsto]ys)) z = (\text{tab}'(xs[\mapsto]ys)) z$

$\langle \text{proof} \rangle$

**lemma** *map-upd-override*:  $(\text{tab}(x \mapsto y)) x = (\text{tab}'(x \mapsto y)) x$

$\langle \text{proof} \rangle$

**lemma** *map-upds-eq-length-suffix*:  $\bigwedge \text{tab } qs.$

$$\text{length } ps = \text{length } qs \implies \text{tab}(ps @ xs[\mapsto] qs) = \text{tab}(ps[\mapsto] qs)(xs[\mapsto][])$$

*<proof>*

**lemma** *map-upds-upds-eq-length-prefix-simp*:

$$\begin{aligned} & \bigwedge \text{tab } qs. \text{length } ps = \text{length } qs \\ & \implies \text{tab}(ps[\mapsto] qs)(xs[\mapsto] ys) = \text{tab}(ps @ xs[\mapsto] qs @ ys) \end{aligned}$$

*<proof>*

**lemma** *map-upd-cut-irrelevant*:

$$\begin{aligned} & \llbracket (\text{tab}(x \mapsto y)) \text{ vn} = \text{Some } el; (\text{tab}'(x \mapsto y)) \text{ vn} = \text{None} \rrbracket \\ & \implies \text{tab } vn = \text{Some } el \end{aligned}$$

*<proof>*

**lemma** *map-upd-Some-expand*:

$$\begin{aligned} & \llbracket \text{tab } vn = \text{Some } z \rrbracket \\ & \implies \exists z. (\text{tab}(x \mapsto y)) \text{ vn} = \text{Some } z \end{aligned}$$

*<proof>*

**lemma** *map-upds-Some-expand*:

$$\begin{aligned} & \bigwedge \text{tab } ys \text{ } z. \llbracket \text{tab } vn = \text{Some } z \rrbracket \\ & \implies \exists z. (\text{tab}(xs[\mapsto] ys)) \text{ vn} = \text{Some } z \end{aligned}$$

*<proof>*

**lemma** *map-upd-Some-swap*:

$$(\text{tab}(r \mapsto w)(u \mapsto v)) \text{ vn} = \text{Some } z \implies \exists z. (\text{tab}(u \mapsto v)(r \mapsto w)) \text{ vn} = \text{Some } z$$

*<proof>*

**lemma** *map-upd-None-swap*:

$$(\text{tab}(r \mapsto w)(u \mapsto v)) \text{ vn} = \text{None} \implies (\text{tab}(u \mapsto v)(r \mapsto w)) \text{ vn} = \text{None}$$

*<proof>*

**lemma** *map-eq-upd-eq*:  $\text{tab } vn = \text{tab}' \text{ vn} \implies (\text{tab}(x \mapsto y)) \text{ vn} = (\text{tab}'(x \mapsto y)) \text{ vn}$

*<proof>*

**lemma** *map-upd-in-expansion-map-swap*:

$$\begin{aligned} & \llbracket (\text{tab}(x \mapsto y)) \text{ vn} = \text{Some } z; \text{tab } vn \neq \text{Some } z \rrbracket \\ & \implies (\text{tab}'(x \mapsto y)) \text{ vn} = \text{Some } z \end{aligned}$$

*<proof>*

**lemma** *map-upds-in-expansion-map-swap*:

$$\begin{aligned} & \bigwedge \text{tab } \text{tab}' \text{ } ys \text{ } z. \llbracket (\text{tab}(xs[\mapsto] ys)) \text{ vn} = \text{Some } z; \text{tab } vn \neq \text{Some } z \rrbracket \\ & \implies (\text{tab}'(xs[\mapsto] ys)) \text{ vn} = \text{Some } z \end{aligned}$$

*<proof>*

**lemma** *map-upds-Some-swap*:

**assumes**  $r \mapsto u$ :  $(\text{tab}(r \mapsto w)(u \mapsto v)(xs \mapsto ys)) \text{ vn} = \text{Some } z$   
**shows**  $\exists z$ .  $(\text{tab}(u \mapsto v)(r \mapsto w)(xs \mapsto ys)) \text{ vn} = \text{Some } z$   
 $\langle \text{proof} \rangle$

**lemma** *map-upds-Some-insert*:  
**assumes**  $z$ :  $(\text{tab}(xs \mapsto ys)) \text{ vn} = \text{Some } z$   
**shows**  $\exists z$ .  $(\text{tab}(u \mapsto v)(xs \mapsto ys)) \text{ vn} = \text{Some } z$   
 $\langle \text{proof} \rangle$

**lemma** *map-upds-None-cut*:  
**assumes** *expand-None*:  $(\text{tab}(xs \mapsto ys)) \text{ vn} = \text{None}$   
**shows**  $\text{tab } \text{vn} = \text{None}$   
 $\langle \text{proof} \rangle$

**lemma** *map-upds-cut-irrelevant*:  
 $\bigwedge \text{tab } \text{tab}' \text{ ys}. \llbracket (\text{tab}(xs \mapsto ys)) \text{ vn} = \text{Some } \text{el}; (\text{tab}'(xs \mapsto ys)) \text{ vn} = \text{None} \rrbracket$   
 $\implies \text{tab } \text{vn} = \text{Some } \text{el}$   
 $\langle \text{proof} \rangle$

**lemma** *dom-vname-split*:  
 $\text{dom } (\text{lname-case } (\text{ename-case } (\text{tab}(x \mapsto y)(xs \mapsto ys)) \text{ a}) \text{ b})$   
 $= \text{dom } (\text{lname-case } (\text{ename-case } (\text{tab}(x \mapsto y)) \text{ a}) \text{ b}) \cup$   
 $\text{dom } (\text{lname-case } (\text{ename-case } (\text{tab}(xs \mapsto ys)) \text{ a}) \text{ b})$   
 $(\text{is } ?\text{List } x \text{ xs } y \text{ ys} = ?\text{Hd } x \text{ y} \cup ?\text{Tl } xs \text{ ys})$   
 $\langle \text{proof} \rangle$

**lemma** *dom-map-upd*:  $\bigwedge \text{tab}. \text{dom } (\text{tab}(x \mapsto y)) = \text{dom } \text{tab} \cup \{x\}$   
 $\langle \text{proof} \rangle$

**lemma** *dom-map-upds*:  $\bigwedge \text{tab } \text{ys}. \text{length } xs = \text{length } ys$   
 $\implies \text{dom } (\text{tab}(xs \mapsto ys)) = \text{dom } \text{tab} \cup \text{set } xs$   
 $\langle \text{proof} \rangle$

**lemma** *dom-ename-case-None-simp*:  
 $\text{dom } (\text{ename-case } \text{vname-tab } \text{None}) = \text{VName } ' (\text{dom } \text{vname-tab})$   
 $\langle \text{proof} \rangle$

**lemma** *dom-ename-case-Some-simp*:  
 $\text{dom } (\text{ename-case } \text{vname-tab } (\text{Some } a)) = \text{VName } ' (\text{dom } \text{vname-tab}) \cup \{\text{Res}\}$   
 $\langle \text{proof} \rangle$

**lemma** *dom-lname-case-None-simp*:  
 $\text{dom } (\text{lname-case } \text{ename-tab } \text{None}) = \text{EName } ' (\text{dom } \text{ename-tab})$   
 $\langle \text{proof} \rangle$

**lemma** *dom-lname-case-Some-simp*:  
 $\text{dom } (\text{lname-case } \text{ename-tab } (\text{Some } a)) = \text{EName } ' (\text{dom } \text{ename-tab}) \cup \{\text{This}\}$

$\langle \text{proof} \rangle$

**lemmas** *dom-lname-ename-case-simps* =  
     *dom-ename-case-None-simp dom-ename-case-Some-simp*  
     *dom-lname-case-None-simp dom-lname-case-Some-simp*

**lemma** *image-comp*:  
      $f \circ g \circ A = (f \circ g) \circ A$   
 $\langle \text{proof} \rangle$

**lemma** *dom-locals-init-lvars*:  
     **assumes**  $m: m = (\text{methd } (\text{the } (\text{methd } G \ C \ \text{sig})))$   
     **assumes**  $\text{len}: \text{length } (\text{pars } m) = \text{length } \text{pvs}$   
     **shows**  $\text{dom } (\text{locals } (\text{store } (\text{init-lvars } G \ C \ \text{sig } (\text{invmode } m \ e) \ a \ \text{pvs } s)))$   
          $= \text{parameters } m$   
 $\langle \text{proof} \rangle$

**lemma** *da-e2-BinOp*:  
     **assumes**  $\text{da}: (\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg \langle \text{BinOp binop } e1 \ e2 \rangle_e \gg A$   
     **and**  $\text{wt-e1}: (\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash e1 :: -e1T$   
     **and**  $\text{wt-e2}: (\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash e2 :: -e2T$   
     **and**  $\text{wt-binop}: \text{wt-binop } G \ \text{binop } e1T \ e2T$   
     **and**  $\text{conf-s0}: s0 :: \preceq (G, L)$   
     **and**  $\text{normal-s1}: \text{normal } s1$   
     **and**  $\text{eval-e1}: G \vdash s0 \ -e1 \rightarrow v1 \rightarrow s1$   
     **and**  $\text{conf-v1}: G, \text{store } s1 \vdash v1 :: \preceq e1T$   
     **and**  $\text{wf}: \text{wf-prog } G$   
     **shows**  $\exists \ E2. (\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s1))$   
          $\gg (\text{if need-second-arg binop } v1 \text{ then } \langle e2 \rangle_e \text{ else } \langle \text{Skip} \rangle_s) \gg E2$   
 $\langle \text{proof} \rangle$

## main proof of type safety

**lemma** *eval-type-sound*:  
     **assumes**  $\text{eval}: G \vdash s0 \ -t \rightarrow (v, s1)$   
     **and**  $\text{wt}: (\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash t :: T$   
     **and**  $\text{da}: (\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg t \gg A$   
     **and**  $\text{wf}: \text{wf-prog } G$   
     **and**  $\text{conf-s0}: s0 :: \preceq (G, L)$   
     **shows**  $s1 :: \preceq (G, L) \wedge (\text{normal } s1 \longrightarrow G, L, \text{store } s1 \vdash t \rightarrow v :: \preceq T) \wedge$   
          $(\text{error-free } s0 = \text{error-free } s1)$   
 $\langle \text{proof} \rangle$

**corollary** *eval-type-soundE* [consumes 5]:  
     **assumes**  $\text{eval}: G \vdash s0 \ -t \rightarrow (v, s1)$   
     **and**  $\text{conf}: s0 :: \preceq (G, L)$   
     **and**  $\text{wt}: (\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash t :: T$   
     **and**  $\text{da}: (\text{prg} = G, \text{cls} = \text{acc } C, \text{lcl} = L) \vdash \text{dom } (\text{locals } (\text{snd } s0)) \gg t \gg A$   
     **and**  $\text{wf}: \text{wf-prog } G$   
     **and**  $\text{elim}: \llbracket s1 :: \preceq (G, L); \text{normal } s1 \implies G, L, \text{snd } s1 \vdash t \rightarrow v :: \preceq T; \text{error-free } s0 = \text{error-free } s1 \rrbracket \implies P$   
     **shows**  $P$   
 $\langle \text{proof} \rangle$



**corollary** *eval-ts*:
$$\begin{aligned} & \llbracket G \vdash s - e \rightarrow v \rightarrow s'; \text{wf-prog } G; s::\preceq(G, L); (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash e::-T; \\ & \quad (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s)) \gg \text{In1 } e \gg A \rrbracket \\ \implies & \quad s'::\preceq(G, L) \wedge (\text{normal } s' \longrightarrow G, \text{store } s' \vdash v::\preceq T) \wedge \\ & \quad (\text{error-free } s = \text{error-free } s') \\ & \langle \text{proof} \rangle \end{aligned}$$
**corollary** *evals-ts*:
$$\begin{aligned} & \llbracket G \vdash s - es \rightarrow vs \rightarrow s'; \text{wf-prog } G; s::\preceq(G, L); (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash es::Ts; \\ & \quad (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s)) \gg \text{In3 } es \gg A \rrbracket \\ \implies & \quad s'::\preceq(G, L) \wedge (\text{normal } s' \longrightarrow \text{list-all2 } (\text{conf } G (\text{store } s')) \text{ vs } Ts) \wedge \\ & \quad (\text{error-free } s = \text{error-free } s') \\ & \langle \text{proof} \rangle \end{aligned}$$
**corollary** *evar-ts*:
$$\begin{aligned} & \llbracket G \vdash s - v \rightarrow vf \rightarrow s'; \text{wf-prog } G; s::\preceq(G, L); (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash v::T; \\ & \quad (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s)) \gg \text{In2 } v \gg A \rrbracket \implies \\ & \quad s'::\preceq(G, L) \wedge (\text{normal } s' \longrightarrow G, L, (\text{store } s') \vdash \text{In2 } v \gg \text{In2 } vf::\preceq \text{In1 } T) \wedge \\ & \quad (\text{error-free } s = \text{error-free } s') \\ & \langle \text{proof} \rangle \end{aligned}$$
**theorem** *exec-ts*:
$$\begin{aligned} & \llbracket G \vdash s - c \rightarrow s'; \text{wf-prog } G; s::\preceq(G, L); (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash c::\sqrt{}; \\ & \quad (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s)) \gg \text{In1r } c \gg A \rrbracket \\ \implies & \quad s'::\preceq(G, L) \wedge (\text{error-free } s \longrightarrow \text{error-free } s') \\ & \langle \text{proof} \rangle \end{aligned}$$
**lemma** *wf-eval-Fin*:

**assumes** *wf*: *wf-prog* *G*  
**and** *wt-c1*:  $(\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash \text{In1r } c1::\text{In1 } (\text{PrimT } \text{Void})$   
**and** *da-c1*:  $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } (\text{Norm } s0))) \gg \text{In1r } c1 \gg A$   
**and** *conf-s0*:  $\text{Norm } s0::\preceq(G, L)$   
**and** *eval-c1*:  $G \vdash \text{Norm } s0 - c1 \rightarrow (x1, s1)$   
**and** *eval-c2*:  $G \vdash \text{Norm } s1 - c2 \rightarrow s2$   
**and** *s3*:  $s3 = \text{abupd } (\text{abrupt-if } (x1 \neq \text{None}) \ x1) \ s2$   
**shows**  $G \vdash \text{Norm } s0 - c1 \text{ Finally } c2 \rightarrow s3$   
 $\langle \text{proof} \rangle$

## 48 Ideas for the future

In the type soundness proof and the correctness proof of definite assignment we perform induction on the evaluation relation with the further preconditions that the term is welltyped and definitely assigned. During the proofs we have to establish the welltypedness and definite assignment of the subterms to be able to apply the induction hypothesis. So large parts of both proofs are the same work in propagating welltypedness and definite assignment. So we can derive a new induction rule for induction on the evaluation of a wellformed term, were these propagations is already done, once and forever. Then we can do the proofs with this rule and can enjoy the time we have saved. Here is a first and incomplete sketch of such a rule.

**theorem** *wellformed-eval-induct* [*consumes 4, case-names Abrupt Skip Expr Lab Comp If*]:

**assumes** *eval*:  $G \vdash s0 - t \rightarrow (v, s1)$   
**and** *wt*:  $(\text{prg}=G, \text{cls}=\text{acc } C, \text{lcl}=L) \vdash t::T$   
**and** *da*:  $(\text{prg}=G, \text{cls}=\text{acc } C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s0)) \gg t \gg A$   
**and** *wf*: *wf-prog* *G*

**and** *abrupt*:  $\bigwedge s \ t \ abr \ L \ accC \ T \ A.$   
 $\llbracket (\langle prg = G, cls = accC, lcl = L \rangle) \vdash t :: T;$   
 $\langle prg = G, cls = accC, lcl = L \rangle \vdash \text{dom} (\text{locals} (\text{store} (\text{Some } abr, s))) \gg t \gg A$   
 $\rrbracket \implies P \ L \ accC \ (\text{Some } abr, s) \ t \ (\text{arbitrary3 } t) \ (\text{Some } abr, s)$   
**and** *skip*:  $\bigwedge s \ L \ accC. \ P \ L \ accC \ (\text{Norm } s) \ \langle \text{Skip} \rangle_s \ \Diamond \ (\text{Norm } s)$   
**and** *expr*:  $\bigwedge e \ s0 \ s1 \ v \ L \ accC \ eT \ E.$   
 $\llbracket (\langle prg = G, cls = accC, lcl = L \rangle) \vdash e :: -eT;$   
 $\langle prg = G, cls = accC, lcl = L \rangle$   
 $\vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0) :: \text{state}))) \gg \langle e \rangle_e \gg E;$   
 $P \ L \ accC \ (\text{Norm } s0) \ \langle e \rangle_e \ [v]_e \ s1 \rrbracket$   
 $\implies P \ L \ accC \ (\text{Norm } s0) \ \langle \text{Expr } e \rangle_s \ \Diamond \ s1$   
**and** *lab*:  $\bigwedge c \ l \ s0 \ s1 \ L \ accC \ C.$   
 $\llbracket (\langle prg = G, cls = accC, lcl = L \rangle) \vdash c :: \sqrt{};$   
 $\langle prg = G, cls = accC, lcl = L \rangle$   
 $\vdash \text{dom} (\text{locals} (\text{store} ((\text{Norm } s0) :: \text{state}))) \gg \langle c \rangle_s \gg C;$   
 $P \ L \ accC \ (\text{Norm } s0) \ \langle c \rangle_s \ \Diamond \ s1 \rrbracket$   
 $\implies P \ L \ accC \ (\text{Norm } s0) \ \langle l \cdot c \rangle_s \ \Diamond \ (\text{abupd} (\text{absorb } l) \ s1)$   
**and** *comp*:  $\bigwedge c1 \ c2 \ s0 \ s1 \ s2 \ L \ accC \ C1.$   
 $\llbracket G \vdash \text{Norm } s0 \ -c1 \rightarrow s1; G \vdash s1 \ -c2 \rightarrow s2;$   
 $\langle prg = G, cls = accC, lcl = L \rangle \vdash c1 :: \sqrt{};$   
 $\langle prg = G, cls = accC, lcl = L \rangle \vdash c2 :: \sqrt{};$   
 $\langle prg = G, cls = accC, lcl = L \rangle \vdash$   
 $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0) :: \text{state}))) \gg \langle c1 \rangle_s \gg C1;$   
 $P \ L \ accC \ (\text{Norm } s0) \ \langle c1 \rangle_s \ \Diamond \ s1;$   
 $\bigwedge Q. \llbracket \text{normal } s1;$   
 $\bigwedge C2. \llbracket (\langle prg = G, cls = accC, lcl = L \rangle)$   
 $\vdash \text{dom} (\text{locals} (\text{store } s1)) \gg \langle c2 \rangle_s \gg C2;$   
 $P \ L \ accC \ s1 \ \langle c2 \rangle_s \ \Diamond \ s2 \rrbracket \implies Q$   
 $\rrbracket \implies Q$   
 $\rrbracket \implies P \ L \ accC \ (\text{Norm } s0) \ \langle c1;; c2 \rangle_s \ \Diamond \ s2$   
**and** *if*:  $\bigwedge b \ c1 \ c2 \ e \ s0 \ s1 \ s2 \ L \ accC \ E.$   
 $\llbracket G \vdash \text{Norm } s0 \ -e \rightarrow b \rightarrow s1;$   
 $G \vdash s1 \ -(\text{if the-Bool } b \text{ then } c1 \text{ else } c2) \rightarrow s2;$   
 $\langle prg = G, cls = accC, lcl = L \rangle \vdash e :: -\text{PrimT Boolean};$   
 $\langle prg = G, cls = accC, lcl = L \rangle \vdash (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) :: \sqrt{};$   
 $\langle prg = G, cls = accC, lcl = L \rangle \vdash$   
 $\text{dom} (\text{locals} (\text{store} ((\text{Norm } s0) :: \text{state}))) \gg \langle e \rangle_e \gg E;$   
 $P \ L \ accC \ (\text{Norm } s0) \ \langle e \rangle_e \ [b]_e \ s1;$   
 $\bigwedge Q. \llbracket \text{normal } s1;$   
 $\bigwedge C. \llbracket (\langle prg = G, cls = accC, lcl = L \rangle) \vdash (\text{dom} (\text{locals} (\text{store } s1)))$   
 $\gg (\text{if the-Bool } b \text{ then } c1 \text{ else } c2)_s \gg C;$   
 $P \ L \ accC \ s1 \ \langle \text{if the-Bool } b \text{ then } c1 \text{ else } c2 \rangle_s \ \Diamond \ s2$   
 $\rrbracket \implies Q$   
 $\rrbracket \implies Q$   
 $\rrbracket \implies P \ L \ accC \ (\text{Norm } s0) \ \langle \text{If}(e) \ c1 \ \text{Else } c2 \rangle_s \ \Diamond \ s2$   
**shows**  $P \ L \ accC \ s0 \ t \ v \ s1$   
 $\langle \text{proof} \rangle$

**end**

## Chapter 20

## Evaln

## 49 Operational evaluation (big-step) semantics of Java expressions and statements

**theory** *Evaln* **imports** *TypeSafe* **begin**

Variant of *eval* relation with counter for bounded recursive depth. In principal *evaln* could replace *eval*.

Validity of the axiomatic semantics builds on *evaln*. For recursive method calls the axiomatic semantics rule assumes the method ok to derive a proof for the body. To prove the method rule sound we need to perform induction on the recursion depth. For the completeness proof of the axiomatic semantics the notion of the most general formula is used. The most general formula right now builds on the ordinary evaluation relation *eval*. So sometimes we have to switch between *evaln* and *eval* and vice versa. To make this switch easy *evaln* also does all the technical accessibility tests *check-field-access* and *check-method-access* like *eval*. If it would omit them *evaln* and *eval* would only be equivalent for welltyped, and definitely assigned terms.

**consts**

*evaln* :: *prog*  $\Rightarrow$  (*state*  $\times$  *term*  $\times$  *nat*  $\times$  *vals*  $\times$  *state*) *set*

**syntax**

*evaln* :: [*prog*, *state*, *term*, *nat*, *vals* \* *state*]  $\Rightarrow$  *bool*  
 (|- -> -> -> - [61,61,80, 61,61] 60)  
*evaln* :: [*prog*, *state*, *var* , *vvar* , *nat*, *state*]  $\Rightarrow$  *bool*  
 (|- -> -> -> - [61,61,90,61,61,61] 60)  
*eval-n*:: [*prog*, *state*, *expr* , *val* , *nat*, *state*]  $\Rightarrow$  *bool*  
 (|- -> -> -> - [61,61,80,61,61,61] 60)  
*evalsn*:: [*prog*, *state*, *expr list*, *val list*, *nat*, *state*]  $\Rightarrow$  *bool*  
 (|- -> -> -> - [61,61,61,61,61,61] 60)  
*execn* :: [*prog*, *state*, *stmt* , *nat*, *state*]  $\Rightarrow$  *bool*  
 (|- -> -> -> - [61,61,65, 61,61] 60)

**syntax** (*xsymbols*)

*evaln* :: [*prog*, *state*, *term*, *nat*, *vals*  $\times$  *state*]  $\Rightarrow$  *bool*  
 (+- -> -> -> - [61,61,80, 61,61] 60)  
*evaln* :: [*prog*, *state*, *var* , *vvar* , *nat*, *state*]  $\Rightarrow$  *bool*  
 (+- -> -> -> - [61,61,90,61,61,61] 60)  
*eval-n*:: [*prog*, *state*, *expr* , *val* , *nat*, *state*]  $\Rightarrow$  *bool*  
 (+- -> -> -> - [61,61,80,61,61,61] 60)  
*evalsn*:: [*prog*, *state*, *expr list*, *val list*, *nat*, *state*]  $\Rightarrow$  *bool*  
 (+- -> -> -> - [61,61,61,61,61,61] 60)  
*execn* :: [*prog*, *state*, *stmt* , *nat*, *state*]  $\Rightarrow$  *bool*  
 (+- -> -> -> - [61,61,65, 61,61] 60)

**translations**

$G \vdash s - t \quad \succ - n \rightarrow w \dashv s' \implies (s, t, n, w \dashv s') \in \text{evaln } G$   
 $G \vdash s - t \quad \succ - n \rightarrow (w, \quad s') \leq (s, t, n, w, \quad s') \in \text{evaln } G$   
 $G \vdash s - t \quad \succ - n \rightarrow (w, x, s') \leq (s, t, n, w, x, s') \in \text{evaln } G$   
 $G \vdash s - c \quad - n \rightarrow (x, s') \leq G \vdash s - \text{In1r } c \succ - n \rightarrow (\Diamond \quad, x, s')$   
 $G \vdash s - c \quad - n \rightarrow \quad s' \implies G \vdash s - \text{In1r } c \succ - n \rightarrow (\Diamond \quad, \quad s')$   
 $G \vdash s - e \succ v \quad - n \rightarrow (x, s') \leq G \vdash s - \text{In1l } e \succ - n \rightarrow (\text{In1 } v \quad, x, s')$   
 $G \vdash s - e \succ v \quad - n \rightarrow \quad s' \implies G \vdash s - \text{In1l } e \succ - n \rightarrow (\text{In1 } v \quad, \quad s')$   
 $G \vdash s - e \succ v f \quad - n \rightarrow (x, s') \leq G \vdash s - \text{In2 } e \succ - n \rightarrow (\text{In2 } v f, x, s')$   
 $G \vdash s - e \succ v f \quad - n \rightarrow \quad s' \implies G \vdash s - \text{In2 } e \succ - n \rightarrow (\text{In2 } v f, \quad s')$   
 $G \vdash s - e \doteq v \quad - n \rightarrow (x, s') \leq G \vdash s - \text{In3 } e \succ - n \rightarrow (\text{In3 } v \quad, x, s')$   
 $G \vdash s - e \doteq v \quad - n \rightarrow \quad s' \implies G \vdash s - \text{In3 } e \succ - n \rightarrow (\text{In3 } v \quad, \quad s')$

**inductive evaln G intros**

— propagation of abrupt completion

$$Abrupt: \quad G \vdash (Some\ xc, s) - t \succ - n \rightarrow (arbitrary3\ t, (Some\ xc, s))$$

— evaluation of variables

$$LVar: \quad G \vdash Norm\ s - LVar\ vn \Rightarrow lvar\ vn\ s - n \rightarrow Norm\ s$$

$$\begin{aligned} FVar: \quad & \llbracket G \vdash Norm\ s0 - Init\ statDeclC - n \rightarrow s1; G \vdash s1 - e - \succ a - n \rightarrow s2; \\ & (v, s2') = fvar\ statDeclC\ stat\ fn\ a\ s2; \\ & s3 = check-field-access\ G\ accC\ statDeclC\ fn\ stat\ a\ s2' \rrbracket \implies \\ & G \vdash Norm\ s0 - \{accC, statDeclC, stat\}e..fn \Rightarrow v - n \rightarrow s3 \end{aligned}$$

$$\begin{aligned} AVar: \quad & \llbracket G \vdash Norm\ s0 - e1 - \succ a - n \rightarrow s1; G \vdash s1 - e2 - \succ i - n \rightarrow s2; \\ & (v, s2') = avar\ G\ i\ a\ s2 \rrbracket \implies \\ & G \vdash Norm\ s0 - e1.[e2] \Rightarrow v - n \rightarrow s2' \end{aligned}$$

— evaluation of expressions

$$\begin{aligned} NewC: \quad & \llbracket G \vdash Norm\ s0 - Init\ C - n \rightarrow s1; \\ & G \vdash s1 - halloc\ (CInst\ C) \succ a \rightarrow s2 \rrbracket \implies \\ & G \vdash Norm\ s0 - NewC\ C - \succ Addr\ a - n \rightarrow s2 \end{aligned}$$

$$\begin{aligned} NewA: \quad & \llbracket G \vdash Norm\ s0 - init-comp-ty\ T - n \rightarrow s1; G \vdash s1 - e - \succ i' - n \rightarrow s2; \\ & G \vdash abupd\ (check-neg\ i')\ s2 - halloc\ (Arr\ T\ (the-Intg\ i')) \succ a \rightarrow s3 \rrbracket \implies \\ & G \vdash Norm\ s0 - New\ T[e] - \succ Addr\ a - n \rightarrow s3 \end{aligned}$$

$$\begin{aligned} Cast: \quad & \llbracket G \vdash Norm\ s0 - e - \succ v - n \rightarrow s1; \\ & s2 = abupd\ (raise-if\ (\neg G, snd\ s1 \vdash v\ fits\ T)\ ClassCast)\ s1 \rrbracket \implies \\ & G \vdash Norm\ s0 - Cast\ T\ e - \succ v - n \rightarrow s2 \end{aligned}$$

$$\begin{aligned} Inst: \quad & \llbracket G \vdash Norm\ s0 - e - \succ v - n \rightarrow s1; \\ & b = (v \neq Null \wedge G, store\ s1 \vdash v\ fits\ RefT\ T) \rrbracket \implies \\ & G \vdash Norm\ s0 - e\ InstOf\ T - \succ Bool\ b - n \rightarrow s1 \end{aligned}$$

$$Lit: \quad G \vdash Norm\ s - Lit\ v - \succ v - n \rightarrow Norm\ s$$

$$\begin{aligned} UnOp: \quad & \llbracket G \vdash Norm\ s0 - e - \succ v - n \rightarrow s1 \rrbracket \\ & \implies G \vdash Norm\ s0 - UnOp\ unop\ e - \succ (eval-unop\ unop\ v) - n \rightarrow s1 \end{aligned}$$

$$\begin{aligned} BinOp: \quad & \llbracket G \vdash Norm\ s0 - e1 - \succ v1 - n \rightarrow s1; \\ & G \vdash s1 - (if\ need-second-arg\ binop\ v1\ then\ (In1l\ e2)\ else\ (In1r\ Skip)) \\ & \quad \succ - n \rightarrow (In1\ v2, s2) \rrbracket \\ & \implies G \vdash Norm\ s0 - BinOp\ binop\ e1\ e2 - \succ (eval-binop\ binop\ v1\ v2) - n \rightarrow s2 \end{aligned}$$

$$Super: \quad G \vdash Norm\ s - Super - \succ val-this\ s - n \rightarrow Norm\ s$$

$$\begin{aligned} Acc: \quad & \llbracket G \vdash Norm\ s0 - va \Rightarrow (v, f) - n \rightarrow s1 \rrbracket \implies \\ & G \vdash Norm\ s0 - Acc\ va - \succ v - n \rightarrow s1 \end{aligned}$$

$$Ass: \quad \llbracket G \vdash Norm\ s0 - va \Rightarrow (w, f) - n \rightarrow s1; \rrbracket$$

$$G \vdash \quad s1 \text{ --} e \text{--} \succ v \quad \text{--} n \rightarrow s2 \rVert \implies \\ G \vdash \text{Norm } s0 \text{ --} va := e \text{--} \succ v \text{--} n \rightarrow \text{assign } f \ v \ s2$$

$$\text{Cond: } \rVert G \vdash \text{Norm } s0 \text{ --} e0 \text{--} \succ b \text{--} n \rightarrow s1; \\ G \vdash \quad s1 \text{ --} (\text{if the-Bool } b \text{ then } e1 \text{ else } e2) \text{--} \succ v \text{--} n \rightarrow s2 \rVert \implies \\ G \vdash \text{Norm } s0 \text{ --} e0 \text{ ? } e1 : e2 \text{--} \succ v \text{--} n \rightarrow s2$$

*Call:*

$$\rVert G \vdash \text{Norm } s0 \text{ --} e \text{--} \succ a' \text{--} n \rightarrow s1; G \vdash s1 \text{ --} args \dot{=} \succ vs \text{--} n \rightarrow s2; \\ D = \text{invocation-declclass } G \text{ mode } (store \ s2) \ a' \ statT \ (\llbracket name = mn, parTs = pTs \rrbracket); \\ s3 = \text{init-lvars } G \ D \ (\llbracket name = mn, parTs = pTs \rrbracket) \ mode \ a' \ vs \ s2; \\ s3' = \text{check-method-access } G \ accC \ statT \ mode \ (\llbracket name = mn, parTs = pTs \rrbracket) \ a' \ s3; \\ G \vdash s3' \text{--} Methd \ D \ (\llbracket name = mn, parTs = pTs \rrbracket) \text{--} \succ v \text{--} n \rightarrow s4 \\ \rVert \\ \implies \\ G \vdash \text{Norm } s0 \text{ --} \{accC, statT, mode\} e.mn(\{pTs\}args) \text{--} \succ v \text{--} n \rightarrow (\text{restore-lvars } s2 \ s4)$$

$$\text{Methd: } \rVert G \vdash \text{Norm } s0 \text{ --} body \ G \ D \ sig \text{--} \succ v \text{--} n \rightarrow s1 \rVert \implies \\ G \vdash \text{Norm } s0 \text{ --} Methd \ D \ sig \text{--} \succ v \text{--} Suc \ n \rightarrow s1$$

$$\text{Body: } \rVert G \vdash \text{Norm } s0 \text{--} Init \ D \text{--} n \rightarrow s1; G \vdash s1 \text{ --} c \text{--} n \rightarrow s2; \\ s3 = (\text{if } (\exists \ l. \text{ abrupt } s2 = \text{Some } (\text{Jump } (\text{Break } l))) \vee \\ \text{abrupt } s2 = \text{Some } (\text{Jump } (\text{Cont } l))) \\ \text{then } abupd \ (\lambda \ x. \text{Some } (\text{Error CrossMethodJump})) \ s2 \\ \text{else } s2 \rVert \implies \\ G \vdash \text{Norm } s0 \text{--} Body \ D \ c \\ \text{--} \succ the \ (locals \ (store \ s2) \ Result) \text{--} n \rightarrow abupd \ (\text{absorb } Ret) \ s3$$

— evaluation of expression lists

*Nil:*

$$G \vdash \text{Norm } s0 \text{ --} [] \dot{=} \succ [] \text{--} n \rightarrow \text{Norm } s0$$

$$\text{Cons: } \rVert G \vdash \text{Norm } s0 \text{ --} e \text{--} \succ v \text{--} n \rightarrow s1; \\ G \vdash \quad s1 \text{ --} es \dot{=} \succ vs \text{--} n \rightarrow s2 \rVert \implies \\ G \vdash \text{Norm } s0 \text{ --} e \# es \dot{=} \succ v \# vs \text{--} n \rightarrow s2$$

— execution of statements

$$\text{Skip: } G \vdash \text{Norm } s \text{--} Skip \text{--} n \rightarrow \text{Norm } s$$

$$\text{Expr: } \rVert G \vdash \text{Norm } s0 \text{ --} e \text{--} \succ v \text{--} n \rightarrow s1 \rVert \implies \\ G \vdash \text{Norm } s0 \text{ --} Expr \ e \text{--} n \rightarrow s1$$

$$\text{Lab: } \rVert G \vdash \text{Norm } s0 \text{ --} c \text{--} n \rightarrow s1 \rVert \implies \\ G \vdash \text{Norm } s0 \text{ --} l \cdot c \text{--} n \rightarrow abupd \ (\text{absorb } l) \ s1$$

$$\text{Comp: } \rVert G \vdash \text{Norm } s0 \text{ --} c1 \text{--} n \rightarrow s1; \\ G \vdash \quad s1 \text{ --} c2 \text{--} n \rightarrow s2 \rVert \implies \\ G \vdash \text{Norm } s0 \text{ --} c1;; c2 \text{--} n \rightarrow s2$$

$$\text{If: } \rVert G \vdash \text{Norm } s0 \text{ --} e \text{--} \succ b \text{--} n \rightarrow s1; \\ G \vdash \quad s1 \text{ --} (\text{if the-Bool } b \text{ then } c1 \text{ else } c2) \text{--} n \rightarrow s2 \rVert \implies \\ G \vdash \text{Norm } s0 \text{ --} If(e) \ c1 \ Else \ c2 \text{--} n \rightarrow s2$$

$$\text{Loop: } \rVert G \vdash \text{Norm } s0 \text{ --} e \text{--} \succ b \text{--} n \rightarrow s1; \\ \text{if the-Bool } b \\ \text{then } (G \vdash s1 \text{ --} c \text{--} n \rightarrow s2 \wedge$$

$$\begin{aligned} & G \vdash (abupd \ (absorb \ (Cont \ l)) \ s2) \ -l \cdot \ While(e) \ c \ -n \rightarrow s3 \\ & \text{else } s3 = s1 \parallel \implies \\ & G \vdash Norm \ s0 \ -l \cdot \ While(e) \ c \ -n \rightarrow s3 \end{aligned}$$

$$Jmp: G \vdash Norm \ s \ -Jmp \ j \ -n \rightarrow (Some \ (Jump \ j), \ s)$$

$$\begin{aligned} & Throw: \parallel G \vdash Norm \ s0 \ -e \ -> a' \ -n \rightarrow s1 \parallel \implies \\ & G \vdash Norm \ s0 \ -Throw \ e \ -n \rightarrow abupd \ (throw \ a') \ s1 \end{aligned}$$

$$\begin{aligned} & Try: \parallel G \vdash Norm \ s0 \ -c1 \ -n \rightarrow s1; \ G \vdash s1 \ -xalloc \rightarrow s2; \\ & \text{if } G, s2 \vdash catch \ tn \ \text{then } G \vdash new\text{-}xcpt\text{-}var \ vn \ s2 \ -c2 \ -n \rightarrow s3 \ \text{else } s3 = s2 \parallel \\ & \implies \\ & G \vdash Norm \ s0 \ -Try \ c1 \ Catch(tn \ vn) \ c2 \ -n \rightarrow s3 \end{aligned}$$

$$\begin{aligned} & Fin: \parallel G \vdash Norm \ s0 \ -c1 \ -n \rightarrow (x1, s1); \\ & G \vdash Norm \ s1 \ -c2 \ -n \rightarrow s2; \\ & s3 = (\text{if } (\exists \ err. \ x1 = Some \ (Error \ err)) \\ & \text{then } (x1, s1) \\ & \text{else } abupd \ (abrupt\text{-}if \ (x1 \neq None) \ x1) \ s2) \parallel \implies \\ & G \vdash Norm \ s0 \ -c1 \ Finally \ c2 \ -n \rightarrow s3 \end{aligned}$$

$$\begin{aligned} & Init: \parallel the \ (class \ G \ C) = c; \\ & \text{if } initd \ C \ (globs \ s0) \ \text{then } s3 = Norm \ s0 \\ & \text{else } (G \vdash Norm \ (init\text{-}class\text{-}obj \ G \ C \ s0) \\ & \quad -(\text{if } C = Object \ \text{then } Skip \ \text{else } Init \ (super \ c)) \ -n \rightarrow s1 \ \wedge \\ & \quad G \vdash set\text{-}lvars \ empty \ s1 \ -init \ c \ -n \rightarrow s2 \ \wedge \\ & \quad s3 = restore\text{-}lvars \ s1 \ s2) \parallel \\ & \implies \\ & G \vdash Norm \ s0 \ -Init \ C \ -n \rightarrow s3 \end{aligned}$$

**monos**

*if-def2*

**declare** *split-if*  $[split \ del]$  *split-if-asm*  $[split \ del]$   
*option.split*  $[split \ del]$  *option.split-asm*  $[split \ del]$   
*not-None-eq*  $[simp \ del]$   
*split-paired-All*  $[simp \ del]$  *split-paired-Ex*  $[simp \ del]$

$\langle ML \rangle$

**inductive-cases** *evaln-cases*:  $G \vdash s \ -t \ -n \rightarrow vs'$

**inductive-cases** *evaln-elim-cases*:

$G \vdash (Some \ xc, \ s) \ -t$	$\succ -n \rightarrow vs'$
$G \vdash Norm \ s \ -In1r \ Skip$	$\succ -n \rightarrow xs'$
$G \vdash Norm \ s \ -In1r \ (Jmp \ j)$	$\succ -n \rightarrow xs'$
$G \vdash Norm \ s \ -In1r \ (l \cdot \ c)$	$\succ -n \rightarrow xs'$
$G \vdash Norm \ s \ -In3 \ (\Box)$	$\succ -n \rightarrow vs'$
$G \vdash Norm \ s \ -In3 \ (e \# es)$	$\succ -n \rightarrow vs'$
$G \vdash Norm \ s \ -In1l \ (Lit \ w)$	$\succ -n \rightarrow vs'$
$G \vdash Norm \ s \ -In1l \ (UnOp \ unop \ e)$	$\succ -n \rightarrow vs'$
$G \vdash Norm \ s \ -In1l \ (BinOp \ binop \ e1 \ e2)$	$\succ -n \rightarrow vs'$
$G \vdash Norm \ s \ -In2 \ (LVar \ vn)$	$\succ -n \rightarrow vs'$
$G \vdash Norm \ s \ -In1l \ (Cast \ T \ e)$	$\succ -n \rightarrow vs'$
$G \vdash Norm \ s \ -In1l \ (e \ InstOf \ T)$	$\succ -n \rightarrow vs'$
$G \vdash Norm \ s \ -In1l \ (Super)$	$\succ -n \rightarrow vs'$
$G \vdash Norm \ s \ -In1l \ (Acc \ va)$	$\succ -n \rightarrow vs'$
$G \vdash Norm \ s \ -In1r \ (Expr \ e)$	$\succ -n \rightarrow xs'$
$G \vdash Norm \ s \ -In1r \ (c1 ;; c2)$	$\succ -n \rightarrow xs'$
$G \vdash Norm \ s \ -In1l \ (Methd \ C \ sig)$	$\succ -n \rightarrow xs'$
$G \vdash Norm \ s \ -In1l \ (Body \ D \ c)$	$\succ -n \rightarrow xs'$

$G \vdash \text{Norm } s - \text{In1l } (e0 \text{ ? } e1 : e2)$	$\succ -n \rightarrow vs'$
$G \vdash \text{Norm } s - \text{In1r } (\text{If}(e) \ c1 \ \text{Else } c2)$	$\succ -n \rightarrow xs'$
$G \vdash \text{Norm } s - \text{In1r } (l \bullet \text{While}(e) \ c)$	$\succ -n \rightarrow xs'$
$G \vdash \text{Norm } s - \text{In1r } (c1 \ \text{Finally } c2)$	$\succ -n \rightarrow xs'$
$G \vdash \text{Norm } s - \text{In1r } (\text{Throw } e)$	$\succ -n \rightarrow xs'$
$G \vdash \text{Norm } s - \text{In1l } (\text{NewC } C)$	$\succ -n \rightarrow vs'$
$G \vdash \text{Norm } s - \text{In1l } (\text{New } T[e])$	$\succ -n \rightarrow vs'$
$G \vdash \text{Norm } s - \text{In1l } (\text{Ass } va \ e)$	$\succ -n \rightarrow vs'$
$G \vdash \text{Norm } s - \text{In1r } (\text{Try } c1 \ \text{Catch}(tn \ vn) \ c2)$	$\succ -n \rightarrow xs'$
$G \vdash \text{Norm } s - \text{In2 } (\{accC, statDeclC, stat\}e..fn)$	$\succ -n \rightarrow vs'$
$G \vdash \text{Norm } s - \text{In2 } (e1.[e2])$	$\succ -n \rightarrow vs'$
$G \vdash \text{Norm } s - \text{In1l } (\{accC, statT, mode\}e.mn(\{pT\}p))$	$\succ -n \rightarrow vs'$
$G \vdash \text{Norm } s - \text{In1r } (\text{Init } C)$	$\succ -n \rightarrow xs'$
$G \vdash \text{Norm } s - \text{In1r } (\text{Init } C)$	$\succ -n \rightarrow xs'$

**declare** *split-if* [split] *split-if-asm* [split]  
*option.split* [split] *option.split-asm* [split]  
*not-None-eq* [simp]  
*split-paired-All* [simp] *split-paired-Ex* [simp]  
 <ML>

**lemma** *evaln-Inj-elim*:  $G \vdash s - t \succ -n \rightarrow (w, s') \implies \text{case } t \text{ of } \text{In1 } ec \Rightarrow$   
 $(\text{case } ec \text{ of } \text{Inl } e \Rightarrow (\exists v. w = \text{In1 } v) \mid \text{Inr } c \Rightarrow w = \Diamond)$   
 $\mid \text{In2 } e \Rightarrow (\exists v. w = \text{In2 } v) \mid \text{In3 } e \Rightarrow (\exists v. w = \text{In3 } v)$   
 <proof>

The following simplification procedures set up the proper injections of terms and their corresponding values in the evaluation relation: E.g. an expression (injection *In1l* into terms) always evaluates to ordinary values (injection *In1* into generalised values *vals*).

<ML>  
**declare** *evaln-AbruptIs* [intro!]

**lemma** *evaln-Callee*:  $G \vdash \text{Norm } s - \text{In1l } (\text{Callee } l \ e) \succ -n \rightarrow (v, s') = \text{False}$   
 <proof>

**lemma** *evaln-InsInitE*:  $G \vdash \text{Norm } s - \text{In1l } (\text{InsInitE } c \ e) \succ -n \rightarrow (v, s') = \text{False}$   
 <proof>

**lemma** *evaln-InsInitV*:  $G \vdash \text{Norm } s - \text{In2 } (\text{InsInitV } c \ w) \succ -n \rightarrow (v, s') = \text{False}$   
 <proof>

**lemma** *evaln-FinA*:  $G \vdash \text{Norm } s - \text{In1r } (\text{FinA } a \ c) \succ -n \rightarrow (v, s') = \text{False}$   
 <proof>

**lemma** *evaln-abrupt-lemma*:  $G \vdash s - e \succ -n \rightarrow (v, s') \implies$   
 $\text{fst } s = \text{Some } xc \longrightarrow s' = s \wedge v = \text{arbitrary3 } e$   
 <proof>

**lemma** *evaln-abrupt*:  
 $\wedge s'. G \vdash (\text{Some } xc, s) - e \succ -n \rightarrow (w, s') = (s' = (\text{Some } xc, s) \wedge$   
 $w = \text{arbitrary3 } e \wedge G \vdash (\text{Some } xc, s) - e \succ -n \rightarrow (\text{arbitrary3 } e, (\text{Some } xc, s)))$   
 <proof>



$\langle ML \rangle$

**lemma** *evaln-LitI*:  $G \vdash s - \text{Lit } v \multimap (\text{if normal } s \text{ then } v \text{ else arbitrary}) - n \rightarrow s$   
 $\langle \text{proof} \rangle$

**lemma** *CondI*:  
 $\bigwedge s1. \llbracket G \vdash s - e \multimap b - n \rightarrow s1; G \vdash s1 - (\text{if the-Bool } b \text{ then } e1 \text{ else } e2) \multimap v - n \rightarrow s2 \rrbracket \implies$   
 $G \vdash s - e \text{ ? } e1 : e2 \multimap (\text{if normal } s1 \text{ then } v \text{ else arbitrary}) - n \rightarrow s2$   
 $\langle \text{proof} \rangle$

**lemma** *evaln-SkipI* [intro!]:  $G \vdash s - \text{Skip} - n \rightarrow s$   
 $\langle \text{proof} \rangle$

**lemma** *evaln-ExprI*:  $G \vdash s - e \multimap v - n \rightarrow s' \implies G \vdash s - \text{Expr } e - n \rightarrow s'$   
 $\langle \text{proof} \rangle$

**lemma** *evaln-CompI*:  $\llbracket G \vdash s - c1 - n \rightarrow s1; G \vdash s1 - c2 - n \rightarrow s2 \rrbracket \implies G \vdash s - c1;; c2 - n \rightarrow s2$   
 $\langle \text{proof} \rangle$

**lemma** *evaln-IfI*:  
 $\llbracket G \vdash s - e \multimap v - n \rightarrow s1; G \vdash s1 - (\text{if the-Bool } v \text{ then } c1 \text{ else } c2) - n \rightarrow s2 \rrbracket \implies$   
 $G \vdash s - \text{If}(e) \text{ } c1 \text{ Else } c2 - n \rightarrow s2$   
 $\langle \text{proof} \rangle$

**lemma** *evaln-SkipD* [dest!]:  $G \vdash s - \text{Skip} - n \rightarrow s' \implies s' = s$   
 $\langle \text{proof} \rangle$

**lemma** *evaln-Skip-eq* [simp]:  $G \vdash s - \text{Skip} - n \rightarrow s' = (s = s')$   
 $\langle \text{proof} \rangle$

## evaln implies eval

**lemma** *evaln-eval*:  
**assumes** *evaln*:  $G \vdash s0 - t \multimap - n \rightarrow (v, s1)$   
**shows**  $G \vdash s0 - t \multimap - n \rightarrow (v, s1)$   
 $\langle \text{proof} \rangle$

**lemma** *Suc-le-D-lemma*:  $\llbracket \text{Suc } n \leq m'; (\bigwedge m. n \leq m \implies P (\text{Suc } m)) \rrbracket \implies P m'$   
 $\langle \text{proof} \rangle$

**lemma** *evaln-nonstrict* [rule-format (no-asm), elim]:  
 $\bigwedge ws. G \vdash s - t \multimap - n \rightarrow ws \implies \forall m. n \leq m \longrightarrow G \vdash s - t \multimap - m \rightarrow ws$   
 $\langle \text{proof} \rangle$

**lemmas** *evaln-nonstrict-Suc* = *evaln-nonstrict* [OF - le-refl [THEN le-SucI]]

**lemma** *evaln-max2*:  $\llbracket G \vdash s1 - t1 \multimap - n1 \rightarrow ws1; G \vdash s2 - t2 \multimap - n2 \rightarrow ws2 \rrbracket \implies$

$G \vdash s1 \text{ } -t1 \succ -\max n1 \ n2 \rightarrow ws1 \ \wedge \ G \vdash s2 \text{ } -t2 \succ -\max n1 \ n2 \rightarrow ws2$

$\langle \text{proof} \rangle$

**corollary** *evaln-max2E* [consumes 2]:

$\llbracket G \vdash s1 \text{ } -t1 \succ -n1 \rightarrow ws1; G \vdash s2 \text{ } -t2 \succ -n2 \rightarrow ws2; \\ G \vdash s1 \text{ } -t1 \succ -\max n1 \ n2 \rightarrow ws1; G \vdash s2 \text{ } -t2 \succ -\max n1 \ n2 \rightarrow ws2 \rrbracket \implies P \rrbracket \implies P$

$\langle \text{proof} \rangle$

**lemma** *evaln-max3*:

$\llbracket G \vdash s1 \text{ } -t1 \succ -n1 \rightarrow ws1; G \vdash s2 \text{ } -t2 \succ -n2 \rightarrow ws2; G \vdash s3 \text{ } -t3 \succ -n3 \rightarrow ws3 \rrbracket \implies \\ G \vdash s1 \text{ } -t1 \succ -\max (\max n1 \ n2) \ n3 \rightarrow ws1 \ \wedge \\ G \vdash s2 \text{ } -t2 \succ -\max (\max n1 \ n2) \ n3 \rightarrow ws2 \ \wedge \\ G \vdash s3 \text{ } -t3 \succ -\max (\max n1 \ n2) \ n3 \rightarrow ws3$

$\langle \text{proof} \rangle$

**corollary** *evaln-max3E*:

$\llbracket G \vdash s1 \text{ } -t1 \succ -n1 \rightarrow ws1; G \vdash s2 \text{ } -t2 \succ -n2 \rightarrow ws2; G \vdash s3 \text{ } -t3 \succ -n3 \rightarrow ws3; \\ \llbracket G \vdash s1 \text{ } -t1 \succ -\max (\max n1 \ n2) \ n3 \rightarrow ws1; \\ G \vdash s2 \text{ } -t2 \succ -\max (\max n1 \ n2) \ n3 \rightarrow ws2; \\ G \vdash s3 \text{ } -t3 \succ -\max (\max n1 \ n2) \ n3 \rightarrow ws3 \rrbracket \implies P \\ \rrbracket \implies P$

$\langle \text{proof} \rangle$

**lemma** *le-max3I1*:  $(n2::nat) \leq \max n1 \ (\max n2 \ n3)$

$\langle \text{proof} \rangle$

**lemma** *le-max3I2*:  $(n3::nat) \leq \max n1 \ (\max n2 \ n3)$

$\langle \text{proof} \rangle$

$\langle ML \rangle$

**eval implies evaln**

**lemma** *eval-evaln*:

**assumes** *eval*:  $G \vdash s0 \text{ } -t \succ \rightarrow (v, s1)$   
**shows**  $\exists n. G \vdash s0 \text{ } -t \succ -n \rightarrow (v, s1)$

$\langle \text{proof} \rangle$

**end**

# Chapter 21

## Trans

**theory** *Trans* **imports** *Evaln* **begin**

**constdefs** *groundVar*:: *var*  $\Rightarrow$  *bool*  
*groundVar* *v*  $\equiv$  (case *v* of  
  *LVar* *ln*  $\Rightarrow$  *True*  
  | {*accC*,*statDeclC*,*stat*}*e*..*fn*  $\Rightarrow$   $\exists$  *a*. *e*=*Lit* *a*  
  | *e1*..*e2*  $\Rightarrow$   $\exists$  *a* *i*. *e1* = *Lit* *a*  $\wedge$  *e2* = *Lit* *i*  
  | *InsInitV* *c* *v*  $\Rightarrow$  *False*)

**lemma** *groundVar-cases* [consumes 1, case-names *LVar FVar AVar*]:

**assumes** *ground*: *groundVar* *v* **and**  
    *LVar*:  $\bigwedge$  *ln*.  $\llbracket v = \text{LVar } ln \rrbracket \Longrightarrow P$  **and**  
    *FVar*:  $\bigwedge$  *accC statDeclC stat a fn*.  
       $\llbracket v = \{accC, statDeclC, stat\} (Lit\ a) .. fn \rrbracket \Longrightarrow P$  **and**  
    *AVar*:  $\bigwedge$  *a i*.  $\llbracket v = (Lit\ a) .. [Lit\ i] \rrbracket \Longrightarrow P$

**shows** *P*

$\langle$ proof $\rangle$

**constdefs** *groundExprs*:: *expr list*  $\Rightarrow$  *bool*  
*groundExprs* *es*  $\equiv$  *list-all* ( $\lambda$  *e*.  $\exists$  *v*. *e*=*Lit* *v*) *es*

**consts** *the-val*:: *expr*  $\Rightarrow$  *val*

**primrec**

*the-val* (*Lit* *v*) = *v*

**consts** *the-var*:: *prog*  $\Rightarrow$  *state*  $\Rightarrow$  *var*  $\Rightarrow$  (*vvar*  $\times$  *state*)

**primrec**

*the-var* *G* *s* (*LVar* *ln*) = (*lvar* *ln* (*store* *s*), *s*)

*the-var-FVar-def*:

*the-var* *G* *s* ({*accC*,*statDeclC*,*stat*}*a*..*fn*) = *fvar* *statDeclC* *stat* *fn* (*the-val* *a*) *s*

*the-var-AVar-def*:

*the-var* *G* *s* (*a*..*i*) = *avar* *G* (*the-val* *i*) (*the-val* *a*) *s*

**lemma** *the-var-FVar-simp*[*simp*]:

*the-var* *G* *s* ({*accC*,*statDeclC*,*stat*}(*Lit* *a*)..*fn*) = *fvar* *statDeclC* *stat* *fn* *a* *s*

$\langle$ proof $\rangle$

**declare** *the-var-FVar-def* [*simp del*]

**lemma** *the-var-AVar-simp*:

*the-var*  $G\ s\ ((Lit\ a).[Lit\ i]) = avar\ G\ i\ a\ s$   
 $\langle proof \rangle$

**declare** *the-var-AVar-def* [*simp del*]

**consts**

*step* ::  $prog \Rightarrow ((term \times state) \times (term \times state))\ set$

**syntax** (*symbols*)

*step* ::  $[prog, term \times state, term \times state] \Rightarrow bool\ (\vdash \mapsto 1\ -[61,82,82]\ 81)$

*stepn*::  $[prog, term \times state, nat, term \times state] \Rightarrow bool$   
 $(\vdash \mapsto -[61,82,82]\ 81)$

*step\**::  $[prog, term \times state, term \times state] \Rightarrow bool\ (\vdash \mapsto * -[61,82,82]\ 81)$

*Ref* ::  $loc \Rightarrow expr$

*SKIP* ::  $expr$

**translations**

$G \vdash p \mapsto 1\ p' == (p, p') \in step\ G$

$G \vdash p \mapsto n\ p' == (p, p') \in (step\ G)^{\wedge n}$

$G \vdash p \mapsto * p' == (p, p') \in (step\ G)^*$

$Ref\ a == Lit\ (Addr\ a)$

$SKIP == Lit\ Unit$

**inductive** *step G intros*

*Abrupt*:  $\llbracket \forall v. t \neq \langle Lit\ v \rangle;$   
 $\forall t. t \neq \langle l \cdot Skip \rangle;$   
 $\forall C\ vn\ c. t \neq \langle Try\ Skip\ Catch(C\ vn)\ c \rangle;$   
 $\forall x\ c. t \neq \langle Skip\ Finally\ c \rangle \wedge xc \neq Xcpt\ x;$   
 $\forall a\ c. t \neq \langle FinA\ a\ c \rangle \rrbracket$   
 $\implies$   
 $G \vdash (t, Some\ xc, s) \mapsto 1\ (\langle Lit\ arbitrary \rangle, Some\ xc, s)$

*InsInitE*:  $\llbracket G \vdash (\langle c \rangle, Norm\ s) \mapsto 1\ (\langle c' \rangle, s') \rrbracket$   
 $\implies$   
 $G \vdash (\langle InsInitE\ c\ e \rangle, Norm\ s) \mapsto 1\ (\langle InsInitE\ c'\ e \rangle, s')$

*NewC*:  $G \vdash (\langle NewC\ C \rangle, Norm\ s) \mapsto 1\ (\langle InsInitE\ (Init\ C)\ (NewC\ C) \rangle, Norm\ s)$

*NewCInitd*:  $\llbracket G \vdash Norm\ s -halloc\ (CInst\ C) \succ a \rightarrow s' \rrbracket$

$\implies$   
 $G \vdash (\langle InsInitE\ Skip\ (NewC\ C) \rangle, Norm\ s) \mapsto 1\ (\langle Ref\ a \rangle, s')$

*NewA*:

$G \vdash (\langle New\ T[e] \rangle, Norm\ s) \mapsto 1\ (\langle InsInitE\ (init-comp-ty\ T)\ (New\ T[e]) \rangle, Norm\ s)$

*InsInitNewAIdx*:

$\llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1\ (\langle e' \rangle, s') \rrbracket$

$\implies$

$G \vdash (\langle InsInitE\ Skip\ (New\ T[e]) \rangle, Norm\ s) \mapsto 1\ (\langle InsInitE\ Skip\ (New\ T[e']) \rangle, s')$

*InsInitNewA*:

$\llbracket G \vdash abupd\ (check-neg\ i)\ (Norm\ s) -halloc\ (Arr\ T\ (the-Intg\ i)) \succ a \rightarrow s' \rrbracket$

$$\begin{aligned} &\Rightarrow \\ &G \vdash (\langle \text{InsInitE Skip (New T[Lit i])} \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Ref } a \rangle, s') \end{aligned}$$

*CastE:*

$$\begin{aligned} &\llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 (\langle e' \rangle, s') \rrbracket \\ &\Rightarrow \\ &G \vdash (\langle \text{Cast } T \ e \rangle, \text{None}, s) \mapsto 1 (\langle \text{Cast } T \ e' \rangle, s') \end{aligned}$$

*Cast:*

$$\begin{aligned} &\llbracket s' = \text{abupd (raise-if } (\neg G, s \vdash v \text{ fits } T) \text{ ClassCast) (Norm } s) \rrbracket \\ &\Rightarrow \\ &G \vdash (\langle \text{Cast } T \ (\text{Lit } v) \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Lit } v \rangle, s') \end{aligned}$$

$$\text{InstE: } \llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 (\langle e'::\text{expr} \rangle, s') \rrbracket$$

$$\begin{aligned} &\Rightarrow \\ &G \vdash (\langle e \text{ InstOf } T \rangle, \text{Norm } s) \mapsto 1 (\langle e' \rangle, s') \end{aligned}$$

$$\text{Inst: } \llbracket b = (v \neq \text{Null} \wedge G, s \vdash v \text{ fits RefT } T) \rrbracket$$

$$\begin{aligned} &\Rightarrow \\ &G \vdash (\langle (\text{Lit } v) \text{ InstOf } T \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Lit } (\text{Bool } b) \rangle, s') \end{aligned}$$

$$\text{UnOpE: } \llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 (\langle e' \rangle, s') \rrbracket$$

$$\begin{aligned} &\Rightarrow \\ &G \vdash (\langle \text{UnOp unop } e \rangle, \text{Norm } s) \mapsto 1 (\langle \text{UnOp unop } e' \rangle, s') \end{aligned}$$

$$\text{UnOp: } G \vdash (\langle \text{UnOp unop } (\text{Lit } v) \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Lit } (\text{eval-unop unop } v) \rangle, \text{Norm } s)$$

$$\text{BinOpE1: } \llbracket G \vdash (\langle e1 \rangle, \text{Norm } s) \mapsto 1 (\langle e1' \rangle, s') \rrbracket$$

$$\begin{aligned} &\Rightarrow \\ &G \vdash (\langle \text{BinOp binop } e1 \ e2 \rangle, \text{Norm } s) \mapsto 1 (\langle \text{BinOp binop } e1' \ e2 \rangle, s') \end{aligned}$$

$$\text{BinOpE2: } \llbracket \text{need-second-arg binop } v1; G \vdash (\langle e2 \rangle, \text{Norm } s) \mapsto 1 (\langle e2' \rangle, s') \rrbracket$$

$$\begin{aligned} &\Rightarrow \\ &G \vdash (\langle \text{BinOp binop } (\text{Lit } v1) \ e2 \rangle, \text{Norm } s) \\ &\mapsto 1 (\langle \text{BinOp binop } (\text{Lit } v1) \ e2' \rangle, s') \end{aligned}$$

$$\text{BinOpTerm: } \llbracket \neg \text{need-second-arg binop } v1 \rrbracket$$

$$\begin{aligned} &\Rightarrow \\ &G \vdash (\langle \text{BinOp binop } (\text{Lit } v1) \ e2 \rangle, \text{Norm } s) \\ &\mapsto 1 (\langle \text{Lit } v1 \rangle, \text{Norm } s) \end{aligned}$$

$$\begin{aligned} \text{BinOp: } &G \vdash (\langle \text{BinOp binop } (\text{Lit } v1) \ (\text{Lit } v2) \rangle, \text{Norm } s) \\ &\mapsto 1 (\langle \text{Lit } (\text{eval-binop binop } v1 \ v2) \rangle, \text{Norm } s) \end{aligned}$$

$$\text{Super: } G \vdash (\langle \text{Super} \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Lit } (\text{val-this } s) \rangle, \text{Norm } s)$$

$$\text{AccVA: } \llbracket G \vdash (\langle va \rangle, \text{Norm } s) \mapsto 1 (\langle va' \rangle, s') \rrbracket$$

$$\begin{aligned} &\Rightarrow \\ &G \vdash (\langle \text{Acc } va \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Acc } va' \rangle, s') \end{aligned}$$

$$\text{Acc: } \llbracket \text{groundVar } va; ((v, vf), s') = \text{the-var } G \ (\text{Norm } s) \ va \rrbracket$$

$$\begin{aligned} &\Rightarrow \\ &G \vdash (\langle \text{Acc } va \rangle, \text{Norm } s) \mapsto 1 (\langle \text{Lit } v \rangle, s') \end{aligned}$$

$$\text{AssVA: } \llbracket G \vdash (\langle va \rangle, \text{Norm } s) \mapsto 1 (\langle va' \rangle, s') \rrbracket$$

$$\begin{aligned} &\Rightarrow \\ &G \vdash (\langle va := e \rangle, \text{Norm } s) \mapsto 1 (\langle va' := e \rangle, s') \end{aligned}$$

$$\text{AssE: } \llbracket \text{groundVar } va; G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 (\langle e' \rangle, s') \rrbracket$$

$$\begin{aligned}
& \Rightarrow \\
& G \vdash (\langle va := e \rangle, Norm\ s) \mapsto 1\ (\langle va := e' \rangle, s') \\
Ass: \quad & \llbracket groundVar\ va; ((w, f), s') = the-var\ G\ (Norm\ s)\ va \rrbracket \\
& \Rightarrow \\
& G \vdash (\langle va := (Lit\ v) \rangle, Norm\ s) \mapsto 1\ (\langle Lit\ v \rangle, assign\ f\ v\ s') \\
\\
CondC: \quad & \llbracket G \vdash (\langle e0 \rangle, Norm\ s) \mapsto 1\ (\langle e0' \rangle, s') \rrbracket \\
& \Rightarrow \\
& G \vdash (\langle e0? e1:e2 \rangle, Norm\ s) \mapsto 1\ (\langle e0'? e1:e2 \rangle, s') \\
Cond: \quad & G \vdash (\langle Lit\ b? e1:e2 \rangle, Norm\ s) \mapsto 1\ (\langle if\ the-Bool\ b\ then\ e1\ else\ e2 \rangle, Norm\ s) \\
\\
\\
CallTarget: \quad & \llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1\ (\langle e' \rangle, s') \rrbracket \\
& \Rightarrow \\
& G \vdash (\langle \{accC, statT, mode\} e.mn(\{pTs\}args) \rangle, Norm\ s) \\
& \mapsto 1\ (\langle \{accC, statT, mode\} e'.mn(\{pTs\}args) \rangle, s') \\
CallArgs: \quad & \llbracket G \vdash (\langle args \rangle, Norm\ s) \mapsto 1\ (\langle args' \rangle, s') \rrbracket \\
& \Rightarrow \\
& G \vdash (\langle \{accC, statT, mode\} Lit\ a.mn(\{pTs\}args) \rangle, Norm\ s) \\
& \mapsto 1\ (\langle \{accC, statT, mode\} Lit\ a.mn(\{pTs\}args') \rangle, s') \\
Call: \quad & \llbracket groundExprs\ args; vs = map\ the-val\ args; \\
& D = invocation-declclass\ G\ mode\ s\ a\ statT\ (\llbracket name=mn, parTs=pTs \rrbracket); \\
& s' = init-lvars\ G\ D\ (\llbracket name=mn, parTs=pTs \rrbracket)\ mode\ a'\ vs\ (Norm\ s) \rrbracket \\
& \Rightarrow \\
& G \vdash (\langle \{accC, statT, mode\} Lit\ a.mn(\{pTs\}args) \rangle, Norm\ s) \\
& \mapsto 1\ (\langle Callee\ (locals\ s)\ (Methd\ D\ (\llbracket name=mn, parTs=pTs \rrbracket)) \rangle, s') \\
\\
Callee: \quad & \llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1\ (\langle e':expr \rangle, s') \rrbracket \\
& \Rightarrow \\
& G \vdash (\langle Callee\ lcls-caller\ e \rangle, Norm\ s) \mapsto 1\ (\langle e' \rangle, s') \\
\\
CalleeRet: \quad & G \vdash (\langle Callee\ lcls-caller\ (Lit\ v) \rangle, Norm\ s) \\
& \mapsto 1\ (\langle Lit\ v \rangle, (set-lvars\ lcls-caller\ (Norm\ s))) \\
\\
Methd: \quad & G \vdash (\langle Methd\ D\ sig \rangle, Norm\ s) \mapsto 1\ (\langle body\ G\ D\ sig \rangle, Norm\ s) \\
\\
Body: \quad & G \vdash (\langle Body\ D\ c \rangle, Norm\ s) \mapsto 1\ (\langle InsInitE\ (Init\ D)\ (Body\ D\ c) \rangle, Norm\ s) \\
\\
InsInitBody: \\
& \llbracket G \vdash (\langle c \rangle, Norm\ s) \mapsto 1\ (\langle c' \rangle, s') \rrbracket \\
& \Rightarrow \\
& G \vdash (\langle InsInitE\ Skip\ (Body\ D\ c) \rangle, Norm\ s) \mapsto 1\ (\langle InsInitE\ Skip\ (Body\ D\ c') \rangle, s') \\
InsInitBodyRet: \\
& G \vdash (\langle InsInitE\ Skip\ (Body\ D\ Skip) \rangle, Norm\ s) \\
& \mapsto 1\ (\langle Lit\ (the\ ((locals\ s)\ Result)) \rangle, abupd\ (absorb\ Ret)\ (Norm\ s)) \\
\\
\\
FVar: \quad & \llbracket \neg\ initied\ statDeclC\ (globs\ s) \rrbracket \\
& \Rightarrow \\
& G \vdash (\langle \{accC, statDeclC, stat\} e..fn \rangle, Norm\ s) \\
& \mapsto 1\ (\langle InsInitV\ (Init\ statDeclC)\ (\{accC, statDeclC, stat\} e..fn) \rangle, Norm\ s) \\
InsInitFVarE: \\
& \llbracket G \vdash (\langle e \rangle, Norm\ s) \mapsto 1\ (\langle e' \rangle, s') \rrbracket \\
& \Rightarrow \\
& G \vdash (\langle InsInitV\ Skip\ (\{accC, statDeclC, stat\} e..fn) \rangle, Norm\ s) \\
& \mapsto 1\ (\langle InsInitV\ Skip\ (\{accC, statDeclC, stat\} e'..fn) \rangle, s') \\
InsInitFVar: \\
& G \vdash (\langle InsInitV\ Skip\ (\{accC, statDeclC, stat\} Lit\ a..fn) \rangle, Norm\ s)
\end{aligned}$$

$$\mapsto 1 \ (\langle \{accC, statDeclC, stat\} Lit \ a..fn \rangle, Norm \ s)$$

— Notice, that we do not have literal values for *vars*. The rules for accessing variables (*Acc*) and assigning to variables (*Ass*), test this with the predicate *groundVar*. After initialisation is done and the *FVar* is evaluated, we can't just throw away the *InsInitFVar* term and return a literal value, as in the cases of *New* or *NewC*. Instead we just return the evaluated *FVar* and test for initialisation in the rule *FVar*.

$$\begin{aligned} AVarE1: \llbracket G \vdash (\langle e1 \rangle, Norm \ s) \mapsto 1 \ (\langle e1 \rangle', s') \rrbracket \\ \implies \\ G \vdash (\langle e1.[e2] \rangle, Norm \ s) \mapsto 1 \ (\langle e1'.[e2] \rangle, s') \end{aligned}$$

$$\begin{aligned} AVarE2: G \vdash (\langle e2 \rangle, Norm \ s) \mapsto 1 \ (\langle e2 \rangle', s') \\ \implies \\ G \vdash (\langle Lit \ a.[e2] \rangle, Norm \ s) \mapsto 1 \ (\langle Lit \ a.[e2]' \rangle, s') \end{aligned}$$

— *Nil* is fully evaluated

$$\begin{aligned} ConsHd: \llbracket G \vdash (\langle e::expr \rangle, Norm \ s) \mapsto 1 \ (\langle e'::expr \rangle, s') \rrbracket \\ \implies \\ G \vdash (\langle e \# es \rangle, Norm \ s) \mapsto 1 \ (\langle e' \# es \rangle, s') \end{aligned}$$

$$\begin{aligned} ConsTl: \llbracket G \vdash (\langle es \rangle, Norm \ s) \mapsto 1 \ (\langle es \rangle', s') \rrbracket \\ \implies \\ G \vdash (\langle (Lit \ v) \# es \rangle, Norm \ s) \mapsto 1 \ (\langle (Lit \ v) \# es' \rangle, s') \end{aligned}$$

$$Skip: G \vdash (\langle Skip \rangle, Norm \ s) \mapsto 1 \ (\langle SKIP \rangle, Norm \ s)$$

$$\begin{aligned} ExprE: \llbracket G \vdash (\langle e \rangle, Norm \ s) \mapsto 1 \ (\langle e \rangle', s') \rrbracket \\ \implies \\ G \vdash (\langle Expr \ e \rangle, Norm \ s) \mapsto 1 \ (\langle Expr \ e' \rangle, s') \\ Expr: G \vdash (\langle Expr \ (Lit \ v) \rangle, Norm \ s) \mapsto 1 \ (\langle Skip \rangle, Norm \ s) \end{aligned}$$

$$\begin{aligned} LabC: \llbracket G \vdash (\langle c \rangle, Norm \ s) \mapsto 1 \ (\langle c \rangle', s') \rrbracket \\ \implies \\ G \vdash (\langle l \bullet c \rangle, Norm \ s) \mapsto 1 \ (\langle l \bullet c' \rangle, s') \\ Lab: G \vdash (\langle l \bullet Skip \rangle, s) \mapsto 1 \ (\langle Skip \rangle, abupd \ (absorb \ l) \ s) \end{aligned}$$

$$\begin{aligned} CompC1: \llbracket G \vdash (\langle c1 \rangle, Norm \ s) \mapsto 1 \ (\langle c1 \rangle', s') \rrbracket \\ \implies \\ G \vdash (\langle c1;; c2 \rangle, Norm \ s) \mapsto 1 \ (\langle c1';; c2 \rangle, s') \end{aligned}$$

$$Comp: G \vdash (\langle Skip;; c2 \rangle, Norm \ s) \mapsto 1 \ (\langle c2 \rangle, Norm \ s)$$

$$\begin{aligned} IfE: \llbracket G \vdash (\langle e \rangle, Norm \ s) \mapsto 1 \ (\langle e \rangle', s') \rrbracket \\ \implies \\ G \vdash (\langle If \ (e) \ s1 \ Else \ s2 \rangle, Norm \ s) \mapsto 1 \ (\langle If \ (e') \ s1 \ Else \ s2' \rangle, s') \\ If: G \vdash (\langle If \ (Lit \ v) \ s1 \ Else \ s2 \rangle, Norm \ s) \\ \mapsto 1 \ (\langle if \ the-Bool \ v \ then \ s1 \ else \ s2' \rangle, Norm \ s) \end{aligned}$$

*Loop*:  $G \vdash (\langle l \cdot \text{While}(e) \ c \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{If}(e) \ (\text{Cont } l \cdot c;; \ l \cdot \text{While}(e) \ c) \ \text{Else } \text{Skip} \rangle, \text{Norm } s)$

*Jmp*:  $G \vdash (\langle \text{Jmp } j \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Skip} \rangle, (\text{Some } (\text{Jump } j), s))$

*ThrowE*:  $\llbracket G \vdash (\langle e \rangle, \text{Norm } s) \mapsto 1 \ (\langle e' \rangle, s') \rrbracket$

$\implies$

$G \vdash (\langle \text{Throw } e \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Throw } e' \rangle, s')$

*Throw*:  $G \vdash (\langle \text{Throw } (\text{Lit } a) \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Skip} \rangle, \text{abupd } (\text{throw } a) \ (\text{Norm } s))$

*TryC1*:  $\llbracket G \vdash (\langle c1 \rangle, \text{Norm } s) \mapsto 1 \ (\langle c1' \rangle, s') \rrbracket$

$\implies$

$G \vdash (\langle \text{Try } c1 \ \text{Catch}(C \ vn) \ c2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Try } c1' \ \text{Catch}(C \ vn) \ c2 \rangle, s')$

*Try*:  $\llbracket G \vdash s \rightarrow s' \rrbracket$

$\implies$

$G \vdash (\langle \text{Try } \text{Skip} \ \text{Catch}(C \ vn) \ c2 \rangle, s)$

$\mapsto 1 \ (\text{if } G, s \vdash \text{catch } C \text{ then } (\langle c2 \rangle, \text{new-xcpt-var } vn \ s') \text{ else } (\langle \text{Skip} \rangle, s'))$

*FinC1*:  $\llbracket G \vdash (\langle c1 \rangle, \text{Norm } s) \mapsto 1 \ (\langle c1' \rangle, s') \rrbracket$

$\implies$

$G \vdash (\langle c1 \ \text{Finally } c2 \rangle, \text{Norm } s) \mapsto 1 \ (\langle c1' \ \text{Finally } c2 \rangle, s')$

*Fin*:  $G \vdash (\langle \text{Skip } \text{Finally } c2 \rangle, (a, s)) \mapsto 1 \ (\langle \text{FinA } a \ c2 \rangle, \text{Norm } s)$

*FinAC*:  $\llbracket G \vdash (\langle c \rangle, s) \mapsto 1 \ (\langle c' \rangle, s') \rrbracket$

$\implies$

$G \vdash (\langle \text{FinA } a \ c \rangle, s) \mapsto 1 \ (\langle \text{FinA } a \ c' \rangle, s')$

*FinA*:  $G \vdash (\langle \text{FinA } a \ \text{Skip} \rangle, s) \mapsto 1 \ (\langle \text{Skip} \rangle, \text{abupd } (\text{abrupt-if } (a \neq \text{None}) \ a) \ s)$

*Init1*:  $\llbracket \text{inited } C \ (\text{globs } s) \rrbracket$

$\implies$

$G \vdash (\langle \text{Init } C \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{Skip} \rangle, \text{Norm } s)$

*Init*:  $\llbracket \text{the } (\text{class } G \ C) = c; \neg \text{inited } C \ (\text{globs } s) \rrbracket$

$\implies$

$G \vdash (\langle \text{Init } C \rangle, \text{Norm } s)$

$\mapsto 1 \ (\langle (\text{if } C = \text{Object} \text{ then } \text{Skip} \text{ else } (\text{Init } (\text{super } c))) \;; \ \text{Expr } (\text{Callee } (\text{locals } s) \ (\text{InsInitE } (\text{init } c) \ \text{SKIP}))) \rangle, \text{Norm } (\text{init-class-obj } G \ C \ s))$

— *InsInitE* is just used as trick to embed the statement *init c* into an expression

*InsInitESKIP*:

$G \vdash (\langle \text{InsInitE } \text{Skip } \text{SKIP} \rangle, \text{Norm } s) \mapsto 1 \ (\langle \text{SKIP} \rangle, \text{Norm } s)$

**lemma** *rtranc1-imp-rel-pow*:  $p \in R^* \implies \exists n. p \in R^n$   
 <proof>

**end**



## Chapter 22

### AxSem

## 50 Axiomatic semantics of Java expressions and statements (see also Eval.thy)

**theory** *AxSem* **imports** *Evaln TypeSafe* **begin**

design issues:

- a strong version of validity for triples with premises, namely one that takes the recursive depth needed to complete execution, enables correctness proof
- auxiliary variables are handled first-class (-i Thomas Kleymann)
- expressions not flattened to elementary assignments (as usual for axiomatic semantics) but treated first-class =i explicit result value handling
- intermediate values not on triple, but on assertion level (with result entry)
- multiple results with semantical substitution mechanism not requiring a stack
- because of dynamic method binding, terms need to be dependent on state. this is also useful for conditional expressions and statements
- result values in triples exactly as in eval relation (also for xcpt states)
- validity: additional assumption of state conformance and well-typedness, which is required for soundness and thus rule hazard required of completeness

restrictions:

- all triples in a derivation are of the same type (due to weak polymorphism)

**types** *res = vals* — result entry

**syntax**

*Val* :: *val*  $\Rightarrow$  *res*

*Var* :: *var*  $\Rightarrow$  *res*

*Vals* :: *val list*  $\Rightarrow$  *res*

**translations**

*Val* *x*  $\Rightarrow$  (*In1* *x*)

*Var* *x*  $\Rightarrow$  (*In2* *x*)

*Vals* *x*  $\Rightarrow$  (*In3* *x*)

**syntax**

*Val*- :: [*pttrn*]  $\Rightarrow$  *pttrn* (*Val*:- [*951*] *950*)

*Var*- :: [*pttrn*]  $\Rightarrow$  *pttrn* (*Var*:- [*951*] *950*)

*Vals*- :: [*pttrn*]  $\Rightarrow$  *pttrn* (*Vals*:- [*951*] *950*)

**translations**

$\lambda \text{Val}:v . b == (\lambda v. b) \circ \text{the-In1}$

$\lambda \text{Var}:v . b == (\lambda v. b) \circ \text{the-In2}$

$\lambda \text{Vals}:v. b == (\lambda v. b) \circ \text{the-In3}$

— relation on result values, state and auxiliary variables

**types** '*a assn* = *res*  $\Rightarrow$  *state*  $\Rightarrow$  '*a*  $\Rightarrow$  *bool*

**translations**

*res*  $\leq$  (*type*) *AxSem.res*

*a assn*  $\leq$  (*type*) *vals*  $\Rightarrow$  *state*  $\Rightarrow$  *a*  $\Rightarrow$  *bool*

**constdefs**

*assn-imp* :: '*a assn*  $\Rightarrow$  '*a assn*  $\Rightarrow$  *bool* (infixr  $\Rightarrow$  25)

$P \Rightarrow Q \equiv \forall Y s Z. P Y s Z \longrightarrow Q Y s Z$

**lemma** *assn-imp-def2* [iff]:  $(P \Rightarrow Q) = (\forall Y\ s\ Z. P\ Y\ s\ Z \longrightarrow Q\ Y\ s\ Z)$   
 ⟨proof⟩

## assertion transformers

### 51 peek-and

#### constdefs

*peek-and*  $:: 'a\ assn \Rightarrow (state \Rightarrow bool) \Rightarrow 'a\ assn$  (**infixl**  $\wedge$ , 13)  
 $P \wedge. p \equiv \lambda Y\ s\ Z. P\ Y\ s\ Z \wedge p\ s$

**lemma** *peek-and-def2* [simp]:  $peek\ and\ P\ p\ Y\ s = (\lambda Z. (P\ Y\ s\ Z \wedge p\ s))$   
 ⟨proof⟩

**lemma** *peek-and-Not* [simp]:  $(P \wedge. (\lambda s. \neg f\ s)) = (P \wedge. Not \circ f)$   
 ⟨proof⟩

**lemma** *peek-and-and* [simp]:  $peek\ and\ (peek\ and\ P\ p)\ p = peek\ and\ P\ p$   
 ⟨proof⟩

**lemma** *peek-and-commut*:  $(P \wedge. p \wedge. q) = (P \wedge. q \wedge. p)$   
 ⟨proof⟩

#### syntax

*Normal*  $:: 'a\ assn \Rightarrow 'a\ assn$

#### translations

*Normal*  $P == P \wedge. normal$

**lemma** *peek-and-Normal* [simp]:  $peek\ and\ (Normal\ P)\ p = Normal\ (peek\ and\ P\ p)$   
 ⟨proof⟩

### 52 assn-supd

#### constdefs

*assn-supd*  $:: 'a\ assn \Rightarrow (state \Rightarrow state) \Rightarrow 'a\ assn$  (**infixl**  $;$ , 13)  
 $P ;. f \equiv \lambda Y\ s'\ Z. \exists s. P\ Y\ s\ Z \wedge s' = f\ s$

**lemma** *assn-supd-def2* [simp]:  $assn\ supd\ P\ f\ Y\ s'\ Z = (\exists s. P\ Y\ s\ Z \wedge s' = f\ s)$   
 ⟨proof⟩

### 53 supd-assn

#### constdefs

*supd-assn*  $:: (state \Rightarrow state) \Rightarrow 'a\ assn \Rightarrow 'a\ assn$  (**infixr**  $;$ , 13)  
 $f ;. P \equiv \lambda Y\ s. P\ Y\ (f\ s)$

**lemma** *supd-assn-def2* [simp]:  $(f ;. P)\ Y\ s = P\ Y\ (f\ s)$   
 ⟨proof⟩

**lemma** *supd-assn-supdD* [elim]:  $((f ;. Q) ;. f)\ Y\ s\ Z \Longrightarrow Q\ Y\ s\ Z$

$\langle proof \rangle$

**lemma** *supd-assn-supdI* [elim]:  $Q \ Y \ s \ Z \implies (f \ .; (Q \ ;. f)) \ Y \ s \ Z$

$\langle proof \rangle$

## 54 subst-res

**constdefs**

*subst-res*  $:: 'a \ assn \Rightarrow res \Rightarrow 'a \ assn$  ( $\leftarrow-$  [60,61] 60)  
 $P \leftarrow w \equiv \lambda Y. P \ w$

**lemma** *subst-res-def2* [simp]:  $(P \leftarrow w) \ Y = P \ w$

$\langle proof \rangle$

**lemma** *subst-subst-res* [simp]:  $P \leftarrow w \leftarrow v = P \leftarrow w$

$\langle proof \rangle$

**lemma** *peek-and-subst-res* [simp]:  $(P \ \wedge. \ p) \leftarrow w = (P \leftarrow w \ \wedge. \ p)$

$\langle proof \rangle$

## 55 subst-Bool

**constdefs**

*subst-Bool*  $:: 'a \ assn \Rightarrow bool \Rightarrow 'a \ assn$  ( $\leftarrow=$  [60,61] 60)  
 $P \leftarrow= b \equiv \lambda Y \ s \ Z. \exists v. P \ (Val \ v) \ s \ Z \ \wedge \ (normal \ s \longrightarrow the-Bool \ v=b)$

**lemma** *subst-Bool-def2* [simp]:

$(P \leftarrow= b) \ Y \ s \ Z = (\exists v. P \ (Val \ v) \ s \ Z \ \wedge \ (normal \ s \longrightarrow the-Bool \ v=b))$

$\langle proof \rangle$

**lemma** *subst-Bool-the-BoolI*:  $P \ (Val \ b) \ s \ Z \implies (P \leftarrow= the-Bool \ b) \ Y \ s \ Z$

$\langle proof \rangle$

## 56 peek-res

**constdefs**

*peek-res*  $:: (res \Rightarrow 'a \ assn) \Rightarrow 'a \ assn$   
 $peek-res \ Pf \equiv \lambda Y. Pf \ Y \ Y$

**syntax**

@*peek-res*  $:: pptrn \Rightarrow 'a \ assn \Rightarrow 'a \ assn$  ( $\lambda \cdot \cdot$  - [0,3] 3)

**translations**

$\lambda w \cdot P \ == \ peek-res \ (\lambda w. P)$

**lemma** *peek-res-def2* [simp]:  $peek-res \ P \ Y = P \ Y \ Y$

$\langle proof \rangle$

**lemma** *peek-res-subst-res* [simp]:  $peek-res \ P \leftarrow w = P \ w \leftarrow w$

$\langle proof \rangle$

**lemma** *peek-subst-res-allI*:

$(\bigwedge a. T\ a\ (P\ (f\ a) \leftarrow f\ a)) \implies \forall a. T\ a\ (peek\text{-}res\ P \leftarrow f\ a)$   
 $\langle proof \rangle$

## 57 ign-res

**constdefs**

$ign\text{-}res \quad :: \quad 'a\ assn \Rightarrow 'a\ assn \quad (-\downarrow [1000]\ 1000)$   
 $P\downarrow \quad \equiv \lambda Y\ s\ Z. \exists Y. P\ Y\ s\ Z$

**lemma** *ign-res-def2* [simp]:  $P\downarrow\ Y\ s\ Z = (\exists Y. P\ Y\ s\ Z)$   
 $\langle proof \rangle$

**lemma** *ign-ign-res* [simp]:  $P\downarrow\downarrow = P\downarrow$   
 $\langle proof \rangle$

**lemma** *ign-subst-res* [simp]:  $P\downarrow \leftarrow w = P\downarrow$   
 $\langle proof \rangle$

**lemma** *peek-and-ign-res* [simp]:  $(P \wedge. p)\downarrow = (P\downarrow \wedge. p)$   
 $\langle proof \rangle$

## 58 peek-st

**constdefs**

$peek\text{-}st \quad :: \quad (st \Rightarrow 'a\ assn) \Rightarrow 'a\ assn$   
 $peek\text{-}st\ P \equiv \lambda Y\ s. P\ (store\ s)\ Y\ s$

**syntax**

$@peek\text{-}st \quad :: \quad pttrn \Rightarrow 'a\ assn \Rightarrow 'a\ assn \quad (\lambda\text{-}.. - [0,3]\ 3)$

**translations**

$\lambda s.. P \quad == \quad peek\text{-}st\ (\lambda s. P)$

**lemma** *peek-st-def2* [simp]:  $(\lambda s.. Pf\ s)\ Y\ s = Pf\ (store\ s)\ Y\ s$   
 $\langle proof \rangle$

**lemma** *peek-st-triv* [simp]:  $(\lambda s.. P) = P$   
 $\langle proof \rangle$

**lemma** *peek-st-st* [simp]:  $(\lambda s.. \lambda s'. P\ s\ s') = (\lambda s.. P\ s\ s)$   
 $\langle proof \rangle$

**lemma** *peek-st-split* [simp]:  $(\lambda s.. \lambda Y\ s'. P\ s\ Y\ s') = (\lambda Y\ s. P\ (store\ s)\ Y\ s)$   
 $\langle proof \rangle$

**lemma** *peek-st-subst-res* [simp]:  $(\lambda s.. P\ s) \leftarrow w = (\lambda s.. P\ s \leftarrow w)$   
 $\langle proof \rangle$

**lemma** *peek-st-Normal* [simp]:  $(\lambda s..(Normal (P s))) = Normal (\lambda s.. P s)$   
 ⟨proof⟩

## 59 ign-res-eq

**constdefs**

$ign-res-eq :: 'a\ assn \Rightarrow res \Rightarrow 'a\ assn$  ( $\downarrow = -$  [60,61] 60)  
 $P \downarrow = w \equiv \lambda Y.. P \downarrow \wedge. (\lambda s.. Y = w)$

**lemma** *ign-res-eq-def2* [simp]:  $(P \downarrow = w) Y s Z = ((\exists Y.. P Y s Z) \wedge Y = w)$   
 ⟨proof⟩

**lemma** *ign-ign-res-eq* [simp]:  $(P \downarrow = w) \downarrow = P \downarrow$   
 ⟨proof⟩

**lemma** *ign-res-eq-subst-res*:  $P \downarrow = w \leftarrow w = P \downarrow$   
 ⟨proof⟩

**lemma** *subst-Bool-ign-res-eq*:  $((P \leftarrow = b) \downarrow = x) Y s Z = ((P \leftarrow = b) Y s Z \wedge Y = x)$   
 ⟨proof⟩

## 60 RefVar

**constdefs**

$RefVar :: (state \Rightarrow vvar \times state) \Rightarrow 'a\ assn \Rightarrow 'a\ assn$  (**infixr**  $..$ ; 13)  
 $vf ..; P \equiv \lambda Y s.. let (v, s') = vf s in P (Var v) s'$

**lemma** *RefVar-def2* [simp]:  $(vf ..; P) Y s =$   
 $P (Var (fst (vf s))) (snd (vf s))$   
 ⟨proof⟩

## 61 allocation

**constdefs**

$Alloc :: prog \Rightarrow obj-tag \Rightarrow 'a\ assn \Rightarrow 'a\ assn$   
 $Alloc G otag P \equiv \lambda Y s Z..$   
 $\forall s' a.. G \vdash s -halloc\ otag \succ a \rightarrow s' \longrightarrow P (Val (Addr a)) s' Z$

$SXAlloc :: prog \Rightarrow 'a\ assn \Rightarrow 'a\ assn$   
 $SXAlloc G P \equiv \lambda Y s Z.. \forall s'.. G \vdash s -salloc \rightarrow s' \longrightarrow P Y s' Z$

**lemma** *Alloc-def2* [simp]:  $Alloc G otag P Y s Z =$   
 $(\forall s' a.. G \vdash s -halloc\ otag \succ a \rightarrow s' \longrightarrow P (Val (Addr a)) s' Z)$   
 ⟨proof⟩

**lemma** *SXAlloc-def2* [simp]:  
 $SXAlloc G P Y s Z = (\forall s'.. G \vdash s -salloc \rightarrow s' \longrightarrow P Y s' Z)$   
 ⟨proof⟩

**validity****constdefs**

$type-ok :: prog \Rightarrow term \Rightarrow state \Rightarrow bool$   
 $type-ok\ G\ t\ s \equiv$   
 $\exists L\ T\ C\ A. (normal\ s \longrightarrow (\llbracket prg=G, cls=C, lcl=L \rrbracket) \vdash t :: T \wedge$   
 $\quad (\llbracket prg=G, cls=C, lcl=L \rrbracket) \vdash dom\ (locals\ (store\ s)) \gg t \gg A) \wedge s :: \preceq(G, L)$

**datatype**  $'a\ triple = triple\ ('a\ assn)\ term\ ('a\ assn)$   
 $(\{(1-)\} / -> / \{(1-)\}) \quad [3, 65, 3] 75$

**types**  $'a\ triples = 'a\ triple\ set$

**syntax**

$var-triple :: ['a\ assn, var, 'a\ assn] \Rightarrow 'a\ triple$   
 $(\{(1-)\} / -=> / \{(1-)\}) \quad [3, 80, 3] 75$   
 $expr-triple :: ['a\ assn, expr, 'a\ assn] \Rightarrow 'a\ triple$   
 $(\{(1-)\} / --> / \{(1-)\}) \quad [3, 80, 3] 75$   
 $exprs-triple :: ['a\ assn, expr\ list, 'a\ assn] \Rightarrow 'a\ triple$   
 $(\{(1-)\} / -\#> / \{(1-)\}) \quad [3, 65, 3] 75$   
 $stmt-triple :: ['a\ assn, stmt, 'a\ assn] \Rightarrow 'a\ triple$   
 $(\{(1-)\} / \dot{-}> / \{(1-)\}) \quad [3, 65, 3] 75$

**syntax** (*xsymbols*)

$triple :: ['a\ assn, term, 'a\ assn] \Rightarrow 'a\ triple$   
 $(\{(1-)\} / -\succ / \{(1-)\}) \quad [3, 65, 3] 75$   
 $var-triple :: ['a\ assn, var, 'a\ assn] \Rightarrow 'a\ triple$   
 $(\{(1-)\} / -\succ / \{(1-)\}) \quad [3, 80, 3] 75$   
 $expr-triple :: ['a\ assn, expr, 'a\ assn] \Rightarrow 'a\ triple$   
 $(\{(1-)\} / --\succ / \{(1-)\}) \quad [3, 80, 3] 75$   
 $exprs-triple :: ['a\ assn, expr\ list, 'a\ assn] \Rightarrow 'a\ triple$   
 $(\{(1-)\} / -\dot{\succ} / \{(1-)\}) \quad [3, 65, 3] 75$

**translations**

$\{P\}\ e \dashv \succ \{Q\} == \{P\}\ In1l\ e \succ \{Q\}$   
 $\{P\}\ e = \succ \{Q\} == \{P\}\ In2\ e \succ \{Q\}$   
 $\{P\}\ e \dot{=} \succ \{Q\} == \{P\}\ In3\ e \succ \{Q\}$   
 $\{P\}\ .c. \{Q\} == \{P\}\ In1r\ c \succ \{Q\}$

**lemma** *inj-triple*:  $inj\ (\lambda(P, t, Q). \{P\}\ t \succ \{Q\})$   
 $\langle proof \rangle$

**lemma** *triple-inj-eq*:  $(\{P\}\ t \succ \{Q\} = \{P'\}\ t' \succ \{Q'\}) = (P = P' \wedge t = t' \wedge Q = Q')$   
 $\langle proof \rangle$

**constdefs**

$mtriples :: ('c \Rightarrow 'sig \Rightarrow 'a\ assn) \Rightarrow ('c \Rightarrow 'sig \Rightarrow expr) \Rightarrow$   
 $(('c \Rightarrow 'sig \Rightarrow 'a\ assn) \Rightarrow ('c \times 'sig)\ set \Rightarrow 'a\ triples)$   
 $(\{(\{(1-)\} / --\succ / \{(1-)\} \mid -)\} [3, 65, 3, 65] 75)$   
 $\{\{P\}\ tf \dashv \succ \{Q\} \mid ms\} \equiv (\lambda(C, sig). \{Normal(P\ C\ sig)\}\ tf\ C\ sig \dashv \succ \{Q\}\ C\ sig)\ 'ms$

**consts**

$triple-valid :: prog \Rightarrow nat \Rightarrow 'a\ triple \Rightarrow bool$   
 $(\dashv \vdash \dot{-} :- [61, 0, 58] 57)$

$ax\text{-}valids :: prog \Rightarrow 'b \text{ triples} \Rightarrow 'a \text{ triples} \Rightarrow bool$   
 $(-,||=- [61,58,58] 57)$   
 $ax\text{-}derivs :: prog \Rightarrow ('b \text{ triples} \times 'a \text{ triples}) \text{ set}$

### syntax

$triples\text{-}valid :: prog \Rightarrow nat \Rightarrow 'a \text{ triples} \Rightarrow bool$   
 $(-||=-:- [61,0, 58] 57)$   
 $ax\text{-}valid :: prog \Rightarrow 'b \text{ triples} \Rightarrow 'a \text{ triple} \Rightarrow bool$   
 $(-,|-=- [61,58,58] 57)$   
 $ax\text{-}Derivs :: prog \Rightarrow 'b \text{ triples} \Rightarrow 'a \text{ triples} \Rightarrow bool$   
 $(-,||=- [61,58,58] 57)$   
 $ax\text{-}Deriv :: prog \Rightarrow 'b \text{ triples} \Rightarrow 'a \text{ triple} \Rightarrow bool$   
 $(-,|-=- [61,58,58] 57)$

### syntax (*xsymbols*)

$triples\text{-}valid :: prog \Rightarrow nat \Rightarrow 'a \text{ triples} \Rightarrow bool$   
 $(-||=-:- [61,0, 58] 57)$   
 $ax\text{-}valid :: prog \Rightarrow 'b \text{ triples} \Rightarrow 'a \text{ triple} \Rightarrow bool$   
 $(-,|-=- [61,58,58] 57)$   
 $ax\text{-}Derivs :: prog \Rightarrow 'b \text{ triples} \Rightarrow 'a \text{ triples} \Rightarrow bool$   
 $(-,|-=- [61,58,58] 57)$   
 $ax\text{-}Deriv :: prog \Rightarrow 'b \text{ triples} \Rightarrow 'a \text{ triple} \Rightarrow bool$   
 $(-,|-=- [61,58,58] 57)$

**defs**  $triple\text{-}valid\text{-}def: G \models n:t \equiv \text{case } t \text{ of } \{P\} \ t \succ \{Q\} \Rightarrow$   
 $\forall Y \ s \ Z. P \ Y \ s \ Z \longrightarrow \text{type-ok } G \ t \ s \longrightarrow$   
 $(\forall Y' \ s'. G \vdash s - t \succ - n \longrightarrow (Y',s') \longrightarrow Q \ Y' \ s' \ Z)$   
**translations**  $G \models n:ts == \text{Ball } ts \ (triple\text{-}valid \ G \ n)$   
**defs**  $ax\text{-}valids\text{-}def: G, A \models ts \equiv \forall n. G \models n:A \longrightarrow G \models n:ts$   
**translations**  $G, A \models t == G, A \models \{t\}$   
 $G, A \vdash ts == (A, ts) \in ax\text{-}derivs \ G$   
 $G, A \vdash t == G, A \vdash \{t\}$

**lemma**  $triple\text{-}valid\text{-}def2: G \models n:\{P\} \ t \succ \{Q\} =$   
 $(\forall Y \ s \ Z. P \ Y \ s \ Z$   
 $\longrightarrow (\exists L. (\text{normal } s \longrightarrow (\exists C \ T \ A. (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t :: T \wedge$   
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash_{\text{dom}} (\text{locals } (\text{store } s)) \gg t \gg A)) \wedge$   
 $s :: \preceq (G, L))$   
 $\longrightarrow (\forall Y' \ s'. G \vdash s - t \succ - n \longrightarrow (Y',s') \longrightarrow Q \ Y' \ s' \ Z))$   
 $\langle \text{proof} \rangle$

**declare**  $split\text{-}paired\text{-}All \ [simp \ del] \ split\text{-}paired\text{-}Ex \ [simp \ del]$   
**declare**  $split\text{-}if \ [split \ del] \ split\text{-}if\text{-}asm \ [split \ del]$   
 $option.\text{split} \ [split \ del] \ option.\text{split}\text{-}asm \ [split \ del]$   
 $\langle ML \rangle$

### inductive $ax\text{-}derivs \ G$ intros

$empty: G, A \vdash \{\}$   
 $insert: [G, A \vdash t; G, A \vdash ts] \Longrightarrow$   
 $G, A \vdash insert \ t \ ts$   
 $asm: ts \subseteq A \Longrightarrow G, A \vdash ts$



*weaken*:  $\llbracket G, A \vdash ts'; ts \subseteq ts' \rrbracket \implies G, A \vdash ts$

*conseq*:  $\forall Y s Z . P \ Y s Z \longrightarrow (\exists P' Q'. G, A \vdash \{P'\} \ t \succ \{Q'\} \wedge (\forall Y' s' Z'. P' \ Y' s' Z' \longrightarrow Q' \ Y' s' Z')) \implies G, A \vdash \{P\} \ t \succ \{Q\}$

*hazard*:  $G, A \vdash \{P \wedge. \text{Not} \circ \text{type-ok } G \ t\} \ t \succ \{Q\}$

*Abrupt*:  $G, A \vdash \{P \leftarrow (\text{arbitrary} \beta \ t) \wedge. \text{Not} \circ \text{normal}\} \ t \succ \{P\}$

— variables

*LVar*:  $G, A \vdash \{\text{Normal } (\lambda s.. P \leftarrow \text{Var } (lvar \ vn \ s))\} \ LVar \ vn \Rightarrow \{P\}$

*FVar*:  $\llbracket G, A \vdash \{\text{Normal } P\} . \text{Init } C . \{Q\}; G, A \vdash \{Q\} \ e \succ \{\lambda Val:a.. fvar \ C \ stat \ fn \ a \ ..; R\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ \{\text{acc } C, C, \text{stat}\} e..fn \Rightarrow \{R\}$

*AVar*:  $\llbracket G, A \vdash \{\text{Normal } P\} \ e1 \succ \{Q\}; \forall a. G, A \vdash \{Q \leftarrow \text{Val } a\} \ e2 \succ \{\lambda Val:i.. avar \ G \ i \ a \ ..; R\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ e1.[e2] \Rightarrow \{R\}$

— expressions

*NewC*:  $\llbracket G, A \vdash \{\text{Normal } P\} . \text{Init } C . \{\text{Alloc } G \ (C \text{Inst } C) \ Q\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ \text{NewC } C \succ \{Q\}$

*NewA*:  $\llbracket G, A \vdash \{\text{Normal } P\} . \text{init-comp-ty } T . \{Q\}; G, A \vdash \{Q\} \ e \succ \{\lambda Val:i.. abupd \ (\text{check-neg } i) .; \text{Alloc } G \ (\text{Arr } T \ (\text{the-Intg } i)) \ R\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ \text{New } T[e] \succ \{R\}$

*Cast*:  $\llbracket G, A \vdash \{\text{Normal } P\} \ e \succ \{\lambda Val:v.. \lambda s.. abupd \ (\text{raise-if } (\neg G, s \vdash v \text{ fits } T) \ \text{ClassCast}) .; Q \leftarrow \text{Val } v\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ \text{Cast } T \ e \succ \{Q\}$

*Inst*:  $\llbracket G, A \vdash \{\text{Normal } P\} \ e \succ \{\lambda Val:v.. \lambda s.. Q \leftarrow \text{Val } (\text{Bool } (v \neq \text{Null} \wedge G, s \vdash v \text{ fits } \text{RefT } T))\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ e \ \text{InstOf } T \succ \{Q\}$

*Lit*:  $G, A \vdash \{\text{Normal } (P \leftarrow \text{Val } v)\} \ Lit \ v \succ \{P\}$

*UnOp*:  $\llbracket G, A \vdash \{\text{Normal } P\} \ e \succ \{\lambda Val:v.. Q \leftarrow \text{Val } (\text{eval-unop } unop \ v)\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ \text{UnOp } unop \ e \succ \{Q\}$

*BinOp*:

$\llbracket G, A \vdash \{\text{Normal } P\} \ e1 \succ \{Q\}; \forall v1. G, A \vdash \{Q \leftarrow \text{Val } v1\} \ (\text{if need-second-arg binop } v1 \text{ then } (In1l \ e2) \text{ else } (In1r \ \text{Skip})) \succ \{\lambda Val:v2.. R \leftarrow \text{Val } (\text{eval-binop } binop \ v1 \ v2)\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ \text{BinOp } binop \ e1 \ e2 \succ \{R\}$

*Super*:  $G, A \vdash \{\text{Normal } (\lambda s.. P \leftarrow \text{Val } (\text{val-this } s))\} \ \text{Super} \succ \{P\}$

*Acc*:  $\llbracket G, A \vdash \{\text{Normal } P\} \ va \Rightarrow \{\lambda Var:(v,f).. Q \leftarrow \text{Val } v\} \rrbracket \implies G, A \vdash \{\text{Normal } P\} \ \text{Acc } va \succ \{Q\}$

*Ass*:  $\llbracket G, A \vdash \{\text{Normal } P\} \ va \Rightarrow \{Q\}; \forall vf. G, A \vdash \{Q \leftarrow \text{Var } vf\} \ e \succ \{\lambda Val:v.. \text{assign } (\text{snd } vf) \ v .; R\} \rrbracket \implies$

$$G, A \vdash \{ \text{Normal } P \} \text{ va} := e \multimap \{ R \}$$

$$\begin{aligned} \text{Cond: } & \llbracket G, A \vdash \{ \text{Normal } P \} \text{ e0} \multimap \{ P' \}; \\ & \forall b. G, A \vdash \{ P' \leftarrow b \} \text{ (if } b \text{ then } e1 \text{ else } e2) \multimap \{ Q \} \rrbracket \implies \\ & G, A \vdash \{ \text{Normal } P \} \text{ e0} \text{ ? } e1 : e2 \multimap \{ Q \} \end{aligned}$$

*Call:*

$$\begin{aligned} & \llbracket G, A \vdash \{ \text{Normal } P \} \text{ e} \multimap \{ Q \}; \forall a. G, A \vdash \{ Q \leftarrow \text{Val } a \} \text{ args} \dot{=} \{ R \text{ } a \}; \\ & \forall a \text{ vs } \text{invC declC } l. G, A \vdash \{ (R \text{ } a \leftarrow \text{Vals } \text{vs } \wedge. \\ & (\lambda s. \text{declC} = \text{invocation-declclass } G \text{ mode (store } s) \text{ a statT } (\text{name} = \text{mn}, \text{parTs} = \text{pTs})) \wedge \\ & \text{invC} = \text{invocation-class mode (store } s) \text{ a statT } \wedge \\ & l = \text{locals (store } s)) \}; \\ & \text{init-lvars } G \text{ declC } (\text{name} = \text{mn}, \text{parTs} = \text{pTs}) \text{ mode } a \text{ vs} \wedge. \\ & (\lambda s. \text{normal } s \longrightarrow G \vdash \text{mode} \rightarrow \text{invC} \preceq \text{statT}) \rrbracket \\ \text{Methd declC } (\text{name} = \text{mn}, \text{parTs} = \text{pTs}) \multimap \{ \text{set-lvars } l \text{ .; } S \} \rrbracket \implies \\ & G, A \vdash \{ \text{Normal } P \} \{ \text{accC}, \text{statT}, \text{mode} \} e \cdot \text{mn}(\{ \text{pTs} \} \text{args}) \multimap \{ S \} \end{aligned}$$

$$\begin{aligned} \text{Methd: } & \llbracket G, A \cup \{ \{ P \} \text{ Methd} \multimap \{ Q \} \mid ms \} \vdash \{ \{ P \} \text{ body } G \multimap \{ Q \} \mid ms \} \rrbracket \implies \\ & G, A \vdash \{ \{ P \} \text{ Methd} \multimap \{ Q \} \mid ms \} \end{aligned}$$

$$\begin{aligned} \text{Body: } & \llbracket G, A \vdash \{ \text{Normal } P \} \text{ .Init } D. \{ Q \}; \\ & G, A \vdash \{ Q \} \text{ .c. } \{ \lambda s.. \text{abupd (absorb Ret) .; } R \leftarrow (\text{In1 (the (locals } s \text{ Result)))) \} \rrbracket \\ & \implies \\ & G, A \vdash \{ \text{Normal } P \} \text{ Body } D \text{ c} \multimap \{ R \} \end{aligned}$$

— expression lists

$$\text{Nil: } G, A \vdash \{ \text{Normal } (P \leftarrow \text{Vals } []) \} [] \dot{=} \{ P \}$$

$$\begin{aligned} \text{Cons: } & \llbracket G, A \vdash \{ \text{Normal } P \} \text{ e} \multimap \{ Q \}; \\ & \forall v. G, A \vdash \{ Q \leftarrow \text{Val } v \} \text{ es} \dot{=} \{ \lambda \text{Vals:vs}.. R \leftarrow \text{Vals } (v \# \text{vs}) \} \rrbracket \implies \\ & G, A \vdash \{ \text{Normal } P \} \text{ e} \# \text{es} \dot{=} \{ R \} \end{aligned}$$

— statements

$$\text{Skip: } G, A \vdash \{ \text{Normal } (P \leftarrow \Diamond) \} \text{ .Skip. } \{ P \}$$

$$\begin{aligned} \text{Expr: } & \llbracket G, A \vdash \{ \text{Normal } P \} \text{ e} \multimap \{ Q \leftarrow \Diamond \} \rrbracket \implies \\ & G, A \vdash \{ \text{Normal } P \} \text{ .Expr } e. \{ Q \} \end{aligned}$$

$$\begin{aligned} \text{Lab: } & \llbracket G, A \vdash \{ \text{Normal } P \} \text{ .c. } \{ \text{abupd (absorb } l) \text{ .; } Q \} \rrbracket \implies \\ & G, A \vdash \{ \text{Normal } P \} \text{ .l. c. } \{ Q \} \end{aligned}$$

$$\begin{aligned} \text{Comp: } & \llbracket G, A \vdash \{ \text{Normal } P \} \text{ .c1. } \{ Q \}; \\ & G, A \vdash \{ Q \} \text{ .c2. } \{ R \} \rrbracket \implies \\ & G, A \vdash \{ \text{Normal } P \} \text{ .c1;;c2. } \{ R \} \end{aligned}$$

$$\begin{aligned} \text{If: } & \llbracket G, A \vdash \{ \text{Normal } P \} \text{ e} \multimap \{ P' \}; \\ & \forall b. G, A \vdash \{ P' \leftarrow b \} \text{ .(if } b \text{ then } c1 \text{ else } c2). \{ Q \} \rrbracket \implies \\ & G, A \vdash \{ \text{Normal } P \} \text{ .If(e) } c1 \text{ Else } c2. \{ Q \} \end{aligned}$$

$$\begin{aligned} \text{Loop: } & \llbracket G, A \vdash \{ P \} \text{ e} \multimap \{ P' \}; \\ & G, A \vdash \{ \text{Normal } (P' \leftarrow \text{True}) \} \text{ .c. } \{ \text{abupd (absorb (Cont } l)) \text{ .; } P \} \rrbracket \implies \\ & G, A \vdash \{ P \} \text{ .l. While(e) c. } \{ (P' \leftarrow \text{False}) \downarrow = \Diamond \} \end{aligned}$$

$$\text{Jmp: } G, A \vdash \{ \text{Normal } (\text{abupd } (\lambda a. (\text{Some (Jump } j))) \text{ .; } P \leftarrow \Diamond) \} \text{ .Jmp } j. \{ P \}$$

$$\begin{aligned} \text{Throw: } & \llbracket G, A \vdash \{ \text{Normal } P \} \text{ e} \multimap \{ \lambda \text{Val:a}.. \text{abupd (throw } a) \text{ .; } Q \leftarrow \Diamond \} \rrbracket \implies \\ & G, A \vdash \{ \text{Normal } P \} \text{ .Throw } e. \{ Q \} \end{aligned}$$

*Try*:  $\llbracket G, A \vdash \{ \text{Normal } P \} .c1. \{ \text{SXAlloc } G \ Q \};$   
 $G, A \vdash \{ Q \wedge (\lambda s. G, s \vdash \text{catch } C) ;. \text{new-xcpt-var } vn \} .c2. \{ R \};$   
 $(Q \wedge (\lambda s. \neg G, s \vdash \text{catch } C)) \Rightarrow R \rrbracket \Longrightarrow$   
 $G, A \vdash \{ \text{Normal } P \} .\text{Try } c1 \ \text{Catch}(C \ vn) \ c2. \{ R \}$

*Fin*:  $\llbracket G, A \vdash \{ \text{Normal } P \} .c1. \{ Q \};$   
 $\forall x. G, A \vdash \{ Q \wedge (\lambda s. x = \text{fst } s) ;. \text{abupd } (\lambda x. \text{None}) \}$   
 $.c2. \{ \text{abupd } (\text{abrupt-if } (x \neq \text{None}) \ x) ;. R \} \rrbracket \Longrightarrow$   
 $G, A \vdash \{ \text{Normal } P \} .c1 \ \text{Finally } c2. \{ R \}$

*Done*:  $G, A \vdash \{ \text{Normal } (P \leftarrow \Diamond \wedge. \text{initd } C) \} .\text{Init } C. \{ P \}$

*Init*:  $\llbracket \text{the } (\text{class } G \ C) = c;$   
 $G, A \vdash \{ \text{Normal } ((P \wedge. \text{Not } \circ \text{initd } C) ;. \text{supd } (\text{init-class-obj } G \ C)) \}$   
 $.(\text{if } C = \text{Object then Skip else Init } (\text{super } c)). \{ Q \};$   
 $\forall l. G, A \vdash \{ Q \wedge (\lambda s. l = \text{locals } (\text{store } s)) ;. \text{set-lvars empty} \}$   
 $.\text{init } c. \{ \text{set-lvars } l ;. R \} \rrbracket \Longrightarrow$   
 $G, A \vdash \{ \text{Normal } (P \wedge. \text{Not } \circ \text{initd } C) \} .\text{Init } C. \{ R \}$

— Some dummy rules for the intermediate terms *Callee*, *InsInitE*, *InsInitV*, *FinA* only used by the smallstep semantics.

*InsInitV*:  $G, A \vdash \{ \text{Normal } P \} \text{InsInitV } c \ v \multimap \{ Q \}$   
*InsInitE*:  $G, A \vdash \{ \text{Normal } P \} \text{InsInitE } c \ e \multimap \{ Q \}$   
*Callee*:  $G, A \vdash \{ \text{Normal } P \} \text{Callee } l \ e \multimap \{ Q \}$   
*FinA*:  $G, A \vdash \{ \text{Normal } P \} .\text{FinA } a \ c. \{ Q \}$

### constdefs

*adapt-pre* :: 'a *assn*  $\Rightarrow$  'a *assn*  $\Rightarrow$  'a *assn*  $\Rightarrow$  'a *assn*  
*adapt-pre*  $P \ Q \ Q' \equiv \lambda Y \ s \ Z. \forall Y' \ s'. \exists Z'. P \ Y \ s \ Z' \wedge (Q \ Y' \ s' \ Z' \longrightarrow Q' \ Y' \ s' \ Z)$

### rules derived by induction

**lemma** *cut-valid*:  $\llbracket G, A' \rrbracket \models ts; G, A \rrbracket \models A \rrbracket \Longrightarrow G, A \rrbracket \models ts$   
 $\langle \text{proof} \rangle$

**lemma** *ax-thin* [*rule-format* (*no-asm*)]:  
 $G, (A' :: 'a \text{ triple set}) \rrbracket \vdash (ts :: 'a \text{ triple set}) \Longrightarrow \forall A. A' \subseteq A \longrightarrow G, A \rrbracket \vdash ts$   
 $\langle \text{proof} \rangle$

**lemma** *ax-thin-insert*:  $G, (A' :: 'a \text{ triple set}) \rrbracket \vdash (t :: 'a \text{ triple}) \Longrightarrow G, \text{insert } x \ A \rrbracket \vdash t$   
 $\langle \text{proof} \rangle$

**lemma** *subset-mtriples-iff*:  
 $ts \subseteq \{ \{ P \} \ mb \multimap \{ Q \} \mid ms \} = (\exists ms'. ms' \subseteq ms \wedge ts = \{ \{ P \} \ mb \multimap \{ Q \} \mid ms' \})$   
 $\langle \text{proof} \rangle$

**lemma** *weaken*:  
 $G, (A' :: 'a \text{ triple set}) \rrbracket \vdash (ts' :: 'a \text{ triple set}) \Longrightarrow !ts. ts \subseteq ts' \longrightarrow G, A \rrbracket \vdash ts$   
 $\langle \text{proof} \rangle$

### rules derived from conseq

In the following rules we often have to give some type annotations like:  $G, A \vdash \{P\} t \succ \{Q\}$ . Given only the term above without annotations, Isabelle would infer a more general type were we could have different types of auxiliary variables in the assumption set ( $A$ ) and in the triple itself ( $P$  and  $Q$ ). But *ax-derivs.Methd* enforces the same type in the inductive definition of the derivation. So we have to restrict the types to be able to apply the rules.

**lemma** *conseq12*:  $\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P' :: 'a \text{ assn}\} t \succ \{Q'\};$   
 $\forall Y s Z. P Y s Z \longrightarrow (\forall Y' s'. (\forall Y Z'. P' Y s Z' \longrightarrow Q' Y' s' Z') \longrightarrow$   
 $Q Y' s' Z) \rrbracket$   
 $\implies G, A \vdash \{P :: 'a \text{ assn}\} t \succ \{Q\}$   
 $\langle \text{proof} \rangle$

**lemma** *conseq12'*:  $\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P' :: 'a \text{ assn}\} t \succ \{Q'\}; \forall s Y' s'.$   
 $(\forall Y Z. P' Y s Z \longrightarrow Q' Y' s' Z) \longrightarrow$   
 $(\forall Y Z. P Y s Z \longrightarrow Q Y' s' Z) \rrbracket$   
 $\implies G, A \vdash \{P :: 'a \text{ assn}\} t \succ \{Q\}$   
 $\langle \text{proof} \rangle$

**lemma** *conseq12-from-conseq12'*:  $\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P' :: 'a \text{ assn}\} t \succ \{Q'\};$   
 $\forall Y s Z. P Y s Z \longrightarrow (\forall Y' s'. (\forall Y Z'. P' Y s Z' \longrightarrow Q' Y' s' Z') \longrightarrow$   
 $Q Y' s' Z) \rrbracket$   
 $\implies G, A \vdash \{P :: 'a \text{ assn}\} t \succ \{Q\}$   
 $\langle \text{proof} \rangle$

**lemma** *conseq1*:  $\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P' :: 'a \text{ assn}\} t \succ \{Q\}; P \Rightarrow P' \rrbracket$   
 $\implies G, A \vdash \{P :: 'a \text{ assn}\} t \succ \{Q\}$   
 $\langle \text{proof} \rangle$

**lemma** *conseq2*:  $\llbracket G, (A :: 'a \text{ triple set}) \vdash \{P :: 'a \text{ assn}\} t \succ \{Q'\}; Q' \Rightarrow Q \rrbracket$   
 $\implies G, A \vdash \{P :: 'a \text{ assn}\} t \succ \{Q\}$   
 $\langle \text{proof} \rangle$

**lemma** *ax-escape*:  
 $\llbracket \forall Y s Z. P Y s Z$   
 $\longrightarrow G, (A :: 'a \text{ triple set}) \vdash \{\lambda Y' s' (Z' :: 'a). (Y', s') = (Y, s)\}$   
 $t \succ$   
 $\{\lambda Y s Z'. Q Y s Z\}$   
 $\rrbracket \implies G, A \vdash \{P :: 'a \text{ assn}\} t \succ \{Q :: 'a \text{ assn}\}$   
 $\langle \text{proof} \rangle$

**lemma** *ax-constant*:  $\llbracket C \implies G, (A :: 'a \text{ triple set}) \vdash \{P :: 'a \text{ assn}\} t \succ \{Q\} \rrbracket$   
 $\implies G, A \vdash \{\lambda Y s Z. C \wedge P Y s Z\} t \succ \{Q\}$   
 $\langle \text{proof} \rangle$

**lemma** *ax-impossible [intro]*:  
 $G, (A :: 'a \text{ triple set}) \vdash \{\lambda Y s Z. \text{False}\} t \succ \{Q :: 'a \text{ assn}\}$   
 $\langle \text{proof} \rangle$

**lemma** *ax-nochange-lemma*:  $\llbracket P \ Y \ s; \text{All } (op = w) \rrbracket \implies P \ w \ s$   
 $\langle \text{proof} \rangle$

**lemma** *ax-nochange*:  
 $G, (A::(\text{res} \times \text{state}) \text{ triple set}) \vdash \{\lambda Y \ s \ Z. (Y, s) = Z\} \ t \succ \{\lambda Y \ s \ Z. (Y, s) = Z\}$   
 $\implies G, A \vdash \{P::(\text{res} \times \text{state}) \text{ assn}\} \ t \succ \{P\}$   
 $\langle \text{proof} \rangle$

**lemma** *ax-trivial*:  $G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} \ t \succ \{\lambda Y \ s \ Z. \text{True}\}$   
 $\langle \text{proof} \rangle$

**lemma** *ax-disj*:  
 $\llbracket G, (A::'a \text{ triple set}) \vdash \{P1::'a \text{ assn}\} \ t \succ \{Q1\}; G, A \vdash \{P2::'a \text{ assn}\} \ t \succ \{Q2\} \rrbracket$   
 $\implies G, A \vdash \{\lambda Y \ s \ Z. P1 \ Y \ s \ Z \vee P2 \ Y \ s \ Z\} \ t \succ \{\lambda Y \ s \ Z. Q1 \ Y \ s \ Z \vee Q2 \ Y \ s \ Z\}$   
 $\langle \text{proof} \rangle$

**lemma** *ax-supd-shuffle*:  
 $(\exists Q. G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} .c1. \{Q\} \wedge G, A \vdash \{Q ;. f\} .c2. \{R\}) =$   
 $(\exists Q'. G, A \vdash \{P\} .c1. \{f ;. Q'\} \wedge G, A \vdash \{Q'\} .c2. \{R\})$   
 $\langle \text{proof} \rangle$

**lemma** *ax-cases*:  
 $\llbracket G, (A::'a \text{ triple set}) \vdash \{P \wedge. C\} \ t \succ \{Q::'a \text{ assn}\};$   
 $G, A \vdash \{P \wedge. \text{Not} \circ C\} \ t \succ \{Q\} \rrbracket \implies G, A \vdash \{P\} \ t \succ \{Q\}$   
 $\langle \text{proof} \rangle$

**lemma** *ax-adapt*:  $G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} \ t \succ \{Q\}$   
 $\implies G, A \vdash \{\text{adapt-pre } P \ Q \ Q'\} \ t \succ \{Q'\}$   
 $\langle \text{proof} \rangle$

**lemma** *adapt-pre-adapts*:  $G, (A::'a \text{ triple set}) \models \{P::'a \text{ assn}\} \ t \succ \{Q\}$   
 $\longrightarrow G, A \models \{\text{adapt-pre } P \ Q \ Q'\} \ t \succ \{Q'\}$   
 $\langle \text{proof} \rangle$

**lemma** *adapt-pre-weakest*:  
 $\forall G \ (A::'a \text{ triple set}) \ t. G, A \models \{P\} \ t \succ \{Q\} \longrightarrow G, A \models \{P'\} \ t \succ \{Q'\} \implies$   
 $P' \Rightarrow \text{adapt-pre } P \ Q \ (Q'::'a \text{ assn})$   
 $\langle \text{proof} \rangle$

**lemma** *peek-and-forget1-Normal*:  
 $G, (A::'a \text{ triple set}) \vdash \{\text{Normal } P\} \ t \succ \{Q::'a \text{ assn}\}$   
 $\implies G, A \vdash \{\text{Normal } (P \wedge. p)\} \ t \succ \{Q\}$   
 $\langle \text{proof} \rangle$

**lemma** *peek-and-forget1*:

$$G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} \ t \succ \{Q\}$$

$$\implies G, A \vdash \{P \wedge. p\} \ t \succ \{Q\}$$

*<proof>*

**lemmas** *ax-NormalD* = *peek-and-forget1* [of - - - - normal]

**lemma** *peek-and-forget2*:

$$G, (A::'a \text{ triple set}) \vdash \{P::'a \text{ assn}\} \ t \succ \{Q \wedge. p\}$$

$$\implies G, A \vdash \{P\} \ t \succ \{Q\}$$

*<proof>*

**lemma** *ax-subst-Val-allI*:

$$\forall v. G, (A::'a \text{ triple set}) \vdash \{(P' \quad v) \leftarrow \text{Val } v\} \ t \succ \{(Q \ v)::'a \text{ assn}\}$$

$$\implies \forall v. G, A \vdash \{(\lambda w. P' \ (\text{the-In1 } w)) \leftarrow \text{Val } v\} \ t \succ \{Q \ v\}$$

*<proof>*

**lemma** *ax-subst-Var-allI*:

$$\forall v. G, (A::'a \text{ triple set}) \vdash \{(P' \quad v) \leftarrow \text{Var } v\} \ t \succ \{(Q \ v)::'a \text{ assn}\}$$

$$\implies \forall v. G, A \vdash \{(\lambda w. P' \ (\text{the-In2 } w)) \leftarrow \text{Var } v\} \ t \succ \{Q \ v\}$$

*<proof>*

**lemma** *ax-subst-Vals-allI*:

$$(\forall v. G, (A::'a \text{ triple set}) \vdash \{(P' \quad v) \leftarrow \text{Vals } v\} \ t \succ \{(Q \ v)::'a \text{ assn}\})$$

$$\implies \forall v. G, A \vdash \{(\lambda w. P' \ (\text{the-In3 } w)) \leftarrow \text{Vals } v\} \ t \succ \{Q \ v\}$$

*<proof>*

## alternative axioms

**lemma** *ax-Lit2*:

$$G, (A::'a \text{ triple set}) \vdash \{\text{Normal } P::'a \text{ assn}\} \ \text{Lit } v \multimap \{\text{Normal } (P \downarrow = \text{Val } v)\}$$

*<proof>*

**lemma** *ax-Lit2-test-complete*:

$$G, (A::'a \text{ triple set}) \vdash \{\text{Normal } (P \leftarrow \text{Val } v)::'a \text{ assn}\} \ \text{Lit } v \multimap \{P\}$$

*<proof>*

**lemma** *ax-LVar2*:  $G, (A::'a \text{ triple set}) \vdash \{\text{Normal } P::'a \text{ assn}\} \ \text{LVar } vn \multimap \{\text{Normal } (\lambda s. P \downarrow = \text{Var } (\text{lvar } vn \ s)))\}$

*<proof>*

**lemma** *ax-Super2*:  $G, (A::'a \text{ triple set}) \vdash$

$$\{\text{Normal } P::'a \text{ assn}\} \ \text{Super} \multimap \{\text{Normal } (\lambda s. P \downarrow = \text{Val } (\text{val-this } s)))\}$$

*<proof>*

**lemma** *ax-Nil2*:

$$G, (A::'a \text{ triple set}) \vdash \{\text{Normal } P::'a \text{ assn}\} \ \square \multimap \{\text{Normal } (P \downarrow = \text{Vals } \square)\}$$

*<proof>*

**misc derived structural rules**

**lemma** *ax-finite-mtriples-lemma*:  $\llbracket F \subseteq ms; \text{finite } ms; \forall (C, sig) \in ms. G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } (P \ C \ sig)::'a \text{ assn} \} \text{ mb } C \ sig \multimap \{ Q \ C \ sig \} \rrbracket \implies$   
 $G, A \Vdash \{ \{ P \} \text{ mb } \multimap \{ Q \} \mid F \}$

$\langle \text{proof} \rangle$

**lemmas** *ax-finite-mtriples* = *ax-finite-mtriples-lemma* [*OF subset-refl*]

**lemma** *ax-derivs-insertD*:

$G, (A::'a \text{ triple set}) \Vdash \text{insert } (t::'a \text{ triple}) \ ts \implies G, A \vdash t \wedge G, A \Vdash ts$

$\langle \text{proof} \rangle$

**lemma** *ax-methods-spec*:

$\llbracket G, (A::'a \text{ triple set}) \Vdash \text{split } f \ ' \ ms; (C, sig) \in ms \rrbracket \implies G, A \vdash ((f \ C \ sig)::'a \text{ triple})$

$\langle \text{proof} \rangle$

**lemma** *ax-finite-pointwise-lemma* [*rule-format*]:  $\llbracket F \subseteq ms; \text{finite } ms \rrbracket \implies$

$((\forall (C, sig) \in F. G, (A::'a \text{ triple set}) \vdash (f \ C \ sig)::'a \text{ triple})) \longrightarrow (\forall (C, sig) \in ms. G, A \vdash (g \ C \ sig)::'a \text{ triple})) \longrightarrow$   
 $G, A \Vdash \text{split } f \ ' \ F \longrightarrow G, A \Vdash \text{split } g \ ' \ F$

$\langle \text{proof} \rangle$

**lemmas** *ax-finite-pointwise* = *ax-finite-pointwise-lemma* [*OF subset-refl*]

**lemma** *ax-no-hazard*:

$G, (A::'a \text{ triple set}) \vdash \{ P \wedge. \text{type-ok } G \ t \} \ t \multimap \{ Q::'a \text{ assn} \} \implies G, A \vdash \{ P \} \ t \multimap \{ Q \}$

$\langle \text{proof} \rangle$

**lemma** *ax-free-wt*:

$(\exists T \ L \ C. (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash t::T) \longrightarrow G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } P \} \ t \multimap \{ Q::'a \text{ assn} \} \implies$   
 $G, A \vdash \{ \text{Normal } P \} \ t \multimap \{ Q \}$

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

**declare** *ax-Abrupts* [*intro!*]

**lemmas** *ax-Normal-cases* = *ax-cases* [*of - - normal*]

**lemma** *ax-Skip* [*intro!*]:  $G, (A::'a \text{ triple set}) \vdash \{ P \leftarrow \Diamond \} . \text{Skip}. \{ P::'a \text{ assn} \}$

$\langle \text{proof} \rangle$

**lemmas** *ax-SkipI* = *ax-Skip* [*THEN conseq1, standard*]

**derived rules for methd call**

**lemma** *ax-Call-known-DynT*:

$\llbracket G \vdash \text{IntVir} \rightarrow C \preceq \text{statT};$   
 $\forall a \text{ vs } l. G, A \vdash \{ (R \ a \leftarrow \text{Vals } vs \wedge. (\lambda s. l = \text{locals } (store \ s))) ;$   
 $\text{init-lvars } G \ C \ (\text{name} = mn, \text{parTs} = pTs) \ \text{IntVir } a \text{ vs} \}$   
 $\text{Methd } C \ (\text{name} = mn, \text{parTs} = pTs) \multimap \{ \text{set-lvars } l \ .; S \};$   
 $\forall a. G, A \vdash \{ Q \leftarrow \text{Val } a \} \ \text{args} \multimap$   
 $\{ R \ a \wedge. (\lambda s. C = \text{obj-class } (the \ (heap \ (store \ s) \ (the-Addr \ a)))) \wedge$   
 $C = \text{invocation-declclass}$   
 $G \ \text{IntVir } (store \ s) \ a \ \text{statT} \ (\text{name} = mn, \text{parTs} = pTs) \}$   $\rrbracket$

$$\begin{aligned}
& G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } P \} \ e \multimap \{ Q::'a \text{ assn} \} \\
& \implies G, A \vdash \{ \text{Normal } P \} \ \{ \text{acc}C, \text{stat}T, \text{IntVir} \} e \cdot mn(\{pTs\}args) \multimap \{ S \} \\
& \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *ax-Call-Static*:

$$\begin{aligned}
& \llbracket \forall a \text{ vs } l. \ G, A \vdash \{ R \ a \leftarrow \text{Vals } vs \ \wedge. \ (\lambda s. \ l = \text{locals } (\text{store } s)) \} ;. \\
& \quad \text{init-lvars } G \ C \ (\llbracket \text{name}=mn, \text{parTs}=pTs \rrbracket) \ \text{Static any-Addr } vs \} \\
& \quad \text{Methd } C \ (\llbracket \text{name}=mn, \text{parTs}=pTs \rrbracket) \multimap \{ \text{set-lvars } l \ ; \ S \}; \\
& \quad G, A \vdash \{ \text{Normal } P \} \ e \multimap \{ Q \}; \\
& \quad \forall a. \ G, (A::'a \text{ triple set}) \vdash \{ Q \leftarrow \text{Val } a \} \ \text{args} \dot{\multimap} \{ (R::\text{val} \Rightarrow 'a \text{ assn}) \ a \\
& \quad \wedge. \ (\lambda s. \ C = \text{invocation-declclass} \\
& \quad \quad G \ \text{Static } (\text{store } s) \ a \ \text{stat}T \ (\llbracket \text{name}=mn, \text{parTs}=pTs \rrbracket)) \} \\
& \rrbracket \implies G, A \vdash \{ \text{Normal } P \} \ \{ \text{acc}C, \text{stat}T, \text{Static} \} e \cdot mn(\{pTs\}args) \multimap \{ S \} \\
& \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *ax-Methd1*:

$$\begin{aligned}
& \llbracket G, A \cup \{ \{ P \} \ \text{Methd} \multimap \{ Q \} \mid ms \} \vdash \{ \{ P \} \ \text{body } G \multimap \{ Q \} \mid ms \}; \ (C, sig) \in ms \rrbracket \implies \\
& \quad G, A \vdash \{ \text{Normal } (P \ C \ sig) \} \ \text{Methd } C \ sig \multimap \{ Q \ C \ sig \} \\
& \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *ax-MethdN*:

$$\begin{aligned}
& G, \text{insert}(\{ \text{Normal } P \} \ \text{Methd } C \ sig \multimap \{ Q \}) \ A \vdash \\
& \quad \{ \text{Normal } P \} \ \text{body } G \ C \ sig \multimap \{ Q \} \implies \\
& \quad G, A \vdash \{ \text{Normal } P \} \ \text{Methd } C \ sig \multimap \{ Q \} \\
& \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *ax-StatRef*:

$$\begin{aligned}
& G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } (P \leftarrow \text{Val } \text{Null}) \} \ \text{StatRef } rt \multimap \{ P::'a \text{ assn} \} \\
& \langle \text{proof} \rangle
\end{aligned}$$

## rules derived from Init and Done

**lemma** *ax-InitS*:  $\llbracket \text{the } (\text{class } G \ C) = c; \ C \neq \text{Object};$

$$\begin{aligned}
& \forall l. \ G, A \vdash \{ Q \ \wedge. \ (\lambda s. \ l = \text{locals } (\text{store } s)) \} ;. \ \text{set-lvars empty} \} \\
& \quad .\text{init } c. \ \{ \text{set-lvars } l \ ; \ R \}; \\
& \quad G, A \vdash \{ \text{Normal } ((P \ \wedge. \ \text{Not} \circ \text{initd } C) \ ;. \ \text{supd } (\text{init-class-obj } G \ C)) \} \\
& \quad .\text{Init } (\text{super } c). \ \{ Q \} \rrbracket \implies \\
& \quad G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } (P \ \wedge. \ \text{Not} \circ \text{initd } C) \} .\text{Init } C. \ \{ R::'a \text{ assn} \} \\
& \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *ax-Init-Skip-lemma*:

$$\begin{aligned}
& \forall l. \ G, (A::'a \text{ triple set}) \vdash \{ P \leftarrow \Diamond \ \wedge. \ (\lambda s. \ l = \text{locals } (\text{store } s)) \} ;. \ \text{set-lvars } l' \} \\
& \quad .\text{Skip}. \ \{ (\text{set-lvars } l \ ; \ P) :: 'a \text{ assn} \} \\
& \langle \text{proof} \rangle
\end{aligned}$$

**lemma** *ax-triv-InitS*:  $\llbracket \text{the } (\text{class } G \ C) = c; \text{init } c = \text{Skip}; \ C \neq \text{Object};$

$$\begin{aligned}
& P \leftarrow \Diamond \Rightarrow (\text{supd } (\text{init-class-obj } G \ C) \ ; \ P); \\
& \quad G, A \vdash \{ \text{Normal } (P \ \wedge. \ \text{initd } C) \} .\text{Init } (\text{super } c). \ \{ (P \ \wedge. \ \text{initd } C) \leftarrow \Diamond \} \rrbracket \implies \\
& \quad G, (A::'a \text{ triple set}) \vdash \{ \text{Normal } P \leftarrow \Diamond \} .\text{Init } C. \ \{ (P \ \wedge. \ \text{initd } C) :: 'a \text{ assn} \} \\
& \langle \text{proof} \rangle
\end{aligned}$$



**lemma** *ax-Init-Object*:  $wf\text{-}prog\ G \implies G, (A::'a\ triple\ set) \vdash$   
 $\{Normal\ ((supd\ (init\text{-}class\text{-}obj\ G\ Object) \ .; P \leftarrow \Diamond) \wedge. Not \circ initd\ Object)\}$   
 $.Init\ Object. \{(P \wedge. initd\ Object)::'a\ assn\}$   
 $\langle proof \rangle$

**lemma** *ax-triv-Init-Object*:  $\llbracket wf\text{-}prog\ G;$   
 $(P::'a\ assn) \Rightarrow (supd\ (init\text{-}class\text{-}obj\ G\ Object) \ .; P) \rrbracket \implies$   
 $G, (A::'a\ triple\ set) \vdash \{Normal\ P \leftarrow \Diamond\} .Init\ Object. \{P \wedge. initd\ Object\}$   
 $\langle proof \rangle$

## introduction rules for Alloc and SXAlloc

**lemma** *ax-SXAlloc-Normal*:  
 $G, (A::'a\ triple\ set) \vdash \{P::'a\ assn\} .c. \{Normal\ Q\}$   
 $\implies G, A \vdash \{P\} .c. \{SXAlloc\ G\ Q\}$   
 $\langle proof \rangle$

**lemma** *ax-Alloc*:  
 $G, (A::'a\ triple\ set) \vdash \{P::'a\ assn\} \ t \succ$   
 $\{Normal\ (\lambda Y\ (x, s)\ Z. (\forall a. new\text{-}Addr\ (heap\ s) = Some\ a \longrightarrow$   
 $Q\ (Val\ (Addr\ a))\ (Norm\ (init\text{-}obj\ G\ (CInst\ C)\ (Heap\ a)\ s))\ Z)) \wedge.$   
 $heap\text{-}free\ (Suc\ (Suc\ 0))\}$   
 $\implies G, A \vdash \{P\} \ t \succ \{Alloc\ G\ (CInst\ C)\ Q\}$   
 $\langle proof \rangle$

**lemma** *ax-Alloc-Arr*:  
 $G, (A::'a\ triple\ set) \vdash \{P::'a\ assn\} \ t \succ$   
 $\{\lambda Val:i:. Normal\ (\lambda Y\ (x, s)\ Z. \neg the\text{-}Intg\ i < 0 \wedge$   
 $(\forall a. new\text{-}Addr\ (heap\ s) = Some\ a \longrightarrow$   
 $Q\ (Val\ (Addr\ a))\ (Norm\ (init\text{-}obj\ G\ (Arr\ T\ (the\text{-}Intg\ i))\ (Heap\ a)\ s))\ Z)) \wedge.$   
 $heap\text{-}free\ (Suc\ (Suc\ 0))\}$   
 $\implies$   
 $G, A \vdash \{P\} \ t \succ \{\lambda Val:i:. abupd\ (check\text{-}neg\ i) \ .; Alloc\ G\ (Arr\ T\ (the\text{-}Intg\ i))\ Q\}$   
 $\langle proof \rangle$

**lemma** *ax-SXAlloc-catch-SXcpt*:  
 $\llbracket G, (A::'a\ triple\ set) \vdash \{P::'a\ assn\} \ t \succ$   
 $\{(\lambda Y\ (x, s)\ Z. x = Some\ (Xcpt\ (Std\ xn)) \wedge$   
 $(\forall a. new\text{-}Addr\ (heap\ s) = Some\ a \longrightarrow$   
 $Q\ Y\ (Some\ (Xcpt\ (Loc\ a)), init\text{-}obj\ G\ (CInst\ (SXcpt\ xn))\ (Heap\ a)\ s)\ Z))$   
 $\wedge. heap\text{-}free\ (Suc\ (Suc\ 0))\} \rrbracket$   
 $\implies$   
 $G, A \vdash \{P\} \ t \succ \{SXAlloc\ G\ (\lambda Y\ s\ Z. Q\ Y\ s\ Z \wedge G, s \vdash catch\ SXcpt\ xn)\}$   
 $\langle proof \rangle$

**end**



## Chapter 23

# AxSound

## 62 Soundness proof for Axiomatic semantics of Java expressions and statements

theory *AxSound* imports *AxSem* begin

validity

consts

*triple-valid2*:: *prog*  $\Rightarrow$  *nat*  $\Rightarrow$  *'a triple*  $\Rightarrow$  *bool*  
 ( *-*  $\models$  ::- [61,0, 58] 57)  
*ax-valids2*:: *prog*  $\Rightarrow$  *'a triples*  $\Rightarrow$  *'a triples*  $\Rightarrow$  *bool*  
 ( *-*  $\models$  ::- [61,58,58] 57)

**defs** *triple-valid2-def*:  $G \models n::t \equiv \text{case } t \text{ of } \{P\} \triangleright \{Q\} \Rightarrow$   
 $\forall Y \ s \ Z. \ P \ Y \ s \ Z \longrightarrow (\forall L. \ s::\preceq(G,L)$   
 $\longrightarrow (\forall T \ C \ A. \ (\text{normal } s \longrightarrow ((\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t::T \wedge$   
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s)) \gg t \gg A)) \longrightarrow$   
 $(\forall Y' \ s'. \ G \vdash s - t \triangleright - n \longrightarrow (Y', s') \longrightarrow Q \ Y' \ s' \ Z \wedge s'::\preceq(G,L))))$

This definition differs from the ordinary *triple-valid-def* manly in the conclusion: We also ensures conformance of the result state. So we don't have to apply the type soundness lemma all the time during induction. This definition is only introduced for the soundness proof of the axiomatic semantics, in the end we will conclude to the ordinary definition.

**defs** *ax-valids2-def*:  $G, A \models::ts \equiv \forall n. (\forall t \in A. G \models n::t) \longrightarrow (\forall t \in ts. G \models n::t)$

**lemma** *triple-valid2-def2*:  $G \models n::\{P\} \triangleright \{Q\} =$   
 $(\forall Y \ s \ Z. \ P \ Y \ s \ Z \longrightarrow (\forall Y' \ s'. \ G \vdash s - t \triangleright - n \longrightarrow (Y', s') \longrightarrow$   
 $(\forall L. \ s::\preceq(G,L) \longrightarrow (\forall T \ C \ A. \ (\text{normal } s \longrightarrow ((\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t::T \wedge$   
 $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s)) \gg t \gg A)) \longrightarrow$   
 $Q \ Y' \ s' \ Z \wedge s'::\preceq(G,L))))$   
 <proof>

**lemma** *triple-valid2-eq* [rule-format (no-asm)]:  
 $\text{wf-prog } G \implies \text{triple-valid2 } G = \text{triple-valid } G$   
 <proof>

**lemma** *ax-valids2-eq*:  $\text{wf-prog } G \implies G, A \models::ts = G, A \models ts$   
 <proof>

**lemma** *triple-valid2-Suc* [rule-format (no-asm)]:  $G \models \text{Suc } n::t \longrightarrow G \models n::t$   
 <proof>

**lemma** *Methd-triple-valid2-0*:  $G \models 0::\{\text{Normal } P\} \text{ Methd } C \text{ sig} \triangleright \{Q\}$   
 <proof>

**lemma** *Methd-triple-valid2-SucI*:  
 $\llbracket G \models n::\{\text{Normal } P\} \text{ body } G \ C \ \text{sig} \triangleright \{Q\} \rrbracket$   
 $\implies G \models \text{Suc } n::\{\text{Normal } P\} \text{ Methd } C \ \text{sig} \triangleright \{Q\}$   
 <proof>

**lemma** *triples-valid2-Suc*:

$Ball\ ts\ (triple\text{-}valid2\ G\ (Suc\ n)) \implies Ball\ ts\ (triple\text{-}valid2\ G\ n)$   
 $\langle proof \rangle$

**lemma**  $G \models n::insert\ t\ A = (G \models n::t \wedge G \models n::A)$   
 $\langle proof \rangle$

### soundness

**lemma** *Method-sound*:

**assumes** *recursive*:  $G, A \cup \{\{P\}\} \text{Method} \multimap \{Q\} \mid ms \models \{\{P\}\} \text{body}\ G \multimap \{Q\} \mid ms$   
**shows**  $G, A \models \{\{P\}\} \text{Method} \multimap \{Q\} \mid ms$   
 $\langle proof \rangle$

**lemma** *valids2-inductI*:  $\forall s\ t\ n\ Y'\ s'.\ G \vdash s \multimap t \multimap n \rightarrow (Y', s') \longrightarrow t = c \longrightarrow$   
 $Ball\ A\ (triple\text{-}valid2\ G\ n) \longrightarrow (\forall Y\ Z.\ P\ Y\ s\ Z \longrightarrow$   
 $(\forall L.\ s::\preceq(G, L) \longrightarrow$   
 $(\forall T\ C\ A.\ (normal\ s \longrightarrow (\llbracket prg=G, cls=C, lcl=L \rrbracket \vdash t::T) \wedge$   
 $\llbracket prg=G, cls=C, lcl=L \rrbracket \vdash dom\ (locals\ (store\ s)) \gg t \gg A) \longrightarrow$   
 $Q\ Y'\ s'\ Z \wedge s'::\preceq(G, L))) \implies$   
 $G, A \models \{\{P\}\} c \multimap \{Q\}$   
 $\langle proof \rangle$

**lemma** *da-good-approx-evalnE* [consumes 4]:

**assumes** *evaln*:  $G \vdash s0 \multimap t \multimap n \rightarrow (v, s1)$   
**and** *wt*:  $\llbracket prg=G, cls=C, lcl=L \rrbracket \vdash t::T$   
**and** *da*:  $\llbracket prg=G, cls=C, lcl=L \rrbracket \vdash dom\ (locals\ (store\ s0)) \gg t \gg A$   
**and** *wf*: *wf-prog*  $G$   
**and** *elim*:  $\llbracket normal\ s1 \implies nrm\ A \subseteq dom\ (locals\ (store\ s1));$   
 $\wedge\ l.\ \llbracket abrupt\ s1 = Some\ (Jump\ (Break\ l)); normal\ s0 \rrbracket$   
 $\implies brk\ A\ l \subseteq dom\ (locals\ (store\ s1));$   
 $\llbracket abrupt\ s1 = Some\ (Jump\ Ret); normal\ s0 \rrbracket$   
 $\implies Result \in dom\ (locals\ (store\ s1))$   
 $\rrbracket \implies P$   
**shows**  $P$   
 $\langle proof \rangle$

**lemma** *validI*:

**assumes**  $I: \bigwedge n\ s0\ L\ accC\ T\ C\ v\ s1\ Y\ Z.$   
 $\llbracket \forall t \in A.\ G \models n::t; s0::\preceq(G, L);$   
 $normal\ s0 \implies \llbracket prg=G, cls=accC, lcl=L \rrbracket \vdash t::T;$   
 $normal\ s0 \implies \llbracket prg=G, cls=accC, lcl=L \rrbracket \vdash dom\ (locals\ (store\ s0)) \gg t \gg C;$   
 $G \vdash s0 \multimap t \multimap n \rightarrow (v, s1); P\ Y\ s0\ Z \rrbracket \implies Q\ v\ s1\ Z \wedge s1::\preceq(G, L)$   
**shows**  $G, A \models \{\{P\}\} t \multimap \{Q\}$   
 $\langle proof \rangle$

$\langle ML \rangle$

**lemma** *valid-stmtI*:

**assumes**  $I: \bigwedge n\ s0\ L\ accC\ C\ s1\ Y\ Z.$   
 $\llbracket \forall t \in A.\ G \models n::t; s0::\preceq(G, L);$

$normal\ s0 \Longrightarrow (\llbracket prg=G, cls=accC, lcl=L \rrbracket) \vdash c::\checkmark;$   
 $normal\ s0 \Longrightarrow (\llbracket prg=G, cls=accC, lcl=L \rrbracket) \vdash_{dom} (locals\ (store\ s0)) \gg \langle c \rangle_s \gg C;$   
 $G \vdash s0 -c-n \rightarrow s1; P\ Y\ s0\ Z \Longrightarrow Q \diamond s1\ Z \wedge s1::\preceq(G,L)$   
**shows**  $G, A \models::\{ \{P\} \langle c \rangle_s \succ \{Q\} \}$   
 $\langle proof \rangle$

**lemma** *valid-stmt-NormalI*:

**assumes**  $I: \bigwedge n\ s0\ L\ accC\ C\ s1\ Y\ Z.$   
 $\llbracket \forall t \in A. G \models n::t; s0::\preceq(G,L); normal\ s0; (\llbracket prg=G, cls=accC, lcl=L \rrbracket) \vdash c::\checkmark;$   
 $(\llbracket prg=G, cls=accC, lcl=L \rrbracket) \vdash_{dom} (locals\ (store\ s0)) \gg \langle c \rangle_s \gg C;$   
 $G \vdash s0 -c-n \rightarrow s1; (Normal\ P)\ Y\ s0\ Z \Longrightarrow Q \diamond s1\ Z \wedge s1::\preceq(G,L)$   
**shows**  $G, A \models::\{ \{Normal\ P\} \langle c \rangle_s \succ \{Q\} \}$   
 $\langle proof \rangle$

**lemma** *valid-var-NormalI*:

**assumes**  $I: \bigwedge n\ s0\ L\ accC\ T\ C\ vf\ s1\ Y\ Z.$   
 $\llbracket \forall t \in A. G \models n::t; s0::\preceq(G,L); normal\ s0;$   
 $(\llbracket prg=G, cls=accC, lcl=L \rrbracket) \vdash t::=T;$   
 $(\llbracket prg=G, cls=accC, lcl=L \rrbracket) \vdash_{dom} (locals\ (store\ s0)) \gg \langle t \rangle_v \gg C;$   
 $G \vdash s0 -t-\succ vf-n \rightarrow s1; (Normal\ P)\ Y\ s0\ Z \Longrightarrow Q\ (In2\ vf)\ s1\ Z \wedge s1::\preceq(G,L)$   
**shows**  $G, A \models::\{ \{Normal\ P\} \langle t \rangle_v \succ \{Q\} \}$   
 $\langle proof \rangle$

**lemma** *valid-expr-NormalI*:

**assumes**  $I: \bigwedge n\ s0\ L\ accC\ T\ C\ v\ s1\ Y\ Z.$   
 $\llbracket \forall t \in A. G \models n::t; s0::\preceq(G,L); normal\ s0;$   
 $(\llbracket prg=G, cls=accC, lcl=L \rrbracket) \vdash t::=T;$   
 $(\llbracket prg=G, cls=accC, lcl=L \rrbracket) \vdash_{dom} (locals\ (store\ s0)) \gg \langle t \rangle_e \gg C;$   
 $G \vdash s0 -t-\succ v-n \rightarrow s1; (Normal\ P)\ Y\ s0\ Z \Longrightarrow Q\ (In1\ v)\ s1\ Z \wedge s1::\preceq(G,L)$   
**shows**  $G, A \models::\{ \{Normal\ P\} \langle t \rangle_e \succ \{Q\} \}$   
 $\langle proof \rangle$

**lemma** *valid-expr-list-NormalI*:

**assumes**  $I: \bigwedge n\ s0\ L\ accC\ T\ C\ vs\ s1\ Y\ Z.$   
 $\llbracket \forall t \in A. G \models n::t; s0::\preceq(G,L); normal\ s0;$   
 $(\llbracket prg=G, cls=accC, lcl=L \rrbracket) \vdash t::=T;$   
 $(\llbracket prg=G, cls=accC, lcl=L \rrbracket) \vdash_{dom} (locals\ (store\ s0)) \gg \langle t \rangle_l \gg C;$   
 $G \vdash s0 -t-\succ vs-n \rightarrow s1; (Normal\ P)\ Y\ s0\ Z \Longrightarrow Q\ (In3\ vs)\ s1\ Z \wedge s1::\preceq(G,L)$   
**shows**  $G, A \models::\{ \{Normal\ P\} \langle t \rangle_l \succ \{Q\} \}$   
 $\langle proof \rangle$

**lemma** *validE [consumes 5]*:

**assumes**  $valid: G, A \models::\{ \{P\} t \succ \{Q\} \}$   
**and**  $P: P\ Y\ s0\ Z$   
**and**  $valid-A: \forall t \in A. G \models n::t$   
**and**  $conf: s0::\preceq(G,L)$   
**and**  $eval: G \vdash s0 -t-\succ -n \rightarrow (v, s1)$   
**and**  $wt: normal\ s0 \Longrightarrow (\llbracket prg=G, cls=accC, lcl=L \rrbracket) \vdash t::T$   
**and**  $da: normal\ s0 \Longrightarrow (\llbracket prg=G, cls=accC, lcl=L \rrbracket) \vdash_{dom} (locals\ (store\ s0)) \gg t \gg C$   
**and**  $elim: \llbracket Q\ v\ s1\ Z; s1::\preceq(G,L) \rrbracket \Longrightarrow concl$   
**shows**  $concl$

$\langle \text{proof} \rangle$

**lemma** *all-empty*:  $(!x. P) = P$

$\langle \text{proof} \rangle$

**corollary** *evaln-type-sound*:

**assumes** *evaln*:  $G \vdash s0 \text{ --}t\text{--}n \rightarrow (v, s1)$  **and**  
           *wt*:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash t :: T$  **and**  
           *da*:  $(\text{prg}=G, \text{cls}=\text{acc}C, \text{lcl}=L) \vdash \text{dom} (\text{locals} (\text{store } s0)) \gg t \gg A$  **and**  
           *conf-s0*:  $s0 :: \preceq (G, L)$  **and**  
           *wf*: *wf-prog*  $G$   
**shows**  $s1 :: \preceq (G, L) \wedge (\text{normal } s1 \longrightarrow G, L, \text{store } s1 \vdash t \text{--}v :: \preceq T) \wedge$   
            $(\text{error-free } s0 = \text{error-free } s1)$

$\langle \text{proof} \rangle$

**corollary** *dom-locals-evaln-mono-elim* [*consumes 1*]:

**assumes**  
   *evaln*:  $G \vdash s0 \text{ --}t\text{--}n \rightarrow (v, s1)$  **and**  
   *hyps*:  $\llbracket \text{dom} (\text{locals} (\text{store } s0)) \subseteq \text{dom} (\text{locals} (\text{store } s1)) \rrbracket$   
            $\wedge \text{vv } s \text{ val. } \llbracket v = \text{In2 } vv; \text{normal } s1 \rrbracket$   
            $\implies \text{dom} (\text{locals} (\text{store } s))$   
            $\subseteq \text{dom} (\text{locals} (\text{store } ((\text{snd } vv) \text{ val } s))) \rrbracket \implies P$

**shows**  $P$

$\langle \text{proof} \rangle$

**lemma** *evaln-no-abrupt*:

$\bigwedge s \ s'. \llbracket G \vdash s \text{ --}t\text{--}n \rightarrow (w, s') ; \text{normal } s' \rrbracket \implies \text{normal } s$   
 $\langle \text{proof} \rangle$

**declare** *inj-term-simps* [*simp*]

**lemma** *ax-sound2*:

**assumes**   *wf*: *wf-prog*  $G$   
           **and**   *deriv*:  $G, A \vdash ts$   
**shows**  $G, A \models ts$

$\langle \text{proof} \rangle$

**declare** *inj-term-simps* [*simp del*]

**theorem** *ax-sound*:

*wf-prog*  $G \implies G, (A :: 'a \text{ triple set}) \vdash (ts :: 'a \text{ triple set}) \implies G, A \models ts$   
 $\langle \text{proof} \rangle$

**lemma** *sound-valid2-lemma*:

$\llbracket \forall v \ n. \text{Ball } A (\text{triple-valid2 } G \ n) \longrightarrow P \ v \ n; \text{Ball } A (\text{triple-valid2 } G \ n) \rrbracket$   
 $\implies P \ v \ n$

$\langle \text{proof} \rangle$

**end**





## Chapter 24

### AxCompl

### 63 Completeness proof for Axiomatic semantics of Java expressions and statements

**theory** *AxCompl* **imports** *AxSem* **begin**

design issues:

- proof structured by Most General Formulas (-, Thomas Kleymann)

**set of not yet initialized classes**

**constdefs**

*nyinitcls* :: *prog*  $\Rightarrow$  *state*  $\Rightarrow$  *qname* *set*  
*nyinitcls* *G* *s*  $\equiv$  { *C*. *is-class* *G* *C*  $\wedge$   $\neg$  *initd* *C* *s* }

**lemma** *nyinitcls-subset-class*: *nyinitcls* *G* *s*  $\subseteq$  { *C*. *is-class* *G* *C* }  
 $\langle$ *proof* $\rangle$

**lemmas** *finite-nyinitcls* [*simp*] =  
*finite-is-class* [*THEN nyinitcls-subset-class* [*THEN finite-subset*], *standard*]

**lemma** *card-nyinitcls-bound*: *card* (*nyinitcls* *G* *s*)  $\leq$  *card* { *C*. *is-class* *G* *C* }  
 $\langle$ *proof* $\rangle$

**lemma** *nyinitcls-set-locals-cong* [*simp*]:  
*nyinitcls* *G* (*x*, *set-locals* *l* *s*) = *nyinitcls* *G* (*x*, *s*)  
 $\langle$ *proof* $\rangle$

**lemma** *nyinitcls-abrupt-cong* [*simp*]: *nyinitcls* *G* (*f* *x*, *y*) = *nyinitcls* *G* (*x*, *y*)  
 $\langle$ *proof* $\rangle$

**lemma** *nyinitcls-abupd-cong* [*simp*]:!!*s*. *nyinitcls* *G* (*abupd* *f* *s*) = *nyinitcls* *G* *s*  
 $\langle$ *proof* $\rangle$

**lemma** *card-nyinitcls-abrupt-congE* [*elim*!]:  
*card* (*nyinitcls* *G* (*x*, *s*))  $\leq$  *n*  $\Longrightarrow$  *card* (*nyinitcls* *G* (*y*, *s*))  $\leq$  *n*  
 $\langle$ *proof* $\rangle$

**lemma** *nyinitcls-new-xcpt-var* [*simp*]:  
*nyinitcls* *G* (*new-xcpt-var* *vn* *s*) = *nyinitcls* *G* *s*  
 $\langle$ *proof* $\rangle$

**lemma** *nyinitcls-init-lvars* [*simp*]:  
*nyinitcls* *G* ((*init-lvars* *G* *C* *sig* *mode* *a'* *pvs*) *s*) = *nyinitcls* *G* *s*  
 $\langle$ *proof* $\rangle$

**lemma** *nyinitcls-emptyD*:  $\llbracket$ *nyinitcls* *G* *s* = {} $\rrbracket$ ; *is-class* *G* *C*  $\Longrightarrow$  *initd* *C* *s*  
 $\langle$ *proof* $\rangle$

**lemma** *card-Suc-lemma*:

$\llbracket \text{card } (\text{insert } a \ A) \leq \text{Suc } n; a \notin A; \text{finite } A \rrbracket \implies \text{card } A \leq n$   
 $\langle \text{proof} \rangle$

**lemma** *nyinitcls-le-SucD*:

$\llbracket \text{card } (\text{nyinitcls } G \ (x,s)) \leq \text{Suc } n; \neg \text{initd } C \ (\text{globs } s); \text{class } G \ C = \text{Some } y \rrbracket \implies$   
 $\text{card } (\text{nyinitcls } G \ (x, \text{init-class-obj } G \ C \ s)) \leq n$   
 $\langle \text{proof} \rangle$

**lemma** *initd-gext'*:  $\llbracket s \leq |s'; \text{initd } C \ (\text{globs } s) \rrbracket \implies \text{initd } C \ (\text{globs } s')$

$\langle \text{proof} \rangle$

**lemma** *nyinitcls-gext*:  $\text{snd } s \leq | \text{snd } s' \implies \text{nyinitcls } G \ s' \subseteq \text{nyinitcls } G \ s$

$\langle \text{proof} \rangle$

**lemma** *card-nyinitcls-gext*:

$\llbracket \text{snd } s \leq | \text{snd } s'; \text{card } (\text{nyinitcls } G \ s) \leq n \rrbracket \implies \text{card } (\text{nyinitcls } G \ s') \leq n$   
 $\langle \text{proof} \rangle$

**init-le**

**constdefs**

$\text{init-le} :: \text{prog} \Rightarrow \text{nat} \Rightarrow \text{state} \Rightarrow \text{bool} \quad (\text{+init} \leq - \ [51,51] \ 50)$   
 $G \vdash \text{init} \leq n \equiv \lambda s. \text{card } (\text{nyinitcls } G \ s) \leq n$

**lemma** *init-le-def2* [simp]:  $(G \vdash \text{init} \leq n) \ s = (\text{card } (\text{nyinitcls } G \ s) \leq n)$

$\langle \text{proof} \rangle$

**lemma** *All-init-leD*:

$\forall n::\text{nat}. G, (A::'a \ \text{triple set}) \vdash \{P \ \wedge. \ G \vdash \text{init} \leq n\} \ t \succ \ \{Q::'a \ \text{assn}\}$   
 $\implies G, A \vdash \{P\} \ t \succ \ \{Q\}$   
 $\langle \text{proof} \rangle$

## Most General Triples and Formulas

**constdefs**

$\text{remember-init-state} :: \text{state assn} \quad (\doteq)$   
 $\doteq \equiv \lambda Y \ s \ Z. \ s = Z$

**lemma** *remember-init-state-def2* [simp]:  $\doteq \ Y = \text{op} =$

$\langle \text{proof} \rangle$

**consts**

$\text{MGF} :: [\text{state assn}, \text{term}, \text{prog}] \Rightarrow \text{state triple} \ (\{-\} \dashv \rightarrow \{-\rightarrow\} [3,65,3] 62)$   
 $\text{MGFn} :: [\text{nat} \quad , \text{term}, \text{prog}] \Rightarrow \text{state triple} \ (\{=-\} \dashv \rightarrow \{-\rightarrow\} [3,65,3] 62)$

**defs**

*MGF-def:*

$$\{P\} t \succ \{G \rightarrow\} \equiv \{P\} t \succ \{\lambda Y s' s. G \vdash s -t \rightarrow (Y, s')\}$$

*MGFn-def:*

$$\{=:n\} t \succ \{G \rightarrow\} \equiv \{\dot{=} \wedge. G \vdash \text{init} \leq n\} t \succ \{G \rightarrow\}$$

**lemma** *MGF-valid: wf-prog*  $G \implies G, \{\dot{=}\} \vdash \{=:n\} t \succ \{G \rightarrow\}$

*<proof>*

**lemma** *MGF-res-eq-lemma [simp]:*

$$(\forall Y' Y s. Y = Y' \wedge P s \longrightarrow Q s) = (\forall s. P s \longrightarrow Q s)$$

*<proof>*

**lemma** *MGFn-def2:*

$$\begin{aligned} G, A \vdash \{=:n\} t \succ \{G \rightarrow\} &= G, A \vdash \{\dot{=} \wedge. G \vdash \text{init} \leq n\} \\ &\quad t \succ \{\lambda Y s' s. G \vdash s -t \rightarrow (Y, s')\} \end{aligned}$$

*<proof>*

**lemma** *MGF-MGFn-iff:*

$$G, (A::\text{state triple set}) \vdash \{\dot{=}\} t \succ \{G \rightarrow\} = (\forall n. G, A \vdash \{=:n\} t \succ \{G \rightarrow\})$$

*<proof>*

**lemma** *MGFnD:*

$$\begin{aligned} G, (A::\text{state triple set}) \vdash \{=:n\} t \succ \{G \rightarrow\} &\implies \\ G, A \vdash \{(\lambda Y' s' s. s' = s \quad \wedge P s) \wedge. G \vdash \text{init} \leq n\} \\ t \succ \{(\lambda Y' s' s. G \vdash s -t \rightarrow (Y', s') \wedge P s) \wedge. G \vdash \text{init} \leq n\} \end{aligned}$$

*<proof>*

**lemmas**  $MGFnD' = MGFnD$  [of - - -  $\lambda x. \text{True}$ ]

To derive the most general formula, we can always assume a normal state in the precondition, since abrupt cases can be handled uniformly by the abrupt rule.

**lemma** *MGFNormalI:*  $G, A \vdash \{\text{Normal} \dot{=}\} t \succ \{G \rightarrow\} \implies$

$$G, (A::\text{state triple set}) \vdash \{\dot{=}\} t \succ \{G \rightarrow\}$$

*<proof>*

**lemma** *MGFNormalD:*

$$G, (A::\text{state triple set}) \vdash \{\dot{=}\} t \succ \{G \rightarrow\} \implies G, A \vdash \{\text{Normal} \dot{=}\} t \succ \{G \rightarrow\}$$

*<proof>*

Additionally to *MGFNormalI*, we also expand the definition of the most general formula here

**lemma** *MGFn-NormalI:*

$$\begin{aligned} G, (A::\text{state triple set}) \vdash \{\text{Normal}((\lambda Y' s' s. s' = s \wedge \text{normal } s) \wedge. G \vdash \text{init} \leq n)\} t \succ \\ \{\lambda Y s' s. G \vdash s -t \rightarrow (Y, s')\} \implies G, A \vdash \{=:n\} t \succ \{G \rightarrow\} \end{aligned}$$

*<proof>*

To derive the most general formula, we can restrict ourselves to welltyped terms, since all others can be uniformly handled by the hazard rule.

**lemma** *MGFn-free-wt:*

$$\begin{aligned} (\exists T L C. (\text{prg} = G, \text{cls} = C, \text{lcl} = L) \vdash t::T) \\ \longrightarrow G, (A::\text{state triple set}) \vdash \{=:n\} t \succ \{G \rightarrow\} \end{aligned}$$

$$\implies G, A \vdash \{=:n\} \ t \succ \{G \rightarrow\}$$

*<proof>*

To derive the most general formula, we can restrict ourselves to welltyped terms and assume that the state in the precondition conforms to the environment. All type violations can be uniformly handled by the hazard rule.

**lemma** *MGFn-free-wt-NormalConformI*:

$$\begin{aligned} & (\forall \ T \ L \ C \ . \ (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t :: T \\ & \longrightarrow G, (A :: \text{state triple set}) \\ & \quad \vdash \{ \text{Normal}((\lambda Y' \ s' \ s. \ s'=s \wedge \text{normal } s) \wedge. G \vdash \text{init} \leq n) \wedge. (\lambda s. s :: \preceq(G, L)) \} \\ & \quad t \succ \\ & \quad \{ \lambda Y \ s' \ s. G \vdash s - t \succ \rightarrow (Y, s') \} \\ & \implies G, A \vdash \{=:n\} t \succ \{G \rightarrow\} \end{aligned}$$

*<proof>*

To derive the most general formula, we can restrict ourselves to welltyped terms and assume that the state in the precondition conforms to the environment and that the term is definitely assigned with respect to this state. All type violations can be uniformly handled by the hazard rule.

**lemma** *MGFn-free-wt-da-NormalConformI*:

$$\begin{aligned} & (\forall \ T \ L \ C \ B. \ (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash t :: T \\ & \longrightarrow G, (A :: \text{state triple set}) \\ & \quad \vdash \{ \text{Normal}((\lambda Y' \ s' \ s. \ s'=s \wedge \text{normal } s) \wedge. G \vdash \text{init} \leq n) \wedge. (\lambda s. s :: \preceq(G, L)) \\ & \quad \wedge. (\lambda s. (\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash \text{dom } (\text{locals } (\text{store } s)) \gg t \gg B) \} \\ & \quad t \succ \\ & \quad \{ \lambda Y \ s' \ s. G \vdash s - t \succ \rightarrow (Y, s') \} \\ & \implies G, A \vdash \{=:n\} t \succ \{G \rightarrow\} \end{aligned}$$

*<proof>*

## main lemmas

**lemma** *MGFn-Init*:

**assumes** *mgf-hyp*:  $\forall m. \text{Suc } m \leq n \longrightarrow (\forall t. G, A \vdash \{=:m\} \ t \succ \{G \rightarrow\})$   
**shows**  $G, (A :: \text{state triple set}) \vdash \{=:n\} \ (\text{Init } C) \succ \{G \rightarrow\}$   
*<proof>*

**lemmas**  $\text{MGFn-InitD} = \text{MGFn-Init} \ [\text{THEN } \text{MGFnD}, \text{ THEN } \text{ax-NormalD}]$

**lemma** *MGFn-Call*:

**assumes** *mgf-methods*:  
 $\forall C \ \text{sig}. \ G, (A :: \text{state triple set}) \vdash \{=:n\} \ \langle (\text{Methd } C \ \text{sig}) \rangle_e \succ \{G \rightarrow\}$   
**and** *mgf-e*:  $G, A \vdash \{=:n\} \ \langle e \rangle_e \succ \{G \rightarrow\}$   
**and** *mgf-ps*:  $G, A \vdash \{=:n\} \ \langle ps \rangle_t \succ \{G \rightarrow\}$   
**and** *wf*: *wf-prog*  $G$   
**shows**  $G, A \vdash \{=:n\} \ \langle \{ \text{acc } C, \text{stat } T, \text{mode} \} e \cdot \text{mn}(\{ pTs \} \cap ps) \rangle_e \succ \{G \rightarrow\}$   
*<proof>*

**lemma** *eval-expression-no-jump'*:

**assumes** *eval*:  $G \vdash s0 - e \rightarrow v \rightarrow s1$   
**and** *no-jmp*:  $\text{abrupt } s0 \neq \text{Some } (\text{Jump } j)$   
**and** *wt*:  $(\text{prg}=G, \text{cls}=C, \text{lcl}=L) \vdash e :: -T$   
**and** *wf*: *wf-prog*  $G$   
**shows**  $\text{abrupt } s1 \neq \text{Some } (\text{Jump } j)$   
*<proof>*

To derive the most general formula for the loop statement, we need to come up with a proper loop invariant, which intuitively states that we are currently inside the evaluation of the loop. To define

such an invariant, we unroll the loop in iterated evaluations of the expression and evaluations of the loop body.

**constdefs**

$unroll :: prog \Rightarrow label \Rightarrow expr \Rightarrow stmt \Rightarrow (state \times state) set$

$unroll\ G\ l\ e\ c \equiv \{(s, t). \exists\ v\ s1\ s2. \\ G \vdash s - e \rightarrow v \rightarrow s1 \wedge the-Bool\ v \wedge normal\ s1 \wedge \\ G \vdash s1 - c \rightarrow s2 \wedge t = (abupd\ (absorb\ (Cont\ l))\ s2)\}$

**lemma unroll-while:**

**assumes**  $unroll: (s, t) \in (unroll\ G\ l\ e\ c)^*$   
**and**  $eval-e: G \vdash t - e \rightarrow v \rightarrow s'$   
**and**  $normal-termination: normal\ s' \longrightarrow \neg the-Bool\ v$   
**and**  $wt: (\langle prg = G, cls = C, lcl = L \rangle) \vdash e :: -T$   
**and**  $wf: wf-prog\ G$   
**shows**  $G \vdash s - l \cdot While(e)\ c \rightarrow s'$

$\langle proof \rangle$

$\langle ML \rangle$

**lemma MGFn-Loop:**

**assumes**  $mgf-e: G, (A :: state\ triple\ set) \vdash \{=:n\} \langle e \rangle_e \succ \{G \rightarrow\}$   
**and**  $mgf-c: G, A \vdash \{=:n\} \langle c \rangle_s \succ \{G \rightarrow\}$   
**and**  $wf: wf-prog\ G$   
**shows**  $G, A \vdash \{=:n\} \langle l \cdot While(e)\ c \rangle_s \succ \{G \rightarrow\}$

$\langle proof \rangle$

**lemma MGFn-FVar:**

**fixes**  $A :: state\ triple\ set$   
**assumes**  $mgf-init: G, A \vdash \{=:n\} \langle Init\ statDeclC \rangle_s \succ \{G \rightarrow\}$   
**and**  $mgf-e: G, A \vdash \{=:n\} \langle e \rangle_e \succ \{G \rightarrow\}$   
**and**  $wf: wf-prog\ G$   
**shows**  $G, A \vdash \{=:n\} \langle \{accC, statDeclC, stat\} e..fn \rangle_v \succ \{G \rightarrow\}$

$\langle proof \rangle$

**lemma MGFn-Fin:**

**assumes**  $wf: wf-prog\ G$   
**and**  $mgf-c1: G, A \vdash \{=:n\} \langle c1 \rangle_s \succ \{G \rightarrow\}$   
**and**  $mgf-c2: G, A \vdash \{=:n\} \langle c2 \rangle_s \succ \{G \rightarrow\}$   
**shows**  $G, (A :: state\ triple\ set) \vdash \{=:n\} \langle c1\ Finally\ c2 \rangle_s \succ \{G \rightarrow\}$

$\langle proof \rangle$

**lemma Body-no-break:**

**assumes**  $eval-init: G \vdash Norm\ s0 - Init\ D \rightarrow s1$   
**and**  $eval-c: G \vdash s1 - c \rightarrow s2$   
**and**  $jmpOk: jumpNestingOkS\ \{Ret\}\ c$   
**and**  $wt-c: (\langle prg = G, cls = C, lcl = L \rangle) \vdash c :: \surd$   
**and**  $clsD: class\ G\ D = Some\ d$   
**and**  $wf: wf-prog\ G$   
**shows**  $\forall\ l. abrupt\ s2 \neq Some\ (Jump\ (Break\ l)) \wedge$

$\text{abrupt } s2 \neq \text{Some } (\text{Jump } (\text{Cont } l))$   
 $\langle \text{proof} \rangle$

**lemma** *MGFn-Body*:

**assumes**  $\text{wf}: \text{wf-prog } G$   
**and**  $\text{mgf-init}: G, A \vdash \{=:n\} \langle \text{Init } D \rangle_s \succ \{G \rightarrow\}$   
**and**  $\text{mgf-c}: G, A \vdash \{=:n\} \langle c \rangle_s \succ \{G \rightarrow\}$   
**shows**  $G, (A::\text{state triple set}) \vdash \{=:n\} \langle \text{Body } D \ c \rangle_e \succ \{G \rightarrow\}$   
 $\langle \text{proof} \rangle$

**lemma** *MGFn-lemma*:

**assumes**  $\text{mgf-methds}$ :  
 $\bigwedge n. \forall C \text{ sig. } G, (A::\text{state triple set}) \vdash \{=:n\} \langle \text{Methd } C \text{ sig} \rangle_e \succ \{G \rightarrow\}$   
**and**  $\text{wf}: \text{wf-prog } G$   
**shows**  $\bigwedge t. G, A \vdash \{=:n\} t \succ \{G \rightarrow\}$   
 $\langle \text{proof} \rangle$

**lemma** *MGF-asm*:

$\llbracket \forall C \text{ sig. is-methd } G \ C \text{ sig} \longrightarrow G, A \vdash \{\dot{=}\} \text{In1l } (\text{Methd } C \text{ sig}) \succ \{G \rightarrow\}; \text{wf-prog } G \rrbracket$   
 $\implies G, (A::\text{state triple set}) \vdash \{\dot{=}\} t \succ \{G \rightarrow\}$   
 $\langle \text{proof} \rangle$

**nested version**

**lemma** *nesting-lemma'* [rule-format (no-asm)]:

**assumes**  $\text{ax-derivs-asm}$ :  $\bigwedge A \text{ ts. } ts \subseteq A \implies P \ A \ ts$   
**and**  $\text{MGF-nested-Methd}$ :  $\bigwedge A \text{ pn. } \forall b \in \text{bdy } pn. P \ (\text{insert } (\text{mgf-call } pn) \ A) \ \{\text{mgf } b\}$   
 $\implies P \ A \ \{\text{mgf-call } pn\}$   
**and**  $\text{MGF-asm}$ :  $\bigwedge A \ t. \forall pn \in U. P \ A \ \{\text{mgf-call } pn\} \implies P \ A \ \{\text{mgf } t\}$   
**and**  $\text{finU}$ : finite  $U$   
**and**  $\text{uA}$ :  $uA = \text{mgf-call}' U$   
**shows**  $\forall A. A \subseteq uA \longrightarrow n \leq \text{card } uA \longrightarrow \text{card } A = \text{card } uA - n$   
 $\longrightarrow (\forall t. P \ A \ \{\text{mgf } t\})$   
 $\langle \text{proof} \rangle$

**lemma** *nesting-lemma* [rule-format (no-asm)]:

**assumes**  $\text{ax-derivs-asm}$ :  $\bigwedge A \text{ ts. } ts \subseteq A \implies P \ A \ ts$   
**and**  $\text{MGF-nested-Methd}$ :  $\bigwedge A \text{ pn. } \forall b \in \text{bdy } pn. P \ (\text{insert } (\text{mgf } (f \text{ pn})) \ A) \ \{\text{mgf } b\}$   
 $\implies P \ A \ \{\text{mgf } (f \text{ pn})\}$   
**and**  $\text{MGF-asm}$ :  $\bigwedge A \ t. \forall pn \in U. P \ A \ \{\text{mgf } (f \text{ pn})\} \implies P \ A \ \{\text{mgf } t\}$   
**and**  $\text{finU}$ : finite  $U$   
**shows**  $P \ \{\} \ \{\text{mgf } t\}$   
 $\langle \text{proof} \rangle$

**lemma** *MGF-nested-Methd*:  $\llbracket$

$G, \text{insert } (\{\text{Normal } \dot{=}\} \langle \text{Methd } C \text{ sig} \rangle_e \succ \{G \rightarrow\}) \ A$   
 $\vdash \{\text{Normal } \dot{=}\} \langle \text{body } G \ C \text{ sig} \rangle_e \succ \{G \rightarrow\}$   
 $\rrbracket \implies G, A \vdash \{\text{Normal } \dot{=}\} \langle \text{Methd } C \text{ sig} \rangle_e \succ \{G \rightarrow\}$   
 $\langle \text{proof} \rangle$

**lemma** *MGF-deriv*:  $\text{wf-prog } G \implies G, (\{\}::\text{state triple set}) \vdash \{\dot{=}\} t \succ \{G \rightarrow\}$

$\langle proof \rangle$

### simultaneous version

**lemma** *MGF-simult-Methd-lemma: finite ms  $\implies$*   
 $G, A \cup (\lambda(C, sig). \{Normal \doteq\} \langle Methd \ C \ sig \rangle_e \succ \{G \rightarrow\}) \text{ ' } ms$   
 $\vdash (\lambda(C, sig). \{Normal \doteq\} \langle body \ G \ C \ sig \rangle_e \succ \{G \rightarrow\}) \text{ ' } ms \implies$   
 $G, A \vdash (\lambda(C, sig). \{Normal \doteq\} \langle Methd \ C \ sig \rangle_e \succ \{G \rightarrow\}) \text{ ' } ms$   
 $\langle proof \rangle$

**lemma** *MGF-simult-Methd: wf-prog G  $\implies$*   
 $G, (\{\} :: state \ triple \ set) \vdash (\lambda(C, sig). \{Normal \doteq\} \langle Methd \ C \ sig \rangle_e \succ \{G \rightarrow\})$   
 $\text{ ' } Collect \ (split \ (is-methd \ G))$   
 $\langle proof \rangle$

### corollaries

**lemma** *eval-to-evaln:  $\llbracket G \vdash s \rightarrow t \succ \rightarrow (Y', s'); type-ok \ G \ t \ s; wf-prog \ G \rrbracket$*   
 $\implies \exists n. G \vdash s \rightarrow t \succ \rightarrow n \rightarrow (Y', s')$   
 $\langle proof \rangle$

**lemma** *MGF-complete:*  
**assumes** *valid:*  $G, \{\} \models \{P\} \ t \succ \{Q\}$   
**and** *mgf:*  $G, (\{\} :: state \ triple \ set) \vdash \{\doteq\} \ t \succ \{G \rightarrow\}$   
**and** *wf:*  $wf-prog \ G$   
**shows**  $G, (\{\} :: state \ triple \ set) \vdash \{P :: state \ assn\} \ t \succ \{Q\}$   
 $\langle proof \rangle$

**theorem** *ax-complete:*  
**assumes** *wf:*  $wf-prog \ G$   
**and** *valid:*  $G, \{\} \models \{P :: state \ assn\} \ t \succ \{Q\}$   
**shows**  $G, (\{\} :: state \ triple \ set) \vdash \{P\} \ t \succ \{Q\}$   
 $\langle proof \rangle$

**end**



## Chapter 25

### AxExample

## 64 Example of a proof based on the Bali axiomatic semantics

**theory** *AxExample* **imports** *AxSem Example* **begin**

**constdefs**

*arr-inv* :: *st*  $\Rightarrow$  *bool*  
*arr-inv*  $\equiv \lambda s. \exists \text{obj } a \ T \text{ el. } \text{globs } s \ (\text{Stat } \text{Base}) = \text{Some } \text{obj} \wedge$   
 $\text{values } \text{obj} \ (\text{Inl } (\text{arr}, \text{Base})) = \text{Some } (\text{Addr } a) \wedge$   
 $\text{heap } s \ a = \text{Some } (\text{tag}=\text{Arr } T \ 2, \text{values}=\text{el})$

**lemma** *arr-inv-new-obj*:

$\bigwedge a. \llbracket \text{arr-inv } s; \text{new-Addr } (\text{heap } s) = \text{Some } a \rrbracket \Longrightarrow \text{arr-inv } (\text{gupd}(\text{Inl } a \mapsto x) \ s)$   
 $\langle \text{proof} \rangle$

**lemma** *arr-inv-set-locals* [*simp*]: *arr-inv* (*set-locals* *l s*) = *arr-inv* *s*  
 $\langle \text{proof} \rangle$

**lemma** *arr-inv-gupd-Stat* [*simp*]:

*Base*  $\neq$  *C*  $\Longrightarrow \text{arr-inv } (\text{gupd}(\text{Stat } C \mapsto \text{obj}) \ s) = \text{arr-inv } s$   
 $\langle \text{proof} \rangle$

**lemma** *ax-inv-lupd* [*simp*]: *arr-inv* (*lupd*(*x*  $\mapsto$  *y*) *s*) = *arr-inv* *s*  
 $\langle \text{proof} \rangle$

**declare** *split-if-asm* [*split del*]

**declare** *lvar-def* [*simp*]

$\langle \text{ML} \rangle$

**theorem** *ax-test*: *tprg*,( $\{\} :: 'a \text{ triple set}$ ) $\vdash$

$\{\text{Normal } (\lambda Y \ s \ Z :: 'a. \text{heap-free four } s \wedge \neg \text{initd } \text{Base } s \wedge \neg \text{initd } \text{Ext } s)\}$   
 $\text{.test } [\text{Class } \text{Base}].$   
 $\{\lambda Y \ s \ Z. \text{abrupt } s = \text{Some } (\text{Xcpt } (\text{Std } \text{IndOutBound}))\}$   
 $\langle \text{proof} \rangle$

**lemma** *Loop-Xcpt-benchmark*:

$Q = (\lambda Y \ (x, s) \ Z. x \neq \text{None} \longrightarrow \text{the-Bool } (\text{the } (\text{locals } s \ i))) \Longrightarrow$   
 $G, (\{\} :: 'a \text{ triple set}) \vdash \{\text{Normal } (\lambda Y \ s \ Z :: 'a. \text{True})\}$   
 $\text{.lab1} \bullet \text{While}(\text{Lit } (\text{Bool } \text{True})) (\text{If}(\text{Acc } (\text{LVar } i)) (\text{Throw } (\text{Acc } (\text{LVar } \text{xcpt}))) \text{Else}$   
 $(\text{Expr } (\text{Ass } (\text{LVar } i) (\text{Acc } (\text{LVar } j)))). \{Q\}$   
 $\langle \text{proof} \rangle$

**end**